2
Fundamentação Teórica

Neste capítulo são apresentados conceitos e abordagens utilizadas e mencionadas no documento. Inicialmente são apresentados os conceitos usados na tese, seguida por uma visão geral da linguagem de modelagem UML Testing Profile, o paradigma de sistemas multiagentes, sistemas autoadaptativos que realizam autoteste e o framework JADE que é voltado para a criação de agentes de software.

2.1.
Conceitos Gerais da Área de Testes de Software

A terminologia de teste aplicada na descrição dessa tese está em conformidade com especificações e padrões reconhecidos e adotados pela academia e indústria. As referências utilizadas foram as seguintes: glossário modelo de maturidade TMMi (*Test Maturity Modelo Integration*) (TMMi, 2012), vocabulário usado no IEEE 829-2008 (829-2008, 2012) (responsável por descrever padrões de documentação para testes de software), glossário do órgão internacional de certificações da área de teste chamado de ISTQB *(International Software Testing Qualifications Board*) (ISTQB, 2012), e nas definições adotadas pela linguagem de modelagem UML Testing Profile (UTP, 2012; Baker et al., 2007).

Caso de Teste (*Test case*): Um conjunto de valores de entrada, pré-condições de execução, resultados esperados, pós-condições de execução, desenvolvido para um objetivo particular ou condição de teste, como, por exemplo, exercitar um caminho particular do sistema em teste ou verificar o comportamento de algum requisito específico (TMMi, 2012).

Componente (*Component*): Menor parte do sistema que pode ser testada de forma isolada (ISTQB, 2012; TMMi, 2012). Um componente pode ser composto por diversos subcomponentes, que juntos permitem a execução de um ou mais testes.

Cenário de Teste (*Test scenario ou Test context*): Contexto de teste composto por um conjunto de estados e dados necessários para que os casos de teste possam ser executados. O foco central é garantir que o componente ou sistema possa funcionar em conjunto com novos ou existentes procedimentos de negócios de usuários ou procedimentos operacionais (ISTQB, 2012).

Ferramenta de Teste (*Test tool*): Produto de software que dá suporte a uma ou mais atividades de um teste, entre elas, planejamento e controle, especificação, construção de arquivos iniciais e dados, execução e análise de testes (ISTQB, 2012).

Ferramenta de gerenciamento de teste (*Test Management tool*): Ferramenta que provê suporte para o gerenciamento de teste e controla parte de um processo de teste. Geralmente oferece diversos recursos, como, por exemplo, a possibilidade de definir o escalonamento de execução dos testes, organizar os logs com os resultados dos testes, reportar as atividades de teste realizadas, gerenciar incidentes identificados, etc (ISTQB, 2012).

Log de teste (*Test log*): Registro cronológico de detalhes relevantes sobre a execução de testes (ISTQB, 2012).

Nível de teste (*Test level*): Um grupo de atividades de teste que são organizadas e gerenciadas de forma conjunta. Um nível de teste está relacionado a responsabilidades em um projeto. Exemplos de níveis de teste são: testes unitários ou de componente, integração, sistema e aceitação (TMMi, 2012).

Oráculo de Teste (*Test oracle*): Instrumentos utilizados para determinar se os resultados produzidos correspondem ao desejado (ISTQB, 2012). Uma das formas é comparar esperado com obtido. Outra forma é utilizar assertivas pontuais ou verificadores estruturais. Já uma terceira forma é utilizar funções inversas.

Risco (*Risk*): Fator que pode resultar em consequências futuras negativas; normalmente expresso em termos de impacto e possibilidade (TMMi, 2012).

Risco de produto (*Product risk*): Risco diretamente relacionado ao objeto/artefato do teste (TMMi, 2012).

Testador (*Tester*): Profissional que está envolvido nas atividades de teste de um sistema ou componente (TMMi, 2012).

Teste (*Test*): Um conjunto de um ou mais casos de teste (829-2008, 2012).

Teste caixa branca (*White-box testing*): Teste baseado em uma análise da estrutura interna do componente ou do sistema em teste (ISTQB, 2012).

Teste caixa preta (*Black-box testing*): Teste funcional ou não funcional sem referenciar a estrutura interna do componente ou sistema em teste (ISTQB, 2012).

Teste de aceitação (*Acceptance testing*): Teste formal relacionado às necessidades dos usuários, requisitos e processos de negócios. É realizado para estabelecer se um sistema satisfaz ou não os critérios de aceitação, e para possibilitar aos usuários, aos clientes e às outras entidades autorizadas decidir aceitar ou não determinado sistema (ISTQB, 2012).

Teste de carga (*Load testing*): Tipo de teste de desempenho realizado para avaliar o comportamento de um componente ou sistema com carga crescente, por exemplo, número de usuários paralelo e/ou o número de transações, para determinar qual a carga pode ser manipulada por um componente ou sistema (ISTQB, 2012).

Teste de desempenho (*Performane testing*): Processo que determina o desempenho de um produto de software (ISTQB, 2012).

Teste de integração (*Integration testing*): Teste realizado com a finalidade de expor defeitos nas interfaces e nas interações entre componentes ou sistemas integrados (ISTQB, 2012).

Teste de regressão (*Regression testing*): Teste realizado em um programa previamente testado após alguma modificação feita e com a finalidade de assegurar que defeitos não tenham sido introduzidos nas áreas não alteradas do

software como resultado da referida modificação. Este teste é realizado quando o software ou seu ambiente é alterado (ISTQB, 2012).

Teste de segurança (*Safety testing*): Teste que determina a segurança de um produto de software (ISTQB, 2012).

Teste de sistema (*System testing*): Testa um sistema integrado para verificar se ele atende aos requisitos especificados (ISTQB, 2012).

Teste de usabilidade (*Usability testing*): Teste que tem por objetivo verificar a facilidade que o software ou site possui de ser claramente compreendido e manipulado pelo usuário (ISTQB, 2012).

Teste funcional (*Functional testing*): Teste baseado em uma análise da especificação de funcionalidade de um componente ou sistema (ISTQB, 2012).

Teste não funcional (*Non-functional testing*): Teste dos atributos de um componente ou sistema que não se relacionam com a funcionalidade (por exemplo, confiabilidade, eficiência, usabilidade, manutenibilidade e portabilidade) (ISTQB, 2012).

Tipo de teste (*Test type*): Um grupo de atividades de teste que visa testar um componente ou sistema focado em um específico objetivo de teste, isto é, teste funcional, usabilidade, regressão, etc. Um tipo de teste pode estar presente em um ou mais níveis de teste (ISTQB, 2012).

Suite de teste (*Test suite*): Executa um conjunto de casos de teste em uma ordem pré-definida, onde a pós-condição de um teste é frequentemente usada como pré-condição do teste seguinte (ISTQB, 2012).

2.2. UML Testing Profile (UTP)

Sistemas de software estão cada vez mais complexos, havendo assim a necessidade do uso de novos paradigmas para construí-los. Um desses paradigmas é o desenvolvimento dirigido a modelos (*Model Driven Development*), que apresenta impacto na redução de tempo para melhorar a

qualidade dos produtos desenvolvidos. Esse paradigma em particular preocupa-se com introdução de modelos em todo processo de desenvolvimento, permitindo abstrações e automações. Para esse fim, uma linguagem gráfica e padronizada chamada de Unified Modeling Language (UML) (Booch et al., 2005; UML, 2012) foi desenvolvida para ajudar na construção dos sistemas de software. A UML permite que requisitos do sistema, assim como especificações de projeto, sejam criados e visualizados a partir de uma forma gráfica. Consequentemente, ela ajuda na comunicação entre membros de uma mesma equipe melhorando a validação dos conceitos modelados. A partir desses benefícios a UML tornou-se amplamente adotada como tecnologia de desenvolvimento em toda indústria de software.

No entanto, o desenvolvimento de sistemas de alta qualidade requer além de processos de desenvolvimento bem definidos, processos de testes de software. Relacionado a isso há o processo de testes baseados em modelos (*Model-Based Test - MBT*) no contexto da UML. Como a UML fornece meios limitados para o design e desenvolvimento de testes, um consórcio composto por seis instituições foi formado pela OMG (*Object Management Group*) (OMG, 2012) para desenvolver um perfil (*profile*) para a UML 2.x capaz de permitir a modelagem de conceitos de teste, e assim aplicar o paradigma de testes baseados em modelos. As instituições participantes que possuíam ampla experiência na área de teste de software foram às seguintes: Ericsson, Fraunhofer/FOKUS, IBM/Rational, Motorola, Telelogic and University of Lübeck. Esse profile foi denominado de UML Testing Profile (UTP) (UTP, 2012; Baker et al., 2007).

Perfil para UML pode ser considerado uma especialização da linguagem de modelagem. Dessa maneira a UTP estende a UML a partir da adição de novos estereótipos, atributos e restrições para permitir a modelagem de conceitos relacionados à área de testes. Como a UTP é um padrão da OMG, que utiliza UML e por ser uma abordagem bem aceita pela comunidade de modelos que contempla diversos conceitos importantes da área de teste, decidimos utilizá-la na tese.

Alguns dos conceitos propostos na UTP são os seguintes: veredictos, test context, casos de teste (*test case*), árbitro, SUT (System Under Test), Data Pool, Data Partition, Timer, Timezone, entre outros. Esses conceitos são agrupados em: arquitetura de teste, dados de teste, comportamento dos testes e tempo. Detalhes são apresentados em (UTP, 2012; Baker et al., 2007).

Assim como mencionado na seção 2.1, test context refere-se a uma entidade que possui um ou mais casos de teste implementados. Já um caso de teste representa algum teste especifico criado para validar alguma funcionalidade ou artefato(s) pertencente(s) ao sistema analisado. Enquanto que o SUT (*System Under Test*) é o sistema em teste. Esses conceitos são considerados pela UTP e são mencionados em diversos pontos do documento.

2.3. Sistemas Multiagentes

Sistemas de softwares complexos geralmente são compostos por um grande número de partes, que realizam diversas interações. Muito desses subsistemas podem ter sido acoplados por outros sistemas ou se comunicam com softwares independentes. Conforme (Fayad et al., 1999), mesmo com tais dificuldades, esse tipo de complexidade apresenta algumas regularidades. A primeira é a capacidade da complexidade assumir a forma de uma hierarquia. Outra regularidade é que sistemas hierárquicos evoluem mais rapidamente do que sistemas não hierárquicos de tamanho semelhante. Por fim, é possível diferenciar as interações que ocorrem entre subsistemas e interações que ocorrem dentro dos subsistemas.

Com essas informações, os engenheiros de software desenvolveram algumas técnicas para lidar com tal complexidade mais facilmente:

- Decomposição: A mais básica técnica para enfrentar grandes problemas é dividi-los em pedaços menores e mais gerenciáveis.
- Abstração: É o processo de definição de um modelo simplificado de um sistema, que enfatiza propriedades importantes para a resolução e representação do problema.
- Organização: É o processo de definir e gerenciar os relacionamentos entre várias entidades que têm a capacidade de resolver problemas.

Com base nessas informações foram desenvolvidas diversas abordagens para lidar com a complexidade dos sistemas modernos. Alguns exemplos seriam: a programação orientada a objetos, a programação orientada a componentes, reutilização de software, e frameworks. Uma dessas abordagens é a orientação a agentes.

De acordo com (Wooldridge e Jennings, 1998), um agente tem as seguintes propriedades:

- Autonomia: Um agente deve ser capaz de operar sem intervenção externa (de um usuário), e ter algum tipo de controle sobre suas ações e seu estado interno.
- Habilidade Social: Um agente deve ser capaz de interagir com outros agentes, humanos ou não, em benefício de seus objetivos.
- Pró-Atividade: Um agente inteligente não deve apenas responder aos estímulos de outros agentes e do seu ambiente, devendo também tomar iniciativa para que seus objetivos sejam atingidos.
- Reatividade: Um agente deve ser capaz de observar o seu ambiente e responder às mudanças que ocorram nele.

Para alguns pesquisadores da área de Inteligência Artificial (IA), o termo agente tem um significado mais forte e mais específico. Segundo os pesquisadores dessa área, um agente é como um sistema de computação que, além de possuir as características mencionadas, são conceituados ou implementados usando conceitos que normalmente são aplicados a humanos.

Outras características que podem estar presentes em agentes de softwares são as seguintes:

- Adaptabilidade: São capazes de modificar em algum grau seu comportamento devido às mudanças do ambiente e de outros agentes.
- Benevolência: Capacidade que um agente possui de cooperar com outros agentes;
- Mobilidade: Habilidade que um agente possui de migrar de uma plataforma (ambiente) para outra através de uma rede de computadores;
- Racionalidade: São capazes de selecionar suas ações baseadas em seus objetivos;
- Veracidade: Um agente não vai propositalmente comunicar informações falsas, porém pode fazê-lo acidentalmente caso haja, por exemplo, algum erro de execução. No entanto, seu objetivo é permitir o envio de informações verdadeiras.

Assim, de acordo com (Jennings e Wooldridge, 2000), a abordagem conhecida como Sistema Multiagente (SMA) usa agentes para trabalharem de forma colaborativa para atingir objetivos em comuns ou diferentes, podendo ser

utilizada no desenvolvimento de sistemas distribuídos complexos. Essa abordagem procura reduzir o custo e a complexidade associados ao desenvolvimento de sistemas distribuídos Além disso, atualmente ela tem sido aplicada em diferentes domínios de aplicação, como, por exemplo, mercado financeiro (Bispo, Costa et al., 2009), alinhamento de esquemas de banco de dados (Figueiredo, Costa et al., 2010), reputação (Costa et al., 2008b), etc.

2.4. Sistemas Autoadaptativos que Realizam Autoteste

A complexidade dos sistemas correntes tem influenciado a comunidade de engenharia de software a procurar por sistemas capazes de ajustar ou adaptar seus comportamentos em resposta a mudanças no ambiente. Nesse contexto, computação autonômica e consequentemente sistemas autoadaptativos tornaram-se uma das direções mais promissoras.

Computação autonômica foi um termo proposto pela IBM (IBM, 2003) para descrever sistemas computacionais considerados autogerenciáveis. Tais sistemas possuem quatro propriedades: self-configuration, self-optimization, self-healing and self-protecting. Detalhes de cada self-* são apresentados em (IBM, 2003).

A Figura 1 ilustra um processo padrão de autoadaptação (também chamado de control-loop) proposto pela IBM (IBM, 2003). Tal processo é similar a um modelo de agente genérico proposto por (Russel e Norvig, 2003), em que um agente inteligente acompanha, a partir de sensores, o que acontece em seu ambiente, usando-os para definir quais ações executar.

A partir dos sensores os dados são efetivamente coletados por um monitor para que possam ser analisados. Com essa análise pode-se realizar um planejamento sobre qual ação executar para atingir algum objetivo. Após decidir qual ação será usada, ela é configurada e executada de forma apropriada no sistema.

Apesar de diversas abordagens (Dobson et al., 2006; Garlan et al., 2004; Kaiser et al., 2003) descreverem como sistemas podem realizar autoadaptações, elas não implementam agentes de software. Assim, propriedades relacionadas aos agentes e consideradas importantes para sistemas autoadaptativos não são contempladas, como, por exemplo, autonomia, aprendizado, raciocínio e pró-ativo (Wooldridge e Jennings, 1998; Jennings e Wooldridge, 2000).

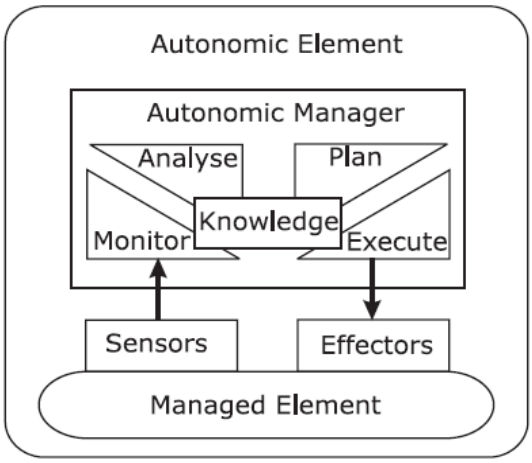

Figura 1. Processo de autoadaptação proposto pela IBM

Outro ponto comentado na literatura é o conceito de autoteste (*self-test*) mencionado por diversos trabalhos, como, (Denaro et al., 2007; Stevens et al., 2007; Wen et al., 2005). Tal conceito está relacionado à execução de um conjunto de testes, pelo próprio agente autoadaptativo, responsáveis por garantir que uma nova ação escolhida a partir da autoadaptação consiga atingir um objetivo desejado, assim como não gerar problemas ao restante do sistema. No entanto, essa é uma linha de pesquisa que continua procurando soluções que possam ser reusadas em diferentes domínios de aplicação, já que as abordagens propostas geralmente são desenvolvidas de forma *ad hoc*.

2.5. Framework JADE

De acordo com (Bellifemine et al. 2012a; Bellifemine et al. 2012b), JADE foi escrito em Java devido à programação orientada a objetos em ambientes distribuídos heterogêneos. Tal framework foi escolhido, pois facilita o desenvolvimento de SMAs, já que representam diversos conceitos da área, e por ser FIPA-Complaint (FIPA, 2012). Além disso, o framework oferece um conjunto de ferramentas gráficas, onde são realizadas operações administrativas e monitoramentos dos estados dos agentes.

Na Figura 2 são ilustrados containers (ambientes de execução dos agentes) executando em uma mesma máquina ou em máquinas distintas podendo fazer parte de uma mesma plataforma. A partir disso, agentes podem migrar de um container para outro, a fim de atender uma das características chave dos agentes (FIPA, 2012).

Em cada plataforma deve ser definido um container principal (Main Container), responsável por registrar novos containers. O Main Container hospeda o agente AMS (*Agent Management System*) e o agente DF (*Directory Facilitator*). O primeiro agente é responsável por realizar a gerência completa dos agentes que fazem parte da plataforma referente, ou seja, realiza seus cadastros, suas remoções, oferece serviço de páginas brancas (busca de agentes por nome), etc. Já o segundo agente fornece um serviço de páginas amarelas onde agentes podem procurar por outros agentes que prestam serviços necessários para que seus objetivos sejam alcançados.

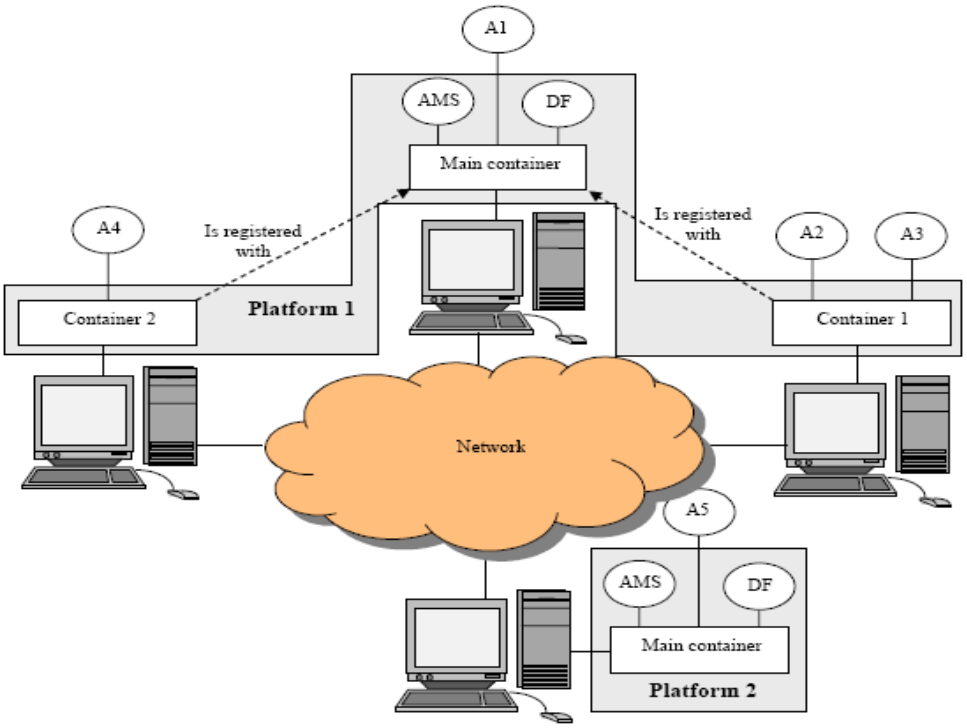

Figura 2: Plataformas de agentes do JADE

Cada agente JADE executa um ou mais comportamentos (*Behaviour*). JADE oferece dois tipos de comportamentos (ver Figura 3): simples e compostos. Os simples são comportamentos que não possuem outros comportamentos, como, por exemplo, OneShotBehaviour e CyclicBehaviour. Por outro lado, os compostos são aqueles que possuem comportamentos inclusos, como, FSMBehaviour, SequentialBehaviour e ParallelBehaviour. O comportamento FSMBehaviour, mencionado em diferentes pontos do documento, permite a construção de um fluxo de execução baseado em uma

máquina de estado finito. Esse tipo de comportamento é utilizado na tese para definir diferentes processos de autoadaptação em agentes de software, sendo apresentada em detalhe no capítulo 4.

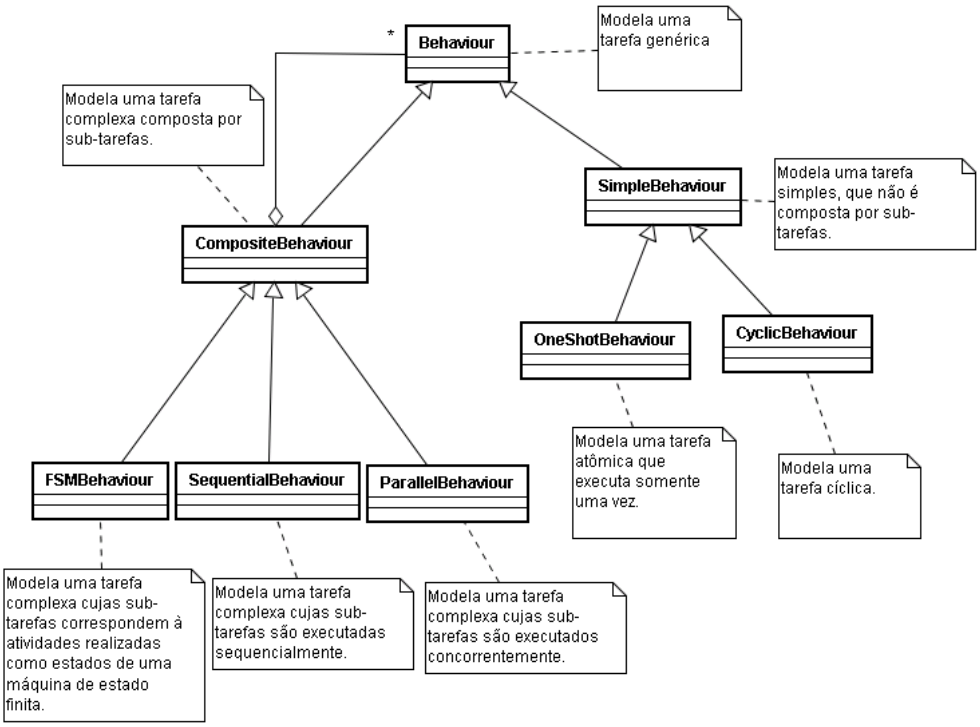

Figura 3. Diagrama de classes do JADE