**INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA**

Departamento de Engenharia de

Eletrónica e Telecomunicações e de Computadores

# Inteligência Artificial na Verificação e Teste de Software para Desenvolvimento Ágil

FREDERICO ALEXANDRE FERREIRA

(Bacharel)

Dissertação de natureza científica realizada para a obtenção de grau de Mestre em Engenharia Informática e de Computadores

Orientador: Professor Doutor Luís Filipe Graça Morgado

Júri:

Presidente: Professor Doutor Nuno Miguel Soares Datia

Arguente: Professor Doutor Porfírio Pena Filipe

Vogal: Professor Doutor Luís Filipe Graça Morgado

**Dezembro de 2016**

“Everything should be made as simple as possible, but not simpler”

Albert Einstein

# Resumo

As metodologias ágeis convivem bem com a mudanças que ocorrem ao longo de um projeto de desenvolvimento de software, sendo por isso cada vez mais adotada para a sua gestão. As metodologias ágeis e em particular o “scrum”, têm por base um conjunto de princípios que incluem a entrega incremental de funcionalidade e cujo somatório corresponde no final à totalidade do produto pretendido.

Os testes, tal como nas outras metodologias, são fundamentais para garantir a qualidade do produto, mas aqui correm obrigatoriamente em cada iteração, testando funcionalidades novas e antigas. Em cada iteração são criados novos planos de testes compostos pelos scripts das para as novas funcionalidades e pela reutilização dos scripts da funcionalidade anterior. Os testes às funcionalidades anteriores são conhecidos por testes de regressão.

A presente dissertação apresenta uma nova abordagem para a automatização dos testes de regressão usados no desenvolvimento ágil, que é fácil de enquadrar com a metodologia e a prática das equipas e que inclui:

- Uma forma de modelação do sistema em testes tendo por base as histórias de uso;
- Um algoritmo para a geração automática de planos de teste:
- Uma arquitetura de um agente artificial e um dialeto para execução de testes no sistema em testes;
- Uma sequência de atividades para verificação da consistênci dos requisitos.

**Palavras-chave:** agentes inteligentes, testes funcionais, teste de regressão, planeamento automático, strips, desenvolvimento ágil, scrum, teste baseado em modelo

# Abstract

Agile methodologies can cope better with changes occurring during the software development and are, for that reason, more and more used for project management. Agile methodology, particularly “scrum”, are based on a common set of principles including incremental delivery of functionality, that in the end represent the full product.

Testing, like in other methodologies, are fundamental to guaranty product quality, but here that run on each iteration, testing new and current functionality. In each iteration new testing plans are created composed with test scripts for new functionality and reusing scripts for existing functionality. Tests of the old functionality are known as regression testing.

The present dissertation suggests a new approach to automate regression tests used in agile development, easy to integrate with the methodology and team’s practice and includes:

- A modeling form of the system in tests based on user stories;
- An algorithm for automatic generation of testing plans;
- An architecture of artificial agent and a dialect for tests execution on the system in tests;
- A sequence of activities to verify the consistency of requirements.

**Keywords:** artificial intelligence agent, functional tests, regression tests, planning, strips, agile development, scrum, model-based testing

# Agradecimentos

Ao Prof. Dr. Luís Morgado, por ter aceite a tarefa adicional da orientação deste trabalho, pela sua compreensão, tolerância e apoio. E ainda por, como professor ter criado a motivação e a emoção nos temas da engenharia de software e da inteligência artificial.

Ao Prof. Luís Osório, pelo trabalho continuado de ligação entre a Universidade e a indústria, o que motivou a continuar os meus estudos nesta instituição.

Aos docentes do Instituto Superior de Engenharia de Lisboa pelo profissionalismo, dedicação, apoio e amizade como se relacionam com os alunos e em particular aqueles que dedicaram as suas vidas e a capacidade produtiva a esta instituição de ensino e onde gostaria de realçar o Prof. Dr. Hélder Pita, o Prof. Dr. Walter Vieira, o Prof. Dr. Paulo Trigo e o Prof. Dr. Manuel Barata.

À minha família: Gabi, Marta e Duarte, pelo apoio constante e tolerante nesta minha recente atividade de estudo, que algumas dificuldades e limitações trouxe à nossa vida comum, mas que em nenhum momento me cobraram, muito obrigado.

Aos meus pais, já falecidos.

# Índice

# Tabela de figuras

# 1 Introdução

Os sistemas de informação de uma organização eram até alguns anos maioritariamente usados para a automatização de processos e práticas manuais, os seus ganhos resultavam muitas vezes (apenas) da redução de custos pela substituição de trabalhadores e por economias de consumo de materiais e meios. Os planos e orçamentos das empresas eram feitos a cinco ou mais anos, as produções definidas para economias de escala (e menos para necessidades de grupos de consumidores), as novas soluções e aplicações planeadas a um ou mais anos. O departamento de Sistemas de Informação era visto como um custo de exploração (tal como a eletricidade ou as comunicações) e por vezes enquadrado (a reportar) à direção financeira. A implementação das novas aplicações, era da responsabilidade do IT, a metodologia de desenvolvimento o em cascata ("*waterfall"),* os novos aplicativos desenhados e implementados com versões testadas e suportadas dos sistemas operativos, dos compiladores e de outro software próprio e de terceiros. Os ambientes de execução eram internos, com equipamentos padronizados, com redes previsíveis e controladas, com tempos reservados para processamentos de "*batch",* com tempos de proteção para realizar salvaguardas ("backups") e as evoluções dos sistemas perfeitamente calendarizadas.

Com a metodologia em cascata o sistema é totalmente definido e "fechado" nas etapas iniciais de visão, análise e conceção, o desenvolvimento é uma tradução (o mais linear possível) das definições anteriores, as alterações e mudanças são evitadas e os testes globais planeados em paralelo com a análise e são implementados numa única etapa no final do desenvolvimento, para suportar o processo de aceitação da aplicação desenvolvida, antes da sua instalação em produção. No cerne deste método de desenvolvimento está a convicção de que é possível definir completamente o sistema na fase de análise, de forma a evitar todas as alterações e mudanças durante o processo de desenvolvimento, assumindo que o mundo exterior, empresa, mercado, legislação também se manterão de alguma forma imutáveis ou que não terão impacto na aplicação em desenvolvimento.

Os testes de integração e funcionais podem por isso ser preparados ao longo do processo e executados de uma só vez no final, para demonstrar o bom funcionamento da

aplicação. No apoio ao teste são usadas ferramentas para testes de carga, sequenciadores ou simuladores, mas a automatização da sua repetição não é um problema, pois tendem a ocorrer uma vez.

No contexto económico atual, as organizações usam os sistemas de informação para melhorar a eficácia e a eficiência dos seus processos produtivos, para se promoverem em mercados dispersos e globais, para incorporar e aumentar o valor dos bens e serviços que disponibilizam. As organizações tendem a competir no mercado global, caracterizado pelo excesso de oferta e onde a diferenciação das empresas tem por base a sua inovação. Alguma inovação passa por apresentar novos processos de comercialização, que assentam em formas de relação com os clientes, por exemplo usando as redes sociais, ou de interligações que envolvem diferentes organizações da cadeia de produção ou ainda pelo uso de aplicativos de software para acrescentar valor a produtos de diferentes naturezas e origens.

Os consumidores usam cada vez mais dispositivos móveis com configurações diversas, com capacidades sempre crescentes, a operar sobre redes de largura de banda e latências incontroláveis, em cenários de múltiplas aplicações, com reduzida tolerância para experiências com aplicativos de baixo desempenho, elevados tempos de resposta ou mesmo interfaces desatualizadas ou de uso complexo. Num mundo de excesso de oferta, uma fraca experiência de uso, pequenas dificuldades ou limitações levam facilmente o consumidor a trocar de aplicativo e consequentemente de produto ou de fornecedor.

Os sistemas de informação neste novo enquadramento competitivo deixam de ser apenas meios de apoio à atividade interna da empresa, para passarem a ser o centro do seu desenvolvimento, num processo descrito por algumas consultoras como "transformação digital", que coloca os sistemas de informação no centro da estratégia das organizações.

A organização precisa de garantir a sua presença nos novos canais disponibilizados pela internet, pelas redes sociais e pelos dispositivos moveis e definem novas formas de contacto com o público-alvo, com novos paradigmas de serviço, caracterizado pela ubiquidade e pela disponibilidade permanente. As novas aplicações assentam em novas integrações de sistemas internos e externos, alteram processos de negócio, assentam no

s*elf-service* e na rapidez, retirando a mediação humana e substituindo algumas funções por algoritmos e processos "inteligentes".

A inovação é fundamental para a exploração dos canais novos e atuais e a inovação não é um ato único, mas antes um contínuo, que se alimenta do ciclo de experiência e refinamento. As novas aplicações que têm pressa em chegar aos mercados e aos seus utilizadores, são disponibilizadas de forma incremental, por razões de tempo, mas também para avaliar formas de uso e o interesse de algumas das suas opções. A mudança é constante, pelo grau experimental de alguns requisitos, mas também para acompanhar a evolução do mercado e da tecnologia, o processo de desenvolvimento já não pode evitar a mudança, tem de viver com a mudança, de se adaptar continuamente à mudança. A metodologia de gestão de projeto passa a ser ágil, os ciclos de desenvolvimento curtos terminam com recolha de feedback e entradas em produção, a arquitetura do sistema emerge com a construção do próprio sistema e algum do código produzido precisa de ser generalizado, refinado, melhorado e refeito ("re-factoring") ao longo do projeto. O processo de refinamento de uma nova aplicação prolonga-se ao longo de muito tempo com base nos ciclos que incluem a análise, desenvolvimento, testes, demonstração, recolha de feedback e entrada em produção. Os requisitos adicionados em cada ciclo somam-se aos anteriores e precisam de formar um novo todo coerente, podendo para a introdução de um novo requisito ser necessário rever, complementar, reescrever ou mesmo eliminar alguma da funcionalidade existente anteriormente. A avaliação de cada nova funcionalidade tem de ser feita per si, mas também na consistência com o processo total de que faz parte e do impacto que tem.

Sendo o processo de evolução gradual, com requisitos em constante evolução, é requerido um grande e continuado apoio dos utilizadores chave do negócio ao longo de todo o desenvolvimento. Este apoio compete diretamente com muitas atividades diárias desses utilizadores e essa disponibilidade é um dos principais limitadores da velocidade de desenvolvimento, tendo a equipa técnica de ter uma dimensão ajustada a essa capacidade disponível de análise de novos requisitos e de recolha de feedback. As equipas técnicas tendem por isso, a ser mais pequenas e o trabalho de desenvolvimento mais distribuído ao longo do tempo, embora já com a aplicação em uso. O número de programadores, técnicos consultores é, aos dias de hoje, muito inferior às necessidades de profissionais competentes, o que se traduz numa elevada rotação de postos de trabalho

com substituições dos membros da equipa e a perca do detalhe das funcionalidades anteriormente implementadas e em particular à avaliação do potencial impacto, que possam ter nas novas funcionalidades. Será por isso necessário dispor de alguma forma de ajuda às equipas de análise para permitir validar a coerência do sistema, ainda ao nível dos requisitos e da análise de impactos.

Em termos tecnológicos as novas aplicações tendem a envolver muitos sistemas, tecnologias heterogéneas e correm maioritariamente fora das áreas controladas pelos SI em dispositivos, ferramentas, configurações, redes e horários diversos. A complexidade do desenvolvimento aumentou exponencialmente, as aplicações disponibilizadas têm por vezes funcionalidade com responsabilidade "contratual", onde os erros são por isso caros e inadmissíveis para os patrocinadores internos e para os consumidores internos e externos.

No processo de desenvolvimento de software, os testes servem para verificar o bom funcionamento do sistema e a conformidade com os requisitos que lhe deram origem. A quantidade de testes a realizar está diretamente relacionada com a dimensão e criticidade do sistema, mas também com outros fatores que incluem os ambientes, configurações e dispositivos em que o sistema possa ser usado, processo de desenvolvimento ou condicionantes processuais ou legais relativos à produção, validação ou uso do sistema. De forma geral todos estes elementos contribuem para o aumento do esforço associado aos testes dentro do processo de desenvolvimento de software.

Os testes usados ao longo do processo de desenvolvimento são de diferentes tipos, realizados por diferentes pessoas e com objetivos diferentes, desde o validar de um pedaço de código à demonstração de toda a funcionalidade de um processo de negócio, ou à capacidade de suportar condições mais ou menos adversas, por exemplo aumentos de carga ou ataques à segurança. As atividades de teste estão enquadradas no decorrer do projeto em função do tipo de projeto e das metodologias técnicas e de gestão usadas no projeto.

As metodologias ágeis de desenvolvimento de software e em particular o Scrum têm vindo a ganhar importância crescente no desenvolvimento de aplicações empresariais, uma vez que lidam melhor com a mudança e a indefinição de requisitos. Na base desse processo está o desenvolvimento iterativo e incremental da solução, no modelo de

prototipagem evolutiva que termina no produto desejado e na sua passagem a produção. Cada iteração, denominada de sprint, tem habitualmente uma duração de 2 a 4 semanas, que termina com a apresentação do trabalho produzido ao cliente (ou destinatários da aplicação) para validação e recolha de feedback. Ao contrário de um protótipo, o código tem a qualidade de uma versão de produção e poderá por isso ir a produção.

Os requisitos do sistema são descritos em histórias de uso, que descrevem a funcionalidade pretendida e que são o ponto de partida para a análise mais detalhada do sistema. No início de cada sprint são selecionadas as histórias a implementar, com base no seu valor para o negócio e do seu grau de risco. A análise detalhada, o desenho, implementação e testes da funcionalidade são feitos durante o sprint, no final do qual é demonstrada. No final do sprint a demonstração da funcionalidade implementada é muito importante, ensinar o uso do sistema e para recolher feedback e tem normalmente por base scripts e casos de uso da aplicação, que podem ser reusados e que podem também ser usados como testes de regressão nos sprints seguintes. Um processo comum de negócio, poderá envolver funcionalidade desenvolvida em diferentes sprints, representando uma jornada de uso do sistema, por exemplo a compra de livros por um cliente num site junta diversas histórias de uso.

Os testes de integração, funcionais ou de sistema continuam a ser um elemento fundamental do processo de garantia de qualidade de cada uma das entregas (“*releases”*) das aplicações. Para garantir a qualidade é necessário, em cada entrega, testar o novo código, validar as funcionalidades acrescentadas e verificar que o código e as funcionalidades anteriores não foram destruídas pelos novos desenvolvimentos, testando novamente esse código. Numa entrega os testes ao código já existente, são denominados de testes de regressão e são realizados por repetição do processo usado anteriormente, são por isso normalmente monótonos, repetitivos, de trabalho intensivo, sendo por isso candidatos ideais à automatização com agentes inteligentes.

A inteligência artificial tem vindo a ganhar relevância na realização de atividades que requerem conhecimento ou capacidades de decisão e que até há pouco tempo eram apenas desempenhadas por humanos, mas que por serem repetitivas, monótonas e de trabalho intensivo são melhor desempenhadas por autómatos, libertando os humanos para outras atividades de maior criatividade e valor.

## 1.1 Motivação

O desenvolvimento de aplicações empresariais para a web e para dispositivos móveis, nos tempos atuais enfrenta um conjunto de dificuldades, cujas principais são:

- Maior rotação das equipas e logo perca de conhecimento informal;
- Menor controlo sobre os ambientes de uso das aplicações desenvolvidas;
- Maior número de sistemas envolvidos, internos e externos na construção de uma aplicação;
- Ausência de tempos de "offline" ou de "down time" das aplicações;
- Aumento da criticidade e responsabilidade das aplicações disponibilizadas a utilizadores finais;
- Sistemas, aplicações e serviços, em constante evolução, com ciclos curtos de entrega e frequentes alterações (refazer) do código.

Todos estes fatores contribuem para a importância dos testes e para o aumento de volume de trabalho e consequente custo que lhe está associado.

Evitar o aumento dos custos dos testes, nomeadamente a pela sua automatização é muito importante para o processo de desenvolvimento de aplicações neste novo quadro da "transformação digital".

A motivação para o presente estudo vem da necessidade de intervir no aumento de custos do desenvolvimento de software, pelo inerente aumento dos testes, com a possibilidade de substituir algumas atividades manuais por atividade de agentes inteligentes a colaborar na atividade de testes.

O "Scrum" é uma das metodologias com maior uso dentro das metodologias ágeis de desenvolvimento e tem por base uma lista de funcionalidades a implementar no projeto,

denominada de “backlog”, funcionalidades essas que serão entregues de forma incremental, em ciclos de desenvolvimento de 2 a 4 semanas e que se denominam “sprints”. Cada ciclo de desenvolvimento, tem início com a seleção de um conjunto de requisitos a implementar e termina com a apresentação ao cliente (a quem se destina o projeto) das funcionalidades implementadas, para validação e recolha de feedback. Com base no feedback, podem ser incluídos, retirados ou trocados requisitos da lista que depois é ordenada para ser usada no próximo ciclo de desenvolvimento.

Um requisito é descrito com base numa história de uso (“user story”), que pode descrever uma nova funcionalidade, uma alteração ou um bug. A história de uso liga um ator a uma operação ou procedimento com um objetivo concreto de uma realização ou atividade no sistema. Uma história de uso poderá ser partida em tarefas, mas deve estar contida num ciclo de desenvolvimento. Assim cada ciclo acrescenta um conjunto de novas histórias de uso às funcionalidades que já existiam e que são demonstradas no seu final. Uma característica importante do ciclo de desenvolvimento desta metodologia, é que o código produzido deverá ter qualidade de produção e consequentemente terem sido testadas as novas funcionalidades e as anteriores ao ciclo. O teste da funcionalidade anterior é chamado de regressão e é fundamental para verificar que não foram partidas funcionalidades anteriores.

As histórias de uso podem ser agrupadas de diversas formas, por exemplo numa sequência lógica de uso com outras histórias formando uma jornada de utilização (“journey”). A jornada representa casos de uso do sistema pelos utilizadores, cuja forma pode ir mudando com a introdução de novas funcionalidades. Uma jornada pode mudar em consequência da alteração ou remoção de uma história de uso ou até pela introdução de novas.

## 1.2 Objetivos

O objetivo do presente estudo é uma abordagem para a automatização testes de regressão de software para aplicações empresariais desenvolvidas no contexto web com metodologias ágeis e cujas principais caraterísticas são:

- Permitir guardar, acrescentar, gerir, usar e reusar testes às funcionalidades do sistema;
- Documentar as execuções dos testes;
- Permitir sequenciar testes individuais para o teste completo de processos de negócio;
- Gerar automaticamente planos com sequências de histórias de uso;
- Interagir com o sistema em teste com base nas interfaces que expõe, écrans e serviços, de forma independente da linguagem com que este foi implementado;
- Ser de fácil de expandir para incluir novas interfaces e acessos aos sistemas em teste;
- Ser de fácil integração com os processos ágeis de desenvolvimento, com reduzido incremento do trabalho de análise atual e integrável com as ferramentas em uso;
- Ao nível da análise uso de uma linguagem muito livre para a descrição de estados do modelo;
- Ao nível da interação com o sistema em testes, uso de um conjunto de comandos simples para a interação com browser e para o consumo de serviços;
- Uso de código aberto e expansível;
- Escalabilidade da solução.

A estas caraterísticas seria interessante acrescentar uma possibilidade de ligar a ferramenta aos repositórios existentes de gestão de requisitos por forma a melhor integrar no ambiente de desenvolvimento, mas que não é aprofundada no presente trabalho.

## 1.3 A abordagem ao problema

A abordagem proposta passa pelo enriquecimento da descrição das histórias de uso e a construção de um agente inteligente para a execução dos testes no sistema em testes. O enriquecimento das histórias de uso inclui uma caracterização de estado, com base em pré e pós condições de execução e a inclusão do teste funcional da história com base nas interfaces do sistema em testes. A caracterização de estado das histórias de uso permite verificar a sua consistência e gerar planos automáticos de testes de jornada de uso (utilização de um utilizador na concretização de um processo de negócio) e o agente a execução coordenadas das histórias com base no plano de jornada. Esta aproximação de modelização permite uma verificação a nível abstrato pela ligação entre histórias numa jornada e a sua concretização em testes executáveis no sistema em testes com recolha dos resultados. O processo é indicado para a realização dos testes de regressão e em software de desenvolvimento ágil, pela importância que este tipo de testes tem nestas metodologias de desenvolvimento.

## 1.4 Organização do documento

O documento tem está organizado em sete capítulos com a seguinte estrutura:

**Capítulo 1: Introdução**

Capítulo onde é feita a apresentação geral do trabalho, da motivação que lhe deu origem, os objetivos pretendidos, a forma geral da solução e onde é apresentada a organização do documento.

**Capítulo 2: Enquadramento teórico**

Capítulo destinado ao enquadramento teórico onde são apresentadas as principais áreas envolvidas na construção da solução proposta, primeiro, os modelos gerais e depois o dos testes baseados em modelo, onde a abordagem defendida tem melhor enquadramento.

**Capítulo 3: Proposta de solução**

Neste capítulo é descrita a solução proposta para o modelo de arquitetura de uma ferramenta de testes de regressão baseada no uso de agente inteligente e o processo de modelização do sistema em testes para permitir a geração automática e execução de testes.

**Capítulo 4: Arquitetura da Solução**

Este capítulo descreve a arquitetura geral proposta para a realização da solução, descrevendo a estrutura interna, o comportamento dinâmico e outras particularidades do sistema proposto.

**Capítulo 5: Implementação**

Este capítulo descreve as opções tomadas na implementação do demonstrador em termos de linguagens e de componentes. Apresenta ainda as instruções para a sua instalação.

**Capítulo 6: Conclusões**

Este capítulo encerra o documento com uma avaliação da solução apresentada e indicação de alguns caminhos para potenciais trabalhos futuros para evolução da solução apresentada.

# 2 Enquadramento teórico

A solução apresentada envolve várias áreas de conhecimento nomeadamente a engenharia de software, a inteligência artificial e testes. A engenharia de software na zona das metodologias de desenvolvimento, da gestão de requisitos e na arquitetura e desenho da solução considerando, que necessita de um desenho modular facilmente expansível para integrar novas bibliotecas e em particular módulos de testes específicos de cada sistema em testes (SUT – "System Under Test"). A inteligência artificial é envolvida em três áreas, agentes inteligentes, planeadores e, potencialmente, mineração de dados nos dados registados como resultado dos testes. E finalmente a área de testes propriamente dita, para montagens dos processos de automatização.

Em Meziane, Farid and Vadera, Sunil (2010), descrevem diversas utilizações de técnicas de inteligência artificial nas práticas de engenharia de software. O capítulo14 descreve alguns dos desenvolvimentos em curso na área e perspetivas futuras ligadas ao potencial de uso da inteligência artificial nas diferentes fases do ciclo de vida do desenvolvimento de software e consequentemente na engenharia de software. Relativamente à prática de testes é afirmado que apesar dos desenvolvimentos na área, é uma atividade ainda não dominada e segundo a referência contida a Bertolino (2007), são sugeridas algumas áreas para investigação futura: desenvolvimento de uma teoria universal de testes, automação total dos testes, desenho focado na simplificação dos testes e desenvolvimento de estratégias integradas para redução dos custos da repetição de testes.

Os sistemas baseados no conhecimento ("Knowledge Based Systems") foram usados por Bering and Craford (1988), para a implementação de um sistema especialista ("Expert System") programado em Prolog usado na geração de dados de teste a partir das entradas de um programa cobol em teste. Com a mesma técnica, os sistemas baseados no conhecimento, DeMaise and Muratore (1991) implementaram um sistema especialista para assistir ao teste do software do Space Shuttle passando para 56 dias um processo que antes demorava 77 e envolvia 40 pessoas.

Ambos os sistemas estavam muito adaptados às aplicações em teste e Samson (1990, 1993) propôs um ambiente genérico o REQSPERT, tomando como partida um conjunto

de requisitos classificados em funcionais e não-funcionais, para identificar métricas adequadas e produzindo planos de teste. Como nos outros dois sistemas descritos, a instanciação do sistema para responder às particularidades de cada aplicação em testes parece apresentar um grande problema ao nível dos custos do projeto.

O uso de algoritmos genéticos em otimização tem tido um uso crescente desde a sua introdução nos anos 80, mesmo quando o uso de outras técnicas de inteligência artificial foi sendo reduzido. O foco de uso desta técnica em termos de testes ocorreu para a geração de casos ótimos de testes descritos por diversos autores (Baresel, Binkley, Harman, & Korel, 2004; Baudry, Fleurey, Jezequel, & Le Traon, 2002a, 2002b; Briand, Feng, & Labiche, 2002; Briand, Labiche, & Shousha, 2005; Harman & McMinn, 2007; Liaskos, Roper, & Wood, 2007; Nguyen, Perini, & Tonella, 2008; Ribeiro, 2008; Tonella, 2004; Wappler & Wegener, 2006).

Uma outra vantagem associada à geração de planos de testes com algoritmos genéticos é a de permitir incorporar o nível de conhecimento ou preparação dos utilizadores, defendendo alguns autores, que os testes sendo feitos por engenheiros informáticos (ainda que não os que implementaram a aplicação em testes) estão polarizados por um conhecimento do domínio que os afasta do uso normal do sistema por utilizadores inexperientes e dos caminhos que os testes deveriam realmente cobrir. Defendem por isso que os algoritmos genéticos podem ser usados para simular o comportamento do utilizador inexperiente, cuja evolução de comportamento pode ser captada pela evolução dos genes, pela reprodução dos indivíduos condicionada à função de adequação ("fitness function").

Apesar do uso descrito no planeamento de testes, no apoio à engenharia de software os algoritmos genéticos são fundamentalmente usados na elaboração de planeamento do projeto e na estimativa de esforço.

Em inteligência artificial, partindo da técnica de planeamento, von Mayrhauser, Scheetz, Dahlman & Howe (2000), propuseram que em vez do uso de testes de caixa branca fosse usado o modelo do domínio para a produção dos testes. O primeiro exemplo desta aproximação foi o sistema Sleuth de 1994, construído em 3 camadas, em que a de topo gera a sequência de comandos, a do meio define os comandos individuais e a de baixo instancia os parâmetros necessários à sua execução.

(Ghallab, Nau, & Traverso, 2004), usaram um planeador, que partindo de um estado inicial e um objetivo poderia gerar um plano formado por uma sequência de operadores. Concluíram que nalguns casos o planeamento poderia ser muito lento, eventualmente devido ao algoritmo de planeamento usado. O estudo evoluiu para o uso de UML, com lógica de restrições e diagramas de transição de estados para modelizar o domínio (Scheetz, von Mayrhauser, & France, 1999; von Mayrhauser, Scheetz, & Dahlman, 1999).

Von Mayrhauser et al. (2000) mostrou que o uso de planeamento automático, tem a vantagem de permitir alterar os planos gerados para poder simular erros potenciais ocorrido no uso do sistema em testes.

Memon, Pollack & Soffa (1999) defendeu que os testes gerados por humanos para o teste de interfaces gráficas, necessitam de um grande número de sequências de interações, tornado o processo ineficiente e provavelmente incompleto. Isto ao contrário do planeamento automático que pode gerar um plano para aplicação de operadores desde o estado inicial até colocar o sistema em teste no estado objetivo (final).

## 2.1 Testes baseados em modelos

Testes baseados em modelos ("MBT: Model-based testing"), são o conceito mais se aproxima da abordagem apresentada nesta dissertação, com a separação entre os testes abstratos e concretos. Olli-Pekka Puolitaival, 2008, apresenta uma abordagem para a dinamização do uso de "MBT" no contexto industrial do desenvolvimento de software, onde descreve a importância da linguagem de modelação usada na descrição do sistema para a qualidade dos sistemas gerados. Também comenta o uso de uma notação da específica de testes TTCN-3 com o uso de outras notações orientadas à implementação e salienta a importância da prática dos engenheiros para a seleção da notação usada. Os modelos mais usados são os baseados na descrição de estado com pré e pós condições com uso de notações como "UML+OCL", "B", "Spec#" e "JML - Java Modelling Language".

No mesmo trabalho, o autor defende as vantagens da utilização e "MBT" no contexto do desenvolvimento ágil, em parte pela velocidade de desenvolvimento e pela ausência

de foco na produção de documentação de suporte, apresentando uma lista das ferramentas "MBT" existentes, na sua maioria comerciais e, aparentemente reservadas a algum secretismo e grande controlo ao ponto de algumas não terem sido disponibilizadas para um ensaio. Uma breve descrição de algumas ferramentas ensaiadas foram:

- **LEIRIOS Test Designer**, ferramenta comercial, usa diagramas de classe "UML – Unified Model Language" e maquinas de estado, para modelar o sistema. Permite a geração automática de testes, mas conta com uma ferramenta externa para a execução de testes podendo exportar os testes em diversos formatos;

- **Markov Test Logic**, também é uma ferramenta comercial, dispõe de um modelador próprio e exporta os testes gerados numa representação textual;

- **Conformiq Qtronic**, é também uma ferramenta comercial, disponibilizando os seus próprios componentes para modelização baseados em UML e dispõe de um executor para os testes, mas pode integrar ferramentas externas;

- **Reactis**, é igualmente uma ferramenta comercial destinada a software embebido, usa "stateflow" uma ferramenta gráfica de para a modelização do sistema baseada no conceito de uma máquina de estados finitos;

- **Spec Explorer**, tem uma licença para uso não comercial, é disponibilizada pela Microsoft, correndo apenas em Windows e está fortemente ligada ao Visual Studio. Usa uma modelização baseada em máquina de estados a ASML, que é uma linguagem de especificação executável.

Embora baseada nos mesmos princípios de testes baseados em modelo, a abordagem proposta, difere das soluções anteriores nos seguintes elementos:

- O elemento base para a modelação são as histórias de uso;

- A linguagem de modelação é baseada em prolog, com um baixo formalismo e de sintaxe bastante livre, para promover a sua utilização por quem escreve as histórias de uso e que não tem necessariamente de ser um programador experiente;
- O planeamento com base na caraterização de estados, como é o caso de algumas ferramentas descritas;
- A ferramenta dispõe de executor;
- A linguagem de interação com o sistema em teste é baseada em código aberto e em interfaces standard acesso via browser e webservices (logo independente do sistema em testes);
- A arquitetura do agente inteligente é aberta e expansível.

O planeamento usa como operadores histórias de uso, cujo teste individual é escrito manualmente (e não resultado de planeamento automático). O planeamento automático, com base na técnica STRIPS, define a sequência de operadores para levar o sistema do estado inicial ao estado objetivo. O plano gerado descreve ele próprio um novo operador, que pode ser usado noutras etapas de planeamento. A menor granularidade dos operadores e a possibilidade de reuso de planos, permitirá contrariar a lentidão do planeamento, que também só precisa de ser repetido quando as alterações do sistema em testes o justifiquem (aliás o objetivo é mesmo a sua reutilização em testes de regressão).

O aumento de custo resulta da adaptação ao sistema em testes, o que também ocorre no modelo proposto pelo estudo, contudo sendo o foco nas aplicações web, existe um elevado uso de mecanismos standard e comuns, como webservices e browsers, que permitem uma elevada reutilização e baixa necessidade de desenho de acessos específicos (embora esteja previsto o seu uso e expansão dos meios disponibilizados). A linguagem usada na definição dos operadores usados no planeamento, parece ser de fácil compreensão e facilmente dominável pelos consultores responsáveis pela análise sem necessidade de recorrer a outros perfis técnicos ou formação complementar, fator que também joga a favor da contenção dos custos do projeto.

No caso particular do demonstrador desenvolvido a linguagem de base era python e logo portável para vários sistemas operativos.

## 2.2 Desenvolvimento ágil de software

Um projeto é uma atividade delimitada no tempo conduzida para entregar um novo produto, serviço ou processo pré-definidos, com base num conjunto de recursos e tempo limitados. Um projeto produz sempre algo de único e novo e ele próprio é único e contêm sempre o risco de insucesso por fatores internos e externos. Para gerir a incerteza de um projeto foram desenhadas e implementadas metodologias de gestão, que no caso do desenvolvimento de software estão enquadradas no domínio da engenharia de software.

O objetivo das diferentes metodologias de gestão de projeto é assegurar o resultado da realização, usando processos e práticas bem definidas para coordenar e monitorizar as atividades do projeto. As metodologias mais tradicionais usam um processo sequencial, fechando uma atividade antes de iniciar outra e são por referidas por analogia de modelo em cascata. A sequência de atividades num projeto de software, tem inicio com a recolha de requisitos, a análise detalhada, o desenho, codificação, testes e entrada em produção. As alterações em fases posteriores obrigam à revisão de todas as atividades anteriores, o sistema só no final da construção, quando a quase totalidade das tarefas foi realizada, é que pode ser mostrado aos seus utilizadores e o promove o confronto entre cliente e desenvolvedor em torno do pagamento dos pedidos de alterações. No domínio das aplicações empresariais, em particular nas mais dinâmicas e inovadoras, a gestão tradicional de projeto tem vindo a ser substituída pelas metodologias ágeis de desenvolvimento.

As metodologias ágeis convivem melhor com a mudança, promovem a iteração mais frequente com os clientes do projeto e permitem alguma troca ou substituição de requisitos. Para o conseguir partem de uma implementação iterativa e incremental da solução que é apresentada periodicamente ao cliente para recolha de feedback e ordenação dos requisitos incluir no ciclo seguinte.

Um grupo de programadores de renome e relevância no sector, na sua atividade de desenvolver e ajudar outros desenvolver software foram descobrindo melhores formas de o fazer e com base nesse processo começaram a valorizar:

- Os indivíduos e as interações mais do que processos e ferramentas;
- O software funcional mais do que documentação abrangente;
- A colaboração com cliente mais do que a negociação contratual;
- O responder à mudança mais do que seguir um plano.

Estes princípios definiram o manifesto para o desenvolvimento ágil de software que foi criado em 2001, pelo referido grupo de programadores. Existem diversas práticas de desenvolvimento de software, que respeitam este principio e que constituem o conjunto das metodologias ágeis, como sejam o "Extreme Programming XP", o "Kanban", o "Test Driven Development", o "Feature-Driven Development", o "Agile Unified Process", o "Lean" ou o "Scrum" entre outos.

A "Figura 1 - Distribuição do uso de metodologias ágeis", mostra o uso relativos das diferentes metodologias ágeis, onde o scrum se apresenta como a mais usada.

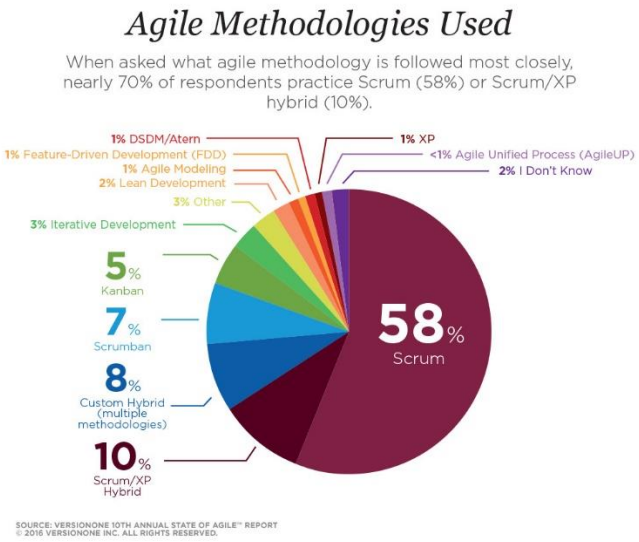


*Figura 1 - Distribuição do uso de metodologias ágeis*

## 2.3 A “framework” de gestão de projetos “Scrum”

O scrum é uma abordagem criada por Jeff Sutherland e formalizada por Ken Schwaber (Schwaber 1997, Sutherland 2004) cujo nome tem por base uma atividade que ocorre numa partida de *rubgy* quando a equipa se reúne para tomar posse da bola aquando da sua colocação em jogo. De acordo com os seus autores a metodologia é baseada num processo empírico, iterativo e incremental orientado à transparência, inspeção e adaptação do código.

O *scrum* é uma *framework* para desenvolver e manter produtos complexos, composta pela equipa de *scrum* e as suas funções, eventos, artefactos e regras. A equipa de *scrum* é composta pelo “product owner”, equipa de desenvolvimento e pelo “scrum master” e tem como principais características ter conhecimentos transversais necessários ao desenvolvimento e ser auto-organizada, sendo a equipa que decide a melhor forma de realizar o trabalho, em vez de ser dirigida por fora da equipa. A equipa responde como um todo pelo trabalho realizado, entregando o produto de forma incremental e interativa promovendo a recolha de feedback do cliente. As entregas são realizadas com software com valor e qualidade necessária para ser usado em produção. Os ciclos de entrega são denominados de *sprints*.

O *sprint* é o core do *scrum* corresponde a uma *time-box* (esforço temporal) de 2 a 4 semanas, que termina com a entrega de uma revisão de produto usável, que incorpora funcionalidade adicional em relação à entrega anterior. O ciclo de *sprint* começa logo a seguir ao ciclo anterior e é composto de um planeamento, um conjunto de reuniões diárias (denominadas *scrum*), do trabalho de desenvolvimento, da revisão e retrospetiva do ciclo. Durante o ciclo não podem ocorrer mudanças de âmbito ou requisitos que possam colocar em causa o objetivo do ciclo, a qualidade do desenvolvimento não pode decrescer e poderá haver um acerto e renegociação do âmbito à medida que a equipa sabe mais sobre o que existe para fazer.

Os principais artefactos usados pelo *scrum* são:

- “product backlog”, que é a lista de toda funcionalidade necessária para o produto e é o repositório de todos os requisitos e alterações a realizar ao produto;

- “sprint backlog”, é o conjunto de elementos selecionados no *product backlog* para implementar no *sprint*;
- e a monitorização transparente do trabalho em curso no *sprint* e da progressão nos objetivos definidos para o produto, muitas vezes apresentados em gráficos que permitem visualmente verificar a progressão.

Nesta metodologia os requisitos são apresentados como histórias de uso que descrevem um utilizador, como pretende usar uma funcionalidade do sistema e a razão pela qual precisa de usar essa funcionalidade. As histórias de uso são essenciais para a implementar uma funcionalidade que vá ao encontro das expectativas do utilizador e é fundamental que cada membro da equipa use as histórias de uso para:

- acompanhar tudo o que implementa no produto;
- avaliar o seu trabalho pela perspetiva do utilizador;
- discutir o trabalho e alinhar com o resto da equipa;
- definir as prioridades do seu trabalho.

Uma história de uso deverá conter a seguinte informação:

- quem usa o serviço;
- o que o utilizador ou ator pretende fazer com o serviço;
- porque precisa do serviço;
- e de um modo geral poderá ainda conter os critérios para o teste e a aceitação da história.

Os testes são uma atividade essencial na engenharia de software e permitem verificar se o sistema funciona como era pretendido e também para identificar alguns potenciais problemas, sendo de uso generalizado na indústria na garantia da qualidade do trabalho realizado.

No desenvolvimento ágil de software, os testes não são apenas uma atividade que ocorre uma única vez entre a conclusão de todo o código e a aceitação pelo cliente, mas recorrentemente em cada ciclo de entrega, para nas novas funcionalidades demonstrar e nas antigas validar o seu bom funcionamento.

## 2.4 Testes

O SWEBOK V3.0 ("Guide to the Software Engineering Body of Knowledge") defende que nos anos mais recentes se tem vindo a criar uma visão mais construtiva da atividade de teste de software, levando a que não sejam apenas vistos como uma atividade a realizar no final da fase de codificação com o intuito de detetar falhas, para sim como uma atividade mais constante ao longo do ciclo de desenvolvimento e manutenção como forma de potenciar a qualidade do software, atuando de forma preventiva mostrando fragilidades como contradições, omissões ou ambiguidades na definição e/ou implementação dos requisitos.

Existem vários tipos de testes de software, de modos de teste, de tipos de resultados e outras definições que importa rever para o tratamento do tema e que se estão na lista seguinte baseada no ISO/IEC/IEEE 24765 (versão simplificada):

- **teste de software** consistindo numa verificação dinâmica de que o programa apresenta o comportamento esperado num conjunto finito de casos de testes, escolhidos de forma intencional de um, normalmente infinito, domínio de execução;

- **fiabilidade** é a capacidade de um sistema ou componente desempenhar a funcionalidade requerida nas condições definidas pelo período estabelecido;

- **validação** é o processo de avaliação de um sistema ou componente durante ou no final do processo de desenvolvimento para determinar se respeita os requisitos especificados (“building the right product”);

- **verificação** é o processo de avaliar um sistema ou componente para validar se o produto de uma dada fase de desenvolvimento satisfaz as condições definidas no inicio dessa fase (“building the product right”).

As incorreções ou desajustes encontrados, ainda segundo a definição do IEEE podem ser dos seguintes tipos:

- **engano** é uma ação humana que conduz a um resultado incorreto;

- **falha ou defeito** é um passo, processo ou dados incorretos no sistema;

- **insuficiência** é incapacidade do sistema ou componente para desempenhar a função pretendida com o desempenho requerido;

- **erro** é uma diferença entre um valor calculado, observado ou medido ou uma condição e o valor especificado, verdadeiro ou teoricamente correto ou a condição correta.

Os modelos de teste podem ser de diferentes tipos:

- **caixa branca**, em que existe conhecimento da composição interna do programa e o teste é realizado seguindo diversos caminhos lógicos e observando o que se passa internamente;

- **caixa negra**, em que não existe conhecimento a composição interna do programa, o teste é realizado com base nas especificações e observando o comportamento externo. As condições de teste tendem a crescer sem limites e em termos práticos é apenas testado um subconjunto dos caminhos possíveis;

- **caixa cinzenta**, que é uma mistura dos dois modelos descritos.

Os tipos de testes podem ser:

- **testes unitários**, realizados pelo programador para em modelo e caixa branca testar individualmente a funcionalidade que implementou;
- **testes de integração** realizados pelo programador para testar a integração entre componentes usando o modelo de caixa branca e para alguns de caixa preta;
- **testes funcionais ou de sistema**, teste global normalmente realizado por uma equipa externa em modelo de caixa preta e que pode incluir testes de stress, de performance de usabilidade;
- **testes de aceitação**, normalmente realizados pelo cliente em modelo de caixa preta;
- **testes de regressão**, que são testes seletivos para verificar que uma dada modificação não teve impacto no código ou funcionalidades anteriores, são realizados em modelo de caixa preta, normalmente baseados em casos de teste;
- **testes "beta"**, realizados com um conjunto limitado e controlado de utilizadores a usarem o sistema de forma real e antes de o expor ao uso global.

Tal como o desenvolvimento, também os testes têm um ciclo de vida, que inclui as seguintes fases:

- Especificação dos testes;
- Preparação do ambiente;
- Planeamento dos testes;
- Desenho dos testes;
- Automação dos testes;
- Execução;

- Monitorização e reporte;
- Repetição de testes e regressão;
- Avaliação e fecho dos testes.

A execução dos testes pode ser: manual; por aplicação de comandos capturados num anterior teste manual; baseados em scripts de comandos; ou de geração automática com base em palavras-chave ou em modelos do sistema.

A automação dos testes é mais do que apenas a sua execução, depende da testabilidade do produto e é muito vulnerável à obsolescência rápida com a evolução do sistema. Por outro lado, a automação não consegue substituir o humano por completo e por vezes o foco na automação pode levar à perca do objetivo principal, o bom teste do produto. Em termos económicos o apelo para automatização de testes é enorme, consideremos por exemplo o produto Jira da Atlassian, que tem mais de 14.000 testes unitários e 19.000 testes funcionais usáveis em testes de regressão, o que se fossem realizados inteiramente por um humano a uma velocidade de 50 testes por hora, exigiria mais de 3 meses de trabalho (660 horas) e muitos milhares de euros para uma única execução num produto que apresenta regularmente novas funcionalidades.

### 2.4.1 Testes de regressão

A unidade de testes de uma funcionalidade é à história de uso. O teste deverá ser escrito e suportado de forma a poder mostrar a implementação da funcionalidade no ciclo de desenvolvimento em que foi implementada, mas também a poder ser usado nos testes de regressão dos ciclos seguintes.

No processo de desenvolvimento um programador deverá desenvolver os testes unitários, por forma a testar a correção do seu código, o que normalmente realiza com a mesma linguagem com que programou e em modelo de caixa branca, isto é, com conhecimento das particularidades internas do código.

Os testes ao nível da história de uso são testes de caixa preta, realizados nas interfaces (API, "webservices", ecrãs) do sistema em testes comparando a resposta ao conjunto de entradas fornecidas e verificando assim o comportamento do sistema. Estes testes são muitas vezes feitos de forma manual com base em guiões de teste, ou suportados em ferramentas e descritos numa linguagem independente da usada no desenvolvimento do sistema.

Os testes de jornada são compostos por sequências lógicas de testes de histórias e pretendem mostrar o funcionamento ao longo de um caso de teste, por exemplo desde o registo de uma nova apólice de seguro num sistema até à emissão do primeiro recibo. Também estes são muitas vezes realizados e documentados de forma manual com trabalho intensivo e cuja automatização é bastante interessante.

Para permitir a automatização ao nível da jornada de testes é necessário, que exista um sistema de suporte aos testes das histórias de uso, que tenha a flexibilidade para criar as pré-condições que um teste individual de uma história precisa e encadear saídas e entradas entre testes numa sequência testes de histórias de uso que fazem parte de uma jornada ou caso de teste.

### 2.4.2 Testes baseados em modelos

O sistema em teste, ou o próprio ambiente de teste, podem ser descritos em modelos abstratos com descrição da funcionalidade ou do comportamento do sistema e esses modelos podem então ser usados na geração automática de testes. Estes testes são referidos por testes baseados em modelos ("MBT: Model-based testing") e são compostos pelas seguintes atividades:

- Modelização do sistema em testes e ou do seu ambiente;
- Geração de testes abstratos a partir do modelo criado;
- Concretização dos testes abstratos em por forma a torna-los executáveis;
- Execução dos testes no sistema em testes e recolha dos resultados;
- Análise dos resultados obtidos e tomada das medidas corretivas necessárias.

As principais vantagens dos testes baseados em modelos são:

- Aumentam a possibilidade de deteção de erros;
- Redução dos tempos e custos de testes, em particular na sua escrita e manutenção;
- Melhoria da qualidade dos testes por medida do nível de cobertura das funcionalidades implementadas;
- Deteção de precoce de erros nas especificações durante o processo de modelação (e antes da codificação do sistema);
- Rastreabilidade entre os requisitos e o modelo e entre este e os testes e seus resultados;
- Atualização automática dos testes para refletir alterações dos requisitos e da consequente alteração do modelo.

Quanto às principais limitações são a necessidade de manutenção do modelo de sistema atualizado, a modelização errada do sistema gera testes sem sentido e o potencial grande volume de informação gerados pelos testes automatizados.

A modelização do sistema pode ser baseada com um dos seguintes paradigmas:

- Estados do sistema com caraterização de condições pré e pós execução;
- Transições, isto é descrição das transições entre estados;
- Comportamento histórico dos comportamentos ao longo do tempo;
- Funcional, com coleções de funções matemáticas;
- Operacional, coleções de processos executáveis em paralelo;

- Estatístico com base nas probabilidades de valores de entrada e de acontecimentos;
- Fluxo de dados.

Dentro destes paradigmas os mais usados são os 2 primeiros relativos aos estados do sistema, com base em notações específicas criadas para descrever o sistema.

### 2.4.3 Planeamento automático

A automatização dos testes pressupõe que o plano seja obtido de forma automática e servirá para guiar a execução automática dos testes. Um sistema de planeamento capaz de produzir automaticamente o plano precisará das seguintes funções (Rich, 1985):

- Escolher a melhor regra (operação) a aplicar com base na heurística definida;
- Aplicar a nova regra (operação) para atingir o novo estado do sistema;
- Detetar quando a solução pretendida foi atingida;
- Detetar caminhos sem seguimento (“dead ends”) por forma a serem abandonados e a redirecionar o sistema para caminhos de maior resultado.

A modelização dos operadores com uma estrutura do tipo STRIPS e a definição de estado com base no conjunto de axiomas disponíveis a cada momento permitem responder aos requisitos para o planeamento. Podemos assim usar uma forma de planeamento para criar um plano de testes que permita ligar um estado inicial até a um estado final. Este caminho, se existir, verifica a coerência entre os requisitos e a execução e um dado processo (ou processos) de negócio.

Para adaptar ao caso das aplicações empresariais precisamos ainda de poder definir estados intermédios de passagem obrigatórios. O sistema deverá gerar sublanços do plano para ligar os diferentes estados, inicial, intermédios e final, em que cada sublanço é um

plano gerado de forma igual ao plano total e que pode ser visto como um novo operador, gerado automaticamente, a partir dos operadores de base introduzidos no sistema. Com este conceito o sistema ganha a capacidade de usar planos dentro de planos, permitindo abstrair complexidade e reusar trabalho anterior de geração de planos. Um exemplo que demonstra a necessidade de impor estados intermédios, poderá ser o criar e apagar uma entidade do sistema, sem imposição do estado intermédio um planeador não iria gerar este passo que não conduz a nenhum axioma final. A passagem por um estado intermédio permitirá planear esse teste.

O planeamento poderá ser realizado com diversos algoritmos, mas para o processo em estudo pretendemos que seja sequencial. Um processamento sequencial permite seguir um diagnóstico causa efeito para o rastreio dos erros que sejam encontrados na implementação, enquanto umas execuções de atividades em paralelo tornam esse processo de rastreio mais difícil e menos determinístico (algumas pequenas diferenças de velocidades entre os ramos podem levar a resultados diferentes ou a erros intermitentes). Nada impede, contudo, que vários planos de teste possam ser executados em paralelo para testes da implementação, o que permitirá avaliar o comportamento global da aplicação e a deteção de erros.

O foco na geração do plano está na verificação da consistência dos requisitos e na sequência de aplicação dos operadores, pelo se pretende encontrar um caminho que possa ligar estado inicial e final, sempre que exista, mas não há um objetivo em encontrar o caminho mais curto ou ótimo. A utilização de macro operadores, facilmente levará à geração de caminhos menos otimizados, o que poderá ser visto de forma dual, numa primeira fase como vantajosos pois exercitam mais a aplicação, mas podem ser menos eficientes em termos de testes de regressão. Neste estudo a opção foi para valorizar a cobertura de teste da aplicação e, portanto, de relaxamento relativamente à escolha do caminho ótimo, seria, contudo, possível atribuir um custo a cada caminho e usar um algoritmo que o considerasse para a seleção do caminho ótimo, haverá seguramente casos em que isto seja o mais adequado.

Os planos gerados poderão ser executados repetidamente sobre a aplicação em testes, constituindo a base dos testes de regressão, elemento fundamental para o desenvolvimento ágil. Os passos da execução e os seus resultados deverão ser registados,

para documentar a sua execução e para permitir a análise e diagnóstico de eventuais anomalias.

Naturalmente a alteração de um operador ou dos estados impostos na geração de um plano, inviabilizarão o plano existente e implicam a geração de um novo. Esta é uma das vantagens inerente do sistema proposto, que é a da atualização automática dos planos de teste quando são alterados os operadores.

### 2.4.4 A ferramenta de testes

A ferramenta e processos propostos têm por foco a automatização dos testes de regressão a usar de forma repetitiva no software em teste. O teste assenta nos princípios de "caixa preta", isto é, focado no comportamento apresentado, sem conhecimento dos pormenores interiores de implementação e muito virado para arquiteturas orientadas a serviços, onde o foco é no comportamento do sistema e não na sua interface gráfica.

A ferramenta corresponde à implementação de um Agente Artificial Inteligente, composto por uma hierarquia de controladores, que interagem com o sistema em testes (SUT– "system under test") com base em planos introduzidos, ou gerados automaticamente com base numa história de uso. A linguagem usada para descrever os planos, e complementar as histórias de uso, é agnóstica quanto à tecnologia usada no sistema em teste dispondo de um conjunto de verbos para uso com os mecanismos standards de integração ("webservices" e via browser), mas ser complementada e enriquecida com outras ações, para resolver casos específicos ou alargar o uso da ferramenta. A linguagem de escrita dos planos deverá ser simples para ser usada por não programadores (ou pelo menos não com o mesmo nível de exigência dos que trabalham no desenvolvimento do sistema em teste).

## 2.5 Agente Inteligente

Um agente é uma entidade capaz de percecionar o ambiente onde se encontra através de sensores e de interagir com esse mesmo ambiente através de atuadores. Um agente poderá ser um agente robótico ou um agente de software. Um agente será dito racional se

com base na sequência das suas perceções do ambiente e da sua base interna de conhecimentos, conseguir executar as ações certas com vista a maximizar a possibilidade de sucesso, na concretização dos seus objetivos. Um agente inteligente foi definido por Rosenchein (1999), como sendo um dispositivo que interage como o seu ambiente de forma flexível, de forma direcionada aos seus interesses, reconhecendo estados importantes do ambiente e agindo para obter os resultados desejados. Os Agentes Inteligentes são uma área de estudo da Inteligência artificial.

Para o presente trabalho o tipo de agente considerado é baseado no modelo BDI (Belief-Desire-Intention) e que tem por base de referência o comportamento humano e a sua replicação parcial para desenvolver agentes inteligentes (isto é que tenham um comportamento que possamos classificar razoavelmente de inteligente).

O Agente BDI, com base nas suas perceções cria um modelo do que julga ser o mundo, que depois usa para enquadrar os seus desejos construindo as intenções (e planos de ação associados) que possam promover a sua realização.

O raciocínio prático do agente é estruturado com base nas semelhanças ao comportamento humano e criando um estado interno com a seguinte estrutura:

- Estado Denotacional – as crenças, o que o agente "acredita" ser o mundo;
- Estado Motivacional – os desejos, o que o agente deseja atingir;
- Estado Intencional – as intenções, o que o agente tem intenção de concretizar.

Esta estruturação está representada no modelo de agente BDI, que tem como elementos chave do raciocínio pratico os elementos crenças, desejos e intenções e que estão contidos na memória do agente, conforme o diagrama seguinte:

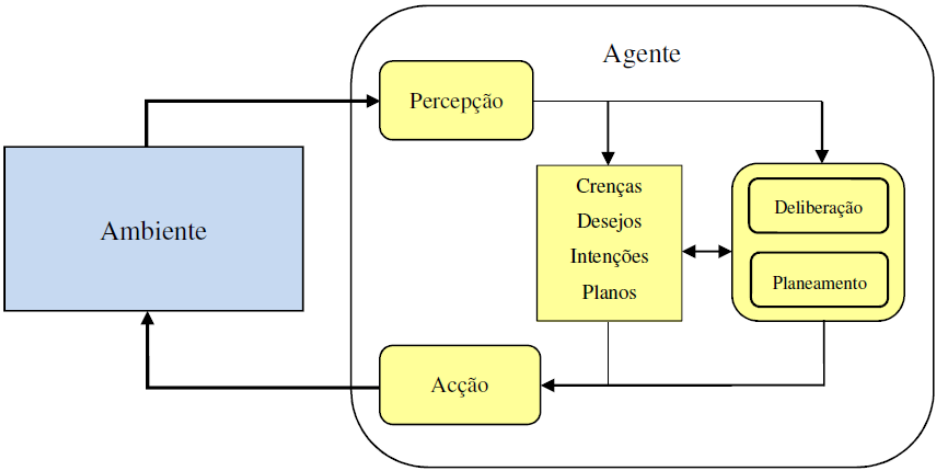


*Figura 2- Arquitetura de um agente BDI*

Na Figura 2- Arquitetura de um agente BDI, o agente capta informação do ambiente com base nas suas perceções, que alimentar as crenças e participar no processo de deliberação, que conjuga desejos para produzir planos de intenções. Estes planos geram ações que atuam no ambiente.

### 2.5.1 Crenças

As crenças representam o que o agente acredita acerca do mundo, as quais não são necessariamente verdadeiras. Se o agente obtém a informação das suas crenças através da perceção, até ocorrer nova perceção poderão ocorrer alterações no mundo que o agente não tem conhecimento. As crenças contêm normalmente toda a informação do mundo que o agente obteve, incluindo dele próprio e de outros agentes que coexistam.

### 2.5.2 Desejos

Os desejos representam o estado de motivação do agente, os fins que o agente deseja alcançar, os quais são estados do mundo a serem concretizados.

### 2.5.3 Intenções

As intenções são os desejos escolhidos pelo agente para serem concretizados, ou seja, o agente delibera sobre os desejos para escolher os que são mais concretizáveis numa determinada situação. O compromisso de concretização das intenções dá origem a planos de ação.

### 2.5.4 Processamento

A tomada de decisão é um processo cíclico, que segue o processo geral:

- Percecionar o ambiente (mundo) em que o agente está inserido;
- Atualizar as crenças do agente;
- Se necessário reconsiderar o plano que está em execução:
    - Considerar as opções (Desejos) e selecionar as opções (Intenções)
    - Planear a ação
- Executar o plano de ação.

# 3 Proposta de solução

A abordagem seguida para a redução do impacto dos custos dos testes no desenvolvimento ágil de software, tem por base o uso da geração de testes baseados num modelo abstrato do sistema e na implementação de um executor para a concretização dos testes no sistema em testes. Como proposta solução global é apresentado o modelo de uma ferramenta de testes, composta por um agente inteligente, um planeador e uma interface de utilizador, para a modelização do sistema em testes e um executor para interagir com o sistema em testes.

A abordagem seguida foi de especialização nas caraterísticas específicas do desenvolvimento ágil, das aplicações empresariais para a web e nos testes de regressão, que definem os seguintes elementos usados na solução:

- No desenvolvimento ágil (scrum em particular), os requisitos são descritos como histórias de uso, que correspondem à realização de uma determinada atividade pelo sistema, e que por isso o mudam de um estado para outro. A unidade de testes deverá por isso ser a história de uso, permitindo a rastreabilidade entre o requisito e o teste respetivo;

- Os testes de regressão são um tipo fundamental de teste no desenvolvimento ágil, caraterizado pela reutilização dos testes já escritos;

- As aplicações empresariais web, expõem interfaces para utilizador que pressupõem o uso de browsers, expondo outras funcionalidades em serviços. Estas duas interfaces podem ser exploradas nos testes da aplicação de forma independente da linguagem em que foram escritas, permitindo a construção e um agente genérico para a execução dos testes.

Embora a solução criada possa servir outras formas e paradigmas de desenvolvimento, pareceu mais interessante procurar focar num ambiente bem definido para maximizar a eficácia em vez de procurar uma generalização para uma atuação com potencial menor valor para este ambiente em concreto.

A abordagem proposta é composta pelos seguintes elementos:

- Modelização do sistema em testes;
- Definição de um conjunto de comandos para interação com o sistema em teste;
- Definição de um modelo de arquitetura para um agente inteligente executor dos testes;
- Utilização de um planeador para a geração automática dos testes abstratos ao sistema;
- Sugestão de complemento ao processo ágil.

A base dos requisitos no desenvolvimento ágil são as histórias de uso, elas descrevem a funcionalidade esperada do sistema e são o elemento base no planeamento do projeto, com base nas suas iterações. Uma história é desenvolvida no contexto de um sprint e é demonstrada no seu final como prova da sua implementação. Para a demonstração da funcionalidade descrita numa história, é necessário executar um conjunto de interações com o sistema em teste, por exemplo usando uma interface web e validar o comportamento do sistema em resultado dessa atividade. Esse conjunto de interações pode ser construído com base num conjunto de comandos propostos no presente trabalho, formando um plano de teste da história de uso que pode mais tarde ser reutilizado, para testar o bom comportamento do sistema.

Os processos de negócio mais complexos são formados por sequências de histórias de uso e representam uma jornada de um do sistema por um dado utilizador na sua atividade com o sistema. A nível abstrato cada história de uso pode ser vista como uma operação individual, que permite realizar atividades e alterar o estado do sistema. A modelização do sistema do sistema com base na descrição de estados é feita história a história descrevendo as pré-condições necessárias à sua execução e as pós-condições resultado da sua atuação. Com base nas pré-condições e pós-condições é possível criar

sequências de histórias que levam o sistema de um dado estado a outro. Essas sequências são referidas por Jornadas. Uma jornada é sequência de testes abstratos, que por si permitem validar a consistência dos requisitos. Os testes são concretizados no teste escrito para cada uma das histórias com base num conjunto de comandos que permitem atuar sobre o sistema em testes, recolher feedback e documentar a realização dos testes. Os testes de jornada e de história são reutilizáveis, daí o seu interesse para os testes de regressão.

A modelo de arquitetura proposto para a ferramenta de testes de regressão tem as seguintes características principais:

- Suporte à execução de testes de histórias de uso, com base num conjunto de comandos desenhados para interagir com aplicações web;
- Suporte à modelização do sistema em testes, para suporte à geração de planos abstratos de testes (de jornada);
- Conversão de planos abstratos de jornada para concretização em sequências de planos concretos de execução de histórias de uso;
- Documentação de todos os testes realizados;
- Interface de utilizador, para a modelização do sistema em testes e controlo da atividade do controlador;
- Reutilização de planos de testes;
- Arquitetura modelar com suporte a componentes de código aberto, arquitetura de fácil extensão para suportar novos comandos e funções específicas, possível expansão para ambiente multiagente.

## 3.1 Teste de uma história de uso

A história de uso descreve uma funcionalidade do sistema a desenvolver no contexto de um único ciclo de desenvolvimento. O teste da história servirá para provar o seu bom e adequado funcionamento. A montagem deste teste é manual, com base num conjunto de verbos disponíveis na ferramenta e que se sejam executados sequencialmente no sistema em teste, registando os resultados obtidos num jornal ou diário de atividade ("log"). Este conjunto de passos forma um plano que é referido como Teste de História. A execução de um passo poderá originar um erro grave, por exemplo uma exceção ou apresentar um resultado de acordo ou não com o esperado e condicionar a continuidade de execução do plano.

Na ferramenta de testes, o plano de um Teste de História é coordenado pelo controlador executivo do agente, que com base no resultado de cada passo decidirá da continuação da execução do plano, ou pela aplicação de ações para a resolução do erro encontrado, antes de volta a executar o passo.

A capacidade de reagir a erros locais pode ser demonstrada por este exemplo: no teste do preenchimento de um formulário web, o servidor redireciona a aplicação a página de login, impedindo a execução do teste de preenchimento do formulário. O controlador ao deparar com o erro, procurará na sua biblioteca de planos locais, um que possa responder ao erro encontrado (login) se existir, aplica a correção encontrada e reexecuta o preenchimento do formulário. Esta correção automática é semelhante à que um operador humano faria e permite retomar o processamento dos testes (a decorrer por exemplo durante a noite sem uma supervisão em tempo real). A biblioteca local poderá ser carregada com outros planos para resolver problemas locais ou exceções. Em cada plano é definido o seu nível de tolerância a falhas, ou seja, se deverá continuar a execução com base num nível de erro.

Cada Teste de História é composto por dois planos, um plano de teste da funcionalidade e um outro para a preparação do ambiente de execução e que corre antes do principal. Se o plano fizer parte de uma sequência de planos (Teste de Jornada) então a preparação não é executada. A execução de um plano ocorre dentro de um contexto recebido, conjunto de variáveis que podem ser usadas no plano e na sua conclusão o novo

contexto será passado para o plano seguinte, caso exista. Finalmente um passo de um plano pode ser um outro plano local, a executar no passo em que está referido.

O plano seguinte ( Figura 3- Plano de "Login" ) descreve o Teste de História do login, com base num conjunto de verbos disponibilizados, a que passa os valores a usar e as condições de sucesso:

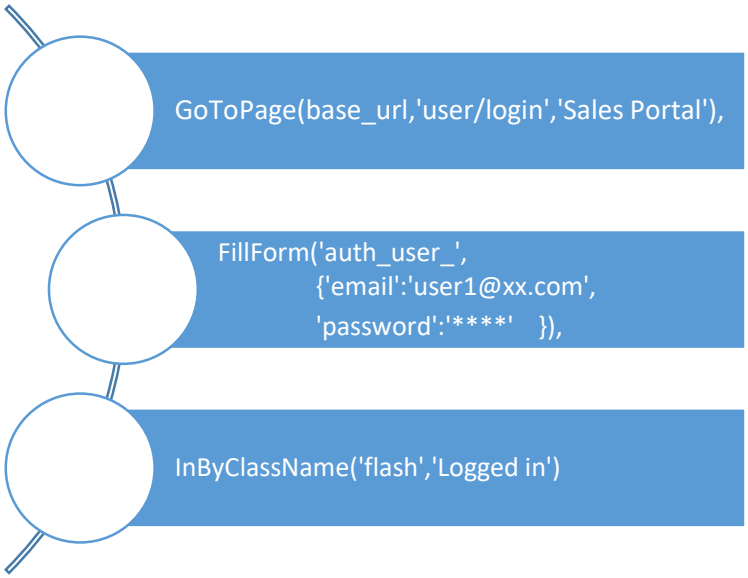


*Figura 3- Plano de "Login"*

No plano apresentado o verbo **GoToPage** indica a navegação para a página 'user/login' e procura no título a designação 'Sales Portal' para verificar o sucesso na navegação. Preenchidos os campos do formulário com sucesso, com os valores indicados, a classe indicada 'flash' deverá ter a indicação de 'Logged in'. No verbo **FillForm** 'auth_user_' é um prefixo acrescentado a cada id dos campos do formulário e depende da framework usada na implementação do sistema em teste. O verbo **InByClassName** é o exemplo de uma verificação do sucesso da sequência de passos anteriores, comparando o conteúdo do elemento da classe "flash" com a cadeia de carateres "Logged in", que neste caso reflete a conclusão da atividade de identificação junto do sistema em testes. À ferramenta de testes podem ser adicionados novos verbos para usar na construção de planos.

Nos verbos criados foi considerada uma notação do tipo “json”, para as estruturas compostas.

## 3.2 Ações

No plano apresentado, os verbos referidos representam ações a executar pelo agente. As ações podem ser de diversos tipos, internas, externas ou referências a outros planos.

As ações internas atuam sobre o próprio controlador, alterando as suas variáveis de controlo, ou interagindo com o seu funcionamento e incluem a definição de variáveis a usar nos verbos dos planos e ações de configuração do controlador, sendo as mais comuns a definição do nível de registo no diário (logging) e da tolerância do controlador. O verbo seguinte é um exemplo de uma ação local de configuração do controlador:

**SetController**( {'logging':'debug'})

que indica que o novo nível de “logging” pretendido é o mais elevado, com registo de todas as atividades com o detalhe máximo para uso em “debug”.

As ações que são referências a outros planos, funcionam como chamadas a uma função nas linguagens de programação, ou seja, a execução transita para o plano referido e quando terminar a execução desse plano é retomada a execução do plano original.

As ações externas implicam atividade fora do controlador, seja interagindo com o sistema em teste ou com o controlador do nível seguinte. As ações externas que interagem com o sistema em teste, diferem na forma como o fazem e naturalmente da interface que usam. Uma interface comum são os serviços web e em particular as interfaces “rest”, com os métodos de “get”, “post”, “update” e “del”. No caso do acesso a um serviço “rest” com o método “get” o verbo usado é o RestGet, que na sua versão mais completa tem a forma:

**RestGet**( url, payload, header=None,check_result=None, filter=None)

Em que “payload” representa a estrutura de informação em formato “json” a enviar para o “url”, “header” permite por exemplo enviar informação de autenticação e “check_result” e “filter” são duas funções, que permitem validar se o resultado é ou não o esperado para o teste e retirar apenas o que se pretende manter disponível para passos seguintes do plano, respetivamente. Naturalmente a preparação deste plano de teste requer a intervenção do programador ou equipa que desenvolveu a funcionalidade, mas não precisa de ser feita na mesma tecnologia em que foi implementada a funcionalidade. Em certa medida permite que o componente ou subsistema possa substituído por um outro que exponha a mesma a interface para efeitos desta funcionalidade.

O código de erro, resultante da execução da ação, é comparado com o nível de tolerância definido para o teste ao nível do plano e poderá determinar o cancelamento da execução caso tenha sido excedido. No caso em que o erro é tolerado pelo plano, será pesquisado na biblioteca de planos, um definido para atuar no caso do erro encontrado e se resultar, retornar a executar o passo que gerou erro. Os códigos de erro podem das gamas ACT_OK, ACT_WARNING, ACT_ERROR e ACT_CRITICAL e os seus detalhes são descritos adiante.

Vejamos um exemplo, num acesso a um site pode ser solicitada uma página cujo acesso está condicionado a utilizadores autenticados no sistema. Num processo de teste manual o técnico efetuaria o "login" no sistema e repetiria o acesso à página bloqueada. Pretende-se que o agente apresente este mesmo comportamento, por isso na presença de um erro de acesso à página e verificando na mensagem de erro que se trata de um acesso sem autenticação, o controlador procurará na sua biblioteca de planos um que lhe permita fazer o "login", caso encontre esse plano e tenha sucesso na sua execução, repetirá a tentativa de execução do passo originador do erro.

## 3.3 Teste de uma jornada de uso

Uma jornada refere um caso de teste do sistema em testes e é composta por vários testes de histórias de uso. A sua execução é sequencial e coordenada pelo controlador tático.

O controlador tático coordena a execução do controlador executivo e é ele que dá indicação dos planos a executar e que configura o controlador executivo.

O controlador tático executa ele próprio planos, da mesma forma que o controlador executivo, contudo com recurso a verbos diferentes, pois as ações são outros planos ou testes de histórias a passar para o controlador executivo. De forma geral o controlador tático e executivo são iguais, apenas as ações que executam são diferentes.

Os planos do controlador tático são, portanto, testes de jornada e podem ser programados manualmente com os verbos disponíveis a este nível. Os verbos disponíveis a este nível para definir cada passo são:

- Referências a planos executivos (testes de histórias) cuja execução será solicitada ao controlador executivo;
- Referências outros planos do nível tático (testes de jornada), para expandir e executar nesse passo;
- Ações de controlo do controlador tático.

Um teste de história é referido por um termo que inclui o seu nome e as variáveis de passagem que utiliza. O exemplo a seguir (Figura 4 - Plano de Jornada - compra) é de um plano para criar um novo produto, um novo cliente na aplicação fictícia “Portal de Vendas”, colocando o produto no cabaz desse cliente:

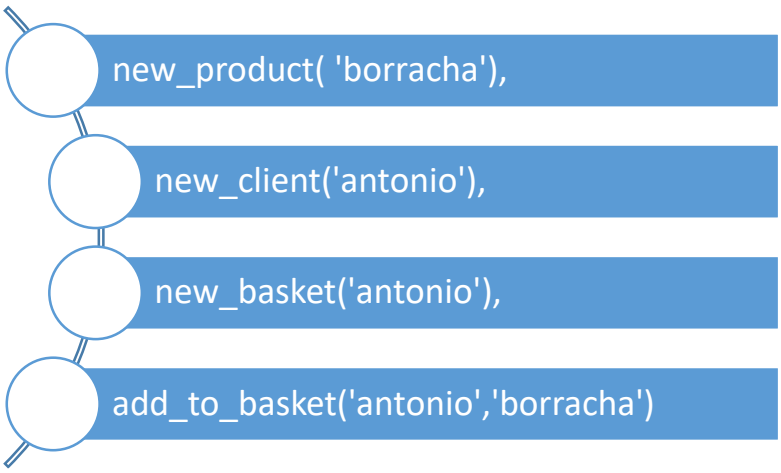


*Figura 4 - Plano de Jornada - compra*

Os termos ‘borracha’ e ‘antonio’ são etiquetas que permitem distinguir um produto de um outro produto ou um cliente de outro a nível lógico e que têm uma representação mais concreta no sistema em testes, com os atributos usado nas entidades desse sistema. A nível lógico as etiquetas são suficientes para distinguir as entidades nas operações dos planos.

A sequência de passos do plano refere testes de histórias, que o controlador tático encarregará o controlador executivo de efetuar no sistema em testes. A passagem para o passo seguinte será condicionada à informação de sucesso retornada pelo controlador executivo. No caso da execução isolada de teste de história o plano será o apresentado na “Figura 5- Plano - novo cliente”

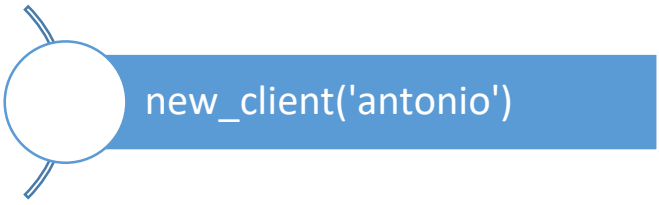


*Figura 5- Plano - novo cliente*

Os verbos dos planos de Teste de Jornada são os nomes dados aos planos de teste de história, pelo utilizador no seu carregamento no sistema.

Os planos de teste (de história ou jornada) são guardados numa base de dados do agente inteligente e podem ser executados repetidamente com periocidade definida como testes de regressão. A instrução para a execução de um dado plano de testes de jornada, pelo controlador tático, é recebida do controlador estratégico, que coordena as repetições de testes, ou seja, que controla todo o processo de execução dos testes de regressão.

## 3.4 Testes de regressão

Os testes de regressão são repetições de execução dos testes disponíveis, que verificam a manutenção do bom funcionamento das funcionalidades existentes. O objetivo da ferramenta proposta é o suporte e execução destes testes nos sistemas em teste, tendo por base um agente artificial inteligente. O agente artificial é composto por controladores, o controlador executivo, o tático e o estratégico, sendo os dois primeiros usados para a execução dos testes de história e de jornada, respetivamente.

O controlo estratégico é o coordenador de toda a atividade do agente. É ele que determina os testes a executar e que dá instruções ao controlador tático. A interface com o utilizador existe apenas a este nível que gere toda a programação de eventos. Os testes de uma história ou jornada podem estar programados para serem repetidos com uma dada periodicidade e é este controlador o responsável por disparar todo o processo nos momentos previstos.

O controlador executivo coordena também a geração automática de planos de jornada, como alternativa à definição manual destes planos. A geração automática parte da definição das histórias de uso complementadas com alguma informação adicional, para com base num planeador produzir o plano de teste de jornada.

## 3.5 Geração automática de testes de jornada

As histórias de uso são uma forma de descrição dos requisitos funcionais de um sistema, que ligam um ator a uma operação ou procedimento que tem um objetivo. Cada uma das histórias tem por base um conjunto de condições à sua aplicação e colocará o sistema numa nova condição ou estado, como resultado da sua atividade. As pré-condições e a atividade podem ser incluídas na descrição de uso, como conjuntos de factos, que precisam de existir e que podem ser retirados ou acrescentados com a operação descrita pela história de uso.

Um dado estado do sistema poderá então ser descrito pelos factos disponíveis e estes condicionam as operações que podem ser realizadas e que levarão o sistema para um novo estado. Assim dados dois estados do sistema, descritos pelos seus factos, podemos determinar o conjunto de operações necessárias para levar o sistema de um estado ao outro.

A linguagem formal STRIPS (Richard Fikes and Nils Nilsson, 1971) usada como input para um planeador com o mesmo nome, pode ser usada para expressar as pré-condições e o resultado produzido a nível abstrato pela história de uso e construir os operadores, necessários à utilização deste planeador STRIPS. A utilização do planeador permite assim a geração automática dos planos para os testes de jornadas, se existir um conjunto de operadores (derivados das histórias de uso) que permitam levar o sistema de um estado inicial ao estado objetivo ou final pretendido. Se o plano existir então podemos verificar que, a nível abstrato, os estados referidos e as operações definidas são consistentes e mais tarde podem conduzir a validação da implementação concreta realizado com o sistema em testes.

Daqui resulta um segundo interesse do sistema proposto, não só pode ser útil para a automatização dos testes de regressão ao sistema implementado, mas numa fase prévia ao desenvolvimento pode ser usado para verificar a consistência das histórias / requisitos do sistema, antes da produção de código.

Um exemplo de um operador STRIPS, usando operações do “mundo dos blocos”, pode ser o seguinte:

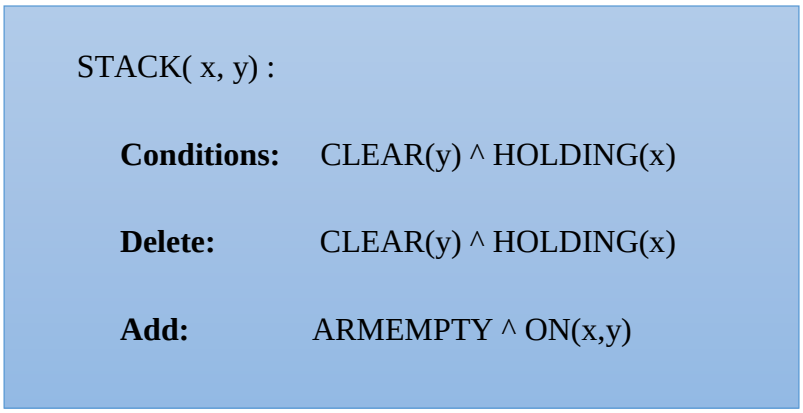


*Figura 6 - Exemplo de operador STRIPS*

Este exemplo (Figura 6 - Exemplo de operador STRIPS) descreve a operação de um robot colocar um bloco "x" sobre o bloco "y", em que "STACK(x,y)" define o nome do operador e indica as variáveis que podem ser usadas, "Conditions" é a lista de condições para a aplicação do operador. Estas condições são um subconjunto dos axiomas que caracterizam um estado. "Delete" e "Add" são duas listas com os axiomas a remover a adicionar respetivamente, como consequência da aplicação do operador ao sistema. Outros potenciais axiomas existentes no estado, não indicados no operador, são considerados irrelevantes para a sua aplicação e são mantidos inalterados pelo operador.

Os estados de um sistema são descritos e diferenciados pelo conjunto de axiomas existentes e consequentemente pelo conjunto de operadores que podem ser usados.

Os axiomas são uma forma simples de descrever o sistema, que podem conter ou não variáveis como no exemplo CLEAR(y) e ARMEMPTY respetivamente. A sua sintaxe é simples e sem grandes restrições, contudo o sistema ficará gradualmente mais complexo com o seu aumento e deverá haver alguma atenção na sua utilização e em particular para evitar duplicações, redundâncias e manter uma boa legibilidade e entendimento da funcionalidade de cada um.

Na implementação proposta, a escrita dos axiomas deverá respeitar as regras da linguagem Prolog, tendo sido apenas acrescentadas as seguintes novas regras necessárias ao mundo das aplicações alvo deste estudo:

- **not_exist(** fact )  -- regra verdadeira quando o facto não estiver definido
- **in(**element, list)  -- regra verdadeira quando o elemento exista na lista
- **not_in(**element, list) -- regra verdadeira quando o elemento não exista na lista

O operador **in**() é normalmente usado para definir o domínio das variáveis usadas nos operadores de estado.

Tendo por base a riqueza da linguagem Prolog, é possível definir e acrescentar mais regras, contudo há uma grande virtude em manter a escrita o mais livre e intuitiva possível por forma a permitir a sua utilização por analistas não necessariamente programadores. As regras acrescentadas que foram referidas, parecem, contudo, vitais para a representação de sistemas comerciais, pois trabalham em mundos abertos de informação, para melhor explicar este ponto vejamos o exemplo:

- Uma história de uso é relativa ao registo de um novo cliente no sistema. Para registar o novo cliente no sistema é necessário que:
    - O cliente exista no domínio dos valores possíveis para a variável que representa clientes;
    - Que o cliente ainda não esteja registado no sistema.
- Para representar estas duas condições, usamos as regras acrescentadas:
    - **in()** e
    - **not_exists()**.

O exemplo seguinte (Figura 7 - Operadores STRIPS para o exemplo Portal de Vendas) apresentas os operadores gerados para um conjunto de histórias de uso de um "Portal de Vendas" fictício.

```
operador(new_client(Client),
          [ not_exist(client(Client)),
            in(Client,['antonio','jose','francisco'])
          ],
          [ client(Client)],
          []
  ).
operador(new_product(Product),
          [ not_exist(poduct(Poduct)), name(Product)],
          [ product(Product)],
          []
  ).
operador(new_basket(Client),
          [not_exist(basket(Client)), client(Client)],
          [basket(Client, [])],
          []
   ).
```

*Figura 7 - Operadores STRIPS para o exemplo Portal de Vendas*

A ferramenta proposta usa um planeador STRIPS implementado em Prolog e a representação acima resulta de um caso real usado pelo demonstrador.

## 3.6 Diário ou jornal do sistema

O registo e documentação dos testes realizados é um requisito importante da maioria dos projetos. Os testes efetuados, os valores usados e os resultados obtidos devem ser registados de forma a permitir a sua análise em diversas dimensões. Uma primeira dimensão, é dar prova do bom funcionamento (ou manutenção do bom funcionamento no caso dos testes de regressão) uma segunda poderá ser obtida por análise dos próprios dados, na classificação de zonas em função da sua estabilidade com as alterações e mesmo contribuir para automatização de diagnósticos (vertente não desenvolvida no presente trabalho).

O registo dos testes é resultado fundamental das ações individuais usadas nos planos e dos controladores executivo e tático, no relacionamento com Teste de História e Teste de Jornada, respetivamente.

O nível de registo no diário é controlado por omissão ao nível do controlador, mas poderá ser alterado o nível do plano, o que fará apenas sentido enquanto em "debug".

Os níveis de registo considerados são:

- **INFO** - corresponde a um teste que respondeu com o valor esperado;
- **WARNING** - corresponde ao retorno de um valor diferente do esperado. Por exemplo na chamada a um serviço REST este está a funcionar, mas o valor retornado não é o esperado;
- **ERROR** - no exemplo anterior o serviço não está a funcionar, por exemplo estamos a usar um endereço errado;

- **CRITICAL** - uma outra situação critica proveniente de uma exceção ao nível do código;
- **DEBUG** - regista mais detalhe da execução e do resultado para apoiar no diagnóstico de problemas.

Estes níveis estão relacionados com os códigos retornados pela execução ou plano da seguinte forma:

- **ACT_OK** - corresponde a uma boa execução do Step (Ação ou Plano) e pode conter valores de ACT_OK até ACT_WARNING exclusive;
- **ACT_WARNING** - corresponde à receção de um valor diferente do esperado e pode conter valores de ACT_WARNING até ACT_ERROR exclusive;
- **ACT_ERROR** - corresponde a um erro, porque por exemplo o serviço não responde e pode conter valores de ACT_ERROR até ACT_CRITICAL exclusive;
- **ACT_CRITICAL** - corresponde a uma exceção ao nível do código e pode conter valores de >= ACT_CRITICAL ou < ACT_ERROR.

A opção por bandas em vez de códigos fixos, permitirá às ações catalogarem códigos próprios dentro do domínio do tipo de resultado. Esses valores poderão tomar um significado específico para outras atividades sem interferirem com o modelo de comportamento do controlador.

A forma de registo no diário é composta por uma parte comum a todos os registos e que inclui o nível do registo (INFO, WARNING, etc.) o módulo a data e hora da atividade e uma segunda parte produzida pela ação ou plano para documentar as suas caraterísticas. O formato geral da componente comum pode ser alterado.

Os diários são armazenados por projeto, para simplificar a documentação e análise dos testes realizados. Uma área de enriquecimento da ferramenta proposta e que não é

abordada no presente trabalho, é a mineração dos resultados para a possível identificação de padrões comportamentais de erro e repetição de cobertura para usar na definição automática de novos planos e dados de teste.

## 3.7 Resumo

A ferramenta de testes descrita, tem por base um agente artificial inteligente, dedicado à execução de testes funcionais de regressão, executados no contexto de caixa preta. Os testes de cada história de uso são carregados manualmente, os testes de jornada podem ser carregados manualmente ou gerados automaticamente, sendo neste caso necessário complementar a escrita das histórias de uso, com uma descrição em STRIPS. Os testes são descritos em planos que são persistidos em base de dados para sua reutilização.

Uma forma interessante de uso da ferramenta passa pela geração dos planos de teste de jornada, antes da codificação de qualquer história, pois isto permite a nível abstrato verificar a coerência dos requisitos antes de proceder à codificação do novo sistema ou de o alterar.

Os diferentes níveis de atividade do agente são da responsabilidade de diferentes controladores, cujo nível de tolerância a falhas pode ser configurado e que dispõe de alguns planos locais para resolução de alguns tipos de erros em execução de testes.

Os resultados dos testes são documentados e podem ser usados para documentar a atividade e outras análises.

# 4 Arquitetura da solução

A ferramenta de testes proposta tem basicamente dois grupos de utilizadores o analista de sistemas, na definição das histórias de uso, na sua transformação em operações e na definição, execução e avaliação dos testes de jornada e o programador na criação, execução e avaliação dos testes de história de uso. Esta proposta de uso poderá ser alterada em função das dimensões das equipas e das competências dos seus elementos. O diagrama seguinte (Figura 8- Casos de uso da ferramenta de testes) apresenta as funcionalidades expostas pela ferramenta de testes para cada tipo de utilizador.

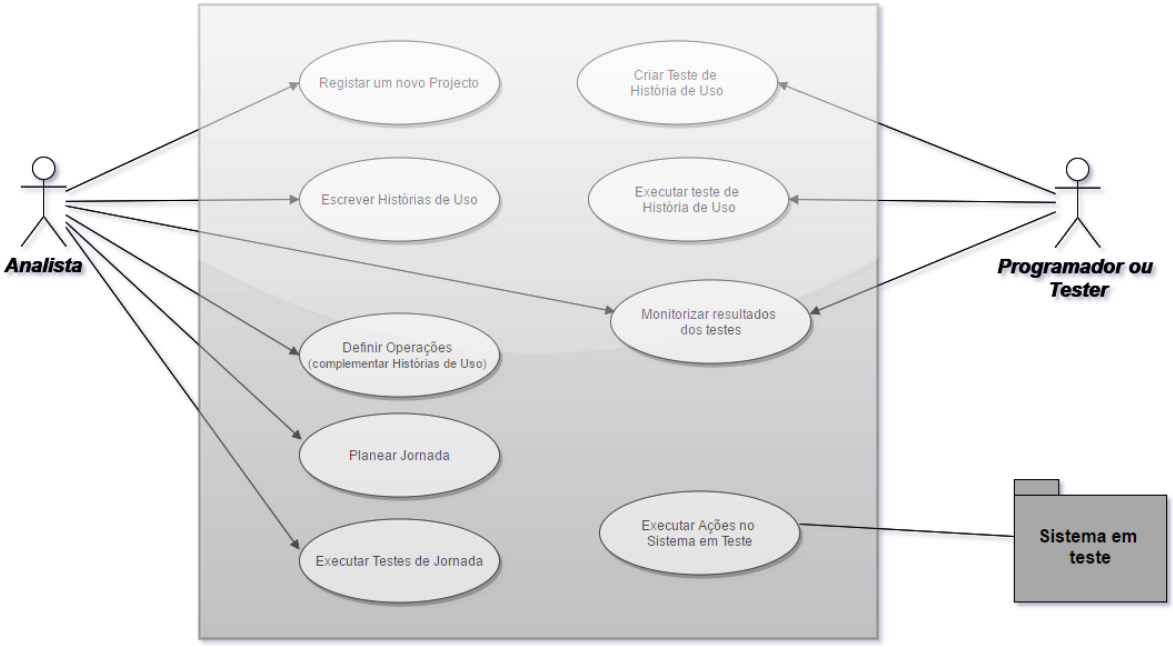


*Figura 8- Casos de uso da ferramenta de testes*

Os atores são o analista, o programador e o sistema em teste. As operações em uso são as necessárias para registar um novo projeto (a ferramenta suporta múltiplos projetos em teste), associar-lhe histórias de uso e planear e executar testes de jornada. Os testes de histórias de uso, precisam de ser escritas pelos programadores, podendo por eles próprios executar testes.

Sendo o objetivo da ferramenta a organização para permitir a execução de testes e de testes de regressão, o diagrama seguinte apresenta a sequência de atividades a realizar para a preparação e realização de testes.

A utilização geral da ferramenta está resumida no diagrama seguinte (Figura 9- Sequências de atividades - Testes de Jornada). A ferramenta de testes pode ser usada na definição de teste por história de uso e cada teste definido poderá ser usado. A ferramenta será também usada para a definição dos testes de jornada, que podem ser carregados manualmente ou gerado automaticamente pela aplicação, sendo para isso necessário incluir mais informação em cada história de uso.

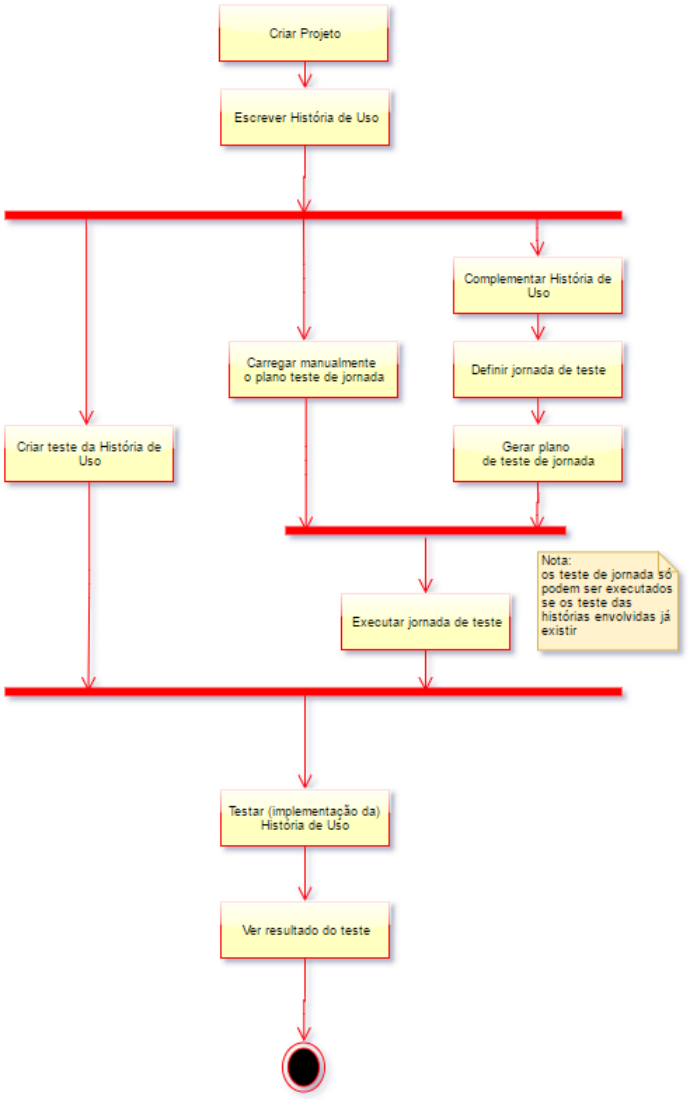


*Figura 9- Sequências de atividades - Testes de Jornada*

A execução de plano de teste de história de uso é autónoma, mas um teste de jornada precisa de todos os testes de história das histórias de uso envolvidas.

O ponto de partida passa pela criação do projeto e das histórias de uso, sendo para cada um possível passar à criação do teste dessa história, complementar a caracterização de estado relativa à história e usar o planeamento automático para o teste de jornal. Em alternativa ao processo automático é possível fazer a seu carregamento manualmente.

A imagem ( Figura 10- Arquitetura da ferramenta de testes) descreve a arquitetura geral do agente de teste (“Tester”). O Agente está construído de forma modular sendo o seu núcleo formado por três controladores de arquitetura BDI, que suportam os níveis Estratégico, Tático e Executivo do Agente. Cada controlador dispõe de uma representação do “seu” mundo (crenças) e de uma área na biblioteca de planos. A biblioteca de planos é persistida numa base de dados comum a todos os controladores. Também o jornal de atividade (“logging”) é comum a todos os controladores, enquanto que o planeador é apena usado pelo controlador executivo e o sistema em testes é apenas acedido pelo controlador executivo.

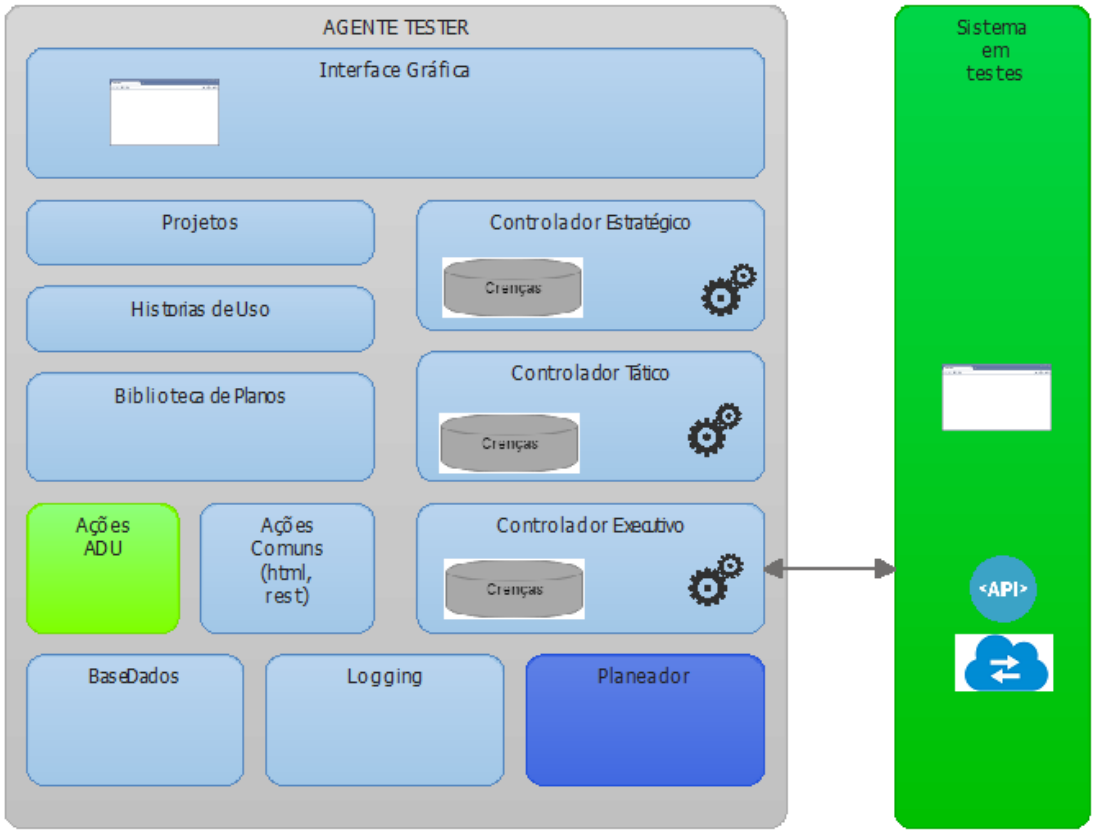


*Figura 10- Arquitetura da ferramenta de testes*

Cada controlador dispõe dos seus planos, que estão guardados na sua área da biblioteca. Os planos são compostos de ações comuns e por ações definidas pelo utilizador (“ADU”) especificas para o sistema em testes. Os planos estão associados a projetos e alguns também a histórias de uso sendo a gestão dos projetos feita no módulo projetos e as histórias de uso no módulo de histórias de uso. A gestão de projetos permite que o mesmo agente possa ser usado em testes de diferentes projetos, com planos definidos e separados para cada um deles. Cada história de uso pertence a um único projeto sendo a sua gestão feita no módulo histórias de uso, este módulo poderá eventualmente ser externo e existir numa outra ferramenta de gestão de requisitos. A implementação poderá ser alterada para permitir que toda a informação necessária venha de um sistema externo ou que venha parcialmente e que seja complementada nesta ferramenta de testes. Uma aplicação de uso comum na gestão de requisitos e de projetos ágeis é o Jira, onde será fácil complementar a descrição das histórias para incluir a informação necessária ao planeamento automático, podendo o seu acesso ser feito via serviços rest.

O planeador usa a linguagem prolog e corre num serviço à parte que expõe serviços rest.

A interface de utilizador permite definir os projetos, histórias de uso, carregar planos manuais, consultar histórico de testes e programar execuções. A interface de utilização tem um acoplamento fraco com o agente e a comunicação entre ambos é feita de forma assíncrona via base de dados, permitindo que a mesma consola (interface de utilizador) possa vir a controlar um sistema multiagente com partilha de uma única base de dados.

A interface com o sistema em testes é feita com base num conjunto de funções gerais integradas no próprio agente, complementadas com funções específicas para o sistema em teste. Os testes são escritos sob a forma de planos que, a baixo nível, invocam as funcionalidades disponibilizadas pela camada de interface com o sistema em teste e a alto nível os planos invocam os planos que usam os testes.

A interface de utilizador expõe as funcionalidades necessárias para a criação de um novo projeto, para a criação e manutenção das histórias de uso e seu enriquecimento, definição manual dos planos de teste de jornada e de história, geração automática de planos, escalonamento de execuções de testes, execução de testes e análise de resultados.

Esta interface interage com a camada “Controlador Estratégico” que por sua vez coordena a execução do “Controlador Tático”.

Os controladores usados no agente derivam todos de um controlador BDI (Figura 11 - Controladores, diagrama de classes), cada uma base de crenças e biblioteca de planos própria. O código executado é igual para todos e as diferenças de funcionamento resultam dos planos e das ações que executam. As ações sim são diferentes em cada controlador e apenas o controlador executivo interage com o sistema em teste.

.

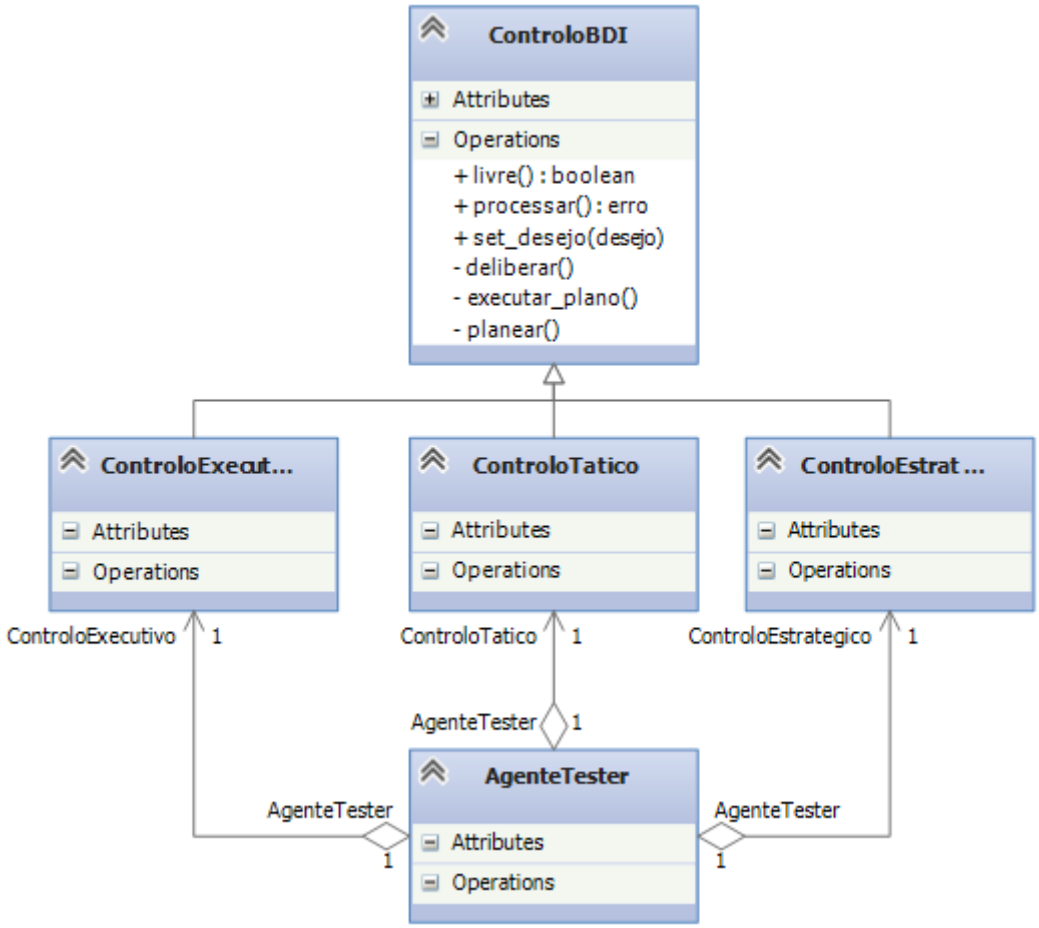


*Figura 11 - Controladores, diagrama de classes*

Os métodos expostos pelo agente servem para colocar desejos (set_desejo), para ativar o processamento (processar) e para verificar se o controlador está ativo ou em espera de novos desejos para realização (livre).

Internamente o controlador dispõe dos métodos necessários para a deliberação (deliberar), para planear (planear) e para executar cada passo de um plano (executar_plano). Na sequência do processamento atualiza a base de crenças (evocando o método da classe que implementa a base de crenças) e usa a biblioteca de planos para encontrar e selecionar as intenções para a concretização de um desejo.

A utilização do sistema desde a definição do projeto à execução de testes de jornada, envolve as seguintes fase de comunicação entre os diferentes componentes que compõem a ferramenta de testes e o sistema em teste:

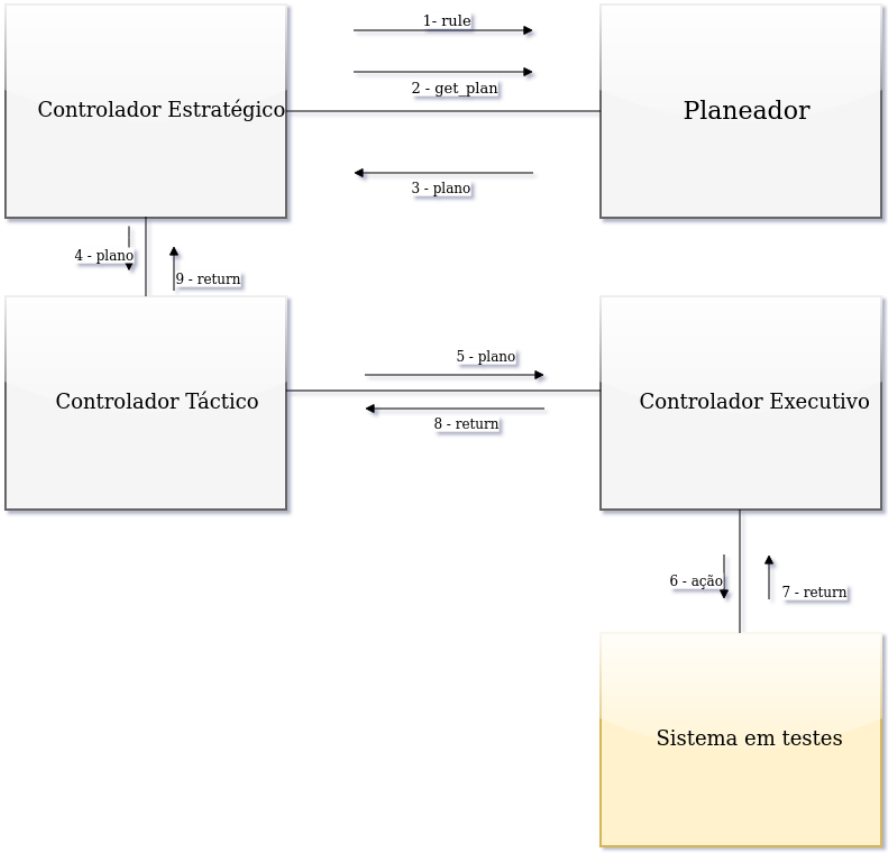


*Figura 12- Comunicação entre os componentes da ferramenta de testes*

Registada uma jornada, materializada pela definição ao nível de um novo estado inicial e estado objetivo e requerido o planeamento automático, o controlador estratégico passa ao planeador todos os operadores do projeto (1- rule) e no final as condições para a elaboração do plano (2 – get_plano) e obtêm como resposta um plano (3 – plano) se existir ou uma indicação de que não existência deste.

Com um novo plano, que é uma sequência de testes de histórias de uso, com parâmetros definidos, o controlador estratégico passa-o ao controlador tático (4 - plano). O controlador tático, para cada uma das ações do plano recebido, evocará a execução de um plano do controlador executivo (5 – plano) que corresponde a um teste de história de uso.

O controlador executivo, com base no plano requerido irá executar cada uma das suas ações interagindo com o sistema em testes (6 – ação), recolhendo informação do sistema em testes (7 – return), esta informação são as perceções com as quais irá alimentar a sua base de crenças e usar no processo de deliberação sobre a continuidade do plano ou a resolução de problemas,

O controlador executivo quando terminar a execução do plano de teste de história, porque correu todas as suas ações ou porque encontrou um erro que não consegue resolver, informará o controlador tático do resultado (8 – return). O controlador tático quando terminar a execução de todas as ações dos planos em curso, ou quando encontrar um erro que não possa resolver, informará o controlador estratégico do resultado (9 – return).

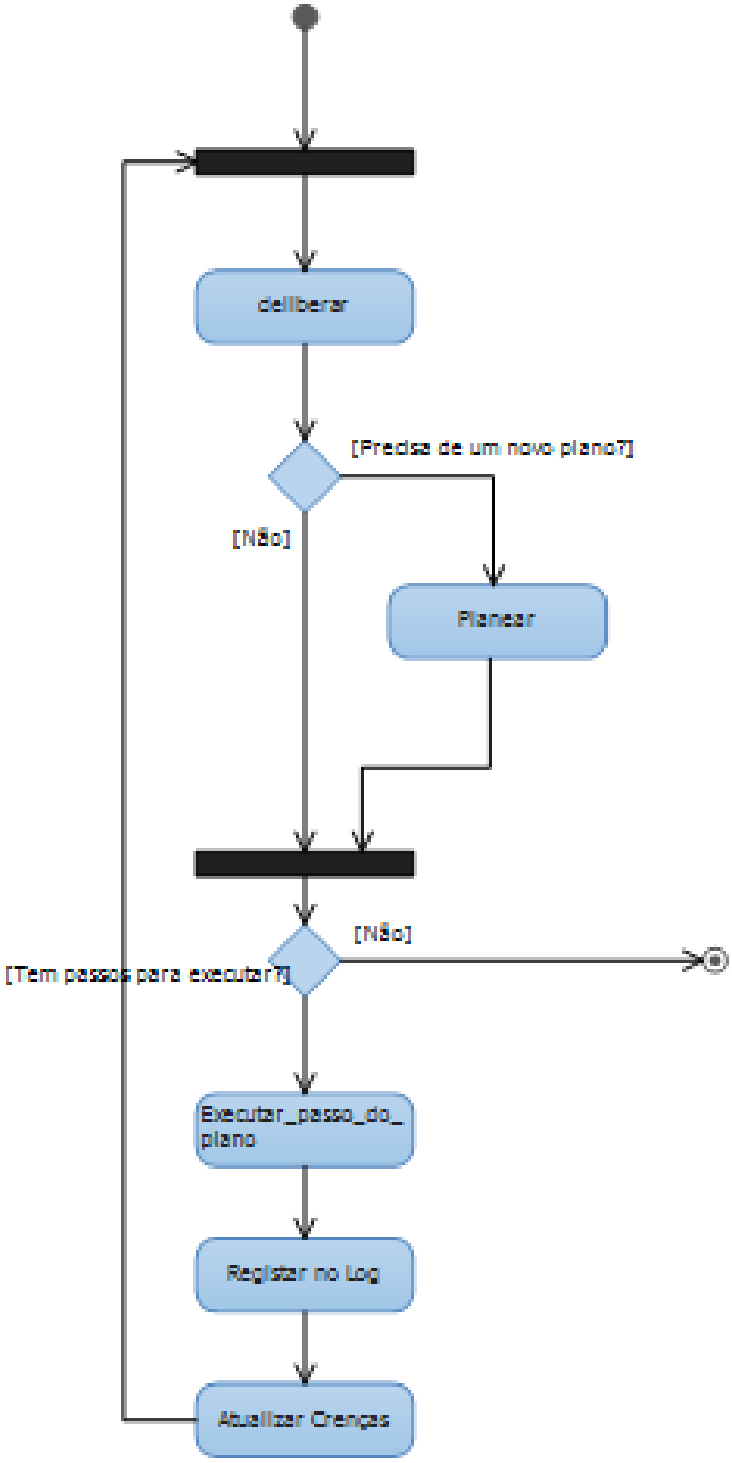


*Figura 13- Controlador - sequência de atividade*

O processamento de cada controlador em resposta á evocação do método "processar" é o indicado na "Figura 13- Controlador - sequência de atividade". O controlador opera num ciclo:

- Começa por deliberar, isto é, por avaliar se tem um plano ou desejos para concretizar;
- Se tiver desejos, mas ainda não tiver uma intenção, refletida num plano de ações, então irá planear;
- Planear é procurar na biblioteca de planos os que respondem ao primeiro desejo da lista e a seguir selecionar o mais adequado;
- A seguir verifica se existe um plano e se ainda tem passos para executar, se não tiver então é porque já terminou a sua atividade e informa o resultado do processo, passando uma perceção para o controlador de nível superior;
- Se tinha um passo (ação) para executar, então executa-o e regista o resultado no jornal do sistema (log);
- Atualiza as crenças;
- E volta a iterar.

A execução e cada passo difere no caso de a ação ser interna ou externa. Uma ação interna atua ao nível do controlo do controlador, por exemplo alterando o seu nível de tolerância a erros, alterando o registo no jornal, etc.

## 4.1 Teste de história de uso

O diagrama seguinte (Figura 14- Sequência de atividades de um Teste de História (de uso)) descreve a sequência de atividades para a criação e execução de um teste de história de uso. O plano de teste, criado pelo programador, está ligado a uma história e esta a um projeto. Um projeto tem várias histórias e as histórias podem ter vários testes, mas uma história só pertence a um projeto e um teste só pertence a uma história.

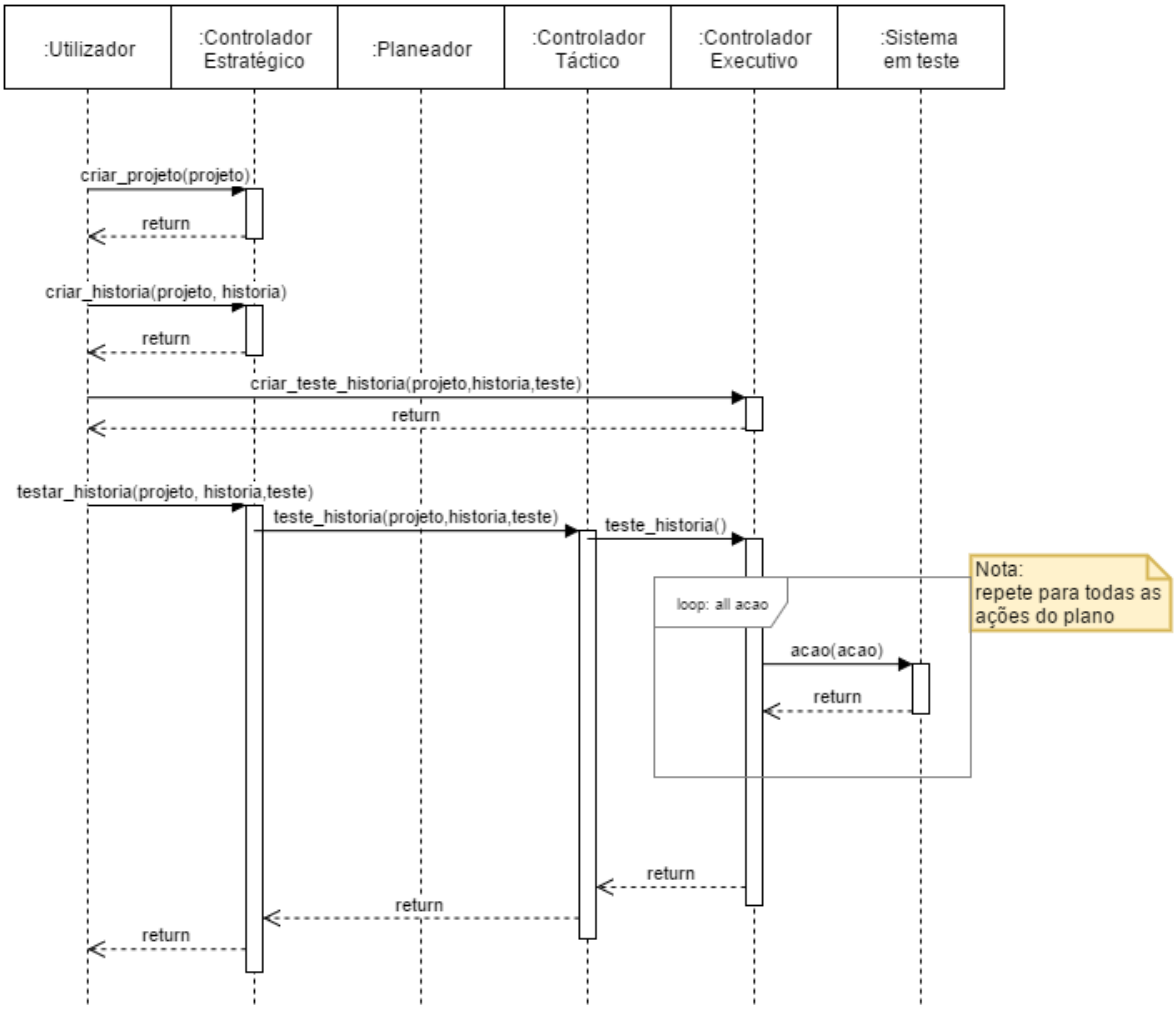


*Figura 14- Sequência de atividades de um Teste de História (de uso)*

## 4.2 Geração automática de teste de jornada

O teste de jornada (Figura 15- Planeamento automático de Jornada (sequência de atividades)) é um plano composto por chamadas a um ou mais testes de história de uso. O plano do teste de jornada pode ser carregado manualmente pelo analista ou programador do projeto e uma vez criado pode ser executado. Um plano de jornada está sempre ligado a um projeto e é a base dos testes de regressão do sistema (uma vez definido, o plano pode ser executado repetidamente pela ferramenta).

Uma alternativa ao carregamento manual do plano de teste de jornada é a sua geração automática pela ferramenta de testes, para isso é necessário complementar cada uma das histórias de uso do projeto, com informação em modelo STRIPS por forma a construir um operador. O diagrama seguinte apresenta a sequência de atividades necessárias ao planeamento automático de uma jornada, para um projeto e histórias de uso anteriormente carregados no sistema. Para a geração do plano não é necessário que o sistema tenha sido implementado ou que existam testes de história de uso já carregados, esta geração é meramente abstrata, gera a ligação lógica entre as histórias de uso e eventualmente poderá depois servir para coordenar uma sequência de testes de histórias de uso (estas sim a interagir com o sistema em teste).

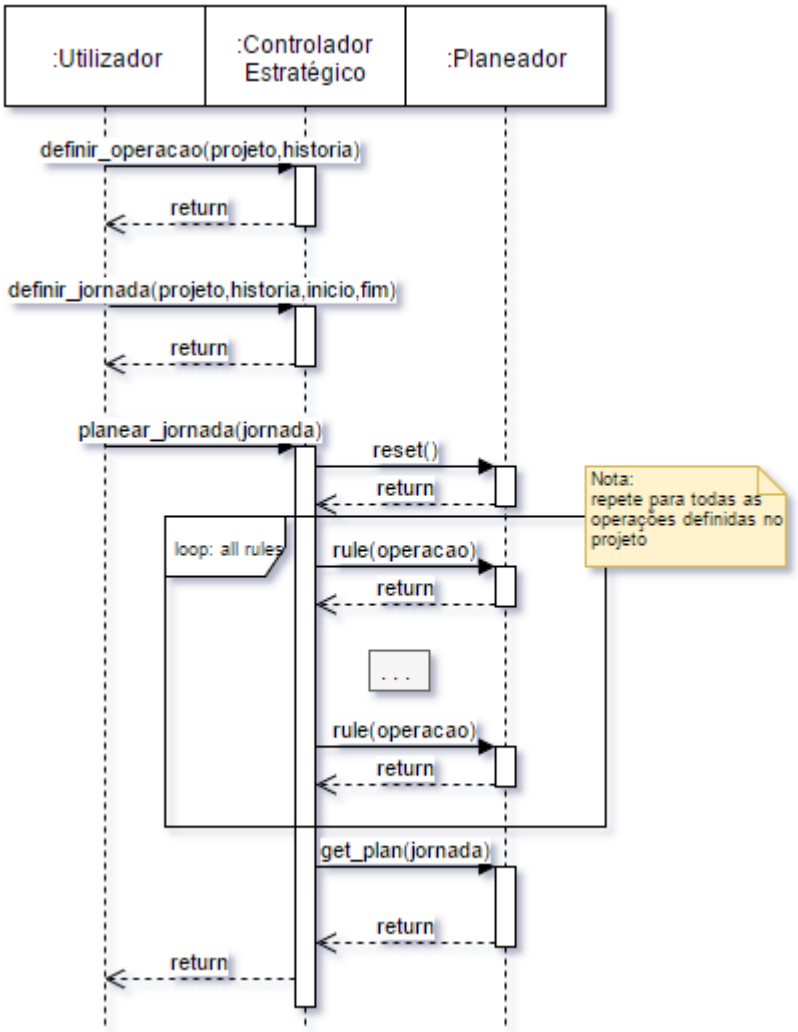


*Figura 15- Planeamento automático de Jornada (sequência de atividades)*

Uma jornada tem um ponto de partida do sistema, o estado inicial e um estado final ou objetivo. Com base num algoritmo de planeamento será produzido um plano de evocação das operações disponíveis para levar o sistema do estado inicial ao estado objetivo, o que neste caso representa o teste de jornada.

Para iniciar o planeamento é primeiro limpa a informação que possa existir no planeador, são carregadas as operações do projeto e a seguir passados os parâmetros que definem a jornada, ou seja o estado inicial e o estado objetivo. Os operadores e a descrição dos estados usam a linguagem STRIPS, na formulação proposta, mas seria possível usar

outros algoritmos de planeamento, embora nos ensaios realizados este tenha mostrado boa adequação ao problema em causa.

## 4.3 Teste de jornada

O teste de jornada é guiado pelo plano que está associado a uma jornada de um projeto e guia a aplicação dos testes de história de uso. Os testes de história de uso são também planos, que contêm ações de interação com o sistema em teste. Estas interações permitem avaliar a resposta do sistema em teste às entradas e estímulos que lhe foram dados e este é o alvo final dos testes, comparar as respostas obtidas do sistema com as respostas / comportamento previsto.

O diagrama seguinte ()) apresenta a sequência de atividades de um teste de jornada, em que o teste de história de uso é igual ao anteriormente apresentado, mas agora dentro de um contexto mais extenso o do teste de jornada.

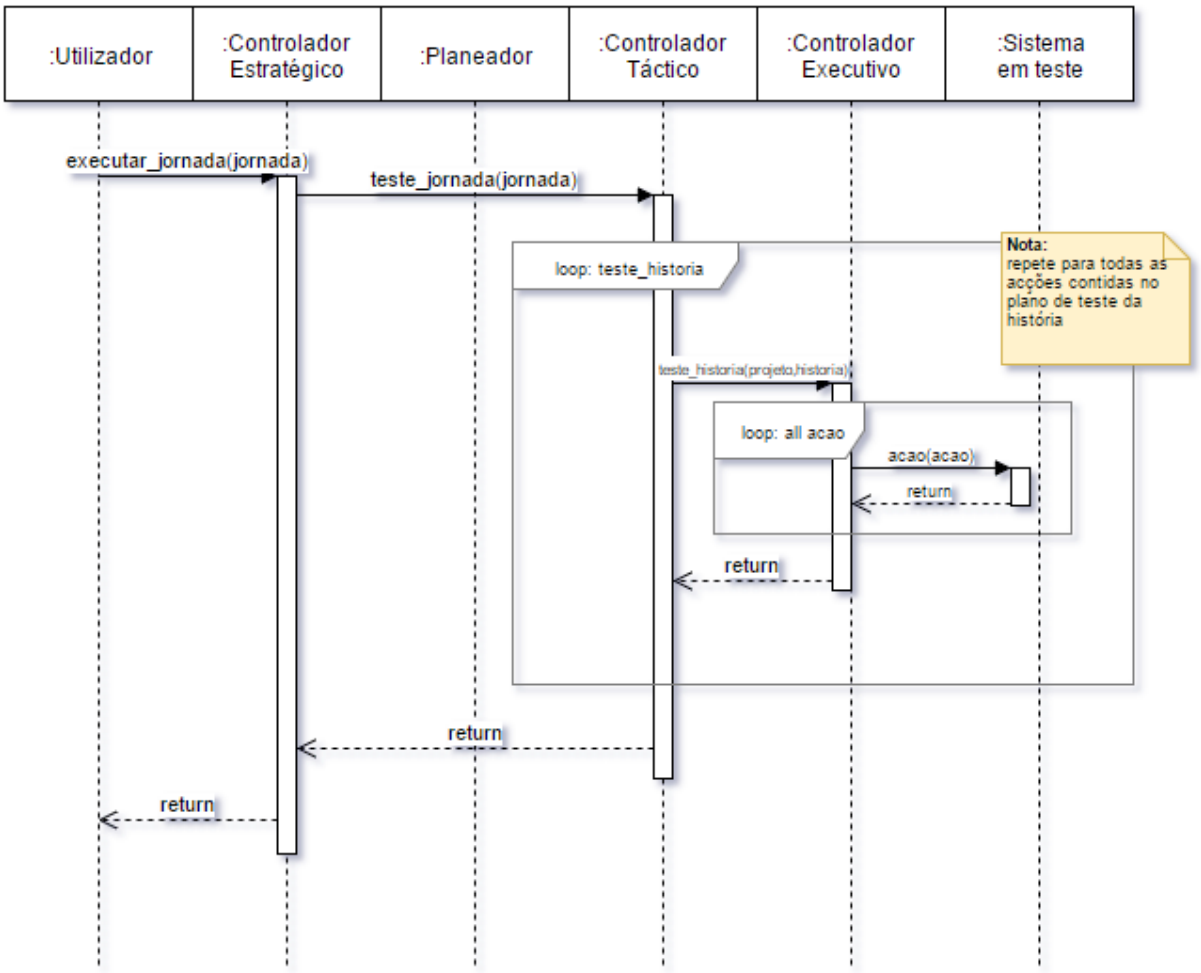


Figura 16- Teste de Jornada (sequência de atividades

)

## 4.4 Execução de teste de história de uso

O teste de história de uso é realizado pela execução de um plano, esse plano é formado por uma sequência de passos e cada passo pode ser de um de três tipos: um novo plano, uma ação de controlo ou uma ação que interage com o sistema em teste.

No caso de o passo ser uma referência para um novo plano (Figura 17 - Desdobramento de um Plano dentro de outro Plano), esse será executado no lugar do passo e a execução retomada, como é exemplificado na figura seguinte:

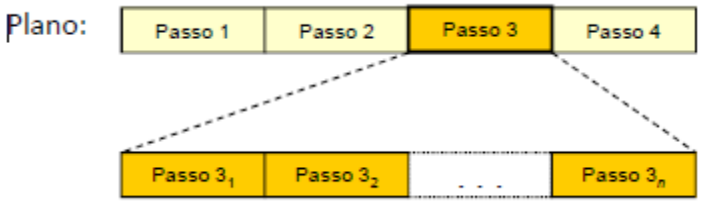


*Figura 17 - Desdobramento de um Plano dentro de outro Plano*

No caso de uma ação de controlo, corresponde a uma alteração do ambiente de execução do plano, por exemplo alterando o nível de tolerância, do formato de registo no jornal de atividade, criação de variáveis locais. Uma vez executado o passo, passa ou seguinte;

Finalmente se for uma ação com interação com o sistema em testes, o controlador executará a ação (normalmente uma chamada a um serviço, uma interface programática ou atuação no browser), comparará o resultado obtido com o esperado e colocará os valores lidos (ou parte) no contexto do plano em execução.

Da execução das duas últimas opções plano e ação no sistema em teste, pode resultar erro e em função do nível de tolerância, abortar a execução do plano, procurar soluções para o erro ou continuar o processamento caso essas soluções não existam.

O diagrama seguinte (Figura 18- Execução de um plano (diagrama de atividades)) descreve a execução de um plano pelo controlador executivo com as variações de tipo de passo, erros obtidos na aplicação do passo e o comportamento do agente em função do nível definido de tolerância.

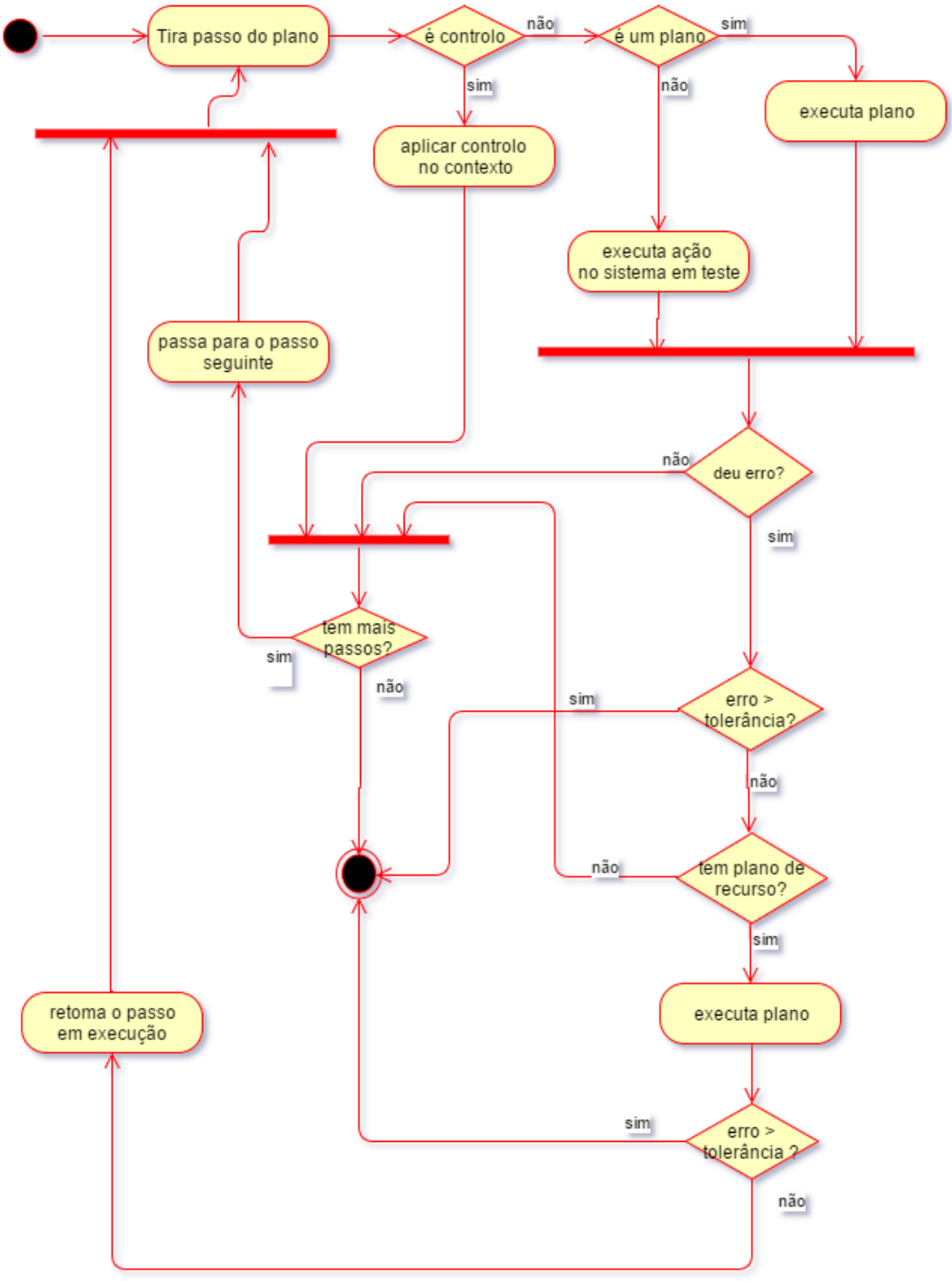


*Figura 18- Execução de um plano (diagrama de atividades)*

A tolerância é um conceito introduzido para a execução de testes no sistema em teste. Nem sempre a resposta do sistema em teste está em linha com a resposta esperada, mas isso poderá não ser razão para abortar toda a sequência de testes, tal como um humano faria existem situações nas quais podemos aplicar uma solução conhecida e retomar os testes, ou mesmo continuar a testar outras áreas e não abortar a sequência, que corre automaticamente durante a noite, aproveitando a disponibilidade do sistema. Um plano alternativo para lidar com um erro ou resultado inesperado poderá ser automaticamente aumentar o nível de registo no jornal para debug, por exemplo e reensaiar o teste, evitando assim desperdiçar uma sequência de atividade de teste do sistema. Quando o controlador entra num processo de recuperação de um erro, o modo interno muda para "correção", alterando ligeiramente e seu comportamento (como acontece num humano em função da sua "disposição") por exemplo impedindo que exista uma recuperação da recuperação e a entrada num ciclo infinito ("deadlock"). O retomar do modo normal acontece no final da recuperação, com o retomar do plano original. Internamente a entrada e saída destes modos é controlada por planos gerados automaticamente e que incluem uma ação para a saída e retorno ao modo normal.

## 4.5 Ações

As ações são o elemento menor de um plano. Elas podem ser atividade meramente interna ao controlador ou ter interação com o sistema em testes.

As ações derivam de uma classe base a Acão que tem um método abstrato o processar, ver: “Figura 19- Ações (diagrama de classes)”.

As ações podem ser de 2 tipos de base, as ações internas e as ações externas. As ações internas correm dentro do controlador sem qualquer interação com o exterior e existem para os 3 controladores e servem para alterar o seu estado ou promover atividade interna do agente. As ações externas representam ações com o exterior do controlador, que podem ser a interação com o sistema em teste ou com o controlador do nível seguinte,

Para o presente estudo foi considerado um conjunto de ações de interação com o sistema em teste, que são a evocação de serviços “rest” e a simulação da utilização de um browser por um utilizador. Podem ser incluídas mais ações desde que respeitem a interface esperada pelo controlador e que se traduz pela forma de evocação da ação e pelo seu retorno. Uma ação pode receber valores constantes ou referências para variáveis cujo valor só é definido em tempo de execução, pelo que a ação deverá saber lidar com essa situação, tal como está implementado na classe de base.

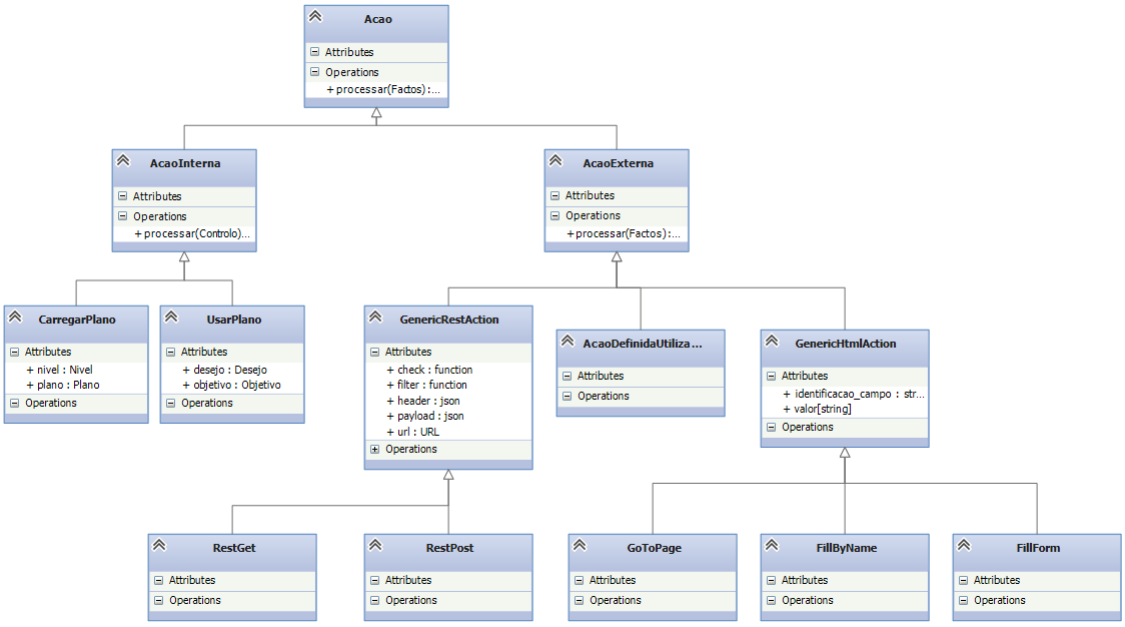


*Figura 19- Ações (diagrama de classes)*

As ações que interagem com o sistema em teste são (de momento) fundamentalmente de dois tipos, derivadas da classe GenericRestAction e GenericHtmlAction. Estas duas classes de base deixam alguns métodos abstratos por implementar que são implementados pelas subclasses.

No caso da classe GenericHtmlAction, todos os métodos de acesso aos dados de uma página e de manipulação do browser estão implementados na classe de base e as subclasses apenas associam os métodos _find() e _action() aos métodos da classe base que têm a funcionalidade pretendida, por exemplo para pesquisar por nome, por id, css, etc.

As classes implementadas são os nomes usados nos verbos dos planos (daí a importância de herdar da classe base em vez de usar parâmetros mais ou menos complicados para escolher o método a usar), como neste exemplo:

Para a construção e planos de teste de história foram criados um conjunto de comandos, que permitem atuar sobre o sistema em testes, simulando o uso de um browser pelo utilizador. Na lista de comandos o nome é composto por uma palavra que indica a ação a realizar e a forma de localizar na página html o elemento onde incidirá a ação. As ações possíveis são “click” para simular a seleção do campo, “fill” para indicar o seu preenchimento, “read” para ler o valor do elemento html e “in” para verificar se inclui o texto indicado em value. A forma de localizar um elemento na página html por ser por “css”, pelo nome da classe do elemento “classname”, o “id” do elemento ou mesmo com uma expressão “xparse” para localizar o elemento na estrutura xhtml da página.

Os parâmetros para a função são a indicação do elemento a encontrar (field), o valor a colocar (value) no caso de ser um preenchimento, que pode ser por acumulação ao valor existente ou por substituição se clear tiver o valor “True”.

A lista destes comandos é a seguinte:

- **ClickByCss**(field, value=None, clear=False, option=None)
- **ClickByClassName**(field, value=None, clear=False, option=None)
- **ClickById**(field, value=None, clear=False, option=None)

- **ClickByLinkText**(field, value=None, clear=False, option=None)
- **ClickByName**(field, value=None, clear=False, option=None)
- **ClickByPartialLinkText**(field, value=None, clear=False, option=None)
- **ClickByXPath**(field, value=None, clear=False, option=None)
- **FillByClassName**(field, value, clear=False, option=None)
- **FillById**(field, value, clear=False, option=None)
- **FillByLinkText**(field, value, clear=False, option=None)
- **FillByName**(field, value, clear=False, option=None)
- **FillByPartialLinkText**(field, value, clear=False, option=None)
- **FillByXPath**(field, value, clear=False, option=None)
- **FillForm(**prefix, values)
- **GoToPage(**base_url, page, title=’’)
- **InByClassName**(field, value=None, clear=False, option=None)
- **InById**(field, value=None, clear=False, option=None)
- **InByName**(field, value=None, clear=False, option=None)
- **InByXPath**(field, value=None, clear=False, option=None)
- **ReadByClassName**(field, value=None, clear=False, option=None)
- **ReadById**(field, value=None, clear=False, option=None)
- **ReadByLinkText**(field, value=None, clear=False, option=None)
- **ReadByName**(field, value=None, clear=False, option=None)
- **ReadByPartialLinkText**(field, value=None, clear=False, option=None)

- **ReadByXPath**(field, value=None, clear=False, option=None)

Para acesso aos serviços "rest" do sistema em teste estão disponibilizados 2 comandos, que recebem como parâmetros o endereço do serviço "url", a carga a colocar no serviço "payload", o "header" a incluir opcionalmente no serviço (por exemplo a informação de segurança e de autenticação). O "check" é usado na verificação dos valores obtidos no acesso ao serviço e será uma função que permita validar a estrutura json e os seus valores. Os valores a considerar para colocar no contexto (e para uso noutras operações) são as resultantes da aplicação da função "filter" à estrutura json recebida no acesso ao serviço. A lista dos comandos de acesso aos serviços "rest" e a seguinte:

- **RestGet(**url, payload, header=None, check=None, filter=None)
- **RestPost(**url, payload, header=None, check=None, filter=None)

Ações de rest são um pouco mais complexas pois permitem utilizar funções de para validar se o resultado é o esperado, pois o serviço poderá devolver uma grande quantidade de valores e apenas alguns serem relevantes para o resultado do teste. De igual forma também podem ser retirados do resultado alguns dados para manter no contexto de execução do plano. Alguns exemplos (Figura 20- Exemplo de uso de serviço REST para o método html GET) de uso com base em funções auxiliares criadas para o efeito são:

- **match_all** procura na estrutura os atributos referidos, neste caso um elemento denominado data e aplica a expressão regular que lhe está associada, neste caso se o conteúdo é uma das palavras "primeiro" ou "segundo".
- **filter** retira os valores que respondam à condição apresentada.

```
RestGet( url,  {'id':'all'},
        check=match_all( {'data': '(primeiro)|(segundo)'} ),
        filter= basic_filter('segundo')
```

Figura 20- Exemplo de uso de serviço REST para o método html GET

Os valores usam sempre uma representação json, para alinhar com a prática comum com serviços rest.

Em todos os verbos, os valores passados podem ser constantes, referência a uma função ou método de um objeto, nestes últimos dois casos será feita uma evocação em "runtime" para obter o valor dinâmico em cada execução.

## 4.6 Plano

Um plano é composto por ações, ver "Figura 21- Planos (diagrama de classes)", que podem ser internas, externas ou definidas pelo utilizador ("ADU") para adaptar o agente a um sistema em teste específico. O plano tem como atributos:

- desejos: a lista de desejos a que responde
- objetivo: a que se destina ou contexto que o plano serve, por exemplo pode ser usado para recuperar de um erro (desejo) para fazer login, o seu objetivo
- parâmetros: de configuração do plano, funcionam como os argumentos de uma função

- encaixe: é uma função que é usada para avaliar da adequação do plano ao desejo e contexto;
- passos: sequência de ações que compõem o plano.

O plano é ativado com o método processar, que devolve a lista de passos que compõem o plano. O método poderia ter outro nome, mas este torna-o semelhante a uma ação, afinal é uma macro ação composta por várias ações.

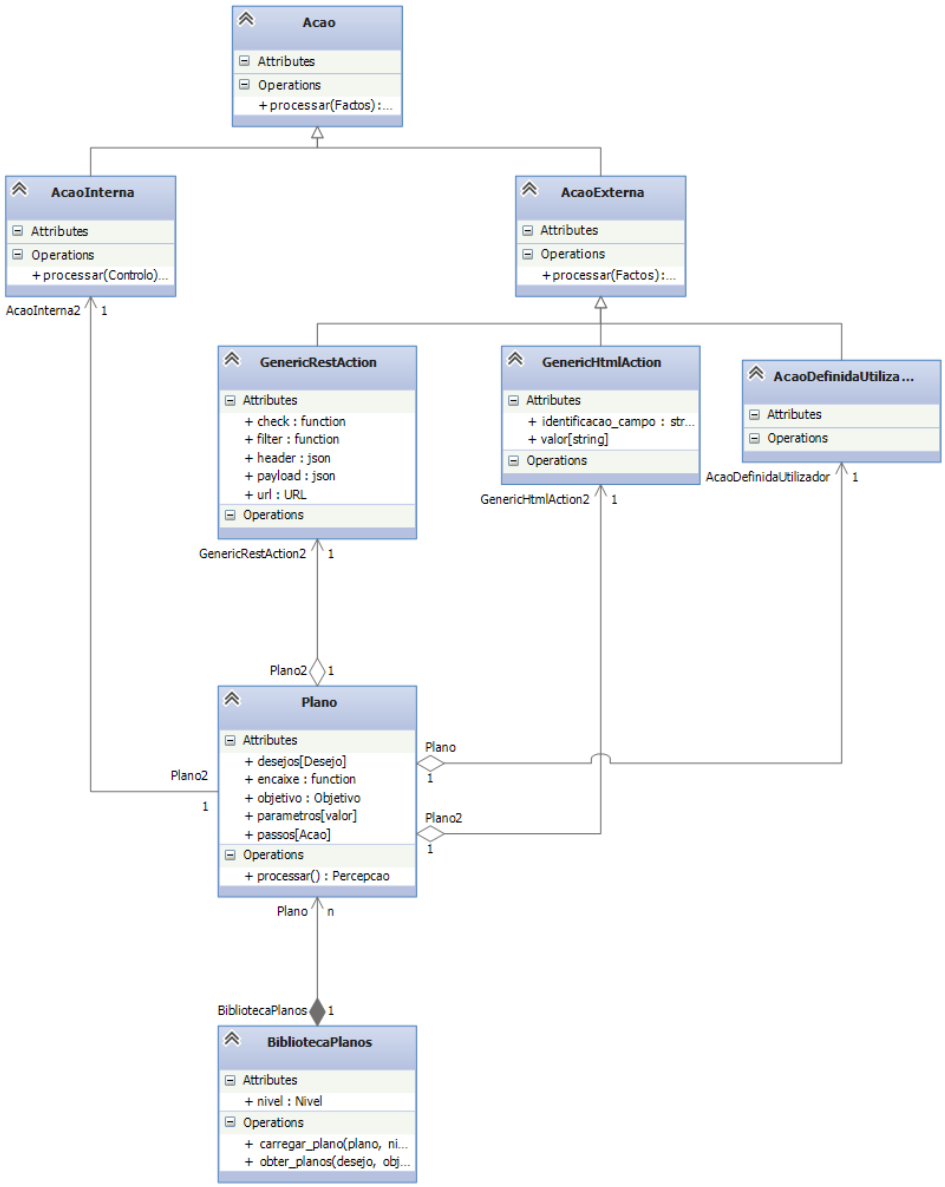


*Figura 21- Planos (diagrama de classes)*

Os planos são guardados na biblioteca de planos, que responde aos métodos de carregar_planos e obter_planos. Carregar planos serve para incluir novos planos na biblioteca do controlador, sendo que esses planos podem ser de carregamento manual ou ter sido gerados automaticamente pelo planeador. Obter_planos, vai percorrer a base de dados de planos, para obter todos os que se encaixam no desejo e objetivo indicados, usando para isso a função de encaixe definida para cada plano.

Todos os planos têm a mesma estrutura, a sua diferença está no nível de controlador onde são usados e nas ações que os compõem.

A seguir são descritos os diferentes tipos de planos suportados pelo agente.

## 4.7 Plano teste história de uso

Um plano local, como referido é conjunto de ações a executar no teste do SUT, que é carregado e mantido no repositório de planos, contendo também um nome, o desejo que realiza, a lista de adequação e o nível de tolerância. O nível de tolerância pode variar de plano para plano em função dos seus objetivos, por exemplo num caso particular ter um serviço a responder poderá bastar, enquanto noutros caso será também necessário que devolva um resultado bem definido e esperado. Uma forma de controlar estes dois casos é com base no nível de tolerância.

O plano é composto por uma lista de ações, cada ação é executada em sequência e é analisado o seu resultado. Algumas das ações não implicam atividade com o sistema em testes, são o caso das referências a outros planos e a ações de controlo do próprio controlador.

No inicio da execução o controlador carrega o contexto recebido, como parâmetros para o plano. Por exemplo um plano para criação de um cliente, poderá ter como parâmetro uma etiqueta “antonio”, que será depois unificado com os dados lidos do sistema em teste. No exemplo referido anteriormente, era criado um cesto de compras (“basket”) para um cliente, mas não era para um cliente qualquer era para um cliente referido por um nome abstrato ao nível do plano de teste de jornada, mas que corresponde

a uma criação efetiva de cliente no sistema em testes, com os campos que este precisa para esse fim.

Na execução em conjunto com outros planos, as variáveis são resultado de operações anteriores e o login já poderá ter sido executado (ou não, daí poder ser melhor tratado como um plano de resposta a um erro, o da necessidade de autenticação no sistema).

O responsável pela execução do teste a uma história é o controlador executivo, que recebe as suas ordens (indicações) a partir do controlador tático, ou seja da camada seguinte de controlo do Agente.

# 5 Demonstrador

O demonstrador da ferramenta de teste foi implementado em python versão 3.x, com um planeador escrito em prolog e com uma interface de utilizador implementada com web2py também framework implementada sob python para o desenvolvimento para a web.

## 5.1 Instalação

### 5.1.1 Tester: portal

A componente de Portal do Tester foi desenvolvida em cima da framework Web2py sob a plataforma windows.

No caso do web2py para Windows poderá ser usada a versão incluída no ficheiro Tester.zip (recomendado) onde estão todas as aplicações necessárias ou fazer o download da versão para Windows em http://www.web2py.com/init/default/download podendo ser usada a versão para Windows "For Normal Users" (não é necessário ser a versão source).

No caso do download e instalação isolada do Web2py, será necessário copiar da sub-directoria "applications" os folders "Tester" e "SalesPortal" da versão fornecida para a posição equivalente do software instalado.

### 5.1.2 Tester: agente de testes

O Agente de Testes foi desenvolvido em python3 e corre de forma autónoma à componente portal do Tester. Para a sua instalação é necessário que esteja instalada uma versão do python com a extensão para **selenium** (framework para simulação de um web browser e que é usada nos testes de interação com uma aplicação web).

No exemplo construído foi usada a simulação com base num browser FireFox, pelo que também ele precisa de estar disponível no ambiente onde corre o agente. É possível simular a utilização de um outro browser (ie, opera, chrome, etc.) sendo nesse caso

necessário ter o respetivo browser instalado e ainda fazer download de uma versão do engenho para o uso com o selenium (ver documentação do selenium).

O download do software python versão 3 para windows poderá ser feito em:

https://www.python.org/downloads/windows/

depois de instalada a versão de python a framework **selenium** poderá ser instalada com a seguinte linha de comando:

pip install --upgrade selenium

que usa o utilitário pip do python para fazer download e instalar a framework indicada.

### 5.1.3 Tester: planeador

A componente planeador usa o swi-prolog, cujo download e instalação poderá ser feito a partir do site:

http://www.swi-prolog.org/download/stable

### 5.1.4 Sistema em teste

O sistema em teste corresponde à aplicação “SalesPortal” instalada em web2py em conjunto com a componente portal do Tester e que já se deverá encontrar instalada.

### 5.1.5 Resumo de componentes e versões usadas

No demostrador foram usados os seguintes componentes:

- Python 3.3.5 com sqlite3 (nativo) e selenium-2.53.2
- Web2py 2.9.11 – stable
- swi-prolog version 6.6.6
- Google Chrome version 51.0.2704.84
- Mozzila Firefox

O ensaio foi realizado com Windows 10.

## 5.2 Arranque do sistema

Uma vez instalado todo o software, devem ser seguidos os seguintes passos para ativar o Tester:

- Colocar-se na diretoria web2py
- Arrancar com os serviços do portal executando o comando web2py
- Escolher uma password (para o administrador do portal) e fazer “start” do serviço
- No browser por norma será aberta uma página de acesso à aplicação “welcome” do web2py no endereço http://127.0.0.1:8000/Welcome
- A password usada no arranque do portal será necessária para investigar algum ticket aberto durante o uso do portal
- Mudar para a diretoria: applications/Tester/modules/planner
- Aí executar clicar sobre swi-server.pl que irá carregar todo o código para a pesquisa em espaços de estados e o necessário para que o swi-server aceite pedidos http, para isso será ainda necessário lançar o serviço num porto (o escolhido para esta implementação foi o 9000 enquanto o web2py corre no 8000)

- Quando terminar o carregamento do código, lançar o serviço com: server(9000).
- Aceda a http://127.0.0.1:8000/Tester deverá obter uma página descritiva da aplicação. Deverá criar um utilizador (sign up) ou ligar-se ao sistema (log in) caso já tenha criado o utilizador.

Está concluída a instalação do sistema Tester.

NOTA: tickets web2py, no sentido de não divulgar informação interna sobre o código, cada vez que ocorre um erro o web2py cria um ticket de erro que pode ser consultado pelo "administrador" do portal.

O sistema exemplo em testes corre também em web2py e corresponde à aplicação SalesPortal a correr no mesmo web2py (poderá ser facilmente colocado num web2py a correr num outro sistema, mas será necessário depois configurar os endereços de acesso).

O Sistema foi instalado e disponível com o arranque do web2py. Pode verificar o seu bom funcionamento acedendo a http://127.0.0.1:8000/SalesPortal

# 6 Conclusões

O presente trabalho mostra que um agente artificial inteligente pode ser usado na automatização da execução de testes de regressão, pela reutilização dos testes manualmente escritos para cada uma das histórias de uso. O agente proposto tem também capacidade de reagir a não conformidades do sistema em teste, aplicando sequências de atividades para a correção das anomalias ou para a recolha de mais informação de diagnóstico. Ao nível mais abstrato da especificação dos requisitos, a incorporação de caraterização complementar de cada história de uso, permite elaborar planos de verificação da coerência dos requisitos e gerar automaticamente planos de teste de regressão que usam os testes de história de uso criados manualmente pelos programadores. Dos modelos de planeamento ensaiados, o STRIPS parece apresentar uma resposta capaz para o planeamento dos testes de regressão, quer em tempo, quer na simplicidade da linguagem usada na escrita dos operadores e na caraterização do estado objetivo ou final. A possibilidade de usar planos anteriores como operadores de planos seguintes, permite atacar o problema do desempenho descritos por modelos semelhantes com recurso a planeadores automáticos.

A linguagem usada para complementar a descrição das histórias de uso parece ser de uso simples e completa para uso no problema proposto, contudo será necessária uma utilização mais generalizada e diversificada em termos de problemas e equipas para poder aferir estas qualidades, para já apenas aparentes.

De igual forma a linguagem usada para definir os planos de testes de história de uso, parece ser bastante versátil e é fácil a inclusão de novos verbos usar novas interfaces com o sistema em teste.

A facilidade dessas linguagens tem a contrapartida da validação ocorrer apenas em execução (na geração e plano ou na execução do teste) e essa é a principal limitação do demonstrador criado.

O sistema em teste pode ser mais que uma aplicação e poderá mesmo ser um conjunto de subsistemas, uma vez que os testes correm à parte e usam as interfaces expostas para atuar no sistema. No exemplo do “portal de vendas”, a expedição da encomenda e os

registos na contabilidade correm em subsistemas diferentes, mas que poderiam ser considerados no conjunto dos testes.

O registo sistemático e coerente de todos os resultados das ações de teste, cruzado com plano que coordenou as ações, permitirá a mineração dos dados para:

- Classificar a robustez de áreas de código, com base nos seus tempos de resposta, volume de ocorrência de erros com a implementação de nova funcionalidade ou outras situações;
- Produzir mapas de cobertura de testes e otimização de recursos para aumentar a cobertura, incidência e eficácia dos testes;
- Produzir planos para apoio ao diagnóstico automático de ocorrências, erros dos testes;
- Gerar planos com dados limite ou aleatórios para testar o sistema, por exemplo usar datas muito no passado, no futuro ou com valores incorretos para avaliar e descobrir o comportamento do sistema em teste.

Partindo da base construída, a área mais interessante para trabalho futuro será o uso de sistemas multiagente para, de forma auto-coordenada, realizarem testes de carga do sistema em teste.

Outra área de potencial interesse era a de complementar os testes de regressão descritos, com testes derivados dos testes de histórias de uso, dedicados à exploração das condições limite, por exemplo dimensões de campos e valores fora do domínio, bem como usar testes comuns de segurança “sql injection” e outros. Esta funcionalidade supõe algum tipo de aprendizagem do agente com base nos testes de história escritos manualmente e depois derivados para o teste das fronteiras.

O crescente desenvolvimento de aplicações para dispositivos móveis coloca novos desafios e cria novas necessidades de testes que cruzem o uso das aplicações com gestos

e uso de funcionalidades dos dispositivos móveis que podem criar situações de erro, pode este ser outro ponto de evolução do sistema apresentado.

# Bibliografia

Afaf Al-Neaimi (2015), Research on Cloud Testing Based on Ontology and Multi-Agent Framework, (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 6 (3), 2015, 2746-2749

Bertolino (2007), Software Testing Research: Achievements, Challenges, Dreams, IEEE Future of Software Engineering(FOSE'07)

Gregory M. Kapfhammer (2003), Software Testing, Department of Computer Science Allegheny College

ISO/IEC/ IEEE 24765 (2010), Systems and software engineering – Vocabulary, Published by ISO in 2011

Krzysztof Apt, Principles of Constraint Programming (2003), Cambridge University Press

Meziane, Farid and Vadera, Sunil (2010), Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects, Information Science Reference (an imprint of IGI Global) 701 E. Chocolate Avenue Hershey PA 17033

Morgado, L. F. G. (2005), Integração de Emoção e Raciocínio em Agentes Inteligentes, Faculdade de Ciências da Universidade de Lisboa

Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger, PathCrawler (2005), Automatic Generation of Path Tests by Combining Static and Dynamic Analysis

Nikolai Kosmatov (2008), Constraint-Based Techniques for Software Testing, CEA LIST, Software Reliability Laboratory

Norvig P., Russell S. (2003), Artificial Intelligence, A Modern Approach. 2nd ed. Prentice Hall

Nguyen, C. D. et al. (2009), Evolutionary Testing of Autonomous Software Agents.

Olli-Pekka Puolitaival (2008), Adapting model-based testing to agile context, VTT Publications 694: http://www.vtt.fi/inf/pdf/publications/2008/P694.pdf

Poole, Mackworth and Goebel (1997), Computational Intelligence: A Logical Approach, Oxford University Press, 1997. Companion code: https://www.cs.ubc.ca/~poole/ci/ci_code.html

Sabih Jamal, Muhammad Aslam, Tauqir Ahmad (2015), Multi-agents based software testing as a service on cloud, Journal of Science International 27(2), pp. 2209-2215, June 2015.

Sam Newman (2015), Building Microservices: Testing, O'Reilly Media ISBN: 978-1-4919-5035-7 | ISBN 10:1-4919-5035-8

Schwaber, Ken and Sutherland, Jeff (2016), The Definitive Guide to Scrum: The Rules of the Game. ©2016 Scrum.Org and ScrumInc. Offered for license under the Attribution Share-Alike license of Creative Commons: http://www.scrumguides.org/docs/scrumguide/v2016/2016-Scrum-Guide-US.pdf

Sebastian Sardina, Lavindra de Silva, Lin Padgham (2006), Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach. AAMAS'06 – Hakodate, Hokkaido Japan. ACM 1-59593-303-4/06/0005

Shore, James (2010), The Art of Agile Development: Stories - http://www.jamesshore.com/Agile-Book/stories.html

SWEBOK v3.0 (2014), Guide to the Software Engineering Body of Knowledge, Published by IEE Computer Society, ISBN-10: 0-7695-5166-1

Vasilios S. Lazarou1, Spyridon K. Gardikiotis2 and Nicos Malevris (2008), Agent Systems in Software Engineering, Edited by Paula Fritzsche, ISBN 978-953-7619-03-9