3
Trabalhos Relacionados

Neste capítulo são apresentados trabalhos relacionados a esta dissertação. Todos eles pretendem acrescentar formas que garantem a qualidade de agentes de software desenvolvidos. Esses trabalhos formam uma visão geral do problema e suas possíveis soluções. Alguns deles podem ser usados de forma complementar a esta proposta de dissertação.

3.1
Teste Unitário de Agentes Usando Agentes Mock e Aspectos

Este trabalho [6] propõe uma solução para o teste da menor unidade de construção de sistemas multiagentes, o próprio agente. A idéia é poder isolar e verificar se um agente responde a suas especificações sob condições normais e anormais de execução. Nesse contexto, a proposta envolve uma abordagem para teste unitário utilizando agentes *mock* e aspectos, assim como uma técnica para enumerar uma lista de possíveis situações passíveis de erro. Essa lista permite a construção de casos de teste mais eficientes, ao considerar casos nos quais os erros são mais suscetíveis.

3.1.1
Abordagem de Teste Unitário

Essa abordagem considera a seguinte questão: como um agente pode ser isolado para que nele sejam aplicados casos de teste? Esse agente possui a denominação de AUT (*Agent Under Test*). Para isso é necessário que todos os outros agentes que interagem com ele sejam simulados de alguma forma. A estratégia mais comum é criar versões “fantoches” (*stubs*) dos outros agentes, onde resultados previamente definidos são retornados.

No entanto, o isolamento do agente pode ser feito utilizando-se o conceito de *mocks*, apresentado por [31] para o teste de objetos. A idéia central de *mock* é bem parecida com a de *stub*. Em contra partida, a inclusão de assertivas para instrumentar as interações com seus vizinhos a torna diferente. Nesse sentido, esse trabalho define o conceito de agentes *mock*.

Um agente *mock* é um agente como qualquer outro que apenas se comunica com o agente AUT. Seu único objetivo é testar o agente AUT. Assim, a sua implementação é bem parecida com a de um script de teste, definindo as mensagens que ele deve enviar e as que espera receber.

A abordagem de teste unitário que utiliza agentes *mock* possui cinco participantes, conforme podemos ver na figura 3.1. *Test Suite*, Casos de Teste, Agente de Teste, Agente Mock e Monitor de Agentes.

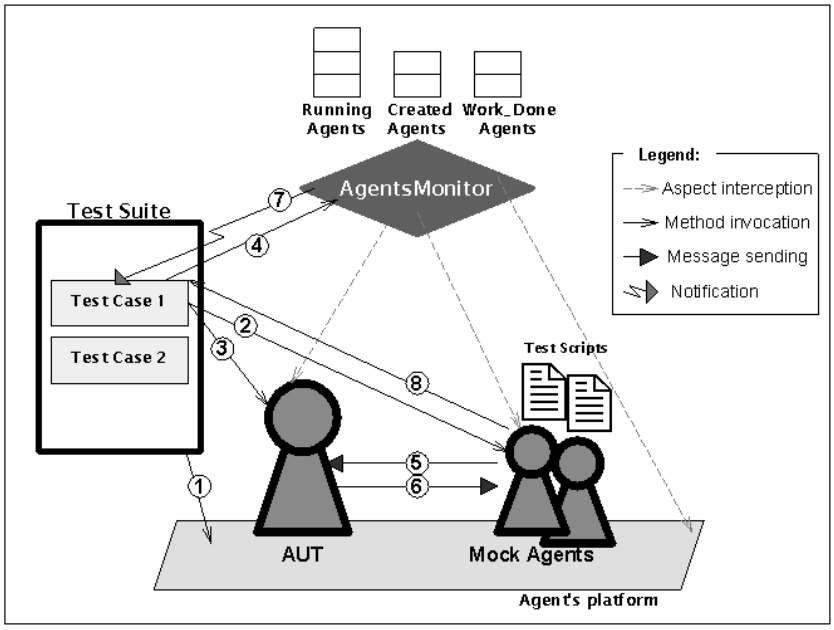

Figura 3.1: Workflow entre os Participantes do Teste Unitário

3.1.2
Projeto de Casos de Teste Baseados em Agentes Mock

Mais importante do que desenvolver casos de teste é fazer casos de teste bastante efetivos. Na maioria das vezes, teste automatizado não pode detectar todos os erros [23]. Portanto, é importante tentarmos fazer os casos de teste o mais completos possíveis, considerando os casos em que esperamos que a probabilidade de erro seja maior.

A proposta inclui uma técnica para projeto de casos de teste baseada em palpites de erros. A idéia central é enumerar uma lista de situações passíveis de erro e projetar os casos de teste baseando-se nessa lista. O processo é descrito a seguir [1]:

```
1. Para cada AUT.
    1.1 Liste o conjunto de papeis desempenhados por ele.
```

[1] adaptação para o processo proposto em inglês

```
2. Para cada papel desempenhado pelo AUT.
    2.1 Liste os outros papeis que interagem com o AUT.
3. Para cada papel que interage com o AUT.
    3.1 Implemente um agente mock que codifica um cenário de
    sucesso.
    3.2 Liste os possíveis cenário anormais que os agentes mock
    podem participar.
    3.3 Implemente em cada agente mock um plano que codifica cada
    cenário anormal.
```

Listagem 3.1: Processo de Enumeração de Situações Passíveis de Erro

3.2 Agente de Teste para o Teste de Agentes e Comunidades

Em [28] uma proposta é apresentada para o teste de agentes e comunidades de agentes. O trabalho é motivado pela dificuldade de se testar agentes de software visto que as técnicas tradicionais não se aplicam.

A dificuldade para o teste de agentes é associada ao fato da comunicação ser feita pela troca de mensagens e não pela chamada de métodos como tradicionalmente encontramos em componentes de software. Outro agravante seria a autonomia e o paralelismo, fazendo com que um agente possa estar correto sozinho e incorreto dentro de uma comunidade. Finalmente, agentes podem ser programados com aprendizado, o que pode fazer com que o mesmo caso de teste obtenha resultados diferentes.

Nesse contexto, o trabalho argumenta que o teste individual de um agente difere do teste de uma comunidade de agentes. Ao testar um único agente, o desenvolvedor está interessado em verificar suas funcionalidades e como ele se comporta para um conjunto de mensagens, entradas do ambiente e condições de erro. Ao testar uma comunidade, o desenvolvedor preocupa-se com a correta interação dos agentes, isto é, se eles estão coordenados e suas mensagens estão corretas.

Uma análise é feita quanto as abordagens utilizadas para realizar testes individuais de agentes e comunidades. Primeiramente é argumentado que para que um agente seja testado individualmente é necessário uma simulação dos agentes que interagem com ele. Para isso outros agentes são implementados com comportamentos predefinidos.

Uma análise também é feita quanto aos requisitos necessários para o teste de comunidades de agentes. Assim, uma proposta de teste deveria poder verificar se os agentes inseridos em determinada comunidade estão recebendo mensagens dos agentes corretos, provendo as respostas corretas e interagindo corretamente com o ambiente.

Para que o trabalho dos desenvolvedores seja facilitado, argumenta-se que a existência de um sistema para visualização de troca de mensagens em tempo de execução poderia ajudar. Outra proposta seria a existência de um controle centralizado para começar, paralisar e controlar a velocidade de execução dos agentes. Com isso, a atividade dos agentes poderia ser monitorada.

3.2.1 O Agente de Teste

A proposta deste trabalho envolve o projeto de um agente de testes que compromete-se a resolver os problemas apresentados anteriormente.

O agente de teste receberia especificações de mensagens do usuário ou de alguma especificações formal. A partir desta informação, scripts de teste poderiam ser gerados para avaliar cada um dos agentes ou comunidades em questão.

O teste de agentes e comunidades feito pelo agente de testes seria feito por:

- 1. Teste individual de um agente quanto a sua habilidade de enviar e receber mensagens específicas;
- 2. Teste individual de um agente quanto a sua habilidade de lidar com mensagens válidas e inválidas;
- 3. Teste de uma comunidade considerando todas as mensagens definidas assim como um numero representativo de mensagens inválidas;
- 4. Manutenção das especificações oficiais das mensagens de uma comunidade;
- 5. Manutenção das especificações das mensagens nos documentos de projeto;
- 6. Coleta de métricas quanto ao uso da rede, comunicação inter-agente e outros itens relacionados a escalabilidade;
- 7. Monitoramento de sistemas multiagentes para detecção de possíveis erros e problemas de performance.

Como futuro trabalho para o agente de testes é planejado um desenvolvimento que facilite o teste de implementações que envolvam algum tipo de inteligência.

3.3 Framework de Testes para agentes PASSI

Em [5] é apresentada solução para a implementação e teste de sistemas multi agentes. O trabalho vai mais além descrevendo diagramas para a notação do comportamento de agentes que não serão levados em conta nessa seção. Ambos são contribuições para a metodologia PASSI [19]. A idéia da metodologia é fornecer apoio prático e teórico para o projeto, implementação e teste de sistemas multiagentes.

Dentro do escopo do PASSI, o teste de agentes é dividido em duas categorias: teste unitário do agente e teste da sociedade. O primeiro se preocupa em verificar o comportamento de cada agente baseando-se no requisitos do sistema. O segundo verifica se a integração entre os diversos agentes do sistema está funcionando corretamente. Essa abordagem se compromete apenas com o primeiro tipo de teste citado.

Ao desenvolver testes unitários para agentes, deve-se considerar o encapsulamento. Habitualmente, as implementações de agentes de software se comunicam através de troca de mensagens. Portanto, testar seu comportamento implica em começar uma interação com o agente e verificar o resultado obtido. Esse tipo de teste é, geralmente, desempenhado por agentes *stubs*.

3.3.1 O Framework de Testes

O objetivo do framework apresentado não é prover uma ferramenta para teste exaustivo mas, sim, um framework que facilite os desenvolvedores a construir testes de maneira barata, com pouco esforço e de forma incremental. O conceito de framework é aplicado já que a solução apresenta um modelo com uma implementação parcial, provendo suporte ao desenvolvedor para criação e execução uniforme e automatizada dos testes. Uniforme uma vez que o desenvolvedor é obrigado a seguir uma arquitetura proposta pelo framework e automatizada pois os testes devem ser capazes de avaliar seu sucesso ou falha automaticamente, sem intervenção humana durante sua execução.

O desenvolvimento do framework foi feito sobre a plataforma JADE [4]. Portanto, os testes suportam apenas agentes do tipo JADE. O framework permite aos desenvolvedores a criação de testes em diversos níveis, atuando apenas como um executor e ferramenta de visualização de resultados.

O framework é baseado num modelo em dois níveis. O primeiro identifica o agente como uma entidade atômica. A preocupação nesse caso é com correta execução das atividades comuns desempenhadas pelo agente. O segundo nível é voltado para o teste de tarefas específicas do agente. Nesse contexto, o framework

define *agent-test* como o conjunto de testes relacionados com as atividades comuns de um dado agente e *task-test* para o conjunto de testes relacionados à tarefas específicas do agente.

Na figura 3.2 podemos ver um diagrama de classes ilustrando as classes responsáveis pela implementação do modelo descrito.

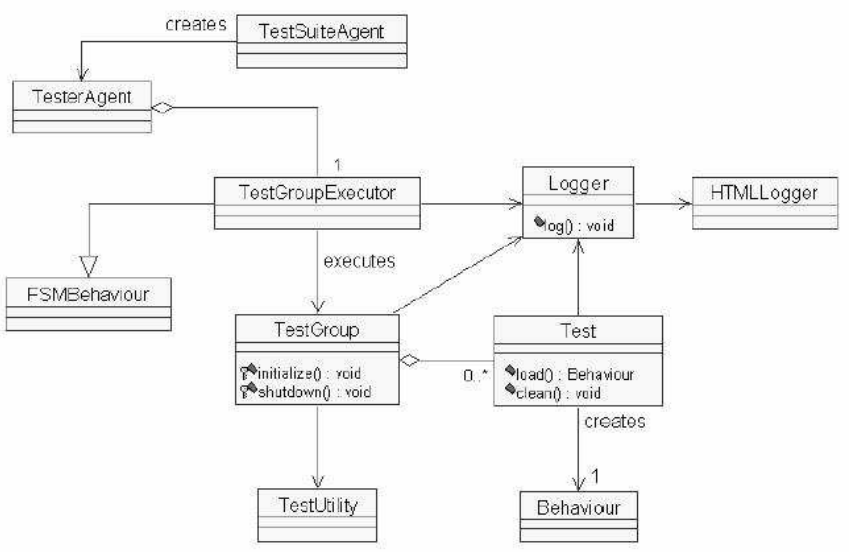

Figura 3.2: Classes do Framework de Teste

3.3.2
Agente para execução de pacotes de teste

A execução de testes individuais ou em grupo pode ser obtida com o lançamento do agente de teste correspondente. Uma forma mais conveniente é a utilização do agente para execução de pacotes de teste, implementado pela classe *TestSuitAgent*. Esse agente cria os agentes de teste adequados e delega a execução dos testes. Durante a atividade de testes, os agentes de teste enviam mensagens informando os resultados dos testes, detalhando eventualmente causas de falhas.

A implementação também fornece uma interface gráfica para a verificação do progresso e sucesso dos testes. Para isso, uma barra de progresso é exibida. Ela apresenta uma cor verde quando os testes são bem sucedidos, mas adquire cor vermelha na presença de falhas.

A figura 3.3 apresenta a interface gráfica para execução dos agentes:

3.4
Uso de XP para desenvolvimento de Sistemas Multiagentes

Em [18] uma crítica é feita quanto a complexidade que abordagens de engenharia de software trazem para o desenvolvimento de sistemas multi agentes.

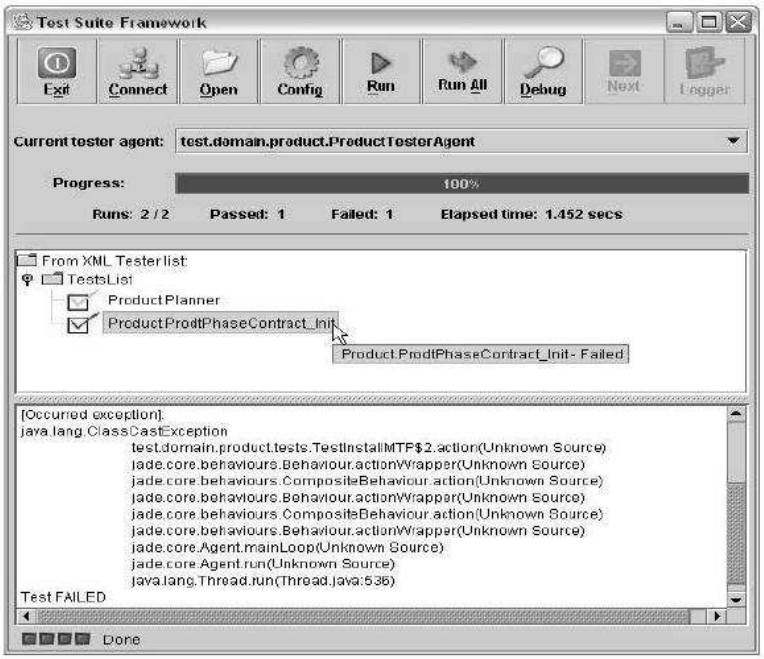

Figura 3.3: Interface Gráfica para Execução de Pacote de Testes

Segundo a crítica, muitas metodologias empregadas seguem abordagens em estilo cascata, nelas as fases de analise, projeto, implementação e teste são claramente separadas, tornando o desenvolvimento de um sistema multi agente dependente de linguagens, processos e ferramentas. Muitas vezes a complexidade gerada pelas metodologias não justifica o desenvolvimento do sistema.

O trabalho chama a atenção para as chamadas metodologias ágeis que tentam solucionar o problema com pequenas iterações e produção precoce de código executável. Para isso, ao invés de se preocupar com a boa formação das fases de desenvolvimento, essas metodologias preocupam-se em manter um código fonte flexível.

Baseando-se na metodologia ágil XP [3], esse trabalho propõe o desenvolvimento de sistemas multi agentes segundo os valores da metodologia XP. Dessa forma, com o intuito de simplificar o desenvolvimento, foi proposto o uso de somente dois modelos: o código fonte para execução dos agentes (incluindo os casos de teste) e um modelo de processo, que descreve os cenários da aplicação. Será aqui apresentado somente o primeiro modelo que diz respeito aos testes implementados.

3.4.1 Teste de agentes

A abordagem para a implementação dos agentes preocupou-se com uma plataforma leve e simples que focasse em características principais de agentes, particularmente, na comunicação por mensagens. A figura 3.4 representa o modelo

dos agentes utilizados.

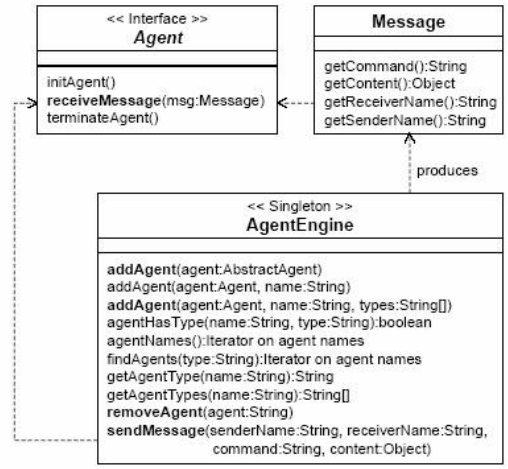

Figura 3.4: Estrutura Simples para Implementação de Agentes

Para que um agente seja implementado seguindo essa plataforma é necessária a extensão da interface *Agent*. As mensagens encapsulam o remetente e destinatário, possuem uma performativa (comando) e um conteúdo em forma de objeto. Os agentes somente podem interagir através do mecanismo *AgentEngine*.

O desenvolvimento dos testes seguiu as recomendações XP, utilizando, portanto, testes de unidade, avaliando quando um módulo implementa suas funcionalidades como os esperado. Nesse sentido, os casos de teste são implementados independentemente para cada agente, onde os agentes adjacentes são simulados por agentes *stubs*.

Segundo a figura 3.5, podemos ver a estratégia utilizada que se beneficia do framework junit [33]:

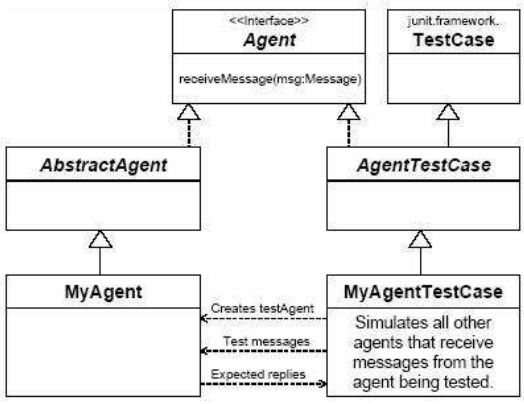

Figura 3.5: Teste de Agentes com JUnit

Os casos de teste para os agentes estendem da classe *TestCase* do JUnit. Além disso, existe uma implementação feita da interface *Agent*, podendo assumir

o papel de um ou mais agentes no sistema. Essa estratégia permite o teste de troca de mensagens, avaliando as respostas esperadas e a troca de estados do agente a ser testado. Outro benefício é que uma única classe de teste concentra os testes unitários para um agente.

3.5 SUNIT

Este trabalho apresenta uma abordagem de desenvolvimento orientado a testes [2] para agentes, juntamente, com um framework de teste, chamado Sunit, que provê um implementação de suporte para a proposta [34].

O desenvolvimento orientado a testes original [17] propõe que os testes automatizados de unidade devem ser definidos antes da implementação das funcionalidades que eles testam. A aplicação desta metodologia para testes de sistemas multiagentes requer uma certa adaptação. O primeiro conceito é o cenário. Cada cenário é um processo elementar que produz algum valor ao iniciador do processo. Em implementações multiagentes os cenários são conceitualmente diferentes pois temos que considerar que agentes autônomos podem iniciá-los. Outra adaptação seria justamente a troca de objetos pelos agentes.

3.5.1 O Processo para Desenvolvimento Orientado a Testes para Agentes

O processo para o desenvolvimento orientado a testes para agentes possui cinco passos como ilustrado na figura 3.6.

Figura 3.6: Processo para Desenvolvimento Orientado a Testes de Agentes

No primeiro passo é escolhido um papel que será desempenhado por um agente. No inicio do desenvolvimento de um cenário, o desenvolvedor identifica o papel que inicia o cenário para começar o ciclo do processo. No segundo passo, um

[2] Adaptação para português para sigla TDD - Test Driven Development

agente que desempenhe o papel escolhido é selecionado e são identificadas algumas de suas tarefas que satisfazem a responsabilidade de desempenhar o papel escolhido no cenário em questão. Nesse estágio o desenvolvedor esboça a estrutura do plano inicial que executará as tarefas escolhidas.

O terceiro passo é referente ao desenvolvimento de testes que validarão as tarefas escolhidas no passo anterior. Esses testes podem ter três níveis diferentes:

– Nível de Teste Estrutural - As tarefas são desenvolvidas segundo uma rede hierárquica de tarefas [20]. Essas estruturas podem ser bastantes complexas. Por isso, é necessário que o desenvolvedor garanta a correção dessa estrutura no primeiro nível de testes;
– Nível de Teste de Ações - Cada plano executado por agentes pode possuir diversos ações. Essas ações devem ser testadas separadamente;
– Nível de Teste Fluidal - Cada agente interage com outros agentes e organizações. Testes devem ser desenvolvidos para garantir que essas interações estão corretas.

No passo quatro, o desenvolvedor implementa as tarefas escolhidas no passo dois e para as quais foram definidos testes no passo três. A tarefas, com seus testes e estruturas, podem ser bastantes difíceis de se identificar em um único passo. Durante a implementação da estrutura do plano, desenvolvedores podem identificar requisitos adicionais para a expansão da estrutura. Nesse ponto, eles podem voltar ao passo três para desenvolver novos testes e avançar novamente para implementar a estrutura no passo quatro.

No quinto e último passo, o desenvolvedor aprimora as decisões de projeto iniciais, como, por exemplo, a estrutura de planos. Ela pode ser transferida para uma melhor que permita a utilização de tarefas já definidas.

3.5.2 O Framework de Testes SUnit

O framework SUnit é uma extensão do framework JUnit [33]. Ele apresenta um ambiente de testes que suporta a criação e execução de testes de maneira uniforme e automática.

O framework SUnit foi desenvolvido para a plataforma Seagent [9]. Assim como outras plataformas conhecidas RETSINA[26] e DECAF[13], os agentes para plataforma Seagent, suportam redes hierárquicas de tarefas para a criação de planos. O framework de teste permite a criação dos testes para planos, confirmação de estado interno dos planos e validação de execução do plano em tempo de execução. O framework também utiliza o lançamento de eventos lançados pela plataforma Seagent para testar a fluidez dos planos.

Por ser uma extensão do framework JUnit, a implementação dos casos de teste devem estender a classe *SeagentTestCase* que, por sua vez, estende a classe *TestCase* do framework JUnit. Assim como descrito no processo para desenvolvimento orientado a testes, o terceiro passo possui três níveis de teste: Estrutural, Ações e Fluidal. A framework é composto por três módulos que visam suportar esses tipos de teste. Cada módulo possui uma classe de teste estendida de *SeagentTestCase*: *SeagentStructuralTestCase*, *SeagentActionTestCase* e *SeagentFlowTestCase*.

O teste estrutural preocupa-se com precisão de tarefas complexas, consistência e integridade entre elas. Para cada tarefa complexa é recomendado que um teste estrutural seja aplicado estendendo-se a classe *SeagentStructuralTestCase* e aplicando neste teste a classe da tarefa a ser testada pelo método *setBehaviourClass()*.

O teste de ações examina resultados e mensagens definidos previamente pelo desenvolvedor. O caso de teste aqui é implementado a partir da extensão da classe *SeagentActionTestCase*.

Finalmente, o teste de fluidez, implementado a partir da extensão da classe *SeagentFlowTestCase* verifica a consistência da ordem de execução ou status das tarefas sendo executadas. Outra verificação é a comparação entre a troca de mensagens com conjuntos pré-definidos de mensagens.