TIAGO FURTADO

UM MÉTODO PARCIALMENTE AUTOMATIZADO PARA CUSTOMIZAÇÃO DE PROCESSOS DE SOFTWARE POR MEIO DE PLANEJAMENTO DISTRIBUÍDO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS – BRASIL
2010

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
F992m
2010

Furtado, Tiago, 1986-
Um método parcialmente automatizado para customização de processos de sofware por meio de planejamento distribuído / Tiago Furtado – Viçosa, MG, 2010.
ix, 87f. : il ; 29cm.

Inclui apêndices.
Orientador: Alcione de Paiva Oliveira.
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 84-87

1. Engenharia de software. 2. Agentes inteligentes (Software) - Planejamento. 3. Software - Desenvolvimento. I. Universidade Federal de Viçosa. II. Título.

CDD 22. ed. 005.1

TIAGO FURTADO

UM MÉTODO PARCIALMENTE AUTOMATIZADO PARA CUSTOMIZAÇÃO DE PROCESSOS DE SOFTWARE POR MEIO DE PLANEJAMENTO DISTRIBUÍDO

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 10 de dezembro de 2010

José Luís Braga
(Co-Orientador)

Vladimir Oliveira Di Iorio
(Co-Orientador)

Regina Maria Maciel Braga Villela

André Gustavo dos Santos

Alcione de Paiva Oliveira
(Orientador)

AGRADECIMENTOS

A meus pais, Dirceu e Cida, por sempre terem dado as condições necessárias e o reconfortante apoio em todas as minhas realizações. Aos meus irmãos, Fred e Juliana, pelo apoio incondicional, apesar da distância e, muitas vezes, da ausência.

Ao meu orientador, Prof. Alcione, por ser tão compreensivo com minhas incertezas, ajudar a manter os rumos do trabalho e acreditar nesta realização.

Ao amigo e afilhado J. Reinaldo e ao grande amigo Hussin, que me mostraram que não há distância para verdadeiras amizades, pois se fazem presentes mesmo estando longe.

Aos amigos mais recentes, mas nem por isso menos importantes, Saulo, Vanessa, Bruno e Arthur. A vida em Viçosa não teria o mesmo sentido sem vocês.

À A.S. Sistemas e ao Prof. Luiz Fontes, por acreditarem no meu potencial e pela oportunidade singular de desenvolvimento profissional.

Aos tantos que fizeram parte de minha vida durante o período de graduação e pós-graduação em Viçosa; colegas de classe, empresários juniores, amigos de amigos e demais “agregados”.

A todos, meus sinceros agradecimentos. Vocês foram, cada um à sua maneira e a seu tempo, importantes em minha vida e fundamentais para a conclusão desta obra.

BIOGRAFIA

TIAGO FURTADO, filho de Dirceu Barreiros Furtado e Maria Aparecida Furtado, brasileiro, nascido em 28 de abril de 1986, na cidade de Barbacena, estado de Minas Gerais.

Ingressou no curso de Bacharelado em Ciência da Computação da Universidade Federal de Viçosa no ano de 2004, concluindo sua graduação em 2007. Em março de 2008 iniciou o mestrado no Programa de Pós-Graduação em Ciência da Computação do Departamento de Informática da mesma universidade, tendo defendido sua dissertação em dezembro de 2010.

Durante a graduação, entre os anos de 2004 e 2007, trabalhou na No Bugs – Empresa Júnior de Informática da Universidade Federal de Viçosa, tendo atuado nos cargos de Diretor Presidente, Diretor de Marketing e Presidente do Conselho Administrativo.

A partir de julho de 2007, passou a integrar o quadro de funcionários da A.S. Sistemas, na função de Auxiliar de Desenvolvimento. Em fevereiro de 2008, assumiu o cargo de Gerente de Desenvolvimento.

SUMÁRIO

LISTA DE FIGURAS......vi

RESUMO......viii

ABSTRACT......ix

1 INTRODUÇÃO......1

1.1 O problema e sua importância......2
1.2 Hipótese......3
1.3 Objetivos......3
1.4 Metodologia......4
1.5 Organização do texto......5

2 SISTEMAS MULTIAGENTES E PLANEJAMENTO DISTRIBUÍDO......6

2.1 Sistemas multiagentes......6
2.2 Agentes planejadores......7
2.3 Arquitetura BDI......9
2.3.1 Estrutura para implementação......10
2.4 Planejamento distribuído......12

3 CUSTOMIZAÇÃO DE PROCESSOS DE SOFTWARE......15

3.1 Processos de software......15
3.1.1 Métodos dirigidos por panejamento......17
3.1.2 Métodos ágeis......18
3.2 Customização de processos de software......18
3.2.1 Automatização da tarefa de customização......21

4 MÉTODO PROPOSTO......23

4.1 Descrição geral......24
4.2 Definições subjacentes......26
4.3 Arquitetura dos agentes......28
4.3.1 Arquitetura comum......29
4.3.2 Gerente de projetos......30
4.3.3 Especialista......31
4.3.4 Agente de serviço......32
4.4 Processamento e gerenciamento de informações......32
4.5 Comunicação......35
4.5.1 Protocolo de inscrição......36
4.5.2 Protocolo de iniciação......36
4.5.3 Protocolo de solicitação de artefato......37
4.5.4 Protocolo de entrega de resultado......38
4.5.5 Protocolo de finalização......39
4.5.6 Protocolo de solicitação de informação......40
4.6 Comportamento......40

4.6.1 Preparação....41
4.6.2 Ciclo de definição....42
4.6.3 Ciclo de simulação....44
4.6.4 Finalização....45

5 IMPLEMENTAÇÃO E ESTUDO DE CASO....47

5.1 Implementação....47
5.1.1 Mapeamento de conceitos....47
5.1.2 Protocolos de comunicação....50
5.1.3 Detalhes adicionais da implementação....55
5.1.4 Limitações....57
5.2 Estudo de caso....58
5.2.1 Modelo de processo....58
5.2.2 Cenário base....59
5.2.3 Cenário limitado....60
5.2.4 Execução e resultados....60

6 CONCLUSÕES E TRABALHOS FUTUROS....63

6.1 Principais contribuições....64
6.2 Trabalhos futuros....65

APÊNDICE A....66

APÊNDICE B....67

APÊNDICE C....80

REFERÊNCIAS BIBLIOGRÁFICAS....84

LISTA DE FIGURAS

Figura 3-1. Prioridades para adaptar o modelo de processo....19
Figura 4-1. Descrição geral das interações entre os agentes....25
Figura 4-2. Propriedades mantidas pelos conjuntos de artefatos da base de conhecimentos de um especialista....33
Figura 4-3. Propriedades mantidas pelos conjuntos de artefatos da base de conhecimentos do Gerente de Projetos....34
Figura 4-4. Protocolo de inscrição....36
Figura 4-5. Protocolo de iniciação....37
Figura 4-6. Protocolo de solicitação de artefato....38
Figura 4-7. Protocolo de entrega de resultado....39
Figura 4-8. Protocolo de finalização....39
Figura 4-9. Protocolo de solicitação de informação....40
Figura 4-10. Visão geral do comportamento do processamento individual dos agentes. 41
Figura 4-11. Ciclo de Definição dos agentes especialistas....42
Figura 4-12. Ciclo de Definição do gerente de projetos....44
Figura 4-13. Ciclo de Simulação dos agentes especialistas....45
Figura 5-1. Definição e implementação do conceito de requisito funcional....48
Figura 5-2. Mapeamento do conceito de fator de risco....48
Figura 5-3. Mapeamento do conceito de papel....49
Figura 5-4. Mapeamento do conceito de modelo de artefato....49
Figura 5-5. Mapeamento do conceito de artefato....50
Figura 5-6. Mapeamento do conceito de atividade do processo....50
Figura 5-7. Mensagens trocadas no protocolo de inscrição....51
Figura 5-8. Mensagens trocadas no protocolo de iniciação....51
Figura 5-9. Mensagens trocadas no protocolo de solicitação de artefato....52
Figura 5-10. Mensagem enviada no protocolo de entrega de resultado....53
Figura 5-11. Representação de tarefas e dependências no processo gerado....53
Figura 5-12. Sinal de finalização....54
Figura 5-13. Mensagens trocadas no protocolo de solicitação de informação....54
Figura 5-14. Exemplo de interação entre os agentes e troca de mensagens....55
Figura 5-15. Predicados Prolog para manipulação de listas....56
Figura 5-16. Exemplo de uso do predicado concat....56

Figura 5-17. Possibilidades de execução do processo adotado no estudo de caso..........59
Figura 5-18. Processo gerado para Cenário Base do estudo de caso...............................61
Figura 5-19. Processo gerado para Cenário Limitado do estudo de caso........................62

RESUMO

FURTADO, Tiago. M. Sc., Universidade Federal de Viçosa, dezembro de 2010. **Um método parcialmente automatizado para customização de processos de software por meio de planejamento distribuído**. Orientador: Alcione de Paiva Oliveira. Coorientadores: José Luis Braga e Vladimir Oliveira Di Iorio.

É notável a importância socioeconômica adquirida pelos produtos de software ao longo dos últimos anos, estando presentes em diversas atividades do cotidiano das pessoas. Para atingir seus objetivos, o software necessita ser desenvolvido de forma a garantir sua qualidade em todas as etapas de seu ciclo de vida. Na Ciência da Computação, a área ligada aos processos de desenvolvimento, gerenciamento e controle da produção é a Engenharia de Software. Mais do que simples ferramenta de controle, o processo é a unidade básica de valor dentro de uma organização. Processos adequados fornecem ferramentas eficientes de acompanhamento e controle de projetos, fornecendo o suporte necessário para gerentes de equipe e projeto. Um importante fator responsável pelo atraso ou cancelamento de projetos de software é o suporte à gerência executiva. Uma possível solução para o problema da falta de um processo adequado é a personalização de um processo para um projeto ou uma equipe específica. Entretanto, tal adaptação demanda recursos humanos, financeiros e, acima de tudo, tempo e, na maioria das vezes, este tempo extra não está disponível. A utilização de técnicas de Inteligência Artificial, mais especificamente as implementadas por meio de sistemas multiagentes, pode proporcionar redução da quantidade de recursos gastos na adaptação de processos de desenvolvimento. Este trabalho estabelece um método baseado em planejamento distribuído para definir processos de software adequados a contextos determinados, com base em características da equipe responsável pelo desenvolvimento de um projeto de software e nos recursos disponíveis. Um sistema implementando o método proposto também é apresentado, juntamente a um estudo de caso conduzido com o intuito de avaliar o comportamento da implementação.

ABSTRACT

FURTADO, Tiago. M. Sc., Universidade Federal de Viçosa, December, 2010. **A partially automated method for software process tailoring employing distributed planning**. Advisor: Alcione de Paiva Oliveira. Co-advisors: José Luis Braga and Vladimir Oliveira Di Iorio.

It is remarkable how software products have acquired increasingly social and economic importance over the past years, as they are present in various activities of everyday life. To achieve its objectives a software system must be developed to ensure quality in all stages of its life cycle. In Computer Science, the area related to development processes, production management and control is Software Engineering. More than just a control tool, a process is the basic unit of value within an organization. Appropriate procedures provide efficient tools for monitoring and controlling projects, providing the necessary support to managers and the project team. An important factor responsible for the delay or cancellation of software projects is support for executive management. A possible solution to the problem of lack of an adequate process is the customization of a process framework for a project or a specific team. However, this adaptation demand human and financial resources and, above all, time. And in most cases, this extra time is not available. The use of artificial intelligence techniques, specifically those implemented by multi-agent systems can provide a reduction in the amount of resources spent on adapting development processes. This work establishes a method based on distributed planning to define appropriate software processes to certain contexts, based on characteristics of the team responsible for developing a software project and the resources available. A system to implement the proposed method is also presented, along with a case study conducted aiming the evaluation of the implementation behavior.

1 INTRODUÇÃO

"Begin at the beginning and go on till you come to the end; then stop"
Lewis Carroll

É notável a importância socioeconômica adquirida pelos produtos de software ao longo dos últimos anos. Capazes de controlar desde dispositivos simples, como cafeteiras ou aspiradores de pó, aos mais sofisticados, como máquinas para tratamento radioterápico e naves espaciais, os sistemas computacionais estão cada vez mais presentes, direta ou indiretamente, na vida do homem moderno e controlam ou participam de muitas relações sociais. Para atingir seus objetivos, o software necessita ser desenvolvido de forma a garantir sua qualidade em todas as etapas de seu ciclo de vida. Na Ciência da Computação, a área ligada aos processos de desenvolvimento, gerenciamento e controle da produção é a Engenharia de Software. O enfoque de engenharia é necessário para garantir a disciplina necessária à obtenção de software de qualidade, segundo Pressman (2005).

As organizações, em geral, possuem diversos processos de negócio para direcionar e sistematizar atividades e tarefas de rotina. Mais do que simples ferramenta de controle, o processo é a unidade básica de valor dentro de uma organização, segundo Verner (2004). Ainda segundo o autor, os processos podem ser tanto uma causa de ineficiência quanto uma fonte potencial de vantagem competitiva, o que faz o processo de desenvolvimento desempenhar papel fundamental no sucesso ou fracasso dos projetos desenvolvidos em organizações que trabalham com a criação de software.

O relatório de The Standish Group (1994) apresenta como um dos principais fatores essenciais ao sucesso de projetos de software o suporte à gerência executiva. Além disso, a falta desse suporte foi um dos principais responsáveis pelo atraso ou cancelamento de projetos na pesquisa realizada. Uma das possíveis conclusões às quais se pode chegar sobre este dado, apesar de não haver nenhuma relação explícita descrita no texto, é que isto decorre da falta de um processo de desenvolvimento de software adequado às necessidades das equipes responsáveis por tais projetos. Processos adequados fornecem ferramentas eficientes de acompanhamento e controle de projetos,

fornecendo o suporte necessário para gerentes de equipe e projeto. Desta forma, estes profissionais podem avaliar constantemente o trabalho em andamento, tomar ações corretivas e reter informações após sua conclusão, a fim de evitar problemas semelhantes no futuro.

Uma possível solução para o problema da falta de um processo adequado é a personalização de um processo para um projeto ou uma equipe específica. Entretanto, tal adaptação demanda recursos humanos, financeiros e, acima de tudo, tempo, pois é necessário analisar as necessidades específicas envolvidas para, então, ajustar processos bem estabelecidos à realidade detectada. Na maioria das vezes, este tempo extra não está disponível, já que mais da metade (52,7%) dos projetos de software abordados na pesquisa de The Standish Group (1994) foram entregues fora do prazo.

Dentre as diversas áreas da Ciência da Computação, encontram-se na Inteligência Artificial várias técnicas para lidar com grandes quantidades de dados e produzir informação útil a partir dos mesmos. Frente ao cenário descrito, a utilização de técnicas de Inteligência Artificial, mais especificamente as implementadas por meio de sistemas multiagentes, pode proporcionar redução da quantidade de recursos gastos na adaptação de processos de desenvolvimento, fornecendo um instrumento adequado às realidades das equipes responsáveis, sem a necessidade de investir tempo extra nesta atividade. Tal suposição fundamenta-se na grande quantidade de variáveis envolvidas no processo e na eficiência com que sistemas multiagentes podem conseguir tratá-las.

1.1 O problema e sua importância

Os processos de desenvolvimento de software comumente adotados na indústria são generalistas, definidos sem considerar as particularidades de cada equipe ou de cada projeto, o que leva a uma aparente padronização da atividade de desenvolvimento. Entretanto, os problemas a serem resolvidos, e os sistemas desenvolvidos, possuem características próprias e, possivelmente, únicas, que devem ser tratadas por equipes diferentes quantitativa e qualitativamente. Nota-se uma disparidade entre os processos, o produto e a equipe de desenvolvimento, na qual o produto final desejado – o software – muitas vezes não satisfaz às necessidades dos clientes, pois não há um processo e uma equipe de desenvolvimento adequados à sua produção. Há, portanto, expectativas frustradas, clientes insatisfeitos e softwares sem a qualidade esperada por seus usuários.

Uma maneira de solucionar o problema é criar processos específicos para cada equipe ou mesmo para determinados projetos, de forma a atender plenamente às necessidades de desenvolvimento envolvidas. Todavia, esta é uma atividade complexa,

que envolve estudar as características dos problemas a serem resolvidos, dos indivíduos e do contexto no qual se encontram todos estes fatores, modelar os fluxos de trabalho e de informações, verificar sua aplicabilidade e fazer correções quando necessário para, então, implantar o novo processo. Toda esta tarefa demanda tempo e consome recursos (financeiros, materiais ou humanos) e se puder ser automatizada, pelo menos em parte, tende a reduzir os custos de todo o processo.

Se o software for considerado o produto final do processo produtivo, como ocorre nos cenários industriais, é possível perceber como os atuais modelos de desenvolvimento de software não são tão amadurecidos quanto os da indústria em geral. Neste cenário, as atividades de planejamento, execução e controle da produção ocorrem de maneira sistemática há vários anos. Assim, as principais falhas já foram detectadas, solucionadas e testadas. O ramo de produção de software ainda é novo e, portanto, ainda está amadurecendo processos de desenvolvimento eficazes e eficientes. Ainda estão sendo elaboradas soluções para as principais falhas encontradas, que precisam ser testadas de forma mais intensa e por mais tempo do que tiveram a oportunidade de ser usadas.

De acordo com o relatório de The Standish Group (1994), a principal causa de falhas em projetos é a incompletude dos requisitos. Um processo de desenvolvimento adequado às necessidades do projeto e da equipe por ele responsável tende a permitir a ocorrência de um número menor de erros nas fases de análise e especificação de requisitos. Outro problema também citado no relatório, a falta de planejamento, poderia ser totalmente solucionado com a adoção de um processo adequado, que preveja este tipo de atividade.

1.2 Hipótese

Com base em características de uma equipe de desenvolvimento de software para solucionar um determinado problema e nos recursos disponíveis, é possível estabelecer, de forma semiautomática, um processo de software adequado à realidade descrita.

1.3 Objetivos

Este trabalho se propõe a estabelecer um método baseado em planejamento distribuído para definir processos de software adequados a contextos determinados, com base em características da equipe responsável pelo desenvolvimento de um projeto de software e nos recursos disponíveis. Neste contexto, serão considerados recursos o tipo de pessoal disponível e o tempo de execução das tarefas e do projeto.

Especificamente, pretende-se:

a) Criar um modelo algébrico dos diversos elementos relacionados a um processo de desenvolvimento de software;
b) Definir conceitualmente um método para estabelecer um processo de desenvolvimento de software a partir de um conjunto de dados de entrada relacionados a um projeto e equipe específicos;
c) Criar um sistema multiagente que implemente o método proposto;
d) Executar o sistema implementado em diferentes cenários, de forma a assegurar que a saída gerada condiz com as mudanças na entrada.

1.4 Metodologia

A primeira etapa do trabalho consistiu em fazer uma revisão bibliográfica tão profunda quanto possível dentro do período destinado à sua realização. Esta revisão visou apontar os possíveis caminhos a serem tomados para se iniciar a definição da técnica de modelagem de processos proposta. Apesar de ter sido feita com base em textos de diferentes áreas do conhecimento, incluindo, mas não se limitando a: Inteligência Artificial, Engenharia de Software, modelagem de processos de negócio e, mais especificamente, de processos de software, não foi encontrado muito material atual referente a pesquisas recentes na área de automatização de customização de processos de software.

Como não foram encontradas muitas indicações na bibliografia levantada, foi necessário definir os possíveis elementos para constituir a modelagem dos conceitos envolvidos com a tarefa de customizar um processo de software. Dentre esses conceitos, escolheu-se quais devem ser fornecidos como entrada para o sistema derivado do método proposto.

A partir da modelagem dos conceitos fundamentais, o método em si foi especificado, na forma de um sistema multiagentes. Foram definidos: a arquitetura dos agentes componentes do sistema; as regras a serem obedecidas para manutenção da base de conhecimentos dos mesmos; os tipos de interação interagente e o formato das mensagens trocadas entre eles.

Com esta base representativa construída, foram escolhidos ambiente e linguagem para implementação e execução do sistema proposto, dentre os disponíveis no mercado. Optou-se por ferramentas de código aberto, de forma que fosse possível realizar eventuais alterações, caso necessárias.

Depois de ter o ambiente preparado, foi realizada a implementação do sistema em

si, utilizando os elementos de representação inicialmente definidos, realizando devidos ajustes conforme a necessidade. De posse da implementação, foram feitas execuções do sistema, simulando cenários a fim de avaliar o cumprimento dos objetivos estabelecidos.

1.5 Organização do texto

Este capítulo apresentou o problema, sua importância e os objetivos que se buscou alcançar com o trabalho. O restante do texto está organizado da seguinte forma:

- O capítulo 2 apresenta os conceitos da teoria sobre sistemas multiagentes e planejamento distribuído utilizados no trabalho, obtidos por meio de pesquisa na literatura da área;
- O capítulo 3 apresenta conceitos fundamentais, fruto de pesquisa bibliográfica, sobre processos de desenvolvimento de software, bem como sobre sua customização;
- O capítulo 4 introduz o método proposto para automatizar parte da atividade de customização de processos de software em contextos específicos;
- O capítulo 5 apresenta a implementação do sistema multiagente proposto e descreve sua utilização em um estudo de caso executado com o intuito de avaliar a eficácia do sistema;
- O capítulo 6 apresenta as conclusões da realização do trabalho e apresenta possibilidades de desdobramentos futuros.

Os apêndices apresentam os códigos relacionados com a implementação e o estudo de caso desenvolvidos.

2 SISTEMAS MULTIAGENTES E PLANEJAMENTO DISTRIBUÍDO

"None of us is as smart as all of us"
Ken Blanchard

"A goal without a plan is just a wish"
Antoine de Saint-Exupéry

A aplicação do conceito de agentes de software em estudos na área de Inteligência Artificial desperta interesse já há algum tempo e este trabalho é fortemente fundamentado nesse conceito. Mais ainda, é pautado nas interações entre agentes e nas habilidades de um grupo de entidades planejar ações que atendam a determinados objetivos. Este capítulo visa apresentar os conceitos fundamentais dos assuntos tratados neste trabalho relacionados a agentes, suas interações e habilidades de planejamento.

2.1 Sistemas multiagentes

Sistemas multiagentes, como o próprio nome indica, são sistemas formados por mais de um agente. Neste sentido, é importante partir de um conceito para o termo agente, a fim de estabelecer uma definição para sistemas formados por grupos destas entidades.

Não há uma única definição para o termo *agente inteligente*, no âmbito dos sistemas de software. Wooldridge (2002) diz que é "um sistema de computador *situado* em algum *ambiente* e que é capaz de *agir autonomamente* nesse ambiente para que possa atingir seus objetivos". Outra abordagem é a noção de *agente racional*: "(...) um agente racional ideal deveria fazer a ação que se espera maximizar sua medida de performance, baseado na sequência de percepções e quaisquer conhecimentos que o agente tenha" (RUSSELL; NORVIG, 2002). Neste trabalho, entende-se o termo *agente* conforme apresentado na definição a seguir.

Definição 2-1. Um *agente* é um sistema de software capaz de agir de forma independente, de acordo com suas percepções e objetivos, dotado de capacidade de comunicação com outros agentes.

A capacidade de comunicarem entre si torna os agentes adequados para a

realização de tarefas em conjunto e permite sua organização em sociedade. Soluções baseadas em sistemas multiagentes podem ser adotadas por razões diversas. O problema pode ter uma natureza distribuída, tanto com relação à disponibilidade de informações quanto ao seu processamento. Por exemplo: sistemas para mineração de dados espalhados em múltiplas bases. Outra motivação, ainda relacionada ao problema, diz respeito ao fato de a modelagem poder ser feita utilizando agentes de maneira natural. Este tipo de situação acontece nos casos em que o próprio problema pode ser compreendido como um sistema multiagente. Exemplos incluem sistemas para estudos do comportamento de sociedades ou simulações de transmissão de dados em rede.

Em se tratando da solução em si, uma possível razão para a utilização de sistemas multiagentes é a simplificação do projeto e implementação do software, em certos casos. Alguns problemas são muito complexos e podem ser divididos em subproblemas a serem resolvidos por sistemas específicos e relativamente simples, se comparados a o que seria necessário para solução do problema original. São exemplos deste tipo de abordagem os projetos de computação distribuída, como SETI@home[1] e Folding@home[2].

A lista de fatores motivadores da adoção de sistemas multiagentes como solução para problemas complexos é extensa. Este texto não tem como objetivo esgotar o assunto. Mais informações podem ser encontradas nos textos de Bordini, Vieira e Moreira (2001) e Wooldridge (2002).

As interações entre diversos agentes podem ser complexas, em função da autonomia e dos objetivos individuais de cada um dos envolvidos. Tal complexidade pode tornar mais difícil a tarefa de atingir um objetivo comum a todo o grupo, pois os agentes podem enfrentar situações nas quais suas metas individuais conflitam com as da sociedade. Este é um dos fatores apontados por Wooldridge (2002) para justificar o estudo de sistemas multiagentes de maneira independente dos sistemas distribuídos convencionais.

2.2 Agentes planejadores

Apesar de agentes serem dotados de certa autonomia para tomar suas decisões, em certas ocasiões esta característica pode não ser suficiente para permitir a um agente atingir seus objetivos. Em situações nas quais os objetivos são complexos, não basta autonomia para decidir quais ações tomar. É necessário que o agente consiga escolher as melhores ações que o levam ao objetivo final. Especialmente em ambientes

1 Disponível em: <http://setiathome.ssl.berkeley.edu>
2 Disponível em: <http://folding.stanford.edu>

desconhecidos, o agente precisa obter retorno sobre o impacto das ações tomadas e tentar antever os resultados das próximas. Desta forma, pode ser necessário estabelecer uma sequência de ações para levá-lo de seu estado atual a um desejado antes mesmo de iniciar qualquer trabalho. Segundo Russell e Norvig (2002), à atividade de determinar tal lista de ações dá-se o nome de Planejamento. Geffner (2002) apresenta um conceito mais genérico, sendo Planejamento a atividade ligada à geração automática de estratégias de ação – planos – a partir da descrição de ações, sensores e objetivos.

É conveniente citar algumas características dos problemas de planejamento e técnicas para solucioná-los apresentadas pelos autores supracitados, a fim de contextualizar adequadamente o cenário no qual se inserem os agentes planejadores.

Para permitir a realização de inferências e criar uma lista adequada de ações para atingir um objetivo, um agente planejador precisa conhecer a estrutura do problema, através da descrição de ações, estados e objetivos. Desta forma, ele pode tirar proveito desse conhecimento e evitar trabalhos sabidamente infrutíferos.

Outro aspecto desejável é a utilização de boas heurísticas, para reduzir o espaço de possíveis soluções a serem exploradas pelo agente. Em problemas de larga escala, esta característica é essencial para possibilitar processamentos em períodos de tempo considerados aceitáveis.

Uma propriedade explorada na solução de muitos problemas de planejamento é a capacidade de decomposição do mesmo em partes menores. Esta estratégia, conhecida em diversas áreas da Ciência da Computação como “dividir para conquistar”, pode aumentar muito a eficiência dos agentes, mas traz consigo questões referentes aos pontos de divisão do problema e junção das soluções. Em cada caso, cabe uma análise dos custos envolvidos; quando os subproblemas têm custos menores, mas a divisão do problema principal ou a junção das soluções forem demasiadamente complexas, pode ser interessante não trabalhar com a decomposição.

Os modelos para problemas de planejamento devem incorporar elementos suficientes para permitir uma solução adequada. Quanto mais complexo é um problema, mais detalhada e sofisticada deve ser sua representação. Problemas de *planejamento clássico* situam-se em ambientes simples – determinísticos, totalmente observáveis, finitos, estáticos e discretos. Neste caso, as estratégias de solução geralmente envolvem a realização de buscas pelo espaço de estados possíveis.

Em ambientes mais complexos, além de ser necessário um modelo mais complexo, também são adotadas estratégias de solução mais sofisticadas e demandantes de recursos. Alguns dos elementos que contribuem para a complexidade de um

problema são: incerteza a respeito do estado do mundo e dos resultados das ações; possibilidade de o agente obter retorno sobre as ações executadas e, assim, poder alterar o curso das próximas ações; noção de tempo, quando é levada em consideração a duração das ações e possibilidade ou não de execução simultânea de duas ou mais delas.

Restrições, de qualquer espécie, também contribuem para a complexidade da modelagem do problema e da sofisticação da solução. Restrições de tempo e recursos são comumente encontradas e um tratamento adequado deve ser dado a elas desde a modelagem do problema até a execução da solução.

As características e técnicas apresentadas não constituem uma lista exaustiva. Não foram apresentados os fundamentos matemáticos e as definições completas de cada uma, pois fogem ao escopo deste texto. Elas foram aqui elencadas para dar dar uma visão geral sobre planejamento, um dos fundamentos da proposta apresentada no capítulo 4.

2.3 Arquitetura BDI

Para que um software possua as características de um agente, é necessário escolher adequadamente as estruturas para representar sua base de conhecimentos e seu comportamento. Dentre as possíveis maneiras de modelar agentes, o autor optou por trabalhar com arquiteturas BDI (do inglês, *Belief*, *Desire and Intention* – Crença, Desejo e Intenção).

O modelo BDI é fundamentado em três conceitos principais, que o nomeiam: crenças, desejos e intenções. Crenças constituem os conhecimentos que o agente possui como verdadeiros, a partir de suas percepções ou inferências. Desejos representam os estados de mundo aos quais o agente deseja chegar. Intenções representam os desejos selecionados pelo agente para serem atingidos.

Segundo Walczak et al. (2006), sistemas BDI podem lidar com ambientes dinâmicos e podem ser utilizados para implementar tanto agentes com comportamento reativo quanto aqueles com objetivos de longo prazo. Neste sentido, agentes BDI agem deliberadamente, não são apenas reativos, apesar de poderem apresentar este tipo de comportamento.

Por ser um modelo bem difundido, foge ao escopo deste texto descrever em detalhes os aspectos dos sistemas baseados em arquiteturas BDI. Os trabalhos com esse tipo de sistema foram motivados pela teoria do raciocínio prático, de Bratman (1987) e se iniciaram baseados nos conceitos apresentados por Bratman, Israel e Pollack (1988). Uma das primeiras e mais conhecidas arquiteturas que o implementam foi desenvolvida

por Georgeff e colaboradores (GEORGEFF, INGRAND, 1989; GEORGEFF, LANSKY, 1987).

2.3.1 Estrutura para implementação

O uso de uma plataforma de desenvolvimento para suportar a implementação de sistemas baseados em arquiteturas BDI é essencial para o desenvolvimento deste trabalho. Neste sentido, foram buscadas ferramentas e linguagens para lidar com este tipo de sistema. A investigação levou à avaliação de duas possibilidades: Jason (BORDINI, HÜBNER, VIEIRA, 2005) e 2APL (DASTANI, HOBO, MEYER, 2007; DASTANI, 2008).

Jason é um interpretador para uma versão estendida da linguagem AgentSpeak, denominada AgentSpeak(L) (RAO, 1996). Segundo os autores do ambiente, "AgentSpeak(L) é uma linguagem de programação baseada em lógica orientada a agentes, com o objetivo de implementar sistemas de planejamento reativos".

Um dos fatores que levaram à investigação desta ferramenta foi o fato de ela ser do conhecimento do autor deste trabalho, que já havia tido contato com o ambiente e conhecia seus recursos, possibilidades e limitações.

Agentes escritos em AgentSpeak são formados por uma base de crenças e uma biblioteca de planos e constituem um sistema de planejamento reativo. Percepções do ambiente que alterem sua base de conhecimentos ou mudança na base de planos originam eventos, aos quais o agente reage, executando um plano aplicável, selecionado a partir da biblioteca.

Um dos pontos fortes mais notáveis da arquitetura é a capacidade de estendê-la. Para todos os principais elementos, uma implementação específica pode ser feita em linguagem Java. Entre os pontos de personalização, pode-se citar: os próprios agentes, arquitetura da base de conhecimentos, funções internas para seleção de eventos e planos, entre outros.

Uma das possíveis dificuldades a serem enfrentadas com o uso de Jason seria a implementação da comunicação entre os agentes, já que até o momento da busca por ferramentas não havia mecanismo para executar a plataforma sobre um ambiente multiagente de conhecimento do autor, como Jade (BELLIFEMINE et al., 2005). Outro importante fator considerado é a dificuldade de alteração da biblioteca de planos durante a execução do agente, visto que não há mecanismos para estabelecer a criação de novos planos e incluí-los à biblioteca.

2APL é uma linguagem, definida por seus autores como "uma linguagem de

programação orientada a agentes baseada em BDI”. A linguagem provê construtos para definição das bases de conhecimentos, de planos e de objetivos dos agentes, bem como recursos para manipulá-las durante a execução do sistema. Para permitir a implementação e execução de sistemas em 2APL, os autores disponibilizam uma plataforma de execução desenvolvida em linguagem Java.

A linguagem provê recursos sintáticos distintos para a definição dos agentes e do sistema multiagentes. Para a definição do sistema, existe um arquivo contendo a lista de agentes, indicando o arquivo que os implementa. É possível ter várias instâncias de agentes baseados em um mesmo arquivo.

A porção da linguagem dedicada à definição individual dos agentes provê construtos para definir o estado inicial das bases de crenças, objetivos e planos dos agentes. A linguagem possui ainda recursos para definir ações e regras de raciocínio. As ações podem ser de atualização da base de crenças, de comunicação, de teste sobre a base de crenças ou de objetivos, de modificação da base de objetivos, além de ações abstratas – mecanismo para encapsular todo um plano em uma ação – e externas – afetam o ambiente no qual o agente se situa. As regras são definidas para alterar a base de planos dos agentes, sendo possível definir regras para geração de planos, de acordo com os objetivos e crenças do agente ou em resposta a eventos externos ou à execução de atividades abstratas, bem como para reparar uma ação cuja execução tenha falhado.

A base de crenças de cada agente é implementada como um programa em linguagem Prolog (BRATKO, 1986), assim como todos os testes executados sobre a base durante a execução dos planos do agente. Esta característica permite aos usuários dessa linguagem de programação usufruir do ambiente sem necessidade de grandes adaptações.

No seu aspecto geral, a linguagem se assemelha a Prolog não apenas nas definições e testes sobre a base de conhecimentos, mas também nas definições de objetivos, planos e regras. Descrever em detalhes dos componentes da linguagem foge ao escopo deste texto. Uma descrição pode ser encontrada no apêndice A.

As versões iniciais da plataforma 2APL foram desenvolvidas sobre o *framework* Jade, plataforma que provê recursos para a construção de sistemas multiagentes, incluindo aspectos relacionados à comunicação e controle de concorrência. A versão mais atual no momento da busca permitia o uso da plataforma integrado ou não ao ambiente Jade. Em ambos os casos, o desenvolvedor do sistema multiagente não precisa se preocupar com a entrega de mensagens trocadas entre os agentes ou com a implementação dos meios para tráfego das mesmas, nem fazer controle de execução

concorrente dos vários agentes. A plataforma gerencia todos esses aspectos.

Por sua flexibilidade, seu poder de expressão e, principalmente, devido ao fato de a plataforma que a implementa gerenciar o processo de comunicação e o paralelismo da implementação, 2APL foi a linguagem escolhida para implementar a proposta apresentada no capítulo 4.

2.4 Planejamento distribuído

Sistemas multiagentes podem ser desenvolvidos com o intuito de resolver problemas de planejamento. Este tipo de problema é descrito por Boutilier (1996) como o de definir políticas ou estratégias de ação efetivas para um grupo de *n* agentes que compartilham objetivos comuns. Apesar de a definição do problema ser semelhante à apresentada na seção 2.2, existem diferenças entre planejamento centralizado (com um agente) e distribuído entre vários agentes. A diferença fundamental, foco da discussão desta seção, reside em como coordenar o grupo de agentes na busca pela solução conjunta para o problema de planejamento.

Antes de iniciar a discussão sobre coordenação, segue a livre tradução de uma definição de problema de planejamento multiagente, apresentada por Weerdt, Mors, Witteveen (2005):

> *Dada uma descrição do estado inicial, um conjunto de objetivos globais, um conjunto de (pelo menos dois) agentes, e para cada agente um conjunto de suas capacidades e seus objetivos particulares, encontrar um plano para cada agente que atinja seus objetivos individuais, de forma que esses planos juntos sejam coordenados e os objetivos globais também sejam atingidos.*

Esta segunda definição deixa clara a necessidade de coordenar os agentes envolvidos com a solução do problema. A discussão sobre estratégias de coordenação conduzida nesta seção não tem como objetivo esgotar o assunto; serve apenas ao propósito de contextualizar o leitor e dar fundamentação às estratégias de coordenação adotadas na proposta apresentada no capítulo 4. Informações mais detalhadas sobre o assunto podem ser encontradas nos trabalhos de Boutilier (1996) e Weerdt, Mors, Witteveen (2005).

Outra característica a ser observada nos problemas de planejamento multiagente é o nível de cooperação entre os agentes. O planejamento é feito com o intuito de que um grupo de agentes atinja um objetivo comum. Neste sentido, metas individuais conflitantes com o objetivo global podem dificultar o estabelecimento dos planos individuais de forma a permitir sua execução conjunta. Assim, é desejável que haja

cooperação entre os envolvidos, para facilitar a obtenção de uma boa solução para o problema global. Entretanto, muitas situações envolvem agentes não cooperativos ou até mesmo competitivos. Nestes casos, além de agirem de forma coordenada, eles precisam fazer acordos para conseguir estabelecer seus planos de ação e, assim, solucionar o problema global. O foco deste texto não é abordar técnicas de negociação, pois o sistema proposto lida com agentes cooperativos. Informações sobre acordos entre agentes podem ser encontradas no texto de Wooldridge (2002), que dedica um capítulo ao assunto.

O problema de coordenação é tratado por meio de estratégias seguindo diferentes abordagens: comunicação interagente, leis ou convenções sociais e aprendizado.

Soluções baseadas em comunicação são baseadas na troca de mensagens entre os agentes para garantir uma seleção adequada de tarefas que leve à solução do problema global.

As leis ou convenções sociais são mecanismos pertencentes ao sistema, como um todo, e são de conhecimento de todos os agentes. Elas limitam a seleção individual de tarefas, garantindo uma execução bem sucedida do plano conjunto, caso todos os agentes sigam as leis ou convenções estabelecidas. Um exemplo deste tipo de mecanismo é a lei de trânsito, estabelecendo que todos os motoristas trafeguem pela pista da direita.

Estratégias baseadas em aprendizado têm como objetivo chegar a uma solução emergente do comportamento dos agentes. Através de repetidas interações com o meio, os agentes avaliam quais tarefas têm mais ou menos utilidade na solução do problema e adaptam suas escolhas apropriadamente.

Além da definição de quais estratégias adotar para coordenar os agentes, uma importante decisão de projeto do sistema é o momento em que elas serão aplicadas. Basicamente, pode haver ações de coordenação a qualquer momento – antes, durante ou após o planejamento individual. Por exemplo: *previamente ao planejamento*, podem ser estabelecidas convenções sociais, limitando a escolha de tarefas dos agentes. *Durante o trabalho individual*, os agentes podem comunicar entre si de forma a evitar possíveis conflitos nos planos gerados. *Após esta etapa*, os planos devem ser unificados e, para isto, podem ser usadas técnicas de negociação, como leilões, para resolver os conflitos detectados.

O objetivo deste capítulo foi o de dar uma visão geral sobre sistemas multiagentes e planejamento distribuído, conceitos utilizados no desenvolvimento do trabalho do qual se originou a proposta apresentada no capítulo 4, bem como apresentar algumas das

possibilidades para implementação de sistemas multiagentes e, dentre as duas mais profundamente estudadas, qual foi a escolhida para estruturar a implementação do método proposto, que é apresentada em detalhes no capítulo 5. Em nenhum momento se intentou conduzir uma discussão exaustiva sobre o assunto, que é muito vasto e foge ao escopo deste texto.

3 CUSTOMIZAÇÃO DE PROCESSOS DE SOFTWARE

"Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away"
Antoine de Saint-Exupery

Profissionais de engenharia de software despendem grande quantidade de esforços para resolver uma variada gama de problemas de diferentes áreas do conhecimento. Entretanto, como aponta Kruchten (2000), as atividades de construção e manutenção de software são difíceis de serem conduzidas e têm ficado ainda mais difíceis. Desafio ainda maior é produzir software capaz de atender aos requisitos de qualidade dos clientes dentro do prazo e com o orçamento previstos.

Há um longo caminho a ser percorrido desde um problema a ser resolvido até sua solução tecnológica – o produto de software. Muita informação referente à área de atuação do cliente deve ser absorvida pela equipe de desenvolvimento, muitas vezes em pequeno espaço de tempo. Além disto, há vários interessados no projeto, muitos deles sem qualquer experiência no assunto a ser tratado. Esta diversidade de pessoas e organizações envolvidas acarreta em uma variada gama de pontos de vista e interesses, muitas vezes conflitantes.

Todos os aspectos supracitados devem ser gerenciados, de forma a ser possível chegar a um produto final que atenda às reais necessidades dos usuários. A coordenação de tantas variáveis (requisitos, envolvidos, recursos limitados, etc.) é complexa e se torna menos difícil – quiçá só seja possível – quando o trabalho é dividido em tarefas, atividades, fases e assim por diante. Isto torna a complicada, e talvez intratável, atividade de construir software em um complexo, mas factível e gerenciável, processo.

3.1 Processos de software

É neste contexto de uma vasta e complexa gama de atividades que levam à criação de um produto de software que se situam os Processos de Desenvolvimento de Software ou, simplesmente, Processos de Software, como alguns autores costumam chamá-los

(FUGGETTA, 2000; PFLEEGER, 2004; SOMMERVILLE, 2007). A seguir serão apresentados conceitos de diferentes autores, a partir dos quais se chegou à definição de Processo de Software adotada neste trabalho. Iniciando por um conceito generalista, “Podemos considerar um conjunto de tarefas ordenadas como sendo um processo: uma série de etapas que envolvem atividades, restrições e recursos para alcançar a saída desejada.” (PFLEEGER, 2004). Um conceito ainda geral, mas específico para o domínio de produção de software afirma que “Um processo de software é um conjunto de atividades que leva à produção de um produto de software.” (SOMMERVILLE, 2007). De forma mais específica, um processo de software é: “Um conjunto coerente de políticas, estruturas organizacionais, tecnologias, procedimentos e artefatos que são necessários para conceber, desenvolver, entregar e manter um produto de software.” (FUGGETTA, 2000).

Apesar de apresentarem ideias ligeiramente diferentes, os três autores citados mencionam o fato de um processo ser um meio para se atingir um objetivo. É com base neste pensamento que se chegou à definição de *Processo de Software*, enunciada a seguir.

Definição 3-1. Um *Processo de Desenvolvimento de Software*, ou *Processo de Software*, é uma sequência ordenada de atividades, que atendem a um conjunto de políticas e restrições, para criar e manter um produto de software.

Neste texto, o termo *processo*, isoladamente, se refere a Processos de Software, a menos que explicitamente informado o contrário.

Kruchten (2000) lista seis princípios, ou melhores práticas, de desenvolvimento de software adotadas por organizações de sucesso: desenvolver software *iterativamente*; gerenciar *requisitos*; usar arquiteturas baseadas em *componentes*; modelar software *visualmente*; verificar continuamente a *qualidade* do software; controlar as *mudanças* no software. Segundo o autor, a adoção destas práticas evita o surgimento dos principais sintomas de problemas de desenvolvimento, bem como ataca diretamente as causas dos mesmos. Espera-se, então, que processos adequadamente definidos proporcionem às equipes de desenvolvimento ferramentas para produzir software dentro do prazo e orçamento estimados, pois, segundo Pfleeger (2004), tornam um conjunto de atividades mais consistente e estruturado. Entretanto, processos são complexos e dependem de julgamento humano (SOMMERVILLE, 2007), o que pode tornar a adoção de métodos sistemáticos para desenvolvimento de software uma tarefa onerosa tanto em tempo quanto em recursos.

A publicação de The Standish Group (1994) apontou os dez principais fatores que

levaram ao cancelamento ou estagnação de projetos. Dentre esses, pelo menos cinco podem ser associados ao processo de desenvolvimento: requisitos incompletos, falta de apoio executivo, requisitos e especificações mutáveis, falta de planejamento e falta de gerenciamento de tecnologia da informação. Por outro lado, dentre os dez principais fatores de sucesso apontados, quatro podem ser atingidos com processos adequados e seguidos corretamente: requisitos claramente identificados, planejamento adequado, entregas menores, visão e objetivos claros. Não foram encontrados dados mais recentes sobre sucesso ou fracasso de projetos de software, mas também não foram encontrados elementos capazes de indicar mudanças neste cenário. A análise dos fatores apresentados em conjunto com as boas práticas recomendadas por Kruchten (2000) permite inferir que a adoção de processos que sigam tais práticas pode se constituir como elemento para conduzir projetos de desenvolvimento de software ao sucesso.

Existem diversos exemplos de processo na literatura e cada um possui características únicas. Entretanto, é possível identificar semelhanças entre conjuntos de processos e, assim, agrupá-los em categorias. Dentre as possibilidades de categorização, uma possível distinção é feita entre Métodos Dirigidos por Planejamento (MDP) e Métodos Ágeis (MA). Este trabalho não aborda, especificamente, nenhuma das duas classes de métodos, mas dá ênfase nos MDP no estudo de caso conduzido. A seguir, uma breve conceituação e exemplos de processos pertencentes a cada categoria.

3.1.1 Métodos dirigidos por panejamento

A construção de software por meio de *Métodos Dirigidos por Planejamento* (MDP) é sistemática, pois este tipo de processo é focado na disciplina, com todas as fases do desenvolvimento muito bem documentadas (SOARES, 2007).

Alguns exemplos de MDP: *Personal Software Process* (PSP) (HUMPHREY, 2000a), *Team Software Process* (TSP) (HUMPHREY, 2000b) e *Rational Unified Process* (RUP) (KRUCHTEN, 2000). Com relação ao RUP, existem ainda processos baseados nele, como o descrito por Wazlawick (2004), e também aqueles a ele assemelhados, como o *Praxis* (PAULA FILHO, 2009). A disponibilidade de uma gama de processos relacionados direta ou indiretamente ao RUP é uma das razões para o enfoque dado a ele neste trabalho.

Uma descrição detalhada do RUP foge ao escopo deste texto, mas uma abordagem sucinta é adequada, dado o interesse neste modelo de processo durante o desenvolvimento do trabalho. Kruchten (2000) o descreve como uma abordagem disciplinada para atribuir tarefas e responsabilidades em uma organização, com o

objetivo de garantir a produção de software de alta qualidade, que atenda às necessidades dos usuários finais.

3.1.2 Métodos ágeis

Os *Métodos Ágeis* (MA), por sua vez, têm como premissas a rapidez na obtenção de resultados e maior capacidade de adequação a mudanças de requisitos (SOARES, 2007). A autora destaca ainda o fato de serem baseados em princípios que enalteçam a comunicação entre os integrantes da equipe de desenvolvimento e se apoiem fortemente nas habilidades individuais dos mesmos, além de suportar a mudança ou o surgimento de novos requisitos do produto durante o desenvolvimento. Os MA são baseados na prática e não definem processos detalhados, mas fornecem conselhos para que a eficácia na modelagem seja alcançada (AMBLER, 2004).

Exemplos de MA comumente citados são: *eXtreme Programming* (XP) (BECK, 2000) e *Scrum* (SCHWABER, BEEDLE, 2001). Descrever as metodologias é algo que foge ao escopo deste texto, mas pode-se dizer que se assemelham, no sentido de seguirem os preceitos da modelagem ágil, mas diferem nas especificidades das orientações fornecidas.

3.2 Customização de processos de software

É possível encontrar na literatura outros processos de software predefinidos, além dos anteriormente citados. Entretanto, existem necessidades específicas não totalmente cobertas ou tratadas adequadamente por todos eles. As equipes de desenvolvimento são diferentes, seja no número de profissionais envolvidos, em suas especialidades ou ainda em seu nível de conhecimento a respeito do problema a ser solucionado pelo software. Por estas, e outras, razões sempre é necessário adequar um processo de software a uma realidade específica (ROYCE, 1998). As mudanças necessárias para adaptar um processo incluem modificação, alteração, adição ou supressão de elementos tais como artefatos, atividades e papéis (KRUCHTEN, 2000). As informações e técnicas sobre adaptação de processos de software apresentadas neste capítulo são baseadas principalmente em literatura relacionada ao Processo Unificado (KRUCHTEN, 2000; ROYCE, 1998).

A adaptação, também chamada de customização neste texto, dos processos de software visa a ajustar os parâmetros de modelos de processo existentes à realidade enfrentada pelas equipes de desenvolvimento. Alguns processos são mais especificamente denominados por seus autores como *frameworks*, como é o caso do

RUP, por causa da pressuposição de que os responsáveis por sua adoção farão adaptações. Além disto, as equipes podem combinar características de dois ou mais processos para adotar um método para desenvolvimento ajustado às suas realidades. Tal combinação pode inclusive envolver processos de categorias distintas, unindo características de MDP e MA, formando o que Soares (2007) define como *Métodos Híbridos*. Este trabalho não contempla este tipo de combinação, mas não há indícios de que tal possibilidade não possa ser explorada.

A customização de um processo pode ser feita para toda uma organização, criando um modelo de processo para todos os projetos da mesma, ou especificamente para um projeto. Em ambos os casos, um trabalho de customização realizado de forma não organizada pode resultar em um modelo de ciclo de vida não adequado à realidade da equipe ou do projeto em questão. Para conduzir o trabalho devem ser considerados dois fatores de variação do processo: complexidade gerencial e complexidade técnica. A figura 3-1 mostra as diferentes prioridades a serem consideradas ao longo das duas dimensões.

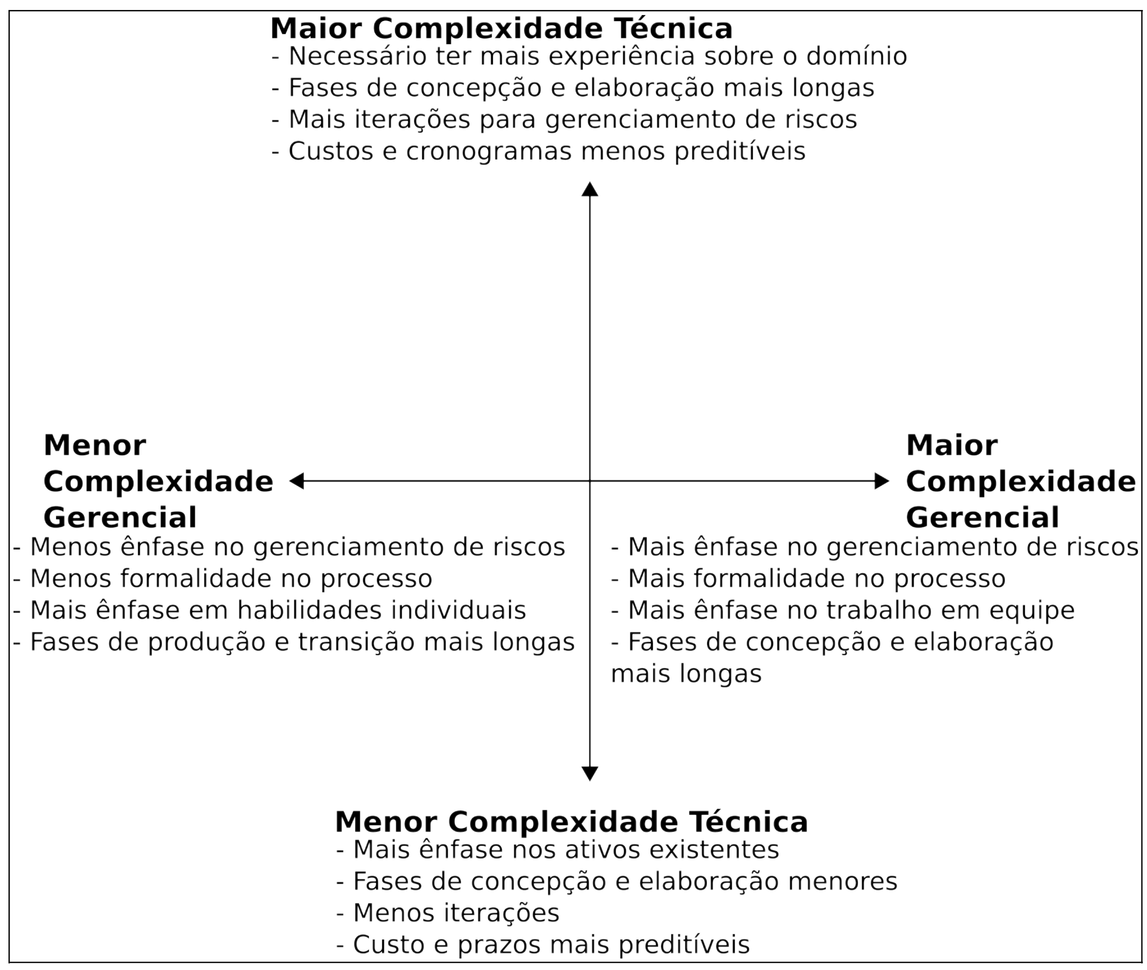

Figura 3-1. *Prioridades para adaptar o modelo de processo*

Fonte: adaptado de Royce (1998)

Para definir processos para toda a organização, os dois fatores devem ser analisados para todos os projetos da mesma. Os parâmetros de customização são definidos pelo ponto em que o projeto (ou conjunto de projetos) se encontra nas duas dimensões. Não foram encontrados critérios objetivos para a definição de tais parâmetros na literatura, o que torna todo o procedimento dependente do julgamento de um ou mais especialistas com conhecimento da realidade em questão e experiência em gerenciamento de projetos.

A definição de um processo de desenvolvimento personalizado, adequado à realidade da equipe ou do projeto em questão, pode demandar uma quantidade considerável de esforço, além de conhecimentos profundos de especialistas, algumas vezes não disponíveis. Além disto, os benefícios da implantação de um novo processo de desenvolvimento nem sempre são imediatos, pois é uma atividade demorada e que envolve mudança de cultura das pessoas envolvidas com o desenvolvimento. Ainda assim, organizações escolhem – ou necessitam – personalizar um modelo de ciclo de vida para que não sigam cegamente um processo, gerando trabalho inútil (KRUCHTEN, 2000). Outro fator importante no direcionamento das decisões é a qualidade do software, que tem forte relação com o processo, pois a qualidade deste é crítica para a qualidade do produto final (SOMMERVILLE, 2007). A possibilidade de ser capaz de garantir a qualidade do produto final é um importante fator competitivo para as organizações que desenvolvem software.

Para ilustrar a necessidade de fazer a customização dos modelos de processos, considere dois produtos distintos: um software para controle dos sistemas de uma aeronave e um sistema de gestão integrada. Eles possuem conjuntos muito diferentes de requisitos, tanto funcionais quanto não funcionais, e as características dos produtos finais diferem em plataforma alvo, tempo de resposta e assim por diante. Portanto, o tipo de esforço necessário para desenvolver cada um deve ser muito diferente, o que conduz ao raciocínio de que é provável que os processos seguidos para desenvolvê-los sejam diferentes. Supondo o contrário, ou seja, que os projetos para desenvolver ambos os produtos sigam o mesmo modelo de processo, é possível distinguir três cenários: (a) o processo pode ser mais adequado ao projeto de desenvolvimento do primeiro software; (b) as atividades podem ser mais direcionadas para o segundo projeto; (c) o processo de desenvolvimento pode não ser adequado para nenhum dos dois. Em todas as situações há problemas gerenciais, impactando diretamente a qualidade do produto final. Nas situações (a) e (b) um dos projetos é mais favorecido pelo processo, enquanto o outro sofrerá, quer com atividades de desenvolvimento excessivamente

documentadas, dada a realidade do projeto, quer com tarefas não detalhadas o suficiente, considerando-se a complexidade do sistema. Pode haver excesso de comunicação entre os integrantes da equipe, desviando o foco de atenção do trabalho principal para a comunicação. O contrário também pode ocorrer, ou seja, haver pouca comunicação entre a equipe, acarretando em membros não adequadamente informados sobre o andamento do projeto e possíveis atrasos do mesmo. A situação (c) é indesejada para os dois projetos, pois exige adaptação das equipes de ambos ao processo.

3.2.1 Automatização da tarefa de customização

A grande quantidade de esforço necessária para empreender trabalhos de customização de processos de software e a falta de critérios objetivos para estabelecer metodologias para o desempenho da tarefa podem levar muitas equipes a deixarem de adotar processos mais adequados às suas atividades. Se houver meios para automatizar esta tarefa, no todo ou em parte, as organizações, principalmente as de menor porte – tanto em número de profissionais quanto em quantidade de recursos disponíveis –, poderão ter mais condições de seguir processos adaptados às suas realidades.

Osterweil (1987) propõe que processos de software sejam descritos como software, ou seja, utilizar técnicas de programação para descrever processos de desenvolvimento de maneira formal e, portanto, executável em um computador. Por outro lado, Sommerville (2007) afirma que não poderá existir, em um futuro próximo, a possibilidade de existir uma automação das atividades de engenharia de software de forma extensa, a ponto de transferir o processo criativo de engenheiros e programadores para o software. O desenvolvimento deste trabalho leva em consideração o uso de tecnologias de software para auxiliar o processo de customização, sem ter a pretensão de eliminar a subjetividade e o julgamento humano.

A possibilidade de criar versões customizadas de processos de desenvolvimento, com menos esforço do que o necessário para conduzir a tarefa sem apoio computacional, pode permitir às organizações a adoção de processos mais condizentes com suas realidades. Esta possibilidade é de grande interesse das equipes que não utilizam versões adaptadas de processos de software e também das que não seguem nenhum processo estruturado, com o intuito de melhorar a qualidade de seus produtos e reduzir custos operacionais. Além disto, as equipes que já empregam versões customizadas podem se beneficiar da redução do esforço necessário para criar adaptações mais rapidamente, em casos de mudança de equipe, ou ainda para fazer ajustes no processo para cada projeto de desenvolvimento.

Outro fator motivador para a customização de processos a ser levado em consideração neste contexto são os modelos de maturidade de software, como CMMI (CMMI PRODUCT TEAM, 2002) e SPICE (ISO/IEC, 2003). Entre outras coisas, eles propõem a melhoria contínua do processo de desenvolvimento. Entretanto, apesar de estabelecerem o objetivo (*o que* fazer), os modelos não definem claramente os meios para atingi-los (*como* fazer). Geralmente, este tipo de trabalho envolve a análise do processo atual e a proposição de alterações no mesmo, de forma a torná-lo cada vez mais gerenciável. A análise e identificação de melhorias são feitas com base na experiência dos profissionais responsáveis pelo gerenciamento da aplicação do modelo de maturidade e verificação da documentação disponível sobre o processo atual e projetos desenvolvidos de acordo com o mesmo. Depois de identificadas as melhorias, o processo é remodelado para incorporá-las. A redução de tempo e dos esforços necessários para realizar uma customização pode permitir aos especialistas analisar um maior número de diferentes configurações de processos em um mesmo período de tempo, em comparação com a customização sem apoio automatizado. Esta possibilidade viabiliza um julgamento com mais parâmetros para comparação e avaliação, permitindo uma escolha mais adequada às necessidades identificadas.

Para o desenvolvimento deste trabalho, foram pesquisadas as principais fontes de informação da área, mas não foi encontrada quantidade significativa de material sobre automatização de customização de processos de software. Uma possível causa é o fato de o uso de técnicas de inteligência e apoio computacional para controlar processos de software ser ainda incipiente, alvo de pesquisas recentes, porém promissoras.

O trabalho de Zhao, Chan, Li (2005) trata da aplicação de tecnologia baseada em agentes para modelagem de processos e serviu de inspiração para o desenvolvimento deste trabalho. No artigo, os autores descrevem um processo de software como a colaboração de um grupo de agentes de processo que sabem como gerenciar as atividades de desenvolvimento de software e agem como desenvolvedores de software, em termos de planejamento, execução e avaliação de seu trabalho.

4 MÉTODO PROPOSTO

“A mind that is stretched by a new experience can never go back to its old dimensions”
Oliver Wendell Holmes, Jr.

Este capítulo apresenta uma proposta para prover meios de automatizar parcialmente a customização de processos de software fundamentada na construção de um sistema multiagente com elementos de planejamento. A solução não é completamente automatizada, pois depende de entradas que podem ter caráter bastante subjetivo e devem ser providas pelos envolvidos no projeto de desenvolvimento de software para o qual o sistema será utilizado. Além disto, a saída gerada deve ser analisada pelo gerente de projeto responsável ou outros envolvidos, de forma que as ações tomadas pela equipe sejam coerentes com a realidade. Tal análise é fundamental para abordar, principalmente, os aspectos não modelados pelo sistema e, portanto, não levados em consideração durante seu processamento.

A decisão de desenvolver um sistema com múltiplos agentes, ao invés de apenas um, tem algumas razões não necessariamente óbvias. A primeira é a natureza distribuída das informações sobre um projeto de desenvolvimento de software; são vários os profissionais envolvidos e cada um é especialista em uma ou mais áreas ligadas ao projeto. Outro importante fator considerado é o número de variáveis a serem controladas e analisadas em um projeto de desenvolvimento de software; além de serem muitas, as relações e interdependências entre elas não são facilmente expressáveis sem que sejam simuladas. Por último, mas não menos importante, foi considerada a engenharia do sistema. Apesar de um sistema multiagente ter, a princípio, uma criação ligeiramente mais complexa quando comparado a um software monolítico, a manutenção de várias partes de software relativamente simples – os agentes – tende a ser igualmente simples, quando comparada à de um único sistema com as mesmas funcionalidades providas pelo sistema multiagente. Além disto, a divisão da tarefa entre vários agentes permite fazer alterações localizadas no processamento, apenas trocando os agentes desejados.

O início do capítulo apresenta uma descrição geral do funcionamento do sistema, suas entradas e saídas, os tipos de agentes definidos e as atribuições de cada um, como se dá o processo de comunicação e qual o mecanismo de coordenação adotado. A seguir, as principais definições e conceitos base são apresentados. Logo após, a arquitetura do sistema e os tipos de agentes são apresentados. Na sequência, são feitas descrições mais detalhadas do processamento e armazenamento de informações, bem como de todo o processo de comunicação. Ao final, é feita uma explicação sobre o comportamento dos agentes e como se dá seu ciclo de execução.

4.1 Descrição geral

O sistema proposto irá trabalhar sobre as entradas providas com o objetivo de gerar uma sequência totalmente ordenada de tarefas relacionadas ao projeto de desenvolvimento de software. Esta saída constitui a representação do processo de desenvolvimento. Neste trabalho, o termo *processo* muitas vezes é utilizado para se referir a tal representação. Como o trabalho de geração do processo de software é realizado por múltiplos agentes, há a necessidade de que todos se comuniquem e, para tanto, linguagem e protocolos de comunicação devem ser de conhecimento de todos. Cada agente trabalhará individualmente, trocando informações com os demais, sendo o objetivo final do grupo a geração de um plano que represente o processo de desenvolvimento. Desta forma, alguns agentes, ou mesmo todos, podem compartilhar alguns conceitos básicos (ontologia) e representações para que possam gerar o plano final. A figura 4-1 dá uma visão geral das interações existente no sistema.

Há basicamente dois tipos de agentes: *Gerente de Projetos* e *Especialistas*. O gerente de projetos coordena toda a criação automatizada do processo de desenvolvimento. É o responsável por coordenar todas as atividades de comunicação necessárias ao trabalho de cada agente, receber os planos gerados individualmente e uni-los adequadamente para compor o processo final. Cada agente especialista é capaz de gerar planos parciais contemplando atividades de seu domínio. Para tal, cada agente executa uma simulação simplificada do trecho do processo de desenvolvimento sob sua responsabilidade, detectando quais atividades serão adicionadas ao processo e em qual ordem. Para que uma tarefa possa ser inserida no plano gerado, todos os seus requisitos devem ser capazes de ser satisfeitos quando o plano final for executado. Esta característica é assegurada por meio de intensa comunicação entre os agentes durante a simulação de execução do processo, para garantir que haverá meios para atender a todos os requisitos de cada atividade.

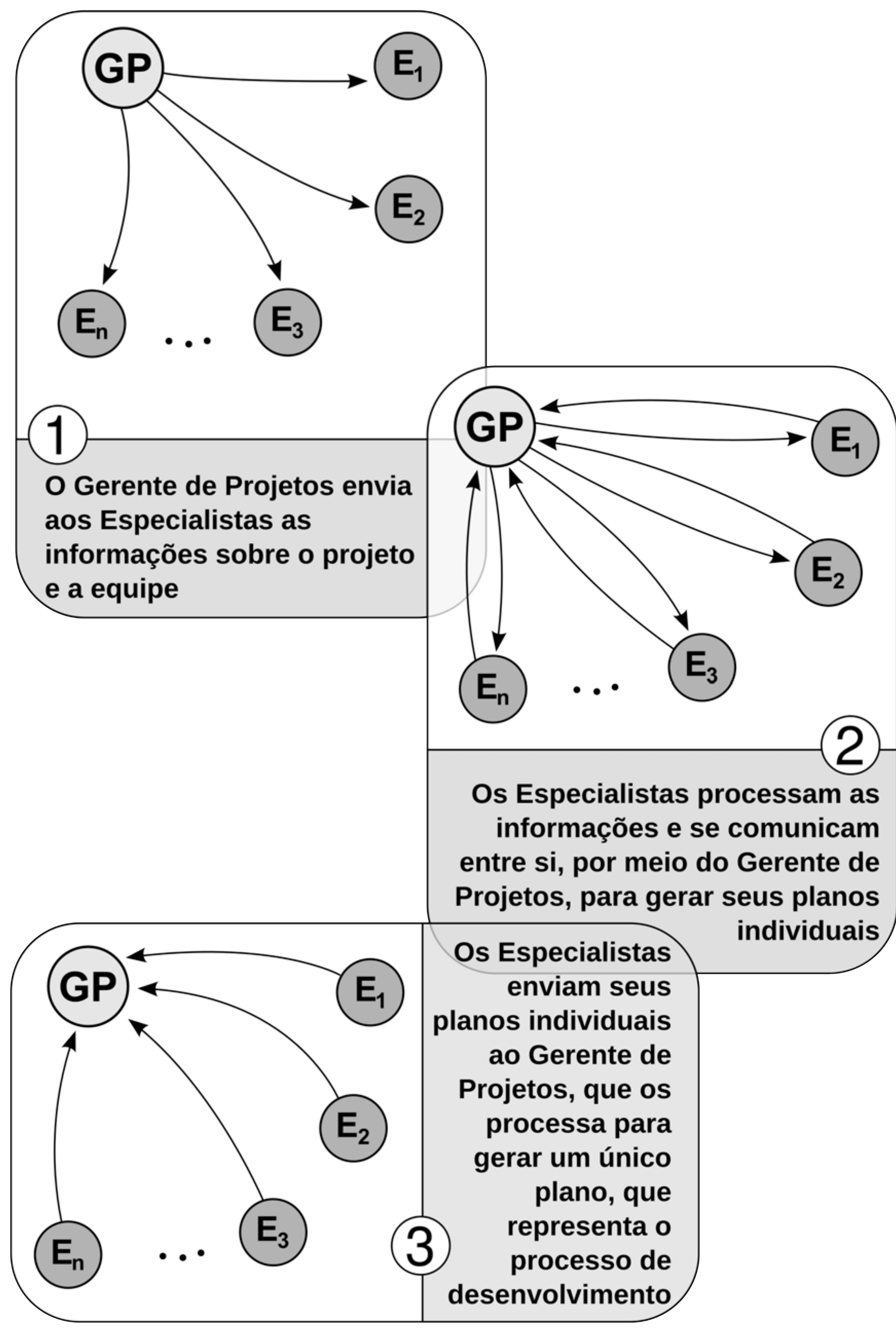

Figura 4-1. *Descrição geral das interações entre os agentes*

Toda comunicação acontece entre dois agentes de cada vez: um especialista e o gerente de projetos. Isto significa que todas as mensagens passam, necessariamente, pelo agente coordenador. Desta forma, quando um especialista tem uma solicitação, esta

é enviada ao gerente de projetos, que é o responsável por encontrar um receptor adequado para a mensagem e fazer o roteamento da mesma. Quando uma resposta for enviada de volta, o coordenador é novamente o responsável pelo encaminhamento, já que ele é o detentor da informação referente ao emissor da solicitação.

Quando um agente especialista termina o trabalho de criação de seu plano parcial, este é enviado ao gerente de projetos. Quando todos os planos locais são recebidos pelo agente coordenador, as tarefas de desenvolvimento são ordenadas de forma a gerar um único plano – forma pela qual o sistema modela o processo de desenvolvimento de software. Tal ordenação leva em consideração os pré-requisitos das atividades, garantindo um plano consistente. Este plano ordenado e executável é a saída desejada da execução do sistema sobre as entradas providas.

4.2 Definições subjacentes

O sistema proposto é baseado em determinados conceitos e representações que serão apresentados nesta seção. Algumas das definições apresentadas são diretamente relacionadas ao RUP, já que este foi o modelo de processo de desenvolvimento escolhido como referência neste trabalho. Outras definições são, de certa forma, genéricas e poderiam ser utilizadas caso o modelo adotado fosse outro. A seguir são apresentadas as definições usadas na descrição desta proposta.

Definição 4-1. Um *papel* define o comportamento e as responsabilidades de um indivíduo ou de toda uma equipe (KRUCHTEN, 2000). Neste trabalho, o conjunto R, de todos os papéis, tem a seguinte definição:

$$R=\{r \mid r=\langle n, T\rangle\}$$

Onde n representa o nome do papel e T é uma lista de todas as tarefas capazes de serem executadas pelos indivíduos ou equipes que desempenham o papel.

Definição 4-2. Um *modelo de artefato* representa um tipo de artefato disponível para uso no projeto. A definição do conjunto M, de todos os modelos de artefatos disponíveis, é assim representada:

$$M=\{m \mid m=\langle R_U, R_R, p\rangle\}$$

Cada modelo m tem como componentes dois conjuntos, R_U, $R_R \subseteq R$, que simbolizam, respectivamente, os papéis com permissão para atualizar e ler os artefatos deste tipo, além de um item opcional, $p \in M \mid p \neq m$, representando um tipo de artefato hierarquicamente superior. Tal relação de superioridade se dá de forma que um artefato do tipo m, em caso de indisponibilidade, possa ser substituído por um do tipo p.

Definição 4-3. Um *requisito funcional* representa uma condição ou característica

com a qual um sistema precisa estar em conformidade. O conjunto P, de todos os requisitos funcionais é definido da seguinte forma:

$$P=\{\rho \mid \rho=\langle I, N, N_T, N_C, C, M_R\rangle\}$$

Cada um dos componentes do requisito ρ é declarativo e discreto. $I \in$ { *Obrigatório, Altamente Desejável, Desejável, Opcional* } representa o nível de importância daquele requisito. N, N_T, N_C, $C \in$ { *verdadeiro, falso* } representam, respectivamente, se o requisito se refere a uma característica nova (em contraste à melhoria de uma já existente), se é novo para a equipe de desenvolvimento, se é novo para o cliente e se está sujeito a mudanças durante a execução do processo. O elemento $M_R \subseteq M$ enumera os modelos de artefato relacionados ao requisito.

Definição 4-4. O *fator de risco* associado a um determinado requisito funcional é uma representação do conceito de risco, definido por Kruchten (2000) como uma variável que, dentro de sua distribuição normal, pode assumir um valor que diminua ou elimine o sucesso de um projeto. Este valor é modelado para transmitir a ideia da importância de um determinado requisito para o sucesso do projeto. Modelado como um número natural, quanto maior o fator de risco de um requisito, maiores são as chances de falhas relacionadas a este requisito impactarem negativamente o resultado do projeto.

Por razões de simplificação do modelo, o cálculo do fator de risco leva em consideração apenas as características explicitamente associadas a um requisito funcional. Riscos ambientais não são levados em consideração nesta proposta. Algebricamente, o fator de risco σ associado ao requisito funcional ρ é assim definido:

$$\sigma=F_I+F_N+F_{NT}+F_{NC}+F_C$$

O valor $F_I \in [1,4]$ é dado de acordo com o valor I de ρ, associando, respectivamente, os valores *Opcional*, *Desejável*, *Altamente Desejável* e *Obrigatório* a seus correspondentes numéricos; 1, 2, 3 e 4. Os componentes F_N, F_{NT}, F_{NC} e F_C são associados respectivamente aos valores N, N_T, N_C e C de ρ. A cada um dos componentes é atribuído o valor 1, caso o valor correspondente seja *verdadeiro* e 0, se *falso*.

Definição 4-5. Uma *instância de artefato*, ou simplesmente *artefato*, é a unidade básica de medida de progresso em um projeto e é o resultado de um esforço de desenvolvimento, pois é o produto final de uma tarefa de desenvolvimento. Neste trabalho, um artefato é uma instância de um modelo de artefato. O conjunto A, de todos os artefatos criados e consumidos no processo gerado, é definido da seguinte maneira:

$$A=\{\alpha \mid \alpha=\langle o, m, s, P_R\rangle\}$$

Em cada artefato α, $o \in R$ representa o papel do proprietário do artefato e $m \in M$ indica o modelo do qual o artefato é instância. O valor de $s \in S =$ { *Não Existente,*

Iniciado, Atualizado, Revisado, Concluído } é um indicativo do estado do artefato. O elemento $P_R \subseteq P$ identifica a quais requisitos funcionais o artefato está relacionado.

Definição 4-6. Uma *ação* é uma função que leva um determinado conjunto de artefatos de um estado a outro. Uma ação π pode ser expressa da seguinte forma:

$$\begin{array}{llll} \pi: & R \times A^n & \rightarrow & A^m \\ & (r, a_0, a_1, \ldots, a_n) & \rightarrow & (a_0', a_1', \ldots, a_m') \end{array}$$

Onde $n, m \in \mathbb{N}^*$ indicam o número de artefatos de entrada e saída, respectivamente.

Definição 4-7. Uma *atividade do processo* é uma sequência de ações designadas a um determinado papel capaz de produzir um resultado observável. Esta definição é uma adaptação da apresentada por Kruchten (2000). O conjunto Φ das atividades definidas em uma descrição de processo é dado por:

$$\Phi = \{\varphi \mid \varphi = \langle r, A_I, A_O, \Pi, d \rangle\}$$

Em cada atividade φ, $r \in R$ estabelece a qual papel a atividade está atribuída, A_I, $A_O \subseteq M \times S$ indicam, respectivamente, os conjuntos de modelos de artefatos de entrada e saída e o estado de cada um, Π representa o conjunto de ações – também chamadas de passos – e $d \in \mathbb{N}^*$ significa a duração da atividade, em Unidades de Tempo (UT). O uso de uma unidade de tempo específica poderia ser uma restrição muito forte, pois a duração de cada tipo de atividade real de desenvolvimento pode ser mais bem expressa por diferentes unidades. Por esta razão, o tempo foi modelado como um recurso abstrato, e não é utilizada nenhuma unidade de medida de tempo específica. O uso de uma unidade de tempo abstrata permite estabelecer medidas de tempo relativas entre as atividades de desenvolvimento.

Definição 4-8. Um *processo viável* permite a execução das atividades na sequência descrita, de forma que todas as dependências de todas as atividades são atendidas, e que as restrições existentes sejam observadas.

4.3 Arquitetura dos agentes

Há dois tipos básicos de agentes envolvidos: *Gerente de Projetos* e *Especialista*. A sociedade é formada por um gerente de projetos (GP) e vários especialistas. Cada tipo de agente tem suas próprias características, mas há semelhanças e ideias gerais compartilhadas por ambos. Desta forma, existem elementos básicos que constituem uma parte de arquitetura comum a todos os agentes da sociedade. Ambos os tipos de agente possuem elementos específicos que os diferenciam. Por fim, cada agente tem sua própria base de conhecimentos, o que torna cada um único.

Por arquitetura de construção dos agentes entende-se o conjunto de tarefas desempenhadas pelos mesmos, o tipo e o formato das informações armazenadas, bem como os protocolos que definem como e quando ocorre comunicação entre eles.

Há ainda um terceiro tipo de agente, o *Agente de Serviço*, que não compartilha a arquitetura definida para os demais.

4.3.1 Arquitetura comum

Todos os agentes do sistema são definidos com base em uma arquitetura formada por duas camadas de atuação e tomada de decisão. A primeira camada lida com o problema de criar o plano representando o processo de desenvolvimento, ou trechos do mesmo. Uma segunda camada executa simulações simplificadas da execução de atividades de desenvolvimento para suportar a definição do processo. Esta simulação não tem a intenção de substituir qualquer parte da execução real do processo, pois trabalha exclusivamente com informações e conceitos abstratos. Seu único propósito é estabelecer a ordem e a dependência entre as atividades de desenvolvimento, com o intuito de que o agente gere um plano executável. As camadas são interdependentes e sua distinção é feita para simplificação do projeto e clareza da especificação. Não é necessário haver separação de elementos de implementação para estabelecê-las.

As diferenças entre as camadas se dão, basicamente, em três aspectos: tipos de objetivos a serem alcançados; informações trocadas e armazenadas; ações executadas.

A primeira camada, denominada *Camada de Definição do Processo*, é caracterizada pelo foco no produto final do trabalho de cada agente: um plano para representar o processo de desenvolvimento de software. Há apenas um tipo de objetivo a ser atingido pelos agentes nessa camada: criar um plano. Os trechos da base de conhecimentos do agente utilizados nesta camada são: a definição do processo de desenvolvimento, com as possíveis atividades e suas pré-condições; dados do projeto, fornecidos como entrada para o sistema; eventuais invariantes a serem observados durante a execução do processo; resultados das atividades de simulação da execução do processo. As ações executadas pelo agente neste nível são: adicionar atividades ao plano, incluindo pré-condições; enviar lista de atividades para outro agente; estabelecer um objetivo para ser alcançado na segunda camada. A última atividade descrita estabelece a conexão entre as duas camadas do agente. Para compor seu plano, o agente faz uma simulação simplificada da execução do processo de desenvolvimento. Esta simulação é feita com base nos objetivos estabelecidos pela camada de definição do processo.

A segunda camada da arquitetura do agente, ou *Camada de Simulação do Processo*, é responsável por simular a execução de atividades de desenvolvimento de software, com base na descrição do modelo de processo. O progresso do processo de desenvolvimento é medido pelos artefatos, conforme definição 4-5. Desta forma, os tipos de objetivos a serem atingidos nesta camada são: criar artefatos; atualizar artefatos. Para tal, as informações acessadas pelo agente, e trocadas com outros membros da sociedade, são: definições de modelos de artefatos; descrições de instâncias de artefatos; dados sobre a disponibilidade de artefatos na sociedade. As ações do agente neste nível são: consultar descrição do modelo de processo, em busca de atividades que o permitam atingir seus objetivos; processar lista de pré-condições de uma atividade; marcar uma atividade como executável; enviar solicitação de artefato; processar solicitação de artefato; enviar resposta a uma solicitação recebida. Quando o agente marcar uma atividade como executável, a camada de definição do processo se encarregará de adicioná-la ao plano sendo gerado.

A tabela 1 mostra um sumário das diferenças entre as duas camadas da arquitetura dos agentes.

Tabela 1. Camadas da Arquitetura

	Camada de Definição do Processo	Camada de Simulação do Processo
Tipos de objetivos	• Criar plano	• Criar artefato • Atualizar artefato
Informações acessadas	• Modelo de processo • Dados do projeto • Invariantes de execução	• Tipos de artefatos • Instâncias de artefatos • Disponibilidade de artefatos
Tipos de ações	• Adicionar atividade ao plano • Enviar plano • Definir objetivo para Camada de Simulação do Processo	• Buscar atividades no modelos de processo • Processar lista de pré-condições de atividades • Marcar atividade como executável • Enviar solicitação de artefato • Processar solicitação de artefato • Responder solicitação de artefato

Esta arquitetura é a base para definição de todos os agentes. O restante da seção detalha as especificidades de cada tipo de agente.

4.3.2 Gerente de projetos

O gerente de projetos é um tipo especial de agente, pois, além de haver apenas um representante deste tipo no sistema, é o responsável pela mediação de toda a

comunicação feita entre os integrantes da sociedade. Este tipo de agente é o responsável direto por executar a tarefa de geração do processo de desenvolvimento. Assim, os outros agentes da sociedade iniciam seus trabalhos mediante solicitação do gerente de projetos. As duas camadas da arquitetura básica são especializadas para que o agente possa desempenhar suas funções.

A camada de definição do processo possui grande importância, pois trata da divisão das tarefas entre os demais agentes, bem como da organização dos planos gerados por cada um deles para a definição do plano final, representando o processo de desenvolvimento completo gerado pelo sistema. Não há diferença entre a definição desta camada e àquela apresentada na arquitetura comum.

A camada de simulação do processo deste tipo de agente, por sua vez, é diferenciada da arquitetura padrão, por tratar exclusivamente de fazer a mediação entre os demais agentes. Por não estar diretamente envolvido com a simulação das atividades de desenvolvimento, este agente acessa apenas informações sobre disponibilidade de artefatos na sociedade e dados dos agentes que a compõem, de forma a ser possível transmitir corretamente as mensagens trocadas entre os especialistas.

4.3.3 Especialista

Os agentes especialistas possuem este nome por representarem indivíduos especializados em alguma área do desenvolvimento de software. Portanto, alguns aspectos específicos da arquitetura comum são ligeiramente modificados para que agentes deste tipo atinjam seu objetivo: construir um plano que representa um trecho do processo de desenvolvimento.

Como cada agente não tem acesso a todas as informações da sociedade, a camada de definição do processo é especializada de forma que o objetivo a ser cumprido será o de criar um trecho do processo.

O fato de o agente ter acesso a apenas uma parte do processo de desenvolvimento sendo gerado limita as possibilidades de simular a execução das atividades de desenvolvimento. É por esta razão que a camada de simulação do processo trabalha com informações abstratas e simplificadas. A simulação é baseada na alteração de estado dos artefatos envolvidos. Quando o agente não tem acesso a um artefato necessário para a execução de uma determinada atividade, ele envia uma solicitação de artefato para o gerente de projetos, que se encarregará de encaminhar a mensagem a um outro agente capaz de fornecer o referido artefato, quando possível. O agente solicitante então se baseia apenas na resposta recebida acerca da disponibilidade do artefato para definir ou

não uma atividade como executável e continuar sua simulação, até que os objetivos sejam atingidos.

4.3.4 Agente de serviço

Os agentes de serviço fornecem informações de um domínio específico aos outros agentes do sistema quando solicitado. Seu principal papel é fazer a interface entre a sociedade de agentes e uma base de conhecimentos externa, alimentada tanto por um banco de informações predefinido quanto pela interação com um ser humano ou outro sistema.

Nesta proposta, há apenas dois agentes de serviço definidos, mas instâncias específicas do sistema podem definir outros agentes, para fornecer informações que agreguem valor ao processo final gerado.

O agente responsável por fornecer informações ligadas ao tempo de execução das atividades do processo é o *Provedor de Tempo*. Este agente detém as informações referentes à duração máxima do processo de cada papel, bem como a duração de cada tarefa.

Estabelecer a relação entre artefatos e requisitos funcionais é o papel do agente *Vinculador de Artefatos*. Ele detém informações referentes a quais artefatos estão ligados a quais requisitos funcionais.

4.4 Processamento e gerenciamento de informações

Todos os agentes têm um estado interno definido pelas informações disponíveis em sua base de conhecimentos em um dado instante, sejam de origem externa, recebidas de outros integrantes da sociedade, ou oriundas de processamento interno ou inferência realizada pelo próprio agente. Alguns destes dados são definidos durante o projeto do agente e são invariantes, ou seja, não sofrem mudanças ao longo do tempo. Entretanto, há outras informações dinâmicas, as quais devem ser gerenciadas pelo agente durante a execução do sistema em algum contexto específico.

As informações estáticas na base de conhecimentos de cada especialista representam os elementos fundamentais de processos de desenvolvimento de software: atividades, modelos de artefatos e ações.

A parte variável da base de conhecimentos dos especialistas é, basicamente, composta de conjuntos de artefatos e uma lista de atividades. O agente precisa ter controle sobre tais elementos para manter sua integridade e permitir a si mesmo chegar a uma solução para seu problema de planejamento local.

Há cinco conjuntos de artefatos gerenciados por cada agente especialista sempre que houver alteração na base de conhecimentos: artefatos a entregar (A_E), artefatos sob controle (A_C), artefatos necessários (A_N), artefatos aguardados (A_A) e artefatos indisponíveis (A_I). A_E são aqueles que devem ser gerados pelo especialista em questão e, por esta razão, podem ser associados diretamente aos objetivos do agente. A_C contém todos os artefatos dos quais o agente tem conhecimento, incluindo tanto os gerados por ele mesmo quanto os disponibilizados pelos outros. A_N é o conjunto de artefatos dos quais o agente necessita, mas é incapaz de produzir, sendo necessária interação com o restante da sociedade. A_A pode ser considerado um conjunto intermediário, pois contém os artefatos dos quais o agente necessita e não é capaz de criar, mas para os quais já foi feita uma solicitação, que aguarda retorno. O último conjunto, A_I, contém os artefatos requisitados para a sociedade, mas não podem ser gerados por nenhum agente. A Figura 4-2 mostra propriedades dos conjuntos de artefatos mantidas em todos os estados da base de conhecimentos.

$$I.\ A_E \subseteq A_C$$
$$II.\ A_N \cap A_A \cap A_I = \varnothing$$
$$III.\ A_C \cap (A_N \cup A_A \cup A_I) = \varnothing$$

Figura 4-2*. Propriedades mantidas pelos conjuntos de artefatos da base de conhecimentos de um especialista*

O especialista também precisa ter controle sobre todas as atividades já escolhidas para compor sua definição local do processo de desenvolvimento, bem como a ordem relativa entre as mesmas. Quando o agente termina seu trabalho, esta lista representa o resultado final a ser enviado ao gerente de projetos.

O Gerente de Projetos (*GP*) é o agente responsável por controlar todo o processo de comunicação e, portanto, deve gerenciar toda informação referente à geração do processo de desenvolvimento trocada entre os agentes especialistas, além de manter atualizadas as informações referentes ao seu produto final: a definição do processo de desenvolvimento de software. De maneira análoga ao que acontece nas bases de conhecimentos dos especialistas, as informações gerenciadas pelo *GP* são compostas de conjuntos de artefatos e listas de atividades de processo.

Quatro conjuntos de artefatos são gerenciados pelo agente: artefatos objetivo (A_O), artefatos correntes (A_C), artefatos aguardados (A_A) e artefatos indisponíveis (A_I). A_O é composto daqueles artefatos que precisam ser gerados e, consequentemente, devem ser ser disponibilizados por um dos especialistas. Os membros de A_C

representam todos os artefatos já disponibilizados por quaisquer dos especialistas. A_A inclui todos os artefatos solicitados por algum especialista, para os quais uma resposta de algum outro agente é esperada. A_I contém os artefatos que não podem ser criados por nenhum dos especialistas. A Figura 4-3 apresenta as propriedades mantidas pelos conjuntos de artefatos em todos os estados da base de conhecimentos do Gerente de Projetos.

$$IV.\ A_C \cap A_O \cap A_I = \emptyset$$
$$V.\ \ A_A \cap A_I = \emptyset$$

***Figura 4-3**. Propriedades mantidas pelos conjuntos de artefatos da base de conhecimentos do Gerente de Projetos*

Outra parte da base de conhecimentos do *GP* é composta de listas de atividades. O agente deve controlar os resultados parciais enviados pelos especialistas. Como cada um desses resultados é uma lista de atividades, o Gerente de Projetos armazena uma lista para cada agente da sociedade. Quando todos os resultados são recebidos pelo *GP*, ele constrói sua própria lista de atividades, que representa o resultado final do trabalho a ser entregue para o usuário, ou seja, o processo de desenvolvimento.

Para assegurar a validade de todas as propriedades listadas nas figuras 4-2 e 4-3, os agentes devem seguir algumas regras para gerenciar os elementos em todos os conjuntos por eles mantidos. A maioria das regras está diretamente relacionada ao processo de comunicação, descrito na subseção 4.5. A seguir são listadas as regras seguidas pelos agentes especialistas:

1. Quando um agente tem como objetivo a construção de um artefato, este é adicionado ao conjunto A_E, caso ainda não esteja em nenhum dos cinco monitorados pelo agente;
2. Quando um artefato está no conjunto A_E e o agente é capaz de gerá-lo, ele é removido de A_E e adicionado a A_C;
3. Quando um artefato está em A_E e o agente não é capaz de construí-lo, ele é removido de A_E e inserido em A_N;
4. Quando um agente solicita um artefato do qual necessita, ele é removido de A_N e inserido em A_A;
5. Quando o agente recebe uma resposta positiva para uma solicitação de artefato prévia, o artefato é removido de A_A e adicionado a A_C;
6. Quando uma resposta negativa a uma solicitação de artefato previamente feita é recebida, o artefato é removido de A_A e inserido em A_I.

O Gerente de Projetos segue um conjunto próprio de regras, que são listadas a seguir:

1. Quando o *GP* identifica um artefato como um objetivo a ser atingido pelo processo parcial gerado por algum especialista, o artefato é adicionado a A_O;
2. Quando um agente solicita um artefato e este não está em A_C ou A_I, ele é adicionado a A_A;
3. Quando uma resposta positiva a uma solicitação de artefato é dada por algum agente, o artefato a ela associado é removido de A_A e inserido em A_C;
4. Quando respostas negativas a uma solicitação prévia de artefato é dada por todos os agentes, o artefato envolvido é removido de A_A e adicionado a A_I;
5. Quando um agente torna um artefato disponível, o mesmo é removido de A_O e adicionado a A_C.

4.5 Comunicação

A troca de mensagens entre os agentes da sociedade é constante e desempenha papel fundamental na execução do sistema. Como cada agente possui apenas uma parte de todo o conhecimento disponível, a agência precisa trabalhar em equipe e trocar informações disponíveis, negociar e colaborar uns com os outros para que possam chegar a uma solução conjunta para o problema comum de gerar o processo de desenvolvimento de software. Para atingir entendimento entre todos os envolvidos, alguns protocolos de comunicação devem ser seguidos e os próximos parágrafos servem ao propósito de descrevê-los.

Uma visão geral do processo de comunicação é apresentada a seguir:

1. Cada especialista se apresenta ao Gerente de Projetos (*GP*)
2. O GP envia todas as informações disponíveis sobre o projeto:
   a) Requisitos funcionais
   b) Modelos de artefato disponíveis
   c) Estado atual dos artefatos conhecidos
3. O GP envia uma mensagem ordenando o início do trabalho aos especialistas e inicia a espera pelos resultados individuais
4. Os agentes comunicam-se entre si, seguindo os protocolos descritos a seguir, adequados a cada situação, tendo o Gerente de Projetos como intermediador do processo comunicativo
5. Cada especialista envia seus resultados individuais para o GP
6. De posse de todas as partes do resultado geradas pelos especialistas, o

Gerente de Projetos as processa e envia mensagem de retorno – positivo ou negativo – para cada especialista

Para garantir que o processo de comunicação delineado seja rigorosamente seguido, há protocolos específicos, incluindo tanto elementos de comunicação quanto de processamento de informações. O restante desta subseção detalha cada um dos seis protocolos definidos.

4.5.1 Protocolo de inscrição

Este protocolo é seguido por todos os especialistas quando o sistema é iniciado. O principal objetivo é tornar o Gerente de Projetos ciente da existência dos agentes especialistas e dos papéis desempenhados por eles. O diagrama de sequência apresentado na figura 4-4 descreve o protocolo.

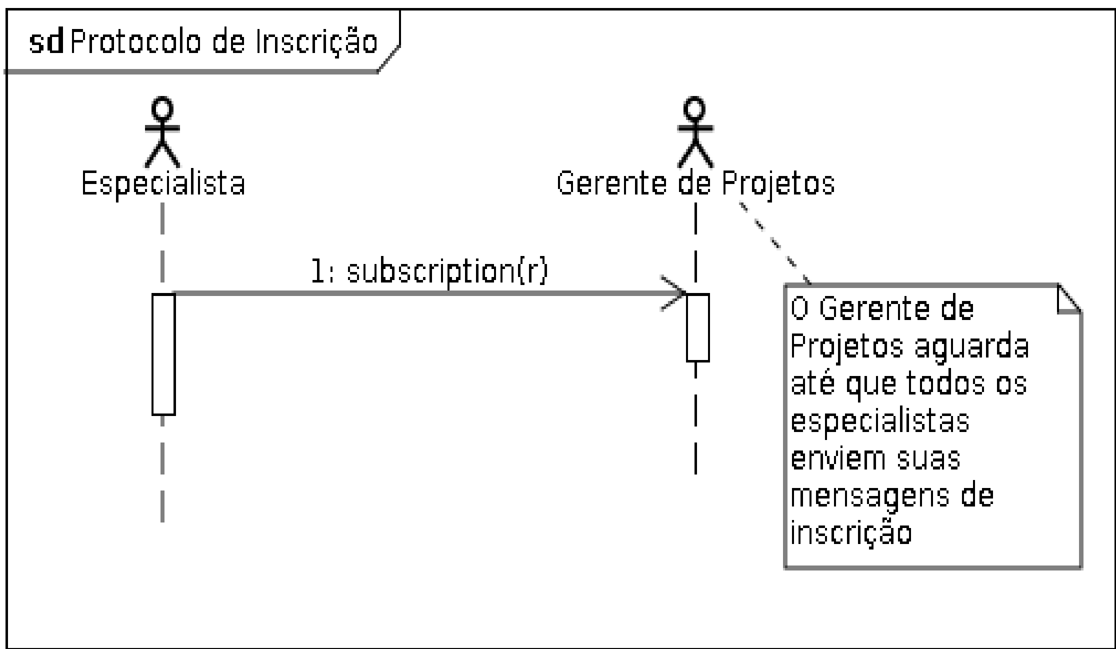

***Figura 4-4**. Protocolo de inscrição*

O especialista envia para o GP todos os papéis que representa. Uma mensagem de confirmação de inscrição é enviada de volta.

4.5.2 Protocolo de iniciação

Todos os agentes seguem este protocolo a fim de obterem todas as informações necessárias para o início de suas atividades. Após a inscrição de todos os especialistas, o Gerente de Projetos envia a todos as informações disponíveis acerca do projeto. Ao fim, um sinal é emitido para que todos os especialistas iniciem seus trabalhos. A figura 4-5 ilustra esta situação.

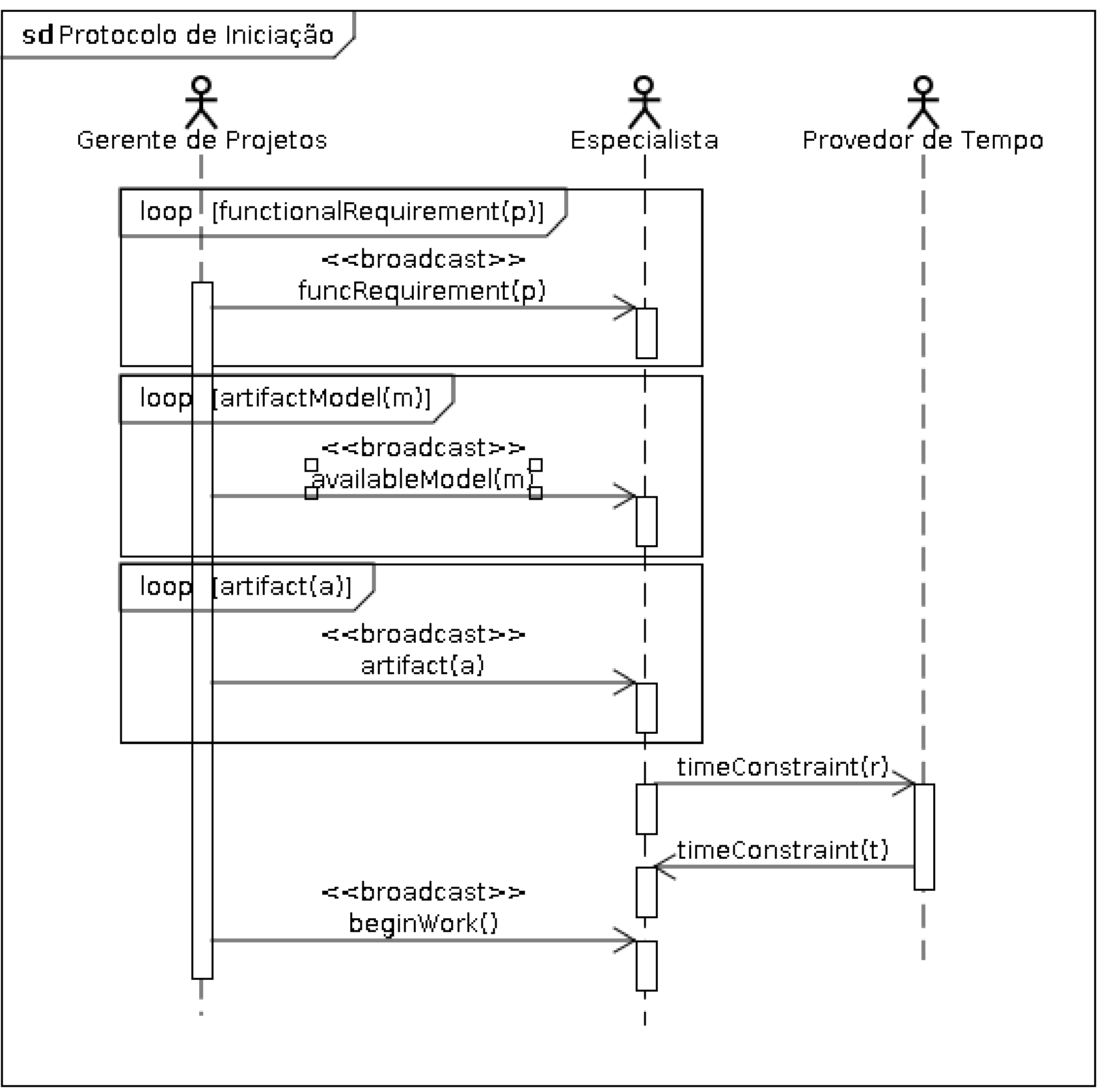

***Figura 4-5**. Protocolo de iniciação*

4.5.3 Protocolo de solicitação de artefato

Durante a execução do sistema, é comum ocorrer a situação na qual um agente necessita de um artefato que seja pré-requisito para uma tarefa prestes a ser adicionada ao processo. Se o especialista não tiver ciência ou não for capaz de produzir o artefato, é necessário solicitá-lo a outro agente. O protocolo representado na figura 4-6 é seguido nestes casos.

Os agentes envolvidos – solicitante e provedor do artefato – se comunicam de forma indireta, tendo o Gerente de Projetos como intermediário, pois todas as mensagens são a ele enviadas e dele recebidas. O solicitante envia uma mensagem ao *GP* e aguarda uma resposta. Este então verificará entre todos os outros agentes se o referido artefato é de ciência de algum deles. Se qualquer especialista responder afirmativamente, o *GP* atualiza sua base de conhecimentos interna e dá o retorno

positivo ao solicitante. Quando o artefato não está disponível para nenhum agente, o *GP* consulta a todos, para tentar encontrar algum especialista capaz de gerá-lo. Em ambos os casos, existindo ou não um agente capaz de produzir o artefato, o Gerente de Projetos atualiza sua base de conhecimentos e envia a resposta adequada ao solicitante.

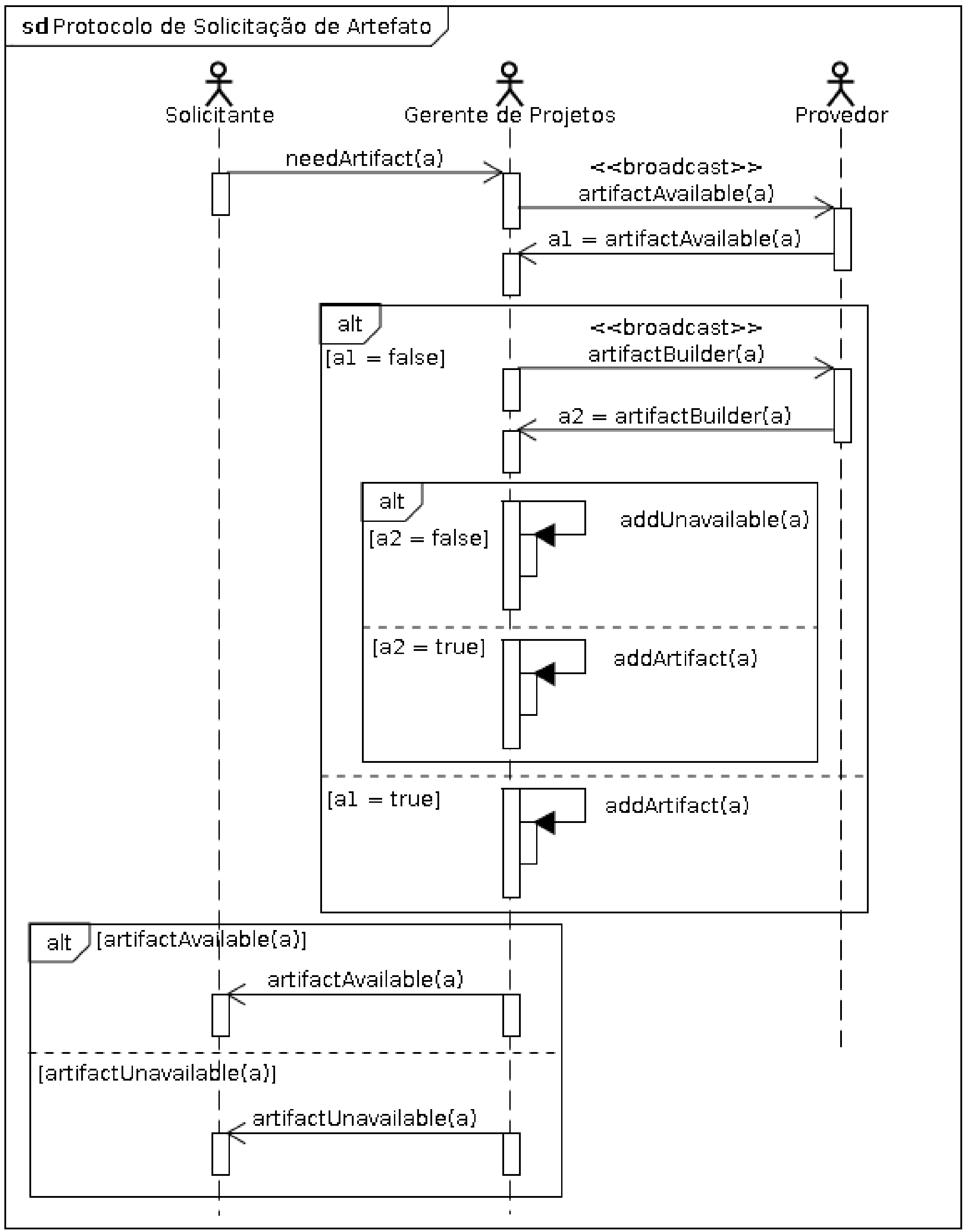

Figura 4-6. *Protocolo de solicitação de artefato*

4.5.4 Protocolo de entrega de resultado

Ao término de suas atividades, um especialista precisa enviar o resultado de seu

trabalho para o Gerente de Projeto. O protocolo ilustrado pela figura 4-7 é seguido nesta ocasião.

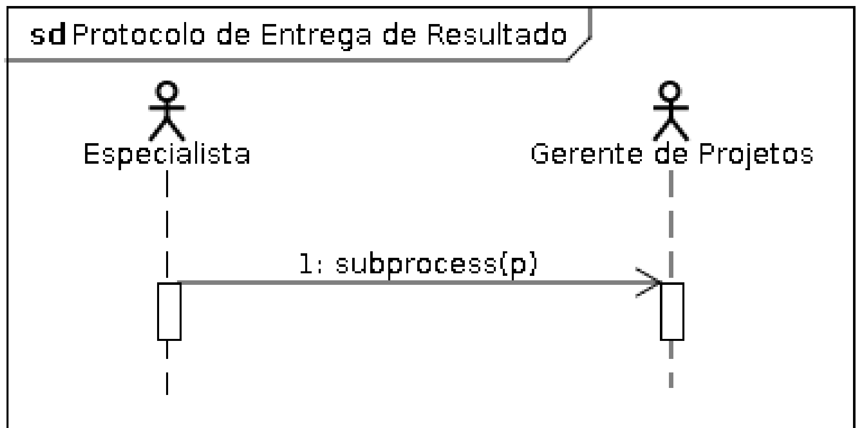

Figura 4-7*. Protocolo de entrega de resultado*

A mensagem enviada, na qual trafega o resultado do processamento, deve conter as atividades do processo, bem como os pré-requisitos de cada uma delas. Esta informação será usada pelo Gerente de Projetos no momento da ordenação das atividades de todos os especialistas para gerar um processo único e coerente.

4.5.5 Protocolo de finalização

O término do trabalho feito pelo Gerente de Projetos, de geração do processo de desenvolvimento de software a partir dos resultados individuais de cada especialista, marca o fim do processamento de todo o sistema. Este fato, assim como o resultado – positivo ou negativo – do processamento, deve ser comunicado a todos os agentes da sociedade, sinalizando o fim da execução de todo o sistema. Este comportamento pode ser visto na figura 4-8.

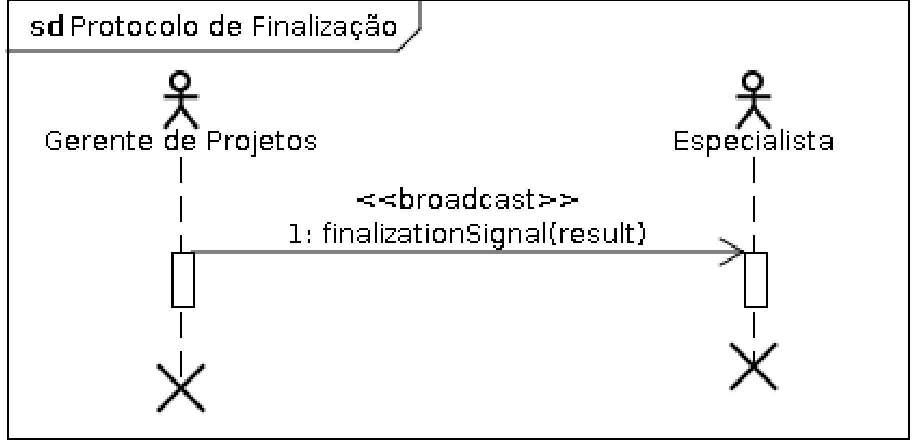

Figura 4-8*. Protocolo de finalização*

4.5.6 Protocolo de solicitação de informação

Durante a execução de suas atividades, o gerente de projeto e os especialistas podem necessitar de informações das quais não dispõem e não podem ser fornecidas por outros agentes, exceto pelos de serviço. Quando uma situação desta natureza ocorre, o agente faz uma solicitação ao agente de serviço e aguarda sua resposta. Para enviar a resposta, pode ser necessário ao agente de serviço realizar uma consulta a um sistema externo, pois sua base de conhecimentos pode estar armazenada em outro local ou pode ser alimentada interativamente, a cada solicitação, por um ser humano. A figura 4-9 ilustra este cenário de comunicação.

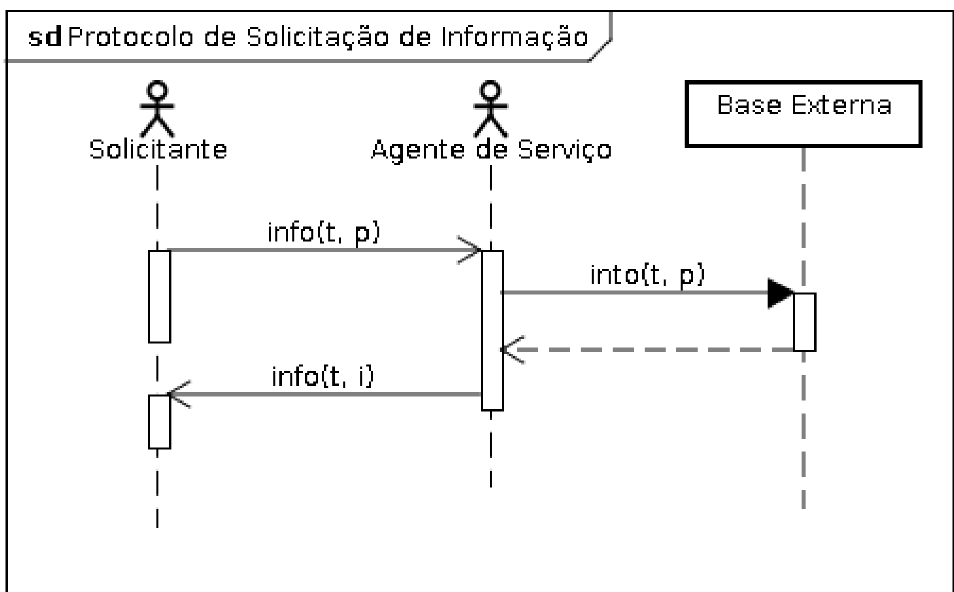

Figura 4-9. *Protocolo de solicitação de informação*

4.6 Comportamento

Apesar de trabalharem com o mesmo tipo de problema, agentes especialistas e o gerente de projetos se comportam de maneira diferente, atuando em diferentes aspectos da tarefa de estabelecer um plano abstrato para o desenvolvimento de software. Mesmo assim, por compartilharem uma parte da arquitetura e trabalharem de forma colaborativa, há semelhanças entre o comportamento de ambos os tipos de agente. O final da execução do Protocolo de Iniciação marca o início do processamento individual de cada agente. De forma geral, o comportamento dos agentes durante esta fase pode ser descrito pelo diagrama da figura 4-10.

Cada uma das atividades dos agentes é especializada, de acordo com o tipo do mesmo. O restante desta seção tem o intuito de detalhá-las.

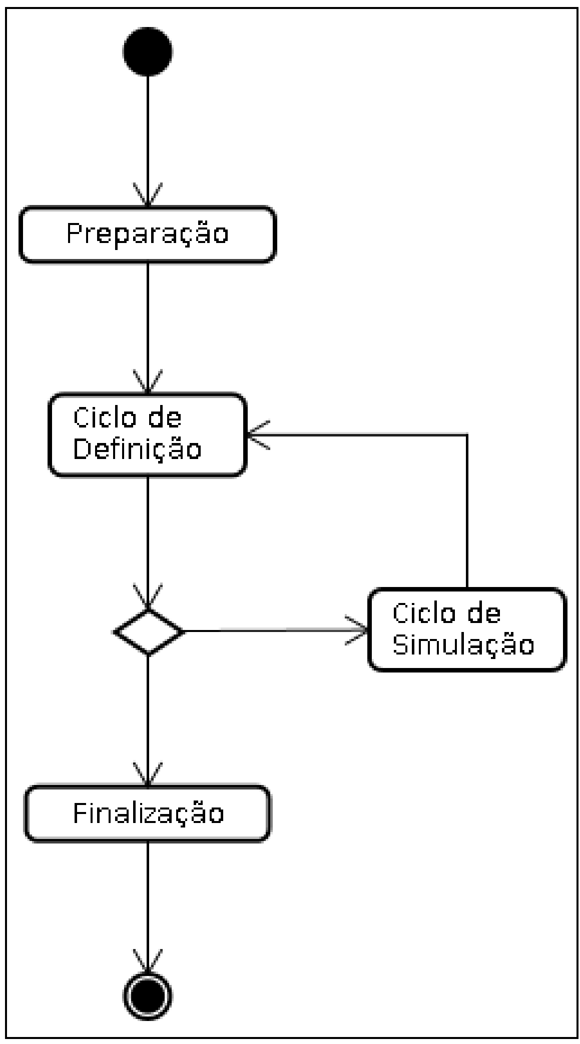

Figura 4-10. *Visão geral do comportamento do processamento individual dos agentes*

4.6.1 Preparação

Nesta etapa do processamento, os agentes organizam sua base de conhecimentos para iniciar o trabalho de planejamento.

O gerente de projetos registra em sua base de conhecimentos o fato de estar aguardando um subprocesso para cada agente inscrito. Este registro é importante para permitir a ele iniciar a geração do plano final apenas após receber todos os planos individuais. Nesta etapa, o agente adiciona cada um dos planos individuais recebidos à sua base e inicia a próxima após o recebimento de todos.

Os agentes especialistas fazem o cálculo do fator de risco para cada requisito funcional e ordenam a lista de requisitos conforme seus fatores de risco. Os requisitos com fator de risco mais alto são os primeiros da lista, pois são considerados os mais importantes do projeto e devem ser tratados o mais cedo possível.

Os especialistas consultam o agente Provedor de Tempo para obterem as

restrições de tempo relacionadas ao processo de cada papel desempenhado por cada um dos agentes e adicionam esta informação à sua base de conhecimentos.

4.6.2 Ciclo de definição

Nesta etapa do processamento, os agentes exercem funções da Camada de Definição do Processo, lidando diretamente com o plano sendo gerado.

```
procedure DefinitionCycle(var Goals: Array of Goal; CurrentPlan: Plan;
                          MarkedTasks: Array of Task; KB: KnowledgeBase)
var
  t: Task; g, p: Goal;

begin
  if Length(Goals) > 0 then
  begin
    g = NextGoal(Goals);

    if not ArtifactModelAvailable(KB, g) then
      if HasParentModel(g) then
      begin
        p := ParentModel(g);
        RemoveGoal(Goals, g);
        AppendGoal(Goals, p);
        exit;
      end;

    if ArtifactAvailable(KB, g) then
      RemoveGoal(Goals, g)
    else
      Build(g);

    SimulationCycle(Goals, GetFuncReqs(KB), KB);
  end
  else
    WorkFinished();

  while Length(MarkedTasks) > 0 do
  begin
    t := NextTask(MarkedTasks);
    if not Duration(t) then
      GetDuration(t);

    if PlanDuration(CurrentPlan) + Duration(t) <= GetTimeLimit(KB) then
      AppendElement(CurrentPlan, t);

    RemoveTask(MarkedTasks);
  end;
end.
```

Figura 4-11. *Ciclo de Definição dos agentes especialistas*

Cada especialista atua conforme o algoritmo descrito na figura 4-11. O agente processa a lista de objetivos. Se houver algum objetivo de geração de artefato não atingido, é adotado o objetivo de simular a geração do mesmo. Caso um artefato esteja

indisponível no sistema, o agente consulta sua base de conhecimentos em busca de um artefato hierarquicamente superior. Caso encontre algum, o objetivo original de gerar o artefato faltante será trocado pelo de gerar o substituto. Se não houver mais objetivos de geração de artefato a serem atingidos, o agente sinaliza a finalização do trabalho de geração de seu plano.

Em cada ciclo, o agente especialista também processa as atividades a serem inseridas no planejamento. Estas atividades são marcadas para serem acrescentadas ao plano em um Ciclo de Simulação. O agente então escolhe uma atividade para adicionar ao plano e, se a informação referente à sua duração não estiver presente na base de conhecimentos, uma consulta ao agente Provedor de Tempo é feita para obtê-la. A adição desta tarefa ao plano pode fazer com que a duração total do mesmo, dada pela soma das durações de todas as tarefas atualmente presentes nele, ultrapasse o limite de tempo estipulado. Neste caso, o agente não adiciona esta tarefa ao plano e marca-a, para que não seja processada novamente em outro Ciclo de Definição. Se tal marcação não for feita, o agente pode entrar em um laço infinito, caso a atividade escolhida para adição em cada execução do ciclo seja uma que não possa ser adicionada ao plano. Se a atividade selecionada puder ser incluída, considerando sua duração, o agente então acrescenta seus pré-requisitos e a própria atividade ao final do plano, eliminando-a em seguida da lista de candidatas à inserção no processo. As dependências de cada atividade são inseridas no plano, pois o trabalho na camada de definição do processo é o de compor uma lista ordenada das atividades de desenvolvimento, levando em consideração os possíveis pré-requisitos de cada uma, não permitindo que uma atividade dependente de um determinado artefato seja inserida na lista sem que o mesmo seja de conhecimento do agente ou meios para sua obtenção estejam disponíveis.

O gerente de projetos atua conforme o algoritmo expresso na figura 4-12. O agente processa todos os planos individuais e os unifica, formando o plano final, que representa o processo de desenvolvimento gerado. Para mesclar os processos armazenados, o agente analisa o primeiro item do primeiro plano em sua base e verifica se é uma dependência ou uma tarefa. Se for uma tarefa, adiciona-a ao processo final e remove-a do processo original. Se for uma dependência, o agente busca em sua base de conhecimentos pelo pré-requisito. Se não encontrá-lo, busca no processo se há alguma atividade que gere o artefato em questão. Se a dependência for satisfeita, ela é removida do processo original e o agente analisa o próximo item da lista. Se o pré-requisito não estiver satisfeito, o gerente de projetos analisa o primeiro elemento do próximo plano individual, reiniciando o ciclo de verificações, até que todos os elementos sejam

analisados e os processos individuais estejam sem elementos.

```
procedure DefinitionCycle(var Subplans: Array of Plan; CurrentPlan: Plan; KB: KnowledgeBase)
var
  p: Plan; pe: PlanElement;

begin
  if Length(Subplans) > 0 then
  begin
    p = NextSubplan(Subplans);

    pe = NextPlanItem(p);
    if IsTask(pe) then
    begin
      AppendElement(CurrentPlan, pe);
    end
    else
      if not SearchPreReq(KB, pe) then
        if not SearchTaskOutput(p, pe) then
          exit;

    RemoveElement(p, pe);

    if Length(p) = 0 then
    begin
      RemovePlan(Subplans, p);
    end;
  end;
end.
```

***Figura 4-12**. Ciclo de Definição do gerente de projetos*

4.6.3 Ciclo de simulação

Nesta fase, os agentes executam ações da Camada de Simulação do Processo. O intuito dessas ações é permitir selecionar as atividades que levam à criação dos artefatos que devem ser gerados pelo processo. A simulação realizada é simplificada, baseada nos modelos definidos nesta proposta, não sendo, necessariamente, fiel a nenhum *framework* real de desenvolvimento.

O gerente de projetos não executa nenhuma ação durante seu Ciclo de Simulação senão a intermediação da comunicação entre os demais agentes.

Os especialistas, por sua vez, executam o algoritmo apresentado na figura 4-13. Eles analisam sua base de objetivos item a item, em busca de um artefato relacionado ao requisito de maior prioridade, segundo a lista ordenada na fase de Preparação. Se não houver objetivos relacionados ao requisito mais prioritário, ele faz uma busca pelo segundo com maior prioridade, e assim sucessivamente, até encontrar um objetivo para adotar. A partir daí, os agentes procuram encontrar quais, dentre as tarefas de desenvolvimento sob sua responsabilidade, levam à criação do artefato. Quando uma

atividade é encontrada, o agente verifica se suas entradas já estão disponíveis em sua base, tanto por meio de outras atividades que geram tais saídas já presentes no processo quanto pela comunicação com outros agentes do sistema, capazes de produzir tais entradas. Se houver entradas não satisfeitas, o agente adiciona tantos objetivos de criação de artefato quantas forem as entradas inexistentes. Estes objetivos são adicionados com prioridade máxima à base do agente. Quando todas as entradas de uma atividade estão presentes na base de conhecimentos do agente, ele marca a atividade para inclusão no plano, juntamente com seus pré-requisitos, e processa a saída da atividade, adicionando os artefatos gerados à sua base de conhecimentos.

```
procedure SimulationStep(Goals: Array of Goal, FuncReqs: Array of FuncReq, KB:
KnowledgeBase)
var
  i, j: integer; g: Goal; fr: FuncReq; t: Task;
  inputs: Array of Artifact; goalAchieved: boolean;

begin
  fr := NextHighestPriorityReq(FuncReqs);
  for i := 1 to Length(Goals) do
  begin
    g := Goals[i];
    if IsRelatedGoal(fr, g) then
    begin
      if HasRelatedTask(KB, g) then
      begin
        t := RelatedTask(g);
        inputs := TaskInputs(t);
        goalAchieved := true;
        for j := 1 to Length(inputs) do
        begin
          if not AvailableArtifact(inputs[j]) then
          begin;
            AppendGoal(Goals, inputs[j]);
            goalAchieved := false;
          end;
        end;

        if goalAchieved then
        begin
          MarkTask(t);
          exit;
        end;
      end;
    end;
  end;
end.
```

Figura 4-13*. Ciclo de Simulação dos agentes especialistas*

4.6.4 Finalização

Esta é a etapa na qual os agentes eliminam qualquer informação não mais necessária de

sua base de conhecimentos, preparando-se para o término da execução do sistema.

Os especialistas enviam o processo gerado para o gerente de projetos e aguardam o recebimento do sinal de finalização.

O gerente de projetos elimina todos os planos individuais de sua base de conhecimentos, permanecendo apenas o processo final gerado. O agente então envia o sinal de finalização aos especialistas.

5 IMPLEMENTAÇÃO E ESTUDO DE CASO

"I like engineering, but I love the creative input"
John Dykstra

A fim de validar a aplicabilidade da proposta de método para automatizar parte da tarefa de customização de processos de software apresentada no capítulo 4, um sistema multiagente foi desenvolvido para implementá-la. Este capítulo descreve tal implementação, bem como um estudo de caso conduzido com a utilização do sistema desenvolvido.

Este capítulo apresenta os detalhes do sistema implementado, explicitando a relação com os conceitos apresentados na proposta, e apresenta o estudo de caso conduzido com o intuito de avaliar o comportamento da implementação.

5.1 Implementação

O sistema foi implementado em linguagem 2APL, como apresentado na seção 2.3.1. Para permitir trabalhar com todos os conceitos envolvidos no método proposto, foi necessário estabelecer um mapeamento dos mesmos para a linguagem 2APL. Esta seção apresenta os mapeamentos de conceitos, a definição dos tipos de mensagens trocadas em cada protocolo de comunicação, bem como alguns detalhes adicionais e as limitações do sistema.

5.1.1 Mapeamento de conceitos

A fim de ser possível construir uma implementação condizente com a proposta do método, os conceitos foram individualmente mapeados de suas definições algébricas para conjuntos de predicados em 2APL. As definições mapeadas foram aquelas apresentadas na seção 4.2 e serão descritas em detalhes nesta seção, junto às respectivas implementações. A maioria teve uma adaptação direta, a saber: *requisito funcional*, *fator de risco*, *papel*, *modelo de artefato*, *instância de artefato*. As *atividades do processo* foram mapeadas parcialmente e o conceito de *ação* foi indiretamente implementado, já que foi mapeado implicitamente.

A figura 5-1 ilustra o mapeamento entre a definição algébrica do conceito de requisito funcional e sua representação em 2APL.

$$\rho = \langle I, N, N_T, N_C, C, M_R \rangle$$

```
funcRequirement(p).                                (1)
frImportance(p, I).                                (2)
[ frNew(p). ]                                      (3)
[ frNewTeam(p). ]                                  (4)
[ frNewClient(p). ]                                (5)
[ frMutable(p). ]                                  (6)
{ frArtifactModel(p, m). }                         (7)
```

Figura 5-1. *Definição e implementação do conceito de requisito funcional*

Cada um dos componentes da tupla é mapeado em um predicado. Os predicados de (3) a (6) podem ou não estar presentes na definição de um requisito funcional. O predicado é adicionado à base de conhecimentos do agente apenas se o valor dos respectivos componentes na especificação do requisito for verdadeiro. Assim, os predicados representam o requisito funcional *p*, com grau de importância *I*. Os predicados opcionais *frNew(p)*, *frNewTeam(p)*, *frNewClient(p)* e *frMutable(p)*, quando presentes indicam, respectivamente, um requisito novo, novo para a equipe de desenvolvimento, novo para o cliente e mutável durante o desenvolvimento. Cada predicado do tipo *frArtifactModel(p, m)* representa um modelo de artefato associado ao requisito. O conjunto de todos os predicados deste tipo corresponde ao componente M_R de *ρ*.

O conceito de fator de risco, por se tratar de um número natural, foi mapeado como um único predicado, como mostra a figura 5-2.

$$\sigma = F_I + F_N + F_{NT} + F_{NC} + F_C$$

```
frRisk(p, r).                                      (8)
```

Figura 5-2. *Mapeamento do conceito de fator de risco*

O predicado (8) representa o fato de o requisito funcional *p* possui fator de risco *r*, calculado conforme definição *σ*.

O conceito de papel foi mapeado conforme expresso na figura 5-3.

$r = \langle n, T \rangle$

```
role(r).                                       (9)
rName(r, n).                                  (10)
{ rTask(r, t). }                              (11)
```

Figura 5-3. *Mapeamento do conceito de papel*

Os predicados indicam um papel *r*, com nome *n*. O conjunto *T*, de atividades que o papel é habilitado a desempenhar, foi mapeado em um conjunto de predicados com o formato *rTask(r, t)*.

Dois dos conceitos mais utilizados no modelo proposto no capítulo 4 são os de *modelo de artefato* e *artefato*. Por serem informações manipuladas em muitas ocasiões, sua implementação prezou pela facilidade de acesso a determinados dados referentes aos artefatos e modelos. Os mapeamentos dos conceito de modelo de artefato e artefato são apresentados, respectivamente, nas figuras 5-4 e 5-5.

$m = \langle R_U, R_R, p \rangle$

```
artifactModel(m).                             (12)
{ amUpdater(m, ru). }                         (13)
{ amReader(m, rr). }                          (14)
[ amParent(m, p). ]                           (15)
```

Figura 5-4. *Mapeamento do conceito de modelo de artefato*

Os conceitos de conjuntos de papéis com permissões de atualização e leitura das instâncias de artefatos do modelo foram implementados como conjuntos de predicados dos tipos (13) e (14) respectivamente. Um predicado *amUpdater(m, ru)* indica que o papel *ru* tem permissão para atualizar os artefatos do modelo *m*. Já *amReader(m, rr)* indica que os artefatos do modelo *m* podem ser lidos pelo papel *rr*. Como um modelo de artefato pode ou não ter a indicação de um modelo hierarquicamente superior, o predicado que modela este conceito é opcional; *amParent(m, p)*, se presente na base de conhecimentos do agente, indica que o modelo *p* é hierarquicamente superior ao modelo *m*.

A implementação do conceito de artefato modelou cada um dos componentes *o*, *m* e *s* nos predicados de (17) a (19). O conjunto de requisitos relacionados ao artefato foi mapeado no conjunto de predicados do tipo *aRequirement(a, p)*, que indica que o requisito *p* relaciona-se com o artefato *a*.

$$\alpha=\langle o, m, s, P_R \rangle$$

```
artifact(a).                                        (16)
aOwner(a, o).                                       (17)
aModel(a, m).                                       (18)
aStatus(a, s).                                      (19)
{ aRequirement(a, p). }                             (20)
```

Figura 5-5*. Mapeamento do conceito de artefato*

Os conceitos de *atividade do processo* e *ação* não foram mapeados em sua totalidade na implementação realizada neste trabalho, em função das simplificações citadas anteriormente. A figura 5-6 mostra a implementação do conceito de atividade do processo. A definição de ação não foi explicitamente mapeada.

$$\varphi=\langle r, A_I, A_O, S, d \rangle$$

```
task(t).                                            (21)
tPerformer(t, r).                                   (22)
{ tInput(t, i, s). }                                (23)
{ tOutput(t, o, s). }                               (24)
tDuration(t, d).                                    (25)
```

Figura 5-6*. Mapeamento do conceito de atividade do processo*

Os componentes mapeados indicam que a atividade *t* é executada pelo papel *r* e possui duração *d*. Os conjuntos de modelos artefatos de entrada e saída são indicados pelos conjuntos de predicados dos tipos (23) e (24), respectivamente. O conjunto de passos da atividade não foi implementado, pois nesta versão simplificada do modelo assumiu-se que cada atividade possui uma ação implicitamente associada que transforma os artefatos de entrada nos de saída.

5.1.2 Protocolos de comunicação

Todos os protocolos de comunicação entre os agentes foram implementados em 2APL. Por ser uma linguagem totalmente preparada para a construção de sistemas multiagentes, há construtos sintáticos para tratar o envio e a recepção de mensagens. Tais construtos permitem a definição de ontologias e linguagens em cada mensagem trocada. Neste trabalho optou-se por não utilizar essa definição explícita, pois foi feita a pressuposição de que todos os agentes compartilham a linguagem e a ontologia relacionadas ao problema. Nesta seção, serão exibidos os formatos de mensagens trocados entre os agentes em cada um dos protocolos.

No protocolo de inscrição há dois tipos de mensagens envolvidas: as enviadas pelos especialistas informando os papéis a serem inscritos e as mensagens de retorno. A figura 5-7 contém o formato geral das mensagens.

```
inform:subscription(R)                              (1)
inform:subscribed(R, S)                             (2)
```

***Figura 5-7**. Mensagens trocadas no protocolo de inscrição*

Mensagens do formato (1) são enviadas pelos especialistas para o gerente de projetos. Cada instância da mensagem indica que o agente desempenha o papel *R*.

O gerente de projetos envia mensagens do tipo (2), indicando ao agente receptor o status da inscrição do papel *R*. Se $S = 1$, o papel foi inscrito corretamente.

O protocolo de iniciação tem um número maior de mensagens envolvidas, já que as informações sobre o contexto do planejamento a ser realizado pelos especialistas são enviadas neste momento. São trocadas mensagens sobre requisitos funcionais, modelos de artefatos, instâncias de artefatos, mensagens de informação e o sinal de início dos trabalhos. A figura 5-8 ilustra as mensagens trocadas neste protocolo.

```
inform:funcRequirement(X, L, N, T, C, M, R)         (3)
inform:availableModels(M)                           (4)
inform:artifact(A, O, M, S, P)                      (5)
request:beginWork(A)                                (6)
request:info(timeConstraint(R))                     (7)
inform:info(timeConstraint(T))                      (8)
```

***Figura 5-8**. Mensagens trocadas no protocolo de iniciação*

Instâncias das mensagens de (3) a (5) são enviadas pelo gerente de projetos para informar aos especialistas sobre os requisitos funcionais, modelos de artefato e artefatos, respectivamente. Uma mensagem do tipo (3) contém todas as informações relacionadas ao requisito funcional *X*, desde seu nível de importância *L* até o conjunto *R* de modelos de artefatos relacionados. Os parâmetros *N*, *T*, *C* e *M* indicam respectivamente um requisito novo, novo para a equipe, novo para o cliente e mutável durante o projeto. Cada um pode assumir os valores *0* ou *1*, sendo que se o valor for *1*, indicará a presença daquela característica no requisito.

Uma mensagem do tipo (4) transmite a informação de que o conjunto *M* de modelo de artefato está disponível e cada um de seus componentes pode ser usado para geração do processo de desenvolvimento.

Instâncias de mensagens do tipo (5) são enviadas pelo gerente de projetos aos especialistas caso haja algum artefato em algum estado já conhecido antes do início dos trabalhos de definição do processo. Cada mensagem contém informações sobre o artefato *A*: o papel *O*, seu proprietário; o modelo *M* do qual é instância; o status *S* em que se encontra e a lista de requisitos *P* aos quais está relacionado.

O sinal de início dos trabalhos, simbolizado pela mensagem (6), é enviado a todos os especialistas, tão logo o gerente de projetos termine de enviar as informações do projeto a todos os especialistas.

Mensagens dos tipos (7) e (8) são trocadas entre o agente Provedor de Tempo e especialistas, para transmitir informações referentes à duração total dos processos individuais. Mensagens do tipo (7) são enviadas pelos especialistas para obter o prazo total do processo do papel *R*. O agente de serviço responde com mensagens do tipo (8), indicando que o prazo total é *T*.

O protocolo de solicitação de artefato é o que possui o maior número de interações entre agentes e, portanto, maior quantidade de tipos de mensagens trocadas. A figura 5-9 contém o formato das mensagens.

```
request:needArtifact(A, S)                         (9)
request:artifactAvailable(A, S)                   (10)
inform:artifactAvailable(A, S)                    (11)
inform:artifactUnavailable(A, S)                  (12)
request:artifactBuilder(A, S)                     (13)
inform:artifactBuilder(A, S)                      (14)
inform:notArtifactBuilder(A, S)                   (15)
```

Figura 5-9*. Mensagens trocadas no protocolo de solicitação de artefato*

Mensagens do tipo (9) são enviadas pelos agentes especialistas quando necessitam de um artefato do modelo *A* no estado *S*, quando não são capazes de produzi-lo sozinhos. Ao receber uma requisição deste tipo, o gerente de projetos consulta sua base de conhecimentos e, caso não tenha informações sobre o artefato em questão, envia uma mensagem do tipo (10) para todos os demais agentes, para verificar se há um artefato disponível com tais características. Os agentes respondem a consultas sobre disponibilidade com mensagens do tipo (11), para respostas positivas, ou (12), para informar a indisponibilidade do artefato.

Se, após receber as respostas sobre a disponibilidade de artefatos do modelo *A* no estado *S*, o gerente de projetos constatar que todas foram negativas, ele envia uma

solicitação do tipo (13) para os agentes, a fim de averiguar se algum dos agentes é o responsável por construir o artefato. Cada especialista informa ao GP uma mensagem do tipo (14) ou (15), indicando que é o responsável pela construção do artefato ou que não é, respectivamente.

Depois de aguardar as respostas, se o gerente de projetos tiver recebido relatos positivos sobre a disponibilidade do artefato ou de seu construtor, envia uma mensagem do tipo (11) ao agente que gerou a solicitação inicial. Caso contrário, uma informação com formato (12) é enviada.

Quando os agentes especialistas finalizam a geração de seus planos individuais, enviam o processo gerado ao gerente de projetos, de acordo com o protocolo de entrega de resultado. A figura 5-10 ilustra o formato da mensagem enviada com o resultado.

```
inform:subprocess(S)                                    (16)
```

***Figura 5-10**. Mensagem enviada no protocolo de entrega de resultado*

A mensagem de tipo (16) informa ao gerente de projetos o resultado *S* do planejamento do especialista, que inclui as atividades e seus pré-requisitos. No corpo do processo gerado, as atividades e dependências são expressas conforme indica a figura 5-11.

```
task(T)                                                 (17)
dep(D, S)                                               (18)

Exemplo de processo:
    [ dep(a1, s1), dep(a2, s2), task(t1), task(t2),
      dep(a3, s2), task(t3) ]
```

***Figura 5-11**. Representação de tarefas e dependências no processo gerado*

Elementos do tipo (17) indicam a execução da tarefa T, enquanto os elementos do tipo (18) indicam um pré-requisito para a próxima tarefa a ser encontrada. No exemplo apresentado na figura 5-11, a tarefa *t1* possui como pré-requisitos os artefatos *a1* e *a2* nos estados *s1* e *s2,* respectivamente. A tarefa *t2* não possui dependências e a tarefa *t3* só pode ser adicionada ao processo final após satisfação da dependência do artefato *a3* com estado *s2*.

É importante ressaltar que, apesar das dependências, o planejamento impõe ordem às tarefas, de forma que se os pré-requisitos de uma tarefa não puderem ser atendidos,

todo o restante do processo do especialista não poderá ser processado. Entretanto, no processo final, poderá haver outras tarefas entre aquelas existentes no plano individual do especialista, em função do processo de unificação dos planos, feitos pelo gerente de projetos. No exemplo da figura 5-11, as tarefas *t1*, *t2* e *t3* serão executadas nesta ordem, embora não necessariamente consecutivamente, pois no processo final poderá haver outras tarefas, provenientes dos planos dos outros especialistas.

Ao final da unificação dos processos individuais, o gerente de projetos executa o protocolo de finalização, enviando uma mensagem do tipo (19), expressa na figura 5-12, para todos os agentes, indicando o término dos trabalhos com status *S*. Se o processo final tiver sido gerado corretamente, *S* assume valor *1*. Se algum problema ocorrer na unificação dos planos e o processo não puder ser gerado, *S* possui valor *0*.

```
inform:finalizationSignal(S)                        (19)
```

Figura 5-12. *Sinal de finalização*

Durante o trabalho dos agentes, quando é necessário obter informações providas pelos agentes de serviço, o protocolo de solicitação de informação é iniciado. A figura mostra os formatos das mensagens de solicitação e envio de respostas.

```
request:info(T [, P])                               (20)
inform:info(T, I)                                   (21)
```

Figura 5-13. *Mensagens trocadas no protocolo de solicitação de informação*

O agente solicitante envia uma mensagem do tipo (20) com o tipo *T* de informação que deseja e, opcionalmente, com os parâmetros *P*. O agente de serviço, após consulta à sua base de conhecimentos ou a um sistema externo, responde com uma mensagem do tipo (21), indicando o conteúdo *I* da informação solicitada.

Como o método proposto faz uso intenso de comunicação e há uma grande variedade de tipos de mensagens trocadas entre os agentes, a figura 5-14 exemplifica a troca de mensagens entre os agentes do sistema, a fim de ilustrar alguns dos aspectos do processo comunicativo envolvido.

Na figura, os círculos representam agentes e as setas mensagens. Os agentes E_1, E_2, E_3 e E_4 são especialistas, *GP* é o gerente de projetos e *AS* representa um agente de serviço. Os números associados a cada mensagem representam a ordem em que as mensagens são enviadas. A figura ilustra uma situação de solicitação de artefato e de

solicitação de informação. O agente E_1 solicita um artefato ao gerente de projetos (mensagem 1), que por sua vez, envia mensagem a todos os outros avaliando sua disponibilidade (2). Para efeito de simplificação da representação, apenas a resposta positiva, enviada pelo agente E_2 (3), está representada. O gerente envia a informação ao agente solicitante (4), que pede ao agente de serviço uma informação sobre o artefato referente à sua solicitação inicial (5). O agente de serviço processa a solicitação e envia uma resposta com a informação (6).

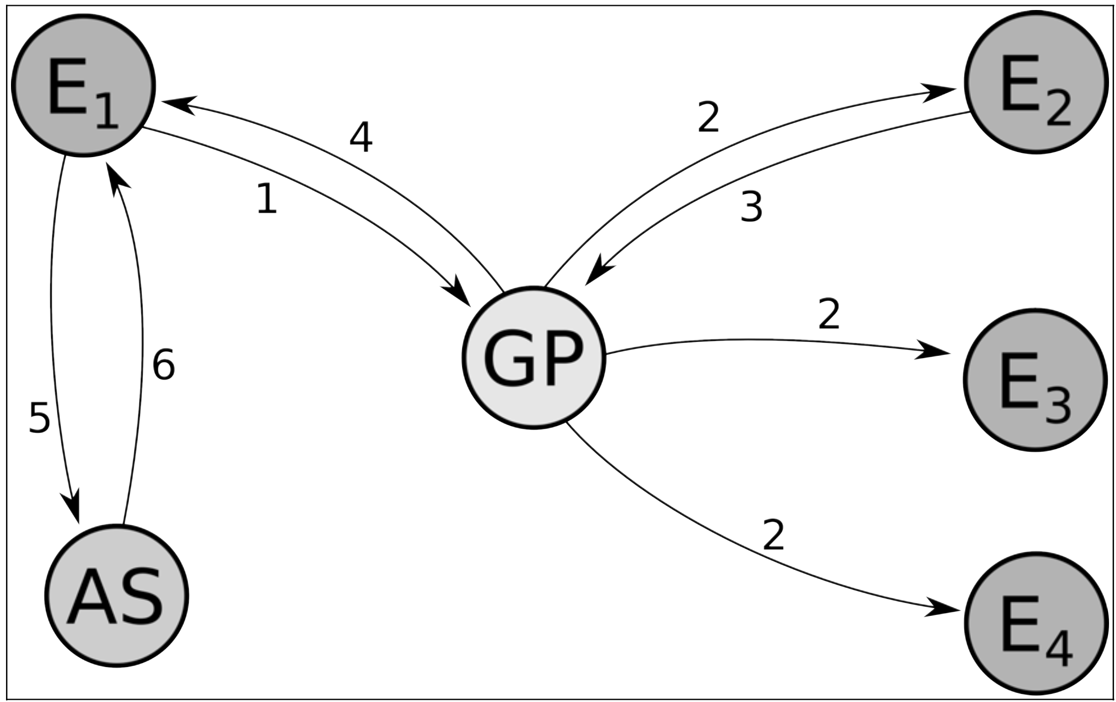

Figura 5-14. *Exemplo de interação entre os agentes e troca de mensagens*

5.1.3 Detalhes adicionais da implementação

A fim de viabilizar a implementação de todos os elementos do método proposto, foi necessário definir alguns elementos adicionais, não diretamente ligados ao método em si. Como a base de conhecimentos dos agentes em 2APL é implementada na forma de um programa Prolog, foram estabelecidas regras para processar algumas variáveis. Parte destes predicados são comumente encontrados em alguns interpretadores Prolog, mas não são usados pelo interpretador da plataforma 2APL. Além disto, para melhor organizar o código do sistema, foi adotada uma estrutura de diretórios, agrupando arquivos de código fonte de conteúdo relacionado. Para permitir um funcionamento eficaz da estrutura de diretórios definida, foi criado uma pequena aplicação em linguagem PHP para processar todo o código e torná-lo compatível com as diretivas de

inclusão de 2APL.

As regras Prolog definidas operam sobre ou criam listas e são utilizadas em diversos pontos da implementação dos agentes. A figura 5-15 mostra os principais predicados definidos.

```
bagof(X, P, L).                                          (1)
concat(L1, L2, R).                                       (2)
equal_lists(L1, L2).                                     (3)
remove_element(X, L, R).                                 (4)
```

Figura 5-15. *Predicados Prolog para manipulação de listas*

O predicado (1) “(...) irá produzir uma lista *L* de todos os objetos *X* que satisfazem ao objetivo *P*” (PALAZZO, 1997). O uso desta função permite tratar como uma lista conjuntos de predicados, como por exemplo os papéis com permissão de atualizar ou acessar artefatos de um modelo.

A função (2) foi definida com o objetivo de fazer concatenação de listas. À variável *R* será atribuída a lista resultado da concatenação de *L1* e *L2*, com os elementos de *L1* adicionados à cabeça de *L2*. Um exemplo do uso de *concat* e do resultado produzido podem ser vistos na figura 5-16.

```
L1 = [ A, B ].
L2 = [ C, D ].
concat(L1, L2, R).
  R = [ A, B, C, D ].
```

Figura 5-16. *Exemplo de uso do predicado concat*

O predicado (3) unifica quando as duas listas *L1* e *L2*, passadas como argumentos, possuem os mesmos elementos, mesmo que em ordem diferente. Já a função (4) é utilizada com o objetivo de remover o elemento *X* da lista *L*, produzindo a lista *R*.

As diretivas de inclusão de 2APL não permitem incluir arquivos localizados em outros diretórios, pois a sintaxe não permite utilizar as barras separadoras. Para contornar esta situação e ser possível organizar o código do sistema multiagente em diretórios, uma pequena aplicação em PHP foi desenvolvida. Seu objetivo é ler toda a estrutura do sistema e criar uma versão do mesmo em um único diretório.

Para tanto, cada arquivo de entrada é lido, processado e salvo com um nome único no diretório de destino. O novo nome do arquivo é formado pela concatenação

dos nomes de seus diretórios pais, separados por uma sublinha (_), e do nome do arquivo. Por exemplo: o arquivo *communication/protocol1/Specialist.2apl* é armazenado no diretório de destino, junto com todos os outros arquivos do sistema, com nome *communication_protocol1_Specialist.2apl*.

Os códigos do sistema multiagente podem ser encontrados no apêndice B.

5.1.4 Limitações

Em função da disponibilidade limitada de tempo e recursos durante a execução deste trabalho, o sistema implementado é limitado em alguns aspectos.

Os elementos que tratam do processamento de requisitos funcionais não foram implementados em sua totalidade, apesar de terem sido mapeados. Toda a estrutura para armazenamento das informações sobre requisitos funcionais na base de conhecimentos foi implementada, mas os trabalhos de priorização de requisitos e vinculação dos mesmos a artefatos não foram implementados. Durante a execução, a presunção é que os requisitos já foram priorizados antes de serem introduzidos na base de conhecimentos dos agentes.

Em face desta limitação, o processo gerado pelo sistema implementado descreve as atividades de desenvolvimento de forma ampla, não havendo instâncias específicas de atividades associadas a cada requisito funcional.

A definição das regras para realizar a priorização de requisitos funcionais e a implementação do agente vinculador de artefatos tendem reduzir ou eliminar esta limitação, tornando o processo gerado mais próximo de um processo de desenvolvimento real, na medida em que haverá instâncias específicas de atividades que abordam determinados requisitos.

Além disto, o sistema implementado, por sua natureza incipiente e experimental, não faz tratamento de erro em diversas situações, especialmente nas de impossibilidade de geração do processo, tanto das partes individuais dos especialistas quanto da unificação feita pelo gerente de projetos. Nestes casos, os agentes podem não conseguir gerar o processo e executarem indefinidamente.

Alguns mecanismos de detecção de potenciais problemas podem ser definidos para evitar que o sistema execute indefinidamente, como limites de tempo de processamento ou número de ciclos de raciocínio sem alterações na base de conhecimentos.

5.2 Estudo de caso

A fim de avaliar o comportamento da implementação feita, um estudo de caso foi criado com o intuito de simular a execução do sistema em diferentes situações. Por ser um trabalho inicial, desenvolvido com recursos limitados, não foi possível experimentá-lo em cenários reais de desenvolvimento. Assim, foi criada uma situação hipotética para realizar uma experimentação simulada.

O principal objetivo do estudo de caso é assegurar que, a partir do mesmo modelo de processo, dadas as características da equipe e do projeto, sejam gerados processos adequados. Desta forma, foram definidos dois cenários com diferenças na disponibilidade de papéis e prazos disponíveis.

O primeiro cenário, denominado *Cenário Base*, representa a equipe em sua capacidade máxima, com todos os papéis à disposição para trabalho no projeto, bem como com prazo suficiente para a execução das atividades. O *Cenário Limitado* foi definido para representar uma limitação de disponibilidade de tempo e pessoal na equipe.

As próximas subseções descrevem o modelo de processo usado na condução do estudo de caso, os cenários para os quais o sistema foi executado e o tipo de resultado esperado em cada um, a execução do sistema e os resultados obtidos.

5.2.1 Modelo de processo

Por se tratar de uma situação hipotética, optou-se por definir um modelo de processo simples, mas capaz de explorar as capacidades do sistema implementado. As atividades foram definidas usando-se como fonte de inspiração os processos RUP (KRUCHTEN, 2000) e Praxis (PAULA FILHO, 2009). Esta subseção apresenta os elementos do modelo. A especificação completa, em linguagem 2APL, pode ser vista no apêndice C.

A seguir está a lista de atividades de processo definidas:

1. Escrever caso de negócio
2. Levantar riscos
3. Redigir visão
4. Especificar requisitos
5. Modelar casos de uso
6. Projetar interface de usuário
7. Modelar classes
8. Modelar interações
9. Projetar arquitetura
10. Implementar classes
11. Testar classes
12. Implementar subsistemas
13. Integrar subsistemas
14. Teste de integração
15. Projetar testes
16. Testar sistema
17. Escrever documentação de usuário
18. Implantar sistema

A figura 5-17 mostra as possibilidades de execução das atividades, observadas suas entradas e saídas.

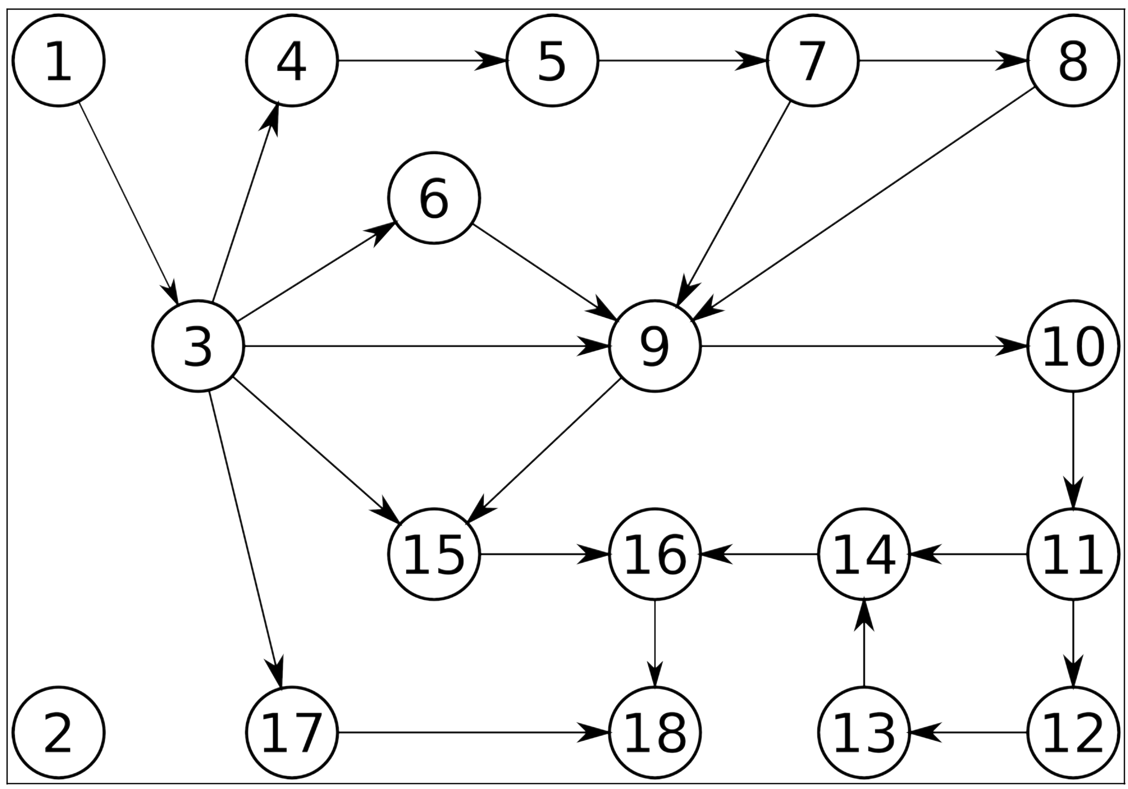

***Figura 5-17**. Possibilidades de execução do processo adotado no estudo de caso*

Os modelos de artefato adotados no modelo de processo seguem listados abaixo:

1. Caso de Negócio
2. Lista de Riscos
3. Visão
4. Especificação de Requisitos
5. Casos de Uso
6. Modelo de Casos de Uso
7. Modelo de Classes
8. Modelo de Interface do Usuário
9. Classes
10. Modelo de Interação
11. Arquitetura
12. Subsistemas
13. Sistema
14. Relatório de Teste de Classe
15. Relatório de Teste de Integração
16. Manual do Usuário
17. Guia de Testes do Sistema
18. Relatório de Testes do Sistema
19. Relatório de Implantação do Sistema

Segue a lista dos papéis que executam as atividades:

1. Analista de Negócio
2. Analista de Sistema
3. Arquiteto
4. Projetista de Interface
5. Programador
6. Projetista de Testes
7. Documentador Técnico
8. Testador
9. Implantador

5.2.2 Cenário base

O primeiro cenário de execução do sistema foi estabelecido como a base para

comparação dos resultados, pois não impôs restrições de disponibilidade de papéis ou modelos de artefato, bem como não foi definido um prazo capaz de ser um desafio à geração dos processos pelos especialistas.

Nestas condições, espera-se um processo gerado com todas as atividades, ordenadas adequadamente. Não é possível prever a ordem exata dos resultados, pois há mais de uma possibilidade de ordenação total das atividades descritas.

5.2.3 Cenário limitado

Este cenário foi concebido para que os agentes sejam confrontados com um maior desafio para gerar o processo de desenvolvimento, visto que há restrições tanto de papéis quanto de modelos de artefato disponíveis. Em função da baixa complexidade da instância de execução, não foi possível estabelecer uma limitação de prazo capaz de representar um desafio à geração do processo sem, contudo, impedir a criação do mesmo.

Os seguintes modelos de artefato foram marcados como indisponíveis neste cenário de execução: *Caso de Negócio*, *Lista de Riscos*, *Casos de Uso* e *Modelo de Casos de Uso*.

Como o modelo de processo é bastante simples, apenas um papel foi removido neste cenário: *Analista de Negócio*.

Nas condições definidas, espera-se uma saída diferente daquela apresentada no Cenário Base, visto que a indisponibilidade de alguns artefatos forçará os agentes a buscarem por substitutos em suas simulações. Especificamente, espera-se uma saída sem as atividades: *Escrever caso de negócio*, *Levantar riscos* e *Modelar casos de uso*.

5.2.4 Execução e resultados

A execução do sistema é simples, porém relativamente lenta. Não foi tomada nenhuma atitude no intuito de eliminar gargalos ou aumentar a performance geral da execução. Há um grande número de mensagens trocadas, o que se deve bastante ao fato de o gerente de projetos ser o intermediador de toda a comunicação.

A execução do sistema resulta na formação de um processo de desenvolvimento na base de conhecimentos do gerente de projetos. Os processos gerados a partir da execução em ambos os cenários propostos são listados a seguir, juntamente com a quantidade de mensagens trocadas entre os agentes e uma avaliação dos resultados, com base nas expectativas apresentadas anteriormente.

A figura 5-18 mostra o processo gerado para o Cenário Base, em sua forma como é representado na base de conhecimentos e em representação gráfica, conforme

definição do modelo do processo apresentada na subseção 5.2.1. Durante a execução que gerou o processo foram trocadas aproximadamente 950 mensagens entre todos os agentes do sistema.

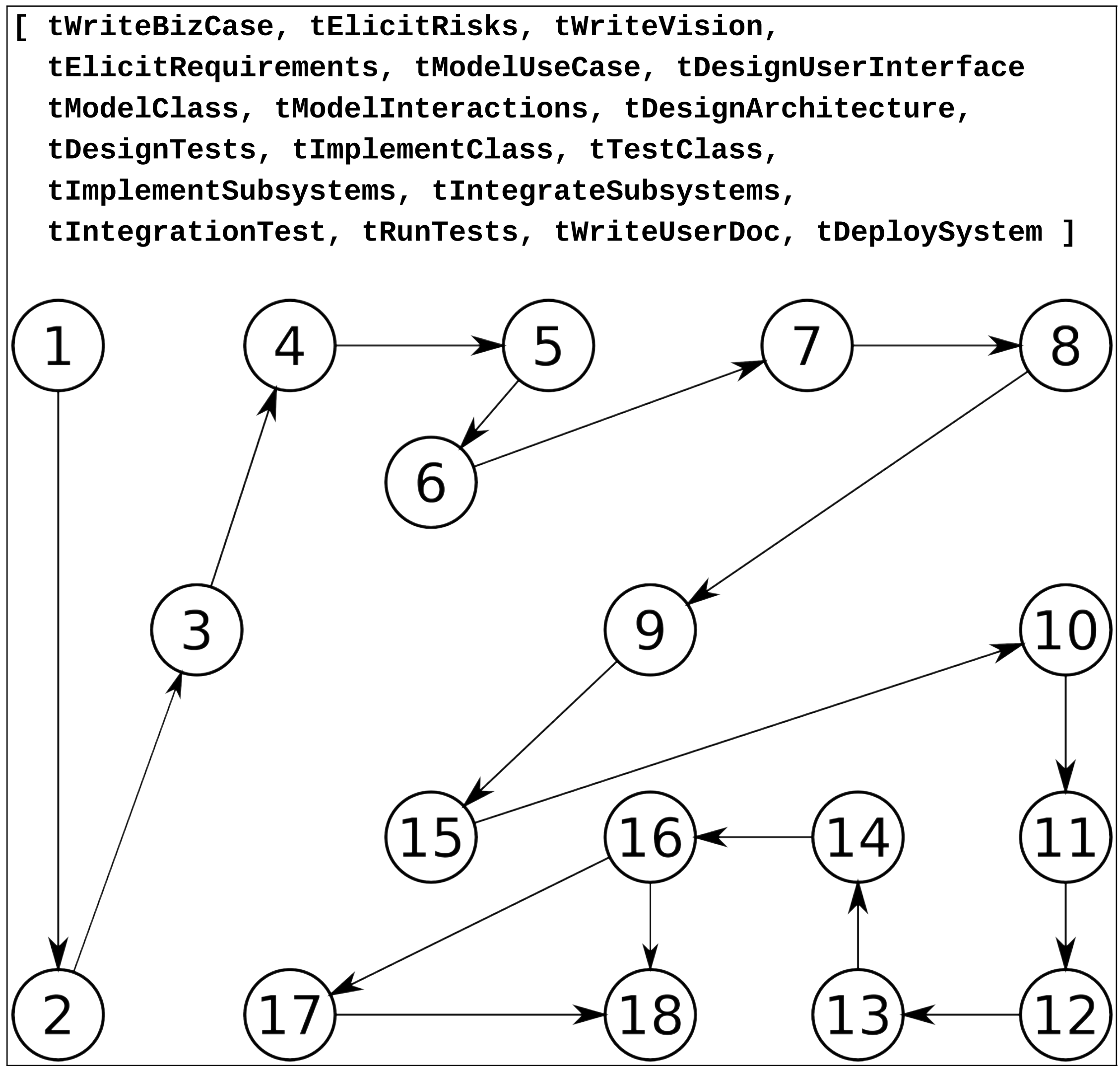

***Figura 5-18**. Processo gerado para Cenário Base do estudo de caso*

O resultado foi como esperado, já que todas as atividades foram contempladas no processo gerado, em uma sequência adequada, em conformidade com as possibilidades apresentadas na figura 5-17.

O resultado da execução do sistema para o Cenário Limitado é apresentado na figura 5-19, de forma análoga ao apresentado para o cenário base. Neste caso, foram trocadas aproximadamente 820 mensagens entre os agentes.

O resultado condiz com a expectativa, pois todas as atividades geradas eram esperadas e estão adequadamente ordenadas, conforme figura 5-17.

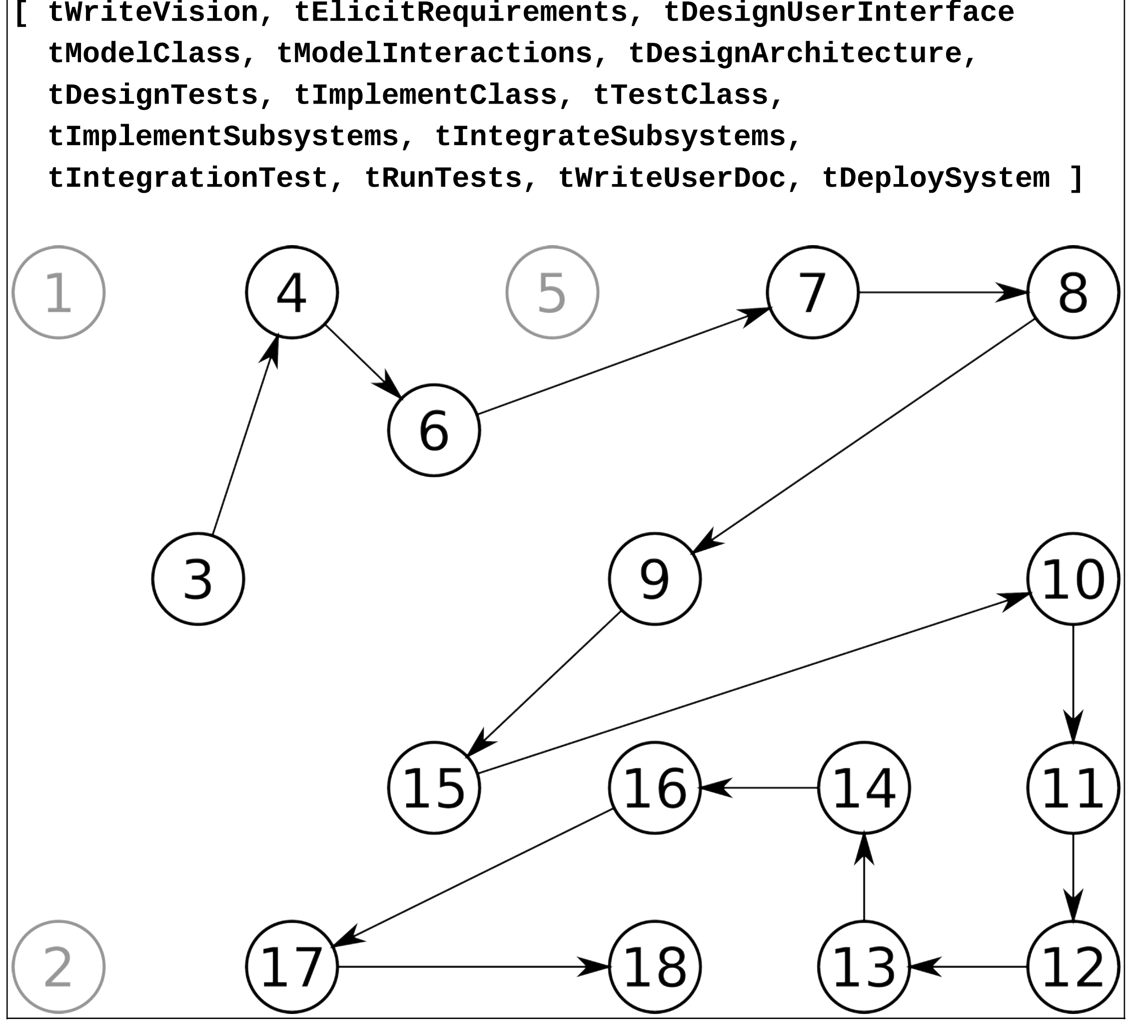

Figura 5-19. *Processo gerado para Cenário Limitado do estudo de caso*

As atividades incluídas no processo foram:

3. Redigir visão
4. Especificar requisitos
6. Projetar interface de usuário
7. Modelar classes
8. Modelar interações
9. Projetar arquitetura
10. Implementar classes
11. Testar classes
12. Implementar subsistemas
13. Integrar subsistemas
14. Teste de integração
15. Projetar testes
16. Testar sistema
17. Escrever documentação de usuário
18. Implantar sistema

Os processos gerados em ambos os cenários de simulação são viáveis, de acordo com a definição 4-8.

De forma geral, o sistema se comportou conforme a expectativa em ambos os cenários delineados para este estudo de caso hipotético. Os resultados são animadores, apesar de a implementação ser incipiente, e sugerem que é possível fornecer apoio automatizado de forma eficaz para a customização de processos de software.

6 CONCLUSÕES E TRABALHOS FUTUROS

“Whatever begins, also ends”
Seneca

A garantia da qualidade dos produtos de software está diretamente ligada à qualidade de todo processo que leva ao seu desenvolvimento. Uma grande parte dos projetos de software fracassa por causa de falhas na condução desse processo, que podem ser ocasionadas pela inadequação do mesmo à equipe por ele responsável. Uma possível maneira de solucionar o problema é fazer adaptações nos modelos de software disponíveis no mercado para as realidades específicas das organizações ou mesmo dos projetos em si. Entretanto, a tarefa de adaptar os elementos do processo a situações específicas é trabalhoso e consome recursos – humanos, de máquina ou financeiros – e tempo, que podem não estar disponíveis.

Uma diminuição na quantidade de recursos necessários para executar a adaptação dos processos permite a cada organização definir um processo próprio, levando em consideração suas próprias particularidades. Os dois elementos mais complexos para realizar a tarefa, pessoal e tempo, podem ter seu custo reduzido se a customização for automatizada em algum grau. A criação de sistemas computacionais para suportar este tipo de tarefa é trabalhosa, pois há muitas variáveis correlacionadas a serem controladas durante o processo de desenvolvimento de software e o próprio produto final é complexo e intangível. Neste sentido, sistemas multiagentes podem ser empregados para simplificar a modelagem do problema e, consequentemente, da solução. Agentes planejadores adequadamente arranjados podem gerar processos de desenvolvimento a partir de dados fornecidos pelas equipes desenvolvedoras de software.

Este trabalho se propôs a criar um método baseado em um sistema multiagentes e planejamento distribuído para definir um processo de software a partir das informações sobre os requisitos funcionais do projeto e de características da equipe, como disponibilidade de pessoal e tipos de artefato criados ao longo do desenvolvimento.

A partir do método proposto e testado por meio da condução de um estudo de caso hipotético com uma implementação do método, os resultados obtidos sugerem a

possibilidade real de customizar processos de software de forma semiautomática, com base em dados sobre o projeto e sobre a equipe por ele responsável.

Tanto o método quanto a implementação propostos são incipientes e levam em consideração uma pequena parte das variáveis envolvidas com a tarefa de customizar processos de desenvolvimento de software. Desta forma, do ponto de vista prático, os resultados ainda são tímidos, mas constituem uma base importante para trabalhos nesta área.

O sistema implementado em linguagem 2APL é consideravelmente ineficiente, mesmo para executar uma instância pequena, como a situação hipotética descrita no Capítulo 5. Este fato não foi considerado um problema, pois nenhum esforço foi direcionado para aumentar a performance do sistema. Focou-se em obter um método eficaz e, neste sentido, logrou-se êxito.

Confirmou-se a hipótese levantada e consideram-se atingidos os objetivos propostos. Modelos algébricos dos diversos elementos relacionados a um processo de desenvolvimento de software foram criados, permitindo a realização de inferências sobre uma base de conhecimentos definida a partir dos mesmos. Foi definido um método para gerar um processo de software a partir de características do projeto (requisitos funcionais e prazo) e da equipe de desenvolvimento por ele responsável (papéis e modelos de artefato disponíveis). Um sistema multiagente baseado no método proposto foi implementado, ainda que parcialmente. Este fato por si não comprova, mas evidencia a aplicabilidade do método. A fim de averiguar a eficácia do método e do sistema implementado, foi conduzido um estudo de caso hipotético, considerando-se dois cenários distintos para os dados de entrada, que resultou na geração de processos também diferentes, conforme o esperado.

6.1 Principais contribuições

Considera-se que os resultados obtidos constituem uma importante contribuição no sentido de iniciar discussões ligadas à automação da tarefa de customização de processos de software, visto que poucos foram os trabalhos relacionados a este assunto encontrados durante a pesquisa bibliográfica realizada.

O método definido e a implementação realizada com base no mesmo constituem contribuições de cunho mais prático. A definição conceitual do método permite lidar com um problema complexo – customizar processos de desenvolvimento de software – de forma mais simplificada e tratável por um sistema computacional. A implementação permitiu aplicar o método e averiguar, com sucesso, a adequação do mesmo na solução

de situações hipotéticas.

6.2 Trabalhos futuros

Mesmo tendo considerado atingidos os objetivos do trabalho, não se pretende tratar o assunto como esgotado, visto que muitos outros aspectos podem ser tratados em trabalhos futuros.

Propõe-se que o conjunto de agentes seja dividido em dois grupos distintos: um para lidar com entrada e saída de dados no sistema e outro para realizar a simulação da execução do processo. Desta forma, não seria necessário que o Gerente de Projetos desempenhasse ambas as funções, liberando-o para exercer o papel de gerente de projetos no processo de desenvolvimento.

Ainda com relação à comunicação, pode ser investigada a possibilidade de adoção de uma forma diferente de organização da sociedade, com o intuito de diminuir a quantidade de mensagens trocadas e, eventualmente, obter melhora no desempenho. Uma possibilidade é a não existência de um coordenador central do processo comunicativo, permitindo aos especialistas trocarem mensagens diretamente entre si.

A incorporação de outros conceitos, como os de fases e iterações, é essencial para tornar o método proposto cada vez mais prático, capaz de lidar com situações e modelos de processo reais. Neste intuito, é interessante avaliar a incorporação de elementos de métodos ágeis e híbridos (SOARES, 2007).

Sugere-se ainda a utilização de elementos de entrada mais elaborados, como o conjunto de boas práticas sugeridas após a aplicação da metodologia apresentada por Satler (2010). A ideia é aproveitar a recomendação para direcionar o trabalho dos agentes na seleção das atividades para compor o processo.

Com relação à geração do processo final, podem ser feitas mudanças no algoritmo utilizado pelo Gerente de Projetos na unificação dos planos individuais, possibilitando a geração de um plano parcialmente ordenado, considerando a execução paralela de atividades.

APÊNDICE A

Para melhor compreensão da implementação proposta no capítulo 5, cujos códigos são apresentados nos apêndices B e C, a figura A-1 traz a definição EBNF da linguagem 2APL. Uma descrição completa da linguagem é dada por Dastani (2008).

```
⟨Agent_Prog⟩       =  { "Include:" ⟨ident⟩
                   |  "BeliefUpdates:" ⟨BelUpSpec⟩
                   |  "Beliefs:" ⟨belief⟩
                   |  "Goals:" ⟨goals⟩
                   |  "Plans:" ⟨plans⟩
                   |  "PG-rules:" ⟨pgrules⟩
                   |  "PC-rules:" ⟨pcrules⟩
                   |  "PR-rules:" ⟨prrules⟩ } ;
⟨BelUpSpec⟩        =  ( "{"⟨belquery⟩"}" ⟨beliefupdate⟩ "{"⟨literals⟩"}" )+ ;
⟨belief⟩           =  ( ⟨ground_atom⟩"." | ⟨atom⟩ ":-" ⟨literals⟩"." )+ ;
⟨goals⟩            =  ⟨goal⟩ { ","⟨goal⟩ } ;
⟨goal⟩             =  ⟨ground_atom⟩ { "and" ⟨ground_atom⟩ } ;
⟨baction⟩          =  "skip" | ⟨beliefupdate⟩ | ⟨sendaction⟩ | ⟨externalaction⟩
                   |  ⟨abstractaction⟩ | ⟨test⟩ | ⟨adoptgoal⟩ | ⟨dropgoal⟩ ;
⟨plans⟩            =  ⟨plan⟩ { "," ⟨plan⟩ } ;
⟨plan⟩             =  ⟨baction⟩ | ⟨sequenceplan⟩ | ⟨ifplan⟩ | ⟨whileplan⟩ | ⟨atomicplan⟩ ;
⟨beliefupdate⟩     =  ⟨Atom⟩ ;
⟨sendaction⟩       =  "send(" ⟨iv⟩ "," ⟨iv⟩ "," ⟨atom⟩ ")" ;
                   |  "send(" ⟨iv⟩ "," ⟨iv⟩ "," ⟨iv⟩ "," ⟨iv⟩ "," ⟨atom⟩ ")" ;
⟨externalaction⟩   =  "@"⟨ident⟩"("⟨atom⟩ "," ⟨Var⟩ ")" ;
⟨abstractaction⟩   =  ⟨atom⟩ ;
⟨test⟩             =  "B(" ⟨belquery⟩ ")" | "G"( ⟨goalquery⟩ ")" | ⟨test⟩ "&" ⟨test⟩ ;
⟨adoptgoal⟩        =  "adopta(" ⟨goalvar⟩ ")" | "adoptz(" ⟨goalvar⟩ ")" ;
⟨dropgoal⟩         =  "dropgoal(" ⟨goalvar⟩ ")" | "dropsubgoals(" ⟨goalvar⟩ ")"
                   |  "dropsupergoals(" ⟨goalvar⟩ ")" ;
⟨sequenceplan⟩     =  ⟨plan⟩ ";" ⟨plan⟩ ;
⟨ifplan⟩           =  "if" ⟨test⟩ "then" ⟨scopeplan⟩ ["else" ⟨scopeplan⟩] ;
⟨whileplan⟩        =  "while" ⟨test⟩ "do" ⟨scopeplan⟩ ;
⟨atomicplan⟩       =  "[" ⟨plan⟩ "]" ;
⟨scopeplan⟩        =  "{" ⟨plan⟩ "}" ;
⟨pgrules⟩          =  ⟨pgrule⟩+ ;
⟨pgrule⟩           =  [⟨goalquery⟩] "<-" ⟨belquery⟩ "|" ⟨plan⟩ ;
⟨pcrules⟩          =  ⟨pcrule⟩+ ;
⟨pcrule⟩           =  ⟨atom⟩ "<-" ⟨belquery⟩ "|" ⟨plan⟩ ;
⟨prrules⟩          =  ⟨prrule⟩+ ;
⟨prrule⟩           =  ⟨planvar⟩ "<-" ⟨belquery⟩ "|" ⟨planvar⟩ ;
⟨goalvar⟩          =  ⟨atom⟩{"and"⟨atom⟩} ;
⟨planvar⟩          =  ⟨plan⟩ | ⟨Var⟩ | "if" ⟨test⟩ "then" ⟨scopeplanvar⟩ ["else" ⟨scopeplanvar⟩]
                   |  "while" ⟨test⟩ "do" ⟨scopeplanvar⟩ | ⟨planvar⟩ ";" ⟨planvar⟩ ;
⟨scopeplanvar⟩     =  "{" ⟨planvar⟩ "}" ;
⟨literals⟩         =  ⟨literal⟩ { "," ⟨literal⟩} ;
⟨literal⟩          =  ⟨atom⟩ | "not" ⟨atom⟩ ;
⟨belquery⟩         =  "true" | ⟨belquery⟩ "and" ⟨belquery⟩ | ⟨belquery⟩ "or" ⟨belquery⟩
                   |  "(" ⟨belquery⟩ ")" | ⟨literal⟩ ;
⟨goalquery⟩        =  "true" | ⟨goalquery⟩ "and" ⟨goalquery⟩ | ⟨goalquery⟩ "or" ⟨goalquery⟩
                   |  "(" ⟨goalquery⟩ ")" | ⟨atom⟩ ;
⟨iv⟩               =  ⟨ident⟩ | ⟨Var⟩ ;
```

Figura A-1. *Definição EBNF da linguagem 2APL*

APÊNDICE B

A especificação dos elementos do sistema multiagente apresentado no capítulo 5 pode ser encontrada a seguir. Os arquivos de código fonte não foram transcritos na íntegra para não sobrecarregar o leitor com detalhes de implementação. Ateve-se a manter o foco no sistema propriamente dito. Cada subseção apresenta um aspecto do sistema, correspondente a um diretório na implementação real.

Fundamentos

Código 1. Definições básicas

```
%% Requisito funcional
BeliefUpdates:
  % AddFuncRequirement(X: func. Req., L: level of importance)
  { not funcRequirement(X) }
  AddFuncRequirement(X, L)
  { funcRequirement(X), frImportance(X, L) }

  % SetFRNew(X: functional requirement)
  { funcRequirement(X) } SetFRNew(X) { frNew(X) }

  % SetFRNewTeam(X: functional requirement)
  { funcRequirement(X) } SetFRNewTeam(X) { frNewTeam(X) }

  % SetFRNewClient(X: functional requirement)
  { funcRequirement(X) } SetFRNewClient(X) { frNewClient(X) }

  % SetFRMutable(X: functional requirement)
  { funcRequirement(X) } SetFRMutable(X) { frMutable(X) }

  % AddRelatedModel(X: functional requirement, M: artifact model)
  { not frArtifactModel(X, M) } AddRelatedModel(X, M) { frArtifactModel(X, M) }

PC-rules:
  % Add a set of related artifact models to a functional requirement
  % addRelatedModels(X: functional requirement, M: set of artifact models)
  addRelatedModels(X, M) <- M = [NextModel|REST] | {
    AddRelatedModel(X, NextModel);
    if B(not REST=[]) then { addRelatedModels(X, REST) }
  }

%% Papel
Beliefs:
  rTask(R, T) :- tPerformer(T, R).

BeliefUpdates:
  % AddRole(R: role, N: name)
  { not role(R) } AddRole(R, N) { role(R), rName(R, N) }

%% Modelo de artefato
Beliefs:
  instanceOf(M, M).
  instanceOf(M, G) :- amParent(M, P), instanceOf(P, G).

BeliefUpdates:
  % AddArtifactModel(X: artifact model)
  { not artifactModel(X) } AddArtifactModel(X) { artifactModel(X) }
```

```
  % AddAMUpdater(X: artifact model, U: updater)
  { artifactModel(X) and not amUpdater(X, U) } AddAMUpdater(X, U) { amUpdater(X, U) }

  % AddAMReader(X: artifact model, R: reader)
  { artifactModel(X) and not amReader(X, R) } AddAMReader(X, R) { amReader(X, R) }

  % SetAMParent(X: artifact model, P: parent model)
  { artifactModel(X) and (not amParent(X, _) or amParent(X, B)) }
  SetAMParent(X, P)
  { not amParent(X, B), amParent(X, P) }

PC-rules:
  % Add a set of artifact model updaters
  % addAMUpdaters(X: artifact model, U: set of updaters)
  addAMUpdaters(X, U) <- U = [NextUpdater|REST] | {
    AddAMUpdater(X, NextUpdater);
    if B(not REST=[]) then { addAMUpdaters(X, REST) }
  }

  % Add a set of artifact model readers
  % addAMReaders(X: artifact model, R: set of readers)
  addAMReaders(X, R) <- R = [NextReader|REST] | {
    AddAMReader(X, NextReader);
    if B(not REST=[]) then { addAMReaders(X, REST) }
  }

%% Artefato
Beliefs:
  status(sFinished, sRevised).
  status(sRevised, sUpdated).
  status(sUpdated, sStarted).
  status(X, X).
  aStatus(A, sRevised) :- aStatus(A, sFinished).
  aStatus(A, sUpdated) :- aStatus(A, sRevised).
  aStatus(A, sStarted) :- aStatus(A, sUpdated).

BeliefUpdates:
  % AddArtifact(X: artifact, O: owner, M: model)
  {  } AddArtifact(X, O, M) { artifact(X), aOwner(X, O), aModel(X, M) }

  { artifact(X) and aStatus(X, B) }
  SetArtifactStatus(X, S)
  { not aStatus(X, B), aStatus(X, S) }

  % SetArtifactStatus(X: artifact, S: status)
  { artifact(X) and not aStatus(X, _) }
  SetArtifactStatus(X, S)
  { aStatus(X, S) }

  % AddRelatedRequirement(X: artifact, P: requirement)
  {  } AddRelatedRequirement(X, P) { aRequirement(X, P) }

PC-rules:
  % Add a set of related requirements to an artifact
  % addRelatedRequirements(X: artifact, P: set of requirements)
  addRelatedRequirements(X, P) <- P = [NextRequirement|REST] | {
    AddRelatedRequirement(X, NextRequirement);
    if B(not REST=[]) then { addRelatedRequirements(X, REST) }
  }

%% Atividade do processo
Beliefs:
  tInputs(T, X)  :- bagof(I, tInput(T, I, S), X).
  tOutputs(T, X) :- bagof(O, tOutput(T, O, S), X).

BeliefUpdates:
  { tDuration(T, X) }
  SetTaskDuration(T, D)
  { not tDuration(T, X), tDuration(T, D) }

  { not tDuration(T, _) } SetTaskDuration(T, D) { tDuration(T, D) }
  { not tBlacklisted(T) } BlacklistTask(T) { tBlacklisted(T) }
  { tBlacklisted(T) } WhitelistTask(T) { not tBlacklisted(T) }

%% Outras
BeliefUpdates:
  % Time constraint   <=> timeConstraint(R, T).
  % SetTimeConstraint(R: role, T: time constraint)
  { } SetTimeConstraint(R, T) { timeConstraint(R, T) }
```

Código 2. Gerente de Projetos

```
%% Camada de Definição do Processo
Beliefs:
  created(process) :- process(P), not P = [],
                      not waiting_subprocess(_),
                      not subprocess(_, _).
  process([]).

BeliefUpdates:
  % Add a task to the process
  { process(Process) and concat(Process, [T], NewProcess) }
  AddTask(T)
  { not process(Process), process(NewProcess) }

  % Redefine an agent's subprocess
  { subprocess(A, X) }
  SetAgentSubprocess(A, S)
  { not subprocess(A, X), subprocess(A, S) }

  % Removes an agent's subprocess
  { subprocess(A, S) } RemoveSubprocess(A) { not subprocess(A, S) }

PC-rules:
  % Merge a set of subprocesses into one process
  % mergeSubprocesses(S: set of subprocesses)
  mergeSubprocesses(S) <- S = [NextSubprocess | REST] | {
    processSubprocess(NextSubprocess);
    if B(not REST=[]) then { mergeSubprocesses(REST) }
  }

  % Process an agent's subprocess to add it to the unified process
  % processSubprocess(S: subprocess)
  processSubprocess(S) <- S = [task(T) | REST] | {
    % If the next item on the list is a task, add it to the process
    AddTask(T);
    [ B(subprocess(A, S));
      SetAgentSubprocess(A, REST)
    ];
    processSubprocess(REST)
  }

  processSubprocess(S) <- S = [dep(D, Status) | REST] | {
    if B( (artifact(A) and aModel(A, D))
         or (artifact(D) and aModel(D, _)) ) then { skip }
    else {
      B(agent(Agent, artifactLinker));
      requestInformation(Agent, artifactModel, D);
      B((artifact(A) and aModel(A, D)) or (artifact(D) and aModel(D, _)))
    };
    if B((artifact(X) and aModel(X, D) and aStatus(X, Status))
         or (artifact(D) and aStatus(D, Status))
         or (tOutput(T, D, S1) and status(S1, Status) and process(P) and member(T, P))
         or (tOutput(T, M, S2) and status(S2, Status)
          and aModel(D, M) and process(P) and member(T, P)) ) then {
      B(assign(DEPOK, 1))
    } else {
      B(assign(DEPOK, 0))
    };
    if B(DEPOK = 1) then {
      % If the dependency has been satisfied, remove it from the subprocess and go on
      [ B(subprocess(A, S));
        SetAgentSubprocess(A, REST)
      ];
      processSubprocess(REST)
    }
  }

  processSubprocess(S) <- S = [] | {
    if B(subprocess(A, [])) then {
      RemoveSubprocess(A)
    }
  }

  processTaskOutputs(T, O) <- O = [M|REST] | {
    B(tPerformer(T, R));
    B(tOutput(T, M, S));
    B(agent(Agent, artifactLinker));
    requestInformation(Agent, artifactName, task(T, M, S));
```

```
    B(artifact(A) and aModel(A, M) and aStatus(A, S));
    if B(not REST=[]) then { processTaskOutputs(T, REST) }
  }

%% Camada de Simulação do Processo
PG-rules:
  created(process) <- waitingSubprocess(_) | {
    B(bagof(X, subscribed(X), AllAgents));
    if B(needingArtifact(Agent, Artifact, Status)) then {
      B(bagof(Y, needingArtifact(_, Y, _), Artifacts));
      processNeedingArtifacts(Artifacts, AllAgents)
    }
  }

  created(process) <- not waitingSubprocess(_) | {
    B(bagof(S, subprocess(_, S), Subprocesses));
    mergeSubprocesses(Subprocesses)
  }

PC-rules:
  processNeedingArtifacts(A, AllAgents) <- A = [Artifact|REST] | {
    if B(needingArtifact(Agent, Artifact, Status)) then {
      if B(availableArtifact(_, Artifact, Status)) then {
        sendArtifactAvailable(Agent, Artifact, Status)
      } else {
        B(remove_element(Agent, AllAgents, Agents));
        queryArtifactAvail(Agents, Artifact, Status);
        B(not waitingArtifactAvail(_, Artifact, Status));
        if B(not availableArtifact(_, Artifact, Status)) then {
          queryArtifactBuilder(Agents, Artifact, Status);
          B(not waitingArtifactBuilder(_, Artifact, Status))
        };
        if B(availableArtifact(_, Artifact, Status)) then {
          sendArtifactAvailable(Agent, Artifact, Status)
        } else {
          sendArtifactUnavailable(Agent, Artifact, Status)
        }
      }
    };
    if B(not REST=[]) then { processNeedingArtifacts(REST, AllAgents) }
  }
```

Código 3. Especialista

```
%% Camada de Definição do Processo
Beliefs:
  subprocess([]).

BeliefUpdates:
  % Adds a task to the agent's subprocess
  { subprocess(S) and concat(S, [ task(T) ], NewSubprocess) }
  AddTask(T)
  { not subprocess(S), subprocess(NewSubprocess) }

  % Adds an artifact dependency to the agent's subprocess
  { subprocess(X) and concat(X, [ dep(D, S) ], NewSubprocess) }
  AddDependency(D, S)
  { not subprocess(X), subprocess(NewSubprocess) }

  % State that the subprocess is being created
  {  } Creating(S) { creating(S) }

  % State that the subprocess creation has ended
  { creating(P) } NotCreating(P) { not creating(P), created(P) }

PG-rules:
  created(subprocess) <- not creating(subprocess) | {
    Creating(subprocess)
  }

  created(subprocess) <- creating(subprocess) | {
    if G(artifact(_, _) or artifact(_)) then { skip }
    else { endSubprocessCreation() }
  }
```

```
PC-rules:
  % End subprocess creation
  endSubprocessCreation() <- creating(S) | {
    B(subprocess(P));
    sendSubprocess(P);
    NotCreating(S)
  }

%% Camada de Simulação do Processo
Beliefs:
  taskInputsSatisfied(T) :- tInputs(T, Inputs),
                            hasArtifacts(T, Inputs).
  hasArtifacts(_, []).
  hasArtifacts(T, [NextModel|REST]) :- tInput(T, NextModel, Status),
                                       artifact(NextModel, Status),
                                       hasArtifacts(T, REST).
  artifact(M, S) :- artifact(A), aModel(A, M), aStatus(A, S).
  artifact(M, S) :- artifactSubstitution(A, X),
                    aModel(A, M), aModel(X, P), instanceOf(M, P),
                    aStatus(X, S).
  artifact(M, S) :- artifactSubstitution(M, X),
                    instanceOf(X, Y), aModel(A, Y), aStatus(A, S).
  processDuration([], 0).
  processDuration([dep(Dep, S)|P], D) :- processDuration(P, D).
  processDuration([task(Task)|P],TD+PD) :- tDuration(Task, TD), processDuration(P, PD).

BeliefUpdates:
  % State that a task is being processed and analyzed
  {  } ProcessingTask(T) { processingTask(T) }

  % State that a task is not being processed and analyzed anymore
  {  } NotProcessingTask(T) { not processingTask(T) }

  % Defines an artifact substitution
  {  } ArtifactSubstitution(Old, New) { artifactSubstitution(Old, New) }

  % State that an artifact is not buildable by the agent
  {  } ArtifactNotBuildable(A, S) { artifactNotBuildable(A, S) }

PG-rules:
  % Pursuit the goal of build an artifact
  artifact(A, S) <- tOutput(T, A, S)
                    and availableModel(A) and tPerformer(T, R)
                    and selfRole(R) and taskInputsSatisfied(T)
                    and not tBlacklisted(T) | {
    if B(not tDuration(T, _)) then {
      B(agent(TimeAgent, timeProvider));
      requestInformation(TimeAgent, tDuration, T)
    };
    B(tDuration(T, Duration));
    if B(subprocess(P) and processDuration(P, PD)
         and timeConstraint(R, Constraint)
         and PD+Duration =< Constraint) then {
      addTask(T)
    } else { BlacklistTask(T) }
  }

  artifact(A, S) <- tOutput(T, A, S) and availableModel(A)
                    and tPerformer(T, R) and selfRole(R) and not taskInputsSatisfied(T)
                    and not tBlacklisted(T) and not processingTask(T) | {
    ProcessingTask(T);
    B(tInputs(T, Inputs));
    processTaskInputs(T, Inputs)
  }

  artifact(A, S) <- tInput(T, A, S) and instanceOf(A, X)
                    and availableModel(X) and not availableModel(A) | {
    ArtifactSubstitution(A, X);
    [ if B(not artifact(X) or not aStatus(X, S)) then {
        if G(artifact(X, S)) then { skip }
        else { adopta(artifact(X, S)) }
      }
    ];
    dropgoal(artifact(A, S))
  }

  artifact(A) <- tInput(T, A, _) and processingTask(T) and aModel(A, M) and artifact(X)
                 and aModel(X, P) and instanceOf(M, P) | {
    ArtifactSubstitution(A, X);
    [ if B(not artifact(X) or not aStatus(X, S)) then {
```

```
        if G(artifact(X, S)) then { skip }
        else { adopta(artifact(X, S)) }
      }
    ];
    dropgoal(artifact(A, S))
  }

  artifact(A, S) <- processingTask(T) and tInput(T, A, B)
                    and status(B, S) and tOutput(T2, A, C)
                    and status(C, S) and selfRole(R) and not tPerformer(T2, R)
                    and not artifactNotBuildable(A) | {
    ArtifactNotBuildable(A, S)
  }

  artifact(A, S) <- artifactNotBuildable(A, S) and availableModel(A)
                    and not needingArtifact(A, S) | {
    requestArtifact(A, S)
  }

  artifact(A, S) <- processingTask(TI) and tInput(TI, X, S) and tOutput(TO, X, S)
                    and artifactSubstitution(X, Y) and instanceOf(Y, A) and selfRole(R)
                    and tPerformer(TI, R) and not tPerformer(TO, R)
                    and not needingArtifact(A, S) | {
    ArtifactNotBuildable(A, S)
  }

  artifact(A, S) <- amParent(A, X) and not tOutput(T, A, S) | {
    ArtifactSubstitution(A, X);
    [ if B(not artifact(X) or not aStatus(X, S)) then {
        if G(artifact(X, S)) then { skip }
        else { adopta(artifact(X, S)) }
      }
    ];
    dropgoal(artifact(A, S))
  }

PC-rules:
  % Adds a task and its dependencies to the subprocess
  % addTask(T: task)
  addTask(T) <- task(T) | {
    B(tInputs(T, Inputs));
    [ if B(not Inputs=[]) then { addDependencies(T, Inputs) };
      AddTask(T)
    ];
   B(tOutputs(T, Outputs));
    processTaskOutputs(T, Outputs);
    if B(tBlacklisted(T)) then { WhitelistTask(T) };
    NotProcessingTask(T)
  }

  % Adds task's dependencies to the subprocess
  % addDependencies(T: task, D: set of dependencies)
  addDependencies(T, D) <- D = [Model|REST] | {
    B(aModel(NextDep, X) and instanceOf(Model, X) and tInput(T, Model, S));
    if B(artifactSubstitution(NextDep, N)) then { AddDependency(N, S) }
    else { AddDependency(NextDep, S) };
    if B(not REST=[]) then { addDependencies(T, REST) }
  }

  % Processes a task's input list to adopt new goals accordingly
  % processTaskInputs(T: task, Inputs: set of inputs)
  processTaskInputs(T, Inputs) <- Inputs = [Model|REST] | {
    [ if B(not aModel(NextArt, Model)) then {
        B(tInput(T, Model, S));
        adopta(artifact(Model, S))
      }
    ];
    if B(not REST=[]) then { processTaskInputs(T, REST) }
  }

  % Processes a task's output list to update the agent's knowledge base
  % processTaskOutputs(T: task, Outputs: set of outputs)
  processTaskOutputs(T, Outputs) <- Outputs = [Model|REST] | {
    B(tOutput(T, Model, S));
    B(agent(Agent, artifactLinker));
    requestInformation(Agent, artifactName, task(T, Model, S));
    B(artifact(A) and aModel(A, Model) and aStatus(A, S));
    if B(not REST=[]) then { processTaskOutputs(T, REST) }
  }
```

Protocolos de Comunicação

Código 4. Protocolo de Inscrição – Gerente de Projetos

```
BeliefUpdates:
  % Subscribe an agent
  % Subscribe(A: agent, R: role)
  { true }
  Subscribe(A, R)
  { subscribed(A), agent(A, R), not waitingSubscription(R) }

PC-rules:
  % Process subscription message
  message(Agent, inform, La, On, subscription(R)) <- true | {
    if B(not agent(_, R)) then { Subscribe(Agent, R) };
    if B(agent(Agent, R)) then { send(Agent, inform, subscribed(R, 1)) }
    else { send(Agent, inform, subscribed(R, 0)) };
    if B(not waitingSubscription(_)) then { adopta(initialize) }
  }
```

Código 5. Protocolo de Inscrição – Especialista

```
BeliefUpdates:
  {  } WaitingSubscription(R) { waitingSubscription(R) }
  {  } NotWaitingSubscription(R) { not waitingSubscription(R) }

PG-rules:
  initialize <- not waitingSubscription(_) | {
    B(bagof(X, selfRole(X), Roles));
    sendSubscription(Roles);
    dropgoal(initialize)
  }

PC-rules:
  % Send list of subscriptions
  % sendSubscriptions(L: list of roles)
  sendSubscription(L) <- L = [NextRole | REST] | {
    B(agent(PM, rProjMgr));
    send(PM, inform, subscription(NextRole));
    WaitingSubscription(NextRole);
    if B(not REST = []) then { sendSubscription(REST) }
  }

  % Process the subscription status message
  % Message format: subscribed(R: role, S: status)
  message(PM, inform, La, On, subscribed(R, S)) <- true | {
    if B(S = 1) then { NotWaitingSubscription(R) }
  }
```

Código 6. Protocolo de Iniciação – Gerente de Projetos

```
BeliefUpdates:
  % Waiting for an agent to create its subprocess
  {  } WaitSubprocess(A) { waitingSubprocess(A) }

  % Temporary belief update rules to handle functional requirements
  {  } TmpFRNew(S) { tmpFRNew(S) }

PC-rules:
  % Send information abount a functional requirement
  % sendFuncRequirement(Receiver: agent, X: functional requirement,
  %     L: level of importance, N: new feature, T: new to team, C: new to client,
  %     M: mutable, R: related artifact models)
  sendFuncRequirement(Receiver, X, L, N, T, C, M, R) <- true | {
    send(Receiver, inform, funcRequirement(X, L, N, T, C, M, R))
  }

  % Send informationa about an artifact model
  % sendArtifactModel(Receiver: agent, M: artifact model, U: updaters, R: readers,
  %     P: parent model)
  sendArtifactModel(Receiver, M, U, R, P) <- true | {
```

```
  send(Receiver, inform, artifactModel(M, U, R, P))
}

% Send information about an available artifact model
% sendAvailableModel(Receiver: agent, M: artifact model)
sendAvailableModel(Receiver, M) <- true | {
  send(Receiver, inform, availableModel(M))
}

% Send information about an artifact
% sendArtifactStatus(Receiver: agent, A: artifact, O: owner, M: model, S: status,
%     P: related requirements)
sendArtifactStatus(Receiver, A, O, M, S, R) <- true | {
  send(Receiver, inform, artifact(A, O, M, S, R))
}

% Send information about a time constraint
% sendTimeConstraint(Receiver: agent, R: role, T: time constraint)
sendTimeConstraint(Receiver, R, T) <- true | {
  send(Receiver, inform, timeConstraint(R, T))
}

% Broadcast project information
% bcProjectInformation(Agents: set of agents)
bcProjectInformation(Agents) <- Agents = [NextAgent | REST] | {
  B(bagof(X, funcRequirement(X), Requirements));
  if B(not Requirements=[]) then {
    sendFuncRequirements(NextAgent, Requirements)
  };
  B(bagof(X, availableModel(X), Models));
  if B(not Models=[]) then { sendAvailableModels(NextAgent, Models) };
  B(bagof(X, artifact(X), Artifacts));
  if B(not Artifacts=[]) then { sendArtifacts(NextAgent, Artifacts) };
  if B(not REST=[]) then { bcProjectInformation(REST) }
}

% Send a set of functional requirements to an agent
% bcFuncRequirements(Agent: agent, R: set of requirements)
sendFuncRequirements(Agent, R) <- R = [NextReq | REST] | {
  [ B(frImportance(NextReq, L));
   if B(frNew(NextReq)) then { B(assign(N, 1)) }
   else { B(assign(N, 0)) };
   if B(frNewTeam(NextReq)) then { B(assign(NT, 1)) }
   else { B(assign(NT, 0)) };
   if B(frNewClient(NextReq)) then { B(assign(NC, 1)) }
   else { B(assign(NC, 0))  };
   if B(frMutable(NextReq)) then { B(assign(M, 1)) }
   else { B(assign(M, 0)) };
   B(bagof(X, frArtifactModel(NextReq, X), R));
   sendFuncRequirement(Agent, NextReq, L, N, NT, NC, M, R)
 ];
 if B(not REST=[]) then { sendFuncRequirements(Agent, REST) }
}

% Send a set of artifact models to an agent
% sendArtifactModels(Agent: agent, M: set of models)
sendArtifactModels(Agent, M) <- M = [NextModel | REST] | {
  [ B(bagof(X, amUpdater(NextModel, X), U));
    B(bagof(X, amReader(NextModel, X), R));
    if B(amParent(NextModel, Parent)) then { B(assign(P, Parent)) }
    else { B(assign(P, 0)) };
    sendArtifactModel(Agent, NextModel, U, R, P)
  ];
  if B(not REST=[]) then { sendArtifactModels(Agent, REST) }
}

% Send a set of available artifact models to an agent
% sendAvailableModels(Agent: agent, M: set of models)
sendAvailableModels(Agent, M) <- true | {
  send(Agent, inform, availableModels(M))
}

% Send a set of artifacts to an agent
% sendArtifacts(Agent: agent, A: set of artifacts)
sendArtifacts(Agent, A) <- A = [NextArtifact | REST] | {
  [ B(aOwner(NextArtifact, Owner));
    B(aModel(NextArtifact, Model));
    B(aStatus(NextArtifact, Status));
    B(bagof(X, aRequirement(NextArtifact, X), Requirements));
    sendArtifactStatus(
```

```
          Agent, NextArtifact, Owner, Model, Status, Requirements)
    ];
    if B(not REST=[]) then { sendArtifacts(Agent, REST) }
  }

  % Broadcast the "begin work" message
  % bcBeginWork(Agents: set of agents)
  bcBeginWork(Agents) <- Agents = [NextAgent | REST] | {
    send(NextAgent, request, beginWork(NextAgent));
    WaitSubprocess(NextAgent);
    if B(not REST = []) then { bcBeginWork(REST) }
  }

PG-rules:
  initialize <- true | {
    B(bagof(X, subscribed(X), Agents));
    bcProjectInformation(Agents);
    bcBeginWork(Agents);
    dropgoal(initialize);
    adoptz(created(process))
  }
```

Código 7. Protocolo de Iniciação – Especialista

```
BeliefUpdates:
  {  } AddAvailableModel(M) { availableModel(M) }

PC-rules:
  % Process received message about a functional requirement
  % Message format: funcRequirement(X: functional requirement,
  %     L: level of importance, N: new feature, T: new feature to team,
  %     C: new feature to client, M: mutable, R: related artifact models)
  message(_, inform, La, On, funcRequirement(X, L, N, T, C, M, R))
      <- true | {
    AddFuncRequirement(X, L);
    if B(N = 1) then { SetFRNew(X) };
    if B(T = 1) then { SetFRNewTeam(X) };
    if B(C = 1) then { SetFRNewClient(X) };
    if B(M = 1) then { SetFRMutable(X) };
    if B(not R=[]) then { addRelatedModels(X, R) }
  }

  % Process received message about an artifact model
  % Message format: artifactModel(M: artifact model, U: updaters, R: readers,
  %     P: parent model)
  message(_, inform, La, On, artifactModel(M, U, R, P)) <- true | {
    AddArtifactModel(M);
    addAMUpdaters(M, U);
    addAMReaders(M, R);
    if B(not P = 0) then { SetAMParent(M, P) }
  }

  % Process received message about an available artifact model
  % Message format: availableModel(M: artifact model)
  message(_, inform, La, On,  availableModels(M)) <- true | {
    if B(not M=[]) then { addAvailableModels(M) }
  }

  addAvailableModels(M) <- M = [Model|REST] | {
    AddAvailableModel(Model);
    if B(not REST=[]) then { addAvailableModels(REST) }
  }

  % Process received message about an artifact
  % Message format: artifact(A: artifact, O: owner, M: artifact model, S: status,
  %     P: related requirements)
  message(_, inform, La, On,  artifact(A, O, M, S, P)) <- true | {
    AddArtifact(A, O, M);
    SetArtifactStatus(A, S);
    if B(not P=[]) then { addRelatedRequirements(A, P) }
  }

  % Process received message about a time constraint
  % Message format: timeConstraint(R: role, T: time constraint)
  message(_, inform, La, On, timeConstraint(R, T)) <- true | {
    SetTimeConstraint(R, T)
  }
```

```
  % Request the time constraint for a set of roles
  % requestTimeConstraint(R: set of roles)
  requestTimeContraint(R) <- R = [NextRole|REST] | {
    if B(not timeConstraint(NextRole, _)) then {
      B(agent(TimeAgent, timeProvider));
      requestInformation(TimeAgent, timeConstraint, NextRole);
      B(timeConstraint(NextRole, _))
    };
    if B(not REST=[]) then { requestTimeContraint(REST) }
  }

  % Process received signal to begin the planning work
  % Message format: beginWork(Agent)
  message(_, request, La, On, beginWork(_)) <- true | {
    B(bagof(X, selfRole(X), Roles));
    requestTimeContraint(Roles);
    adopta(created(subprocess))
  }
```

Código 8. Protocolo de Solicitação de Artefato – Gerente de Projetos

```
BeliefUpdates:
  % Add an artifact need
  % AddArtifactNeed(A: agent, X: artifact, S: status)
  {  } AddArtifactNeed(A, X, S) { needingArtifact(A, X, S) }

  % Remove an artifact need
  % DropArtifactNeed(A: agent, X: artifact, S: status)
  { needingArtifact(A, X, S) }
  DropArtifactNeed(A, X, S)
  { not needingArtifact(A, X, S) }

  % Waiting for the answer of a previous query for the availability of an artifact
  {  }
  WaitingArtifactAvailable(Agent, Artifact, Status)
  { waitingArtifactAvail(Agent, Artifact, Status) }

  % Stop waiting for the answer of a previous query for the availability of an artifact
  {  }
  NotWaitingArtifactAvailable(Agent, Artifact, Status)
  { not waitingArtifactAvail(Agent, Artifact, Status) }

  % Sets the availability of an artifact
  {  }
  AvailableArtifact(Agent, Artifact, Status)
  { availableArtifact(Agent, Artifact, Status) }

  % Waiting for the answer of a previous query for the builder of an artifact
  {  }
  WaitingArtifactBuilder(Agent, Artifact, Status)
  { waitingArtifactBuilder(Agent, Artifact, Status) }

  % Stop waiting for the answer of a previous query for the builder of an artifact
  {  }
  NotWaitingArtifactBuilder(Agent, Artifact, Status)
  { not waitingArtifactBuilder(Agent, Artifact, Status) }

PC-rules:
  % Process a received message requesting an artifact
  % Message format: needArtifact(A: artifact, S: status)
  message(Agent, request, La, On, needArtifact(A, S)) <- true | {
    AddArtifactNeed(Agent, A, S)
  }

  % Query agents for the availability of an artifact
  % queryArtifactAvail(Agents: set of agents, A: artifact,
  %     S: status)
  queryArtifactAvail(Agents, A, S) <- Agents = [NextAgent|REST] | {
    if B(not availableArtifact(_, A, S)) then {
      send(NextAgent, request, artifactAvailable(A, S));
      WaitingArtifactAvailable(NextAgent, A, S)
    };
   if B(not REST=[]) then { queryArtifactAvail(REST, A, S) }
  }
```

```
% Process received message answering positively to a query for an available artifact
% Message format: artifactAvailable(A: artifact, S: status)
message(Sender, inform, La, On, artifactAvailable(A, S))
    <- waitingArtifactAvail(_, A, S) | {
  AvailableArtifact(Sender, A, S);
  B(bagof(X, waitingArtifactAvail(X, A, S) ,Agents));
  dropWaitingArtifactAvailable(Agents, A, S)
}

% Process received message answering negatively to a query for an available artifact
% Message format: artifactUnavailable(A: artifact, S: status)
message(Sender, inform, La, On, artifactUnavailable(A, S))
    <- waitingArtifactAvail(_, A, S) | {
  NotWaitingArtifactAvailable(Sender, A, S)
}

% Query agents if anyone can build an artifact
% queryArtifactBuilder(Agents: set of agents, A: artifact, S: status)
queryArtifactBuilder(Agents, A, S) <- Agents = [NextAgent|REST] | {
  if B(not availableArtifact(_, A, S)) then {
    send(NextAgent, request, artifactBuilder(A, S));
    WaitingArtifactBuilder(NextAgent, A, S)
  };
  if B(not REST=[]) then { queryArtifactBuilder(REST, A, S) }
}

% Process received message answering positively to a previous query for the builder
% of an artifact
% Message format: artifactBuilder(A: artifact model, S: status)
message(Sender, inform, La, On, artifactBuilder(A, S))
    <- waitingArtifactBuilder(_, A, S) | {
  AvailableArtifact(Sender, A, S);
  B(bagof(X, waitingArtifactBuilder(X, A, S), Agents));
  dropWaitingArtifactBuilder(Agents, A, S)
}

% Process received message answering negatively to a previous query for the builder
% of an artifact
% Message format: notArtifactBuilder(A: artifact, S: status)
message(Sender, inform, La, On, notArtifactBuilder(A, S))
    <- waitingArtifactBuilder(_, A, S) | {
  NotWaitingArtifactBuilder(Sender, A, S)
}

% Send a positive answer to a previous request for an artifact
% sendArtifactAvailable(Agent, A: artifact, S: status)
sendArtifactAvailable(Agent, A, S) <- needingArtifact(Agent, A, S) | {
  send(Agent, inform, artifactAvailable(A, S));
  DropArtifactNeed(Agent, A, S)
}

% Send a negative answer to a previous request for an artifact
% sendArtifactUnavailable(Agent, A: artifact, S: status)
sendArtifactUnavailable(Agent, A, S) <- needingArtifact(Agent, A, S) | {
  send(Agent, inform, artifactUnavailable(A, S));
  DropArtifactNeed(Agent, A, S)
}

dropWaitingArtifactAvailable(Agents, A, S) <- Agents = [NextAgent|REST] | {
  NotWaitingArtifactAvailable(NextAgent, A, S);
  if B(not REST=[]) then { dropWaitingArtifactAvailable(REST, A, S) }
}

dropWaitingArtifactBuilder(Agents, A, S) <- Agents = [NextAgent|REST] | {
  NotWaitingArtifactBuilder(NextAgent, A, S);
  if B(not REST=[]) then { dropWaitingArtifactBuilder(REST, A, S) }
}
```

Código 9. Protocolo de Solicitação de Artefato – Especialista

```
BeliefUpdates:
  { not needingArtifact(A, S) } SetNeedingArtifact(A, S) { needingArtifact(A, S) }
  { needingArtifact(A, S) } UnsetNeedingArtifact(A, S) { not needingArtifact(A, S) }
```

```
PC-rules:
  % Request an artifact
  % requestArtifact(A: artifact model, S: status)
  requestArtifact(A, S) <- true | {
    B(agent(PM, rProjMgr));
    if B(not needingArtifact(A, S)) then {
      send(PM, request, needArtifact(A, S));
      SetNeedingArtifact(A, S)
    }
  }

  % Process a received message querying if an artifact is available
  % Message format: artifactAvailable(A: artifact model, S: status)
  message(PM, request, La, On, artifactAvailable(A, S)) <- true | {
    if B(artifact(X) and aModel(X, A) and aStatus(X, S)) then {
      send(PM, inform, artifactAvailable(A, S))
    } else {
      send(PM, inform, artifactUnavailable(A, S))
    }
  }

  % Process a received message querying if the agent is the builder of an artifact
  % Message format: artifactBuilder(A: artifact model, S: status)
  message(PM, request, La, On, artifactBuilder(A, S)) <- true | {
    if B(tOutput(T, A, X) and tPerformer(T, R) and selfRole(R)
         and status(X, S) and subprocess(P) and member(task(T), P))
    then {
      send(PM, inform, artifactBuilder(A, S))
    } else { send(PM, inform, notArtifactBuilder(A, S)) }
  }

  % Process a received message stating the availability of an artifact
  % Message format: artifactAvailable(A: artifact model, S: status)
  message(PM, inform, La, On, artifactAvailable(A, S)) <- true | {
    B(agent(Agent, artifactLinker));
    requestInformation(Agent, artifactName, available(A, S));
    B(artifact(X) and aModel(X, A) and aStatus(X, S));
    UnsetNeedingArtifact(A, S)
  }

  message(PM, inform, La,On, artifactUnavailable(A, S)) <- true | {
    UnsetNeedingArtifact(A, S)
  }
```

Código 10. Protocolo de Entrega de Resultado – Gerente de Projetos

```
BeliefUpdates:
  % Add a created subprocess
  % AddSubprocess(A: agent, P: subprocess, T: task list)
  { not subprocess(A, _) }
  AddSubprocess(A, S)
  { subprocess(A, S), not waitingSubprocess(A) }

PC-rules:
  % Process received message with a created subprocess
  % Message format: subprocess(S: subprocess definition)
  message(Agent, inform, La, On, subprocess(S)) <- true | {
    AddSubprocess(Agent, S)
  }
```

Código 11. Protocolo de Entrega de Resultado – Especialista

```
PC-rules:
  % Send the created subprocess
  % sendSubprocess(S: subprocess)
  sendSubprocess(S) <- true | {
    B(agent(PM, rProjMgr));
    send(PM, inform, subprocess(S))
  }
```

Código 12. Protocolo de Finalização – Gerente de Projetos

```
bcFinalizationSignal(Agents, S) <- Agents = [NextAgent|REST] | {
    send(NextAgent, inform, finalizationSignal(S));
    if B(not REST=[]) then { bcFinalizationSignal(REST, S) }
  }
```

Código 13. Protocolo de Solicitação de Informação – Solicitante

```
PC-rules:
  % Asks for an information
  % requestInformation(A: service agent, Type: type of information)
  requestInformation(A, Type) <- true | {
    send(A, request, info(Type))
  }

  % requestInformation(A: service agent, Type: type of information, Params: parameters)
  requestInformation(A, Type, Params) <- true | {
    send(A, request, info(Type, Params))
  }

  % Process a received message with the duration of a task
  % Message format: info(tDuration, tDuration(T: task, D: duration))
  message(A, inform, La, On, info(tDuration, tDuration(T, D))) <- true | {
    SetTaskDuration(T, D)
  }

  % Process a received message with the time constraint for a role
  % Message format: info(timeConstraint, timeConstraint(R: role, T: time constraint))
  message(A, inform, La, On, info(timeConstraint, timeConstraint(R, T))) <- true | {
    SetTimeConstraint(R, T)
  }

  % Process a received message with the description of an artifact
  % Message format: info(artifactName, task(A: artifact, O: owner, M: model))
  message(Agent, inform, La, On, info(artifact, artifact(A, O, M, S))) <- true | {
    AddArtifact(A, O, M);
    SetArtifactStatus(A, S)
  }

  % Process a received message with the description of an artifact
  % Message format: info(artifactName, task(A: artifact, O: owner, M: model))
  message(Agent, inform, La, On, info(artifact, artifact(A, O, M))) <- true | {
    AddArtifact(A, O, M)
  }
```

Código 14. Protocolo de Solicitação de Informação – Agente de Serviço

```
PC-rules:
  % Process a received messages requesting an information
  % General message format: into(T: type of information, [ P: parameters ])
  message(Requester, request, La, On, info(T)) <- info(T, I) | {
    send(Requester, inform, info(T, I))
  }

  message(Requester, request, La, On, info(T, P)) <- info(T, I) | {
    send(Requester, inform, info(T, I))
  }
```

APÊNDICE C

Este apêndice apresenta a especificação completa do modelo de processo usado no estudo de caso.

Modelos de Artefato

```
% Caso de Negócio
artifactModel(amBizCase).
  amUpdater(amBizCase, rProjManager).
  amReader(amBizCase, rSystemAnalyst).
  amReader(amBizCase, rProjManager).
  amParent(amBizCase, amVision).

% Visão
artifactModel(amVision).
  amUpdater(amVision, rSystemAnalyst).
  amReader(amVision, rSystemAnalyst).
  amReader(amVision, rInterfaceDesigner).
  amReader(amVision, rArchitect).
  amReader(amVision, rTechDocumentor).
  amReader(amVision, rTestDesigner).

% Casos de Uso
artifactModel(amUseCase).
  amUpdater(amUseCase, rSystemAnalyst).
  amReader(amUseCase, rArchitect).
  amParent(amUseCase, amUseCaseModel).

% Arquitetura
artifactModel(amArchitecture).
  amUpdater(amArchitecture, rArchitect).
  amReader(amArchitecture, rArchitect).
  amReader(amArchitecture, rProgrammer).

% Class e
artifactModel(amClass).
  amUpdater(amClass, rProgrammer).
  amReader(amClass, rArchitect).
  amReader(amClass, rProgrammer).
  amParent(amClass, amClassModel).

% Sistem a
artifactModel(amSystem).
  amUpdater(amSystem, rProgrammer).
  amUpdater(amSystem, rDeployer).
  amReader(amSystem, rProgrammer).
  amReader(amSystem, rTechDocumentor).
  amReader(amSystem, rTester).
  amReader(amSystem, rDeployer).

% Modelo de Casos de Uso
artifactModel(amUseCaseModel).
  amUpdater(amUseCaseModel, rSystemAnalyst).
  amReader(amUseCaseModel, rSystemAnalyst).
  amParent(amUseCaseModel, amReqEspecification).

% Modelo de Classes
artifactModel(amClassModel).
  amUpdater(amClassModel, rArchitect).
  amReader(amClassModel, rArchitect).
  amParent(amClassModel, amArchitecture).
```

```
% Modelo de Interação
artifactModel(amInteractionModel).
  amUpdater(amInteractionModel, rArchitect).
  amReader(amInteractionModel, rArchitect).
  amParent(amInteractionModel, amArchitecture).

% Modelo de Interface do Usuário
artifactModel(amInterfaceModel).
  amUpdater(amInterfaceModel, rInterfaceDsesigner).
  amReader(amInterfaceModel, rInterfaceDsesigner).
  amReader(amInterfaceModel, rArchitect).
  amParent(amInterfaceModel, amArchitecture).

% Relatório de Teste de Classe
artifactModel(amClassTestReport).
  amUpdater(amClassTestReport, rProgrammer).
  amReader(amClassTestReport, rProgrammer).

% Relatório de Teste de Integração
artifactModel(amIntegrationTestReport).
  amUpdater(amIntegrationTestReport, rProgrammer).
  amReader(amIntegrationTestReport, rProgrammer).

% Manual do Usuário
artifactModel(amUserManual).
  amUpdater(amUserManual, rTechDocumentor).
  amReader(amUserManual, rTechDocumentor).

% Guia de Testes do Sistema
artifactModel(amTestGuide).
  amUpdater(amTestGuide, rTestDesigner).
  amReader(amTestGuide, rTestDesigner).
  amReader(amTestGuide, rTester).

% Relatório de Testes do Sistema
artifactModel(amTestReport).
  amUpdater(amTestReport, rTester).
  amReader(amTestReport, rTester).

% Relatório de Implantação do Sistema
artifactModel(amDeploymentReport).
  amUpdater(amDeploymentReport, rDeployer).
  amReader(amDeploymentReport, rDeployer).

% Subsistemas
artifactModel(amSubsystem).
  amUpdater(amSubsystem, rProgrammer).
  amReader(amSubsystem, rProgrammer).
  amParent(amSystem).

% Especificação de Requisitos
artifactModel(amReqEspecification).
  amUpdater(amReqEspecification, rSystemAnalyst).
  amReader(amReqEspecification, rSystemAnalyst).

% Lista de Riscos
artifactModel(amRiskList).
  amUpdater(amRiskList, rProjManager).
  amReader(amRiskList, rSystemAnalyst).
  amParent(amRiskList, amVision).
```

Atividades do Processo

```
% Redigir Visão
  task(tWriteVision).
    tPerformer(tWriteVision, rSystemAnalyst).
    tInput(tWriteVision, amBizCase, sFinished).
    tOutput(tWriteVision, amVision, sFinished).

  % Escrever Caso de Negócio
  task(tWriteBizCase).
    tPerformer(tWriteBizCase, rBizAnalyst).
    tOutput(tWriteBizCase, amBizCase, sFinished).
```

```
% Modelar Classes
task(tModelClass).
  tPerformer(tModelClass, rArchitect).
  tInput(tModelClass, amUseCase, sFinished).
  tOutput(tModelClass, amClassModel, sFinished).

% Implementar Classes
task(tImplementClass).
  tPerformer(tImplementClass, rProgrammer).
  tInput(tImplementClass, amClassModel, sFinished).
  tInput(tImplementClass, amArchitecture, sFinished).
  tOutput(tImplementClass, amClass, sRevised).

% Modelar Casos de Uso
task(tModelUseCase).
  tPerformer(tModelUseCase, rSystemAnalyst).
  tInput(tModelUseCase, amReqEspecification, sFinished).
  tOutput(tModelUseCase, amUseCase, sFinished).

% Levantar Requisitos
task(tElicitRequirements).
  tPerformer(tElicitRequirements, rSystemAnalyst).
  tInput(tElicitRequirements, amVision, sFinished).
  tOutput(tElicitRequirements, amReqEspecification, sFinished).

% Modelar Interações
task(tModelInteractions).
  tPerformer(tModelInteractions, rArchitect).
  tInput(tModelInteractions, amClassModel, sFinished).
  tOutput(tModelInteractions, amInteractionModel, sFinished).

% Projetar Arquitetura
task(tDesignArchitecture).
  tPerformer(tDesignArchitecture, rArchitect).
  tInput(tDesignArchitecture, amClassModel, sFinished).
  tInput(tDesignArchitecture, amInteractionModel, sFinished).
  tInput(tDesignArchitecture, amVision, sFinished).
  tInput(tDesignArchitecture, amInterfaceModel, sFinished).
  tOutput(tDesignArchitecture, amArchitecture, sFinished).

% Projetar Interface do Usuário
task(tDesignUserInterface).
  tPerformer(tDesignUserInterface, rInterfaceDsesigner).
  tInput(tDesignUserInterface, amVision, sFinished).
  tOutput(tDesignUserInterface, amInterfaceModel, sFinished).

% Testar Class es
task(tTestClass).
  tPerformer(tTestClass, rProgrammer).
  tInput(tTestClass, amClass, sRevised).
  tOutput(tTestClass, amClassTestReport, sFinished).
  tOutput(tTestClass, amClass, sFinished).

% Teste de Integraçãon
task(tIntegrationTest).
  tPerformer(tIntegrationTest, rProgrammer).
  tInput(tIntegrationTest, amClass, sFinished).
  tInput(tIntegrationTest, amSubsystem, sUpdated).
  tOutput(tIntegrationTest, amIntegrationTestReport, sFinished).
  tOutput(tIntegrationTest, amSubsystem, sFinished).
  tOutput(tIntegrationTest, amSystem, sUpdated).

% Escrever Documentação do Usuário
task(tWriteUserDoc).
  tPerformer(tWriteUserDoc, rTechDocumentor).
  tInput(tWriteUserDoc, amSystem, sRevised).
  tInput(tWriteUserDoc, amVision, sFinished).
  tOutput(tWriteUserDoc, amUserManual, sFinished).

% Projetar Testes
task(tDesignTests).
  tPerformer(tDesignTests, rTestDesigner).
  tInput(tDesignTests, amVision, sFinished).
  tInput(tDesignTests, amArchitecture, sFinished).
  tOutput(tDesignTests, amTestGuide, sFinished).
```

```
% Testar Sistema
task(tRunTests).
  tPerformer(tRunTests, rTester).
  tInput(tRunTests, amTestGuide, sFinished).
  tInput(tRunTests, amSystem, sUpdated).
  tOutput(tRunTests, amTestReport, sFinished).
  tOutput(tRunTests, amSystem, sRevised).

% Implantar Sistem a
task(tDeploySystem).
  tPerformer(tDeploySystem, rDeployer).
  tInput(tDeploySystem, amSystem, sRevised).
  tInput(tDeploySystem, amUserManual, sRevised).
  tOutput(tDeploySystem, amDeploymentReport, sFinished).
  tOutput(tDeploySystem, amSystem, sFinished).

% Implementar Subsistemas
task(tImplementSubsystems).
  tPerformer(tImplementSubsystems, rProgrammer).
  tInput(tImplementSubsystems, amClass, sFinished).
  tOutput(tImplementSubsystems, amSubsystem, sStarted).

% Integrar Subsistemas
task(tIntegrateSubsystems).
  tPerformer(tIntegrateSubsystems, rProgrammer).
  tInput(tIntegrateSubsystems, amSubsystem, sStarted).
  tOutput(tIntegrateSubsystems, amSubsystem, sUpdated).

% Levantar Riscos
task(tElicitRisks).
  tPerformer(tElicitRisks, rBizAnalyst).
  tOutput(tElicitRisks, amRiskList, sFinished).
```

REFERÊNCIAS BIBLIOGRÁFICAS

Ambler, S. W. (2004). **Modelagem Ágil: Práticas Eficazes para a Programação**. Porto Alegre: Bookman, 2004. 347p.

Beck, K. (2000). **Extreme Programming Explained: Embrace Change**. Reading, MA: Addison-Wesley, 2000.

Bellifemine, F., Bergenti, F., Caire, G., Poggi, A. (2005). JADE - A Java Agent Development Framework. In: Bordini, R. H., Dastani, M., Dix, J., Seghrouchni, A. F., eds. **Multi-Agent Programming**. New York: Springer-Verlag, 2005. cap.5, p.125-147.

Bordini, R. H., Hübner, J. F., Vieira, R. (2005). Jason and the Golden Fleece of agent-oriented programming. In: Bordini, R. H., Dastani, M., Dix, J., Seghrouchni, A. F., eds. **Multi-Agent Programming**. Springer-Verlag, 2005. cap.1, p. 3-37.

Bordini, R. H., Vieira, R., Moreira, A. F. (2001). Fundamentos de sistemas multiagentes. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 21, JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, 20, Fortaleza, 2001. **Anais**. Fortaleza: SBC, 2001, v.2, p.3-44.

Boutilier, C. (1996). Planning, Learning and Coordination in Multiagent Decision Processes. In: CONFERENCE ON THEORETICAL ASPECTS OF RATIONALITY AND KNOWLEDGE, 6, Amsterdam, 1996. **Proceedings**. San Francisco: Morgan Kaufmann Publishers Inc, 1996, p.195-210.

Bratko, I. (1986). **Prolog Programming for Artificial Intelligence**. Wokingham, England: Adisson-Wesley Pub. Co, 1986. 423p.

Bratman, M. E. (1987). **Intention, Plans and Practical Reason**. Cambridge, MA: Harvard University Press, 1987. 200p.

Bratman, M. E., Israel, D. J., Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. **Computational Intelligence**, v.4, n.3, p.349-355, 1988.

CMMI Product Team. (2002). **CMMI for Systems Engineering and Software Engineering (CMMI-SE/SW)**. Pittsburg, PA: Software Engineering Institute, Carnegie Mellon University, 2002.

Dastani, Mehdi. (2008). 2APL: A Practical Agent Programming Language. **Autonomous Agents and Multi-Agent Systems**, v.16, n.3, p.214-248, 2008.

Dastani, Mehdi, Hobo, D., Meyer, J. (2007). Practical Extensions in Agent Programming Languages. In: INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 6, 2007. **Proceedings**. ACM Press, 2007.

Fuggetta, A. (2000). Software process: A Roadmap. In: CONFERENCE ON THE FUTURE OF SOFTWARE ENGINEERING, 22, Limerick, 2000. **Proceedings**. New York: ACM Press, 2000, p.25-34.

Geffner, H. (2002). Perspectives on Artificial Intelligence Planning. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 18, Edmonton, 2002. **Proceedings**. Menlo Park: American Association for Artificial Intelligence, 2002, p.1013-1023.

Georgeff, M. P., Ingrand, F. F. (1989). Decision-making in an Embedded Eeasoning System. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 11, Detroit, 1989. **Proceedings**. Detroit, 1989, p.972-978.

Georgeff, M. P., Lansky, A. L. (1987). Reactive Reasoning and Planning. In: NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE, 6, Seatle. Seatle. **Proceedings**. Seatle, 1987, p.677-682.

Humphrey, W. S. (2000a). **The Personal Software Process (PSP)**. Technical report, CMU/SEI-2000-TR-022. ESC-TR-2000-022. 2000.

Humphrey, W. S. (2000b). **The Team Software Process (TSP)**. Technical report, CMU/SEI-2000-TR-023. ESC-TR-2000-023. 2000.

ISO/IEC (2003). International Organization for Standardization (ISO/IEC). **ISO/IEC 15504: Information Technology - Process Assessment, Part 1 to Part 5**. ISO/IEC Intermediate report, 2003.

Kruchten, P. (2000). **The Rational Unified Process: An Introduction**. 2.ed. Addison-Wesley, 2000. 298p.

Osterweil, L. J. (1987). Software Processes are Software Too. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 9, Los Alamitos, 1987. **Proceedings**. Los Alamitos: IEEE Computer Society Press, 1987, p.2-13.

Palazzo, L. A. M. (1997). **Introdução à Programação Prolog**. Pelotas: Editora da Universidade Católica de Pelotas, 1997. 194p.

Paula Filho, W. P. (2009). **Engenharia de Software: Fundamentos, Métodos e Padrões**. 3.ed. Rio de Janeiro: LTC, 2009. 1248p.

Pfleeger, S. L. (2004). **Engenharia de Software: teoria e prática**. 2.ed. São Paulo: Prentice Hall, 2004. 535p.

Pressman, R. S. (2005). **Software Engineering: A Practitioner's Approach**. 6.ed. McGraw-Hill, 2005.

Rao, A. S. (1996). AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In: WORKSHOP ON MODELLING AUTONOMOUS AGENTS IN A MULTI-AGENT WORLD, 7, London, 1996. **Proceedings**. London: Springer-Verlag, 1996, p.42-55.

Royce, W. (1998). **Software Project Management: A Unified Framework**. Boston: Addison-Wesley, 1998. 406p.

Russell, S. J., Norvig, P. (2002). **Artificial Intelligence: A Modern Approach**. 2.ed. Prentice Hall, 2002. 1132p.

Satler, B. T. (2010). **Seleção de Melhores Práticas de Engenharia de Software com Base em Parâmetros Extraídos do Ambiente do Problema**. Viçosa, 2010. 117p. Dissertação de Mestrado – Departamento de Informática – Universidade Federal de Viçosa.

Schwaber, K., Beedle, M. (2001). **Agile Software Development with Scrum**. Englewood Cliffs, NJ: Prentice Hall, 2001. 158p.

Soares, L. S. (2007). **Obtenção de Requisitos para Customização de Processo de Desenvolvimento de Software**. Viçosa, 2007. 77p. Dissertação de Mestrado – Departamento de Informática – Universidade Federal de Viçosa.

Sommerville, I. (2007). **Engenharia de Software**. 8.ed. São Paulo: Pearson Addison-Wesley, 2007. 552p.

The Standish Group. (1994). **The CHAOS report**. Dennis: The Standish Group, 1994. 8p.

Verner, L. (2004). BPM: The Promise and the Challenge. **Queue**, v.2, n.1, p.82-91, 2004.

Walczak, A., Braubach, L., Pokahr, A., Lamersdorf, W. (2006). Augmenting BDI Agents with Deliberative Planning Techniques. In: INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 5, INTERNATIONAL WORKSHOP ON PROGRAMMING MULTIAGENT SYSTEMS, 4, Hakodate, 2006. **Proceedings**. Heidelberg: Springer Berlin, 2006, p.113-127.

Wazlawick, R. S. (2004). **Análise e Projeto de Sistemas de Informação Orientados a Objetos**. Rio de Janeiro: Elsevier, 2004. 298p.

Weerdt, M. de, Mors, A., Witteveen, C. (2005). Multi-agent Planning: An Introduction to Planning and Coordination. In: EUROPEAN AGENT SUMMER SCHOOL, 7, Utrecht, 2005. **Handouts**. Utrecht, 2005, p.1-32.

Wooldridge, M. (2002). **An Introduction to Multiagent Systems**. John Wiley & Sons Ltd, 2002. 348p.

Zhao, X., Chan, K., Li, M. (2005). Applying Agent Technology to Software Process Modeling and Process-Centered Software Engineering Environment. In: ACM SYMPOSIUM ON APPLIED COMPUTING, New York, 2005. **Proceedings**. New York: ACM Press, 2005, p.1529-1533.