

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

**Catálogo de Padrões Arquiteturais:
Uma Visão Organizacional para
Sistemas Multiagentes**

Autor: Tainara Santos Reis
Orientador: Prof^ª. Dr^ª. Milene Serrano
Coorientador: Prof. Dr. Maurício Serrano

Brasília, DF
2018



Sumário

1	INTRODUÇÃO	3
2	CLASSIFICAÇÃO DE PADRÕES ARQUITETURAIS ORGANIZA- CIONAIS PARA SMA	7
2.1	Categorias <i>Mecanic</i> e <i>Organic</i>	7
2.2	Categorias <i>Markets</i> e <i>Hierarchies</i>	10
2.3	Critérios para classificação em categorias e subcategorias	11
2.4	Elementos para a descrição dos padrões arquiteturais	12
3	MASTER-SLAVE	13
4	BLACKBOARD	17
5	CONTRACT NET	23
5.1	FIPA ACL	25
6	MACRON	29
7	GENERALIZED PARTIAL GLOBAL PLANNING	35
	REFERÊNCIAS	41
	APÊNDICE A – IMPLEMENTAÇÃO DOS PADRÕES ARQUITE- TURAIS ORGANIZACIONAIS PARA SMA	43
A.1	<i>Contract Net</i>	43
A.1.1	Contexto	43
A.1.2	Preparação do ambiente	43
A.1.3	Classe <i>BookSellerAgent</i>	44
A.1.4	Classe <i>BookBuyerAgent</i>	48
A.1.5	Resultados da execução	51
A.2	<i>Master-Slave</i>	53
A.2.1	Contexto	53
A.2.2	Preparação do ambiente	53
A.2.3	Classe <i>Main</i>	53
A.2.4	Classes <i>MasterAgent</i> <i>ConcreteCashFlowAgent</i>	54
A.2.5	Classes <i>SlaveAgent</i> e <i>ConcreteSlaveAgent</i>	55
A.2.6	Resultados da execução	56

A.3	<i>Blackboard</i>	56
A.3.1	Contexto	56
A.3.2	Preparação do ambiente	56
A.3.3	Classe <i>Blackboard</i>	56
A.3.4	Classe <i>ManufacturerAgent</i>	58
A.3.5	Classe <i>Company</i>	59
A.3.6	Classe <i>SupplierAgent</i>	59
A.3.7	Classes <i>SupplierAgent1</i> e <i>SupplierAgent2</i>	60
A.3.8	Resultados da execução	62
A.4	<i>MACRON</i>	62
A.4.1	Contexto	62
A.4.2	Preparação do ambiente	63
A.4.3	Classe <i>Main</i>	63
A.4.4	Classe <i>QueryManagerAgent</i>	63
A.4.5	Classe <i>OrganizationalChartManagerAgent</i>	64
A.4.6	Classe <i>FunctionalManager1Agent</i>	65
A.4.7	Resultados da execução	65
A.5	<i>Generalized Partial Global Planning</i>	66

1 Introdução

A utilização de agentes de software constitui uma abordagem adequada para a concepção e a construção de sistemas complexos e distribuídos (JUNIOR et al., 2003). Tais agentes conhecem os interesses dos usuários e podem agir de forma autônoma em prol dos mesmos. Ao invés de exercer controle completo, o papel dos usuários dá-se, preferencialmente, de forma cooperativa com os agentes de software, de modo que esses últimos possam auxiliar esses usuários no cumprimento de suas metas (GREEN et al., 1997).

Agentes de software despertam o interesse de diversos pesquisadores e empresas de software (GREEN et al., 1997). Trata-se de uma área que está emergindo rapidamente dentro da TI (JENNINGS; WOOLDRIDGE, 1996), como um paradigma poderoso para projetar e desenvolver sistemas de software complexos (ZAMBONELLI; JENNINGS; WOOLDRIDGE, 2001).

Sistemas Multiagentes (SMA) encontram espaço em uma série de domínios de aplicação, como: comércio eletrônico, *design* de interface, jogos e gestão de processos industriais e comerciais complexos (JENNINGS; WOOLDRIDGE, 1996). Green et al. (1997) apontam outras áreas de aplicação como controle de tráfego aéreo, *data mining*, recuperação e gestão da informação, educação e assistentes digitais pessoais (*Personal Digital Assistants*, PDAs).

Uma das armadilhas apontadas por Wooldridge (2009, pág. 246) ocorre quando agentes interagem muito livremente ou de forma desorganizada. A dinâmica de SMA é complexa e, portanto, pode se tornar caótica. Para descobrir o que acontecerá em seguida no sistema, muitas vezes, é necessário executar o sistema repetidamente. O número de agentes também influencia em termos de complexidade para gerir de forma eficaz o sistema.

Independentemente do paradigma adotado, é habitual que desenvolvedores comecem a codificar sem uma arquitetura formal clara e bem definida (RICHARDS, 2015). O resultado desta prática, frequentemente, é uma coleção de módulos de código-fonte desorganizados, que não possuem papéis, responsabilidades e relacionamentos claros uns com os outros.

Segundo Richards (2015), aplicações que não possuem uma arquitetura formal são geralmente “acopladas, quebradiças, difíceis de mudar e sem uma visão ou direção clara”. Questões básicas sobre implantação, manutenção e escalabilidade desses sistemas são difíceis de definir.

Uma alternativa é adotar padrões arquiteturais. Estes ajudam a definir as características básicas e o comportamento de uma aplicação (RICHARDS, 2015). O arquiteto de software deve conhecer as características, pontos fortes e fracos de cada padrão arquitetural, afim de optar por aquele que atenda às suas necessidades e aos seus objetivos comerciais específicos.

De acordo com Shaw (1996), descrições uniformes das arquiteturas, quando disponíveis, são instrumentos para simplificar esta escolha. Segundo Avgeriou e Zdun (2005), o processo de descrever, encontrar e aplicar padrões arquiteturais na prática ainda é largamente *ad-hoc* e não sistemático. Isto se deve a várias questões que ainda não foram resolvidas como, por exemplo, em relação à classificação ou catalogação de padrões que podem ser usados por arquitetos de software.

Cabe destacar que existe um paralelo entre a complexidade das organizações e os SMA. Organizações são entidades complexas formadas para superar várias limitações de instâncias individuais, tais como limitações cognitivas, físicas, temporais e institucionais (AART, 2004). Neste sentido, são utilizados conceitos, métodos e técnicas de *design* organizacional humano como princípios arquitetônicos para SMA (AART, 2004).

Uma organização fornece uma estrutura para as interações dos agentes através da definição de papéis, expectativas de comportamento e relações de autoridade (SYCARA, 1998). As organizações são, em geral, conceitualizadas em termos de sua estrutura, isto é, o padrão de informações e relações de controle que existem entre os agentes e a distribuição das capacidades de resolução de problemas entre eles (SYCARA, 1998).

Na resolução cooperativa de problemas, por exemplo, uma estrutura fornece a cada agente uma visão de alto nível de como o grupo soluciona problemas (SYCARA, 1998). Portanto, a estrutura organizacional impõe restrições sobre as formas como os agentes se comunicam e se coordenam. Neste contexto, os padrões arquiteturais para SMA baseados em estruturas organizacionais, podem fornecer soluções práticas para o desenvolvimento de SMA complexos, sendo assim, são objetos de estudo deste trabalho.

O catálogo a seguir baseia-se no trabalho de Argente, Julian e Botti (2006) intitulado *Multi-Agent System Development Based on Organizations*, bem como em outros estudos realizados e conhecimentos adquiridos pela autora dessa monografia junto à massa de artigos investigados e apresentados na Seção anterior. A Figura 1 representa as classificações de cada padrão dentro de categorias e subcategorias definidas, principalmente, pelos autores do referido artigo.

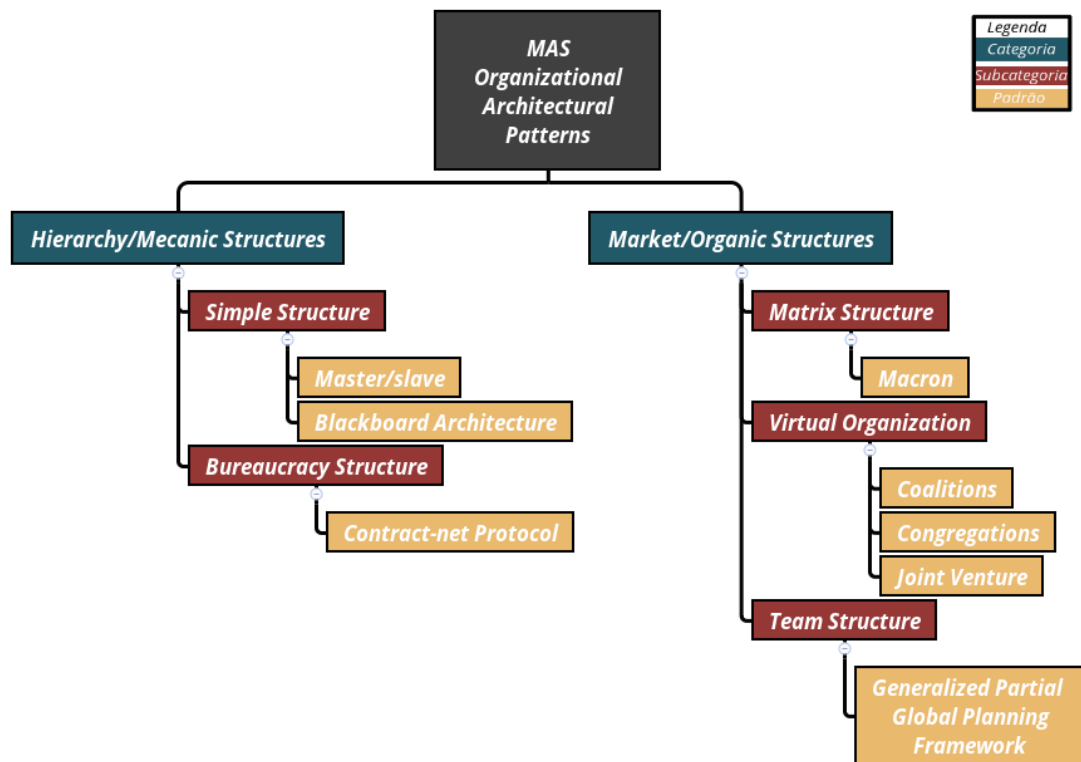


Figura 1 – *MAS - Organizational Architectural Patterns*. Fonte: Autora. Adaptado de: Argente, Julian e Botti (2006).

2 Classificação de padrões arquiteturais organizacionais para SMA

Malone (1990), Sycara (1998), Argente, Julian e Botti (2006) abordam, em seus trabalhos, estruturas de coordenação de SMA a partir do emprego da Teoria Organizacional. Após a análise das categorias de cada trabalho, observou-se que as mesmas possuíam o mesmo conceito, diferenciando-se apenas pelo nome e pelos autores dos trabalhos que as citam. São, portanto, equivalentes e sinônimos para este trabalho: *Market* e *Organic*; *Hierarchy* e *Mecanic*.

A seguir, tais categorias serão apresentadas em detalhe e fomentaram as categorias e subcategorias, as quais os padrões definidos no catálogo foram classificadas. A Figura 1 ilustra a estrutura das categorias e subcategorias.

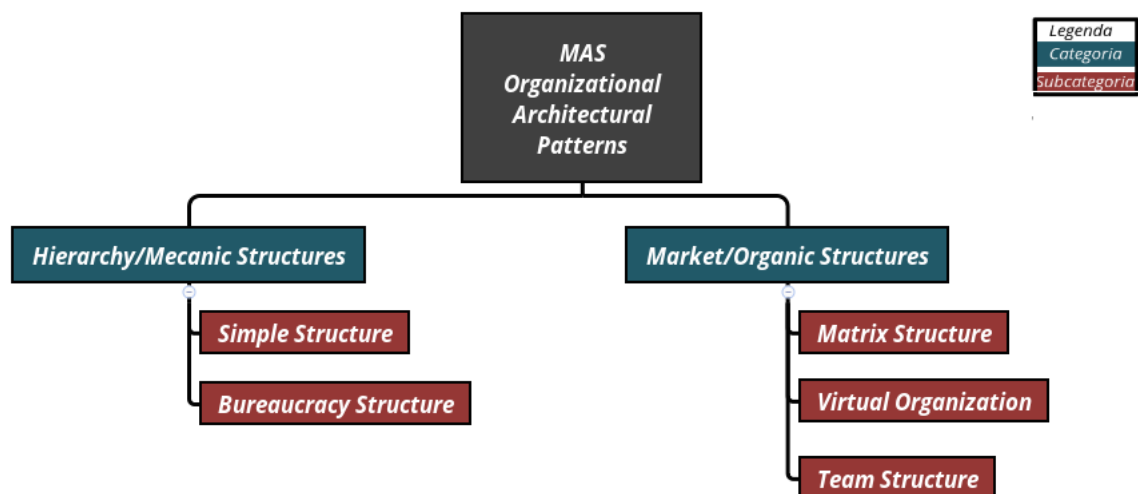


Figura 2 – Categorias e Subcategorias. Fonte: Autora.

2.1 Categorias *Mecanic* e *Organic*

Argente, Julian e Botti (2006) realizam uma revisão das organizações humanas mais conhecidas para que possam ser aplicadas no campo de SMAs. Tais organizações podem ser usadas como base para descrever papéis, padrões e conexões que colaboram com a definição de arquiteturas para SMAs. A Teoria Organizacional (*Organizational Theory*) analisa como funcionam as organizações, suas características principais, as características mais relevantes de seus membros, seus papéis, as relações dos membros, a cadeia de comando, as regras e normas que regem a organização, dentre outros aspectos.

Nesse sentido, Argente, Julian e Botti (2006) definem dois tipos de organizações

humanas: mecânicas e orgânicas. Nas organizações mecânicas, as tarefas são definidas com precisão, e são divididas em partes separadas e especializadas. Existe uma hierarquia de autoridade rígida. Os processos de conhecimento e raciocínio também são centralizados no topo da hierarquia. As comunicações são, principalmente, verticais (entre supervisores e subordinados). São três exemplos deste tipo de organização: *Simple Structure*, *Bureaucracy Structure* e *Matrix Structure*. A Figura 3 apresenta características de cada uma dessas organizações humanas.

		Human Organizations				
		Simple	Bureaucracy	Matrix	Team	Virtual
Organizational features	Centralized capture of decisions	√	√	√	X	X
	Work specialization	X	√	√	√	X
	Generalist members	X	X	X	√	√
	Departmentalization	X	√	√	X	X
	Span of control	√	X	√	X	X
	Formalization of tasks	X	√	√	X	X
	Coordination between specialists	X	√	√	√	X
	Authority	√	√	√	X	X
	Chain of command	X	√	√	X	X
	Several direct managers	X	X	√	X	X
	Department goals	X	√	X	X	X
	Organization goals	√	X	√	√	√
	Shared information	X	X	X	√	√
	Flexibility	X	X	X	√	√
	Business functions outsourcing	X	X	X	X	√

Figura 3 – Características de organizações humanas. Fonte: [Argente, Julian e Botti \(2006\)](#).

Nas organizações orgânicas, as tarefas são ajustadas e redefinidas por meio de trabalho colaborativo em grupos ([ARGENTE; JULIAN; BOTTI, 2006](#)). Existem menos níveis de autoridade e controle, de modo que o controle do conhecimento e das tarefas são distribuídos. Todos os membros devem contribuir para a tarefa comum do departamento. As comunicações são principalmente horizontais entre membros do mesmo departamento, ou mesmo entre diferentes departamentos. Desta forma, eles podem oferecer respostas rápidas e flexíveis. São exemplos deste tipo de organização: *Team Structure* e *Virtual Organizations*.

A subcategoria ***Simple Structure*** representa uma organização com poucos departamentos, onde um indivíduo centraliza a captura de decisões ([ARGENTE; JULIAN; BOTTI, 2006](#)). Esta estrutura apresenta um alto grau de controle, pois o gerente dirige um grande número de pessoal. Este tipo de organização é, frequentemente, empregado em pequenas empresas, onde o gerente e proprietário são a mesma pessoa, bem como em grandes empresas quando o controle é centralizado em um indivíduo.

Esta organização é simples: as responsabilidades são claras, as comunicações são

diretas e a captura de decisões e sua execução é rápida. No entanto, é recomendada somente para organizações pequenas, pois há pouca formalização e o gerente deve lidar com muitas informações. Já que tudo depende de uma única pessoa, se essa pessoa falhar ou tomar uma decisão errada, a empresa pode ser prejudicada (ARGENTE; JULIAN; BOTTI, 2006).

A subcategoria ***Bureaucracy Structure*** é caracterizada, principalmente, por tarefas operacionais e de rotina com alta especialização. Há também muitas regras e regulamentos formalizados. Por haver diversos departamentos, é ocasionado um baixo nível de controle, com o qual os gerentes controlam um pequeno grupo de pessoas ou departamentos (ARGENTE; JULIAN; BOTTI, 2006).

Há uma autoridade central e a tomada de decisões segue uma cadeia de comando. Entre suas vantagens, esta estrutura permite que as atividades padrão sejam realizadas de forma muito eficaz. Os especialistas são reunidos nos mesmos departamentos, facilitando as comunicações entre os funcionários. Graças à ampla presença de regras e regulamentos e à padronização das operações, a tomada de decisões é centralizada em gerentes executivos (ARGENTE; JULIAN; BOTTI, 2006).

No entanto, a alta especialização das tarefas pode criar conflitos em unidades ou departamentos. Os gerentes podem estar mais focados em alcançar seus objetivos individuais do que os objetivos gerais da organização (ARGENTE; JULIAN; BOTTI, 2006).

Além disso, os gerentes podem estar excessivamente concentrados em seguir as regras, o que dificulta suas decisões quando confrontados com novas situações. Portanto, esses tipos de organizações tem dificuldades em responder às mudanças externas (ARGENTE; JULIAN; BOTTI, 2006).

A subcategoria ***Matrix Structure*** possui dois supervisores: o gerente do departamento funcional e o gerente do produto. Portanto, existem duas cadeias de controle. Esse tipo de estrutura é muito comum em empresas de engenharia e gerenciamento de projetos. Facilita a coordenação entre funcionários quando há numerosas atividades complexas e interdependentes (ARGENTE; JULIAN; BOTTI, 2006).

Outra vantagem consiste na melhoria das comunicações e ampliação da flexibilidade. Também reduz a possibilidade dos membros se concentrarem nos objetivos individuais de seus departamentos mais do que nos objetivos gerais da organização. Da mesma forma, facilita a localização efetiva de especialistas (ARGENTE; JULIAN; BOTTI, 2006).

No entanto, pode haver confusão na tomada de decisões, pois existem duas cadeias de comando. Além disso, esta estrutura promove lutas de poder e pode criar tensão, que pode ser diminuída usando técnicas burocráticas, como, por exemplo, uma formalização maior das regras (ARGENTE; JULIAN; BOTTI, 2006).

A subcategoria **Team Structure** elimina barreiras departamentais e descentraliza a tomada de decisão. Equipes ou grupos representam um sistema com vários atores que possuem um objetivo comum: a realização da tarefa global do sistema (ARGENTE; JULIAN; BOTTI, 2006).

Esta tarefa é dividida em sub-tarefas, que são atribuídas aos membros mais qualificados do grupo. Além disso, os membros compartilham todas as informações e estão em constante comunicação uns com os outros. A coordenação entre atores é obtida usando decisões e planos mutuamente aceitos (ARGENTE; JULIAN; BOTTI, 2006).

A subcategoria **Virtual Organization** consiste em uma empresa que terceiriza suas principais funções comerciais. Várias redes de contato são criadas, o que permite que a empresa contrate funções comerciais que os gerentes acreditam que possam ser feitas por outras empresas de uma maneira melhor ou a um custo menor. Esta estrutura oferece flexibilidade, mas reduz o controle de gerenciamento em partes fundamentais da organização (ARGENTE; JULIAN; BOTTI, 2006).

2.2 Categorias *Markets* e *Hierarchies*

Malone (1990, pág. 61) considera três processos fundamentais de coordenação de SMA:

- **Ajuste mútuo:** trata-se do modo mais simples de coordenação, no qual dois ou mais agentes compartilham seus recursos visando atingir um objetivo comum. Nenhum agente tem controle prévio sobre os outros, e a tomada de decisões é um processo conjunto;
- **Supervisão direta:** ocorre quando dois ou mais agentes já estabeleceram um relacionamento em que um agente tem algum controle sobre os outros. Nesta forma de coordenação, o supervisor controla o uso de recursos compartilhados - como trabalho humano, tempo e dinheiro do processo de computador - pelos subordinados, e também pode prescrever certos aspectos de seus comportamentos, e
- **Padronização:** ocorre quando o supervisor estabelece procedimentos padrão para subordinados a serem seguidos em várias situações. Os procedimentos de rotina em empresas ou em software são exemplos de coordenação por meio da padronização.

Por meio desses processos fundamentais de coordenação podem ser elaborados sistemas de coordenação mais sofisticados (MALONE, 1990, pág. 63). São eles:

- **Markets/Mercados:** podem ser considerados uma forma de organização baseada em ajuste mútuo. Os agentes, cada um dos quais controla recursos escassos - como,

por exemplo, trabalho, matéria-prima, bens e dinheiro, concordam em compartilhar alguns de seus respectivos recursos para alcançar o objetivo mútuo. Os recursos são trocados, com ou sem custos explícitos. Uma vez que um contrato foi feito, existe um acordo em que o “comprador” se torna o supervisor do fornecedor, e

- **Hierarchies/Hierarquias:** baseiam-se em processos de supervisão direta. Um grande grupo pode ser dividido em subgrupos se a maior parte da transferência de informações necessária pode ocorrer em subgrupos e se as poucas interações entre subgrupos podem ser tratadas por supervisores. Nesse caso, pode-se utilizar como estratégia a hierarquia. Os subgrupos podem ser coordenados por ajuste mútuo ou por controle hierárquico, dependendo do domínio do aplicativo e das características da tarefa.

Sycara (1998), de modo semelhante, apresenta *Market* e *Hierarchy* como exemplo de organizações que são exploradas na literatura. Complementando o conceito de *Hierarchies*, Sycara (1998) define que esta estrutura é formada por uma autoridade para tomada de decisão e controle concentrada em um único solucionador de problemas (ou grupo especializado) em cada nível da hierarquia. A interação é através da comunicação vertical do agente superior ao subordinado e vice-versa; de modo que agentes superiores exerçam controle sobre recursos e tomada de decisões.

Na estrutura *Market*, o controle é distribuído aos agentes que competem por tarefas ou recursos através de lances e mecanismos que envolvem contratos. Os agentes interagem através de uma variável, como o custo, que é usado para avaliar os serviços (SYCARA, 1998).

2.3 Critérios para classificação em categorias e subcategorias

Baseando-se nos conceitos abordados até o momento, foram elencados critérios para classificar as estruturas organizacionais dos padrões arquiteturais identificados para que sejam catalogados. Os critérios para classificar uma estrutura organizacional como *Hierarchy/Mecanic* são:

- Há uma autoridade - ou seja, um agente ou um grupo de agentes superior - para a tomada de decisão e o controle de recursos;
- A comunicação dá-se verticalmente; da autoridade para o subordinado e vice-versa, e
- A autoridade pode prescrever aspectos do comportamento dos subordinados.

Os critérios para classificar uma estrutura organizacional como *Markets/Organic* são:

- Nenhum agente possui controle sobre os demais;
- Dois ou mais agentes compartilham seus recursos visando atingir um objetivo comum;
- Os agentes competem por tarefas ou recursos através de lances ou contratos, e
- A tomada de decisões ocorre mutuamente.

2.4 Elementos para a descrição dos padrões arquiteturais

Os elementos fundamentais para descrever os padrões arquiteturais identificados neste catálogo são:

Nome do padrão: nome do padrão encontrado na literatura;

Referências: referências para os trabalhos que descrevem o padrão;

Categoria: classificação do padrão em *Market* - e subcategorias *Simple Structure* e *Bureaucracy Structure* -, e *Hierarchy* - com subcategorias *Matrix Structure*, *Virtual Organization* e *Team Structure*;

Problema: problema em que o padrão busca solucionar, descrevendo-se seus objetivos;

Solução: instruções descrevendo como o problema pode ser resolvido;

Protocolos associados: este elemento é opcional. Caso o padrão esteja implementado em alguma plataforma acessível, os protocolos ou bibliotecas associados serão referenciados;

Modelagem: diagramas disponibilizados pela literatura representando o padrão. A notação utilizada deve ser apropriada para SMA como, por exemplo, diagrama de sequência UML ([LARMAN, 2002](#));

Exemplo: quando disponível na literatura, será apresentado um exemplo prático do padrão arquitetural, podendo conter trechos de código e figuras que torne o exemplo mais didático, e

Implementação: a descrição do padrão poderá conter mais um elemento: sua implementação. A implementação dependerá da viabilidade tecnológica para que seja cumprido e preenchido para cada padrão.

3 *Master-Slave*

Este padrão apresenta um sistema regido por um mestre capaz de obter informações de agentes do grupo, criar planos e atribuir tarefas a agentes individuais, a fim de assegurar a coerência global.

Nome do padrão: *Master-Slave*.

Referências: Aridor e Lange (1998).

Categoria: *Simple structure*.

Problema: ocorre quando um agente mestre, (ou *Master*), precisa executar uma tarefa em paralelo com outras tarefas para as quais é responsável. Ocorre também quando este agente deseja executar uma tarefa em um destino remoto.

Solução: Quatro classes participam deste padrão. A Figura 4 ilustra suas relações estruturais.

- **Master:** define um esqueleto de um agente mestre, usando métodos abstratos para serem substituídos na classe *Concrete-Master*.
- **Slave:** define um esqueleto de um agente escravo, usando métodos abstratos para serem sobrescritos pela classe *ConcreteSlave*.
- **ConcreteMaster:** implementa métodos abstratos da classe *Master*.
- **ConcreteSlave:** implementa métodos abstratos da classe *Slave*.

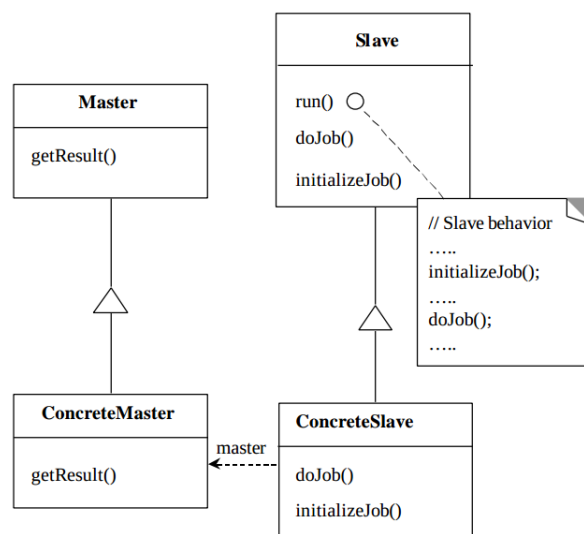


Figura 4 – Participantes do padrão *Master-Slave*. Fonte: Aridor e Lange (1998)

Primeiramente, o agente *Master* cria um agente *Slave*. Este, move-se para um *host* remoto e executa a tarefa para o qual foi designado. Em seguida, retorna com o resultado da tarefa para o agente *Master*.

Modelagem: este padrão pode ser representado pela Figura 5.

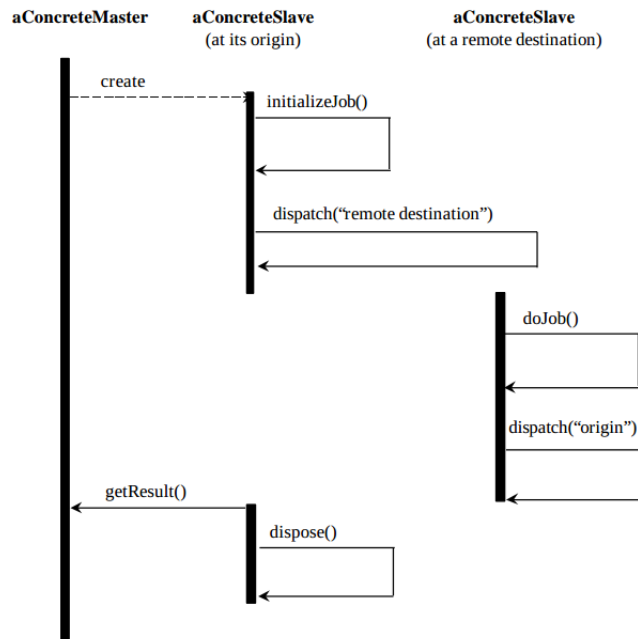


Figura 5 – Diagrama de sequência do padrão *Master-Slave*. Fonte: Aridor e Lange (1998).

Exemplo: O exemplo a seguir baseia-se em um aplicativo que fornece uma GUI para inserir dados e exibir os resultados intermediários de uma tarefa específica a ser realizada remotamente.

Com um único agente para fornecer a GUI e executar essa tarefa, não será possível manter a GUI depois que o agente viajou de sua origem para um destino remoto. A alternativa é adotar um agente escravo para que se mova para outro destino, execute a tarefa atribuída, envie os resultados intermediários e, finalmente, retorne com o resultado da tarefa ao agente mestre. Este, por sua vez, exibe o resultado para seu cliente.

A ideia chave do padrão é usar classes abstratas, mestre e escravo, para localizar as partes invariantes de delegar uma tarefa entre agentes mestres e escravos, consistindo em três principais comportamentos: (i) despachar um escravo de um lado para outro para outros destinos; (ii) iniciar a execução da tarefa, e (iii) lidar com exceções ao executar a tarefa.

Os agentes mestres e escravos são definidos como subclasses de mestre e escravo. Apenas partes variáveis, como a descrição de como a tarefa deve ser executada e como o agente mestre deve lidar com o resultado da tarefa, são implementadas. Na

prática, a classe *Master* possui um método abstrato *getResult()* para definir como lidar com o resultado da tarefa.

A classe *Slave* tem dois métodos abstratos, *initializeJob()* e *doJob()*, que definem as etapas de inicialização a serem executadas antes que o agente viaje para um novo destino e a tarefa a ser executada, respectivamente. Ambas as classes são definidas em termos desses métodos. O código a seguir mostra a classe *Slave* implementada como um *Aglet*.

```
1 public abstract class Slave extends Aglet {
2     Object result = null
3
4     public void onCreation(Object obj) {
5         // Called when the slave is created. Gets the
6         // remote destination, a reference to the master
7         // agent, and other specific parameters.
8     }
9
10    public void run () {
11        // At the origin:
12        initializeJob ();
13        dispatch(destination); // Goes to destination
14        // At the remote destination:
15        doJob(); // Starts on the task.
16        result = ...;
17        // Returns to the origin.
18        // Back at the origin.
19        // Delivers the result to the master and dies.
20        dispose();
21    }
```

Implementação: a fim de demonstrar este padrão, é utilizado um contexto de fechamento de caixa de uma loja. Todos os detalhes da implementação, configuração de ambiente e passos para execução são descritos no Apêndice [A.2](#).

4 *Blackboard*

O padrão arquitetural *Blackboard* (ou Quadro-Negro) tem sido amplamente utilizado para enfrentar problemas com características de incerteza (DONG; CHEN; JENG, 2005). Muitas vezes, não há solução algorítmica direta para problemas dessa natureza e a melhor solução trata-se de uma aproximação.

Esta arquitetura se baseia em um repositório central, onde agentes leem e escrevem em um quadro-negro, e um agente mestre controla o que é lido e escrito por estes agentes. As aplicações deste padrão podem ser encontradas em diferentes tipos de sistemas, como sistemas de comunicação, mobilidade e coordenação.

Nome do padrão: *Blackboard* ou Quadro-Negro.

Referências: Dong, Chen e Jeng (2005), Weiss (1999), Ito e Salleh (2000).

Categoria: *Simple Structure*.

Problema: é adotado em problemas com características de incerteza (DONG; CHEN; JENG, 2005), onde não há solução algorítmica direta para o problema e a melhor solução trata-se de uma aproximação. É adotado também para possibilitar o acesso paralelo a dados e a execução simultânea de tarefas.

Solução: Weiss (1999) apresenta a metáfora a seguir para demonstrar este padrão.

"Imagine um grupo de especialistas humanos ou de agentes sentados ao lado de um grande quadro-negro. Os especialistas estão trabalhando cooperativamente para resolver o problema, usando o quadro-negro como local de trabalho para desenvolver a solução. A resolução de problemas começa quando o problema e a data inicial estão escritos no quadro-negro. Os especialistas observam o quadro-negro, procurando uma oportunidade para aplicar seus conhecimentos à solução em desenvolvimento. Quando um especialista encontra informações suficientes para fazer uma contribuição, ele registra a contribuição no quadro-negro. Esta informação adicional pode permitir que outros especialistas apliquem seus conhecimentos. Este processo de adicionar contribuições para o quadro-negro continua até o problema ter sido resolvido." (WEISS, 1999)

Esta metáfora captura uma série de características importantes deste padrão, cada uma das quais são descritas a seguir.

- **Independência da experiência:** os especialistas (chamados de fontes de conhecimento, *Knowledge Sources* ou KSs) não são treinados para trabalhar unicamente com esse grupo específico de especialistas. Cada um é especialista

em alguns aspectos do problema e pode contribuir para a solução independentemente da combinação particular de outros especialistas na sala.

- **Diversidade em técnicas de resolução de problemas:** o algoritmo interno de representação e inferência adotado por cada KS é oculto;
- **Representação flexível da informação do quadro-negro:** não há quaisquer restrições prévias sobre as informações que podem ser colocadas no quadro-negro;
- **Linguagem de interação comum:** os KS devem ser capazes de interpretar corretamente as informações gravadas no quadro-negro por outros KSs. Pode haver tanto uma representação compreensível apenas entre alguns KSs e uma representação genérica compreensível por todos os KSs;
- **Ativação baseada em eventos:** os KSs são acionados em resposta ao quadro-negro e aos eventos externos. Os eventos do quadro negro incluem a adição de novas informações ao mesmo, uma alteração na informação existente ou a remoção de informações existentes. Cada KS informa o sistema de quadro-negro sobre o tipo de eventos em que ele está interessado. O sistema de quadro-negro grava esta informação e considera diretamente o KS a ser ativado sempre que esse tipo de evento vier a ocorrer;
- **Necessidade de controle:** há um componente de controle que é separado dos KSs e é responsável por gerenciar o fluxo da resolução de problemas. O componente de controle pode ser visto como um especialista na resolução de problemas. Ele analisa o benefício geral das contribuições possíveis pelos KSs para a resolução do problema em questão. Ao ser finalizada a execução de um KS, o componente de controle ativa o KS mais apropriado para execução. Ao ser disparado, o KS utiliza seus conhecimentos para avaliar a qualidade e a importância de sua contribuição. Cada KS disparado informa o componente de controle sobre a qualidade e os custos associados à sua contribuição, sem realmente realizar o trabalho para calcular a contribuição. O componente de controle usa essas estimativas para decidir como proceder, e
- **Geração de solução incremental:** KSs contribuem para a solução conforme apropriado, às vezes refinando, às vezes contradizendo, e às vezes iniciando uma nova linha de raciocínio.

Há, portanto, três componentes principais neste padrão (DONG; CHEN; JENG, 2005):

1. **Fontes de conhecimento (ou *Knowledge Sources*):** cada KSs é um especialista na resolução de certos aspectos do problema geral. Uma vez que ele encontra a informação que precisa no quadro-negro, pode prosseguir sem qualquer ajuda de outras fontes de conhecimento;

2. *Blackboard* (ou **Quadro-negro**): o quadro-negro é a fonte de todos os dados em que a fonte de conhecimento irá operar e é o destino para todas as conclusões a partir de uma fonte de conhecimento, e
3. **Controle**: é um gerente que considera cada solicitação feita à fonte de conhecimento. O controle é responsável por abordar o quadro-negro em termos do que a fonte de conhecimento pode contribuir e o efeito que a contribuição pode ter sobre a solução a ser proposta. O controle tenta manter a resolução de problemas dentro do fluxo correto, de modo a garantir que todos os aspectos cruciais do problema recebam atenção e a equilibrar a importância estabelecida das contribuições dos diferentes especialistas. Os componentes de controle também são fontes de conhecimento, entretanto, se concentram no processo de solução de problemas de domínio específicos.

Modelagem: A Figura 6 representa a arquitetura básica deste padrão.

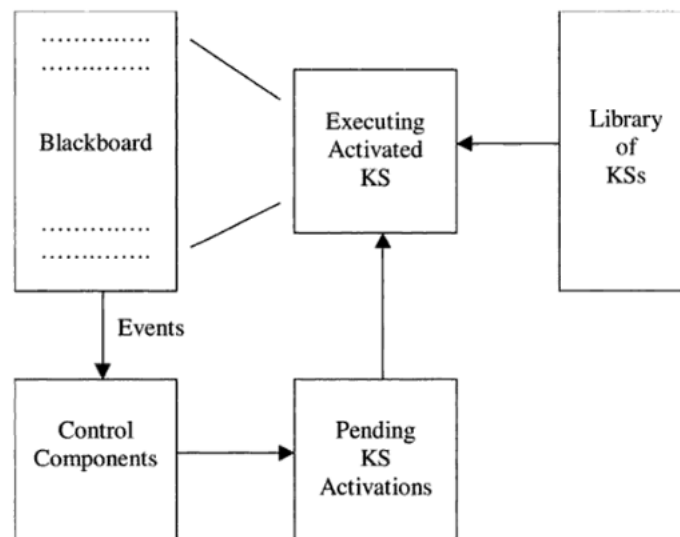


Figura 6 – A arquitetura básica de um sistema *Blackboard*. Fonte: Weiss (1999).

Exemplo: De modo a exemplificar uma utilização do padrão *Blackboard*, é apresentado o *Collaborative Supply Chain System* (CSCS) descrito por Ito e Salleh (2000). Trata-se de um sistema de gerenciamento de cadeia de suprimentos compreendido em uma rede integrada de membros, são eles: fornecedores, fábricas, armazéns, centros de distribuição e varejistas. Devido à complexidade desta rede, toda a cadeia de processos logísticos necessita ser gerenciada.

A colaboração entre os membros desempenha um papel crítico para implementar um sistema eficaz, mas sua implementação não seria fácil apenas com o mecanismo convencional de compartilhamento de informações. O intuito é proporcionar uma coordenação mais rápida e mais visível entre uma empresa, seus clientes e forne-

cedores. O CSCS é projetado para aumentar a velocidade e a certeza da oferta de suprimentos.

Conforme mostrado na Figura 7, cada agente interage e compartilha informações através de um quadro-negro. O quadro-negro regula e fornece informação a cada membro do sistema, são eles: *Supplier Agent* (SA), *Manufacturer Agent* (MA), *Distributor Agent* (DA) e *Customer Agent* (CA). A colaboração destes membros e a introdução de técnicas de negociação são desempenhadas por duas redes:

- *Intranet*: pode ser usada de modo a fornecer serviços de comunicação interna para obter melhores resultados do que os meios convencionais de acesso e transferência de dados. Por meio do *Manufacturer Agent*, os usuários acessam informações diárias, e a empresa pode facilmente exibir informações críticas de usuários externos. Basicamente, o *Manufacturer Agent* atua no controle de inventário regulando os níveis de inventário e negocia com os outros agentes para facilitar a circulação de materiais. A distribuição destes materiais é designada pelo *Distributor Agent*, e
- *Internet*: é uma base unificada composta por todos os membros do sistema e que permite que os usuários se comuniquem para as trocas de informações. Por exemplo, a *Internet* pode ser usada para oferecer a oportunidade de comparar fornecedores, escolher um fornecedor adequado e garantir que o fornecedor selecionado satisfaça os requisitos.

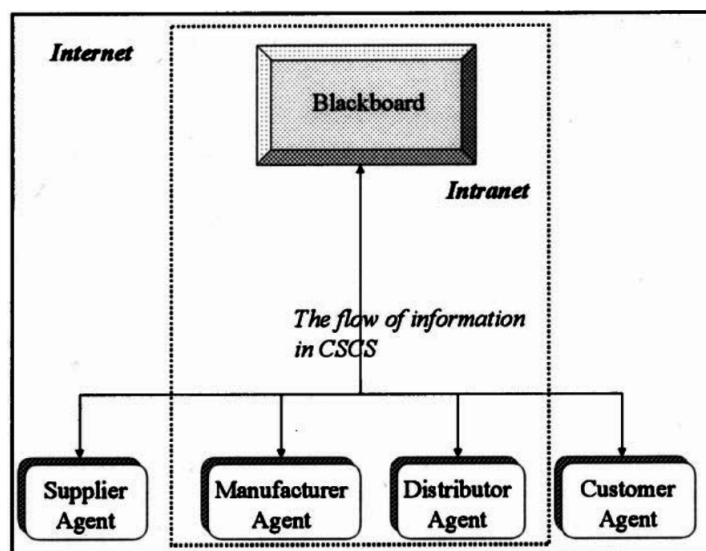


Figura 7 – Interação entre agentes no CSCS. Fonte: Ito e Salleh (2000).

As empresas são obrigadas a cumprir as ordens dos clientes mesmo quando é difícil fazê-lo. Elas devem responder às ordens de forma rápida e eficiente dentro do tempo limitado e disponível para atender às solicitações de suprimentos de seus clientes.

Ordens que surgem de modo imprevisto e com urgência, quando ocorrem, causam atraso na entrega e diminuem a eficácia dos membros do sistema. A colaboração dos membros do CSCS e a introdução de técnicas de negociação fornecem uma solução para esses problemas.

Suponha-se que para a reposição de peças e materiais, uma empresa publica um leilão no quadro-negro. Este leilão pode ser visualizado por toda a comunidade de fornecedores, caracterizando-se, portanto, como uma competição aberta.

Os agentes fornecedores, *Supplier Agents*, de cada empresa fornecedora, exploram a Internet para encontrar os leilões de seu interesse. Quando um agente encontra um leilão de seu interesse, ele indica sua respectiva companhia fornecedora.

Em seguida, o quadro-negro publica os lances apresentados por estas companhias. Os membros realizam revisões dos lances ou negociam com essas empresas. Logo após, o lance mais apropriado é selecionado com base em critérios de seleção.

Se por ventura um problema de entrega ocorre com um dos fornecedores, os agentes colaboram e negociam entre si para substituí-lo. O fornecedor alternativo é escolhido de acordo com os critérios de seleção preparados pelo fabricante (*Manufacturer Agent*). A Figura 8 mostra como esses tipos de atividades ocorrem quando o fornecedor selecionado não entregar os materiais no momento certo. Nessa atividade, o fornecedor em questão, o fabricante e o fornecedor alternativo são controlados pelo *Supplier Agent 1* (SA1), *Manufacturer Agent* (MA) e *Supplier Agent 2* (SA2), respectivamente.

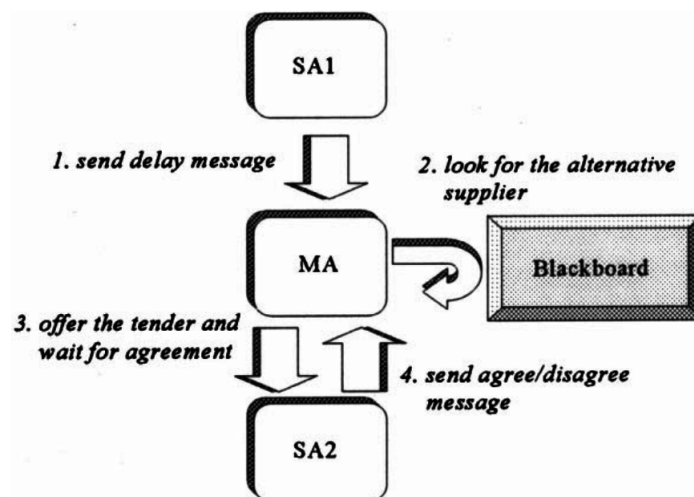


Figura 8 – Processo de colaboração e negociação para substituição de fornecedor. Fonte: Ito e Salleh (2000).

Quando o problema acontece, o SA1 envia uma mensagem de atraso para o MA. Depois de receber a mensagem de atraso, o MA refere-se ao BB para fornecer um fornecedor alternativo para a substituição. Se o SA2 for considerado o fornecedor

alternativo, o MA faz uma consulta ao SA2 e solicita a estimativa de custo da entrega.

Então, o SA2 cita o custo dos materiais e o tempo da entrega estimado e os envia de volta ao MA. Se o SA2 não aceitar a oferta, o MA executará o mesmo procedimento para outro fornecedor alternativo. E deste modo, são realizadas as atividades de colaboração e negociação entre esses agentes, ajudando fabricantes e fornecedores a resolver os problemas durante o processo de entrega.

Implementação: a fim de demonstrar o padrão *Blackboard*, é utilizado o contexto acima para simular a entrega de materiais. Todos os detalhes da implementação, configuração de ambiente e passos para execução são descritos no Apêndice [A.3](#).

5 Contract Net

O *Contract Net* é um protocolo que atende aos critérios definidos para caracterizar-se como padrão arquitetural organizacional. Trata-se de um protocolo para comunicação entre agentes voltado ao controle distribuído de tarefas em execução envolvendo negociação entre agentes. Seu detalhamento, enquanto padrão, é realizado a seguir, baseando-se no protocolo de comunicação *FIPA-Contract-Net*. Para maior entendimento dos atos comunicativos FIPA, a Seção 5.1 busca descrever os mesmos.

Nome do padrão: *Contract Net*.

Referências: Bellifemine, Caire e Greenwood (2007, pág. 23), Bellifemine et al. (2002, pág. 31).

Categoria: *Bureaucracy structure*.

Problema: um agente, o *Initiator* (ou iniciador), que deseja ter alguma tarefa executada por um ou mais agentes, os *Responders* (ou participantes), e ainda deseja otimizar uma função que caracteriza a tarefa. Esta função é comumente expressa, por exemplo, como custo, tempo até a conclusão, e a distribuição justa das tarefas (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

Solução: para uma determinada tarefa, qualquer número de participantes pode responder com uma proposta; os demais devem recusar. As negociações prosseguem com os Participantes que propuseram. O Iniciador solicita um número m de propostas de outros agentes através da emissão de um convite à apresentação de propostas (*Call for Proposals*), que especifica a tarefa e quaisquer condições que o Iniciador coloca sobre a execução da tarefa.

Os participantes que recebem o convite à apresentação de propostas são vistos como potenciais contratados e são capazes de gerar um número n de respostas. Destes, j são o número de propostas para realizar a tarefa, especificada como proposta.

A proposta do Participante inclui as pré-condições que o Participante está definindo para a tarefa, que pode ser o preço, o tempo em que a tarefa será feita, dentre outras. Alternativamente, os Participantes podem se recusar a propor. Uma vez passado o prazo, o Iniciador avalia o valor recebido

Protocolos associados: o *FIPA-Contract-Net* está associado a este padrão e pode ser representado pela Figura 9.

O *FIPA-Contract-Net* permite que o *Initiator* envie um *Call for Proposal* (CFP) a um conjunto de *Responders*, avalie suas propostas e, por fim, realize sua escolha

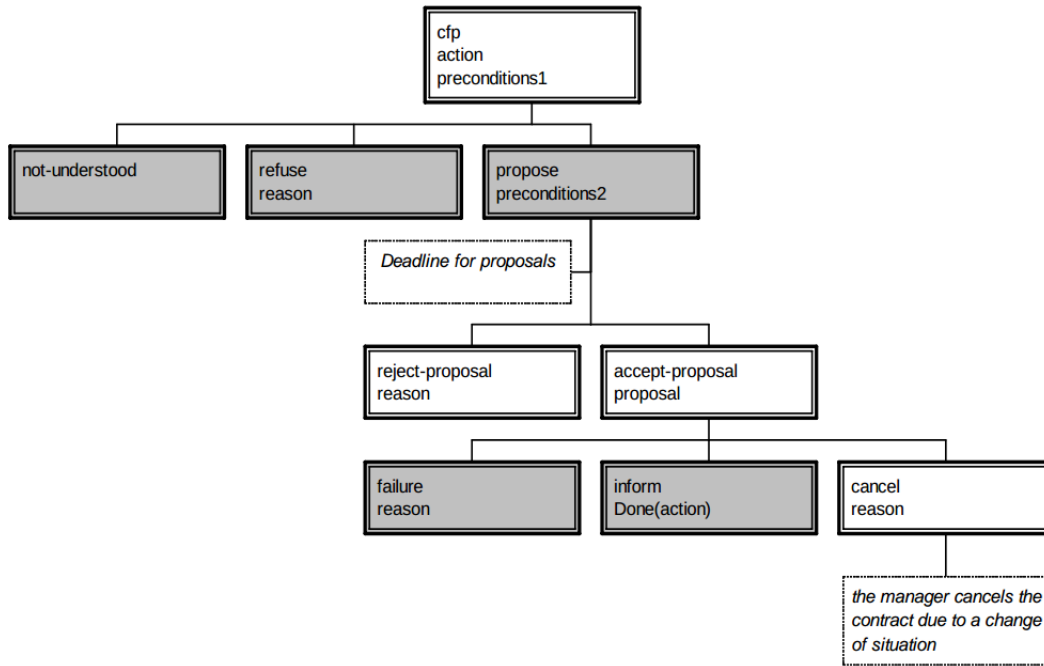


Figura 9 – Protocolo de Interação *FIPA-Contract-Net*. Fonte: Bellifemine et al. (2002, pág. 36)

aceitando uma das propostas ou até mesmo rejeitando a todas (BELLIFEMINE et al., 2002).

A mensagem CFP contém a ação a ser realizada e, se necessário, condições de execução. Os *Responders* podem responder com as seguintes mensagens: *Propose* - sua proposta composta por pré-condições, custo e tempo -, *Refuse* para recusa - ou *Not-Understood* em caso de falhas de comunicação.

Após avaliar todas as propostas, o *Initiator* realiza sua escolha e informa quais foram rejeitadas e quais foram aceitas (através da mensagem *Accept Proposal*). Estes, assim que completam suas tarefas, respondem com *Inform* o resultado de sua ação - como finalizada ou como *Failure*.

O *Initiator* pode decidir cancelar o protocolo, enviando uma mensagem *Cancel*, antes da ação ter sido realizada e a última mensagem ter sido recebida.

O *FIPA-Contract-Net* é implementado por dois comportamentos: *ContractNetInitiator*¹ e *ContractNetResponder*². O primeiro, atua sob o ponto de vista do *Initiator* e cuida do tempo limite de espera das propostas; além de prover os métodos *callback* para cada estado do protocolo. O comportamento *ContractNetResponder* atua sob o ponto de vista do *Responder*, e é responsável, principalmente, por avaliar a ação solicitada, enviar propostas ou recusar o envio de propostas.

Modelagem: este padrão pode ser representado pela Figura 10.

¹ <http://jade.tilab.com/doc/api/jade/proto/ContractNetInitiator.html> (último acesso: Junho 2017)

² <http://jade.tilab.com/doc/api/jade/proto/ContractNetResponder.html> (último acesso: Junho 2017)

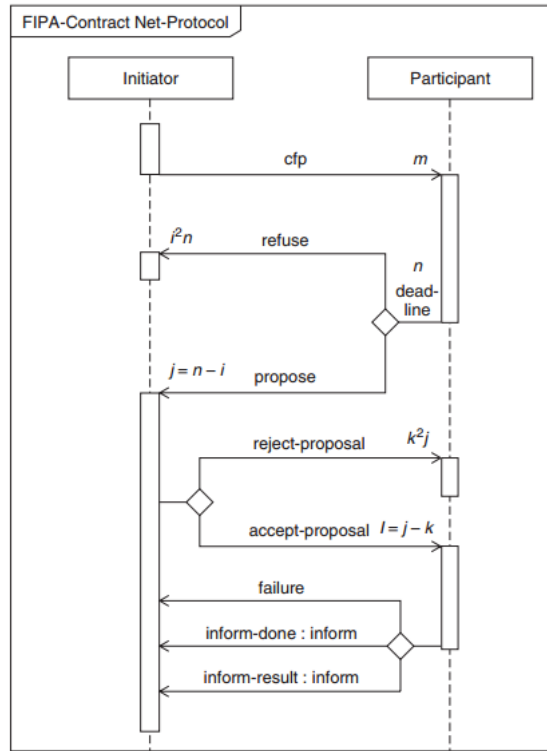


Figura 10 – Protocolo de interação *FIPA-Contract-Net*. Fonte: Bellifemine, Caire e Greenwood (2007, pág. 23).

Implementação: a fim de demonstrar o padrão *FIPA-Contract-Net*, é descrito o exemplo *Book Trading* fornecido pela plataforma JADE³. Todos os detalhes da implementação, configuração de ambiente e passos para execução são descritos no Apêndice A.1.

5.1 FIPA ACL

O FIPA-ACL baseia-se em mensagens que representam ações chamadas atos comunicativos (ou *communicative acts*), como apresenta a Figura 11. São definidos ao todo um conjunto de vinte e dois atos comunicativos (Tabela 1). Alguns dos atos mais comumente usados são *inform*, *request*, *agree*, *not understood*, e *refuse*. Estes capturam a essência da maioria das formas de comunicação básica (BELLIFEMINE; CAIRE; GREENWOOD, 2007, pág. 13). Segundo Bellifemine, Caire e Greenwood (2007), as normas FIPA estabelecem que um agente deve ser capaz de receber qualquer *communicative act* e, no mínimo, responder com uma mensagem *not-understood*, caso a mensagem recebida não possa ser processada. Com base nestes atos comunicativos, o FIPA definiu um conjunto de protocolos de interação, consistindo cada um de uma sequencia de atos comunicativos para coordenar ações multi-mensagem, como o *FIPA-Contract-Net* (Seção 5) para o estabele-

³ <http://jade.tilab.com/dl.php?file=JADE-examples-4.5.0.zip> (último acesso: Junho 2017)

cimento de acordos e vários tipos de leilões (BELLIFEMINE; CAIRE; GREENWOOD, 2007, pág 14).

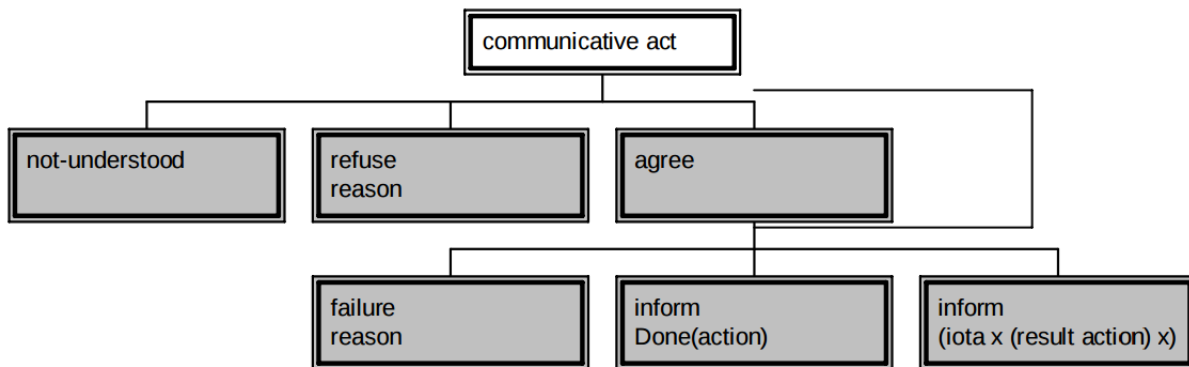


Figura 11 – *Communicative acts*. Fonte: Bellifemine et al. (2002, pág. 32).

Tabela 1 – Atos comunicativos FIPA. Traduzido. Fonte: Bellifemine, Caire e Greenwood (2007, pág. 19).

FIPA communicative act	Descrição
<i>Accept Proposal</i>	A ação de aceitar uma proposta previamente submetida para realizar ação.
<i>Agree</i>	A ação de concordar em realizar alguma ação, possivelmente no futuro.
<i>Cancel</i>	A ação de um agente informando outro agente de que o primeiro agente não tem mais a intenção de que o segundo agente execute alguma ação.
<i>Call for Proposal</i>	A ação de convocação de propostas para executar uma determinada ação.
<i>Confirm</i>	O remetente informa o receptor que uma dada proposição é verdadeira, onde o receptor é conhecido por ser incerto sobre a proposição.
<i>Disconfirm</i>	O remetente informa o receptor de que uma dada proposição é falsa, onde o receptor é conhecido por acreditar, ou acredita que é provável que, a proposição seja verdadeira.
<i>Failure</i>	A ação de dizer a outro agente que houve a tentativa de realizar uma ação, mas a tentativa falhou.
<i>Inform</i>	O remetente informa o receptor que uma dada proposição é verdadeira.
<i>Inform If</i>	Uma ação macro para o agente da ação para informar ao destinatário se uma proposição é verdadeira ou não.
<i>Inform Ref</i>	Uma ação macro que permite ao remetente informar o receptor de algum objeto acreditado pelo remetente para corresponder a um descritor específico, por exemplo, um nome.
<i>Not Understood</i>	O remetente do ato (por exemplo, <i>i</i>) informa o receptor (por exemplo, <i>j</i>) que percebeu que <i>j</i> realizou alguma ação, mas que <i>i</i> não entendeu o que <i>j</i> acabou de fazer. Um caso particular comum é quando <i>i</i> diz a <i>j</i> que <i>i</i> não entende a mensagem que <i>j</i> acaba de enviar para <i>i</i> .
<i>Propagate</i>	O remetente tem a intenção de que o receptor trate a mensagem incorporada como enviada diretamente para o receptor, e quer o receptor identifique os agentes denotados pelo descritor informado e envie a mensagem <i>propagate</i> recebida a eles.
<i>Propose</i>	A ação de perguntar a outro agente se uma determinada proposição é verdadeira ou não.
<i>Proxy</i>	O remetente deseja que o receptor selecione agentes de destino indicados por uma dada descrição e que ele envie uma mensagem incorporada para eles.
<i>Query If</i>	A ação de submeter uma proposta para executar uma determinada ação, dadas certas pré-condições.
<i>Query Ref</i>	A ação de perguntar a outro agente para o objeto referido por uma expressão referencial.
<i>Refuse</i>	A ação de recusar a execução de uma determinada ação, e explicando o motivo da recusa.
<i>Reject Proposal</i>	A ação de rejeitar uma proposta de realização de alguma ação durante uma negociação.
<i>Request</i>	O remetente solicita ao receptor que execute alguma ação. Um exemplo importante de usos do ato de <i>request</i> é solicitar ao receptor que execute outro ato comunicativo.
<i>Request When</i>	O remetente quer que o receptor execute alguma ação quando alguma proposição dada se torna verdadeira.
<i>Request Whenever</i>	O remetente quer que o receptor execute alguma ação assim que alguma proposição se tornar verdadeira e depois cada vez que a proposição se tornar verdadeira novamente.
<i>Subscribe</i>	O ato de solicitar uma intenção persistente de notificar o remetente do valor de uma referência e de notificar novamente sempre que o objeto identificado pela referência muda.

6 MACRON

O padrão MACRON (*Multiagent Architecture for Cooperative Retrieval ONline*) apresenta um sistema de aquisição de informações (DECKER et al., 1995). Nesta organização, é elaborado um planejamento único que gera sub-objetivos a serem alcançados por um grupo de agentes que cooperam entre si.

A cooperação entre agentes implica o gerenciamento de interdependências entre suas tarefas. Ao interagirem, os agentes integram e desenvolvem agrupamentos consistentes de informações de alta qualidade a partir de fontes heterogêneas distribuídas (DECKER et al., 1995).

Tais agentes compartilham informações um com o outro de modo a saberem (i) quais os planos que eles executam para alcançar seus sub-objetivos, (ii) em qual ordem executam tarefas, e (iii) quando as executam. Além disso, os agentes são livres para desenvolver diferentes planos de coleta de informações, a depender da quantidade de tempo que possuem para produzir uma resposta (DECKER et al., 1995).

Nome do padrão: MACRON, *Multiagent Architecture for Cooperative Retrieval ONline*.

Referências: Decker et al. (1995) e Shehory (1998).

Categoria: *Matrix structure*. Isso se deve ao fato de que este padrão possui uma partição bidimensional para unidades: unidades funcionais e unidades de resposta de consulta, a serem descritas a seguir.

Problema: ocorre quando uma informação não pode ser simplesmente recuperada, mas adquirida através de um processo dinâmico, incremental e limitado pelos recursos disponíveis (DECKER et al., 1995).

Solução: O padrão MACRON possui os seguintes objetivos: (i) explorar interdependências entre problemas, (ii) gerenciar a incerteza inerente à busca, (iii) compensar inteligentemente a qualidade da solução devido às limitações de recursos, e (iv) explorar ou evitar redundância, conforme necessário (DECKER et al., 1995).

De acordo com Shehory (1998), o sistema consiste nos seguintes agentes ou grupo de agentes:

- **Query-Manager** (QM) ou gerente de consulta: recebe uma consulta de um usuário, desenvolve um plano inicial de levantamento de informações de alto

nível, recruta agentes das *Functional Units* (FUs) necessárias através dos *Functional Managers* (FMs) para formar uma *Query-Answering Unit* (QAU) e monitora a execução do plano;

- **Functional Unit** (FU) ou unidade funcional: uma coleção de agentes que têm acesso a um determinado tipo de recursos de informação. Cada unidade funcional possui um *Functional Manager*;
- **Functional Manager** (FM) ou gerente funcional: a pedido de um *Query Answering Agent*, o agente FM atribui tarefas a agentes dentro de sua FU. Embora seja para o mesmo tipo de fontes de informação, os agentes dentro de um FU específica podem ter diferentes conhecimentos. O FM leva essas diferenças em conta ao planejar a tarefa a ser atribuída;
- **Functional Agent** (FA) ou agente funcional: planeja a recuperação de informações, solicita-as e as manipula;
- **Low Level Information Agents** (LLIA) ou agentes de informação de baixo nível: são simples agentes, cada um especializado em um tipo específico de informações de recuperação. Tal agente recupera as informações solicitadas por um FA, e
- **Organization Chart Manager** (OCM) ou gerenciador gráfico da organização: é uma memória organizacional onde os agentes podem procurar informações sobre a disponibilidade e as capacidades de outros agentes. Novos agentes são adicionados ao OCM quando eles se juntam ao sistema dinamicamente, durante o tempo de execução. QMs, FMs e agentes funcionais todos consultam o OCM para localizar os agentes apropriados aos quais eles têm que delegar tarefas.

Na Figura 12, são apresentados os componentes do MACRON e as relações entre eles. Alguns detalhes foram omitidos pelo autor, como a comunicação bidirecional de agentes FA com o OCM e de agentes FM com agentes FAs em sua unidade funcional. Cabe ressaltar que uma vez que o QAU foi formado, o monitoramento das FAs é feito pelo QM responsável pela consulta e não pelo FM.

Do ponto de vista organizacional, os agentes no MACRON formam uma organização matricial, consistindo em unidades funcionais e de resposta de consulta como apresentado na Figura 13. Em uma organização matricial humana, as pessoas pertencem a um grupo funcional de longo prazo e curto prazo. Por exemplo, em um ambiente de desenvolvimento de software, existe um grupo de *designers* de interface, um grupo de desenvolvedores de software e outro grupo de *designers* de *hardware*, consistindo em grupos de longo prazo. De modo dinâmico, pessoas destes grupos são alocadas para formar um grupo para um projeto de curto prazo. Tais organizações proporcionam flexibilidade ao sistema, em ambientes com mudanças incertas

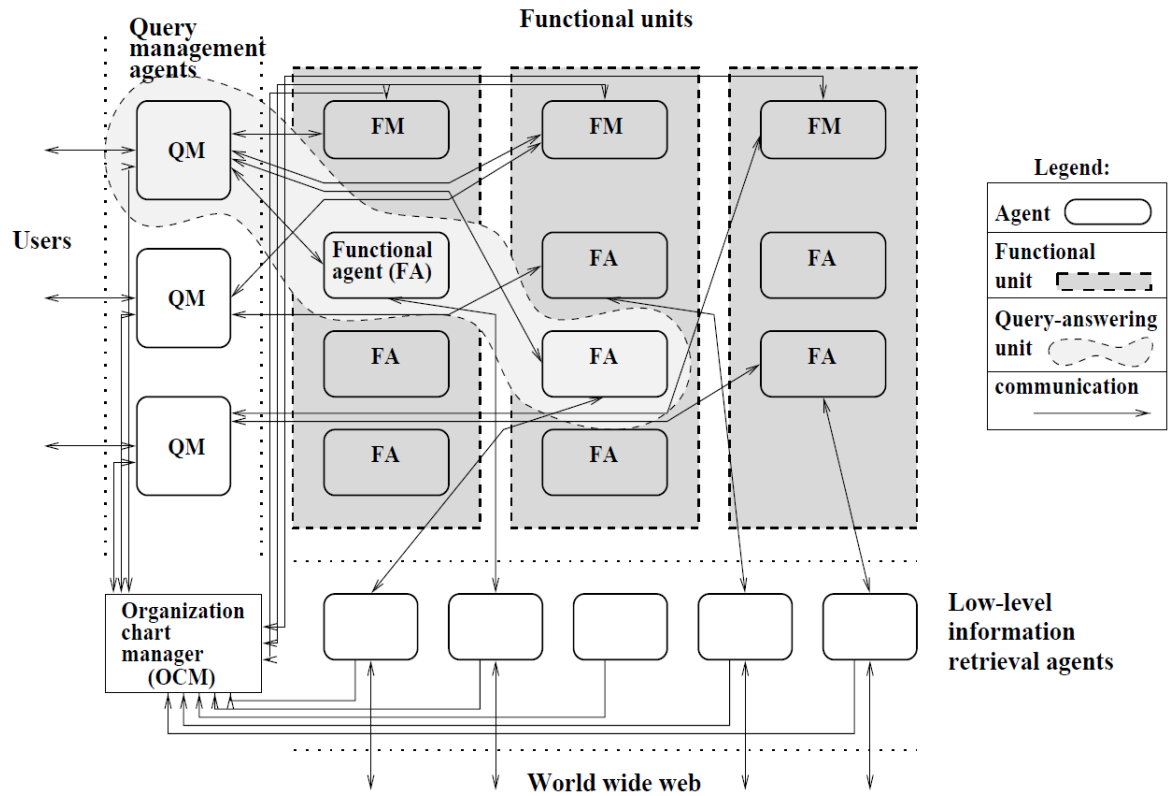


Figura 12 – Organização MACRON. Fonte: Shehory (1998).

e dinâmicas, permitindo ainda a atribuição e o rastreamento de recursos limitados entre os agentes.

Modelagem: este padrão pode ser representado pela Figura 12.

Exemplo: o exemplo a seguir baseia-se em um sistema (Figura 13) capaz de recuperar *reviews* de usuários na internet a cerca de um determinado produto.

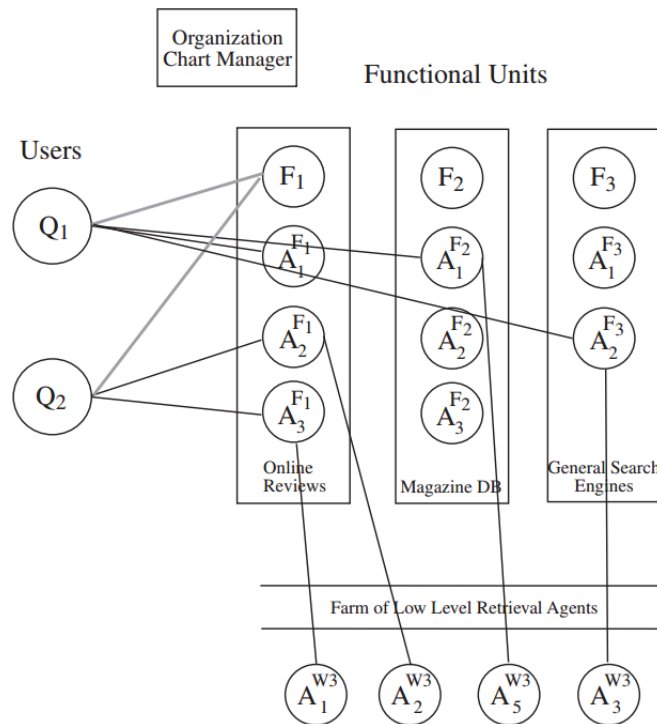


Figura 13 – Exemplo de estrutura MACRON. Fonte: Decker et al. (1995).

Na Figura 14, o agente QM pode expandir o objetivo inicial *Get Review* para *Get Online Reviews* e *Get Published Reviews*. Ele, então, se comunica com agentes FM para obter *Online Reviews* e *Published Reviews*. Os agentes FM, por sua vez, alocam os objetivos *Get Online Reviews* e *Get Published Reviews* aos agentes das respectivas unidades funcionais (FUs).

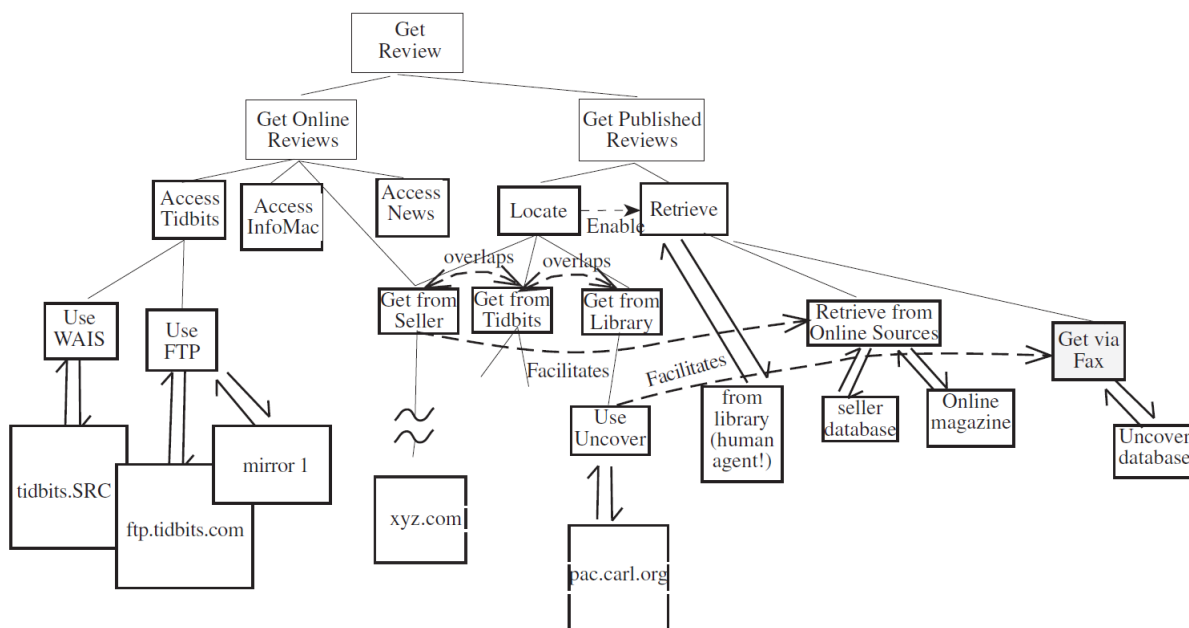


Figura 14 – Árvore de tarefas para recuperar *reviews* sobre um produto. Fonte: Decker et al. (1995).

Suponha-se que cada objetivo é alocado para um agente. Os agentes expandem seus objetivos para alcançar as tarefas primitivas de nível mais baixo, como *Use Wais*, *Use FTP* e *Retrieve Online Magazine*. O agente de consulta verifica o OCM para encontrar o endereço do FM cujas FUs possam obter desempenhar a tarefa *Get Online Reviews*.

Para melhor compreensão da utilidade da estrutura MACRON, a seguir ela será comparada, brevemente, à duas abordagens alternativas. Em uma abordagem de organização de **agentes com funções fixas**, teríamos equipes de agentes, cada equipe consistindo de pelo menos um membro de cada FU. Essas equipes fixas causam vários problemas: como todas as consultas não precisarão de todos os recursos funcionais, alguns membros da equipe ficarão ociosos. Por outro lado, se um membro da equipe necessário estiver temporariamente indisponível, uma consulta poderá falhar. Em terceiro lugar, à medida que forem introduzidos componentes de aprendizagem nos agentes, as equipes começam a diferenciar e desenvolver diferentes especializações em cada uma de suas áreas funcionais. Se o ambiente é dinâmico, pode-se esperar que surjam novas consultas que seriam mais bem respondidas por combinações de agentes experientes que diferem das equipes fixas oferecidas.

Uma segunda abordagem para se comparar com a estrutura MACRON, seria o outro extremo da abordagem anterior, com uma variedade de **agentes sem funções fixas**, onde um agente de consultas pode reunir uma equipe de agentes para lidar com uma consulta específica. Neste caso, não há recursos desperdiçados, e não há problemas se um único agente estiver temporariamente indisponível. Este sistema é mais flexível do que a organização funcional fixa.

No entanto, essa abordagem falha em fornecer métodos simples para reunir a melhor equipe possível, especialmente em um sistema cooperativo. São duas importantes perguntas que o agente responsável pelas consultas deve responder:

- Qual dos agentes que anunciam um serviço ele deve realmente entrar em contato e utilizar seu serviço?
- À medida que os agentes aprendem e se diferenciam, como eles podem comparar suas habilidades relativas sem se conhecerem?

Na organização MACRON, a escolha de um agente funcional é de responsabilidade exclusiva dos gerentes funcionais; que atuam como facilitadores especializados e inteligentes. O MACRON permite o melhor das duas abordagens anteriores: a flexibilidade de montar uma equipe eficiente e a capacidade de atribuir e rastrear de forma inteligente e dinâmica esses recursos do agente.

Implementação: a fim de demonstrar o padrão MACRON, é utilizado o contexto do exemplo acima. Todos os detalhes da implementação, configuração de ambiente e passos para execução são descritos no Apêndice [A.4](#).

7 Generalized Partial Global Planning

O padrão *Generalized Partial Global Planning* (GPGP) atua na coordenação de tarefas locais de cada agente considerando objetivos locais e não locais (DECKER; LESSER, 1992).

Nome do padrão: *Generalized Partial Global Planning* (GPGP) ou Planejamento Global Parcial Generalizado.

Referências: Weerdt e Clement (2009), Decker e Lesser (1992).

Categoria: *Team Structure*.

Problema: é adotado para possibilitar a coordenação distribuída; que pode ser descrita como o agendamento das atividades locais de cada agente considerando preocupações e restrições não locais (DECKER; LESSER, 1992).

Solução: Para abordar o padrão GPGP, será primeiramente apresentada a estrutura PGP exemplificada pela Figura 15. O *Agent 1* possui o objetivo global de completar a tarefa *A1*, e duas sub-tarefas concorrentes, *B1* e *C1*. O *Agent 2* possui o objetivo de completar a tarefa *A2*, e duas sub-tarefas *B2* e *D2* (DECKER; LESSER, 1992).

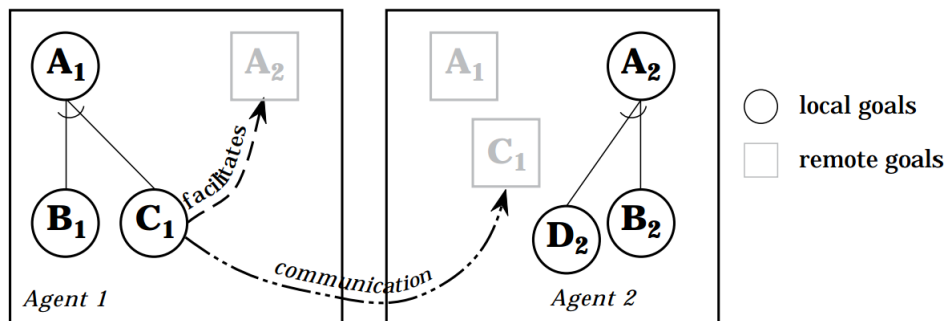


Figura 15 – Exemplo de estrutura PGP. Fonte: Decker e Lesser (1992).

No algoritmo PGP, os agentes *Agent 1* e *Agent 2* vão trocar seus objetivos de mais alto nível, chamados *System Goals*. Havendo ciência desta informação, o *Agent 1* pode determinar que sua sub-tarefa local *C1* facilita a tarefa *A2* (DECKER; LESSER, 1992).

Isto indica que a tarefa *C1*, de algum modo, é útil para que se complete *A2*. Informações sobre o status da sub-tarefa *C1* são enviadas ao *Agent 2* permitindo-o reorganizar sua programação para lidar com informações futuras. Similarmente, o fato de *C1* ser uma tarefa facilitadora, torna-a preferível. Por exemplo, se o *Agent 1* possuir duas tarefas concorrentes, sendo uma delas *C1*, e não havendo outra razão

para dar preferência à outra, o algoritmo PGP agendará *C1* para ocorrer primeiro (DECKER; LESSER, 1992).

A estrutura do PGP não foi projetada apenas para permitir a comunicação de informações necessárias para o planejamento global, mas também para o projeto e análise de algoritmos de coordenação para agentes que operam em ambientes com características diferentes daquelas para as quais o PGP foi projetado. Por exemplo, podem haver agentes heterogêneos - que podem ter diferentes critérios locais de solução de problemas -, agentes dinâmicos - que possuem várias estratégias e métodos diferentes disponíveis para a realização de metas e um conjunto de compensações entre eles - e em tempo real - agentes que podem ter prazos rígidos ou flexíveis (DECKER; LESSER, 1992).

O Planejamento Global Parcial (PGP) é uma abordagem flexível à coordenação distribuída que permite aos agentes responder dinamicamente à sua situação atual. O GPGP tenta estender a abordagem PGP comunicando informações mais abstratas e organizadas hierarquicamente, detectando de maneira geral as relações de coordenação necessárias aos mecanismos de planejamento global parcial e separando o processo de coordenação do planejamento local (DECKER; LESSER, 1992).

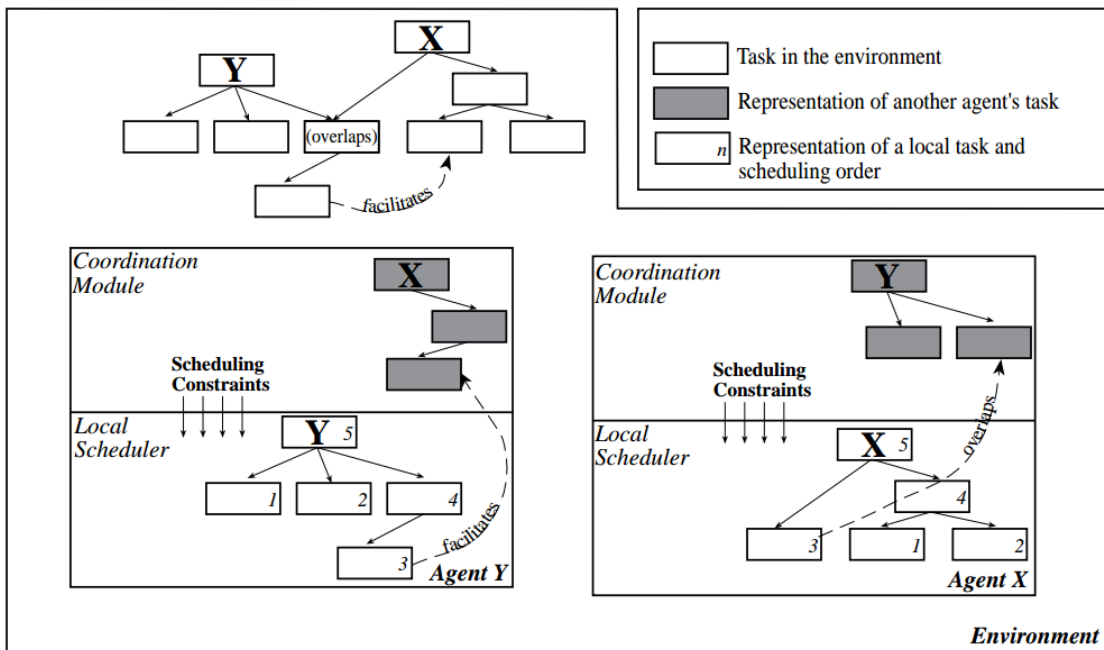


Figura 16 – Exemplo de estrutura GPGP. Fonte: Decker e Lesser (1992).

A estrutura do padrão GPGP pode ser exemplificada pela Figura 16. Basicamente, a informação flui pelo ambiente através do mecanismo de coordenação do agente, *Coordination Module*, e do mecanismo de agendamento local, *Local Scheduler*. Primeiro, um determinado ambiente e/ou domínio de tarefa, em conjunto com um determinado agente, induz determinados relacionamentos de coordenação (*RCs*) entre tarefas nesse ambiente (DECKER; LESSER, 1992).

Cada agente segue algum algoritmo de coordenação que detecta ou até mesmo hipotetiza *RCs* e reage de acordo. Em seguida, este algoritmo produz certos comportamentos, por exemplo, (i) a criação e refinamento de restrições no agendamento local, e (ii) a negociação e criação de estruturas de dados internas (DECKER; LESSER, 1992).

Modelagem: Este padrão pode ser exemplificado pela Figura 16.

Exemplo: De modo a exemplificar uma utilização do padrão, é apresentado o caso abordado no artigo *Coordinated Hospital Patient Scheduling* (DECKER; LI, 1998) e exemplificado pela Figura 17.

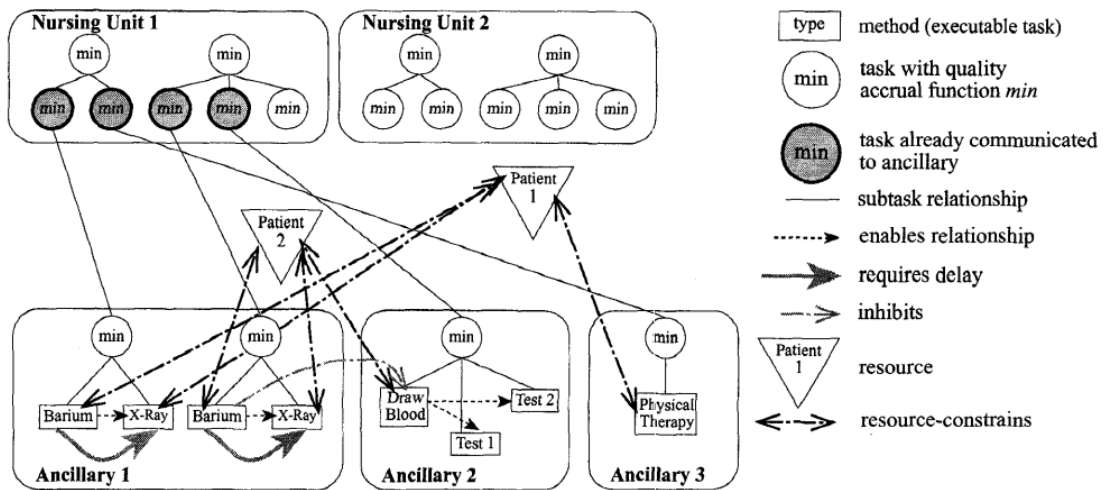


Figura 17 – Exemplo de agendamento de um paciente. Fonte: Decker e Li (1998).

Os pacientes de um hospital geral residem em unidades organizadas por ramos de medicina, como ortopedia ou neurocirurgia. Todos os dias, os médicos solicitam que determinados testes e/ou terapias sejam realizados como parte do diagnóstico e tratamento de um paciente. Os testes são realizados por departamentos auxiliares separados, independentes e dispostos em diferentes locais do hospital. O departamento de radiologia, por exemplo, presta serviços de raio-x e pode receber pedidos de várias unidades diferentes no hospital.

Além disso, cada teste pode interagir com outros testes em relacionamentos, tais como: autorizar, requisitar, atrasar, impedir outros testes. Essas relações entre tarefas indicam quando a execução de uma tarefa altera as características de outra tarefa e, neste caso, é principalmente a duração. A partir dessa visão da estrutura da tarefa, o problema de agendamento do hospital tem as seguintes características peculiares:

- Tarefas não têm redundância. Cada teste só pode ser feito por um único departamento auxiliar;

- As funções de acúmulo de qualidade de tarefas não executáveis são sempre mínimas. Isto porque aqui todos os testes precisam ser feitos;
- A qualidade final não é importante: um teste é concluído ou não é, e
- Agentes diferentes, representando diferentes *Nursing Units* (unidades de enfermagem) ou diferentes *Ancillary Units* (unidades auxiliares), podem usar diferentes regras para fazer programações locais. Por exemplo, as *Nursing Units* podem tentar minimizar o tempo de permanência do paciente, enquanto a *Ancillary Unit* tenta maximizar o uso do equipamento e/ou minimizar os tempos depreendidos em configuração dos mesmos.

Neste problema de agendamento do hospital, alguns testes só podem ser feitos com o paciente fisicamente presente. Assim, surge um problema de restrição de recursos, ou ***shared resource-constrained relationship***, em que **o paciente se torna um recurso crucial não-compartilhável** para diferentes agentes.

Quando vários agentes tentam usar o mesmo recurso não compartilhável em horários sobrepostos, apenas um agente pode realmente obter o recurso e executar seu trabalho. Outros que não falharam em obter o recurso, desperdiçaram tempo e esforço da unidade.

A ideia por trás do mecanismo de coordenação com restrição de recursos é que quando um agente pretende executar uma tarefa com restrição de recurso, ele deve enviar um lance informando o intervalo de tempo que necessita do recurso e a sua prioridade local.

Após um pequeno *delay* de comunicação, ele passa a conhecer todos os lances que foram dados pelos outros agentes ao mesmo tempo que seu próprio lance. Como todos os agentes que fizeram lances terão as mesmas informações iniciais, se todos usarem a mesma regra comumente aceita para decidir quem receberá o intervalo de tempo, eles poderão obter o mesmo resultado nessa rodada de lances.

O agente que ganhou manterá sua programação e executará essa tarefa no intervalo de tempo que ele solicitou no lance, e todos os outros marcarão esse intervalo de tempo com um compromisso de não fazer e ou sequer tentar executar a tarefa neste período que necessite do recurso. Todos os agentes que não obtiveram seus intervalos de tempo nessa rodada serão reprogramados e realizarão lances novamente. O processo detalhado é o seguinte:

Por exemplo, suponha que o atraso de comunicação seja uma unidade de tempo e que hajam três agentes e tarefas da seguinte maneira:

- O agente A tem as tarefas A11 e A12, cada uma de duração 3;
- O agente B tem a tarefa B11 de duração 2;

- O agente C tem a tarefa C11 de duração 4 e a tarefa C12 de duração 1;
- A11, B11 e C11 se caracterizam como *shared resource-constrained relationship*.

O fluxo seria o seguinte:

Tempo 0: cada agente comunica estruturas de tarefas.

Tempo 1: cada agente faz sua própria programação local: Agente A: A11 (1-3), A12 (4-6); Agente B: B11 (1-2); Agente C: C12 (1-1). Momentaneamente a tarefa C11 não está no calendário local do Agente C. A razão podem ser causas externas, como C11 não estar habilitado.

O Agente A envia um lance para o intervalo de tempo 1-3 com prioridade 3.

O agente B envia um lance para o intervalo de tempo 1-2 com prioridade 4.

Tempo 2: cada agente reúne informações sobre o lance que aconteceu no tempo 1.

O Agente A descobre que outro agente ganhou o intervalo de tempo de 1 a 2, por isso marca esse intervalo de tempo como ocupado e, em seguida, tenta se reprogramar. Seus novos horários são: A12 (2-4), A11 (5-7).

O Agente A envia um lance de 5-7 com prioridade 3.

O Agente B ganhou, por isso mantém o seu horário e inicia a execução de B11.

O Agente C coloca C11 (3-6) na programação geral. O Agente C envia um lance para 3-6 com prioridade 2.

Tempo 3: o Agente A descobre que ganhou, por isso mantém sua programação e começa a executar A12.

O Agente C descobre que perdeu, por isso marca o intervalo de tempo 5-7 ocupado. A nova programação é C11 (7-10).

O Agente C envia um lance para o intervalo de tempo 7-10.

Tempo 4: o Agente C venceu, ele mantém sua programação e, portanto, aguardará até o tempo 7 para iniciar sua execução.

Implementação: a fim de demonstrar este padrão, o código apresentado no Apêndice [A.5](#) simula o fluxo do contexto apresentado acima.

Referências

- AART, C. van. *Organizational Principles for Multi-Agent Architectures*. [S.l.]: Springer Science & Business Media, 2004. Citado na página 4.
- ARGENTE, E.; JULIAN, V.; BOTTI, V. Multi-agent system development based on organizations. *Electronic Notes in Theoretical Computer Science*, v. 150, n. 3, p. 55 – 71, 2006. ISSN 1571-0661. Proceedings of the First International Workshop on Coordination and Organisation (CoOrg 2005) Proceedings of the First International Workshop on Coordination and Organisation (CoOrg 2005). Disponível em: <<http://www.sciencedirect.com/science/article/pii/S157106610600329X>>. Citado 6 vezes nas páginas 4, 5, 7, 8, 9 e 10.
- ARIDOR, Y.; LANGE, D. B. Agent design patterns: elements of agent application design. In: ACM. *Proceedings of the second international conference on Autonomous agents*. [S.l.], 1998. p. 108–115. Citado 2 vezes nas páginas 13 e 14.
- AVGERIOU, P.; ZDUN, U. Architectural patterns revisited—a pattern. 2005. Citado na página 4.
- BELLIFEMINE, F. et al. Jade programmer’s guide. *Jade version*, v. 3, 2002. Disponível em: <<http://jade.tilab.com/doc/programmersguide.pdf>>. Citado 3 vezes nas páginas 23, 24 e 26.
- BELLIFEMINE, F. L.; CAIRE, G.; GREENWOOD, D. *Developing multi-agent systems with JADE*. [S.l.]: John Wiley & Sons, 2007. v. 7. Citado 4 vezes nas páginas 23, 25, 26 e 27.
- DECKER, K. et al. Macron: an architecture for multi-agent cooperative information gathering. In: *Proceedings of the CIKM-95 Workshop on Intelligent Information Agents*. [S.l.: s.n.], 1995. Citado 2 vezes nas páginas 29 e 32.
- DECKER, K.; LI, J. Coordinated hospital patient scheduling. In: IEEE. *Multi Agent Systems, 1998. Proceedings. International Conference on*. [S.l.], 1998. p. 104–111. Citado na página 37.
- DECKER, K. S.; LESSER, V. R. Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems*, World Scientific, v. 1, n. 02, p. 319–346, 1992. Citado 3 vezes nas páginas 35, 36 e 37.
- DONG, J.; CHEN, S.; JENG, J.-J. Event-based blackboard architecture for multi-agent systems. In: IEEE. *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*. [S.l.], 2005. v. 2, p. 379–384. Citado 2 vezes nas páginas 17 e 18.
- GREEN, S. et al. Software agents: A review. *Department of Computer Science, Trinity College Dublin, Tech. Rep. TCS-CS-1997-06*, 1997. Citado na página 3.
- ITO, T.; SALLEH, M. R. A blackboard-based negotiation for collaborative supply chain system. *Journal of Materials Processing Technology*, Elsevier, v. 107, n. 1, p. 398–403, 2000. Citado 4 vezes nas páginas 17, 19, 20 e 21.

JENNINGS, N. R.; WOOLDRIDGE, M. J. Software agents. *IEE review*, p. 17–20, 1996. Citado na página 3.

JUNIOR, G. B. d. S. et al. Padrões arquiteturais para o desenvolvimento de aplicações multiagente. Universidade Federal do Maranhão, 2003. Citado na página 3.

LARMAN, C. *Utilizando UML e padrões*. [S.l.]: Bookman Editora, 2002. Citado na página 12.

MALONE, T. W. Organizing information processing systems: Parallels between human organizations and computer systems. *Cognition, Computation and Cooperation*, Ablex, Norwood, NJ, p. 56–83, 1990. Citado 2 vezes nas páginas 7 e 10.

RICHARDS, M. Software architecture patterns. *O'Reilly Media*, 2015. Citado 2 vezes nas páginas 3 e 4.

SHAW, M. Some patterns for software architectures. *Pattern languages of program design*, v. 2, p. 255–269, 1996. Citado na página 4.

SHEHORY, O. M. *Architectural properties of multi-agent systems*. [S.l.]: Carnegie Mellon University, The Robotics Institute, 1998. Citado 2 vezes nas páginas 29 e 31.

SYCARA, K. P. Multiagent systems. *AI magazine*, v. 19, n. 2, p. 79, 1998. Citado 3 vezes nas páginas 4, 7 e 11.

WEERDT, M. D.; CLEMENT, B. Introduction to planning in multiagent systems. *Multiagent and Grid Systems*, IOS Press, v. 5, n. 4, p. 345–355, 2009. Citado na página 35.

WEISS, G. *Multiagent systems: a modern approach to distributed artificial intelligence*. [S.l.]: MIT press, 1999. Citado 2 vezes nas páginas 17 e 19.

WOOLDRIDGE, M. *An introduction to multiagent systems*. [S.l.]: John Wiley & Sons, 2009. Citado na página 3.

ZAMBONELLI, F.; JENNINGS, N. R.; WOOLDRIDGE, M. Organisational abstractions for the analysis and design of multi-agent systems. In: SPRINGER. *Agent-oriented software engineering*. [S.l.], 2001. p. 235–251. Citado na página 3.

APÊNDICE A – Implementação dos padrões arquiteturais organizacionais para SMA

A.1 *Contract Net*

A.1.1 Contexto

O exemplo a seguir baseia-se na venda e compra de livros entre agentes vendedores (instâncias da classe *BookSellerAgent*) e agentes compradores (instâncias da classe *BookBuyerAgent*).

Sob a perspectiva do agente comprador, este recebe o título do livro a ser comprado como um argumento em linha de comando e solicita periodicamente a todos os agentes vendedores que ofereçam uma oferta. Assim que uma oferta é recebida, o agente comprador aceita e emite uma ordem de compra.

Se mais de um agente vendedor oferecer uma oferta, o comprador compara os preços e realiza a compra do livro mais barato. Tendo comprado o livro, o agente do comprador termina.

Sob a perspectiva do agente vendedor, este possui uma interface, através da classe *BookSellerGui*, na qual o usuário pode inserir novos títulos e seus respectivos preços. Ao receber solicitações para fornecer uma proposta para um livro, eles verificam se possui o título solicitado, podendo responder com o preço caso o possua, ou recusando-se a fazer uma proposta caso não o possua. Em caso de enviar uma proposta com o preço e de receber uma ordem de compra, o livro é vendido e removido do catálogo.

A.1.2 Preparação do ambiente

O exemplo foi rodado em sistema operacional Ubuntu 16.04, utilizando a IDE Eclipse Neon versão 4.6.3. Os seguintes passos devem ser seguidos para rodar este exemplo:

1. Realizar o *download* do arquivo *jadeAll.zip* em <http://jade.tilab.com/download/jade/>¹ e descompactá-lo, bem como descompactar as pastas que encontram-se dentro do mesmo;
2. Editar o arquivo *.bashrc*. Para isso, executar o comando *nano .bashrc* e adicionar as linhas abaixo ao final do arquivo. Substituir */caminho/para-o/jade* com a localização

¹ Último acesso: Junho 2017

da pasta jade no computador. Por fim, salvar a edição.

```
#jade
export JADE_LIB=/caminho/para-o/jade
export JADE_CP=$JADE_LIB/http.
jar:$JADE_LIB/iiop.
jar:$JADE_LIB/jade.
jar:$JADE_LIB/jadeTools.
jar:
$JADE_LIB/commons-codec/commons-codec-1.3.
jar
alias rJade='
java -cp $JADE_CP jade.Boot'
alias cJade='
javac -cp $JADE_CP'
```

3. Importar o código *bookTrading*, presente na pasta *jade/jade-examples/src/examples/bookTrading*, para o Eclipse.
4. Selecionar *Java Build Path > Libraries > Add External JARs*.
5. Adicionar as seguintes bibliotecas do JADE ao projeto:
 - *jade/jade-src/lib/commons-codec/commons-codec- 1.3.jar*;
 - *jade/jade-bin/lib/jade.jar*;
 - *jade/jade-examples/lib/jadeExamples.jar*.
6. Selecionar *Run > Run Configurations > Java Application*.
7. Certificar que os seguintes parâmetros estão correspondentes aos valores a seguir:
 - *Project: BookAgents*
 - *Main class: jade.Boot*
8. Na aba *Arguments* inserir, em *Program Arguments*, o seguinte texto: *-nomtp -gui bookSeller1:sell.BookSellerAgent; bookSeller2:sell.BookSellerAgent; bookBuyer1:sell.BookBuyerAgent("coleccionador")*;
9. Selecionar *Aplly > Run*.

A.1.3 Classe *BookSellerAgent*

Para criar um agente JADE, define-se, basicamente, uma subclasse da classe *jade.core.Agent*² que implementa-se o método *setup()* e os comportamentos do agente. Por meio da classe *jade.core.Agent*, os agentes podem realizar operações básicas da plataforma como registro, configuração, e troca de mensagens. Através do método *setup()*, um agente registra seus serviços nas páginas amarelas e adiciona seus comportamentos.

² <http://jade.tilab.com/doc/api/jade/core/Agent.html> (último acesso: Junho 2017)

Inicialmente, cria-se um catálogo de livros e a interface gráfica para o registro dos mesmos através da classe *BookSellerGui*.

```

37 public class BookSellerAgent extends Agent {
38     // The catalogue of books for sale (maps the title of a book to its price)
39     private Hashtable catalogue;
40     // The GUI by means of which the user can add books in the catalogue
41     private BookSellerGui myGui;
42
43     // Put agent initializations here
44     protected void setup() {
45         // Create the catalogue
46         catalogue = new Hashtable();
47
48         // Create and show the GUI
49         myGui = new BookSellerGui(this);
50         myGui.showGui();

```

Em seguida, registra-se o serviço de venda de livros (um serviço do tipo *Book-selling*) nas páginas amarelas para que esteja disponível para outros agentes. Para publicar seus serviços, um agente deve inscrever-se no agente DF (*Directory Facilitator agent*). O DF reúne e associa descrições de serviço aos seus identificadores (AID, *Agent Identifier*); de modo que os agentes visitantes possam contactá-lo à procura de agentes que prestam os serviços de que necessitam.

O padrão FIPA estabelece que cada instância de agente é identificada pelo AID (*Agent Identifier*). Na plataforma JADE, um AID é uma instância da classe *jade.core.AID*. Esta provê o método *getAID()* que retorna o nome global do agente na plataforma no seguinte formato: *<nome_local>@<nome-plataforma>*.

O agente deve, portanto, fornecer seu AID e uma lista de seus serviços fornecidos através de uma descrição adequada, como uma instância da classe *DFAgentDescription*³ e, ao final, invocar o método estático *register()* da classe *DFService*⁴.

```

52     // Register the book-selling service in the yellow pages
53     DFAgentDescription dfd = new DFAgentDescription();
54     dfd.setName(getAID());
55     ServiceDescription sd = new ServiceDescription();
56     sd.setType("book-selling");
57     sd.setName("JADE-book-trading");
58     dfd.addServices(sd);
59     try {
60         DFService.register(this, dfd);
61     }
62     catch (FIPAException fe) {
63         fe.printStackTrace();
64     }

```

³ <http://jade.tilab.com/doc/api/jade/domain/FIPAAgentManagement/DFAgentDescription.html> (último acesso: Junho 2017)

⁴ <http://jade.tilab.com/doc/api/jade/domain/DFService.html> (último acesso: Junho 2017)

Por fim, ainda no método *setup()*, são adicionados os comportamentos que atendem às consultas dos agentes compradores (através da classe *OfferRequestsServer* e os comportamentos que atendem às ordens de compra dos agentes compradores (através da classe *PurchaseOrdersServer*).

```

66      // Add the behaviour serving queries from buyer agents
67      addBehaviour(new OfferRequestsServer());
68
69      // Add the behaviour serving purchase orders from buyer agents
70      addBehaviour(new PurchaseOrdersServer());
71  }

```

Além do método *setup()*, a classe possui o método *takeDown()* que é invocado imediatamente antes de um agente ter encerrado seus serviços com objetivo de executar várias operações de limpeza.

```

73      // Put agent clean-up operations here
74      protected void takeDown() {
75          // Deregister from the yellow pages
76          try {
77              DFService.deregister(this);
78          }
79          catch (FIPAException fe) {
80              fe.printStackTrace();
81          }
82          // Close the GUI
83          myGui.dispose();
84          // Printout a dismissal message
85          System.out.println("Seller-agent "+getAID().getName()+" terminating.");
86      }

```

A classe *OfferRequestsServer* implementa um dos comportamentos da classe *BookSellerAgent*. Ao estender a classe *jade.core.behaviours.CyclicBehaviour*⁵, o método *action()* repete-se à medida que o *setup()* invoque-o. Através deste comportamento, o agente vendedor pode atender aos pedidos de oferta recebidos de agentes compradores.

Uma instância da classe *jade.lang.acl.MessageTemplate*⁶ possibilita que as mensagens recebidas pelo agente vendedor sejam filtradas. Esta classe especifica templates para serem usados ao chamar o método *receive()*. Quando um modelo é especificado, o método *receive()* retorna a primeira correspondência da mensagem, se houver, e ignora todas as mensagens não correspondentes.

Cabe ressaltar que toda mensagem corresponde a instâncias da classe *ACLMessage*⁷. Esta classe implementa o padrão FIPA-ACL e, assim, disponibiliza um conjunto de atributos que estão de acordo com as especificações FIPA.

⁵ <http://jade.tilab.com/doc/api/jade/core/behaviours/CyclicBehaviour.html> (último acesso: Junho 2017)

⁶ <http://jade.tilab.com/doc/api/jade/lang/acl/MessageTemplate.html> (último acesso: Junho 2017)

⁷ <http://jade.cselt.it/doc/api/jade/lang/acl/ACLMessage.html> (último acesso: Junho 2017)

Neste caso, espera-se receber a performativa CFP (*Call For Proposal*) onde o agentes compradores solicitam propostas aos agentes vendedores especificando o nome do livro que desejam comprar. Se o livro solicitado estiver disponível no catálogo local, o agente vendedor responde com uma mensagem *PROPOSE* especificando o preço. Caso contrário, uma mensagem *REFUSE* é enviada.

```

108     private class OfferRequestsServer extends CyclicBehaviour {
109         public void action() {
110             MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
111             ACLMessage msg = myAgent.receive(mt);
112             if (msg != null) {
113                 // CFP Message received. Process it
114                 String title = msg.getContent();
115                 ACLMessage reply = msg.createReply();
116
117                 Integer price = (Integer) catalogue.get(title);
118                 if (price != null) {
119                     // The requested book is available for sale. Reply with the price
120                     reply.setPerformative(ACLMessage.PROPOSE);
121                     reply.setContent(String.valueOf(price.intValue()));
122                 }
123                 else {
124                     // The requested book is NOT available for sale.
125                     reply.setPerformative(ACLMessage.REFUSE);
126                     reply.setContent("not-available");
127                 }
128                 myAgent.send(reply);
129             }
130             else {
131                 block();
132             }
133         }
134     } // End of inner class OfferRequestsServer

```

De modo semelhante, a classe *PurchaseOrdersServer* também especifica um comportamento do tipo *CyclicBehaviour* e aguarda o recebimento de uma performativa específica: *ACCEPT_PROPOSAL*. Esta é recebida quando o agente comprador aceita a proposta e a venda do livro pode, portanto, ser processada. O agente vendedor remove o livro comprado de seu catálogo e responde com uma mensagem *INFORM* para notificar o comprador de que a compra foi concluída.

```

144     private class PurchaseOrdersServer extends CyclicBehaviour {
145         public void action() {
146             MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.ACCEPT_PROPOSAL);
147             ACLMessage msg = myAgent.receive(mt);
148             if (msg != null) {
149                 // ACCEPT_PROPOSAL Message received. Process it
150                 String title = msg.getContent();
151                 ACLMessage reply = msg.createReply();
152
153                 Integer price = (Integer) catalogue.remove(title);
154                 if (price != null) {
155                     reply.setPerformative(ACLMessage.INFORM);
156                     System.out.println(title+" sold to agent "+msg.getSender().getName());

```

```

157         }
158         else {
159             // The requested book has been sold to another buyer in the meanwhile .
160             reply.setPerformative(ACLMessage.FAILURE);
161             reply.setContent("not-available");
162         }
163         myAgent.send(reply);
164     }
165     else {
166         block();
167     }
168 }
169 } // End of inner class OfferRequestsServer
170 }

```

A.1.4 Classe *BookBuyerAgent*

A classe *BookBuyerAgent* implementa os comportamentos relacionados à compra do livro de interesse do agente comprador. No método *setup()*, adiciona-se um comportamento do tipo *TickerBehaviour*⁸. Esta classe possibilita que execute-se um trecho de código por um período de tempo pré-definido por meio do método *onTick()*. Para este caso, foi definida a duração de um minuto.

Uma instância da classe *ServiceDescription*⁹ é criada para definir um serviço do tipo *book-selling*, para que, através de uma instância da classe *DFAgentDescription()*, sejam buscados agentes vendedores que possuam em seu catálogo o *targetBookTitle*. Por fim, os AIDs dos agentes que prestam o serviço solicitado são salvos em uma lista chamada *sellerAgents*.

```

36 public class BookBuyerAgent extends Agent {
37     // The title of the book to buy
38     private String targetBookTitle;
39     // The list of known seller agents
40     private AID[] sellerAgents;
41
42     // Put agent initializations here
43     protected void setup() {
44         // Printout a welcome message
45         System.out.println("Hallo! Buyer-agent "+getAID().getName()+" is ready.");
46
47         // Get the title of the book to buy as a start-up argument
48         Object[] args = getArguments();
49         if (args != null && args.length > 0) {
50             targetBookTitle = (String) args[0];
51             System.out.println("Target book is "+targetBookTitle);
52
53             // Add a TickerBehaviour that schedules a request to seller agents every minute
54             addBehaviour(new TickerBehaviour(this, 60000) {

```

⁸ <http://jade.tilab.com/doc/api/jade/core/behaviours/TickerBehaviour.html> (último acesso: Junho 2017)

⁹ <http://jade.tilab.com/doc/api/jade/domain/FIPAAgentManagement/ServiceDescription.html> (último acesso: Junho 2017)

```

55     protected void onTick() {
56         System.out.println("Trying to buy "+targetBookTitle);
57         // Update the list of seller agents
58         DFAgentDescription template = new DFAgentDescription();
59         ServiceDescription sd = new ServiceDescription();
60         sd.setType("book-selling");
61         template.addServices(sd);
62         try {
63             DFAgentDescription[] result = DFService.search(myAgent, template);
64             System.out.println("Found the following seller agents:");
65             sellerAgents = new AID[result.length];
66             for (int i = 0; i < result.length; ++i) {
67                 sellerAgents[i] = result[i].getName();
68                 System.out.println(sellerAgents[i].getName());
69             }
70         }
71         catch (FIPAException fe) {
72             fe.printStackTrace();
73         }

```

Em seguida, o comportamento *RequestPerformer* é adicionado ao agente comprador.

```

75         // Perform the request
76         myAgent.addBehaviour(new RequestPerformer());

```

Caso nenhum livro tenha sido especificado, o agente é finalizado.

```

80     else {
81         // Make the agent terminate
82         System.out.println("No target book title specified");
83         doDelete();
84     }
85 }

```

Logo após, é definida a classe *RequestPerformer*. Primeiramente, é enviada uma mensagem CFP para todos os agentes vendedores da lista *sellerAgents*.

```

98     private class RequestPerformer extends Behaviour {
99         private AID bestSeller; // The agent who provides the best offer
100         private int bestPrice; // The best offered price
101         private int repliesCnt = 0; // The counter of replies from seller agents
102         private MessageTemplate mt; // The template to receive replies
103         private int step = 0;
104
105         public void action() {
106             switch (step) {
107                 case 0:
108                     // Send the cfp to all sellers
109                     ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
110                     for (int i = 0; i < sellerAgents.length; ++i) {
111                         cfp.addReceiver(sellerAgents[i]);
112                     }
113                     cfp.setContent(targetBookTitle);
114                     cfp.setConversationId("book-trade");
115                     cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique value

```

```

116         myAgent.send(cfp);
117         // Prepare the template to get proposals
118         mt = MessageTemplate.and(MessageTemplate.MatchConversationId("book-trade"),
119             MessageTemplate.MatchInReplyTo(cfp.getReplyWith()));
120         step = 1;
121         break;

```

Em seguida, todas as mensagens recebidas são lidas. Aquelas cujas performativas correspondem a *PROPOSE*, têm seu conteúdo lido, o que corresponde ao preço do livro. Logo após, os preços são comparados até que se encontre o mais barato (*bestPrice*).

```

122         case 1:
123             // Receive all proposals/refusals from seller agents
124             ACLMessage reply = myAgent.receive(mt);
125             if (reply != null) {
126                 // Reply received
127                 if (reply.getPerformative() == ACLMessage.PROPOSE) {
128                     // This is an offer
129                     int price = Integer.parseInt(reply.getContent());
130                     if (bestSeller == null || price < bestPrice) {
131                         // This is the best offer at present
132                         bestPrice = price;
133                         bestSeller = reply.getSender();
134                     }
135                 }
136                 repliesCnt++;
137                 if (repliesCnt >= sellerAgents.length) {
138                     // We received all replies
139                     step = 2;
140                 }
141             }
142             else {
143                 block();
144             }
145             break;

```

Em seguida, é enviado o pedido ao vendedor que forneceu a melhor oferta (*bestSeller*) através da performativa *ACCEPT_PROPOSAL*.

```

146         case 2:
147             // Send the purchase order to the seller that provided the best offer
148             ACLMessage order = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
149             order.addReceiver(bestSeller);
150             order.setContent(targetBookTitle);
151             order.setConversationId("book-trade");
152             order.setReplyWith("order"+System.currentTimeMillis());
153             myAgent.send(order);
154             // Prepare the template to get the purchase order reply
155             mt = MessageTemplate.and(MessageTemplate.MatchConversationId("book-trade"),
156                 MessageTemplate.MatchInReplyTo(order.getReplyWith()));
157             step = 3;
158             break;

```

A performativa *INFORM* significa, neste caso, que a compra foi executada com êxito. Caso receba qualquer outra mensagem, significa que a tentativa de compra falhou.

```

159         case 3:
160             // Receive the purchase order reply
161             reply = myAgent.receive(mt);
162             if (reply != null) {
163                 // Purchase order reply received
164                 if (reply.getPerformative() == ACLMessage.INFORM) {
165                     // Purchase successful. We can terminate
166                     System.out.println(targetBookTitle+" successfully purchased from agent
167                                     "+reply.getSender().getName());
168                     System.out.println("Price = "+bestPrice);
169                     myAgent.doDelete();
170                 }
171                 else {
172                     System.out.println("Attempt failed: requested book already sold.");
173                 }
174
175                 step = 4;
176             }
177             else {
178                 block();
179             }
180             break;
181         }

```

A.1.5 Resultados da execução

A Figura 18 apresenta a inicialização do contêiner e dos serviços da plataforma JADE. A Figura 19 apresenta a plataforma JADE e a estrutura do contêiner.

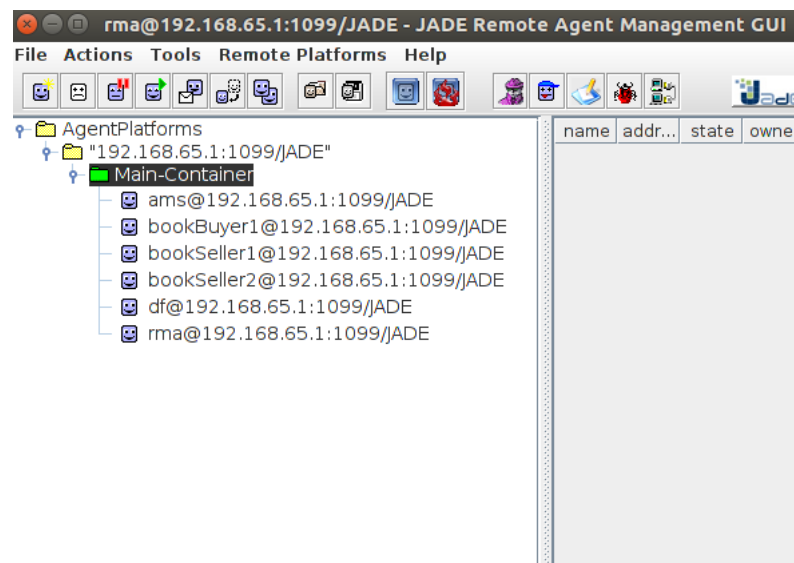


Figura 19 – Plataforma JADE

A Figura 20 apresenta o comportamento cíclico do agente comprador, que realiza várias tentativas de compra do livro a cada minuto.

```

jun 18, 2017 10:56:58 PM jade.core.Runtime beginContainer
INFORMAÇÕES: -----
This is JADE 4.4.0 - revision 6778 of 21-12-2015 12:24:43
downloaded in Open Source, under LGPL restrictions,
at http://jade.tilab.com/
-----
jun 18, 2017 10:56:59 PM jade.imtp.leap.LEAPIMTPManager initialize
INFORMAÇÕES: Listening for intra-platform commands on address:
- jicp://192.168.65.1:1099

jun 18, 2017 10:57:01 PM jade.core.BaseService init
INFORMAÇÕES: Service jade.core.management.AgentManagement initialized
jun 18, 2017 10:57:01 PM jade.core.BaseService init
INFORMAÇÕES: Service jade.core.messaging.Messaging initialized
jun 18, 2017 10:57:01 PM jade.core.BaseService init
INFORMAÇÕES: Service jade.core.resource.ResourceManagement initialized
jun 18, 2017 10:57:01 PM jade.core.BaseService init
INFORMAÇÕES: Service jade.core.mobility.AgentMobility initialized
jun 18, 2017 10:57:01 PM jade.core.BaseService init
INFORMAÇÕES: Service jade.core.event.Notification initialized
jun 18, 2017 10:57:01 PM jade.core.AgentContainerImpl joinPlatform
INFORMAÇÕES: -----
Agent container Main-Container@192.168.65.1 is ready.
-----
Hallo! Buyer-agent bookBuyer1@192.168.65.1:1099/JADE is ready.
Target book is 0 colecionador
Trying to buy 0 colecionador
Found the following seller agents:
bookSeller2@192.168.65.1:1099/JADE
bookSeller1@192.168.65.1:1099/JADE
Attempt failed: 0 colecionador not available for sale

```

Figura 18 – Saídas do console: inicialização da plataforma.

```

Trying to buy 0 colecionador
Found the following seller agents:
bookSeller2@192.168.65.1:1099/JADE
bookSeller1@192.168.65.1:1099/JADE
Attempt failed: 0 colecionador not available for sale
Trying to buy 0 colecionador
Found the following seller agents:
bookSeller2@192.168.65.1:1099/JADE
bookSeller1@192.168.65.1:1099/JADE
Attempt failed: 0 colecionador not available for sale
Trying to buy 0 colecionador
Found the following seller agents:
bookSeller2@192.168.65.1:1099/JADE
bookSeller1@192.168.65.1:1099/JADE
Attempt failed: 0 colecionador not available for sale
Trying to buy 0 colecionador
Found the following seller agents:
bookSeller2@192.168.65.1:1099/JADE
bookSeller1@192.168.65.1:1099/JADE
Attempt failed: 0 colecionador not available for sale
Trying to buy 0 colecionador
Found the following seller agents:
bookSeller2@192.168.65.1:1099/JADE
bookSeller1@192.168.65.1:1099/JADE
Attempt failed: 0 colecionador not available for sale

```

Figura 20 – Saídas do console: tentativas de compra do agente comprador

Ao inserir no catálogo o livro procurado pelo agente comprador (Figura 21), o agente comprador encontra o livro desejado e realiza sua compra (Figura 22).

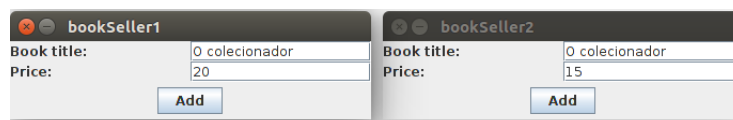


Figura 21 – Interface para catalogação dos livros.

```

Trying to buy 0 colecionador
Found the following seller agents:
bookSeller2@192.168.65.1:1099/JADE
bookSeller1@192.168.65.1:1099/JADE
0 colecionador sold to agent bookBuyer1@192.168.65.1:1099/JADE
0 colecionador successfully purchased from agent bookSeller2@192.168.65.1:1099/JADE
Price = 15
Buyer-agent bookBuyer1@192.168.65.1:1099/JADE terminating.

```

Figura 22 – Saídas do console: finalização da compra.

A.2 Master-Slave

A.2.1 Contexto

O exemplo a seguir baseia-se na Figura 4 apresentada na Seção 3. Em resumo, um agente *ConcreteCashFlowAgent* deseja obter o balanço total de pagamentos da sua caixa registradora.

A.2.2 Preparação do ambiente

O exemplo foi rodado em sistema operacional Ubuntu 16.04, utilizando a IDE Eclipse Neon versão 4.6.3. Os seguintes passos devem ser seguidos para rodar este exemplo e os exemplos a seguir:

1. Faça o download do código através do GitHub¹⁰ e descomprima a pasta;
2. Abra a IDE Eclipse Neon;
3. Clique em *File > Import > General > Existing Projects into Workspace*;
4. Selecione a pasta que contém o código;
5. Clique em *Finish*;
6. No menu *Package Explorer* navegue até a classe *Main* do package **master_slave** e clique em **Run**.

A.2.3 Classe *Main*

Basicamente, o *cashFlowAgent*, instância de *ConcreteCashFlowAgent*, precisa ler um arquivo CSV para que, em seguida, obtenha o balanço todos dos pagamentos realizados naquele dia.

```

1 package master_slave;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         //Read the csv file with the Cash Register

```

¹⁰ <https://github.com/tainarareis/MultiagentsSystemsCatalog>

```

7      ConcreteCashFlowAgent cashFlowAgent = new ConcreteCashFlowAgent();
8      cashFlowAgent.readCSV();
9
10     //Get total balance of payments
11     cashFlowAgent.getResult();
12 }
13 }

```

A.2.4 Classes *MasterAgent* *ConcreteCashFlowAgent*

A classe *MasterAgent* é interface para que a classe concreta *ConcreteCashFlowAgent* e outras concretas que virem a existir possam se basear.

```

1 package master_slave;
2
3 public interface MasterAgent {
4
5     abstract void readCSV();
6
7     abstract void getResult();
8 }

```

O *ConcreteCashFlowAgent* possui uma instância *slave* do *ConcreteSlaveAgent* para realizar tarefas de baixo nível para ele. Os métodos *readCSV()* e *getResult()* têm como objetivo controlar como e quando o *slave* executará tais tarefas.

```

1 package master_slave;
2
3 import java.io.BufferedReader;
4 import java.io.FileReader;
5 import java.io.IOException;
6
7 public class ConcreteCashFlowAgent {
8
9     ConcreteSlaveAgent slave = new ConcreteSlaveAgent();
10
11     protected void readCSV() {
12         //Inform file Path
13         String csvFile =
14             "/home/tainara/eclipse-workspace/MultiagentSystemsCatalog/src/master_slave/cash_register.csv";
15
16         //Each new line is recognized by empty string
17         String line = "";
18
19         //Comma is used as separator
20         String cvsSplitByComma = ",";
21
22         try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {
23
24             while ((line = br.readLine()) != null) {
25
26                 String[] field = line.split(cvsSplitByComma);
27                 slave.addPayment(Float.parseFloat(field[1]),
28                                 Float.parseFloat(field[2]));
29             }
30         }
31     }
32 }

```

```

29
30         System.out.println("Payment [Item= " + field[0] +
31             " , Quantity= " + field[1] +
32             " , Amount=" + field[2] + "]");
33
34     }
35
36     } catch (IOException e) {
37         e.printStackTrace();
38     }
39 }
40
41 protected void getResult() {
42     float total = slave.getTotalBalance();
43     System.out.println("The total balance of payments was $" + total);
44 }
45
46 }
```

A.2.5 Classes *SlaveAgent* e *ConcreteSlaveAgent*

A classe *SlaveAgent* possui os métodos comuns para futuros *slaves* deste contexto. A classe *ConcreteSlaveAgent* implementa tais métodos. Basicamente, o *slave* pode adicionar um pagamento (*addPayment(float quantity, float amount)*) e também obter o balanço final (método *getTotalBalance()*).

```

1 package master_slave;
2
3 public abstract class SlaveAgent {
4
5     protected abstract void addPayment(float quantity, float amount);
6
7     protected abstract float getTotalBalance();
8
9 }
```

```

1 package master_slave;
2
3 public class ConcreteSlaveAgent extends SlaveAgent {
4
5     private float totalBalance;
6
7     protected void addPayment(float quantity, float amount) {
8         float result = quantity * amount;
9         this.totalBalance += result;
10    }
11
12    protected float getTotalBalance() {
13        return this.totalBalance;
14    }
15
16 }
```

A.2.6 Resultados da execução

Apenas para exemplificar este contexto e o fluxo de eventos que ocorreria quando executada a implementação apresentada, a Figura 23 apresenta a saída no console da IDE Eclipse Neon.

```
<terminated> Main (2) [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (Jun 23, 2018, 8:39:25 PM)
The requested service can be executed by the Functional Manager 1.
Functional Manager 1: Contacting functional unit to execute service...
    Access < Tidbits > < Use Wais | Use FTP >
    Access < InfoMac >
    Get From < Seller >
    Access < News >
Service finished.
```

Figura 23 – Saídas do console: *Master-Slave* executa o serviço solicitado.

A.3 Blackboard

A.3.1 Contexto

O exemplo a seguir baseia-se no contexto apresentado na Seção 4. Em resumo, o *Blackboard* recebe uma ordem de uma *Company*, e deve se encarregar de fazer com que esta se cumpra dentro do tempo delimitado, a partir da cooperação dos agentes *SupplierAgent1* e *SupplierAgent2*.

A.3.2 Preparação do ambiente

Devem ser seguidos os mesmos passos da Seção A.4.2, porém, deve-se optar pelo *package blackboard*.

A.3.3 Classe *Blackboard*

O *Blackboard* conhece tanto a *Company* (instância *oliverEnterprise*), quanto o *ManufacturerAgent*, e os agentes *SupplierAgent1* e *SupplierAgent2*. Inicialmente, *oliverEnterprise* publica o leilão (método *publishAuction()*). O *Blackboard* passa a conhecer quais são as informações do leilão e notifica os *supplier agents*. Cada agente realiza um lance, informando quanto tempo levaria para realizar a entrega do produto solicitado pela *oliverEnterprise*. O *ManufacturerAgent*, coleta os lances e aplica seu critério de seleção para escolher qual agente deve realizar a entrega. Em seguida, o *Blackboard* prossegue controlando o fluxo de entrega até que o produto chegue ao destino final (método *controlDelivery(String selectedAgent)*).

1 `package blackboard;`
 2
 3 `import java.util.HashMap;`

```

4  import java.util.Map;
5
6  /**
7   * Blackboard is responsible for controlling the auction for supply distribution.
8   */
9  public class Blackboard {
10
11     private static Company oliverEnterprise = new Company();
12     private static ManufacturerAgent manufacturerAgent = new ManufacturerAgent();
13     private static Map auctionInformations = new HashMap<>();
14     private static Map bids = new HashMap<>();
15
16     private static SupplierAgent1 supplierAgent1 = new SupplierAgent1();
17     private static SupplierAgent2 supplierAgent2 = new SupplierAgent2();
18
19     public static void main(String[] args) {
20
21         auctionInformations = oliverEnterprise.publishAuction();
22         supplierAgent1.notifyAgent(auctionInformations);
23         supplierAgent2.notifyAgent(auctionInformations);
24
25         bids.put("Supplier Agent 1", supplierAgent1.bid());
26         bids.put("Supplier Agent 2", supplierAgent2.bid());
27
28         manufacturerAgent.applySelectionCriteria(bids);
29         controlDelivery(manufacturerAgent.getSelectedAgent());
30
31     }
32
33     private static void controlDelivery(String selectedAgent) {
34
35         Boolean deliveryProblems;
36
37         switch (selectedAgent) {
38             case "Supplier Agent 1":
39                 supplierAgent1.deliver();
40                 deliveryProblems = supplierAgent1.reportDeliveryProblems();
41                 if (deliveryProblems == false) {
42                     supplierAgent1.finishDelivery();
43                     break;
44                 } else {
45                     String alternativeAgent = manufacturerAgent.getAlternativeAgent();
46                     controlDelivery(alternativeAgent);
47                     break;
48                 }
49             case "Supplier Agent 2":
50                 supplierAgent2.deliver();
51                 deliveryProblems = supplierAgent2.reportDeliveryProblems();
52                 if (deliveryProblems == false) {
53                     supplierAgent2.finishDelivery();
54                     break;
55                 } else {
56                     String alternativeAgent = manufacturerAgent.getAlternativeAgent();
57                     controlDelivery(alternativeAgent);
58                 }
59         }
60     }
61 }

```

A.3.4 Classe *ManufacturerAgent*

O lance mais apropriado é selecionado pelo *ManufacturerAgent* com base em critérios de seleção (método *applySelectionCriteria(Map bids)*), que neste caso é o lance cujo tempo de entrega associado for o menor. Se, por ventura, um problema de entrega ocorre com um dos fornecedores, o *ManufacturerAgent* busca um fornecedor alternativo para substituí-lo.

```

1  package blackboard;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  /**
7   * ManufacturerAgent applies the selection criteria to select the supplier dynamically.
8   */
9  public class ManufacturerAgent {
10
11      Map bids = new HashMap<>();
12      String selectedAgent;
13      String alternativeAgent;
14
15      protected void applySelectionCriteria(Map bids) {
16          int sa1DeliverTime = (int) bids.get("Supplier Agent 1");
17          int sa2DeliverTime = (int) bids.get("Supplier Agent 2");
18
19          if (sa1DeliverTime < sa2DeliverTime) {
20              this.setSelectedAgent("Supplier Agent 1");
21              this.setAlternativeAgent("Supplier Agent 2");
22              System.out.println("Manufacturer Agent selected: Supplier Agent 1.");
23              System.out.println("Obs: If any problem occurs, Supplier Agent 2 will be the alternative supplier.");
24          } else {
25              this.setSelectedAgent("Supplier Agent 2");
26              this.setAlternativeAgent("Supplier Agent 1");
27              System.out.println("Manufacturer Agent selected: Supplier Agent 2.");
28              System.out.println("Obs: If any problem occurs, Supplier Agent 1 will be the alternative supplier.");
29          }
30      }
31
32      protected void setSelectedAgent(String selectedAgent) {
33          this.selectedAgent = selectedAgent;
34      }
35
36      protected String getSelectedAgent() {
37          return selectedAgent;
38      }
39
40      protected String getAlternativeAgent() {
41          System.out.println("Searching for alternative agent to end the delivery.");
42          return alternativeAgent;
43      }
44
45      protected void setAlternativeAgent(String alternativeAgent) {
46          this.alternativeAgent = alternativeAgent;
47      }
48
49  }

```

A.3.5 Classe *Company*

A *Company* publica um leilão no quadro-negro (através do método *publishAuction()*), informando: o endereço da entrega do produto (*Deliver Address*), o produto que deseja que seja entregue (*Product Name*), a quantidade do mesmo (*Quantity*), e o tempo limite que o agente deve levar para entregá-lo (*Time limit to deliver*).

```

1 package blackboard;
2
3 import java.util.Arrays;
4 import java.util.HashMap;
5 import java.util.Map;
6
7 /*
8  * A company is interested in having a certain product delivered.
9  * So, it can publish an auction at the Blackboard to ask an agent to do it.
10  */
11 public class Company {
12
13     Map auctionInformations = new HashMap<>();
14
15     public Company() {
16         this.auctionInformations.put("Company Name", "Oliver Enterprise");
17         this.auctionInformations.put("Delivery Address", "324 Street – Salt Lake City – UT");
18         this.auctionInformations.put("Product Name", "Wall paint");
19         this.auctionInformations.put("Quantity", 20);
20         this.auctionInformations.put("Time limit to deliver", 800);
21     }
22
23     protected Map publishAuction() {
24         System.out.println("Company Oliver Enterprise publishing auction informations to Blackboard...");
25         return this.auctionInformations;
26     }
27 }

```

A.3.6 Classe *SupplierAgent*

A classe *SupplierAgent* fornece os atributos e métodos comuns aos agentes fornecedores: o inventário de produtos (*inventory*), o tempo de entrega (*timeToDeliver*), as informações do leilão quando vir a receber alguma notificação sobre o mesmo (*auctionInformation*), e o controle de inventário (por meio dos métodos *addProduct(String productName, int code)* e *removeProductQuantity(String productName, int quantity)*).

```

1 package blackboard;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 /**
7  * Each Supplier Agent has an inventory with a collection of products available,
8  * and a certain time to deliver. This agent can participate of company's auctions.
9  * If an auction is published at the Blackboard and the Supplier Agent has the called
10  * product, it can offer its service to to Company.
11  */

```

```

12 public class SupplierAgent {
13
14     Map inventory = new HashMap<>();
15     int timeToDeliver;
16     Map auctionInformation = new HashMap<>();
17
18     protected void addProduct(String productName, int code) {
19         this.inventory.put(code, productName);
20     }
21
22     protected void removeProductQuantity(String productName, int quantity) {
23         int oldQuantity = (int) this.inventory.get(productName);
24         this.inventory.put(productName, oldQuantity - quantity);
25     }
26
27     protected int getTimeToDeliver() {
28         return this.timeToDeliver;
29     }
30
31 }

```

A.3.7 Classes *SupplierAgent1* e *SupplierAgent2*

Os agentes fornecedores, *SupplierAgent1* e *SupplierAgent2*, herdam os atributos e métodos da classe pai (*SupplierAgent*). Além de serem capazes de controlar seu inventário, implementam métodos para: serem notificados quando um leilão é publicado (*notifyAgent(Map auctionInformations)*), realizarem lances no leilão (*bid()*), realizarem a entrega (*deliver()*), reportarem quaisquer problemas que os impeçam de finalizar a entrega (*reportDeliveryProblems()*), e, por fim, finalizarem a entrega (*finishDelivery()*).

```

1 package blackboard;
2
3 import java.util.Map;
4
5 /**
6  * SupplierAgent1 has its own inventory and time to deliver.
7  */
8 public class SupplierAgent1 extends SupplierAgent{
9
10     public SupplierAgent1() {
11         super();
12         this.inventory.put("Roof tile", 60);
13         this.inventory.put("Brick", 58);
14         this.inventory.put("Wooden door", 85);
15         this.inventory.put("Wall paint", 74);
16         this.timeToDeliver = 90;
17         System.out.println("Supplier Agent 1 added to Blackboard.");
18     }
19
20     protected void notifyAgent (Map auctionInformations) {
21         this.auctionInformation = auctionInformations;
22         System.out.println("Supplier Agent 1: Notified.");
23     }
24
25     protected int bid () {

```

```

26     int auctiontimeLimit = (int) this.auctionInformation.get("Time limit to deliver");
27     if (this.getTimeToDeliver() <= auctiontimeLimit) {
28         System.out.println("Supplier Agent 1, Bid = " + this.getTimeToDeliver() + ".");
29         return this.getTimeToDeliver();
30     } else {
31         System.out.println("Supplier Agent 1, Bid = None.");
32         return 0;
33     }
34 }
35
36 protected void deliver () {
37     System.out.println("Supplier Agent 1 delivering " + this.auctionInformation.get("Product Name") + ".");
38 }
39
40 protected Boolean reportDeliveryProblems() {
41     System.out.println("Supplier Agent 1 facing problems to deliver.");
42     return true;
43 }
44
45 protected void finishDelivery() {
46     System.out.println("Supplier Agent 1 finished delivery on time.");
47     this.removeProductQuantity((String) this.auctionInformation.get("Product Name"), (int)
48         this.auctionInformation.get("Quantity"));
49 }
50 }

```

```

1 package blackboard;
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 /**
7  * SupplierAgent2 has its own inventory and time to deliver.
8  */
9 public class SupplierAgent2 extends SupplierAgent{
10
11     public SupplierAgent2() {
12         super();
13         this.inventory.put("Tile", 95);
14         this.inventory.put("Sand", 86);
15         this.inventory.put("Eletrical wiring", 21);
16         this.inventory.put("Wall paint", 74);
17         this.timeToDeliver = 100;
18         System.out.println("Supplier Agent 2 added to Blackboard.");
19     }
20
21     protected void notifyAgent (Map auctionInformations) {
22         this.auctionInformation = auctionInformations;
23         System.out.println("Supplier Agent 2: Notified.");
24     }
25
26     protected int bid() {
27         int auctiontimeLimit = (int) this.auctionInformation.get("Time limit to deliver");
28         if (this.getTimeToDeliver() <= auctiontimeLimit) {
29             System.out.println("Supplier Agent 2, Bid = " + this.getTimeToDeliver() + ".");
30             return this.getTimeToDeliver();
31         } else {
32             System.out.println("Supplier Agent 2, Bid = None");

```

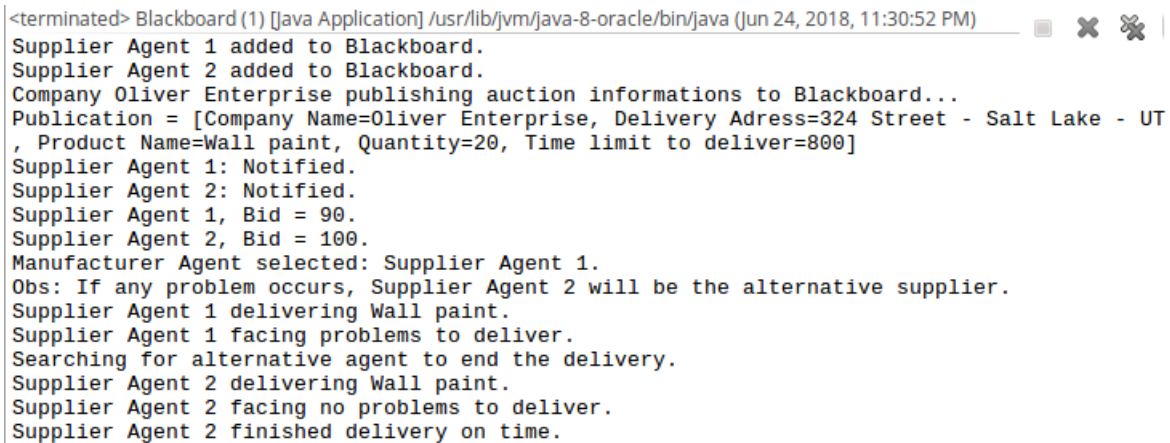
```

33         return 0;
34     }
35 }
36
37 protected void deliver() {
38     System.out.println("Supplier Agent 2 delivering " + this.auctionInformation.get("Product Name") + ".");
39 }
40
41 protected Boolean reportDeliveryProblems() {
42     System.out.println("Supplier Agent 2 facing no problems to deliver.");
43     return false;
44 }
45
46 protected void finishDelivery() {
47     System.out.println("Supplier Agent 2 finished delivery on time.");
48 }
49
50 }

```

A.3.8 Resultados da execução

De modo a exemplificar este contexto e o fluxo de eventos que ocorreria quando executada a implementação apresentada, é apresentada a saída no console da IDE Eclipse Neon (Figura 24).



```

<terminated> Blackboard (1) [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (Jun 24, 2018, 11:30:52 PM)
Supplier Agent 1 added to Blackboard.
Supplier Agent 2 added to Blackboard.
Company Oliver Enterprise publishing auction informations to Blackboard...
Publication = [Company Name=Oliver Enterprise, Delivery Address=324 Street - Salt Lake - UT
, Product Name=Wall paint, Quantity=20, Time limit to deliver=800]
Supplier Agent 1: Notified.
Supplier Agent 2: Notified.
Supplier Agent 1, Bid = 90.
Supplier Agent 2, Bid = 100.
Manufacturer Agent selected: Supplier Agent 1.
Obs: If any problem occurs, Supplier Agent 2 will be the alternative supplier.
Supplier Agent 1 delivering Wall paint.
Supplier Agent 1 facing problems to deliver.
Searching for alternative agent to end the delivery.
Supplier Agent 2 delivering Wall paint.
Supplier Agent 2 facing no problems to deliver.
Supplier Agent 2 finished delivery on time.

```

Figura 24 – Saídas do console: *Blackboard* controla o solicitado.

A.4 MACRON

A.4.1 Contexto

O exemplo a seguir baseia-se no contexto apresentado na Seção 6. Em resumo, um agente *QueryManagerAgent* deseja obter um serviço, especificamente, o serviço *Get Online Reviews*. O diagrama apresentado na Seção 14 é base da implementação deste contexto.

A.4.2 Preparação do ambiente

Devem ser seguidos os mesmos passos da Seção A.4.2, porém, deve-se optar pelo *package* **macron**.

A.4.3 Classe *Main*

A *Main*, nesta implementação, apenas recebe as requisições dos *Query Manager Agents*, e, no caso, apenas do *QueryManagerAgent*. Quando o serviço que este busca está disponível, é delegado que seja executado.

```

1 package macron;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         QueryManagerAgent queryManager = new QueryManagerAgent();
7         Boolean isServiceAvailable;
8
9         //QueryManager wants to retrieve all online reviews about some product,
10        isServiceAvailable = queryManager.findFunctionalManager("Get Online Reviews");
11
12        //The query can only occur if there is an agent that can execute it.
13        if (isServiceAvailable == true) {
14            queryManager.executeQuery();
15        }
16    }
17 }
```

A.4.4 Classe *QueryManagerAgent*

O *QueryManagerAgent* consulta o *OrganizationChartManagerAgent* para verificar se há algum agente que ofereça o serviço que busca, através da função *findFunctionalManager(String requestedService)*. Também delega ao *OrganizationChartManagerAgent* que controle a execução do serviço quando o mesmo é encontrado e está disponível, através da função *executeQuery()*.

```

1 package macron;
2
3 public class QueryManagerAgent {
4
5     String requestedService;
6     OrganizationChartManagerAgent ocmAgent = new OrganizationChartManagerAgent();
7
8     //Verify if there is any functional manager that can offer the requested service
9     protected Boolean findFunctionalManager(String requestedService) {
10        return this.ocmAgent.findFunctionalManager(requestedService);
11    }
12
13    //Delegates ocmAgent to control the query execution
14    protected void executeQuery() {
15        this.ocmAgent.ordenateServiceExecution();
16    }
17 }
```

```

16     }
17 }

```

A.4.5 Classe *OrganizationalChartManagerAgent*

O *OrganizationalChartManagerAgent* procura *Functional Managers* através do método *findFunctionalManager(String requestedService)* cujas *Functional Units* possam desempenhar o serviço *Get Online Reviews*. Sua principal função é conhecer os gerentes funcionais, sua disponibilidade, suas capacidades, e assim pode ser consultado pelo *Query Manager* a qualquer instante.

```

1  package macron;
2
3  import java.util.HashMap;
4  import java.util.Map;
5
6  public class OrganizationChartManagerAgent {
7
8      //OCM known agents
9      private FunctionalManager1Agent fm1Agent = new FunctionalManager1Agent();
10     private FunctionalManager2Agent fm2Agent = new FunctionalManager2Agent();
11
12     //All services this agents offers
13     private Map allServices = new HashMap<>();
14
15     //The agent localized to execute the service to the query manager
16     private String functionalManagerLocalized;
17
18     public OrganizationChartManagerAgent() {
19         this.allServices.put(fm1Agent.getService(), fm1Agent.getName());
20         this.allServices.put(fm2Agent.getService(), fm2Agent.getName());
21     }
22
23     protected Boolean findFunctionalManager(String requestedService) {
24         this.functionalManagerLocalized = (String) this.allServices.get(requestedService);
25         if (this.functionalManagerLocalized != null) {
26             System.out.println("The requested service can be executed by the " + this.functionalManagerLocalized +
27                               ".");
28             return true;
29         } else {
30             System.out.println("The requested service has no functional manager that can do it.");
31             return false;
32         }
33     }
34
35     protected void ordenateServiceExecution() {
36         if (this.functionalManagerLocalized == "Functional Manager 1") {
37             this.fm1Agent.executeService();
38         } else if (this.functionalManagerLocalized == "Functional Manager 2") {
39             this.fm2Agent.executeService();
40         }
41     }
42 }

```

A.4.6 Classe *FunctionalManager1Agent*

A pedido do *QueryManagerAgent*, o *FunctionalManager1Agent* atribui tarefas a agentes dentro da sua *Functional Unit*. Esta é responsável por alocar os recursos necessários para realizar o serviço. Neste caso, os recursos são: *Tidbits*, *Wais*, *FTP*, *InfoMac*, *Seller*, e *News*. Todos são consultados para recuperar os *online reviews* solicitados. Caso houvesse sido solicitado um outro serviço, como, por exemplo, *Get Published Reviews*, seriam necessários os recursos *Tidbits*, *Library*, *Online Sources*, *Fax*, e *Seller*. Ou seja, pra executar diferentes serviços, seriam necessários adotar os mesmos recursos. Cabe ao *Functional Manager*, na estrutura MACRON tomar as decisões a cerca de como fazer a melhor utilização de tais recursos de modo dinâmico.

```

1 package macron;
2
3 public class FunctionalManager1Agent extends FunctionalManagerAgent {
4
5     public FunctionalManager1Agent() {
6         this.name = "Functional Manager 1";
7         this.service = "Get Online Reviews";
8     }
9
10    protected void executeService() {
11        System.out.println("Functional Manager 1: Contacting functional unit to execute service...");
12        this.allocateResources();
13        this.getOnlineReviews();
14        System.out.println("Service finished");
15    }
16
17    protected void allocateResources() {
18        //Strategy of resources allocation here
19    }
20
21    protected void getOnlineReviews() {
22        System.out.println("Access Tidbits < Use Wais | Use FTP >");
23        System.out.println("Access InfoMac");
24        System.out.println("Get From Seller");
25        System.out.println("Access News");
26    }
27
28 }
```

A.4.7 Resultados da execução

Apenas para exemplificar este contexto e o fluxo de eventos que ocorreria quando executada a implementação apresentada, a Figura 25 apresenta a saída no console da IDE Eclipse Neon.

```

<terminated> Main (2) [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (Jun 23, 2018, 8:39:25 PM)
The requested service can be executed by the Functional Manager 1.
Functional Manager 1: Contacting functional unit to execute service...
    Access < Tidbits > < Use Wais | Use FTP >
    Access < InfoMac >
    Get From < Seller >
    Access < News >
t Service finished.

```

Figura 25 – Saídas do console: *FunctionalManager1Agent* executa o serviço solicitado.

A.5 Generalized Partial Global Planning

A seguir é apresentado um pseudo-código que simula o fluxo entre os agentes simulando as ações descritas na Seção 7.

```

1  package gpgp;
2
3  public class GPGP {
4
5      public static void main(String[] args) {
6
7          //At time=0 each agent creates its task structure
8          Agent agentA = new Agent();
9          agentA.addTask(A11, 3); //addTask(taskName, duration)
10         agentA.addTask(A12,3);
11
12         Agent agentB = new Agent();
13         agentB.addTask(B11,2);
14
15         Agent agentC = new Agent();
16         agentC.addTask(C11, 4);
17         agentC.addTask(C12, 1);
18
19         //At time=1, each agent makes its own schedule
20         agentA.schedule(A11, 1, 3); //schedule(taskName, startTime, endTime)
21         agentA.schedule(A12, 4, 6);
22         agentB.schedule(B11, 1, 2);
23         agentC.schedule(C12, 1, 1);
24
25         //At time=2, gpgp plans the tasks considering the global needs
26         GeneralizedPartialGlobalPlanner gpgp = new GeneralizedPartialGlobalPlanner();
27         winner = gpgp.generatePlanning(1, 2) //generates the winner for the interval from time 1 to 2
28         gpgp.notifyAgents(agentA, agentB, agentC); //notifies all agents about the winner of the interval
29
30         //Consider the winner was agentB...
31         agentB.executeTask(B11);
32         agentA.busyInterval(1, 2); //busyInterval(startTime, endTime)
33         agentC.busyInterval(1, 2);
34
35         //agentA reschedules, and agentC schedules task C11
36         agentA.schedule(A11, 5, 7);
37         agentA.schedule(A12, 2, 4);
38         agentC.schedule(C11, 3, 6);
39
40         //At time=3
41         winner = gpgp.generateaPlanning(3, 4);
42         gpgp.notifyAgents(agentA, agentB, agentC);

```

```
43
44     //Consider the winner was agentA with task A11...
45     agentA.executeTask(A11);
46     agentB.busyInterval(5, 7);
47     agentC.busyInterval(5, 7);
48
49     //agentC reschedules
50     agentC.schedule(C11, 7, 10);
51
52     //At time=4
53     winner = gpgp.generatePlanning(7, 10);
54
55     //Consider the winner was agentA with task C11...
56     agentC.executeTask(C11);
57
58     }
59 }
```
