TSuite: Uma ferramenta para seleção ótima de casos de teste manuais para regressão

Lorena do C. Caldas e Antônio M. da S. Pitangueira

Instituto Federal de Educação, Ciência e Tecnologia da Bahia, Salvador, Bahia, BRASIL

Resumo

Este trabalho aborda o desenvolvimento de uma ferramenta *web* chamada TSuite, criada para compor uma solução candidata à resolução do problema NP-Completo da seleção de casos de teste, através da avaliação de suas importâncias e custos unitários. A seleção apoia as condições de: tempo, prioridade e complexidade dos cenários de execução do teste, considerando o tempo disponível no projeto de teste, assim como o impacto da inserção do novo plano de execução de teste dentro deste projeto. Dessa forma, o TSuite automatiza a tarefa de escolher situações para teste de regressão, reduzindo seu custo de execução e ampliando a variedade de complexidade das situações, o que conduz ao aumento da cobertura de teste sobre o produto, no conjunto de unidades selecionadas. O TSuite utiliza a abordagem híbrida em sua estrutura o que envolve um método específico e outro de otimização da Engenharia de *Software* Baseada em Buscas.

Palavras Chave: Teste de *software*; seleção de casos; otimização; projeto de teste; teste de regressão.

Email address: lorecaldas, antoniomauricio@ifba.edu.br (Lorena do C. Caldas e Antônio M. da S. Pitangueira)

1. Introdução

Projetos de *software* compreendem atividades voltadas à construção das funcionalidades de um sistema. Dentre os processos que compõem o seu desenvolvimento está o teste de *software*.

As tarefas de teste correspondem aos esforços para controlar a qualidade do produto final entregue ao cliente, o que não implica dizer que atribui soluções ao conjunto [1]. Porém certificam que os critérios contratados estão inclusos no *software* e são acionados corretamente por um ator.

Na computação, a área de Engenharia de *Software* normatiza os aspectos do desenvolvimento de *software*, [8]. Processos e métodos são definidos como meio de manter um padrão de eficiência e qualidade do *software*. Muitas de suas vertentes utilizam técnicas de otimização para compor tais soluções.

A otimização auxilia na busca de melhores resultados, dado um problema que não possa ser facilmente solucionado diretamente por métodos convencionais, [18] apud [20].

O teste de *software* contém atividades que possuem um alto valor de execução, apesar de ter um maior valor agregado, se comparados. Para atestar qualidade de um produto o teste de *software* deve atuar sobre a linha de riscos e defeitos de um projeto de sistema [1]. Essa atividade permeia todas as fases de desenvolvimento e pode despender períodos elevados de operação entre planejamento, execução e entrega. A mínima tentativa de reduzir o tempo de abordagem e alcance dos problemas procurados na ferramenta conduz à minimização da receita de teste e consequente benefício ao projeto de *software*.

Para que o *software* possa ser devidamente testado é necessário observar as condições de operação dele além das ações que poderão ser executadas. Isso para que seja possível analisar a maior parte das funcionalidades envolvidas em seu contexto. A cada nova rodada de execução dos testes sobre a ferramenta os objetivos e cenários podem variar, fazendo-se necessário encontrar e criar novas situações de teste. Essa tarefa é demorada porque é necessário percorrer todas as funções do sistema de forma a encontrar o maior número de casos que se encaixem no objetivo atual da operação [2].

O TSuite foi criado com intuito de diminuir os custos da execução da tarefa de seleção de casos de teste considerando cenários de operações que contenham recursos escassos disponíveis e grande volume de dados de teste manuais. Antes de sua criação, seria necessário escolher cada uma das situações manualmente. Tarefa essa despendiosa, por conta da quantidade de tempo escalada. Essa redução é alcançada porque o TSuite realiza essa função automaticamente.

Por ser uma ferramenta de suporte à tomada de decisão, ela é indicada aos executores de teste que necessitem planejar como testar novamente o *software*, reduzindo o projeto a um conjunto prioritário de unidades de execução de teste. Sua utilização é adequada aos projetos de teste cadastrados na ferramenta de gerência: TestLink. O TestLink é uma ferramenta gratuita e *open source*, mantida pela comunidade de *software* livre Teamst, que oferece os serviços de cadastro, manutenção e suporte à execução de projetos de teste, [6].

Assim, ao usar o TSuite o executor poderá listar as situações de teste mais próximas às condições atuais dos recursos disponíveis ao projeto. Ao informar um arquivo contendo o projeto de teste e seus parâmetros obrigatórios, a ferramenta gera um conjunto reduzido das unidades de teste contendo somente as situações prioritárias.

Testes foram realizados para validar os benefícios que o TSuite trás e comprovou-se que ele reduz o tempo de execução da tarefa de seleção manual em aproximadamente 50%. Também, provou-se que a utilização do TSuite independe do conhecimento total do sistema ou experiência do responsável pela execução da seleção dos casos do teste. Além disso, o TSuite aumenta a quantidade de casos selecionados em até 6,9% porque maximiza a utilização dos recursos disponíveis, por conta da justiça atribuída aos casos de complexidade baixa, com a aplicação do método NSGAII na seleção complementar dos casos. No conjunto das unidades de teste selecionadas, aquelas que possuam complexidade baixa tem sua quantidade ampliada em 16,7%, em relação ao método de solução do problema da Mochila e 55,5%, se o método Aleatório for utilizado, em comparação com o método NSGAII.

Neste trabalho, a seção Teste de *Software* trás definições do processo e aborda características que o envolve. A seção Métodos e Algoritmos descreve a estrutura interna do TSuite e explica o escopo de suas principais funções. Já a seção Testes e Resultados contém as hipóteses e resultados da validação da utilidade da ferramenta. Os tópicos descritos anteriormente estão dispostos respectivamente nesta ordem e são encontrados a seguir.

2. Teste de *Software*

De acordo com [2], teste de *software* é um processo que estrutura passos, após planejá-los, com a intenção de identificar defeitos e falhas no produto. Dito isto, testar é uma atividade onde um sistema ou um componente é verificado múltiplas vezes sob condições objetivas, de forma a gerar resultados específicos. O critério de término do teste é analisado com base no resultado de sucesso ou falha da execução dos cenários de teste.

O teste de *software* vem sendo utilizado com freqüência nos projetos de *software* como uma forma de elevar o controle sobre a qualidade do produto disponibilizado

ao cliente, [7]. As atividades de teste de *software* costumam consumir grande custo do projeto. Suas execuções demandam tempo e maturidade do processo de desenvolvimento. Na maioria das empresas que produzem *software*, ela faz parte do processo de garantia de qualidade [1].

Segundo [2], a etapa de testes, dentro do esquema citado acima, pode chegar a comprometer 40% dos gastos de todo o processo de desenvolvimento do *software*. Esse valor é atribuído ao tempo gasto na execução da fase de teste, necessário à análise e maturação do sistema, decorrente da correção dos *bugs*. Qualquer forma de redução de escopo dessa tarefa, sem gerar prejuízos ao processo de qualidade estabelecido, é válida.

Observando o cenário exposto acima, o TSuite é uma ferramenta que foi criada para coletar as informações de um projeto e as selecionar automaticamente. Isso facilita a tomada de decisão, pois o gestor passa a ter apoio, porque ele conta com os dados do projeto dispostos em um plano fiel à realidade atual do projeto, possibilitando uma análise embasada e direta.

De acordo com essa proposta, o TSuite atua na redução de esforços da tarefa de planejamento da fase de execução do teste de regressão, além de propiciar a escolha coerente ao alvo atual do projeto das, considerando suas funcionalidades e recursos disponíveis para as executar.

Dado um projeto de teste que contenha diversas execuções passadas, não será mais necessário ao gestor analisar todo o sistema para selecionar manualmente as situações de execução que se enquadrem ao escopo das funcionalidades modificadas e associadas. Com isso, o custo despendido(gasto) na execução da tarefa de escolha de situações de teste sofre redução, já que o executor poderá direcionar esforços a outras atividades.

A seleção manual das condições de teste tem interferência negativa em dois pontos principais. O primeiro é o esforço despendido, algo que está diretamente ligado ao tempo proposto ao exercício da tarefa. Ao considerar o tamanho do projeto, essa é uma atividade que pode levar de poucos minutos a muitas horas de execução. O segundo, que também influencia no tempo de execução, é a base de escolha das condições. Os critérios da seleção geralmente estão apoiados no conhecimento de regra de negócio do projeto, por parte de seu executor, [8].

Outro fator que também pode ser considerado na abordagem do problema aqui relatado é a dificuldade em escolher as situações de acordo com a combinação dos parâmetros dos dados de teste. O cruzamento manual entre todos os critérios de escolha dos casos é uma atividade que demanda muito esforço, algo que é condicionado inclusive pelas características de: entendimento das regras de negócio e experiência profissional do executor. Ainda que existam métodos de filtragem de conteúdo, nas ferramentas utilizadas para manutenção dos dados, sempre será realizado um levantamento incompleto, no sentido de prioridade e diversidade do resultado retornado para a consulta. Ainda considerando esse cenário, a dificuldade aumentaria caso o projeto utilizasse somente o modelo de planilhas para a criação e persistência dos dados.

De acordo com os fatores descritos, o objetivo principal deste trabalho é a criação de uma ferramenta que automatize a tarefa de seleção dos casos de teste para auxiliar na resolução do problema de busca que considere múltiplos objetivos. Isso porque, existe a necessidade de reduzir o tempo de execução da tarefa de seleção de casos de teste, que é despendiosa, sem que ocorra desprezo da importância dos cenários prioritários ao projeto e da cobertura desses casos por complexidade. Além disso sua função também é retirar do executor da atividade a necessidade de ter experiência com a execução da atividade de seleção e a imposição de conhecimento total das regras de negócio do sistema, porque essas condições concentram em poucos indivíduos a responsabilidade e habilidade de selecionar os casos para a regressão. Sendo assim, o TSuite forma uma base de conhecimento, no momento da tomada de decisão, que influencia diretamente na qualidade do produto.

Confrontando com o método de seleção manual, os principais benefícios em utilizar o TSuite são: minimização do tempo e consequente custo de execução da tarefa, independência do recurso pessoal utilizado na gerência e seleção das situações coerentes com a disponibilidade dos recursos do projeto de teste e funções do sistema.

Importante ressaltar que o custo da qualidade é um fator essencial no projeto de *software* e impacta diretamente na correta execução das tarefas enquadradas na fase de teste. O tópico seguinte aborda os aspectos concernentes a esse assunto.

2.0.1. O custo da qualidade

Um estudo realizado por [15], citado por [1], ajuda a entender a evolução do valor monetário na tarefa de redução de defeitos do *software*, pois [15] afirma que o custo da correção de um defeito cresce a cada fase que o projeto avança. Por exemplo, a retirada de um defeito em produção é dez vezes maior do que em comparação à fase de implementação do *software*.

Na figura 1 a referência [15] traça um gráfico de acordo com uma função pelos vetores gasto (dólares) versus fase (processo). A função permanece constante até a fase de especificação, onde os modelos de base do *software* comportam grandes mudanças a um custo baixo. Já na fase de construção, quando o *software* começa a ser codificado, a função cresce lentamente, até um momento próximo ao seu limite. Na fase de teste, quando próxima da etapa de produção, a função sobe bruscamente, simbolizando a elevação no custo da correção de defeitos [1].

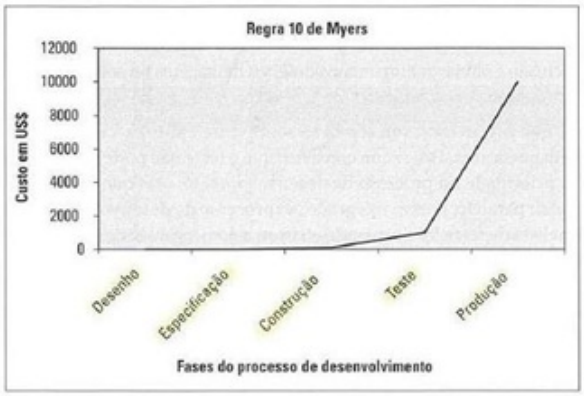

Figura 1: Regra de Myers [1] apud [15]

Inicialmente, na fase de desenho, o custo chega a ser zero (U$0) para a correção de um defeito, enquanto que na fase de produção ele alcança um limite de U$1.000. Isso ilustra uma evolução por múltiplos de 10 a cada mudança de fase que o defeito permanece no *software*, até o momento em que ele for descoberto.

O mesmo autor afirma que testes realizados no código podem reduzir de 30% a 50% dos defeitos de um programa. E os testes na interface do *software* seguem percentual igual à estatística anterior.

Apesar da teoria de [15] ter sido formulada na década de 70, ela ainda incentiva pesquisas nesse sentido nos dias atuais. Mudanças na etapa de teste, como ser posto em um processo à parte da programação do *software*, impulsionaram o estudo [3].

O autor reflete sobre os custos do teste através de um estudo de caso fictício estruturado com base em sua experiência em estudos de caso verídicos. Na figura 2 ele analisa o retorno do investimento em teste de *software*.

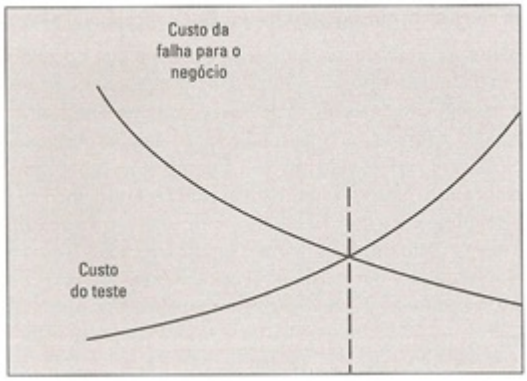

Figura 2: Cobertura de Teste [3]

Essa figura mostra o cálculo do lucro que o projeto teve em evitar o prejuízo decorrente da correção dos incidentes somente com o *software* em produção [3]. Ela prova que ainda que o custo de montagem e operação de um processo de teste possa ser elevado, ele compensa o gasto com a correção desses defeitos falhados quando o *software* já está sendo utilizado pelo cliente.

Uma condição ideal à execução de teste, antes e após a entrada dele em produção, seria considerar apenas as situações de operação essenciais à versão atual do programa, [7]. O TSuite tem como objetivo alcançar a melhor cobertura de execução dessas situações, dado um grande sistema, em um prazo e orçamento estipulados dinamicamente. Melhor no sentido de variedade da complexidade e prioridade dos casos na execução.

Para que isso ocorra da forma mais fiel possível ao estado atual do projeto de teste - aos recursos disponíveis - as unidades de teste que serão selecionadas ganham prioridades e pesos. Fatores que são de utilidade na condição de rastreio e divisão de importância das situações de teste, dentro da fase de execução.

Os dois estudos seguem a linha de definir que quanto mais cedo ocorrer a descoberta e a correção do defeito no *software*, menos custo ele absorve do projeto. A reflexão que as teorias trazem é de que a prevenção ao erro no projeto e no código deve ocorrer. Isso para que ele não permaneça no *software* como defeito e venha a ser manifestado no sistema como falha a um usuário, gerando prejuízo e desgaste exacerbados ao projeto.

Considerando isso, o TSuite ajuda a reduzir os custos do planejamento de execução das condições de teste, já que a tarefa de seleção é executada automaticamente. A consequência dessa ação confere em um recurso pessoal disponível para realizar outras atividades.

Apesar do custo de teste ser elevado e por isso condicionar o projeto de teste, outro fator do projeto de *software* que influencia diretamente na qualidade do produto, é a ponderação dos riscos ao projeto, levantados de acordo com a análise de risco. Essa análise considera os fatos que possam causar perdas ao projeto e que por isso necessitam ter suporte.

A execução de teste de *software* tem como função atuar sobre a linha de riscos do *software*. O executor dos testes verifica o comportamento relacionado a cada tipo de situação a que o sistema é submetido. Essa verificação é realizada através da avaliação das regras de negócio e do ambiente de operação do aplicativo.

A figura 2 apresenta até onde deve ir a cobertura de teste com relação ao custo dele. Conforme a função 'Custo de teste' cresce, a função 'Custo da falha para o negócio' diminui. Como pertencem a um plano comum elas tendem a se encontrar, em um dado momento.

O ponto no gráfico, que simboliza essa interação, é o momento ideal de interromper os testes sem gerar prejuízos ao projeto. A reta pontilhada perpendicular às funções é correspondente a essa barreira de custo. Significando que quanto mais as funções estiverem comumente próximas dela, o custo da falha (produção) não compensa o do defeito (confecção).

2.0.2. *Produtos de Teste*

Assim como na Computação existem os conceitos de *hardware* e *software*, surgiu a denominação *testware*, na Engenharia de *Software*. Segundo [2], *testware* são todos os elementos produzidos por profissionais de teste na etapa de verificação e validação do *software*. A validação corresponde à etapa de estudo e avaliação da regra de negócio modelada ao sistema. Já a verificação referencia a execução do teste de *software*. São exemplos de unidades de *testware*: plano do projeto de teste, *checklists* e *scripts* de execução de teste.

O plano do projeto de teste corresponde a um relatório que contém os esforços da equipe de teste para um dado projeto [1]. Fazem parte desta listagem, os recursos, tanto pessoal quanto os equipamentos, os custos para integrar seu processo ao projeto, assim como o tempo que deverá ser gasto na execução de cada atividade.

O plano do projeto de teste não deve ser confundido com o plano de execução do teste. O primeiro estima e contabiliza em apenas um documento os esforços da equipe durante todo o período que o projeto estiver em desenvolvimento. Enquanto que o segundo reúne o *testware* que será o produto das iterações de produção do *software*, conforme as especificações contidas no primeiro.

No TSuite, o plano de execução de teste tem seu conceito aplicado apenas de forma representativa, porque a seleção deve ocorrer antes de os casos serem executados. A evidência de uma ou mais suítes, como referência ao modelo atual do sistema, o torna implícito.

Os casos de teste são o *testware* que correspondem aos *scripts* de execução do teste, [14]. Tais *scripts* são a estrutura que descrevem o passo-a-passo do teste. Por exemplo, considerando os testes funcionais manuais eles são documentos que contém ações enumeradas em ordem linear de raciocínio e execução e resultados esperados relacionadas à elas. Trazendo o foco para os testes funcionais automatizados, eles são sequências de métodos parametrizados dinamicamente em uma ferramenta de gravação e reporte do estado do teste. Sendo assim, cada caso de teste cobre uma funcionalidade ou regra do sistema.

O TSuite não é uma ferramenta para criar *scprits* automatizados de condições de testes. No entanto ele visa automatizar a tarefa de selecionar os casos de teste manuais derivados de uma fonte de dados que contenha essas unidades de teste.

Conforme [7] os casos de teste podem ser positivos ou negativos. Os casos de teste positivos cobrem as funcionalidades do sistema realizando operações que levem a certificar em saídas corretas. Ou seja, eles confirmam que o *software* faz o que realmente deveria fazer. Já os casos de teste negativos visam demonstrar que o sistema não faz o que não deveria fazer.

Para [1], as suítes de teste são conjuntos de casos de teste. Esses agrupamentos podem ser criados para identificar módulos do *software* ou "versionamento" dos casos de teste. No segundo caso, eles ganham uma *baseline* própria. As *baselines* contém as suítes de execução de uma iteração do *software*. Para a maioria das ferramentas de gerenciamento de testes, elas fazem parte de um plano de execução de teste.

A figura 3 exibe o formato de um projeto de teste quando ele não está passando por nenhuma execução. Essa é também a forma com que as ferramentas de gerência de *testware* armazenam seus dados, já que seguindo este formato vários planos de execução podem ser criados aproveitando os casos que já sofreram teste.

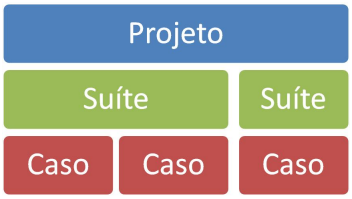

Figura 3: Formato de um Projeto de Teste [14]

Ainda segundo [1], um plano de execução de teste contém uma ou mais suítes. Para ele, cada suíte contém casos de teste, que por princípio, possuem relações entre si. Esse plano segue o planejamento do cronograma proposto no plano do projeto de teste. Sendo assim, o plano de execução de teste deve ser mensurado conforme os prazos e custos do plano de projeto de teste. A figura 4 ajuda a diferenciar o conceito de plano de execução dos casos de teste do conceito de plano do projeto de teste, comparando-a com a figura anterior (3).

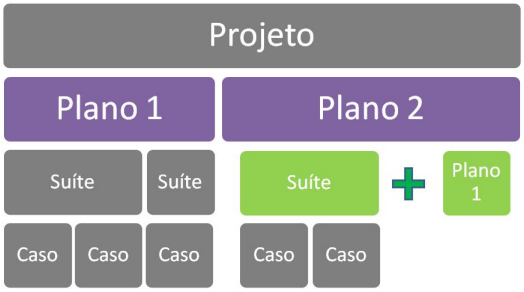

Figura 4: Formato do Projeto de Teste para a execução dos casos [14]

Dentro do projeto de teste tais fatores devem nortear seu planejamento: a probabilidade de ocorrência dos riscos e seus impactos associados. Um método eficaz que busca aproximar a proposta do plano ao nível de conhecimento e produção da equipe de testes é a estimativa.

2.0.3. *Estimativas de Teste*

A estimativa de teste pode ser utilizada na fase de planejamento do teste, mas como define [12], ela não consegue acompanhar as mudanças que o projeto deverá sofrer, em curto prazo. Isso inclui o impacto dos riscos

sobre o projeto e consequente *testware* dele.

De acordo com [12] a estimativa não é uma ciência exata, pois depende de inúmeros fatores, tais como: experiência da equipe, tamanho e complexidade do projeto e tempo disponível para a execução.

O conceito de estimativa serve como base à tomada de decisão inerente ao gestor do projeto. Na atividade de execução da seleção de condições de teste, a operação automática confere a ele uma base mais próxima ao real, sem que ele precise entender as particularidades de todas as ações que compõem o sistema, nem ignorar o alcance desejado da qualidade.

Para que a qualidade do *software* possa ser considerada no projeto é necessário estabelecer coordenação entre as fases de programação e testes. Isso porque o processo do projeto é comum a eles. No caso do processo de gerenciamento do desenvolvimento de *software*, o modelo empregado é em iterações. Ele é incremental, devendo cada versão do *software* atribuir ou atualizar funcionalidades ao sistema.

A metodologia de gerenciamento de teste de *software* não deve estar atrelada à metodologia de gerenciamento da programação do *software*. No entanto, ela deve acompanhar cada fase de implementação do *software*.

A metodologia em V (lê-se vê) é definida sob tal perspectiva, e apresenta-se como ideal ao modelo de negócios dividido por iterações. Isso porque, para cada etapa de criação do *software* deverá existir uma equivalente para o teste [1].

O modelo em V ajuda a entender a sequência de execução do TSuite. Para cada nova versão do sistema seu escopo é modificado considerando-se os acréscimos e remoções realizados. Com isso, é necessário ampliar o nível de execução das condições de teste, para cobrir essas alterações do *software* e garantir o controle da qualidade, que é o objetivo central do Teste de Sistema.

A análise dos fatores de risco e do custo da qualidade são importantes no planejamento do projeto de *software* e do teste de *software*. O que norteia tais métodos são as métricas, que são as médias quantitativas recolhidas na execução de tarefas para nortear estudos e planejamentos de projetos futuros. No entanto, para que a análise seja realizada em profundidade satisfatória é necessário compreender os conceitos que dirigem as atividades do teste.

O gestor de testes, quando planeja a fase de teste para o projeto, escreve um cronograma específico para a equipe, conforme a base de conhecimento que seus integrantes possuem. No cronograma, para cada atividade de execução listada existe uma abordagem de teste específica diretamente relacionada. Tais especificações correspondem aos níveis, técnicas e tipos de teste.

2.0.4. Níveis e Tipos de Teste

Conforme [13], o teste de *software* possui propriedades funcionais e não funcionais. As propriedades funcionais envolvem os testes do tipo Funcional. Eles, de acordo com [8], são testes realizados para verificar a existência e integridade das funcionalidades que devem estar presentes no *software* que está sendo analisado.

Os testes funcionais geralmente são executados conforme a visão da técnica de Caixa Preta, onde o que se busca é verificar o *software* sem que seja necessário conhecer sua estrutura interna. Ou seja, ele é voltado exclusivamente aos efeitos que a ação do ator sobre o sistema poderá provocar visivelmente. O TSuite utiliza o conceito desse tipo de teste como base e resultado da seleção de casos.

Os testes não funcionais, explica [8], são realizados para verificar a resposta do *software* sob as situações que ele pode ser submetido. São itens de observação: desempenho, usabilidade, segurança, acessibilidade, manutenibilidade, recuperação, dentre outros.

Os testes funcionais e os testes não funcionais podem utilizar um ou mais dos seguintes níveis de teste: Teste de Componente, Teste de Integração, Teste de Sistema e Teste de Aceite.

O Teste de Sistema verifica a união das funcionalidades do sistema de uma forma completa. Ele não deve testar particularidades do *software* como trechos de código-fonte ou módulos. Mas sim, a forma de execução – se com sucesso ou falha, conforme siga o fluxo do sistema, [1]. Nesse tipo de teste, deve-se seguir uma sequência de execução lógica dentro de um ou vários módulos do sistema; exemplo: Cadastrar formulário no módulo Pessoa. O objetivo deste tipo de teste é encontrar o máximo de defeitos instalados no programa, antes que ocorra o Teste de Aceite, [14].

O Teste de Aceite considera a avaliação do usuário a respeito do sistema. Muitas vezes o próprio cliente realiza esse tipo de teste, pois ele verifica as conformidades implementadas ao sistema em comparação com o projeto contratado. No entanto, demais aspectos do *software* também refletem na aceitação. Ele está relativamente envolvido com os aspectos não funcionais do *software*. Isso, por conta da expectativa do cliente a respeito do sistema. Vale considerar que neste nível de teste, o usuário não busca encontrar defeitos.

Tanto os testes do tipo Funcional quanto do tipo Não Funcional podem ser levados em uma sequência progressiva ou regressiva. Os testes progressivos estão de acordo com a mais nova versão do sistema. Já os regressivos, tem o foco nas versões anteriores. Para esta prática também se dá o nome “reteste”. O TSuite seleciona os casos de teste para executá-los novamente, por isso ela é uma ferramenta para apoiar a tomada de decisão na seleção de casos de teste de regressão.

Dentro do método de execução de teste regressivo, existem as formas de cobertura - do sistema - parcial e total, segundo [2]. Com a cobertura total, todos os módulos correspondentes às versões anteriores do *software* precisam ser executados novamente. Na cobertura parcial, busca-se agrupar somente um número de funcionalidades por vez. Ou seja, a execução ocorre para um determinado módulo do sistema. Tal técnica é conhecida como *Smoke Test* ou Teste Básico. Mas ela só ganha essa conotação quando aplicada sob condições de tempo de execução restrito para o *software* em produção, como, por exemplo, o "reteste" das funcionalidades que sofreram alteração após o *software* já estar alocado no ambiente do cliente, [7]. O *Smoke Test*, portanto, está empregado no contexto do teste de Manutenção.

O TSuite desconsidera o tipo de teste de regressão que será realizada na execução. No entanto, ele tem como foco a execução de teste em iterações.

Para esclarecer os conceitos e fronteiras da produção de *software* e impulsionar a resolução dos problemas considerados pelas técnicas reflexivas citadas acima – planejamento, análise dos fatores de risco e estimativa - uma nova área da Engenharia de *Software* foi criada. Pesquisas relacionadas às atividades de otimização da Engenharia de *Software* estão contidas em uma área da computação conhecida como *Search Based Software Engineering*, [5]. A SBSE contém uma subárea de pesquisa que envolve somente os problemas decorrentes das atividades de teste, a *Search Based Software Testing* (SBSE), [8].

2.1. **Otimização**

Segundo [18] existem problemas da Engenharia de *Software* que não podem ser facilmente solucionados por suas técnicas tradicionais. Nesses casos, a escrita de um modelo é menos custosa do que, por exemplo, documentar o passo-a-passo de como fazê-lo, contemplando as normas e regras do projeto. Para [10] um modelo é uma visão abstrata de um problema real. Através dele é possível fazer uma projeção conceitual do que é funcional.

Um modelo matemático é conveniente para formular questões onde as operações matemáticas básicas não conseguiriam acompanhar a evolução de um projeto, que é aleatória. Para que o processo possa ser formatado desta maneira ele deve conter variáveis numéricas. Elas correspondem aos recursos que participarão do cálculo.

Para que um algoritmo seja ótimo é necessário que ele satisfaça uma condição especial. Isso é, contando que através de métodos convencionais ela não poderia ser alcançada da melhor forma possível. Desta maneira, conforme escreveu [20], a atividade de otimização visa maximizar ou minimizar a função satisfatória de um modelo matemático. O TSuite utiliza, também, um algoritmo ótimo como forma de selecionar as situações de teste que tenham maior relevância em um dado momento do projeto de teste.

Coeficientes e variáveis são elementos quantificados que definem uma função satisfatória. As variáveis podem ser restringidas, conforme o progresso das equações estabelecidas no modelo. Daí a relevância em aplicar o modelo matemático sob a ótica da computação. A utilização de um algoritmo, escrito em uma linguagem de programação, oferece suporte à execução do modelo dinamicamente, facilitando sua reutilização.

Neste artigo são aplicadas múltiplas funções de satisfação que convergem em um único objetivo final, a seleção mais conveniente. Elas estão de acordo com as características que classificam os casos de teste. São elas: prioridade da funcionalidade, complexidade, tipo e tempo de execução dos casos de teste e prioridade da funcionalidade associada à suíte de teste que contém esse caso.

2.1.1. O problema da seleção de casos de teste

De acordo com [19] é descrito um estudo de caso onde o problema da seleção de casos de testes é questionado. São pré-condições dele: um módulo de um programa que possua uma suíte de teste correspondente e que esse módulo tenha sofrido alterações. O problema é definido como: encontrar uma suíte de teste reduzida para as modificações ocorridas no programa. Ele explica através de [20] e [16] que o problema de seleção de casos de teste envolve a *release* atual e as anteriores do *software*. Uma *release* é uma versão do sistema, também conhecida como *build*. Uma versão pode conter novas partes do sistema e/ou modificações de seus módulos já existentes.

A abordagem de seleção utilizada no TSuite também compartilha desta premissa. Para realizar a análise de impacto originada pela mudança das funcionalidades no *software* é necessário comparar o estado atual dele em relação ao que já foi executado. Isso, para proporcionar uma melhor cobertura de teste sobre ele.

De acordo com [7] a seleção de casos de teste é um problema NP-Completo. No campo da Computação, esse tipo de problema segue sem uma resolução total, já que o bom desempenho em tempo polinomial é uma questão que tende a não ser alcançada. Mas, ao voltar o foco à formulação de um algoritmo de seleção de casos de teste funcionais manuais essa questão pode ser parcialmente abstraída. Isso porque na maioria das empresas, uma ferramenta de gerência de *testware* é utilizada para construir e alterar casos de teste finitos. Tais programas contém uma base de dados que guardam os atributos desses produtos de trabalho, o que torna o processo de validação passível de ser realizado e facilita a sua busca e restrição de consulta por outras ferramentas de gerência ou extração desses dados.

Outra facilitação é a remoção da obrigatoriedade de geração automática do modelo pelo sistema. Também, a maioria das ferramentas de gerência de *testware* contém

um módulo relacionado à geração de relatório, fazendo com que seja possível a extração de uma parte desses dados. Dessa forma, o TSuite torna dinâmica a tarefa de geração de um modelo a cada vez que o *testware* mudar, sendo que fica a cargo do executor informar o projeto e suas alterações. Assim, ocorre rastreio entre o código e os casos de teste manuais.

O problema ainda é mais facilmente solucionado com a atribuição obrigatória de complexidade, custo e tempo de execução, para cada caso de teste candidato à seleção. Essas características possibilitam a conferência de pesos dos casos para que seja realizada a seleção de forma coerente com a necessidade do projeto.

O TSuite utiliza um modelo matemático multiobjetivo na resolução do problema de seleção dos casos de teste. Ele é uma adaptação do modelo encontrado no artigo [8] apud [5], apresentado na figura 7. As funções que devem ser avaliadas nesse processo são: minimizar a despesa da execução dos testes e maximizar a importância dos casos escolhidos. Cada uma dessas funções segue uma premissa apresentada na figura 5, onde, 1 corresponde ao objetivo de minimização e 2 ao objetivo de maximização. Ambas recebem o parâmetro 't' na unidade 'j'– posição do caso de teste dentro do projeto de teste(tj).

A premissa 1 estabelece as variáveis 'k', 'j' e 't', na somatória que resulta no tempo total gasto durante a execução dos casos. A variável 'k' corresponde ao valor de tempo de um determinado caso de teste, enquanto 'j' é a unidade/quantidade de casos de teste.

Na figura 5 a inequação a) é a representação da restrição do problema. O tempo de execução gasto com a soma de todos os casos de teste deve ser menor que ou igual ao tempo disponível para a tarefa.

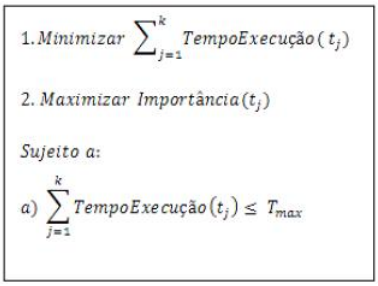

Figura 5: Modelo do problema de seleção dos casos do TSuite

No trabalho [5], antes da seleção é realizado o levantamento de uma base contendo os casos prioritários do projeto. Caso o projeto não tenha recursos disponíveis para selecionar todos, a seleção ótima é executada. No TSuite são selecionados quantos casos prioritários for possível, no tempo disponível para a atividade, utilizando, inclusive, a seleção ótima. Se algum prioritário não for selecionado, isso não será considerado, pois aqueles mais prioritários foram selecionados conforme sua classificação de importância(valor do peso no *ranking*).

2.2. Trabalhos Correlatos

A seguir são apresentadas as soluções que possuem relação com o TSuite. Todas têm em comum o fato de terem sido desenvolvidas para tornar automática a seleção de casos de teste para regressão e, portanto, serviram de base à formação da idéia deste trabalho e implementação das funções do TSuite.

2.2.1. *Estudo de Caso: O algoritmo TestTube*

TestTube é uma solução publicada por [19] para regredir um conjunto de casos de teste unitários. Essa suíte de teste deve possuir correspondência com o código alterado no *software*. Ele identifica os casos conforme a geração de uma nova versão do sistema.

A figura 6 ajuda a entender a ideia básica por trás do TestTube. Nela, os quadrados representam subprogramas -rotinas- do sistema e os círculos correspondem às variáveis dele. Sendo assim, sem utilizar o TestTube, todos os casos de teste precisariam ser executados novamente, com a mudança da versão do sistema. Após utilizá-lo, somente o caso 3 precisaria sofrer esta ação, já que ele contempla todas as alterações de código que o sistema recebeu.

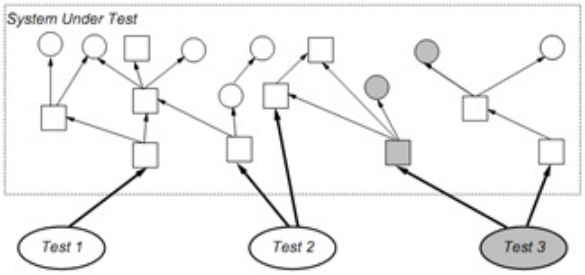

Figure 1: Selective Retesting of a New Version.

Figura 6: Ideia básica do TestTube [19]

Como resultado do estudo de caso, percebeu-se uma redução superior a 50% dos casos selecionados para a regressão.

No TestTube, a cada versão do sistema uma nova base de dados do *software*, contendo o mapa de funcionalidades do sistema, é construída. Com isso, é preciso gerar um novo modelo. No TSuite o modelo é gerado conforme a situação que o executor informe à ferramenta, fator que reduz seu custo operacional e o torna uma ferramenta independente de um projeto específico.

2.2.2. *Estudo de caso: Seleção de teste para alterações emergenciais*

O trabalho [7] propõe a criação de uma ferramenta para selecionar casos de teste para uma regressão rápida. Tal regressão é do tipo *Smoke Test*.

Um modelo é criado conforme o incremento de versões do código do sistema, obtido através de suas publicações em um repositório. Assim uma matriz de correspondência entre os módulos de código-fonte e os casos de teste é criada. Considera-se que para cada escopo de código (funcionalidades do sistema) exista um caso de teste automático em referência.

A proposta do artigo leva em consideração a restrição de tempo imposta à execução de caso de teste. Então, após a seleção de casos de teste que participem da alteração que o código-fonte sofreu, se o tempo restante não for suficiente para concluir a tarefa, um mecanismo de otimização é acionado. Através desse procedimento a seleção passa a obedecer a um período de execução. A escolha é feita contando os casos por sua cobertura e priorização da funcionalidade. Isso após o ator atribuir pesos aos casos positivos e negativos, de forma a somar o valor total de 100% [7].

No entanto, essa ideia difere da proposta inserida neste trabalho, pois ela trata somente os casos unitários automatizados e restringe a seleção somente ao contexto da regressão básica. O TSuite não diferencia a seleção por seu tipo de regressão.

2.2.3. *Estudo de caso: Uma abordagem multiobjetiva para o problema da seleção de casos de teste para regressão*

O artigo [8] apud [5] descreve a formatação do problema de seleção de casos de teste através do modelo matemático multiobjetivo. O modelo considera as variáveis de tempo de execução, importância e risco dos casos candidatos à seleção, para realizar duas minimizações e uma maximização, em cada uma de suas três funções objetivos, respectivamente. Os valores são restritos ao tempo de execução dos casos, tempo de execução disponível, precedência e cobertura dos casos.

A primeira função de minimização contém uma somatória do tempo de execução de todos os casos de teste e deve ser menor ou igual ao tempo máximo (restrição a)). A segunda função de minimização avalia o risco de cada um dos casos de teste para ponderar seu grau de impacto, dentro do projeto, e assim, sua importância. A função de maximização serve para selecionar os casos com maior *score* dentro do projeto, dado um contexto.

Os casos de menor risco e maior importância devem ser selecionados, sem que a soma de seus tempos de execução unitários, ultrapasse o tempo máximo disponível no projeto (restrição b)). Assim como define a restrição c), todos os casos devem ser selecionados avaliando sua precedência(relacionada ao risco) e sua cobertura (importância). O modelo é exibido na figura 7.

O TSuite utiliza este modelo como base para formular sua função específica, na solução do problema de seleção dos casos. No entanto atua de forma resumida, pois a premissa de verificação do risco não é considerada separadamente. Ela está implícita na maximização

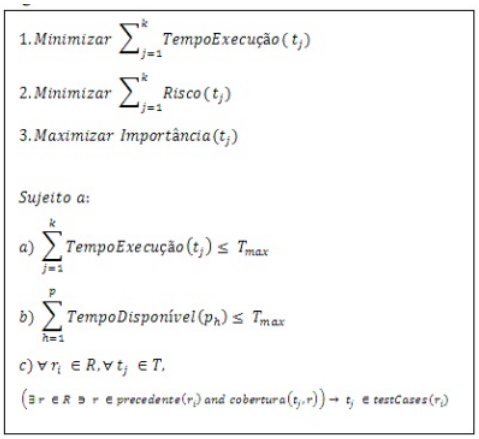

Figura 7: Modelo multiobjetivo para a seleção de casos de teste, [5]

da importância, onde a análise de impacto é contabilizada sobre o peso unitário dos casos de teste. A restrição de cobertura também ocorre na incrementação do peso. Porém, a análise de risco não é restrita à precedência dos casos. Essas características somam à prioridade das suítes do projeto e são contabilizadas como peso dos casos.

A adaptação do modelo também ocorre na restrição da importância, que fica condicionada somente ao critério de peso do caso. Isso porque essa variável representa: a complexidade, tipo e prioridade do caso e prioridade da suíte que o contém, de uma só vez.

Este modelo foi utilizado porque considera a visão do cliente sobre os aspectos do sistema, o que significa dizer que ele está de acordo com o nível de Teste de Aceite. Como o Teste de Sistema possui a estrutura de execução parecida com o Teste de Aceite, utilizando a técnica de Caixa Preta, a visão do cliente é adequada, já que os casos que são o foco deste trabalho são voltados aos testes manuais funcionais de Sistema.

2.2.4. *Estudo de caso: Regressão de testes manuais na prática*

Assim como nos estudos de caso anteriores, [11] busca selecionar uma suíte de teste que contenha os casos de teste que cubram as alterações de um sistema. No entanto, ele determina uma forma de "retestar" os casos de teste manuais.

Este trabalho, inicialmente cita as diferenças entre os casos de teste manuais e automatizados. A primeira delas está na abrangência da área de cobertura das funcionalidades do *software*. Nos testes unitários, somente uma pequena parte do código-fonte é coberta, já os casos manuais possuem uma extensão maior, pois o seu método de execução é fim-a-fim. Ou seja, um passo-a-passo que retrata ação e resultado. A segunda diferença refere-se ao estilo de escrita do *script*. Enquanto os testes automatizados só podem ser escritos de uma forma, que provavelmente não mudará, os casos de teste manuais, que seguem o fluxo funcional do sistema e dependem da formação cultural do analista, cobrem par-

tes mais abrangentes do sistema, conforme as regras de negócio estiverem definidas e o estilo de escrita do analista.

Dessa forma, o trabalho [11] propõe um estudo de caso baseado na seleção de casos de teste automatizados parcialmente. Eles são assim chamados porque automatizam a sua forma de leitura, por padronizar sua forma de escrita. Isso, para poder verificar a aplicabilidade dos casos de testes manuais, com redução da insuficiência dos resultados de minimização da suíte de teste otimizada automaticamente. No entanto, o artigo não define a forma de fazê-lo.

Uma abordagem complementar à [11] seria fomentar a maneira de escrita e armazenagem dos casos de teste manuais a um nível padronizado. Portanto, com o intuito de abstrair essa etapa de coleta dos dados mantidos na ferramenta de gerência de *testware*, este artigo impõe que: os casos estejam dispostos em suítes distintas. Também, que sejam utilizadas informações classificatórias (*tags*) associadas ao escopo dos casos de teste. Exemplos são: funcionalidade igual à [INCLUIR], complexidade igual à média, etc. Assim como torna obrigatória a descrição de prioridade, complexidade e tempo de execução para cada um dos casos de teste concorrentes na seleção.

Ressalta-se que o TSuite considera que cada caso de teste possui somente uma funcionalidade associada. No caso, ao retratar a edição de um registro, por exemplo, a ação de pesquisa que conduz à edição deve estar contida nos passos de execução, já que só é possível atribuir uma funcionalidade a cada um deles. Com isso, é válido que somente a ação principal esteja informada nesta associação.

O tópico a seguir irá descrever a estrutura do TSuite, informando sua arquitetura e os escopos de código-fonte que contém as principais funções do sistema.

3. Métodos e Algoritmos

O TSuite foi desenvolvido sobre a plataforma Java EE(Enterprise Edition), para tornar a utilização do *software* independente do ambiente de operação. A implementação *web* facilita acessá-lo porque dessa forma não é necessário o usuário estar fisicamente instalado em um local de trabalho, nem mesmo impõe a réplica de instalação em cada um dos terminais de execução. A linguagem de programação Java foi usada para receber o suporte do Java EE e possibilitar a integração do *framework* JSF(Java Server Faces) ao ambiente de desenvolvimento.

O TSuite implementa a arquitetura em três camadas, contendo a abstração do padrão de projeto MVC(Model-View-Control/Modelo-Visão-Controle). Na camada de Visão os *frameworks* JSF e RichFaces foram integrados às páginas JSP. A camada de Modelo contém as classes base do projeto, algumas delas são: Caso, Suíte e Plano. Essas classes são "populadas" até o momento da importação do projeto de teste. A camada de Controle possui as regras de configuração, importação e exportação do projeto de teste, incluindo a seleção dos casos.

O TSuite utiliza um método específico de abstração do problema de seleção dos casos de teste. Uma rotina específica foi definida para gerar essa seleção. Ela pondera o peso e tempo dos casos, para selecionar o maior número possível deles, de acordo com o tempo disponível ao projeto de teste. O tópico a seguir descreve a estrutura do TSuite e explica o seu fluxo de execução.

3.0.5. A estrutura do TSuite

Dentre as classes Java mais relevantes da camada de Modelo estão as classes: Caso e Suíte. Um objeto Caso contém as características de: nome, tipo, complexidade, funcionalidade, além do peso, que é uma variável calculada em duas etapas. As variáveis complexidade e funcionalidade contém, cada qual, a instância de sua classe correspondente, que são Complexidade e Tag. Desta forma, cada objeto do tipo Caso, contém indiretamente, o tempo - atributo da complexidade - e prioridade - atributo da tag e posteriormente do plano.

O diagrama de classes exposto na figura 8 ajuda a entender a estrutura interna do TSuite, exibindo a hierarquia e relação das principais classes do projeto. Ela ilustra somente as classes que participam de forma direta da seleção dos casos.

A classe Plano contém uma lista de objetos do tipo Suite, assim como a classe Suíte contém uma lista de objetos do tipo Caso. Essa relação restringe o sistema aos conceitos de plano e suíte de teste, porque cada um deles é formado por um conjunto de uma ou mais suítes e casos, nesta ordem.

Na classe SelecionarCasos existem duas listas de objetos do tipo Caso, que representam as casos disponíveis e os casos selecionados. A lista dos casos disponíveis possui os casos de todos os planos do projeto. A lista de casos selecionados contém os casos disponíveis que passaram pelas restrições impostas pelo problema formulado na ferramenta. São essas restrições: peso dos casos, custo dos casos -tempo- e tempo disponível no projeto para mais uma execução dos casos. Ainda nesta classe, os planos são "mapeados" por prioridade.

As suítes que participam do plano atual de execução no projeto formam o plano de prioridade 1. As suítes que são associadas ao plano 1 ganham a prioridade 2. Assim como aquelas associadas ao plano 1, a partir da segunda ordem ganham, a prioridade 3. As demais suítes, aquelas que não foram associadas ao plano 1, 2 ou 3, constituem o plano 4 e recebem igual prioridade. Existe a possibilidade de os planos 2 e 3 não existirem, pelo motivo de o usuário do TSuite não gerar o modelo do sistema, não ocorrendo assim a formação dos planos 2 e 3. Neste Caso, o projeto do sistema passa a conter somente os planos 1 e 4.

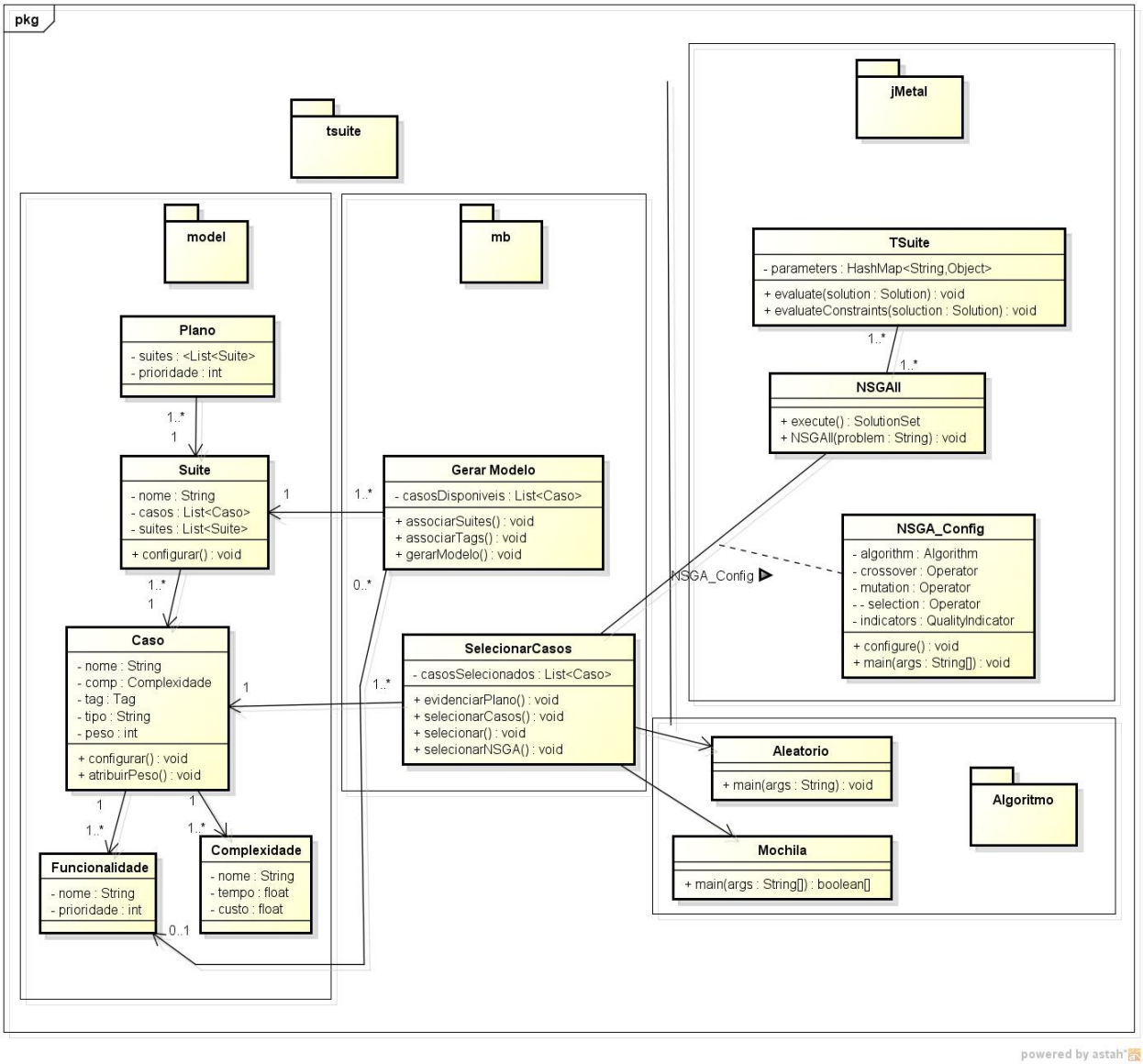

Figura 8: Diagrama de classes do TSuite

3.0.6. A seleção dos casos

O TSuite utiliza os métodos de seleção convencional e de busca ótima para embasar a função de seleção dos casos de teste. Uma rotina específica foi definida para gerar a seleção principal, enquanto que outra, probabilística, é invocada em segunda ordem.

A abordagem híbrida foi escolhida para alcançar o melhor tempo e resultado possível, no problema de seleção dos casos do TSuite. O custo computacional de utilizar somente o método NSGAII é elevado pois ele possue um processo próprio, formado por múltiplas rotinas, que necessitam ser iteradas a cada geração criada no algoritmo, [9]. Enquanto isso, o método do TSuite realiza a "gula" dos recursos disponíveis no projeto(tempo) sem necessitar "visitar" novamente todo o espaço de busca do modelo. Essa combinação garante um resultado otimizado e restringe o tempo de operação da seleção complementar a um tempo de seleção aceitável. Sendo assim, o resultado obtido na seleção passa a não priorizar somente os casos de complexidade alta e do tipo positivo, por conta do modelo específico. Nem tampouco seleciona os casos somente pseudo aleatoriamente, conforme funciona o modelo probabilístico.

O modelo matemático do TSuite possui dois objetivos e foi implementado também em dois passos. O algoritmo específico é utilizado na primeira parte da seleção dos casos, assim como na restrição do tempo disponível ao projeto, na segunda parte da seleção. O algoritmo probabilístico é utilizado na segunda parte da seleção dos casos. Ele é usado para priorizar os casos dos planos 3 e 4 do projeto, após a seleção convencional por priorização estática, na primeira parte da seleção. A seleção probabilística foi aplicada para oferecer justiça aos casos de menor peso, que podem coincidentemente, em sua maioria, ser casos de complexidade baixa e tipo negativo. A justiça ocorre no aumento da possibilidade de seleção desses casos, em relação aos casos de maior peso. Desta forma, o modelo matemático também sofre divisão, pois a função de maximização da importância dos casos ocorre linearmente(estaticamente), na parte 1 da seleção, ao passo que ocorre quase dinamicamente(pseudo aleatoriamente), na segunda parte.

Utilizando a visão da estrutura em árvore do projeto de teste importado no TSuite, os planos de prioridade 1 e 2 participam da primeira parte da seleção, enquanto que os planos de prioridade 3 e 4 participam da segunda parte. Os detalhes dos métodos escolhidos para selecionar os casos estão descritos a seguir:

1. Abordagem Algorítmica do TSuite
   Um algoritmo foi especificamente escrito para implementar duas funções de preparação da seleção e a própria função de seleção. Os métodos de geração do modelo do sistema e evidência do(s) plano(s) de execução de teste atual(is) contém um escopo composto pela iteração de todas as suítes de teste do projeto, assim como seus casos de teste, para compor uma base virtual do projeto e possibilitar a seleção desses dados.

   A função de seleção, inicialmente ordena os casos pelos seus pesos, de maneira decrescente, para facilitar a seleção dos casos mais prioritários. Sendo assim, a próxima etapa é iterá-los novamente, comparando seus tempos unitários com o tempo total disponível no projeto, para selecionar o maior número possível de casos enquanto a despesa não se esgota. Ver a figura 9 apresentada abaixo, contendo o escopo do método selecionar().

```
// Ordena a lista de casos disponíveis, do maior peso para o menor
casos = ordenarCasos(casosDisponiveis);
casosDisponiveis = casos;

/*Parte I da seleção - utilizando um algoritmo específico
 * para selecionar
*  os casos de teste de acordo com a despesa disponível
Os casos devem pertencer às faixa 1 e 2 da atribuição de pesos*/
for (Caso caso : casosDisponiveis) {
    if (tempoTotal >= caso.getComp().getTempo()
            && caso.getPeso() >= 21) {
        casosSelecionados.add(caso);
        tempoTotal = tempoTotal - caso.getComp().getTempo();
    }
}
Repositorio.setCasosSelecionados(casosSelecionados);
casosDisponiveis.removeAll(casosSelecionados);
Repositorio.setCasosDisponiveis(casosDisponiveis);

if(tempoTotal > 0){
    selecionarNSGA();
}
```

Figura 9: O método selecionar()

O algoritmo convencional, desenvolvido para o TSuite, seleciona os casos por seus pesos. A atribuição dos pesos aos casos ocorre em dois momentos.

O primeiro momento é a importação do projeto, quando os objetos do tipo Caso são criados e recebem *tags*. O objeto do tipo Tag possui uma prioridade, que varia entre 1 e 5. Através do método configurar(), da classe Caso, o sistema identifica as características de: tipo, complexidade e prioridade da função associada ao caso, para calcular seu peso inicial. O produto deste cálculo é a soma dos valores relacionados a cada uma dessas características (*tags*).

Se, por exemplo, o caso for do tipo positivo ele ganha o valor 1. Para o tipo negativo o valor do peso permanece inalterado. Os casos de complexidade alta recebem o valor 3, de média o valor 2 e baixa o valor 1. A prioridade da funcionalidade associada ao caso também contribui nessa primeira soma do peso. Como as prioridades só podem variar entre 1 e 5, os casos ganham os valores decrescentes, conforme seja o número de sua prioridade. Assim, a funcionalidade de prioridade 1 agrega o valor 5 ao peso do caso, assim como a prioridade 5 agrega somente 1. Cada uma das 5 funcionalidades só pode ter uma prioridade atribuída, condição essa limitada através da função de Configuração de Tags, no TSuite.

A segunda forma de atribuir pesos aos casos é no momento em que o usuário informa qual(is) a(s)

suíte(s), na quantidade máxima de duas(2), participam do plano atual do projeto.

Assim como exibe a figura 10, a primeira faixa compreende os casos contidos nas suítes do plano atual de execução dos testes e participam os valores a partir de 31 até 40. Da segunda faixa participam os casos que possuem associação com o plano atual de execução dos testes e, portanto, são considerados os casos das suítes do plano de execução anterior, no projeto de teste. Seus valores variam entre 21 a 30. A terceira faixa contém os casos associados ao plano anterior de execução, ou seja, são os casos das suítes associadas a partir da segunda ordem, com o plano atual. A faixa varia entre os valores 11 e 20. A quarta e última faixa de valores, que devem estar entre 1 e 10, é integrado por aqueles casos que não possuem relação com nenhum dos outros módulos do plano 1, 2 ou 3, no sistema.

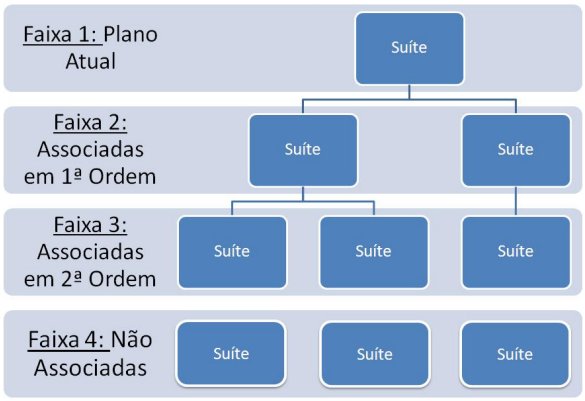

Figura 10: A divisão do Projeto de Teste em faixas de peso para a seleção

2. O Algoritmo NSGAII: Probabilístico
   Como proposta de implementação da solução do problema de seleção de casos de teste, [7] referencia o algoritmo genético. Seguindo igual entendimento, o algoritmo utilizado na implementação da seleção de casos de teste, proposto neste trabalho corresponde a um algoritmo baseado em busca genética, o NSGA-II, descrito por [5]. Ele foi escolhido por ser o algoritmo multiobjetivo mais utilizado na abordagem da seleção de casos de teste e estar presente em diversas publicações do âmbito de otimização, incluindo [8] e [17], o que comprova a sua utilidade. O NSGAII é utilizado no TSuite após a seleção dos casos que participam das faixas de peso para seleção 1 e 2. O que significa dizer que ele seleciona os casos contidos nas faixas 3 e 4.

   O algoritmo NSGAII é considerado uma metaheurística. Segundo [8] define-se que algoritmos de metaheurística representam uma classe de algoritmos de busca genéricos. O que implica dizer que as particularidades como: complexidade e tamanho dos problemas que eles tentam resolver não são considerados.

O mesmo autor explica que a execução de uma atividade metaheurística termina quando uma função traçada é satisfeita por um algoritmo ótimo. A estratégia metaheurística é divida em dois tipos. A otimização mono-objetiva busca completar apenas uma função de satisfação, enquanto que a multiobjetiva é guiada a suprir várias delas.

Para chegar a uma solução comum, que atinja a todos os objetivos do problema satisfatoriamente, é preciso evidenciar o conceito de frente de Pareto. De acordo com [5], este termo define um conjunto das soluções que não são piores que nenhuma outra, incluindo umas em relação às outras, aos objetivos do problema.

O método frente de Pareto está embasado sobre outro conceito: a dominância. Dentro de um conjunto de resultados retornados pelas funções, aquele que obtiver o maior ou menor resultado, em pelo menos um dos objetivos, em comparação com todos os outros, é o dominante. Desta forma, ele é considerado o melhor.

O algoritmo genético é um algoritmo evolucionário que utiliza como base a teoria da evolução através da seleção natural, descrita por Charles Darwin, onde cada elemento é definido na Engenharia de *Software* como um indivíduo ou cromossomo, [5].

Observando a reprodução sexuada das espécies, um indivíduo, pode evoluir relacionando-se e se adaptar ao ambiente através de características próprias ou herdadas que o tornem apto.

O algoritmo genético deve implementar as operações de *crosover* e *mutation*. *Crossover* é o processo de cruzamento entre duas soluções estruturais para gerar duas novas. *Mutation* realiza a atribuição de uma nova característica, comum a todos os cromossomos, quando uma geração possui muitos indivíduos parecidos, para os tornar variados, [8]. No TSuite o algoritmo genético foi utilizado para priorizar pelo menos um peso do plano 3 e outro do plano 4. Isso garante a variedade de cenários selecionados, já que o modo específico prioriza os casos de complexidade alta. Dessa forma, ocorre uma melhor distribuição dos recursos e ocorre justiça aos casos de complexidade baixa.

Conforme exibe a 15, o método selecionarNSGA() passa os valores dos pesos para receber N valores que contenham os melhores valores objetivos, de uma faixa entre 1 a 10 e outra de 11 a 20. Esses intervalos compreendem às faixas 3 e 4 dos

pesos. Uma classe de representação do problema de seleção do TSuite foi criada para conter o formato e restrições da premissa de maximização da importância dos casos. Desta forma, o objetivo do NSGAII passou a ser a extração dos maiores e melhores valores de peso possíveis, dadas as faixas 1 a 10 - do plano 3 - e 11 a 20 - do plano 4 -, no projeto. A passagem de valores entre os métodos é apresentada na figura 11.

```
problemName_        = "TSuite";
numberOfObjectives_ = 2;
numberOfConstraints_= 2;

lowerLimit_ = new double[numberOfVariables_];
upperLimit_ = new double[numberOfVariables_];
if(numberOfVariables_ >= 20){
    lowerLimit_[0] = 1;
    upperLimit_[0] = 10;
    lowerLimit_[1] = 11;
    upperLimit_[1] = 20;
}
else{
    lowerLimit_[0] = 1;
    upperLimit_[0] = 5;
    lowerLimit_[1] = 6;
    upperLimit_[1] = 10;
}
```

Figura 11: Implementação da maximização da importância dos casos dos planos 3 e 4

A premissa de maximização da importância dos casos é aplicada valor a valor, através dos métodos evaluate() e evaluateConstraints(), na classe de representação do problema do TSuite. A seguir a figura 12 apresenta o escopo do método eva luate().

```
public void evaluate(Solution2 solution) throws JMException {
  Variable[] variable  = solution.getDecisionVariables();

  double [] f = new double[numberOfObjectives_];

  f[0] = (int )variable[0].getValue()/4 +  variable[0].getValue() * 1.15;
  f[1] = (int)variable[1].getValue() * 0.95;

  solution.setObjective(0,f[0]);
  solution.setObjective(1,f[1]);
} // evaluate
```

Figura 12: Aplicação das equações de maximização às variáveis de decisão, no método evaluate()

O método evaluate() aplica, sobre um par de variáveis, duas equações que conduzem à divisão dos valores nas duas faixas de peso. A equação que a solução f[0] recebe realiza o acréscimo de mais de 1/4 em seu valor, para que ela fique na faixa de 11 a 20. Já a solução f[1] recebe o decremento de 5% de seu valor, para que ela corresponda aos valores entre 1 e 10.

O método evaluateConstraints(), exibido na figura 13, realiza a restrição das soluções às condições da função objetivo. O objetivo relacionado à essa ação é fazer com que seja diminuído o número de ocorrências de repetições de soluções retornadas como objetivos da função. Além de aumentar a distância entre esses valores.

O tamanho da população deve ser um número par, por conta da função de *crossover*, que realiza o

```
x1 =  (int)x1*1.75 - x2*0.5 + 1.5;
x2 =  x2 + 1;

if(x1 == x2){
    if(x1+2 <= 20){
        constraint[0] =  x1 + 2;
        constraint[1] =  x2 + 1;
    }
}else{
    if(x1>x2){
        constraint[0] =  x1;

        if(x1+1 <= 20)
           constraint[1] =  x1 + 1;
        else
           constraint[1] =  x1 - 5 ;

    }else if (x2>x1){
        constraint[0] =  x2;

        if(x1+1 <= 20)
           constraint[1] =  x2 + 1;
        else
           constraint[1] =  x2 - 5;
    }
}
```

Figura 13: O método evaluateConstraints, responsável por propiciar a variedade das soluções selecionadas

cruzamento por pares. Por causa disso, esse valor é tratado no escopo da classe do problema TSuite, pois, no caso de ser ímpar recebe um número a mais para torná-lo par.

A quantidade de rodadas que o algoritmo é executado corresponde à quantidade de gerações que ele irá produzir até atingir as soluções ótimas. No TSuite este valor foi configurado para 15, para limitar o custo computacional e mantê-lo estável, bem como meio de prevenção à conversão total das soluções que é o estágio onde o *crossover* torna as soluções muito próximas umas das outras.

O método selecionar(), que chama o método probabilístico selecionarNSGA() recebe um conjunto de soluções ótimas e itera a lista de casos disponíveis para selecionar aqueles que o tenham como seu peso. Como os casos continuam ordenados de maneira decrescente - dos mais prioritários aos menos prioritários - os casos de maior peso são selecionados primeiro. Isso ocorre quando, mais uma vez, os tempos unitários dos casos são comparados com a tempo total disponível no projeto, e são decrementados até que o máximo de casos sejam selecionados.

Caso o tempo ainda não esteja esgotado, o método selecionarNSGAII() é executado novamente, tornando-o um método recursivo, como exibe a figura 15.

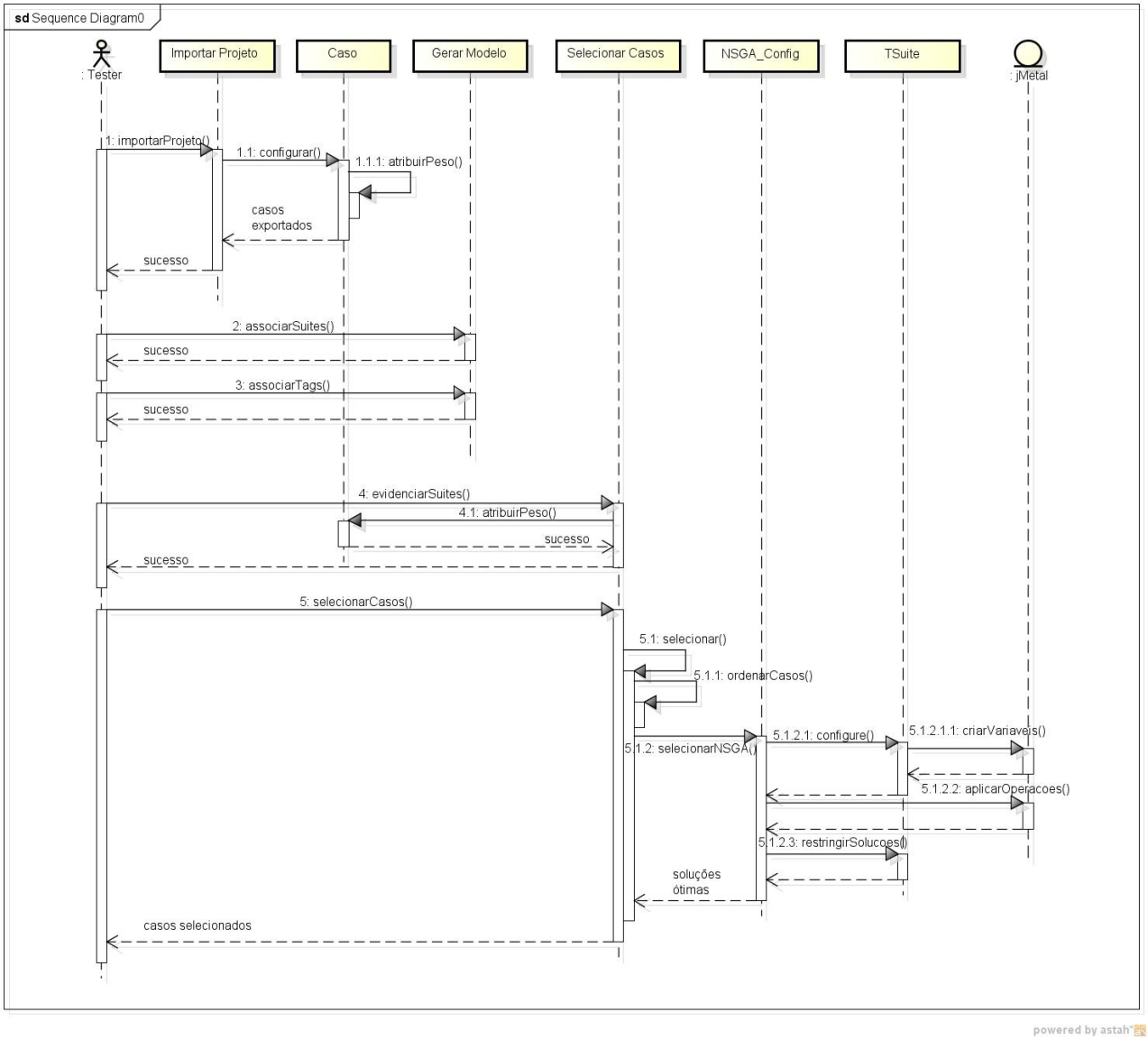

Figura 14: Diagrama de Sequência das principais funcionalidades do TSuite

```
public void selecionarNSGA() throws ClassNotFoundException,
JMException{
    /*Parte II da seleção - utilizando um algoritmo genético para popular
     *  lista de acordo com a despesa restante
    O AG é utilizado para aumentar a variedade/justica dos casos*/
    if(tempoTotal>Repositorio.getComplexidades().get(2).getTempo()
            && Repositorio.getCasosDisponiveis().size() > 0){
        NSGAII_Config config = new NSGAII_Config("TSuite");
        config.populationSize_ = casosDisponiveis.size();
        config.maxEvaluations_ = 15;

        Algorithm nsgaII = config.configure();
        SolutionSet grupoSolucoes = nsgaII.execute();
        for(Caso caso : casosDisponiveis){
            Iterator<Solution> iterator = grupoSolucoes.iterator();
            while(iterator.hasNext()){
            Solution solucao = iterator.next();
            System.out.println(solucao.getAggregativeValue());
                if((caso.getPeso() == (int)solucao.getAggregativeValue()
                        || (caso.getPeso() == (int)solucao.getAggregativeValue()/2)
                        && caso.getComp().getTempo() <= tempoTotal)){
                    casosSelecionados.add(caso);
                    tempoTotal = tempoTotal - caso.getComp().getTempo();
                    System.out.println(caso.getNome());
```

Figura 15: O método selecionarNSGA()

A figura 16 resume os passos que cada uma das etapas da seleção dos casos realiza.

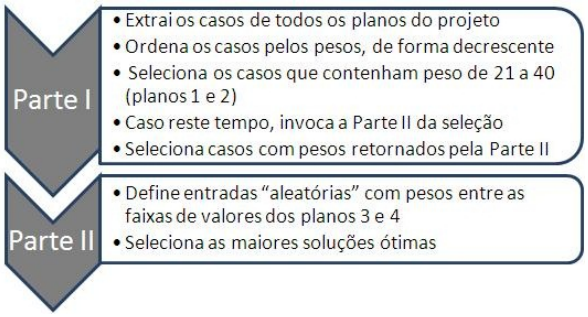

Figura 16: Ações da seleção dos casos, listadas por partes

O diagrama de sequência, ilustrado na figura 14, exibe a visão total dos fluxos principais que relacionam-se à execução da função de seleção dos casos.

Para que a quantidade de casos de complexidade baixa possa ser aumentada, o que evita o desperdício de recursos, o sistema seleciona também os casos que possuam a metade dos pesos correspondentes às soluções retornadas pelo NSGAII. Essa restrição não ocorre para os casos selecionados através do método da Mochila porque seus valores retornados seguem uma ordem crescente, significando que os casos de menor peso, disponíveis para a seleção, serão visitados primeiro, e desta forma ocorre a justiça para eles.

Ao utilizar o algoritmo NGAII, os valores retornados tendem a ser, em maior quantidade, do plano 3, já que as restrições impostas "filtram" essas soluções a cada geração e evidentemente, os valores desta faixa contém maior valor do que aqueles retornados para os valores do plano 4. Já na situação de seleção dos casos, utilizando o algoritmo Aleatório, não há meios que definam uma ordem de retorno dessas soluções, podendo, a inserção da premissa que seleciona os casos com a metade do peso, conduzir à injustiça, neste processo. A diferença entre os algoritmos NSGAII, de solução ao problema da Mochila e Aleatório serão discutidos na seção Testes e Resultados.

3.0.7. Ferramentas utilizadas

Para abstrair a tomada de decisão concernente à seleção complementar de casos de teste, o TSuite utiliza o *framework* jMetal. Ele foi escolhido porque oferece o suporte necessário à utilização das operações do algoritmo genético NSGAII, além de outras metaheurísticas. Ele é um *software* gratuito e de código aberto que está atualmente acessível nas tecnologias: DotNet - com a linguagem C# (jMetal.Net) - e Java - com as linguagens C++ (jMetalCpp) e Java(simplesmente jMetal). O TSuite utiliza a versão 4.0 do jMetal para plataforma Java, pois é o pacote estável mais recente, no momento.

Na estrutura do TSuite o jMetal tornou-se um pacote de classes que foi utilizado como uma biblioteca para oferecer suporte à rotina complementar de seleção. O pacote jmetal.metaheuristics.nsgaII do *framework* jMetal contém a classe Java NSGAII. A implementação do NSGAII no jMetal é uma forma orientada a objetos da classe NSGAII escrita originalmente em linguagem C.

O pacote jmetal.problems contém o conjunto de classes de representação dos problemas suportados pelas metaheurísticas compiladas pelo jMetal. Dentro desta pasta a classe TSuite foi criada, contendo a estrutura do problema de seleção conforme a abstração específica deste trabalho (ver figura 8).

3.0.8. O fluxo de execução do TSuite

O TSuite foi desenvolvido para selecionar os casos de teste que melhor se enquadrem à situação atual do projeto. Para que a seleção ocorra é necessário ao executor completar uma sequência de passos obrigatórios. Ele precisa informar os dados de entrada como: funcionalidades mais prioritárias do sistema, associação entre os módulos do sistema, suítes que participam do plano atual do projeto, tempo de execução de cada caso de teste, por complexidade, assim como o próprio projeto de teste. Tal projeto deve conter casos com 3 *tags* cada um, correspondentes às características de: tipo, complexidade e funcionalidade.

Já na introdução do sistema o executor deverá informar até 5 *tags* que correspondam às funcionalidades do sistema relacionadas aos casos. A operação é assim disposta para atribuir prioridades às funcionalidades do sistema. Um exemplo seria informar a *tag* [INCLUIR] com prioridade 1, ou seja, a principal. O benefício incutido nessa ação é retirar a obrigatoriedade de atribuir prioridade caso a caso, pois entende-se que somente no momento da seleção é possível assimilar a relevância deles, de acordo com a situação atual do projeto.

O TSuite tem como entrada mandatória que a funcionalidade associada a um caso esteja informada entre colchetes (símbolos '[' ']'). Esse é um padrão estabelecido aqui para que seja possível ao sistema e fácil ao usuário do TSuite diferenciar as funcionalidades das outras informações do *software* que devam ser associadas em forma de *tag*.

A configuração de parâmetros do projeto segue igual formatação da função de configurar funcionalidades, porque abstrai a atividade de informar caso a caso o parâmetro de tempo. Tarefa essa que seria desnecessária, pois entende-se que a base de conhecimento dos recursos do projeto é constituída pela média retirada por um critério da equipe, incluindo da atividade de seleção de casos para regressão. Nesse caso, assume-se o critério da complexidade. Ou seja, esses seriam valores comuns a todos os casos que pertençam a uma das categorias de complexidade: baixa, média e alta.

A informação de complexidade corresponde à quantidade de passos de um caso versus a dificuldade de execução que ele contém. No projeto de teste utilizado para

a realização dos testes do TSuite o critério utilizado foi: se o caso possui até 5 passos, ele é de complexidade baixa; se contém até 10 ele é de complexidade média; e se contiver acima de dez 10 passos ele é considerado de complexidade alta. A depender da quantidade de pré-condições ou pós-condições que ele possua essa classificação poderia ser alterada para uma ou outra complexidade. Esse técnica foi adotada na ferramenta por ser um padrão em projetos de teste de médio porte. A quantidade de passos relacionada à complexidade do caso deve refletir a dimensão do *testware* do projeto de teste.

Ao gerar um modelo o usuário realiza as funções de associar suítes e associar funcionalidades à relações de suítes. O passo, que não é obrigatório, permite definir a relação entre as suítes de teste do projeto, gerando indiretamente um modelo do sistema. Uma suíte associada a uma funcionalidade confere na priorização de todos os seus casos que contiverem tal *tag*(funcionalidade). A interface gráfica da ação de gerar modelo é ilustrada na figura 17.

Figura 17: Passo Gerar Modelo, da aplicação TSuite

Assim como o passo Gerar Modelo, a função de Selecionar Casos é constituída por partes. A evidência do plano realiza o cadastro do plano atual e dos demais, o que pode incluir o plano de integração.

Das etapas aqui relatadas, somente os passos Gerar Modelo e Exportar Projeto são opcionais. No entanto a tarefa de Gerar Modelo é o único meio de aplicar a análise de impacto no momento da execução da seleção dos casos, o que implica dizer que, sem informá-lo, essa análise será desconsiderada. A tabela 1 lista todos os passos contidos no TSuite e as explica em resumo.

3.1. Testes e Resultados

O cenário para a realização dos testes de validação do TSuite foi o centro de teste de uma empresa de tecnologia da informação. O centro de teste, no momento da aplicação dos testes, continha 23 profissionais, sendo que 6 deles possuía mais de 4 anos de experiência na área, 7 com 1 a 4 anos de experiênca e 10 com menos de 1 ano de experiência. O teste foi constituído de duas partes, a primeira com foco sobre a vantagem em utilizar a forma automática na seleção dos casos em relação ao método manual. A segunda parte evidencia a necessidade em priorizar os casos dos planos 3 e 4 para oferecer justiça aos casos de complexidade baixa, dentro dessas faixas.

O método de seleção utilizado no primeiro teste foi puramente convencional, com a intenção de oferecer condições de comparação com a segunda etapa de testes. O arquivo XML utilizado na importação do projeto de teste é uma projeção dos dados de um sistema real para gerenciamento de serviços sociais. O projeto fictício foi mantido na ferramenta TestLink e portanto o arquivo gerado segue seu padrão de formatação. Os casos de teste receberam as *tags* de: tipo, complexidade e funcionalidade, para atender às necessidades de importação da ferramenta.

A validação foi realizada levando em consideração três tipos de executores de teste. Os dois primeiros grupos foram formados por profissionais de teste com experiência na realização da tarefa de seleção e execução de casos de teste manuais através da ferramenta TestLink. A diferença entre eles é que, a primeira equipe foi constituída pelos indivíduos que possuem entendimento sobre as regras de negócio do projeto importado na ferramenta, enquanto que o segundo não. A terceira equipe foi formada por profissionais que possuem o nível de experiência inferior aos grupos citados anteriormente, mas que obteve acesso às regras do sistema.

O teste foi constituído de duas etapas e aplicado para três equipes, com cada equipe contendo 2 profissionais de teste. Cada uma das fases ocorreu em períodos distintos, mas contou com o ambiente de operação e pessoal iguais. No primeiro teste realizado os grupos receberam a atividade de selecionar manualmente os casos de teste de regressão que possuíssem maior relevância de acordo com o objetivo do plano de execução atual do projeto. Foi dado o tempo de 30 minutos para ambos realizarem a atividade de selecionar uma suíte de teste prioritária em meio a 63 casos de teste de complexidade baixa ou alta. O projeto de teste modelado não possuía casos de complexidade média, pois eles foram retirados para evidenciar ainda mais a desvantagem em utilizar a abordagem puramente convencional. É perceptível na tabela 3 que os casos de complexidade baixa estão em menor quantidade do que aqueles de complexidade alta.

Complexidade do caso de teste	**Tempo(min)**
Baixa	12
Média	24
Alta	48
	*
Total(Tempo)	8h

Tabela 2: Recursos disponíveis ao Teste 1

Módulo do sistema	Objetivo
Login e Cadastro de usuário	Autenticar Usuário no sistema
Configurar Tags	Atribuir prioridade aos casos que possua uma funcionalidade associada a ele
Configurar Parâmetros	Informar os custos unitários dos casos de teste
Importar Projeto	Extrair as suítes e casos de teste do projeto de teste
Gerar Modelo	Criar um modelo virtual do projeto de teste
Selecionar Casos	Restringir o projeto de teste a um conjunto prioritário de situações do sistema
Exportar Projeto	Disponibilizar ao TestLink a suíte de regressão gerada

Tabela 1: Ações do sistema

Tipo de Ex.	Variáveis	Equipe Verde	Equipe Azul	Equipe Vermelho
Manual	Tempo(min)	22	30	30
	Qtd.	12	22	14
	Variedade	7-A|10-B	5-A|17-B	6-A|8-B
	Resto(min)	84	296	156
TSuite	Tempo(min)	19	15	16
	Qtd.	13	13	13
	Variedade	9-A|4-B	9-A|4-B	9-A|4-B
	Resto(min)	0	0	0

Tabela 3: Resultados obtidos para o Teste 1

Foi obrigatório levar em consideração também os casos de integração entre esse módulo e os demais. O resultado segue a forma das hipóteses 1 e 2 e esquema de resposta ilustrados nas tabelas 3 e 4.

Na tabela 2 são listados os recursos conhecidos, no problema passado aos grupos Verde, Azul e Vermelho, para a validação da ferramenta. Tanto o tempo de execução unitário dos casos de prioridade baixa quanto o tempo total gasto para a realização da tarefa seguem as métricas reais da equipe de teste alocada para o projeto relatado. O tempo de execução unitário para os casos de complexidade média e alta são hipotéticos(fictícios) mas são múltiplos do tempo de complexidade baixa propositalmente, para atribuir proporcionalidade aos casos conforme suas complexidades.

Os grupos utilizaram as informações da tabela citada acima, tabela 2, para selecionar os casos de teste através do TSuite. O modelo e especificação do sistema ficaram disponíveis somente para as equipes: Verde e Vermelho. A equipe Azul somente teve acesso ao modelo do sistema no momento da seleção automática dos casos. Os resultados alcançados por esse meio foram analisados e estão dispostos na tabelas 3 e 4.

A tabela 3 exibe os resultados obtidos na execução do teste, para as duas fases que o constituiu: seleção manual e seleção com o TSuite. As linhas correspondem aos dados relacionados a um grupo de execução, em separado. Cada uma das linhas é apresenta duas vezes, para possibilitar a comparação dos resultados entre os métodos de seleção: manual e automático. A linha 'Tempo' apresenta o tempo de execução da tarefa; a linha 'Qtd.' exibe a quantidade total de casos de teste selecionados; a linha 'Variedade' apresenta a quantidade de casos selecionados, por complexidade dos casos de teste - a legenda 'A' refere-se à complexidade alta e a *tag* 'B' à complexidade baixa; a linha de título 'Resto' apresenta o recurso de tempo total que não foi utilizado.

A tabela 4 segue o mesmo formato da tabela 3, à exceção da informação de 'Variedade' que não é exibida. O conteúdo das células exibe a análise de seu resultado dentro de valor contido em uma escala. Para a seção 'Manual', a linha 'Tempo' apresenta o tempo de execução da tarefa, na escala: <Satisfatório, No Limite e Excedido>; a linha 'Qtd.' exibe a quantidade total de casos de teste selecionados, na escala <Suficiente, Muito e Pouco>, em relação ao resultado apresentado pelo TSuite; a linha de título 'Resto' apresenta o recurso de tempo total que não foi utilizado, na escala <Desperdício e Ideal>. Na seção 'TSuite', as linhas de nomes iguais aos da seção Manual possui conteúdos que permitem a comparação dos critérios do problema, entre os métodos, mas a linha 'Tempo' apresenta o tempo de execução da tarefa, na escala: <Melhorou, Mantido e Piorou>.

As hipóteses 1, 2 e 3 foram formuladas para fundamentar a proposta de automatizar a atividade de seleção dos casos de teste para regressão. Cada uma delas será explicada a seguir:

1. Hipótese 1: O tempo de seleção manual de casos de testes manuais depende do entendimento do indivíduo da regra de negócio do sistema e da experiência do profissional de teste na execução desta tarefa.

   Prova da Hipótese 1: A tabela 3 exibe o conteúdo através do qual é possível comparar o tempo de execução da atividade baseada na experiência do profissional de teste em relação ao teste manual e conhecimento de regra de negócio do sistema. A equipe de nome Verde conseguiu cumprir a tarefa em menor tempo e obteve vantagem relação

Tipo de Ex.	Variáveis	Equipe Verde	Equipe Azul	Equipe Vermelho
Manual	Tempo(min)	Satisfatório	No Limite	No Limite
	Qtd.	Suficiente	Muito	Pouco
	Resto(min)	Desperdício	Desperdício	Desperdício
TSuite	Tempo(min)	Melhorou	Melhorou	Melhorou
	Qtd.	Suficiente	Suficiente	Suficiente
	Resto(min)	Ideal	Ideal	Ideal

Tabela 4: Análise dos Resultados do Teste

às demais, por possuir entendimento do negócio do sistema e experiência na execução da tarefa de seleção de casos de teste. As equipes de nome Azul e Vermelho alcançaram o tempo limite para a realização da atividade. O grupo Azul, por não conhecer o sistema e o grupo Vermelho, por não possuir experiência na execução da tarefa. Essa hipótese prova que a ferramenta, independe da relação de experiência do profissional que a utiliza, assim como do conhecimento das regras de negócio do *software*, conforme exibe a tabela 4. No entanto, ela não descarta que ele necessite entender, ao menos, a estrutura do negócio para gerar o modelo do sistema no TSuite.

2. Hipótese 2: A seleção automática de casos de teste reduz o tempo de execução da tarefa de escolha de situações de teste, em relação à seleção manual.

   Prova da Hipótese 2: A tabela de resultados do teste 1, tabela 3, exibe a redução dos valores correspondentes ao tempo de execução da tarefa proposta, comparando o tipo de execução manual com o tipo de execução automático. As equipes Azul e Vermelho, juntas, alcançaram aproximadamente 50% de redução no tempo de execução da tarefa de seleção. A figura 18 ilustra que dentro do tempo de 30 minutos ambas as equipes alcançaram diminuição do período de execução da atividade de seleção dos casos utilizando o TSuite.

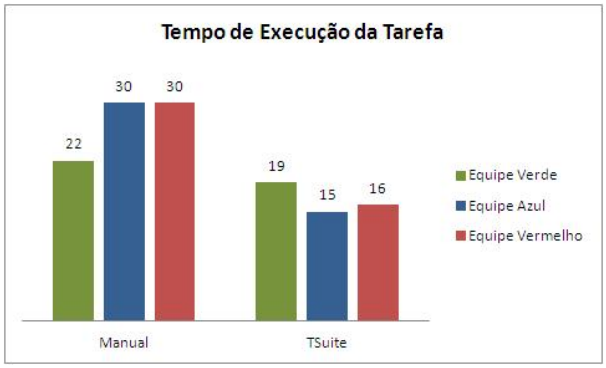

Figura 18: Tempo de Execução da atividade de seleção

3. Hipótese 3: O TSuite evita o desperdício de recursos limitados do projeto de teste.

   Prova da Hipótese 3: De acordo com as tabelas 3 e 4 e o gráfico exibido na figura 19, todas as equipes desperdiçaram recursos limitados do problema para selecionar os casos de teste, utilizando o método manual. Ao passo que ao ter como apoio o TSuite esses recursos não sofreram prejuízo, provando que a seleção através da ferramenta ocorre com base no aproveitamento máximo dos recursos disponíveis. Essa condição é alcançada porque o TSuite itera a lista de casos disponíveis até que não seja mais possível consumir o tempo unitário de um caso do tempo total disponível para a execução da atividade.

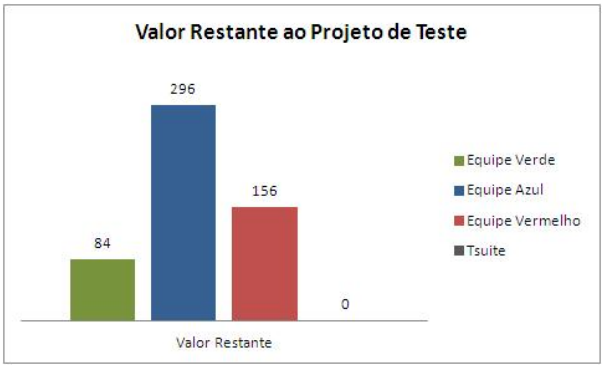

Figura 19: Recursos desperdiçados

Além das provas citadas acima, impressões como nível de dificuldade da execução da tarefa de seleção e satisfação, foram questionadas. Como resultado, todas as equipes consideraram a seleção que utilizou o TSuite, 'Fácil' e 'Ótima', respectivamente. As escalas utilizadas nessa medição foram: <Fácil, Moderada e Difícil> e <Ruim, Regular, Boa e Ótima>, nesta ordem.

A segunda etapa de testes do TSuite visou provar que a utilização de um algoritmo genético, na seleção complementar dos casos, conduz a um resultado mais proveitoso ao objetivo do projeto em seu estado atual. Desta vez, o projeto de teste utilizado para realizar a validação continha 84 casos de teste com a quantidade de casos por complexidade(baixa, média e alta) sendo de exatamente 1/3 deste valor(28).

Esse teste foi novamente aplicado no cenário do teste da etapa 1, mas somente para o grupo Verde. Isso foi feito para verificar a aplicação real dos casos, além da comparação de variedade dos casos selecionados.

Variáveis	Ex. Manual	Ex. TSuite	Ex. Aleatório	Ex. Mochila	Ex. NSGAII
Tempo(min)	26	*11	11	11	11
Qtd.	19	23	29	29	31
Variedade	6-A|6-M|7-B	9-A|10-M|4-B	7-A|13-M|9-B	7-A|10-M|12-B	6-A|11-M|14-B
Resto(min)	204	0	0	0	0

Tabela 5: Resultados do Teste 2

*Tempo médio

A utilização de uma técnica de busca ótima foi concebida pela necessidade em ampliar a cobertura de testes do plano, sem que ocorresse desvio de prioridade para atingir esse efeito. Realizando justiça aos casos de complexidade baixa é possível ao executor selecionar um número maior de casos de teste para a execução manual. O gráfico da figura 20 prova isso, quando exibe a seleção de mais testes de complexidade baixa do que aqueles de complexidade alta, simbolizando que na vida real os analistas de teste ou gestores optam por selecionar vários casos, possibilitando a verificação de diversos módulos do sistema, a concentrar esforços somente nas suítes que participem do plano atual de execução.

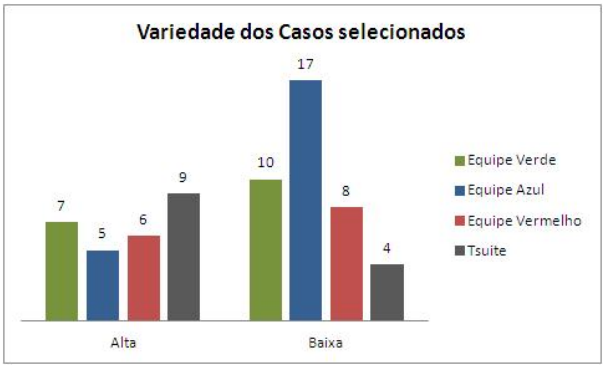

Figura 20: Variedade dos casos selecionados

Os tópicos a seguir descrevem os algoritmos que foram testados com o intuito de favorecer a seleção que propicie sua utilização real.

O problema aplicado na validação segue o modelo disposto na tabela 2 para o teste anterior, com a mudança do tempo total disponível, que passou de 8 horas para 12 horas, já que o tamanho do projeto agora é maior. Todas as formas de validação utilizaram os mesmos relacionamentos entre suítes e tags na geração do modelo e evidência do plano atual. A tabela 5 segue o formato da tabela 3 da primeira etapa de testes.

As informações utilizadas para preencher o resultado da execução dos métodos: Algoritmo do TSuite, Aleatório, Mochila e NSGAII correspondem ao melhor caso extraído após a comparação de 10 execuções de cada um deles. A exceção ocorreu para o algoritmo TSuite, que retorna sempre uma solução fixa, já que a seleção ocorrerá sempre na mesma ordem em que os casos disponíveis receberem seus pesos e sejam alocados na lista de casos disponíveis para a seleção.

A 4ª execução corresponde à medida em que o algoritmo NSGAII apresentou desempenho superior aos demais. O pior caso obtido para o algoritmo NSGAII foi a quantidade de 25 casos, valor esse que é o mais frequente para o algoritmo da Mochila. O pior caso do algoritmo Aleatório foi a quantidade de 22 casos e não foi possível detectar a quantidade mais frequente, pois ele não obteve resultados iguais. O pior caso obtido para o algoritmo da Mochila foi de 23 casos selecionados. O melhor caso obtido para os algoritmos de solução do problema da Mochila e o algoritmo Aleatório foi a quantidade 29. O algoritmo NSGAII possui a quantidade de 26 casos como a mais frequente e 31 como seu melhor caso. A figura 21 exibe a evolução da quantidade de casos retornados pelo algortimo NSGAII durante as 10 execuções que possibilitaram defenir o pior caso, o melhor caso e o mais frequente.

Nota-se que, todos os testes realizados com o TSuite conseguiram consumir todo o tempo disponível no projeto para a regressão dos casos, até mesmo para a etapa de testes 1. Isso se deve ao fato de que a rotina gulosa fica a cargo do método específico do TSuite e portanto independe do método auxiliar utilizado. O valor foi "zerado" porque a quantidade de casos candidatos à seleção foi superior à quantidade de custo alocado para seu grupo prioritário, tornando assim, uma próxima seleção possível, neste contexto, até que o tempo total fosse esgotado. Observa-se também que nenhum outro método, a não ser o NSGAII, permitiu ao TSuite selecionar mais casos de complexidade baixa.

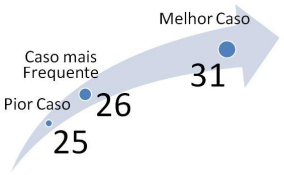

Figura 21: Classificação do desempenho do algoritmo NSGAII para 10 execuções sobre o problema TSuite

3.1.1. Algoritmo para solução do Problema da Mochila

O algoritmo contendo o problema matemático conhecido como o problema da Mochila foi utilizado no TSuite para comparar os resultados da seleção complementar dos casos de teste, em relação ao algoritmo genético NSGAII. Ele é um algoritmo de busca e seleção que tem como base o princípio de frente de Pareto e portanto, atende aos mesmos objetivos que o NSGAII, no TSuite.

O problema da mochila consiste na ideia de preencher um recipiente com tantos itens disponíveis quanto for possível à sua capacidade total. A abstração deste algoritmo utiliza os pesos, seus benecífios e quantidade dos itens como parâmetros de operação, [4].

Apesar de este algoritmo ser bastante simples e de uso difundido na comunidade de otimização, ele fica em desvantagem em relação ao algoritmo NSGAII. As soluções geradas por seu processo de execução são mais próximas entre si do que aquelas geradas pelo NSGAII. Isso se deve à aplicação do método de frente de Pareto que dispõe as soluções candidatas em um plano, [17], sem que se leve em consideração a distância entre elas, [4], diferente do NSGAII que seleciona as soluções mais distantes possíveis, entre si, nessa frente, [8].

As restrições impostas ao algoritmo, foram configuradas da seguinte forma: quantidade de itens igual a quantidade de casos não selecionados no projeto, para os planos 3 e 4 (70), peso igual a 10, pois é uma dimensão do objeto que pode ser considerada como o primeiro limite de restrição na composição da mochila, além do tamanho unitário de 20 objetos, que é a segunda restrição.

Para compará-lo com o NSGAII, uma lista de atributos foi invocada, para criar valores randômicos, entre 1 e 20, e assim gerar resultados comparados par a par, através da frente de Pareto.

3.1.2. Algoritmo de Busca Aleatória

O algoritmo Aleatório segue um modelo que recebe os parâmetros similares aos recebidos pelo algoritmo de solução ao problema da Mochila. No entanto, ele não aplica os conceitos de dominância e geração de funções objetivos. Com isso, a prioridade dos casos não reflete a importância da utilização real dos casos, porque não há limitações que o possibilite.
‘ Esse algoritmo foi utilizado para provar que a geração de valores randômicos não supre a condição de priorização dos casos que visem a variedade dos casos, pois a cada execução do algoritmo ele apresentará um resultado totalmente divergente do anterior em seus critérios. Isso pode, inclusive, fazer com que os casos selecionados sejam inadequados ao objetivo do plano de execução atual do projeto de testes.

As hipóteses 4 e 5 foram formuladas para dar base à utilização do algoritmo NGSAII. Elas serão explicadas à seguir:

1. Hipótese 4: A utilização do algoritmo NSGAII amplia a justiça sobre os casos de complexidade baixa.

   Prova da Hipótese 4: Observando, na tabela 5, os resultados obtidos com o teste do Grupo Verde, e o gráfico de comparação com a primeira etapa de testes, exibido na figura 25, é possível verificar que enquanto o método utilizado era puramente convencional, ou seja, específico, os casos de prioridade alta obtinham grande prioridade, em relação aos casos de prioridade baixa. Ao passo que, ao utilizar o algoritmo NSGAII todas as complexidades puderam ser aproximadamente selecionados conforme o custo que eles consomem do projeto. Isso é perceptível ao comparar a quantidade de casos retornados por complexidade para o melhor caso de execução do algoritmo. Os casos de prioridade baixa conseguiram atingir maior quantidade do que os casos de complexidade alta e média, o que significa dizer que os recursos foram melhor utilizados e que por isso o desperdício de recursos foi diminuído e a cobertura dos cenários de teste do plano atual de execução do “reteste” foi ampliada.

Os gráficos de dispersão de valores, apresentados abaixo nas figuras 22, 23, 24 e 25, também provam o benefício em utilizar o algoritmo NSGAII em relação aos algoritmos de solução do problema da Mochila e Aleatório. Cada um dos pontos apresentados nos gráficos simboliza uma solução alcançada com a aplicação de certa técnica de seleção. Esses números correspondem às variáveis geradas de forma aleatória, restrito somente ao mínimo de 1 e máximo de 20, que a depender do método utilizado passaram por restrições de adequação à função objetiva.

O algoritmo NSGAII tem como resultado valores concentrados, dentro de um espaço de busca. Isso ocorre por conta da função objetivo, que restringe a exploração gradativa desse espaço. O gráfico 22 exibe o resultado das 4 primeiras iterações dentro das 10 execuções realizadas, de forma a retornar o máximo de valores ótimos possíveis.

Os valores das soluções ocupam um espaço de busca comum por conta da implementação da função objetivo. Esses valores tendem a ficar mais próximos entre si à medida que o número de gerações - rodadas - que essa função é iterada, aumenta. Isso acontece porque a população inicial é gerada de forma aleatória e vai sendo “filtrada” a cada geração de indivíduos que o algoritmo cria.

Analisando a figura 22, entende-se a utilização do número 15 como valor padrão da quantidade de gerações que o modelo deve ser executado. Ao utilizar a quantidade iterações 5 os valores retornados pelo algoritmo revelam que ela não consegue explorar todo o espaço de busca disposta para a função objetivo, enquanto que os valores 10 e 20 ficam muito próximos em seus resultados, representando, respectivamente, momentos de ascensão e queda do desempenho da busca do algoritmo NSGAII aplicado ao problema de priorização dos pesos dos casos contidos nos planos 3 e 4, do problema de seleção de casos do TSuite. A quantidade de 15 gerações consegue explorar

todo o espaço de busca porque atinge as restrições extremas do espaço de busca das soluções.

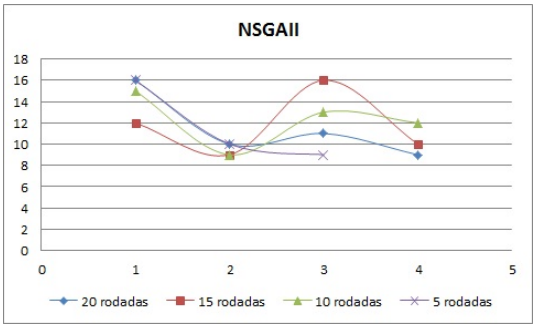

Figura 22: Dispersão dos valores das soluções ao utilizar o algoritmo NSGAII

Com a utilização do algoritmo da Mochila, os valores passam a crescer gradativamente no espaço de busca. Ou seja, os resultados são gerados sequencialmente mas sem que ocorra a comparação desses resultados com os valores obtidos anteriormente. Com isso, os resultados alcançados nem sempre são variados, porque não ocorre busca cíclica como no NSGAII.

Com a utilização do método da Mochila os valores tendem a crescer gradativamente no plano de busca, já que suas soluções não passam pelas várias gerações que o NSGAII tem como base. Os pontos apresentados na figura 23 evidenciam esse nível de flexão das restrições dos valores. O teste também foi realizado comparando-se as 4 primeiras iterações do algoritmo, dentre as 10 execuções pelas quais passou.

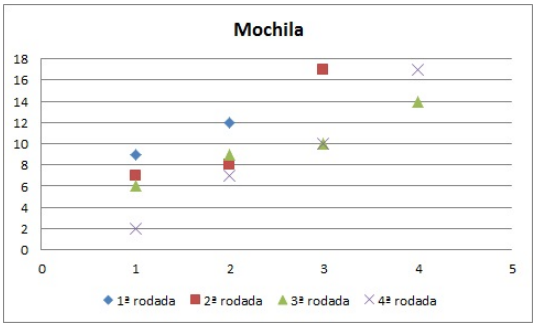

Figura 23: Soluções dispostas nos planos 3 e 4, ao utilizar o algoritmo de solução do problema da Mochila

O algoritmo Aleatório é desprovido de meios que permitam chegar a um objetivo específico, por isso suas soluções são consideradas de baixa relevância à solução de um problema específico, [5]. Isso ocorre pela falta de aplicação da frente de Pareto e da função objetivo, em comparação aos algoritmos da Mochila e NSGAII, respectivamente.

O algoritmo aleatório gera soluções que estão dispersos por todo o espaço de busca, denotando que ele não possui relação com o problema específico a ser resolvido. Sua função cria soluções randômicas, conforme exibe a figura 24.

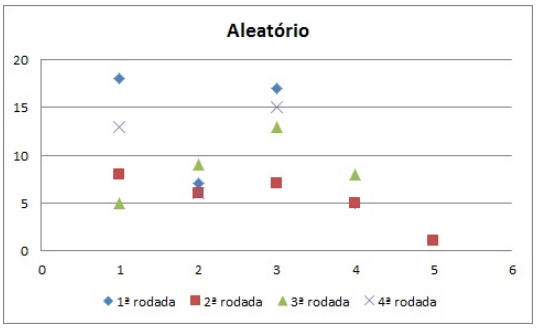

Figura 24: Resultado da seleção utilizando o método Aleatório

O gráfico de comparação, apresentado na figura 25, comprova a utilidade do algoritmo NSGAII para implementar a solução do problema de seleção dos casos no TSuite:

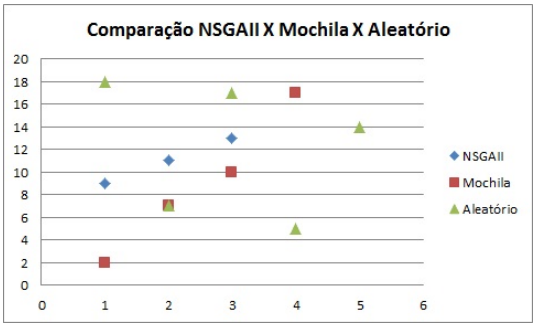

Figura 25: Comparação entre os métodos utilizados na função auxiliar à seleção dos casos

2. Hipótese 5: A utilização do algoritmo NSGAII pode aumentar a quantidade de casos selecionados, por conta da utilização maximizada dos recursos disponíveis.

   Prova da Hipótese 5: A tabela 5 exibe a utilização máxima dos recursos, o que também é propiciado pela maior quantidade dos casos de baixa complexidade selecionados, já que eles conferem em menor despesa ao projeto. Em relação aos algoritmos da Mochila e o Aleatório, nem sempre isso é levado em consideração, já que eles não ponderam a solução mais variada e por isso, podem não proporcionar justiça de complexidade dos casos, por seus pesos, na rotina de seleção principal(gulosa).

   Ainda que o algoritmo NSGAII não retorne bons resultados em todas as execuções, ele busca maximizar as funções objetivo, a fim de retorná-los de maneira proveitosa, então, de alguma forma as soluções retornadas por ele serão coerentes com a resolução do problema de seleção dos casos,

ainda que nem sempre sejam as mais adequadas.

A tabela 5 exibe o aumento da justiça dos casos de complexidade baixa na seleção dos casos, se comparado com o resultado do teste para o método de seleção complementar TSuite e NSGAII. O método específico do TSuite alcançou o resultado de 23 casos selecionados, enquanto o NSGAII conseguiu 31, ambos em seu melhor caso. Isso acontece porque a variação no tamanho dos cenários selecionados conduz também ao aumento da quantidade dos casos selecionados, já que os casos de complexidade baixa consomem menor despesa do projeto.

Em resumo, o algoritmo de solução ao problema da Mochila faz a busca gradual dentro de todo o espaço de busca, pelo qual ele percorre somente uma vez. O algoritmo Aleatório não faz menção à restrição de espaço de busca nem de oscilação e por isso, suas soluções são bastante variadas, mas de pouca relação com objetivos específicos de um problema complexo. O algoritmo NSGAII faz a busca por soluções em um espaço restrito, mas o explora múltiplas vezes, o que confere em um resultado adequado a um objetivo modelado e variado, dentro de seus limites.

Mais uma vez a equipe Verde foi questionada a respeito do resultado do teste, pois ela participou da segunda parte do teste de validação do TSuite. A equipe relatou coerência entre o objetivo do problema passado para a seleção dos casos e resultado obtido através da aplicação do método NSGAII na seleção complementar no TSuite.

4. Conclusão e Trabalhos Futuros

O teste de *software* realiza o controle da qualidade do *software* desenvolvido. Ela contém atividades que custam caro porque despendem grandes esforços, inclusive de outras equipes (análise funcional, programação, modelagem), que possuam tarefas relacionadas, no projeto, e tempo elevado de execução. Qualquer tentativa que reduza o tempo de execução de suas atividades conduz à redução do custo de teste. Pensando nisso, o TSuite foi criado.

A seleção manual de casos de teste é uma tarefa que demanda um longo período de execução e análise. O principal objetivo do TSuite é automatizar tal atividade, considerando o estado atual do projeto de teste e seu impacto sobre suas versões anteriores, assim como o recursos disponíveis para a tarefa. Sua utilização é indicada aos projetos de teste que contenha muitos dados de execução manual e recursos restritos para a execução.

Testes provaram que a utilização do TSuite reduz o tempo de execução da atividade de seleção manual dos casos em, aproximadamente, 50%. Também, que sua execução independe da experiência do profissional de teste, assim como do conhecimento total das regras de negócio do sistema em teste por parte dele. A quantidade de casos selecionados também é ampliada, já que os recursos disponíveis ao projeto são melhor utilizados, por conta da aplicação do método probabilístico NSGAII, que aumenta esse resultado em 6,9%. Sua variedade é afetada, pois ao aplicá-lo, a quantidade de casos de complexidade baixa pode aumentá-la em até 16,7% e 55,5%, em relação aos métodos de solução do problema da Mochila e Aleatório, respectivamente.

Apesar dos benefícios propiciados pela utilização do TSuite na seleção dos casos de teste manuais, ele contém algumas limitações que serão abordadas no tópico seguinte.

4.0.3. Limitações da Ferramenta

Por ser uma ferramenta nova na seleção de casos de teste manuais, o TSuite atende somente aos dados de teste oriundos da ferramenta TestLink, já que essa é a ferramenta *open source* mais utilizada pela comunidade de teste. Assim, os passos de importação e exportação do TSuite estão voltados ao formato do arquivo que tal ferramenta suporta. Com isso, ela extrai somente arquivos do tipo XML. Além disso, tal arquivo deve conter anotações (*tags*) relacionadas a cada caso de teste para que a seleção seja realizada, obrigando os analistas de teste a manter os dados da seguinte forma, na ferramenta de gerenciamento de *testware*.

O TSuite atribui os pesos dos casos de teste conforme as *tags* associadas a cada um dos casos de teste. A seleção é limitada em 5 funcionalidades e o plano atual de execução do projeto deve possuir até duas suítes, condições que podem não satisfazer à situação do projeto.

A precedência dos casos de teste, como data de criação ou data da última execução, ou controle de versões não são considerados no momento da execução. Isso indica que o rastreio dos casos de teste que são selecionados no TSuite não estão de acordo com a gestão de defeitos de um plano já executado no sistema e portanto não avalia a mobilidade de defeitos em seus módulos já desenvolvidos.

A implementação de trabalhos futuros poderá solucionar tais limitações, adequando ainda mais o TSuite às necessidades do processo de teste.

4.0.4. Trabalhos Futuros

O TSuite busca solucionar o problema de seleção de casos de testes manuais, pois automatiza tal tarefa e disponibiliza um resultado que pode ser utilizado na ferramenta de gerenciamento de dados de teste, o TestLink. No entanto, seguem em aberto implementações necessárias à usabilidade do sistema.

Tais limitações podem ser resolvidas conforme desenvolvimento de versões futuras da ferramenta:

1. Possibilitar a leitura de dados contidos em arquivos de outros formatos, que não sejam xml, assim como em planilhas eletrônicas, conforme as extensões .xls e .ods, além da geração de saídas também nestes formatos;

2. Expandir a implementação do algoritmo de seleção para considerar a análise da cronologia entre os casos, a versão dos casos e a hierarquia entre as suítes do projeto, na precedência, com o intuito de descartar os casos desnecessários na seleção;

3. Aplicar a análise de risco sobre a gestão de defeitos do projeto para avaliar a cobertura de defeitos dos casos, o que pode tornar a seleção ainda mais próxima à condição real de execução dos casos de teste;

4. Reduzir o escopo da seleção a um método ou melhorar a variedade dos casos utilizando outro algoritmo genético, comparando-o com o NSGAII para isso.

5. Anexo

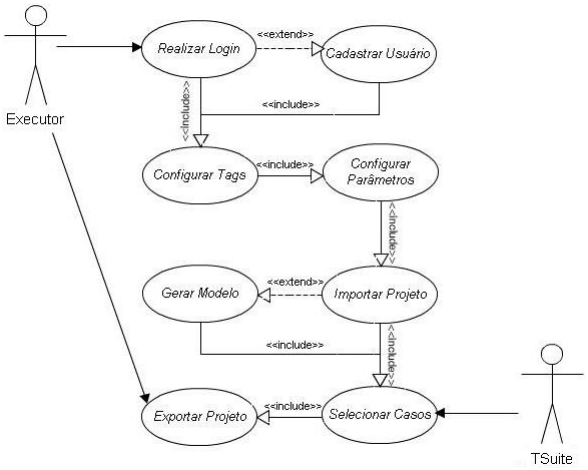

Figura 26: Diagrama de Caso de Uso do TSuite

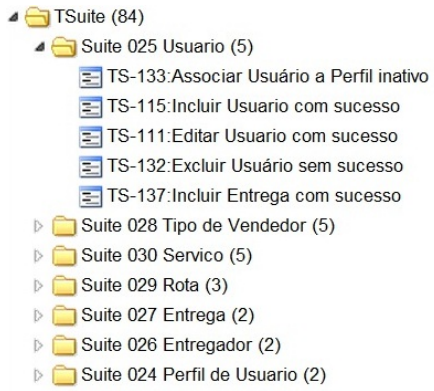

Figura 27: Estrutura em árvore do projeto de teste mantido na ferramenta TestLink

6. Referências

[1] R. Cristalli T. Moreira A. Bastos, e E. Rios. *Base de conhecimento em teste de software*. Martins, São Paulo, 2007.

[2] A. Bartié. *Garantia da Qualidade de Software: adquirindo maturidade organizacional*. Elsevier, Rio de Janeiro, 2002.

[3] R. Black. *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. John Wiley & Sons, New York, 2002.

[4] A. C. Molina C. A. C. Flórez y A. R. Bolaños. Algoritmo multiobjetivo nsga-ii aplicado al problema de la mochila. *Scientia Et. Technica*, XIV(39):206–211, 2008.

[5] F. Freitas G. de Campos C. Maia, R. do Carmo and J. de Souza. A multi-objective approach for the regression test case selection problem. *XLI Simpósio Brasileiro de Pesquisa Operacional (SBPO 2009)*, 2009.

[6] Teamst Community. User manual testlink version 1.9. *http://www.teamst.org*, jul 2012.

[7] F. de A. Farzat e M. de O. Barros. Método para seleção de casos de teste para alterações emergenciais. *I Workshop Brasileiro de Otimização em Engenharia de Software*, 2010.

[8] J. T. de Souza de F. G. Freitas, C. L. B. Maia e G. A. L. de Campos. Otimização em teste de software com aplicação de metaheurísticas. *Sistemas de Informações*, 2010(5), 2010.

[9] Pratap A. Agarwal S. Deb K. Associate Member, IEEE and Meyarivan T. A fast and elitist multiobjective genetic algorithm nsga-ii. *IEEE Transaction on Evolucionary Computation*, 6(5), 2002.

[10] N. D. Pizzolato e A. A. Gandolpho. *Técnicas e Otimização. 1. ed.* LTC, Rio de Janeiro, 2009.

[11] F. Deissenboeck M. Feilkas C. Schlogel E. Juergens, B. Hummel and A. Wubbeke. Regression test selection of manual system tests in practice. *Software Maintenance and Reengineering, European Conference on*, pages 309–312, 2011.

[12] R. Eliza e V. Lagares. *Além da IDE: Estimativa X Teste de Software*, volume ed.92. Java Magazine, 2011.

[13] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36:226–247, 2010.

[14] C. S. IEEE. Ieee standard for software and system test, documentation - standard 829. 2008.

[15] G. J. Myers. *The Art of Software Testing - 2nd edition*. John Wiley & Sons, Inc., 2004.

[16] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22, 1996.

[17] S. A. Shahid and M. Harman. Pareto efficiente multiobjective test case selection. *School of Physical Sciences and Engineering King's College London MSc Advanced Software Engineering*, 2007.

[18] Y. Lai b T. Huangc S. Chungd J. Hunga Y. Lina, C. Choua and F. C. Line. Test coverage optimization for large code problems. *The Journal of Systems and Software 85 (2012)*, page 16–27, 2012.

[19] C. Yih-farn and K. Vo D. S. Rosenblum. Testtube: A system for selective regression testing. In *In Proceedings of the 16th International Conference on Software Engineering*, pages 211–222, 1994.

[20] S. Yoo and M. Harman. *Regression testing minimization, selection and prioritization: a survey*. John Wiley & Sons, Ltd., 2010.

7. Apêndice

7.1. Resultado dos testes

O sistema apresentou o resultado exibido na figura 28 para o teste realizado pelas três equipes: Verde, Azul e Vermelho, ao utilizar a ferramenta TSuite. O resultado foi fixo pois a função selecionar() chamava a rotina específica do TSuite para realizar a seleção dos casos de teste.

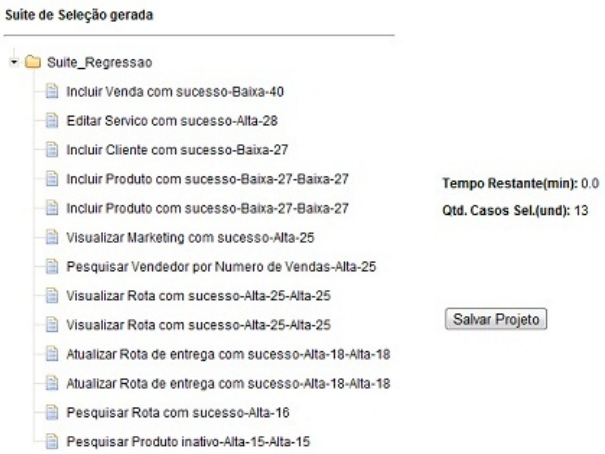

Figura 28: Tela Exportar Projeto: o resultado da validação do teste 1, com o método do TSuite

A figura 29 apresenta o resultado do melhor caso da seleção de casos de teste que utilizava o método probabilístico NSGAII como rotina auxiliar pelo método selecionar().

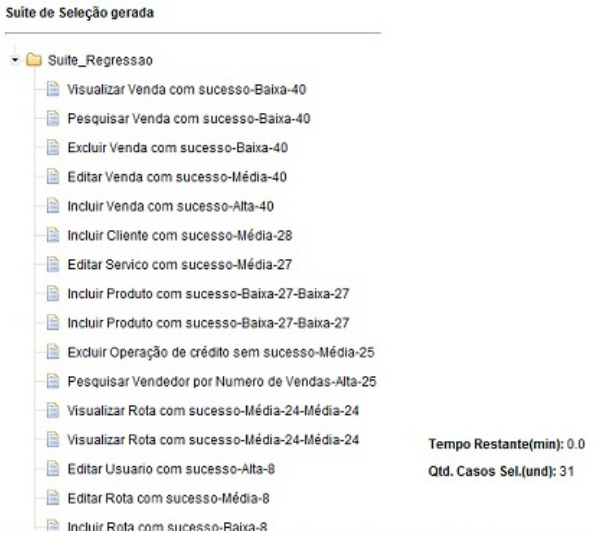

Figura 29: Tela Exportar Projeto: o resultado da validação do teste 2, com o método do NSGAII

7.2. Sistema utilizado na aplicação dos testes

Todos os testes realizados no TSuite levaram em consideração os relacionamentos dos módulos de um sistema hipotético(fictício) de gerenciamento de estabelecimentos comerciais, apresentados na tabela 6.

Suíte 1	**Tag da Suíte 1**	**Suíte 2**	**Tag da Suíte 2**
Venda:	[pesquisar]	Operação:	[excluir]
Venda:	[editar]	Marketing:	[visualizar]
Venda:	[editar]	Cliente:	[incluir]
Venda:	[incluir]	Entrega:	[visualizar]
Venda:	[excluir]	Serviço:	[editar]
Venda:	[visualizar]	Produto:	[incluir]
Venda:	[incluir]	Vendedor:	[pesquisar]
Entrega:	[editar]	Rota:	[pesquisar]
Produto:	[pesquisar]	Marketing:	[editar]

Tabela 6: Relacionamentos entre os módulos do sistema