UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO

DISSERTAÇÃO DE MESTRADO

UM MÉTODO AUTOMÁTICO PARA GERAÇÃO DE CRONOGRAMAS DE TAREFAS DE CORREÇÃO DE BUGS

Fernando de Castro Netto

Orientadores:

Prof.: Márcio de Oliveira Barros

Profa.: Adriana Cesário de Faria Alvim

Rio de Janeiro, abril de 2010.

UM MÉTODO AUTOMÁTICO PARA GERAÇÃO DE CRONOGRAMAS DE TAREFAS DE CORREÇÃO DE BUGS

Fernando de Castro Netto

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM INFORMÁTICA.

Aprovada por:

Prof. Márcio de Oliveira Barros, D.Sc.

Profa. Adriana Cesário de Faria Alvim, D.Sc.

Profa. Fernanda Araujo Baião, D.Sc.

Profa. Claudia Maria Lima Werner, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
ABRIL DE 2010

NETTO, FERNANDO DE CASTRO
Um Método Automático para Geração de Cronogramas de Tarefas de Correção de Bugs [Rio de Janeiro] 2010
90 p. 29,7 cm (PPGI/UNIRIO, M.Sc.,
Informática, 2010)
Dissertação - Universidade Federal do Estado
do Rio de Janeiro, PPGI
1. Engenharia de Software Baseada em Busca
2. Gerência de Projetos de Software
3. Manutenção de Software
I. PPGI/UNIRIO II. Título (série)

À minha amada esposa Marcelle.

Aos meus pais, meu irmão e meus sogros.

Aos demais familiares e amigos.

Agradecimentos

A Deus, pelas bençãos a mim concedidas durante toda minha vida, dando-me força, capacidade e condições para alcançar este objetivo.

À Marcelle, minha querida esposa, companheira nos momentos mais importantes da minha vida, por todo o apoio nestes anos, pelo amor, convívio, compreensão e ajuda, sem quem teria sido extremamente mais difícil realizar este trabalho.

Aos meus pais Marcos e Gloria, ao meu irmão Daniel, aos meus sogros Norberto e Elinea pelo incentivo, apoio, amizade e amor incondicional dedicados durante todas as etapas desta jornada.

Ao meu orientador Prof. Márcio Barros pelo excelente trabalho de orientação por acreditar em mim e no trabalho, por toda ajuda, todo incentivo, aprendizado e amizade ao longo destes anos.

À Profa. Adriana Alvim, igualmente pela ótima orientação, por todo o acompanhamento, pelos ensinamentos, críticas e sugestões que contribuíram muito para meu desenvolvimento e para o trabalho.

Aos demais professores do Programa de Pós Graduação em Informática da UNIRIO, pelo convívio, pelo apoio e pela compreensão, em especial às Professoras Renata Araujo e Fernanda Baião que possibilitaram o surgimento de tantas ideias utilizadas neste trabalho.

Aos colegas que partilharam comigo problemas e alegrias do dia a dia e tantos outros que colaboraram de forma especial neste árduo e prazeroso caminho de aprendizagem, convivência e construção.

Aos funcionários da secretaria, atenciosos e competentes, por sua paciência e dedicação aos nossos problemas administrativos e acadêmicos.

A todos os colegas com quem trabalhei durante esses anos, na RiskControl, no CEPEL e na Petrobras. Em especial agradeço às pessoas com quem convivi mais:

Márcio Lucena, João Paulo Pipa, Sérgio Jesus, Bruno Galvão, Victor Moitinho, Aldinei Bastos, Igor Pontes, Eduardo Netto, Márcio Julião e Gustavo Veronese.

Ao povo brasileiro por patrocinar o estudo nas instituições públicas.

Resumo da Dissertação apresentada ao PPGI/UNIRIO como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

UM MÉTODO AUTOMÁTICO PARA GERAÇÃO DE CRONOGRAMAS DE TAREFAS DE CORREÇÃO DE BUGS.

Fernando de Castro Netto

Abril/2010

Orientadores: Márcio de Oliveira Barros e Adriana Cesário de Faria Alvim

Projetos de software geralmente utilizam Bug Tracking Systems onde desenvolvedores e usuários finais podem registrar a ocorrência de bugs de um determinado software e consultar o andamento das respectivas correções. Os defeitos identificados devem ser corrigidos e novas versões do software contemplando os pacotes com as respectivas correções devem ser liberados. O gerente de projeto deve gerar o cronograma com o agendamento das tarefas de correção de bugs com diferentes prioridades, a fim de minimizar o prazo para realização destas tarefas e garantir que os bugs mais críticos foram corrigidos. Este problema é recorrente na maioria das organizações de software, e dado o elevado número de possíveis diferentes cronogramas, uma ferramenta automática para buscar bons cronogramas seria uma ajuda substancial aos gerentes de projeto. Este trabalho propõe um método que captura informações relevantes dos repositórios de bug e as submete a um algoritmo genético a fim de buscar uma configuração do agendamento das tarefas de correção de bugs próxima à solução ótima. Foram conduzidos testes, considerando uma amostra dos relatórios de bugs extraídos do Eclipse Bugzilla, para avaliar a proposta de solução e os resultados indicaram que os esquemas de agendamento sugeridos pelo método proposto apresentaram resultados expressivamente superiores quando comparados com os esquemas de agendamento efetivamente realizados.

Abstract of Dissertation presented to PPGI/UNIRIO as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

AN AUTOMATED METHOD FOR SCHEDULING BUG FIX TASKS.

Fernando de Castro Netto

April/2010

Advisors: Márcio de Oliveira Barros and Adriana Cesário de Faria Alvim

Software projects usually use Bug Tracking Systems where both developers and end users can report and track the resolution of software defects. These defects must be fixed and new versions of the software incorporating the patches that solve them must be released. The project manager must schedule a set of error correction tasks with different priorities in order to minimize the time required to accomplish these tasks and guarantee that the more important issues have been fixed. This problem is recurrent for most software organizations and, given the enormous number of distinct schedule, an automatically tool for finding good schedules may be helpful to project managers. This work proposes a method which captures relevant information from bug repositories and submits them to a genetic algorithm to find near optimal schedules. The evaluation approach used a subset of the Eclipse bug repository and the results suggested better schedules than the actual schedules followed by the developers.

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

INTRODUÇÃO

1.1. Motivação

Para Pressman (2006) o conceito de qualidade do software é segmentado em qualidade do processo de desenvolvimento do software, que avalia o nível de qualidade dos artefatos gerados durante o desenvolvimento do software (requisitos, modelos de análise e projeto, qualidade do código, documentação do sistema, etc.), e qualidade de conformidade, que mensura o grau com que as especificações de projeto são seguidas durante a fabricação. Para o usuário final, a qualidade está relacionada à capacidade do software satisfazer os requisitos e metas de desempenho especificados.

As atividades da gestão da qualidade são de responsabilidade da equipe de garantia da qualidade do software (Software Quality Assurance – SQA). Dentre as atividades executadas por esta equipe, destacam-se inspeções nos artefatos, testes de software e correções de defeitos, que ao longo deste trabalho serão comumente chamados de bugs.

Considerando que uma equipe de desenvolvimento de software experiente utilize as melhores práticas de garantia de qualidade de software no processo de desenvolvimento, ainda assim é esperado que os usuários finais encontrem defeitos no software que não foram previamente identificados durante as inspeções e testes realizados. O processo de gestão de correções de defeitos de software é uma atividade bastante recorrente nas organizações. De acordo com Boehm e Basili (2001), o custo destas correções pode consumir até 70% do orçamento previsto para o projeto de desenvolvimento do software.

Sistemas conhecidos como Bug Tracking Systems (BTS) são amplamente utilizados na indústria de software para facilitar a comunicação entre as partes envolvidas no processo de desenvolvimento (usuários finais, desenvolvedores, membros da equipe de garantia da qualidade e gerentes de projetos) e gerenciar o processo de correção dos defeitos do software (D'Ambros et al, 2007). Essas

ferramentas mantêm registros de bugs (anomalias no comportamento do software, que no contexto deste trabalho serão tratados como sinônimos de defeitos de software) e de solicitações de melhoria de um determinado conjunto de software. Cada registro de bug (conhecido como bug report) contém um conjunto de atributos que o caracterizam, tais como produto (o projeto de software), componente (o módulo onde foi observada a anomalia), versão (versão do software onde o bug foi identificado), prioridade (a urgência para correção), esforço (quantidade de pessoas-dia necessárias para correção) e descrição (texto com uma breve descrição do bug).

Quando um novo bug é identificado pelo usuário final, o gerente de projeto deve selecionar um desenvolvedor para solucionar o problema. Concluída a correção do bug, um membro da equipe de qualidade de software deverá efetuar um teste para verificar se o problema constatado foi resolvido. Periodicamente, novas versões do software, incorporando as correções dos bugs reportados, devem ser disponibilizadas.

Em sistemas de larga escala, o número de bugs reportados diariamente é muito elevado, aumentando a complexidade da alocação dos desenvolvedores nas correções e o sequenciamento das respectivas correções. Deve-se ressaltar que cada atividade de correção é caracterizada por uma prioridade e um esforço estimado (registrados no relatório de bug), enquanto que cada desenvolvedor é caracterizado por sua habilidade, identificada pelo histórico de correções e desenvolvimentos realizados, que está mantido na base de dados do BTS. Estas informações devem ser consideradas pelo gerente de projeto na elaboração do esquema de agendamento das tarefas de correção de bugs.

Assim, é possível identificar o seguinte problema de otimização: como alocar os desenvolvedores nas atividades de correção de bugs e sequenciar estas atividades de forma a minimizar o custo de correção dos bugs de um determinado software? Entende-se o custo como uma medida de qualidade que tende a balancear os interesses do fornecedor do software (maximizar o número de correções de bugs incorporadas na próxima versão do software) e dos clientes (priorizar as correções de bugs mais urgentes).

O problema acima, nomeado nesta dissertação como Problema de Distribuição e Sequenciamento de Correções de Bug (PDSCB), pode ser visto como uma variação do problema clássico de otimização combinatória da construção de cronograma (scheduling). A elaboração de um cronograma consiste na alocação de recursos a fim de executar um conjunto de tarefas com determinadas restrições (Hart et al, 2005). Algumas adaptações foram feitas no modelo do problema de distribuição e sequenciamento de tarefas de correção de bugs, a fim de torná-lo mais aderente ao contexto do processo de resolução de bugs. Dessa forma, podemos comparar as

correções de bugs com as tarefas que devem ser executadas, enquanto que cada recurso pode ser visto como um desenvolvedor.

Cabe ressaltar a complexidade envolvida na busca por um esquema de agendamento das tarefas de correção de bugs capaz de aumentar a eficiência do processo de resolução de bugs. Em primeiro lugar, é preciso categorizar os diversos bugs identificados em classes, em função do grau de similaridade entre as anomalias. Segundo, deve-se avaliar as habilidades de cada um dos desenvolvedores e mapear as respectivas capacidades de correção por classe de bug. Por fim, para encontrar a melhor solução deve-se avaliar todas as possíveis configurações de distribuição das correções de bugs entre os desenvolvedores e das respectivas sequências de execução.

O número de configurações possíveis para agendamento das tarefas de correção de bugs cresce exponencialmente em função da quantidade de bugs encontrados e dos desenvolvedores disponíveis, inviabilizando um procedimento do tipo força bruta para avaliar cada uma das configurações possíveis e escolher a melhor dentre todas. Assim, uma técnica heurística deve ser desenvolvida com o propósito de buscar soluções aproximadas para o problema exposto.

A motivação deste trabalho está fortemente inserida no contexto das grandes empresas, como é o caso da Petrobras. A criticidade deste problema é ainda maior para estas empresas, onde o número de novas tarefas de correção de bugs nas diversas aplicações utilizadas pela organização cresce expressivamente diariamente e há um contigente muito grande de equipes de desenvolvimento de software, em grande parte, geograficamente distribuídas.

Atualmente, os processos de levantamento das necessidades de manutenção nos sistemas de informação da Petrobras e priorização das respectivas tarefas utilizam instrumentos informais (planilhas e documentos textos). O desenvolvimento do plano de execução das manutenções é feito manualmente pelos líderes de projeto, com base nas estimativas de esforço de execução da cada tarefa de manutenção e nas respectivas prioridades. Considerando a complexidade do planejamento das manutenções nos sistemas de informação da Petrobras e a aplicação de técnicas artesanais para distribuição e sequenciamento das tarefas de manutenção, temos como resultados planos de execução ineficientes que trazem sérios prejuízos para a empresa, tais como o prolongamento na execução das manutenções de sistemas, o aumento no custo das equipes de tecnologia da informação (TI), a insatisfação das áreas de negócio (clientes das equipes de TI) e danos à imagem da empresa.

1.2. Objetivo da Dissertação

Métodos heurísticos são aplicados com bastante frequência nas áreas financeira, industrial e de engenharia com o objetivo de apoiar a melhoria dos processos destas áreas. Contudo, estas iniciativas ainda são recentes na área de Engenharia de Software. O termo Engenharia de Software Baseada em Busca (Search Based Software Engineering - SBSE) foi criado por Harman e Jones (2001) e consiste em uma abordagem dos problemas da Engenharia de Software sob a ótica de problemas de busca e otimização, com a aplicação de técnicas heurísticas para buscar soluções aproximadas. Desde então, houve um acentuado crescimento no número de trabalhos nesta linha de pesquisa, com o desenvolvimento de conferências e revistas técnicas específicas para o tema, conforme constatado pelos trabalhos de Clarke et al. (2003), Harman (2007) e Harman et al. (2009).

Inserido no contexto de SBSE, este trabalho aborda um problema de Engenharia de Software recorrente na maioria das organizações que desenvolvem ou mantém sistemas de software. O trabalho apresenta uma proposta de solução que consiste de um método que captura informações relevantes da base de dados de um BTS que, em conjunto com informações fornecidas pelo gerente de projeto, são transformadas em instâncias do PDSCB e submetidas a uma técnica de otimização baseada em um algoritmo genético associado com uma heurística construtiva, a fim de buscar uma configuração do esquema de agendamento próxima à solução ótima.

Com a aplicação da técnica sugerida espera-se um aumento na eficiência do processo de resolução de bugs, aprimorando o planejamento da liberação de versões futuras do software. Confirmada a existência de benefícios potenciais na solução proposta, entende-se que a técnica apresentada poderia ser implementada nos BTS, gerando ganhos substanciais para as organizações, especialmente no caso da Petrobras.

1.3. Estrutura do Texto

Este trabalho é composto por seis capítulos, incluindo esta introdução.

No capítulo 2, *Gerenciamento de Projetos de Software com Bug Tracking Systems*, aborda-se a utilização dos Bug Tracking Systems nos processos de desenvolvimento e manutenção de software, evidenciando os benefícios ofertados, a estrutura do relatório de bug e o processo de resolução de bug descrito através do ciclo de vida do relatório de bug. São apresentados estudos que vêm sendo realizados na área com o objetivo de automatizar as etapas do processo de resolução de bugs.

O capítulo 3, *Aplicação de Heurísticas para Solucionar Problemas de Otimização*, apresenta uma breve revisão bibliográfica sobre problemas de otimização combinatória, descrevendo as principais características destes problemas e as técnicas de resolução (métodos exatos e heurísticos). Por fim, é explorada a abordagem dos problemas de Engenharia de Software sob a ótica de problemas de otimização e busca, consolidando a área de SBSE.

No capítulo 4, *Distribuição e Sequenciamento das Correções de Bugs*, aborda-se o problema de distribuição e sequenciamento de tarefas de correção de bugs de diferentes prioridades para um conjunto de desenvolvedores disponíveis, a fim de atingir as aspirações dos clientes e dos fornecedores envolvidos no processo de resolução de bugs. O modelo do PDSCB é apresentado, bem como o processo de integração entre as variáveis de decisão do problema com as informações mantidas no Bug Tracking System e a proposta de solução baseada na aplicação de técnicas heurísticas.

No capítulo 5, *Avaliação da Solução Proposta*, são descritos os testes conduzidos para avaliar a eficácia da proposta apresentada nesta dissertação. Extraiu-se uma amostra dos relatórios de bugs armazenados no Eclipse Bugzilla, derivando desta forma as instâncias do PDSCB utilizadas nas análises. Além de verificar a eficácia da solução proposta, foram realizadas análises de sensibilidade nos parâmetros do método proposto que consistiram em analisar isoladamente cada um dos parâmetros com a finalidade de avaliar as influências destes na qualidade da solução sugerida e no tempo de resposta do processamento do algoritmo.

No capítulo 6, *Conclusão*, são apresentadas as considerações finais, contribuições e limitações do trabalho, além de perspectivas de trabalhos futuros.

CAPÍTULO 2

GERENCIAMENTO DE PROJETOS COM BUG TRACKING SYSTEMS

2.1. Introdução

A gestão da qualidade em Engenharia de Software tem como objetivo principal introduzir um conjunto de processos que garantam que o software produzido seja de alta qualidade: estruturas internas adequadas, funcionalidades em conformidade com os requisitos especificados e entrega dentro do orçamento e do cronograma previstos (Robert Glass, 1998). As atividades da gestão da qualidade são de responsabilidade da equipe de garantia da qualidade do software (Software Quality Assurance – SQA).

O objetivo do SQA é remover problemas que comprometam a qualidade do software desenvolvido. Há vários termos utilizados para referenciar estes problemas, tais como bugs, erros e defeitos. O termo “bug” é a expressão mais utilizada para referenciar anomalias encontradas no software. No contexto deste trabalho, um bug pode ser visto como um desvio na especificação observado no software, independente do estágio corrente no seu ciclo de vida. Para Pressman (2006), a distinção entre os termos “erro” e “defeito” está no momento em que o problema foi encontrado: se foi encontrado antes da entrega do produto para o usuário final, então é um erro; caso contrário, se foi reportado pelo usuário final, deve ser considerado um defeito.

A distinção entre os termos “erro” e “defeito” é pertinente sob a justificativa que estas anomalias têm impactos econômicos e no negócio muito diferentes. Segundo Boehm e Basili (2001), para softwares de larga escala, correções de defeitos de software são, em geral, 100 vezes mais custosas do que as correções de erros efetuadas durante as fases de análise dos requisitos e projeto. Além disso, dependendo dos impactos do defeito de software no negócio, é possível que haja solicitação de cancelamento do contrato, reembolso ou devolução do produto por parte do cliente (Karner, 1996).

Sendo assim, defeitos de software devem ser considerados fatores críticos para o sucesso do software desenvolvido. A comunidade de Engenharia de Software tem

buscado iniciativas para redução dos defeitos de software. A técnica mais recomendada é o uso das revisões técnicas formais, cujo objetivo é achar erros durante o desenvolvimento do software, evitando que eles se transformem em defeitos após a entrega do software. Estas revisões devem ser conduzidas ao longo de todas as atividades do ciclo de vida de desenvolvimento do software, servindo como filtro de erros para as atividades subsequentes e removendo erros enquanto eles ainda são relativamente baratos de corrigir e remover.

O conceito de software “zero-defeito” é algo bastante interessante e um ideal a ser buscado. Contudo, sob uma ótica realística é impossível garantir que um software está livre de defeitos, mesmo que as atividades de garantia da qualidade (planejamento, revisões técnicas e testes) tenham sido rigorosamente executadas. Portanto, é natural que os usuários finais identifiquem problemas durante a operação das aplicações. Neste caso, o cliente deve contactar o fornecedor do software, notificando-o da ocorrência destes problemas. Em contrapartida, o fornecedor deve avaliar a notificação e, confirmando a existência de um defeito de software, deve providenciar a correção do mesmo que será incorporada ao software em alguma versão futura.

Em alguns projetos de software, os clientes e os usuários finais trabalham em conjunto com a equipe de desenvolvimento e são responsáveis pela homologação e realização dos testes de aceitação durante o processo de desenvolvimento, ao passo que outros projetos fazem uma clara distinção entre a equipe de desenvolvimento e a equipe de testes. Contudo, em ambos os casos os participantes dos diferentes grupos (usuários, clientes, desenvolvedores e testadores) podem encontrar anomalias no comportamento do software que devem ser comunicadas para as demais partes interessadas.

Ferramentas conhecidas como Bug Tracking System (BTS) são utilizadas para facilitar a comunicação entre as partes envolvidas no processo de correção de bugs. Grandes projetos de software frequentemente utilizam algum BTS, onde desenvolvedores, membros da equipe de SQA e usuários finais podem reportar e consultar a resolução dos bugs de um determinado software. Além das anomalias, estas ferramentas também permitem o registro de solicitações para inclusão de novas funcionalidades no software.

Este capítulo está dividido em sete seções. A Seção 2.1 consiste desta introdução. A Seção 2.2 apresenta o BTS como ferramenta para gerenciar o processo de resolução de bugs. A Seção 2.3 descreve as informações mantidas no relatório do bug (*bug report*), que é o registro armazenado na base de dados do BTS. Na Seção 2.4, descrevemos o processo de resolução dos bugs, evidenciando o ciclo de vida do

relatório de bug. Na Seção 2.5 destacamos alguns problemas recorrentes no processo de resolução dos bugs. A Seção 2.6 destaca algumas tendências de inovação nos BTS, como forma de suprir as carências do processo de resolução de bugs. Por fim, na Seção 2.7 descrevemos alguns trabalhos relacionados ao tema de uso dos BTS no processo de desenvolvimento de software.

2.2. Visão Geral do Bug Tracking System

O principal componente de um BTS é a base de dados que mantém os registros dos bugs e das solicitações de mudança reportadas. Dentre as informações mantidas em um registro, é possível encontrar o dia e a hora da criação do registro, a severidade do problema, a prioridade para que a tarefa seja executada, a descrição do problema e o contato da pessoa que fez a solicitação.

BTS suportam o conceito de ciclo de vida dos registros de problemas, onde a evolução no tratamento de cada problema pode ser monitorada por um campo que informa o estágio corrente do processo de tratamento do problema. Estes sistemas devem permitir ao administrador configurar perfis de acesso dos usuários, mover o registro do problema ao longo do ciclo de vida e remover registros de problemas incorretamente cadastrados. O sistema também deve permitir ao administrador configurar os estágios do ciclo de vida do problema, assim como definir o fluxo de mudança entre os estágios.

Dentre as vantagens ofertadas pelos BTS, deve-se destacar o suporte à comunicação entre as diversas partes envolvidas no processo de desenvolvimento de software, que podem estar geograficamente distribuídas. Tanto usuários finais, clientes, desenvolvedores, membros da equipe de SQA, como gerentes de projeto podem acompanhar a evolução no tratamento dos bugs e das solicitações de melhorias encaminhadas para um determinado software. Além disso, o constante uso desta ferramenta agrega na melhoria da qualidade do software, visto que aumenta a percepção de todos envolvidos das anomalias e das carências existentes, permitindo que as respectivas resoluções sejam executadas (Raymond, 1998). A análise dos registros de bugs também permite a identificação dos módulos mais problemáticos do software (D'Ambros et al, 2007) e, consequentemente, permite que medidas preventivas sejam adotadas para antecipar a detecção de defeitos por parte da equipe de desenvolvedores, antes que os usuários finais percebam, reduzindo o custo com manutenções corretivas. Anvik et al (2005) ressaltam que a utilização dos BTS é capaz de apoiar a evolução do software em conformidade com os interesses dos

usuários finais, uma vez que as versões subsequentes do software devem incorporar correções de bugs e novas funcionalidades solicitadas por estes usuários.

Bugzilla[1] é um dos BTS mais conhecidos, desenvolvido inicialmente para gerenciar os bugs reportados para o navegador Mozilla[2] e que atualmente está sendo largamente utilizado pela indústria de software. Trata-se de um software livre, desenvolvido para operar utilizando arquitetura Web. Serrano e Ciordia (2005) caracterizam o Bugzilla como um sistema de informação clássico, cuja função principal é manter uma base de dados para registro e consulta de dados sobre bugs. As principais funcionalidades ofertadas pelo sistema são: consultar registros de bugs reportados, reportar um novo bug ou uma nova solicitação de mudança e emitir relatórios e gráficos de acompanhamento.

JIRA[3] é outro BTS bastante utilizado na indústria de software. Este sistema é desenvolvido pela empresa Atlassian Software Systems[4] e sua licença é gratuita somente para fins não comerciais. JIRA oferece suporte à análise multi-dimensional, com uma imensa variedade de formatos de relatórios multi-dimensionais, incluindo gráficos de pizza, lineares, em coluna e tabelas estatísticas. Este sistema permite a análise de uma série de informações, tais como o quantitativo de bugs reportados por cliente, o número de bugs reportados ao longo de um determinado período e a média de tempo necessária para resolver um bug. Também é possível a construção de um *dashboard* para acompanhamento das atividades de correção de bugs.

GNATS[5] é o BTS do projeto GNU e, de forma análoga ao Bugzilla e ao JIRA, consiste de um conjunto de ferramentas para monitoramento de bugs reportados para um determinado conjunto de projetos de software. Além da interface Web para interação com os usuários finais, esta ferramenta também provê interfaces de comunicação através de linhas de comando e e-mails. Comparando o GNATS com o Bugzilla e o JIRA, percebe-se que o GNATS é um sistema menos utilizado na indústria de software e a sua versão mais recente foi disponibilizada em março de 2005.

2.3. Anatomia de um Relatório de Bug

Cada registro armazenado na base de dados de um BTS é conhecido como relatório de bug e consiste de um formulário composto por um conjunto de atributos que o

[1] http://www.bugzilla.org
[2] http://www.mozilla.org
[3] http://www.atlasian.com/software/jira
[4] http://www.atlasian.com
[5] http://www.gnu.org/software/gnats

caracterizam. A Figura 2.1 exibe um exemplo deste formulário descrevendo um bug registrado para um dos projetos de software da comunidade de software livre Eclipse[6].

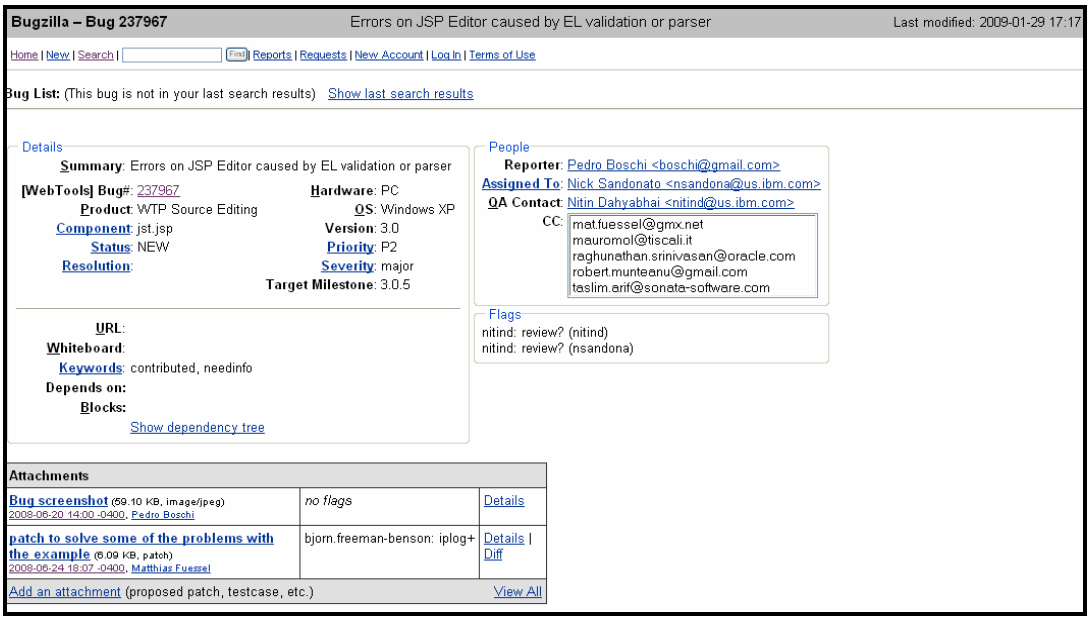

Figura 2.1. Dados Cadastrais de um Relatório de Bug

Os campos de um relatório de bug podem ser categorizados de acordo com a forma de preenchimento (campos pré-definidos, descrições textuais livres e anexos) e quanto às alterações ao longo do tempo (invariantes e variantes no tempo).

Os campos pré-definidos são categorizados em duas classes: fixos e editáveis. Os campos pré-definidos fixos são aqueles que mantêm valores fixos automaticamente gerados pelo formulário de preenchimento, tais como o número de identificação do relatório de bug, a data de criação do relatório de bug e o login do usuário que reportou o defeito. Os campos editáveis oferecem um conjunto de opções delimitado por um domínio de valores que podem ser atribuídos a um determinado atributo. Os campos produto (nome do software), componente (nome do componente onde a anomalia foi observada), versão do produto de software, versão do sistema operacional, prioridade para correção do defeito e severidade do impacto causado pela existência deste defeito são exemplos de campos pré-definidos e editáveis.

Os campos textuais livres incluem o título do relatório de bug, uma descrição completa do defeito de software e comentários adicionais do relatório de bug, registrados por usuários do BTS. A descrição completa geralmente contém informações detalhadas da anomalia, como a enumeração dos efeitos adversos

[6] http://www.eclipse.org

gerados pela existência deste defeito e um guia para reproduzir o defeito no sistema. Os comentários adicionais servem como um fórum de discussão entre os diversos usuários do sistema para prover um melhor entendimento do problema, identificar possíveis semelhanças com outros problemas e buscar soluções para corrigi-lo.

Os usuários relatores dos bugs geralmente anexam ao relatório de bug telas do sistema com detalhes do comportamento indesejado gerado em função da existência do defeito reportado.

Alguns atributos têm seus valores alterados com frequência ao longo do tempo, tais como o desenvolvedor atribuído para correção do defeito de software, o status corrente do processo de correção, o tipo de resolução atribuída ao relatório de bug e a prioridade da resolução. Por outro lado, há atributos que mantêm seus valores invariantes no tempo, tais como o produto e o componente de software associados. As alterações nos valores dos atributos do relatório de bug são registradas no log das atividades que são executadas ao longo do processo de correção de defeitos de software. A partir da leitura deste log é possível mapear todo o histórico do relatório de bug, desde o registro do problema até a conclusão do processo, identificando os desenvolvedores que participaram da resolução e a duração das respetivas atividades.

Os relatórios de bug são categorizados em produtos (Product) e componentes (Component). Cada produto representa um projeto de software e pode conter um ou mais componentes, que representam os diversos módulos de um software. No exemplo da Figura 2.1, o relatório de bug foi associado ao componente “jsp.jsp” do software “Eclipse Web Tools Platform Source Editing”.

Os campos Hardware e OS representam, respectivamente, a arquitetura computacional e o sistema operacional do ambiente onde foi encontrado o bug pelo usuário final. O item Version identifica a versão do software onde foi observado o bug.

Os campos Reporter, Assigned to e QA Contact contêm, respectivamente, o nome e o e-mail para contato da pessoa que identificou o bug, do responsável técnico em prover a correção para o bug e do membro da equipe de SQA que deve validar a correção que será implementada. O campo CC List contém a lista das pessoas interessadas em acompanhar o processo de tratamento do bug em questão.

O campo Priority estabelece a prioridade para correção do bug. Há cinco níveis de prioridade possíveis: P1 (mais prioritário), P2, P3, P4 e P5 (menos prioritário). O campo Severity informa o impacto da ocorrência do bug no software. Há cinco tipos de impacto previstos. Blocker é o impacto mais indesejado, pois impede a continuidade das atividades de desenvolvimento e teste. Critical é usado quando o bug causa algum tipo de parada de sistema, com perda de informação. Um bug é considerado Major quando compromete o comportamento das funções críticas do sistema. Um bug Minor

compromete apenas funções não críticas do sistema. Bugs do tipo Trivial fazem referência a problemas de menor impacto, tais como erros de ortografia e alinhamento de textos.

Além dos campos padrão, o BugZilla permite ao administrador do sistema incluir campos adicionais no template do relatório de bug. As versões mais recentes do Bugzilla incluem no template padrão um campo para armazenar o esforço estimado para correção de um determinado bug. Muitas organizações têm utilizado este recurso como forma de registrar o esforço previsto para as manutenções corretivas e subsidiar o planejamento de liberação de versões (Weiss et al, 2007).

2.4. Processo de Resolução de Bugs

Bugzilla mantém a rastreabilidade do processo de resolução de bugs através do ciclo de vida do relatório de bug, desde o momento da identificação até o término com a resolução atribuída, incluindo a alocação do desenvolvedor e a construção do pacote que inclui a correção do bug conhecida como *bug fix* (Netto et al, 2009). O ciclo de vida do bug está representado na Figura 2.2 e as transições entre os estágios deste ciclo são representadas como alterações no campo Status do relatório de bug.

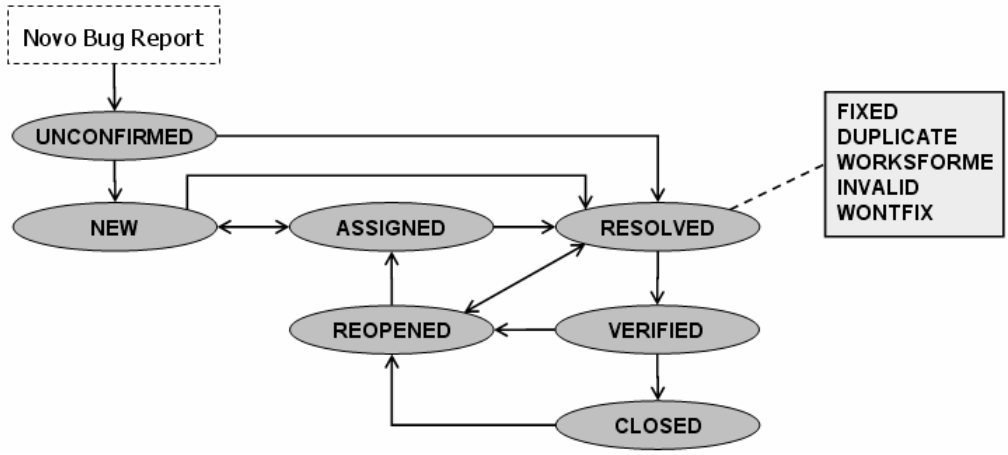

Figura 2.2. Ciclo de Vida de um Relatório de Bug no Bugzilla

Este processo é iniciado sempre que um usuário final ou um membro da equipe de testes encontra uma anomalia no comportamento do software e cria um novo relatório de bug no Bugzilla. Neste momento o relatório de bug permanece no estágio UNCONFIRMED. Periodicamente, ocorre a triagem dos novos relatórios de bug, onde membros da equipe de SQA e desenvolvedores avaliam se os novos registros podem ser classificados como defeitos de software (Anvik e Murphy, 2007). Em caso afirmativo, os registros são migrados para o estágio NEW, ou RESOLVED

caso contrário. Para todo relatório de bug “validado” deve-se estimar o esforço necessário para sua correção, preenchendo o campo correspondente no relatório de bug.

O nível de prioridade atribuído pelo usuário final também pode ser revisto neste momento. A comunidade de software livre Eclipse optou por permitir que usuários finais da aplicação elejam os relatórios de bug mais prioritários a partir do recurso do voto. Quanto mais votos um determinado relatório de bug receber, mais prioritária será a sua correção.

Quando um desenvolvedor é alocado para tratar o defeito reportado, o status do relatório de bug muda para ASSIGNED. Depois que o desenvolvedor providenciou a solução, o registro migra para o estado RESOLVED. Depois que a solução é validada pela equipe de SQA, o registro migra para o estágio VERIFIED e, consequentemente, para CLOSED quando o bug-fix é disponibilizado para o usuário final. Se a solução não for satisfatória, o registro é encaminhado para o estágio REOPENED e, em seguida, retorna o para o estágio ASSIGNED. Qualquer relatório de bug cujo *bug fix* já tenha sido disponibilizado para o usuário final, isto é, que esteja no estado CLOSED, sempre poderá retornar para o estágio REOPENED mais tarde por um usuário final insatisfeito ou por algum membro da equipe de SQA. Conforme observado na Figura 2.2, existem 5 tipos de resolução de bugs: FIXED (quando a correção implica na alteração do código-fonte), DUPLICATE (trata-se de um registro duplicado de outro bug), WORKSFORME (não foi possível reproduzir o bug), INVALID (não se trata de um defeito de software) e WONTFIX (o problema não será corrigido). As versões mais recentes do BugZilla aceitam customizações neste workflow.

2.5. Problemas Decorrentes do Uso do Bug Tracking System

Apesar dos benefícios destacados, o uso do BTSs também produz alguns efeitos colaterais. Dentre os problemas mais comuns destacam-se a dificuldade na triagem dos relatórios de bug, a qualidade do preenchimento dos relatórios de bug e a complexidade da distribuição das tarefas de correção entre os desenvolvedores.

De acordo com Anvik e Murphy (2007), a demanda de triagem de relatórios de bug submetidos pelos usuários finais normalmente supera a quantidade de desenvolvedores disponíveis nas organizações, gerando alguns efeitos indesejáveis. Em primeiro lugar, muito do esforço necessário na análise dos relatórios de bug para avaliar a existência de um problema conforme notificado pelo usuário final é em vão. A Tabela 2.1 e a Tabela 2.2 apresentam as distribuições dos tipos de resolução aplicados nas correções de bugs reportados até dezembro/2008 para os projetos de

software do Eclipse e do Mozilla, respectivamente. De posse dos dados apresentados nestas tabelas, observa-se que apenas 61% e 34% dos bugs do Eclipse e do Mozilla, respectivamente, foram de fato corrigidos. Por outro lado, 39% e 66% do tempo gasto na triagem foram desperdiçados na análise de relatórios de bug que retratavam problemas duplicados, solicitações inválidas ou descrições de bugs que não foram reproduzidos ou que são negligenciáveis. Se usarmos como referência que a triagem de um relatório de bug leva em média 5 minutos, então mais de 32.000 h foram gastas em atividades que em nada melhoraram a qualidade dos softwares do projeto Mozilla.

Tabela 2.1. Distribuição dos Tipos de Resolução Aplicados nos Projetos do Eclipse

RESOLUÇÃO	TOTAL	%
FIXED	131.842	60,99%
INVALID	14.247	6,59%
WONTFIX	14.375	6,65%
LATER	4.512	2,09%
REMIND	1.220	0,56%
DUPLICATE	31.725	14,68%
WORKSFORME	16.949	7,84%
NOT_ECLIPSE	1.305	0,60%
TOTAL	**216.175**	**100%**

Tabela 2.2. Distribuição dos Tipos de Resolução Aplicados nos Projetos do Mozilla

RESOLUÇÃO	TOTAL	%
FIXED	132.738	34,19%
INVALID	40.839	10,52%
WONTFIX	14.182	3,65%
DUPLACTE	123.489	31,81%
WORKSFORME	62.058	15,99%
INCOMPLETE	5.599	1,44%
EXPIRED	9.180	2,36%
MOVED	101	0,03%
TOTAL	**388.186**	**100%**

Outro efeito indesejável, existente devido ao número de relatórios de bug superar a quantidade de pessoas disponíveis para avaliá-los, implica no tratamento indevido com anomalias urgentes reportadas. Durante o processo de triagem, a prioridade de correção de cada bug definida pelo usuário final é revista e o esforço para sua correção deve ser estimado. Relatórios de bug de prioridade mais alta devem ser imediatamente encaminhados para correção. Assim, as pessoas encarregadas da triagem devem avaliar todos os relatórios de bug e a ordem de seleção dos registros para análise pode fazer com que relatórios de bug mais prioritários sejam avaliados por último, consequentemente aumentando o tempo das respectivas correções.

Just et al (2008), Hooimeijer e Weimer (2007) e Bettenburg et al (2008) identificaram problemas na qualidade dos relatórios de bug submetidos pelos usuários finais. Para Just et al (2008), o preenchimento do formulário de um relatório de bug exige um conhecimento técnico que muitas vezes está além do nível de conhecimento do usuário final. Por exemplo, é comum o usuário final desconhecer o nome do componente responsável pela funcionalidade para a qual registrou a ocorrência de um determinado erro. Contudo, o preenchimento desta informação é obrigatória no formulário do relatório de bug.

Além disso, Bettenburg et al (2008) argumentam que há uma grande lacuna entre o que os usuários finais registram na descrição e o que realmente os desenvolvedores esperam que eles descrevam, fazendo com que o processo de triagem dos relatórios de bug seja ainda mais complicado. Por exemplo, os usuários finais, em geral não reportam ou descrevem incorretamente os cenários de reprodução das anomalias descritas, que são justamente as informações mais importantes sob o ponto de vista dos desenvolvedores. Esta observação é reforçada pelo elevado índice de resoluções do tipo WORKSFORME registrados nos projetos do Eclipse e Mozilla, conforme consta na Tabela 2.1 e na Tabela 2.2, respectivamente. Os BTSs não facilitam a captura de informações relevantes, tais como o log da trilha de execução até a ocorrência do problema. A ausência de templates para descrição dos problemas permite a submissão de descrições muito abertas e pouco úteis para a análise dos desenvolvedores.

O problema de distribuição das tarefas de correção foi discutido por Anvik et al (2006). No encerramento do processo de triagem é preciso definir qual desenvolvedor será responsável pela correção de um determinado bug. Trata-se de uma tarefa bastante árdua, dado que o universo de desenvolvedores registrado em um BTS pode ser muito elevado. Assim, é preciso identificar qual dentre estes desenvolvedores deve ser alocado, de forma que a pessoa escolhida tenha o expertise necessário para executar a tarefa. O mapeamento das habilidades dos desenvolvedores consiste em agrupar os bugs em grupos de tarefas e estabelecer qual o sub-conjunto de desenvolvedores mais apropriado para corrigir bugs de cada grupo de tarefas.

2.6. Tendências de Inovação no Bug Tracking System

Muitos estudos na área de análise de BTS sugerem a aplicação de técnicas de mineração de dados na base de dados dos relatórios de bug como forma de obter informações capazes de automatizar etapas do processo de resolução de bugs. Estes

estudos indicam oportunidades para redução de custos do processo de triagem, possibilidade de mapeamento automático dos desenvolvedores mais capacitados para corrigir um novo bug notificado e extração de informações a partir de análise de dados históricos capazes de subsidiar o planejamento da liberação das versões futuras do software. Há também trabalhos no sentido de mensurar e melhorar a qualidade do preenchimento dos relatórios de bug, de forma a facilitar a análise dos problemas notificados. Esta seção apresenta um breve resumo dos principais trabalhos relacionados às tendências de inovação nos BTSs.

Anvik et al (2005) fazem uma ampla descrição dos BTS, evidenciando como estes sistemas ajudam o processo de resolução de bugs e proveem a melhoria contínua da qualidade dos projetos de software. Os repositórios de bugs dos projetos Eclipse e Mozilla foram utilizados como exemplos para destacar as principais características dos BTS e permitiram a observação dos seguintes fatos: (i) 42% e 56% dos relatórios de bug dos projetos Eclipse e Firefox, respectivamente, correspondem a notificações irrelevantes, inválidos ou duplicados; (ii) o número médio de bugs reportados diariamente aumentou nos períodos próximos a liberação de novas versões do software; (iii) no projeto Eclipse há uma grande concentração de e-mails com domínio da IBM por parte dos desenvolvedores que resolveram a maior parte dos bugs; (iv) em ambos os projetos mais de 70% dos bugs foram resolvidos em menos de um mês. De posse dos fatos observados, percebe-se os impactos no processo de triagem, onde membros da equipe de SQA e desenvolvedores gastam parte do tempo analisando relatórios de bug irrelevantes, inválidos ou duplicados, reduzindo desta forma a eficiência do processo. Por fim, os autores defendem o uso de técnicas de aprendizagem automática (*machine learning*) para desenvolver ferramentas capazes detectar automaticamente a existência de bugs duplicados e reduzir os custos do processo de triagem de bugs.

Anvik et al (2006) analisam o problema de alocação de desenvolvedores para as tarefas de correção de bugs. A proposta apresentada pelos autores consiste em analisar uma base de dados de bugs usando um algoritmo de aprendizagem automática para identificar os tipos de bugs que cada desenvolvedor costuma corrigir, identificando desta forma as expertises de cada desenvolvedor na correção de bugs. Os desenvolvedores deveriam ser alocados para resolver novos bugs que sejam associados a categorias de bugs as quais eles estão habilitados a corrigir. Neste trabalho, cada relatório de bug da base de dados é tratado como um documento texto, e após a aplicação de técnicas de mineração de textos (*text mining*) na base de dados, foram definidas categorias de bugs que mapeavam os tipos de problemas mais recorrentes. Cada relatório de bug estava associado a uma categoria de bug

específica. Dado um novo relatório de bug inserido na base de dados, a técnica identifica a categoria de bug mais adequada e recomenda uma lista de desenvolvedores mais apropriados para correção em função do histórico de correções de bugs deste mesma categoria. Os pesquisadores avaliaram a solução proposta utilizando como referência a base de dados de bugs dos projetos Eclipse, Firefox e GCC[7], onde os testes registraram níveis de precisão (índices de assertividade entre as indicações e as seleções realizadas) de 86%, 64% e 6%, respectivamente. As amplitudes dos resultados obtidos denotam que a aplicabilidade desta técnica depende de como as equipes são estruturadas dentro das organizações e como as tarefas são distribuídas entre os membros das equipes de desenvolvimento. Anvik e Murphy (2007) é uma evolução deste trabalho e consiste em introduzir um processo automático de triagem de relatórios de bug que precede a distribuição das tarefas entre os desenvolvedores.

Hooimejer e Weimer (2007) analisam a dificuldade no processo de triagem dos relatórios de bug. O objetivo principal deste trabalho é prover uma técnica capaz de reduzir os custos da triagem dos relatórios de bug, através de um filtro automático que remove da fila de triagem os relatórios de bug que seriam mais difíceis de analisar. Para avaliar o grau de dificuldade em fazer a triagem de um determinado relatório de bug, este trabalho propõe um modelo de análise da qualidade do relatório que considera os seguintes fatores: (i) severidade do problema (quanto menor a severidade do problema, maior a chance de ser descartado); (ii) compressibilidade na leitura do texto descritivo do relatório de bug (quanto mais difícil for a leitura da descrição do bug, mais difícil será a análise do relatório de bug); (iii) reputação do usuário reportador (quanto pior for a reputação do usuário, maiores as chances de ignorar o relatório de bug); (iv) mudanças na severidade e na prioridade (quanto maior o número de alterações nestes itens, mais indefinida fica a análise deste bug); (v) número de comentários adicionais (quanto maior o número de comentários registrados por outros usuários, mais importante e mais fácil será a análise do problema). Por fim, este modelo tende a classificar cada novo relatório de bug em “caro” ou “barato”, de acordo com a dificuldade na sua análise. A técnica proposta sugere que todos os relatórios de bug classificados como “caros” devem ser ignorados, reduzindo desta forma o custo do processo de triagem. Esta proposta talvez seja interessante para sistemas de informação de criticidade moderada, mas para sistemas críticos, tais como sistemas de controle do tráfego aéreo, nenhuma anomalia reportada por um usuário final deve ser ignorada, tendo em vista o impacto catastrófico que um erro

[7] http://www.gcc.gnu.org

pode causar. Ao invés de usar esta técnica como instrumento de seleção dos relatórios de bug que serão analisados, talvez seja mais pertinente usá-la como modelo de priorização de análise das solicitações de anomalias reportadas.

Wang et al (2008) apresentam uma técnica para verificar se novos bugs reportados podem ser classificados como registros duplicados de outros bugs previamente cadastrados na base. Para cada registro de bug, são capturados as descrições textuais e o registro do log da aplicação contendo a lista de mensagens registradas pela aplicação desde a inicialização até a ocorrência da anomalia em destaque. Cada um destes atributos é tratado como um documento texto que, a partir de técnicas de processamento de linguagem natural (NLP – Natural Language Processing), é mapeado em um vetor de palavras chave. Dado um novo registro de bug, são calculados os índices de similaridade do novo registro com cada um dos demais bugs cadastrados. Caso algum dos resultados calculados seja igual ou superior a um limite pré-definido, então o novo bug será considerado como registro duplicado. Os experimentos conduzidos utilizando como referência os bugs do Eclipse e Firefox apresentaram níveis de precisão de 67% e 93%, respectivamente. Esta abordagem exige que nos relatórios de bug sejam anexados os logs de execução das aplicações. Dessa forma, teremos um custo adicional nos servidores referente ao espaço em disco para armazenamento destes arquivos. Tendo em vista que o tamanho destes textos pode ser muito extenso, o desempenho dos algoritmos de mapeamento dos vetores das palavras chave e análise dos indicadores de similaridade pode ser comprometido. Além disso, a análise de similaridade entre os bugs não leva em consideração diversos atributos relevantes do relatório de bug como, por exemplo, o nome do componente. Sendo assim, existe o risco de dois bugs de componentes distintos (ou seja, sem similaridade entre eles) serem considerados duplicados.

Panjer (2007) propõe o uso de técnicas de mineração de dados para buscar estimativas do esforço necessário para a correção de bugs. Nos experimentos realizados, foram considerados 118.371 registros de bugs reportados para o Eclipse de outubro/2001 a maio/2006. Foi utilizado o software Weka[8] para aplicar os seguintes algoritmos de classificação: 0-R, 1-R, Naive Bayesian, Árvores de Decisão e Regressão Logística. Os resultados destes experimentos indicaram um critério de classificação capaz de estimar o tempo necessário para correção dos bugs com um índice de precisão de 34,9%. O autor observou que o número de comentários e a severidade do bug são os atributos que mais influenciam no tempo de correção dos bugs. Dependendo da combinação de valores destes atributos, o tempo de resolução

pode variar de alguns dias a meses. O índice de precisão alcançado nos experimentos realizados não é satisfatório e, com base neste índice, não se pode afirmar a viabilidade da proposta apresentada. Uma análise mais criteriosa poderia agrupar os relatórios de bug conforme proposto por Anvik (2006). Cada um desses grupos de bugs apresenta características específicas, tais como o número de desenvolvedores que atuam na correção de bugs, a taxa média de surgimento de novos bugs e a complexidade das modificações inerente ao componente. Todos estes fatores podem contribuir para o tempo de correção do bug e uma análise de estimativa da duração desta atividade deveria levá-los em consideração. O trabalho descrito considerou todo o universo de bugs reportados para todos os projetos e componentes de software do Eclipse, sem qualquer distinção. Talvez esta premissa também tenha contribuído para o baixo índice de precisão de 39,4%.

Weiss et al (2007) também descrevem uma técnica para prever o esforço necessário para corrigir bugs a partir da aplicação de técnicas de mineração de dados na base de dados de bugs. Dado um novo relatório de bug, o método realiza uma busca na base de dados de bugs e captura os k registros mais similares ao novo bug que já foram corrigidos. O esforço para correção é estimado a partir da média do tempo de correção dos k bugs similares recuperados. Utilizou-se a técnica de similaridade entre textos (*text similarity*), onde os campos textuais com as descrições dos defeitos foram comparados para definir quais seriam os k bugs mais semelhantes ao novo bug. A proposta foi avaliada utilizando como referência a base de dados de defeitos de software do projeto JBoss[9], onde os resultados revelaram que com um grande número de bugs registrados é possível fazer previsões de esforço próximas do realizado. O impacto da inclusão de novos atributos na análise da similaridade entre os bugs deve ser avaliado. Há indícios de que existe dependência entre o tempo de resolução do bug e alguns atributos do relatório de bug, como por exemplo a quantidade de comentários adicionadas por outros usuários e a severidade do bug, conforme visto por Panjer (2007).

Just et al (2008) apresentam os resultados obtidos em um survey cujo o objetivo era identificar quais informações eram reportadas com frequência nos relatórios de bug e quais eram as principais dificuldades no processo de resolução de bugs. Os questionários foram encaminhados para 872 desenvolvedores de software e para 1.354 usuários reportadores de bugs, dos quais 156 e 310 responderam, respectivamente. Os resultados obtidos nesta pesquisa indicaram uma necessidade de aprimoramento dos BTSs para auxiliar os reportadores de bugs a coletar as

[8] http://www.cs.waikato.ac.nz/ml/weka/
[9] http://www.jboss.org

informações pertinentes para os desenvolvedores de software. Os autores receberam comentários dos participantes do survey, que mapearam as características de um bom relatório de bug e como as ferramentas de BTSs poderiam ser aprimoradas. Assim, este trabalho sugere uma série de mudanças na ferramenta e no processo de resolução de bugs, de forma a torná-lo menos custoso: (i) utilização de ferramenta auxiliar para capturar os passos necessários para o desenvolvedor reproduzir o bug adequadamente; (ii) encontrar voluntários para traduzir os relatórios de bug escritos em outros idiomas; (iii) prover dicas para os usuários finais menos experientes, ajudando-os a identificar as informações mais pertinentes para os desenvolvedores; (iv) recompensar os usuários finais quando estes contribuírem para a descoberta de bugs relevantes; (v) classificar os usuários finais quanto à reputação na notificação de bugs, em função da quantidade de bugs reportados e das respectivas contribuições; (vi) refinar a ferramenta de busca por bugs, evitando o excessivo número de bugs duplicados; e (vii) incentivar os usuários finais a inserir informações adicionais nos relatórios de bug existentes.

Bettenburg et al (2008) analisam o problema da qualidade dos bugs. A qualidade de um relatório de bug é uma medida que avalia se o relatório de bug foi devidamente preenchido e se o mesmo contém todas as informações relevantes para a análise do bug, sob a ótica do desenvolvedor. Este trabalho apresenta semelhanças com o trabalho de Just et al (2008), onde também são apresentados os resultados obtidos em um survey com participação de desenvolvedores e usuários finais dos projetos de software do Eclipse, Mozilla e Apache[10]. O survey foi estruturado em duas partes: (i) elaboração de questionários encaminhados para desenvolvedores e usuários que reportaram bugs, com o objetivo de identificar quais as informações mais importantes na análise dos bugs e quais as principais deficiências do processo de análise e resolução de bugs; (ii) 286 relatórios de bug foram selecionados aleatoriamente e encaminhados para os desenvolvedores mensurarem a qualidade em uma escala de cinco níveis. Assim como no trabalho de Just et al (2008), os resultados desta pesquisa indicaram que há uma lacuna entre as informações que os desenvolvedores consideram mais relevantes para analisar os bugs e as informações que são registradas pelos usuários finais. Além disso, também foi observado que há uma grande diversidade na qualidade dos relatórios de bug. Diante destes fatos, este trabalho propõe a utilização de uma ferramenta capaz de mensurar o nível de qualidade de um relatório de bug e de sugerir modificações capazes de melhorar a qualidade do mesmo. Foi desenvolvido um protótipo deste sistema, conhecido como

[10] http://www.apache.org

CUEZILA, e os primeiros experimentos indicaram níveis de precisão de 48% na avaliação da qualidade dos relatórios de bug.

D'Ambros et al (2007) apresentam um conjunto de técnicas que analisam a base de dados dos BTSs e extraem informações que auxiliam na detecção dos componentes de software mais problemáticos. A técnica System Radiography analisa todos os bugs reportados para um determinado sistema com o objetivo de identificar quais os componentes mais problemáticos, isto é, quais as partes do sistema que tendem a apresentar maior número de bugs. Uma matriz de visualização é proposta para analisar a distribuição de bugs abertos ao longo do tempo. Após a criação da matriz, aplica-se um algoritmo que ordena os diversos componentes de software em função das similaridades, isto é, a correlação entre a ocorrência de bugs nos diversos componentes, considerando os intervalos de tempo. O objetivo deste algoritmo é identificar os grupos de componentes que tendem a apresentar um elevado número de bugs no mesmo intervalo de tempo. Os grupos de componentes identificados como mais problemáticos servem como alvo de estudo da segunda técnica apresentada no artigo, o Bug Watch, cujo objetivo é fazer uma análise mais aprofundada de um sub-conjunto de bugs, identificando quais são os bugs mais críticos do sistema. A técnica consiste em apresentar um painel com os relatórios de bug não resolvidos dos componentes mais críticos, representado-os na forma de gráficos de pizza divididos em três setores: Status, Activity e Severity/Priority. De acordo com os autores, estas informações seriam suficientes para revelar a criticidade do relatório de bug. O setor Status identifica as diversas fases do bug (NEW, ASSIGNED, RESOLVED, REOPENED, VERIFIED, CLOSED) com os respectivos marcos de início e fim. O setor Activity identifica as modificações realizadas no bug, isto é, quanto maior o número de sub-divisões neste setor, maior o número de atividades realizadas. O setor Severity/Priority descreve a severidade e a prioridade do bug. Quanto mais escura a cor deste setor, mais importante a correção deste bug se faz para o sistema. Os autores argumentam que os bugs mais críticos são aqueles que apresentam mudanças intensas no ciclo de vida (várias divisões no setor Status), muitas atividades (várias divisões no setor Activity) e severidade / prioridade elevadas (cores escuras no setor Severity / Priority).

2.7. Conclusão

BTSs são ferramentas cada vez mais utilizadas na indústria de software, tanto em projetos de software livre, onde os desenvolvedores e usuários finais estão geograficamente distribuídos, quanto em empresas privadas e públicas de todos os

portes. A automatização das atividades do processo de correção de bugs, associado ao uso do histórico de correções de bugs, possibilita pesquisas inovadoras na área de Engenharia de Software. Estas pesquisas têm recebido muitos adeptos na atualidade, como mostra a cronologia recente dos trabalhos relacionados. Os resultados das pesquisas apresentadas indicam melhorias na qualidade e redução dos custos das operações realizadas no processo de correção de bugs. Contudo, estas iniciativas raramente são implementadas nas organizações.

Um dos grandes desafios da Engenharia de Software é prover subsídios para elaboração de bons planos para o gerenciamento de projetos de software, planos capazes de indicar custos e prazo de entrega que sejam próximos do custo e do tempo observados ao final do projeto. Neste contexto, uma especialização deste problema é a elaboração de um plano de liberação da próxima versão de software, incorporando um conjunto de correções de bugs reportados por usuários, testadores e desenvolvedores.

A solução para este problema deve estimar os esforços de cada correção, priorizar, sequenciar e distribuir as correções entre os desenvolvedores, de forma que os profissionais alocados nestas correções tenham a capacidade necessária para executá-las adequadamente. Assim, a solução para este problema poderia aplicar algumas técnicas propostas nos trabalhos relacionados que foram apresentados neste capítulo para prover os insumos necessários para elaboração do plano de liberação das versões futuras do software. Além disso, o uso destas técnicas associado à aplicação de técnicas de otimização, que serão discutidas no capítulo 3, pode contribuir para a elaboração de planejamentos exequíveis, com os custos das operações e prazos de entrega reduzidos.

CAPÍTULO 3

APLICAÇÃO DE HEURÍSTICAS EM PROBLEMAS DE OTIMIZAÇÃO

3.1. Introdução

Estudos na área de Pesquisa Operacional envolvem o desenvolvimento de algoritmos que utilizam métodos matemáticos e estatísticos para prover soluções ótimas ou aproximadas para problemas complexos. Estas pesquisas são caracterizadas pela busca intensa por algoritmos cada vez mais eficientes e eficazes, isto é, aqueles que são capazes de encontrar soluções de melhor qualidade no menor tempo possível. Considere, por exemplo, uma organização que deseje reduzir custos operacionais e aumentar sua receita. A melhor solução para este problema implica, para a organização, vantagens competitivas essenciais para a sustentabilidade do seu negócio.

Dentre os problemas estudados, destacam-se os problemas de otimização cuja finalidade é buscar soluções que atendam um conjunto de restrições impostas, para maximizar ou minimizar uma determinada função objetivo ou função custo (Papadimitriou e Steiglitz, 1998). Dependendo da natureza do problema, é impossível encontrar a solução ótima, em tempo exequível, a partir de um método exato que se proponha a avaliar todas as soluções possíveis de um dado problema e selecionar a melhor dentre elas. Nestes casos, uma alternativa é utilizar métodos heurísticos, que consistem de algoritmos que buscam uma solução aproximada. Metaheurísticas são procedimentos gerais que fornecem um "guia" para a elaboração de heurísticas usadas para resolver um problema específico (Dréo et al., 2005).

Este capítulo está dividido em sete seções. A Seção 3.1 consiste desta introdução. A Seção 3.2 faz uma breve apresentação dos problemas combinatórios. A Seção 3.3 descreve os métodos exatos para solucionar problemas combinatórios. A Seção 3.4 apresenta abordagens heurísticas para solucionar problemas combinatórios. A Seção 3.5 apresenta as principais metaheurísticas utilizadas para solucionar os problemas de otimização difíceis na Engenharia de Software. Na Seção

3.6, apresentamos alguns exemplos de aplicações de métodos heurísticos para solucionar problemas da área de Engenharia de Software. A Seção 3.7 faz uma análise consolidada das principais questões abordadas neste capítulo.

3.2. Problemas Combinatórios

Problemas combinatórios consistem da procura de grupos, ordenações ou atribuições de um conjunto discreto e finito de objetos, conhecidos como variáveis de decisão, que atendam um conjunto de restrições impostas. Problemas de otimização combinatória são problemas combinatórios cuja finalidade é minimizar ou maximizar o valor de uma determinada função objetivo (Hoos e Stüzle, 2004).

Cada solução gerada para o problema combinatório é conhecida como solução candidata. Se uma determinada solução candidata satisfaz todas as restrições do problema, então ela é chamada de solução viável. Soluções ótimas são soluções viáveis que conduzem a função objetivo para o valor ótimo. O conjunto de todas as soluções candidatas é conhecido como espaço de busca.

A qualidade de uma solução é medida em função do valor gerado pela função objetivo e a sua proximidade do valor ótimo. Quanto mais próximo do valor ótimo estiver o valor calculado pela função objetivo para uma determinada solução candidata, melhor será a qualidade desta solução candidata.

Um problema combinatório pode ser definido de forma generalizada, como, por exemplo, o problema do caixeiro viajante (Garey e Johnson, 1979): encontrar o trajeto que possua a menor distância, começando numa cidade qualquer, entre várias, visitando cada cidade somente uma vez e regressando à cidade inicial. Uma instância é uma especialização do problema generalizado. Por exemplo, uma instância para o problema do caixeiro viajante poderia ser encontrar o menor trajeto para percorrer todas as cidades do estado do Rio de Janeiro.

A função complexidade do tempo de um algoritmo expressa o maior tempo necessário (pior caso) para solucionar um determinado problema. Em geral, esta função é representada na forma $O(n)$, onde n significa o tamanho da instância do problema. Algoritmos de tempo polinomial são aqueles cuja função complexidade do tempo pode ser definida como $O(p(n))$ para uma determinada função polinomial p. Os demais algoritmos, cujos tempos de execução não podem ser delimitados, são conhecidos como algoritmos de tempo exponencial. Garey e Johnson (1979) fizeram uma análise comparativa, descrita na Tabela 3.1, entre os tempos de execução de alguns algoritmos polinomiais e exponenciais. Observe que, no caso dos algoritmos exponenciais (algoritmos 5 e 6), a solução exata para instâncias grandes do problema

não pode ser encontrada em tempo exequível. Nestes casos, devem ser utilizados algoritmos, que executam em tempo polinomial, para encontrar soluções aproximadas para o problema. Embora a referência da análise apresentada na Tabela 3.1 seja antiga, os tempos de resposta obtidos não seriam muito diferentes caso este experimento fosse novamente realizado hoje em dia.

Tabela 3.1. Análise Comparativa de Algoritmos de Tempo Polinomial e Exponencial

Algoritmo	Função Complexidade	Tamanho da Instância do Problema			
		10	20	30	40
1	n	0,00001 seg	0,00002 seg	0,00003 seg	0,00004 seg
2	n^2	0,0001 seg	0,0004 seg	0,0009 seg	0,0016 seg
3	n^3	0,001 seg	0,008 seg	0,027 seg	0,064 seg
4	n^5	0,1 seg	3,2 seg	24,3 seg	1,7 min
5	2^n	0,001 seg	1,0 seg	17,9 min	12,7 dias
6	3^n	0,059 seg	58 min	6,5 anos	3.855 séculos

Os problemas são comumente categorizados em duas classes: problemas de decisão e problemas de otimização. Problemas de decisão são aqueles cujo objetivo é verificar se existe ou não uma solução que atenda a determinadas restrições, enquanto que os problemas de otimização tem por finalidade buscar soluções que atendam um conjunto de restrições impostas, para maximizar ou minimizar uma determinada função objetivo ou função custo (Papadimitriou e Steiglitz, 1998). Dessa forma, a pergunta “existe um trajeto de distância igual a k capaz de percorrer todas as cidades do estado do Rio de Janeiro?” é um exemplo de problema de decisão, ao passo que a pergunta “qual o trajeto de distância mínima capaz de percorrer todas as cidades do estado do Rio de Janeiro?” é um exemplo de problema de otimização.

Adicionalmente, os problemas também são classificados quanto ao tipo de algoritmo utilizado para solucioná-los. A classe de problemas P (*deterministic polynomial time*) reúne os problemas de decisão que são resolvidos por algoritmos determinísticos em tempo polinomial. A classe de problemas NP (*nondeterministic polynomial time*) associa os problemas de decisão que requerem algoritmos não determinísticos para encontrar uma solução em tempo polinomial.

Transformabilidade, também conhecida como redutibilidade, é um conceito muito importante na avaliação da complexidade dos problemas combinatórios e consiste na possibilidade de transformar, em tempo polinomial, qualquer instância de um problema P_1 em instâncias de outro problema P_2 (Reeves, 1993). Se isto for possível, então um algoritmo A usado para resolver instâncias do problema P_2, também poderá ser usado para resolver instâncias do problema P_1.

Um determinado problema P_1 é NP-Difícil se qualquer problema NP puder ser transformado em P_1 em tempo polinomial. Se o problema P_1 pertencer à classe NP, então podemos dizer que P_1 é um problema NP-Completo (Reeves, 1993). Assim, diz-se que a classe dos problemas NP-Completo corresponde ao subconjunto dos problemas mais difíceis em NP, e, se um dos problemas NP-Completo puder ser solucionado em tempo polinomial, então qualquer problema NP também poderá ser resolvido em tempo polinomial (Reeves, 1993).

Garey e Jonhson (1979) fizeram um levantamento dos principais problemas NP-completos. Como resultado foram enumerados uma série de problemas em diversas áreas, tais como teoria dos grafos, projetos de redes, teoria dos conjuntos, armazenamento e recuperação de dados, escalonamento de tarefas, programação matemática, álgebra, teoria dos jogos e lógica.

3.3. Técnicas de Otimização Exatas

Os métodos de otimização capazes de encontrar soluções ótimas para problemas combinatórios são conhecidos como métodos exatos. Esta seção apresenta os seguintes métodos: busca sistemática, programação linear e *branch and bound*.

Técnicas baseadas em busca sistemática exploram sistematicamente todo o espaço de busca, avaliam a qualidade de cada solução candidata e selecionam a melhor solução dentre todas (Hoos e Stüzle, 2004). Conforme visto na Seção 3.1, dependendo da natureza do problema e do tamanho da instância, este método não é exequível em tempo polinomial.

Programação linear é uma técnica de otimização matemática que garante que a solução ótima será encontrada. As entradas para o modelo de programação linear constituem um conjunto de números reais, não negativos, chamados de variáveis de decisão. A finalidade é maximizar ou minimizar o valor de uma expressão, considerando as respectivas restrições. Tanto a expressão que se deseja otimizar quanto as restrições são expressas na forma de equações lineares das variáveis de decisão. Contudo, os problemas combinatórios são caracterizados por variáveis de decisão discretas e, dependendo da complexidade das restrições impostas, pode ser inviável o uso das técnicas de programação linear para solucionar estes problemas (Harman, 2007).

Branch and bound é uma técnica de otimização exata bastante conhecida. Esta abordagem consiste de uma enumeração sistemática de todas as soluções de candidatas, onde o espaço de busca é representado na forma de uma árvore e decomposto sucessivamente. Ao longo deste processo, soluções candidatas são

descartadas em massa, utilizando-se como referência os limites calculados, até que a solução ótima seja encontrada (Papadimitriou e Steiglitz, 1998). A eficiência deste método depende da definição do processo de decomposição do espaço de busca e dos estimadores de limites. A razão da sua baixa popularidade recente está ligada ao fato deste método ser de difícil implementação e pouco flexível, visto que não existe um algoritmo genérico que possa ser aplicado a qualquer problema. Mudanças no modelo do problema de otimização, em geral, não são facilmente acomodadas pelo *branch and bound*, e acabam demandando ajustes no método com alto custo de implementação.

As tendências nos estudos de problemas de otimização combinatória sugerem a criação de algoritmos baseados no uso de heurísticas para produzir soluções aproximadas. A aplicação destas técnicas será descrita em maiores detalhes nas próximas seções deste capítulo.

3.4. Heurísticas

Segundo Reeves (1993), heurísticas são técnicas que tem por finalidade a busca de soluções boas (próximas da solução ótima) para problemas combinatórios. Consistem na aplicação de regras que tentam guiar a solução para um ótimo local (a melhor solução considerando um subconjunto do conjunto das soluções viáveis para o problema), na esperança de que este ponto também seja o ótimo global (a melhor dentre todas soluções possíveis). Estas técnicas, em geral, apresentam baixo custo computacional e criam soluções de qualidade muito superior à média das soluções aleatórias. Contudo, elas não oferecem garantias de qualidade e viabilidade das soluções geradas, e em muitos casos não é possível afirmar quão próxima a solução gerada está da solução ótima.

Apesar das incertezas na qualidade das soluções geradas, os métodos heurísticos são as técnicas mais utilizadas para resolver problemas NP Difíceis. Tendo em vista que o espaço de busca para muitas instâncias desta classe de problemas apresenta tamanho de ordem exponencial, isto dificulta a utilização dos métodos de otimização exatos, apresentados na seção 3.2, e uma abordagem para produzir soluções aproximadas é mais frequentemente utilizada.

Dentre os principais métodos heurísticos, destacam-se os métodos construtivos e as técnicas de busca local. Os métodos construtivos, também conhecidos como métodos gulosos, consistem de procedimentos, baseados em regras específicas do contexto de um determinado problema, que constroem iterativamente uma solução candidata. O método de busca local consiste de um processo de busca que inicia com

a avaliação de uma solução candidata inicial e move-se iterativamente para outras soluções, até que a melhor solução seja encontrada.

Uma vizinhança N (s, σ) de uma solução s corresponde ao conjunto de soluções que podem ser geradas a partir de uma operação σ, chamada de movimento (Reeves, 1993). Cada elemento do conjunto N (s, σ) é conhecido como solução vizinha de s. Um ótimo local é a melhor solução considerando os elementos de uma determinada relação de vizinhança, ao passo que um ótimo global é a melhor solução para qualquer elemento do espaço de busca do problema.

Cada iteração do método de busca local avalia as soluções do conjunto vizinhança da solução corrente. Um movimento para uma nova solução é feito se o novo vizinho possuir qualidade superior à solução corrente. Este processo é conhecido como Melhoria Iterativa (Hoos e Stüzle, 2004) e pode ser implementado nas seguintes maneiras: (i) o movimento é feito para o primeiro vizinho de qualidade superior à solução corrente (Primeira Melhoria); (ii) toda vizinhança é analisada e busca-se o vizinho de melhor qualidade (Melhor Melhoria). A busca é encerrada quando não há nenhum vizinho melhor do que a solução corrente.

O problema deste método é que a solução encontrada corresponde ao ótimo local de uma determinada relação de vizinhança, que pode estar distante do ótimo global e implicar em uma solução final de baixa qualidade. Existem algumas abordagens clássicas para tentar contornar este problema: aumentar o tamanho da vizinhança, reiniciar o processo de busca a partir de soluções candidatas iniciais diferentes e permitir alguns movimentos para soluções de qualidade inferior à solução corrente. Esta última abordagem é utilizada pela metaheurística Arrefecimento Simulado, que será descrita em maiores detalhes na Seção 3.5.1.

3.5. Metaheurísticas

Na área de otimização, muitas heurísticas são frequentemente desenvolvidas com o objetivo de produzir soluções próximas da solução ótima. Contudo, a grande maioria destas heurísticas foi concebida para solucionar problemas específicos, inviabilizando a aplicação destas para solucionar outros problemas (Dréo et al., 2005).

Metaheurísticas consistem em procedimentos genéricos que podem ser aplicados em diferentes contextos e servem como um “guia” para construção de novas heurísticas para solucionar problemas específicos. Em geral, a definição destes procedimentos foi inspirada em algum paradigma da física (Arrefecimento Simulado), biologia (Algoritmos Genéticos) ou etologia (Colônia de Formigas). As próximas subseções apresentam exemplos de metaheurísticas.

3.5.1. Arrefecimento Simulado

Esta técnica foi introduzida por Kirkpatrick et al. (1983) e consiste de um modelo baseado em um processo térmico utilizado na metalurgia para obtenção de estados de um sólido de baixa energia.

Este processo é dividido em duas etapas: (i) inicialmente, o material é conduzido para altas temperaturas no qual ele se funde; (ii) a temperatura do material é lentamente reduzida, de forma controlada, até que o sólido atinja uma configuração com a menor energia interna e, consequentemente, com uma redução nos defeitos do material. A metaheurística faz uma analogia da temperatura com a função objetivo que se deseja minimizar ou maximizar, ao passo que a configuração do material corresponde a solução do problema. O pseudocódigo deste processo está descrito na Figura 3.1.

```
selecionar solução inicial s;
selecionar temperatura inicial t > 0;
selecionar função de redução de temperatura α;
enquanto critério de parada não for atingido
   enquanto (nº de iterações < N)
      selecionar um vizinho s' ∈ N (s)
      ΔE = f(s') - f (s)
      se (ΔE ≤ 0) então
         s <- s'
      se não
         gerar aleatóriamente r ∈ [0, 1]
         se (r < e (-ΔE/t)) então
            s <- s'
         fim se
      fim se
      t = α (t)
   fim enquanto
fim enquanto
```

Figura 3.1. Descrição em pseudocódigo do Arrefecimento Simulado

O algoritmo está descrito em sua forma genérica e um conjunto de decisões precisa ser feito durante a implementação. Este conjunto de decisões pode ser categorizado em dois grupos: (i) o primeiro grupo consiste dos parâmetros genéricos do processo de arrefecimento, tais como a temperatura inicial, o esquema de resfriamento (a definição da função α e do número de iterações N) e o critério de parada; (ii) os parâmetros do segundo grupo envolvem questões específicas do problema, tais como o mapeamento do espaço de busca, a definição da função objetivo e a definição da relação de vizinhança entre as soluções.

Este método pode ser visto como uma variação do método de busca local onde alguns movimentos para soluções de qualidade inferior são permitidos, tendo como objetivo contornar o problema do ótimo local. A variação de energia interna ΔE avalia a ocorrência de melhoria na solução corrente. Um $\Delta E \leq 0$ significa que a nova solução produz um resultado melhor ou igual à solução corrente e, portanto, o movimento será aceito. Caso contrário, sorteia-se um número real $r \in [0,1]$. Se $r < e\ (-\Delta E/\ t\ )$, então a modificação é aceita.

3.5.2. Algoritmos Genéticos

Algoritmos genéticos são técnicas de busca apresentadas por Holland (1975) e são inspiradas na evolução biológica das espécies. A dinâmica dos algoritmos genéticos é inspirada no paradigma da seleção natural e tem por objetivo gerar gradativamente, ao longo de diversas iterações, soluções cada vez melhores. Uma população consiste de um conjunto de indivíduos (também conhecidos como cromossomos), cada um representando uma solução candidata para o problema.

O processo de reprodução define a criação de uma nova geração de indivíduos a partir da geração corrente, aplicando os processos de seleção, recombinação (*crossover*) e mutação. A seleção garante que as melhores soluções da população corrente serão utilizadas no processo de reprodução. A recombinação consiste na mistura dos genes de dois cromossomos (cromossomos pais) para gerar um novo cromossomo (cromossomo filho).

A mutação é caracterizada por uma pertubação aleatória em um ou mais genes de um determinado cromossomo a fim de gerar um novo indivíduo. A principal contribuição da mutação é permitir diversidade ao longo das sucessivas gerações, evitando a convergência do algoritmo para um mínimo (ou máximo) local. A Figura 3.2 descreve em pseudocódigo a dinâmica do algoritmo genético.

```
gerar população inicial P_o
P <- P_o
enquanto critério de parada não for atingido
   para cada indivíduo p ∈ P
      avaliar qualidade de p
   fim para
   gerar conjunto B de melhores indíviduos de P
   gerar conjunto R de recombinações entre indivíduos de P
   gerar conjunto M de mutações
   P <- B ∪ R ∪ M
fim enquanto
selecionar melhor indivíduo de P
```

Figura 3.2. Descrição em pseudocódigo do Algoritmo Genético

Algoritmos genéticos foram inicialmente negligenciados nas pesquisas em função do alto custo computacional. Contudo, nos últimos anos, o número de trabalhos na área de pesquisa operacional associados ao uso de algoritmos genéticos cresceu expressivamente (Dréo et al., 2005). Há uma forte tendência na maioria dos trabalhos recentes na área de pesquisa operacional em adotar algoritmos genéticos como método de otimização principal, conforme constatado por Harman (2009).

A razão disto está baseada em uma maior flexibilidade dos algortimos genéticos em relação aos demais métodos para resolver diferentes cenários de um mesmo problema. Por exemplo, dado um conjunto de instância um mesmo problema, é possível definir diferentes configurações (tamanho da população, critério de parada e parâmetros do processo de reprodução), onde cada configuração é mais adequada para lidar com um determinado tipo de instância a fim de alcançar a melhor relação custo (tempo de execução do algoritmo) x benefício (qualidade da solução). Além disso, o avanço das arquiteturas computacionais, que permite a execução paralela de várias buscas do algoritmo genético, também contribuiu para o crescimento no número de adeptos.

3.5.3. Colônia de Formigas

Esta técnica foi introduzida por Colorni et al. (1991) e consiste de um modelo baseado na analogia do processo de busca por comida realizado pelas colônias de formigas. Durante a busca por comida, as formigas depositam ao longo do caminho uma substância conhecida como feromônio, que serve como sinalizador da melhor rota. As demais formigas, ao perceberem a existência desta substância, direcionam seus trajetos para as áreas com maior concentração do feromônio. Como resultado, as formigas encontram rapidamente o caminho mais curto até o local onde encontra-se a comida.

De acordo com Dréo et al. (2005), esta metaheurística tem uma série de características interessantes, tais como possibilidade de paralelismo, flexibilidade para acomodar modificações e robustez, justificando o aumento do número de adeptos desta técnica nos últimos anos. Harman (2007) também ressalta o aumento do número de trabalhos na área de Engenharia de Software com aplicação da metaheurística Colônia de Formigas.

3.6. Aplicação de Soluções Heurísticas na Engenharia de Sotware

A área de Engenharia de Software é caracterizada por um conjunto de processos inter-relacionados e com restrições de orçamento, prazo e recursos. A gestão destes

processos tem impacto relevante na sustentabilidade das organizações, haja visto a importância dos softwares na gestão das atividades dos mais diversos tipos de empresa. Assim, vemos que é essencial uma visão otimizada dos processos de Engenharia de Software com o objetivo de torná-los cada vez mais eficientes.

Soluções heurísticas são técnicas apropriadas para lidar com problemas complexos cujo espaço de busca cresce em escala exponencial. Ao longo desta seção, veremos que estas técnicas podem ser aplicadas para prover soluções para um conjunto de problemas da área de Engenharia de Software.

Métodos heurísticos são aplicados com bastante frequência nas áreas de engenharia, financeira e industrial, com o objetivo de apoiar a melhoria dos processos. Contudo, estas iniciativas ainda são recentes na área de Engenharia de Software. O termo Engenharia de Software Baseada em Busca (*Search Based Software Engineering - SBSE*) foi criado por Harman e Jones (2001) e consiste em uma abordagem dos problemas da Engenharia de Software sob a ótica de problemas de busca e otimização, com a aplicação de técnicas heurísticas para buscar soluções aproximadas. Desde então, houve um acentuado crescimento no número de trabalhos nesta linha de pesquisa, com o desenvolvimento de conferências e revistas técnicas específicas para o tema, conforme constatado pelos trabalhos de Clarke et al. (2003), Harman (2007) e Harman et al. (2009). A Figura 3.3 apresenta o gráfico do número de publicações por ano, de acordo com o levantamento feito por Haman et al. (2009).

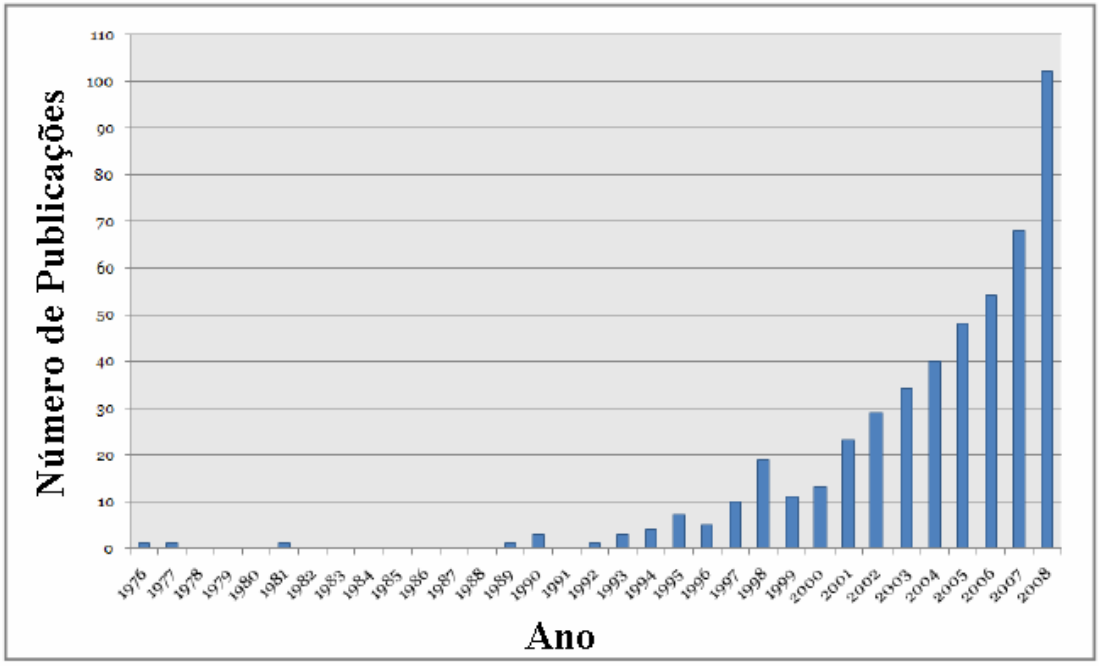

Figura 3.3. Evolução no número de publicações em SBSE (Harman, 2009)

Para Clarke et al (2003) a análise da aderência da aplicação de uma solução heurística deve considerar os seguintes critérios: (i) o espaço de busca do problema deve ser tão grande (crescimento exponencial) que inviabilize a busca da solução através de um método exato em tempo polinomial; (ii) inexistência de soluções completas, isto é, aquelas que são capazes de prover uma solução para o problema para qualquer instância em tempo polinomial; (iii) existência de uma função objetivo adequada ao contexto do problema; (iv) as soluções candidatas devem ser encontradas em tempos exequíveis, isto é, o tempo de resposta do algoritmo não pode comprometer a continuidade do processo.

Se um determinado problema atende a estes quatro critérios, então uma solução heurística é adequada para buscar uma solução aproximada. Esta abordagem está sendo aplicada em diversas áreas da Engenharia de Software, tais como análise de requisitos, projeto arquitetural, manutenção, testes, gerência da configuração e gerenciamento de projetos, com grande concentração na área de testes (cerca de 70%), conforme o levantamento feito por Harman et al. (2009). As próximas subseções apresentarão alguns exemplos de problemas da Engenharia de Software para os quais soluções heurísticas foram propostas.

3.6.1. Problemas de Análise de Requisitos

O ciclo de vida clássico do processo de desenvolvimento de software segmenta as atividades em quatro etapas: análise de requisitos, projeto, codificação e testes (Pressman, 2006). Para Kotonya e Sommerville (1998) as maiores incertezas deste processo estão concentradas na fase de análise de requisitos, quando são executadas as atividades de levantamento, especificação e negociação dos requisitos do software.

Quando o universo de requisitos é muito grande, recomenda-se uma abordagem incremental: inicialmente é feita uma entrega do núcleo do produto, contendo as funcionalidades básicas, e, em seguida, são realizados sucessivos ciclos de desenvolvimentos incrementais com a inclusão de novas funcionalidades ao produto original (Pressman, 2006).

Para Sagrado e Águila (2009) a seleção dos requisitos que deverão ser contemplados no próximo ciclo incremental pode ser visto como o problema da próxima versão (next release problem – NRP). Este problema consiste de: (i) um conjunto de clientes (cada um com uma determinada relevância no contexto do projeto de software); (ii) um conjunto de requisitos do software (para cada requisito há um custo de implementação específico); e (iii) um orçamento definido para a próxima fase do projeto de construção. A solução deste problema deve maximizar o número de

requisitos que serão contemplados na próxima versão, considerando a restrição do orçamento e a importância de cada requisito sob a ótica do cliente que o solicitou.

Sagrado e Águila (2009) propõem a construção de um algoritmo baseado na metaheurística de colônia de formigas para buscar uma solução deste problema. Os resultados dos experimentos indicam soluções mais atraentes do que as soluções recomendadas pelos analistas de requisitos. Os autores também argumentam que uma análise experimental adicional deve ser conduzida com a aplicação de soluções baseadas em métodos construtivos e na metaheurística arrefecimento simulado, para avaliar a técnica mais aderente para o problema.

3.6.2. Problemas de Projeto Arquitetural

A arquitetura de um software consiste na descrição dos principais componentes de um sistema, as interações entre estes elementos, as restrições impostas aos relacionamentos entre os elementos e os padrões que guiam a construção do software (Shaw e Garlan, 1996). Para reduzir a complexidade da construção e aumentar a compreensibilidade da arquitetura, os softwares são divididos em subsistemas, que colaboram entre si para compor a arquitetura completa do software.

Uma boa arquitetura de software é caracterizada pela alta coesão entre os elementos internos de um subsistema e o baixo acoplamento nas interfaces de comunicação dos subsistemas (Pressman, 2006). Estes aspectos são desejáveis, pois reduzem o custo de manutenções, uma vez que reduzem a propagação de efeitos colaterais entre os subsistemas após a aplicação de modificações.

Harman et al. (2002) modelaram o problema da decomposição da arquitetura de um software em módulos a fim de maximizar a coesão entre os componentes internos de um módulo e minimizar o acoplamento nas interfaces de comunicação entre os módulos. A função objetivo consiste de uma equação que contempla índices de coesão e acoplamento da arquitetura. Os autores introduziram um conceito normalizado para representação dos módulos de qualquer software. Por fim, aplicaram um algoritmo genético que apresentou resultados melhores quando comparados a outros trabalhos semelhantes que aplicaram técnicas baseadas em busca local e busca sistemática para resolver o mesmo problema.

Mudanças na arquitetura do software podem comprometer a qualidade dos serviços oferecidos por ele. Kim e Park (2009) analisaram o problema de reconfiguração da arquitetura do software após a aplicação de modificações em algum componente da arquitetura. O processo de seleção de uma arquitetura viável para o software foi definido com um procedimento de busca, para o qual deve-se aplicar alguma solução heurística. A função objetivo calcula o nível de qualidade de cada

solução candidata. Ao final da busca, a melhor solução (com maior nível de qualidade) é selecionada. A técnica proposta consiste de um algoritmo baseado na metaheurística algoritmo genético. Os resultados dos experimentos mostram que a abordagem proposta é capaz de encontrar a melhor configuração da arquitetura do software em pouco tempo, mesmo para instâncias grandes do problema.

3.6.3. Problemas de Manutenção

Refatoração é o processo de aperfeiçoar a estrutura interna do código de um software sem alterar o seu comportamento sob a ótica externa (Fowler, 1999). Este processo consiste de uma inspeção no código fonte do software com o objetivo de evitar a deterioração de sua qualidade, melhorar a compreensibilidade do código e evitar uma possível inclusão de bugs no futuro. Em geral, a refatoração é feita manualmente, onde o código fonte é revisado por programadores experientes. Contudo, em sistemas de larga escala, a condução manual deste processo torna-se muito difícil e pouco viável.

Harman e Tratt (2007) modelaram o processo de refatoração em sistemas de larga escala como um problema de busca, com a finalidade de encontrar uma configuração das estruturas internas do código capaz de minimizar as seguintes métricas: o acoplamento entre as classes e o desvio padrão do número de métodos por classe. O problema em questão envolve múltiplos objetivos que podem ser conflitantes e, nestes casos, não há soluções ótimas (Dréo et al, 2005). Dessa forma, os autores sugerem a busca por soluções baseada no princípio do ótimo de Pareto (Harman, 2007), isto é, encontrar a região do espaço de busca das soluções, conhecida como fronteira de Pareto, que contempla o conjunto das melhores soluções.

3.6.4. Problemas de Testes

A fase de testes consiste na execução de procedimentos, conhecidos como casos de teste, para verificar a ocorrência de anormalidades no comportamento do software. Clarke et al (2003) categorizam os problemas de teste em três classes: teste estrutural, teste baseado na especificação e teste do pior tempo de execução.

Testes estruturais, também conhecidos como testes caixa-branca, tem por objetivo executar todas as instruções de um determinado módulo, ao menos uma vez, e verificar se ocorreu alguma anomalia. Para sistemas de larga escala, a possibilidade de caminhos diferentes de execução das instruções internas (considerando as chamadas aos métodos, desvios condicionais e laços) constitui um espaço de busca muito grande, dificultando a criação dos casos de teste. O problema da geração de um caso de teste mínimo (com menor custo de execução) capaz de verificar todo fluxo de execução das estruturas internas de um software foi modelado por Jones et al (1998),

baseado na analogia com problemas clássicos de cobertura de grafos. Os autores desenvolveram uma ferramenta, baseada na metaheurística do algoritmo genético, para gerar casos de teste para um determinado programa, onde foram implantadas propositalmente inúmeras falhas. Os resultados dos experimentos mostraram que 97% das falhas na estrutura interna do programa examinado foram detectadas.

Testes baseados na especificação são apropriados quando a especificação dos requisitos do software foi desenvolvida utilizando métodos formais, baseados na criação de proposições lógicas para especificar os requisitos. O problema da geração de dados de teste para verificar a conformidade do software com os requisitos foi modelado como um problema de otimização por Tracey et al (1998), que propuseram um algoritmo baseado na metaheurística do arrefecimento simulado para automatizar o processo de teste. Os autores argumentam que a técnica proposta é capaz de reduzir sensivelmente o custo do processo e aumentar a qualidade do software produzido.

O pior tempo de execução é uma métrica muito relevante para sistemas de tempo real, visto que será um dos insumos para o cálculo da disponibilidade do sistema. A definição do pior de tempo de execução de um determinado software pode ser entendida como um problema de busca complexo, cujo resultado calculado depende da arquitetura computacional onde está instalado o software. Khan e Bate (2009) modelaram este problema como um problema de busca e aplicaram algumas soluções heurísticas para calcular o pior tempo de execução.

3.6.5. Problemas de Gerência de Projetos de Software

A construção de um cronograma (*scheduling*) é um processo que consiste na alocação de recursos a fim de executar um conjunto de tarefas com determinadas restrições (Hart et al, 2005). Garey e Johnson (1979) enumeraram alguns tipos de problemas relacionados com a construção de cronogramas que podem ser aplicados em diversos processos, tais como as atividades industriais, a comercialização de produtos e o escalonamento de processos em um computador.

Em geral, todos os problemas de construção de cronogramas compartilham algumas características semelhantes: (i) definem uma série de tarefas que devem ser executadas por um conjunto de recursos; (ii) apresentam restrições (recurso, prazo, orçamento, prioridades, entre outras); e (iii) definem uma função objetivo, que deve ser minimizada ou maximizada (tempo máximo do esquema de execução, tempo médio de execução das tarefas, taxa média de ocupação dos recursos, custo do plano de execução, entre outras).

Dentre os diversos tipos de problemas de construção de cronogramas, destacam-se os Problemas de Construção de Cronogramas de Projetos (*Project Scheduling Problems*), que são versões adaptadas para o contexto da área de Gerenciamento de Projetos (Kolisch et al, 1997). Um Problema de Construção de Cronogramas de Projeto (PCCP) consiste de atividades (tarefas do projeto que precisam ser executadas), recursos (renováveis e não renováveis) e relações de precedência entre as atividades. Uma variação do PCCP bastante conhecida é o Problema de Construção de Cronogramas de Projeto com Restrição de Recursos (Resource Constrained Project Scheduling Problem), cujo objetivo é minimizar o tempo de execução do projeto, conhecido como *makespan*.

Alba e Chicano (2007) descrevem o planejamento das atividades do processo de desenvolvimento de software como uma variação do PCCP. Neste problema, as tarefas são atividades previstas em um ciclo de desenvolvimento de software tradicional (análise, projeto da arquitetura, codificação, testes, etc.) e os recursos correspondem aos empregados da organização. Cada atividade requer um esforço, medido em pessoa-mês, e uma habilidade específica para execução. Por outro lado, cada empregado possui um salário de referência e um conjunto de conhecimentos que define as suas respectivas habilidades.

Com o objetivo de prover uma solução capaz de minimizar o prazo e o custo do projeto, foi desenvolvido um algoritmo genético. Durante os experimentos, foram criados 48 cenários de projetos fictícios e cada um dos cenários foi submetido a 100 execuções do algoritmo genético, a fim de obter resultados estatísticos expressivos. Os resultados indicaram que o custo da solução é maior nas instâncias com maior número de atividades. Por fim, os autores afirmam que os métodos baseados no uso de algoritmos genéticos são bastante aderentes a este tipo de problema e podem servir de base para construção de ferramentas para automatizar o processo de geração de cronogramas de projetos.

Antoniol et al (2005) descreve o problema de planejamento das manutenções em sistemas de larga escala com o objetivo de minimizar o prazo total de execução destas atividades. O modelo deste problema considera atividades de manutenção de software (tarefas) que devem ser executadas por equipes de desenvolvimento (recursos). Cada atividade possui um esforço necessário para execução, enquanto que cada equipe de desenvolvimento possui uma medida de tamanho proporcional à quantidade de desenvolvedores da equipe. O tempo de execução de cada atividade é definido como a razão entre o esforço necessário e o tamanho da equipe de desenvolvimento alocada na execução da tarefa. Foram desenvolvidos três diferentes métodos de busca baseados nas técnicas algoritmo genético, busca local e arrefecimento

simulado. Os experimentos realizados indicaram que o método baseado no uso do algoritmo genético apresentou os melhores resultados, com uma redução superior a 50% do prazo de execução do projeto, quando comparada com os valores realizados.

3.6. Conclusão

Métodos heurísticos podem ser aplicados para encontrar soluções aproximadas para problemas complexos cuja solução ótima não pode ser encontrada para todas as instâncias em tempo exequível. Embora estas técnicas sejam amplamente aplicadas em diversos setores da economia, a cronologia dos trabalhos discutidos na Seção 3.6 mostra que as aplicações na área de Engenharia de Software são recentes e pouco exploradas. O objetivo deste capítulo foi de apresentar as principais motivações que nos levaram a escolha de técnicas heurísticas como forma de aprimorar processos de Engenharia de Software.

No Capítulo 4, o planejamento das atividades de resolução de defeitos em um sistema de software será modelado como um problema de otimização combinatória. Este problema tem fortes semelhanças com alguns dos problemas de Engenharia de Software descritos neste capítulo e, dessa forma, o objetivo deste trabalho é propor um método heurístico capaz de prover uma solução aproximada para este problema.

CAPÍTULO 4

DISTRIBUIÇÃO E SEQUENCIAMENTO DAS CORREÇÕES DE BUGS

4.1. Introdução

Testes de software são processos colaborativos que reúnem diferentes partes interessadas no processo de desenvolvimento do software, tais como desenvolvedores, membros da equipe de garantia da qualidade de software, gerentes de projeto e usuários finais. Mesmo considerando que uma equipe de desenvolvimento experiente utilize as melhores práticas de garantia de qualidade durante o processo de construção do software, ainda é esperado que os usuários finais encontrem defeitos que não foram previamente identificados durante as atividades de inspeção e teste (Pressman, 2006).

Alguns projetos são construídos em parceria com os clientes, onde estes assumem papéis de membros da equipe de desenvolvimento, sendo responsáveis pela especificação dos requisitos, esclarecimentos das regras do negócio e testes de aceitação (Beck, 1999). Outros projetos optam por equipes de teste separadas da equipe de desenvolvimento. Em ambos os casos, as diferentes partes envolvidas (usuários, clientes, desenvolvedores e testadores) podem encontrar defeitos no software. Assim, canais de comunicação são necessários para garantir que os desenvolvedores sejam notificados após a identificação de anomalias no comportamento do software.

Conforme visto no Capítulo 2, BTSs são utilizados para facilitar a comunicação entre as partes envolvidas no processo de resolução de defeitos de software. Em geral, quando uma nova anomalia no software é identificada e registrada através do relatório de bug, o gerente de projeto seleciona um membro da equipe de desenvolvimento para investigar o problema registrado (triagem do relatório de bug para verificar se o problema pode ser considerado um bug) e prover a respectiva correção, caso necessário.

Em sistemas de larga escala, o número de relatórios de bug registrados diariamente no BTS pode ser muito elevado, demandando um esforço considerável da equipe de desenvolvimento para análise e correção dos problemas. A distribuição e sequenciamento destas tarefas têm grande impacto no ciclo de vida do software, visto que serão determinantes na definição das correções de bugs que serão incorporadas em sua próxima versão.

Neste contexto, é possível identificar o seguinte problema de otimização combinatória: como alocar os desenvolvedores nas atividades de correção de bugs e sequenciar estas atividades de forma a minimizar o custo de correção dos bugs de um determinado software?

Neste trabalho, define-se custo como sendo uma medida de qualidade do cronograma das tarefas, refletindo o nível de aderência do esquema aos seus objetivos: (i) reduzir o tempo médio de correção dos bugs; (ii) maximizar o número de bugs corrigidos e incorporados na próxima versão do software, e (iii) priorizar a correção dos bugs mais críticos. Quanto menor o custo do cronograma de tarefas, melhor a qualidade da solução.

Observe que os objetivos (i) e (ii) são positivamente correlacionados, pois quanto menor for o tempo médio da duração das atividades, maior será o número de correções entregues no próximo ciclo. Contudo, pode haver conflito entre estes objetivos e o objetivo (iii), visto que a abordagem de reduzir o tempo médio das atividades implica em priorizar a execução das atividades mais simples, deixando as correções mais complexas (que em alguns casos também podem ser as mais críticas para o cliente) por último. Assim, a função custo proposta na modelagem do problema deve balancear estes objetivos conflitantes.

Analisando o problema abordado, observam-se dois grandes desafios. O primeiro consiste em buscar um modelo de problema de otimização que reflita adequadamente as principais características do contexto do processo de resolução de bugs. Conforme será visto na Seção 4.5, apesar deste problema ter algumas semelhanças com problemas de otimização combinatória previamente estudados, há características do processo de resolução de bugs que fazem deste trabalho pioneiro, tornando ainda mais dificil a modelagem. Ao longo deste capítulo, veremos que o espaço de busca do conjunto solução cresce exponencialmente para grandes instâncias do problema, inviabilizando a aplicação de um método de força bruta para encontrar a solução ótima. Assim, o segundo desafio consiste em buscar um método de otimização exequível e apropriado para resolver as instâncias do problema modelado.

Este capítulo está dividido em seis seções, incluindo esta introdução. A Seção 4.2 apresenta o modelo do problema de otimização combinatória definido para distribuição e sequenciamento das correções de bugs. A Seção 4.3 descreve o processo de integração entre as variáveis de decisão do problema com as informações mantidas no BTS. A Seção 4.4 apresenta a proposta de solução baseada na aplicação de técnicas heurísticas discutidas no Capítulo 3. A Seção 4.5 faz uma análise comparativa deste problema com outros problemas de otimização combinatória. Por fim, a Seção 4.6 apresenta as conclusões deste Capítulo.

4.2. Modelo do Problema de Otimização

Considerando uma organização que desenvolve softwares e utiliza algum BTS para gerir o processo de resolução de bugs, este trabalho analisa o problema de distribuição e sequenciamento de correções de bug (PDSCB) entre os desenvolvedores de uma determinada organização, tendo como objetivo encontrar o cronograma de correções mais eficiente. Esta seção descreve o modelo do problema de otimização adaptado para o contexto do processo de resolução de bugs.

Seja C o conjunto das classes de bugs com $c \geq 1$ elementos. Cada classe de bug descreve um possível tipo de bug de acordo com o critério estabelecido por D'Ambros et al. (2007), onde um tipo de bug pode ser identificado pela associação entre produto (nome do software) e componente (módulo do software onde foi identificada a anomalia). Assim, cada elemento $c_i \in C$ é descrito pelo nome do produto e o nome do componente: $c_i = [produto_i, componente_i]$.

Seja $P = \{[ip_1, w_1], [ip_2, w_2], ... [ip_{np}, w_{np}]\}$ o conjunto dos níveis de prioridade das correções de bug, onde cada elemento $p_i \in P$ contém um identificador único do nível de prioridade (ip_i) e um peso (w_i) representado por um número inteiro positivo que define a relevância do nível de prioridade em comparação aos demais. Considere também que $w_1 > w_2 > ... w_{np}$.

Seja B o conjunto das tarefas de correção de bugs com $b \geq 1$ elementos. Cada elemento $b_i \in B$ contém o código identificador do registro do bug na base de dados do BTS (bug_id_i), a classe $c_i \in C$ do bug, a prioridade de correção $p_i \in P$ e o esforço (e_i) necessário para correção em pessoas-dia. Dessa forma, $b_i = [bug\_id_i, c_i, p_i, e_i]$.

Seja D o conjunto dos desenvolvedores de software com $d \geq 1$ elementos. Cada elemento $d_i \in D$ contém o nome do desenvolvedor ($nome_i$), o e-mail de contato do desenvolvedor ($email_i$) e a disponibilidade ($disp_i$), que define o percentual do tempo diário do desenvolvedor que estará dedicado para correções de bug, onde $0 < disp_i \leq 100\%$. Assim, $d_i = [nome_i, email_i, disp_i]$.

Para efeito de construção do modelo, assume-se que cada desenvolvedor pode executar no máximo uma tarefa de correção de bug por vez (restrição R_1) e cada tarefa será executada somente uma única vez (restrição R_2). Quando um desenvolvedor inicia a execução de uma correção, ele a executa até o final sem interrupções (restrição R_3). Além disso, o modelo também restringe a alocação das tarefas de correção entre os desenvolvedores, estabelecendo que somente desenvolvedores com experiência anterior na correção de bugs de um determinado tipo podem assumir correções de bugs da respectiva classe (restrição R_4).

Assim, torna-se necessário um instrumento para mapear as habilidades de cada desenvolvedor em função das correções de bugs previamente realizadas. Seja a matriz SK = (sk_{ij}) de tamanho |D| x |C|, onde $0 \leq sk_{ij} \leq 1$ e cada elemento sk_{ij} da matriz SK define o nível de experiência anterior do desenvolvedor i na correção de bugs da classe j. Há também outras duas restrições impostas ao modelo: deve haver ao menos um desenvolvedor experiente para cada tipo de bug (restrição R_5) e o somatório do nível de experiência de todos os desenvolvedores para uma determinada classe de bug deve ser igual a 1 (restrição R_6), conforme apresentado na Equação 4.1.

$$\sum_{i=1}^{|D|} sk_{ij} = 1 \quad \forall j \in C \tag{4.1}$$

A Tabela 4.1 apresenta um exemplo de uma matriz SK, considerando os conjuntos D = {D_1, D_2, D_3} de desenvolvedores e C = {C_1, C_2} de classes de bug. Note que o desenvolvedor D_1 tem experiência prévia com correções de todas as classes de bugs, enquanto os desenvolvedores D_2 e D_3 têm apenas experiência prévia com correções de bugs das classes C_2 e C_1, respectivamente. Tendo em vista que a restrição R_4 limita a atribuição das correções a desenvolvedores com experiência anterior nas correções de bugs de uma determinada classe, uma alocação viável deve impedir a alocação de correções de bugs das classes C_1 e C_2 para os desenvolvedores D_2 e D_3, respectivamente.

Tabela 4.1. Exemplo de Matriz SK

	C1	C2
D_1	0.4	0.3
D_2	0.0	0.7
D_3	0.6	0.0

Seja S o conjunto de todas as soluções viáveis baseadas nos conjuntos C, P, B e D, na matriz SK e nas restrições impostas ao modelo. Seja um cronograma de tarefas

de correção de bugs $E \in S$, onde cada elemento $i \in E$ é formado por um trio composto de um desenvolvedor $d_i \in D$, um número inteiro k_i, que representa a quantidade de tarefas de correção de bugs que serão executadas por d_i, e pelo subconjunto $B_i \subset B$ que contém a lista das tarefas de correção de bugs que serão executadas por d_i. Assim, tem-se que $E = \{[d_1, k_1, \{b_{11}, b_{12}, ..., b_{1k1}\}], [d_2, k_2, \{b_{21}, b_{22}, ..., b_{2k2}\}], ..., [d_n, k_n, \{b_{n1}, b_{n2}, ..., b_{nkn}\}]\}$.

Para calcular o custo de um determinado esquema $E \in S$, devem-se calcular dois atributos para cada tarefa de correção de bug: a duração e o tempo de término. A duração de uma tarefa de correção de bug consiste no intervalo de tempo medido entre o momento em que o desenvolvedor iniciou a tarefa e o momento de término da mesma. A duração $d_j(E, i)$ de cada tarefa de correção de bug $b_{ij} \in B_i$ é calculada a partir da divisão entre o esforço (e_j) estimado para a tarefa e a disponibilidade ($disp_i$) do desenvolvedor $d_i \in D$ alocado para correção. O tempo de término $CT_j(E, i)$ de cada tarefa de correção de bug $b_{ij} \in B_i$ em um determinado esquema $E \in S$ consiste no intervalo de tempo medido entre o início das atividades do desenvolvedor $d_i \in D$ alocado para a correção até o término da atividade b_{ij}. Com base na duração e no sequenciamento das correções de bugs, calcula-se o tempo de término de cada correção de bug $b_{ij} \in B$ a partir da soma entre a duração $d_j(E,i)$ e a duração total das atividades precedentes a b_{ij}, conforme definido na Equação 4.2.

$$CT_j(E,i) = \sum_{x=1}^{j} d_x(E,i) \qquad (4.2)$$

A função objetivo deve ter por finalidade avaliar a aderência de uma determinada solução com os objetivos (i), (ii) e (iii) declarados na introdução deste Capítulo. Para atender os objetivos (i) e (ii), a função objetivo mais apropriada consiste na avaliação do somatório do tempo de término das correções de bug. Dessa forma, define-se a melhor solução como a configuração com o menor somatório dos tempos de término das correções de bug. Contudo, tendo em vista a relevância dos níveis de prioridade para o modelo, deve-se considerar o somatório dos tempos de término ponderado pelos pesos das prioridades das respectivas tarefas de correção de bugs efetuadas pelos desenvolvedores. Assim, tem-se que o custo de um determinado esquema $E \in S$ é calculado conforme definido na Equação 4.3.

$$Custo\ (E) = \sum_{i=1}^{|D|} \sum_{j=1}^{K_i} w_j * CT_j(E,i) \qquad (4.3)$$

A Figura 4.1 descreve o modelo conceitual do PDSCB. Na ausência de uma representação padrão de modelos de problema de otimização, optou-se por uma representação baseada no diagrama Entidade-Relacionamento (Elmasri e Navathe, 2005). Este diagrama apresenta os principais objetos do modelo (retângulos), seus atributos (elipses) e os relacionamentos (losangos) entre os objetos com suas respectivas cardinalidades. Dentre os relacionamentos representados, dois foram nomeados para evidenciar sua relevância dentro do modelo. A matriz SK foi representada na forma de um relacionamento de ordem N_c x N_d entre classes de bug e desenvolvedores, enquanto que a entidade associativa *alocação* mapeia cada um dos possíveis esquemas $E \in S$.

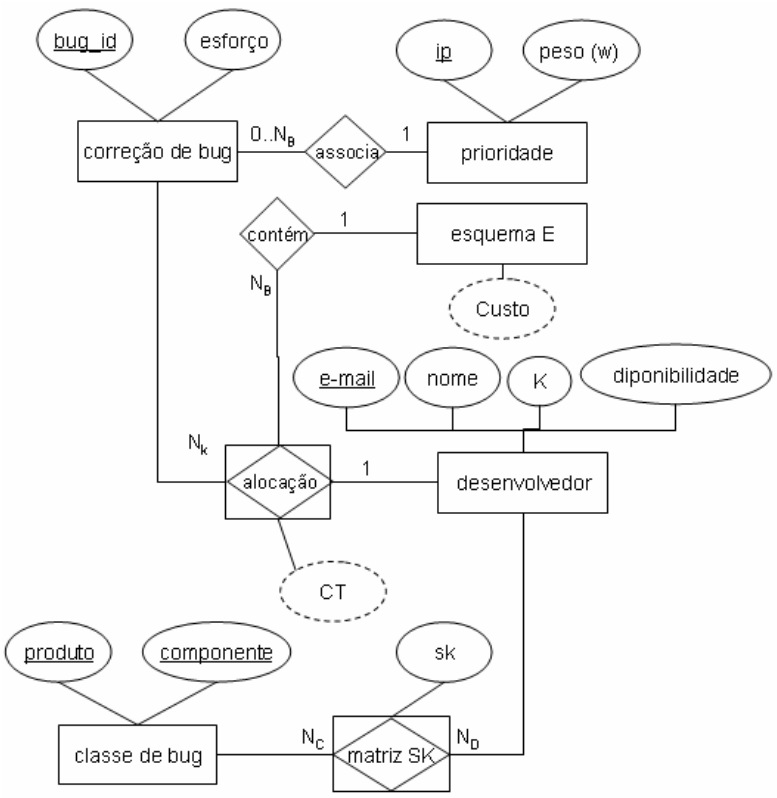

Figura 4.1. Representação do Modelo do PDSCB

Para ilustrar o cálculo destes atributos, segue um exemplo. Considere o conjunto D = {D_1, D_2, D_3} de desenvolvedores, o conjunto C = {C_1, C_2} de classes de bug, o conjunto B = {B_1, B_2, B_3, B_4} de tarefas de correção de bugs que devem ser executadas e a matriz SK definida na Tabela 4.1. Considere que todos os

desenvolvedores estejam 100% do tempo dedicados a tarefas de correção de bugs e o conjunto P = {(P1, 5), (P2, 4), (P3, 3), (P4, 2), (P5, 1)} dos níveis de prioridade das correções de bug. A Tabela 4.2 apresenta os dados deste exemplo.

Tabela 4.2. Conjunto das Tarefas de Correção de Bugs

Tarefas de Correção de Bugs	B1	B2	B3	B4
Esforço (e_j)	5	2	1	3
Classe de Bug (c_j)	C1	C1	C2	C2
Prioridade (np_j)	P4	P3	P5	P5
Peso (w_j)	2	3	1	1

A Figura 4.2 apresenta um possível cronograma das tarefas de correção de bugs para o cenário descrito. Note que a solução proposta está em conformidade com as restrições R_1, R_2, R_3 e R_4 impostas pelo modelo (as restrições R_5 e R_6 são supridas pela matriz SK). Para a configuração proposta, tem-se que $CT_1(E,1) = 2$, $CT_2(E,1) = 5$, $CT_1(E,2) = 1$ e $CT_1(E,3) = 5$.

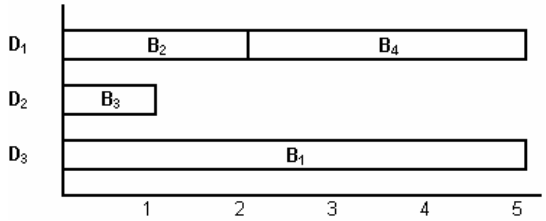

Figura 4.2. Exemplo de uma Solução Viável para o Cenário Proposto

4.3. Integração com o Bug Tracking System

A seção anterior apresentou o modelo do PDSCB. Antes de submeter este modelo a técnicas de otimização, é necessário carregar os dados da instância do problema e preencher os conjuntos dos parâmetros de entrada.

A geração das instâncias do problema demanda uma integração com o BTS que mantém as informações do processo de resolução de bugs, conforme visto no Capítulo 2. Nesta seção será descrito o processo de extração dos dados da base de dados do BTS e geração das instâncias do PDSCB descrito na Seção 4.1.

Inicialmente, deve-se mapear o conjunto D de desenvolvedores. Nesta etapa, a lista de todos os desenvolvedores cadastrados no BTS deve ser apresentada ao gerente de projeto, que deverá selecionar os desenvolvedores que atuarão nas correções de bugs. Assim, constitui-se o conjunto D de desenvolvedores a partir dos indivíduos selecionados pelo gerente de projeto. Os atributos $nome_i$ e $email_i$ de cada elemento $d_i \in D$ são carregados automaticamente do BTS, ao passo que o gerente de

projeto deve informar o nível de disponibilidade de cada desenvolvedor ($disp_i$), onde $0\% < disp_i \leq 100\%$.

Os relatórios de bugs do BTS são classificados em níveis de prioridade (P_1, P_2, ..., P_n). O gerente de projeto deve atribuir um peso w para cada um dos níveis de prioridade, tal que $w_1 > w_2 > w_3 > ... > w_{np}$. Com isso, constitui-se o conjunto P de níveis de prioridade de bugs, onde cada elemento $p_i \in P$ representa a associação entre o código identificador do nível de prioridade e o respectivo peso atribuído pelo gerente de projeto.

Em seguida, deve-se mapear o conjunto B das tarefas de correção de bugs. A lista de todos os relatórios de bugs pendentes de correção deve ser exibida para o gerente de projeto, que deverá selecionar o sub-conjunto de tarefas de correção de bugs que devem ser incorporadas na próxima versão do software. Assim, constitui-se o conjunto B de tarefas de correção a partir dos elementos selecionados pelo gerente de projeto. Os atributos bug_id_i, p_i e e_i correspondem respectivamente aos atributos BUG_ID, PRIORITY e EFFORT de cada relatório de bug. O conjunto C de classes de bug consiste no conjunto definido por todas as associações possíveis entre os atributos PRODUCT e COMPONENT dos relatórios de bug selecionados.

Por fim, de posse dos conjuntos D e C, a matriz SK é construída. Cada elemento sk_{ij} desta matriz representa a fração do total de correções realizadas por cada desenvolvedor $i \in D$ sobre o total de correções de bugs da classe $j \in C$ já efetuadas, desconsiderando as correções de bugs efetuadas pelos desenvolvedores não selecionados pelo gerente de projeto.

Segue um exemplo para ilustrar a construção da matriz SK. Sejam $D = \{D_1, D_2, D_3\}$ o conjunto dos desenvolvedores, $C = \{C_1, C_2\}$ o conjunto das classes de bugs e CR a matriz com o mapeamento do histórico de tarefas correção de bugs executadas extraído do BTS (onde cada elemento cr_{ij} da matriz CR representa a quantidade de tarefas de correções de bugs da classe j executadas pelo desenvolvedor i). A Tabela 4.3 mostra um exemplo da matriz CR.

Tabela 4.3. Histórico das Tarefas de Correção de Bugs Executadas

	C_1	C_2
D_1	40	15
D_2	0	35
D_3	60	0

Considerando este cenário, observa-se que os desenvolvedores D_1, D_2 e D_3 executaram respectivamente 40%, 0% e 60% do total das tarefas de correção de bugs

da classe C_1, enquanto que 30%, 70% e 0% do total das tarefas de correção de bugs da classe C_2 foram executadas pelos mesmos desenvolvedores. Assim, tem-se que a matriz SK correspondente está representada na Tabela 4.1.

4.4. Proposta de Solução

O espaço de busca do PDSCB cresce na proporção de $N_d!$ x $N_b!$, onde N_d e N_b representam respectivamente o número de desenvolvedores e a quantidade de relatórios de bugs com resolução pendente. Por exemplo, para as instâncias dos projetos Eclipse e Firefox, considerando o número de desenvolvedores registrados e a quantidade de bugs pendentes até maio/2008, haveria respectivamente cerca de (1.099! x 28.226!) e (175! x 15.543!) soluções possíveis.

Em função do crescimento fatorial do espaço de busca, faz-se necessário um método de busca de soluções para instâncias do problema em questão. Este método deve ser baseado em alguma técnica de otimização apresentada no Capítulo 3.

O crescimento em escala exponencial do espaço de busca inviabiliza a aplicação das buscas sistemáticas. Além disso, a programação linear também não pode ser aplicada, pois as variáveis de decisão são discretas. Não há nenhuma restrição quanto à aplicação do *Branch and Bound*, mas a dificuldade para implementação deste algoritmo (e, posteriormente, ajustes em função de mudanças no modelo) torna-o menos atraente. Assim, a solução mais apropriada consiste na aplicação de técnicas heurísticas para encontrar uma solução aproximada para o PDSCB.

O método de otimização proposto neste trabalho consiste na especialização de um algoritmo genético combinado com uma heurística construtiva. A escolha da metaheurística do algoritmo genético está sustentada na aderência desta técnica com os problemas de construção de cronogramas para projetos (que tem semelhanças com o PDSCB), conforme observado por Alba e Chicano (2007) e Antoniol et al (2005). A heurística construtiva será utilizada em conjunto com o algoritmo genético para aprimorar o processo de seleção das soluções candidatas a cada iteração.

Um problema bastante recorrente nos algoritmos genéticos é a geração de soluções inviáveis (Xu e Bean, 2007), isto é, soluções geradas pelos processos de mutação e *crossover* que violam alguma das restrições impostas ao problema. Para suprir esta limitação, a representação dos cromossomos deve ser impedir a geração de soluções inviáveis. Neste trabalho, a técnica de representação de cromossomos *Random Key Genetic Algorithm* (RKGA), introduzida por Bean (1994), foi escolhida pelo fato desta garantir a viabilidade de todos os cromossomos gerados.

No algoritmo RKGA adaptado para o contexto do problema de sequenciamento e distribuição de tarefas de correção de bugs, cada cromossomo representa uma possível configuração de distribuição e sequenciamento das correções de bug entre os desenvolvedores. A codificação dos cromossomos é definida como um vetor de N_b genes, onde N_b corresponde ao número de tarefas de correção de bugs. Cada gene contém um número real aleatório definido no intervalo [0,1]. Dado um cromossomo, o esquema E de distribuição e sequenciamento das tarefas de correção de bugs correspondente é construído em duas etapas: (i) alocação dos desenvolvedores para executar as correções de bug e (ii) ordenação das correções de bug de cada desenvolvedor.

Para cada tarefa de correção de bug representada pelos genes do cromossomo, deve-se atribuir um desenvolvedor para execução da tarefa correspondente. O algoritmo lê o valor do alelo de cada gene e atribui um desenvolvedor para proceder com a correção do respectivo bug. A seleção do desenvolvedor para atuar na tarefa de correção de um determinado bug leva em consideração os dados mantidos na matriz SK e na análise da distribuição de frequência acumulada das correções de bug.

Seja a matriz SA, com as mesmas dimensões da matriz SK, que representa a distribuição de frequência acumulada das correções de bugs previamente realizadas. A matriz SA é construída a partir da matriz SK utilizando-se a Equação 4.4. Cada célula da matriz SA corresponde ao somatório dos valores dos elementos da matriz SK na mesma coluna até a linha corrente. A Tabela 4.4 apresenta a matriz SA correspondente para a matriz SK apresentada na Tabela 4.1. A matriz SA tem as seguintes propriedades: (a) todo elemento sa_{ij} é sempre maior ou igual a zero; (b) para qualquer coluna, o valor de cada linha é sempre igual ou maior ao valor da linha anterior e; (c) para qualquer coluna, o valor da última linha é sempre igual a 1.

$$sa_{ij} = \sum_{z=0}^{i} sk_{zj} \qquad (4.4)$$

Tabela 4.4. Matriz SA gerada a partir da matriz SK definida na Tabela 4.1

	BC1	**BC2**
D_1	0.4	0.3
D_2	0.4	1.0
D_3	1.0	1.0

O primeiro passo do processo de seleção de um desenvolvedor para correção de um determinado bug consiste na identificação da classe do bug, representada na forma de colunas na matriz SA. Em seguida, o número real aleatório é recuperado do gene e busca-se na coluna correspondente à classe do bug a primeira linha cujo valor seja maior ou igual ao valor do alelo do gene. O desenvolvedor representado pela linha identificada na busca será alocado na tarefa de correção do bug em questão. Este processo é repetido para cada bug e garante que a distribuição das tarefas de correção entre os desenvolvedores é proporcional ao histórico de correções efetuadas por cada desenvolvedor na classe do bug em questão.

Para ilustrar o procedimento descrito, considere o exemplo definido na Tabela 4.2. Para a tarefa de correção de bug B_1 há dois desenvolvedores (D_1 e D_3) com a habilidade necessária para execução (ver coluna C_1 da matriz SK). A Figura 4.3 apresenta o gráfico da distribuição da frequência acumulada das correções de bugs realizadas por cada desenvolvedor, de acordo com a matriz SA representada na Tabela 4.4. Observe que os desenvolvedores D_1 e D_3 têm, respectivamente, 40% e 60% de chance de serem selecionados para correção de um bug da classe C_1. Conforme descrito no parágrafo anterior, o processo de distribuição de tarefas de correção de bugs lê o valor do alelo do gene e busca na matriz SA o intervalo onde se encontra o número recuperado. Por exemplo, se o valor lido for menor ou igual a 0,4, então o desenvolvedor D_1 será escolhido para atuar na correção do bug. Caso contrário (se o valor for maior do que 0,4), o desenvolvedor D_3 será selecionado.

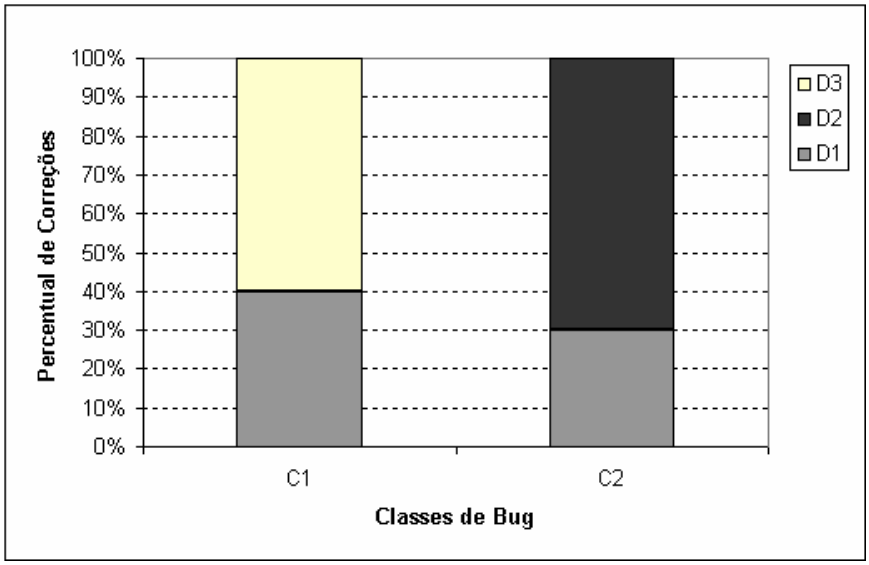

Figura 4.3. Distribuição das Correções de Bugs

Após a distribuição das tarefas de correção de bugs entre os desenvolvedores é preciso definir, para cada desenvolvedor, a ordem de execução das respectivas tarefas. O procedimento de sequenciamento destas tarefas baseia-se na heurística construtiva LIST SCHEDULING (Skutella, 1998). Esta heurística determina que as tarefas devem ser executadas de acordo com a Regra da Razão de Smith (Smith, 1956), uma variação da heurística *Shortest Processing Time* (Bruno et al., 1974) que estabelece que as tarefas devem ser executadas na ordem ascendente da razão d/w, onde d e w representam, respectivamente, a duração e o peso de cada tarefa. Assim, temos que as tarefas mais rápidas e mais prioritárias são executadas primeiro.

Para exemplificar o processo de ordenação das tarefas, considere novamente o exemplo descrito na Tabela 4.2. Considere também que as tarefas B_2 e B_4 foram distribuídas para o mesmo desenvolvedor. Dado que as razões d/w destas tarefas são, respectivamente, 0,67 e 3, de acordo com a heurística LIST SCHEDULING a execução da tarefa B_2 deve preceder a execução de B_4.

O algoritmo genético desenvolvido neste trabalho cria uma população inicial de cromossomos (indivíduos) gerados aleatoriamente. Para cada um dos cromossomos faz-se o mapeamento do cronograma correspondente e calcula-se o custo do esquema, conforme definido pela Equação 4.3. Os cromossomos são então ordenados em ordem ascendente do custo calculado para o esquema correspondente. Quanto menor o custo, melhor a qualidade da solução.

Em seguida, a partir da população inicial ordenada, ocorre o processo de reprodução que define a geração de novos cromossomos ao longo de diversas iterações. A estratégia de reprodução adotada pelo algoritmo genético desenvolvido neste trabalho foi inspirada nos trabalhos de Xu e Bean (2007) e Gonçalves et al. (2008), e consiste dos seguintes procedimentos:

1. Selecionar os T % melhores indivíduos (cromossomos com menor custo) da solução corrente para compor a próxima geração;

2. I % dos novos indivíduos (percentil de novos indivíduos) são gerados pelo processo de recombinação (*crossover*). Para proceder com a recombinação, um dos cromossomos pertencente ao conjunto dos T % melhores indivíduos é aleatoriamente selecionado (pai 1), enquanto que outro cromossomo (pai 2) é aleatoriamente selecionado do conjunto total de indivíduos da população corrente. Para cada gene do cromossomo filho, sorteia-se um número real $X \in [0,1]$. Se X for menor ou igual ao fator C % (percentil de recombinação), então o gene de pai 1 é selecionado. Caso

contrário, seleciona-se o gene do pai 2. Assim, tem-se que o cromossomo filho é composto por uma sequência de genes dos cromossomos pais;

3. (1 – T % - I %) dos novos indivíduos são aleatoriamente gerados (mutantes).

O processo descrito acima é executado até que um critério de parada seja alcançado. O critério adotado pelo algoritmo genético em questão consiste em parar a rotina após a avaliação de G gerações.

4.5. Comparação com Trabalhos Semelhantes

O PDSCB tem semelhanças com o problema NP-Difícil de escalonamento de processos com prioridades diferentes em um ambiente com múltiplos processadores paralelos, cujo objetivo é minimizar o somatório ponderado dos tempos de término das tarefas. Este problema foi previamente estudado por Bruno et al. (1974) e Kumar et al. (2007). Fazendo uma analogia com o PDSCB, temos que os processos e os processadores paralelos correspondem, respectivamente, às tarefas de correção de bugs e aos desenvolvedores.

Contudo, há algumas diferenças entre estes problemas. No PDSCB as tarefas de correções de bug são categorizadas pelas classes de bug e somente desenvolvedores com habilidades adequadas podem ser alocados a esta tarefas. Além disso, cada desenvolvedor tem um nível de disponibilidade para atuação nas tarefas de correção de bug, enquanto no problema de escalonamento de processos as máquinas estão 100% do tempo dedicadas à execução das tarefas.

Podemos definir o PDSCB como uma variação do Problema de Construção de Cronogramas de Projeto (PCCP), apresentado no Capítulo 3, que foi adaptado para o processo de resolução de bugs. Assim, observa-se que este trabalho está inserido no contexto da Engenharia de Software Baseada em Busca, onde um problema do processo de resolução de bugs, recorrente na área de Engenharia de Software, foi modelado como um problema de otimização combinatória e cuja proposta de solução consiste na aplicação de uma técnica baseada no uso de heurísticas para encontrar uma solução aproximada.

Comparando este trabalho com o trabalho de Alba e Chicano (2007), notam-se algumas similaridades entre eles. Ambos propõem métodos baseados no uso de algoritmos genéticos para construir cronogramas de tarefas e promover sua distribuição entre os membros da equipe do projeto de desenvolvimento de software, com a finalidade de encontrar configurações capazes de reduzir o custo e a duração

das atividades. Inspirado nos resultados obtidos no trabalho de Alba e Chicano (2007), este trabalho faz um refinamento do modelo do PCCP adaptado para o contexto do processo de resolução de bugs, considerando a taxonomia dos relatórios de bug mantidos na base de dados do BTS.

A principal diferença entre estes trabalhos consiste que o corrente aborda um contexto mais específico e restrito ao processo de resolução de bugs, enquanto que Alba e Chicano (2007) endereçam um problema mais genérico, onde as tarefas podem ser qualquer atividade do ciclo de desenvolvimento de software, desde a elaboração dos documentos de requisitos até a codificação e teste de programas. O ambiente definido no processo de resolução de bugs é bem diferente do contexto do processo de desenvolvimento de um novo software.

No modelo definido por Alba e Chicano (2007), as metas principais são a redução do custo e do prazo do novo desenvolvimento, aumentando desta forma a margem de lucro e antecipando a geração de receitas. Por outro lado, o custo das correções dos bugs geralmente não é repassado ao cliente, e o orçamento para execução destas tarefas está vinculado ao pagamento periódico das taxas de manutenção do software. Assim, deve ser interesse do fornecedor aprimorar a qualidade do software (que consequentemente reduz o número de bugs identificados pelos usuários finais) e prover o agendamento das correções de forma a minimizar o tempo gasto nestas atividades. Quanto mais rápido for término da tarefa de correção de um bug, mais rápido o desenvolvedor estará disponível para implementar novas soluções e, consequentemente, gerar receitas para o fornecedor. Enquanto isso, o cliente espera que os bugs mais críticos sob sua perspectiva deveriam ser corrigidos o mais cedo possível.

Assim, optou-se neste trabalho em modelar uma função objetivo baseada no somatório ponderado (pelas prioridades dos bugs) dos tempos de término das correções de bug, a fim de balancear os objetivos das partes interessadas (cliente e fornecedor). Adicionalmente, este trabalho propõe a construção de um ferramental capaz de extrair informações relevantes da base de dados do BTS capazes de subsidiar a construção de bons cronogramas das tarefas de correção de bugs.

4.6. Conclusão

Este capítulo abordou o problema de distribuição e sequenciamento de tarefas de correção de bugs de diferentes prioridades para um conjunto de desenvolvedores disponíveis, a fim de atingir as aspirações dos clientes e dos fornecedores envolvidos no processo de resolução de bugs. Tendo em vista a relevância deste problema para a

indústria de software e a respectiva complexidade de sua resolução, observa-se a necessidade de uma ferramenta automatizada para prover bons esquemas de agendamento das tarefas de correção de bugs.

A proposta de solução apresentada neste capítulo consiste de um método que captura informações relevantes da base de dados do BTS, que em conjunto com informações fornecidas pelo gerente de projeto, são transformadas em instâncias do PDSCB e submetidas a uma técnica de otimização, baseada no uso de um algoritmo genético em associação com uma heurística construtiva, a fim de buscar uma configuração do esquema de agendamento próxima à solução ótima. No Capítulo 5, serão conduzidos experimentos utilizando dados do projeto Eclipse para avaliar a eficácia da solução proposta.

CAPÍTULO 5
AVALIAÇÃO DA SOLUÇÃO PROPOSTA

5.1. Introdução

O Capítulo 4 apresentou o modelo do problema de distribuição e sequenciamento de correções de bug (PDSCB), descreveu a interface de integração com o BTS e endereçou uma proposta de solução baseada no uso de técnicas heurísticas para encontrar soluções candidatas. Este capítulo tem por finalidade descrever os testes conduzidos para avaliar a eficácia da proposta apresentada nesta dissertação.

Com base no conjunto de bugs registrados na comunidade de software livre do projeto Eclipse, extraiu-se uma amostra dos relatórios de bugs armazenados no Eclipse Bugzilla, derivando desta forma as instâncias do PDSCB utilizadas nas análises. Além de verificar a eficácia da solução proposta, os testes avaliam diferentes configurações dos parâmetros do algoritmo genético (percentil de *crossover*, percentil de seleção dos melhores indivíduos, percentil de geração de novos indivíduos, percentil de geração de indivíduos mutantes, tamanho da população e critério de parada) e os respectivos impactos na qualidade e no tempo de resposta do método.

Este capítulo está dividido em seis seções, incluindo esta introdução. A Seção 5.2 faz uma breve descrição do projeto Eclipse e os principais softwares desenvolvidos. A Seção 5.3 descreve a construção dos cenários de dados utilizados nos testes com base nos dados extraídos do projeto Eclipse. A Seção 5.4 descreve os resultados obtidos nos testes considerando um cenário de configuração padrão dos parâmetros dos algoritmos genéticos. A Seção 5.5 faz uma análise de sensibilidade dos parâmetros do algoritmo genético, avaliando os impactos na qualidade da solução e no tempo de resposta. Por fim, a Seção 5.6 faz uma análise conclusiva dos resultados obtidos nos testes.

5.2. Projeto Eclipse

Eclipse[11] é uma comunidade de software livre cujos projetos visam a construção de plataformas abertas para desenvolvimento de aplicações, incluindo

ferramentas para construção, implantação e gerenciamento de software, permeando todo o ciclo de vida. O termo Eclipse é geralmente associado à ferramenta Java Integrated Development Environment (Java IDE), que é muito utilizada no mercado para o desenvolvimento de programas na linguagem de programação Java. Contudo, a comunidade Eclipse mantém mais de 60 projetos de software, que são categorizados em sete classes: Enterprise Development, Embedded and Device Development, Rich Client Platform, Rich Internet Applications, Application Frameworks, Application Lifecycle Management (ALM) e Service Oriented Architecture (SOA).

A comunidade de software livre do Eclipse é financiada mediante contribuições dos membros e gerida por um conselho de administração composto por representantes das principais comunidades de software livre, bem como desenvolvedores e clientes estratégicos. O Eclipse emprega uma equipe profissional em tempo integral para prestar serviços à comunidade, mas não emprega desenvolvedores de código aberto, conhecidos como *committers*, que normalmente são empregados de outras organizações ou desenvolvedores independentes e voluntários.

O Eclipse provê serviços para a comunidade de software livre, incluindo os repositórios de código CVS[12] e SVN[13], o sistema Bugzilla[14], as listas de discussão e os sites dos projetos de software relacionados. Estes serviços podem ser acessados a partir do portal do projeto Eclipse que oferece um ambiente colaborativo para suportar os processos de desenvolvimento, manutenção e divulgação dos softwares mantidos pela comunidade.

5.3. Construção dos Cenários de Dados para Análise

O sistema Eclipse Bugzilla segue o modelo convencional de um BTS, apresentado no Capítulo 2. Trata-se de um portal Web, onde os usuários finais cadastram os relatórios de bugs para os softwares mantidos pelo Eclipse e acompanham as respectivas resoluções. Este sistema também atua como um canal de comunicação entre os usuários e os desenvolvedores dos softwares da comunidade, que estão geograficamente espalhados, onde informações sobre bugs, melhorias e novos releases são discutidas. A base de dados deste sistema contém todos os registros dos

[11] http://www.eclipse.org
[12]http://dev.eclipse.org/viewcvs/index.cgi/
[13]http://dev.eclipse.org/viewsvn/index.cgi/
[14]https://bugs.eclipse.org/bugs/

relatórios de bugs, bem como informações cadastrais dos usuários, desenvolvedores e demais membros da comunidade Eclipse.

Além do formato convencional HTML para apresentação dos relatórios de bugs, o Bugzilla também permite a visualização dos relatórios de bug no formato XML. Assim, o processo de extração dos dados do Eclipse Bugzilla consistiu no download dos relatórios de bug no formato XML. No decorrer deste trabalho, foram capturados mais de 200.000 registros de bugs, contemplando a série histórica de outubro/2001 a dezembro/2008. A Tabela 5.1 apresenta a distribuição dos relatórios de bug categorizados por *STATUS* (estado corrente no ciclo de vida do processo de resolução de bug), enquanto que a Tabela 5.2 apresenta a distribuição de resoluções aplicadas.

Tabela 5.1. Distribuição dos Relatórios de Bugs Categorizados por Status

SITUAÇÃO	*STATUS*	*TOTAL*
CONCLUÍDO		**216.175**
	CLOSED	47.944
	RESOLVED	136.461
	VERIFIED	31.770
EM ANDAMENTO		**26.814**
	ASSIGNED	5.406
	NEW	20.883
	REOPENED	525
TOTAL		**242.989**

Tabela 5.2. Distribuição das Resoluções Aplicadas aos Bugs Concluídos

RESOLUÇÃO	*TOTAL*	*%*
FIXED	131.842	60,99%
INVALID	14.247	6,59%
WONTFIX	14.375	6,65%
LATER	4.512	2,09%
REMIND	1.220	0,56%
DUPLICATE	31.725	14,68%
WORKSFORME	16.949	7,84%
NOT_ECLIPSE	1.305	0,60%
TOTAL	**216.175**	**100%**

Atualmente, não há nenhum processo formal para distribuição das tarefas de correção identificadas pelos usuários finais dos softwares do Eclipse. Periodicamente, os membros da equipe de qualidade do Eclipse Bugzilla avaliam os relatórios de bugs encaminhados e fazem uma triagem, para verificar possíveis inconsistências e redundâncias. Feita a triagem, se for constatado que o relatório de bug descreve realmente uma anomalia de que deva ser corrigida, ele é encaminhado para um grupo de e-mail contendo alguns desenvolvedores que decidem entre si quem será o responsável pela correção. Outra abordagem comum é quando algum desenvolvedor

coloca-se como voluntário para corrigir um determinado problema e assume a responsabilidade pelo respectivo relatório de bug.

Para fins de avaliação da solução proposta, foram selecionados os relatórios de bug do Eclipse Bugzilla cuja situação encontrava-se em "Em Andamento" até 31/12/2007 e que foram resolvidos ao longo de 2008. A amostra utilizada nos testes contemplou os relatórios de bug registrados para os três produtos mais críticos (aqueles que concentraram o maior número de relatórios de bugs cadastrados durante o período analisado): Eclipse Platform (Platform), Model Development Tools (MDT) e Web Tools Platform (Web Tools). A Tabela 5.3 apresenta a quantidade de relatórios de bug por produto selecionados para os testes.

Tabela 5.3. Relatórios de Bug Selecionados para os Testes

Produto	Relatórios de Bug
Platform	599
MDT	493
Web Tools	437
Total	**1.529**

O conjunto C das classes de bugs considerou os pares formados pelas associações entre os campos Produto (*PRODUCT*) e Componente (*COMPONENT*) dos relatórios de bugs selecionados. Ao todo, foram mapeadas 53 classes de bugs. A Tabela 5.4 apresenta a relação por produto das cinco classes que concentraram o maior número de relatórios de bugs.

Tabela 5.4. Principais Classes de Bugs dos Produtos Selecionados

Produto	Componente	Nº de Relatórios de Bugs	(%) do Total dos Relatórios Bugs
Platform	UI	263	20,90%
	SWT	134	10,60%
	User Assistance	32	2,50%
	Text	28	2,20%
	Debug	28	2,20%
MDT	UML2	249	19,80%
	OCL	121	9,60%
	UML2Tools	79	6,30%
	XSD	21	1,70%
	EODM	11	0,90%
Web Tools	wst.server	48	3,80%
	jst.ws	43	3,40%
	jst.j2ee	40	3,20%
	releng	30	2,40%
	wst.ws	29	2,30%

O Eclipse Bugzilla define cinco níveis de prioridade (P1, P2, P3, P4, P5), onde os níveis P1 e P5 correspondem, respectivamente, aos níveis de maior e menor

prioridade. Neste trabalho, o conjunto P = {[P1, 5], [P2, 4], [P3, 3], [P4, 2], [P5, 1]} dos níveis de prioridade foi considerado durante os testes.

O conjunto B de tarefas de correção de bugs considerou todos os relatórios de bugs pendentes selecionados para fins da avaliação. Para cada tarefa $b_i \in B$, foi feito o seguinte mapeamento:

- O atributo bug_id_i recebe o valor do campo *BUG_ID* do relatório de bug;
- A classe de bug c_i corresponde à associação entre os campos *PRODUCT* e *COMPONENT* do relatório de bug;
- A prioridade de correção p_i recebe o valor do atributo *PRIORITY* correspondente no conjunto P;
- Como na versão do Eclipse Bugzilla analisada não há atributo de esforço estimados nos relatórios de bug, assumiu-se o esforço para correção e_i como sendo o trabalho de um desenvolvedor em tempo integral durante a duração real do processo de resolução do bug registrada.

O conjunto D dos desenvolvedores consistiu na lista dos desenvolvedores do Eclipse Bugzilla que resolveram ao menos um relatório de bug até 31/12/2007 pertencente a uma das classes de bugs do conjunto C mapeado a partir dos relatórios de bugs selecionados para os testes. Ao todo, foram mapeados 377 desenvolvedores. Para cada desenvolvedor $d_i \in D$, os atributos $nome_i$ e $email_i$ correspondem, respectivamente, ao nome e e-mail de contato dos desenvolvedores, obtidos a partir dos relatórios do Eclipse Bugzilla, enquanto que assumiu-se uma disponibilidade para correção de bugs ($disp_i$) igual a 100%, tendo em vista que esta informação não está registrada no Eclipse Bugzilla.

A matriz SK de habilidades dos desenvolvedores consistiu de uma matriz de ordem 377 x 53. Cada elemento sk_{ij} desta matriz registra a razão entre o número de relatórios de bugs da classe j resolvidos pelo desenvolvedor i e o total de relatórios de bugs da classe j resolvidos pelos 377 desenvolvedores selecionados.

De posse dos dados extraídos do Eclipse Bugzilla, foram construídos dois cenários de dados para os testes. O Cenário 1 contempla somente os relatórios de bugs do componente SWT, que é um dos módulos mais críticos do Eclipse Platform, de acordo com o número de relatórios de bugs reportados diariamente. O Cenário 2 contempla todos os 1.529 relatórios de bugs extraídos. Consequentemente, o número de desenvolvedores do Cenário 1 é menor, pois este cenário contempla somente os desenvolvedores habilitados a corrigir bugs da classe [Platform, SWT]. A Tabela 5.5 apresenta a definição dos cenários utilizados nos testes.

Tabela 5.5. Definição dos Cenários das Instâncias

Cenário	Distribuição dos Relatórios de Bugs por Níveis de Prioridade						Nº de Desenvolvedores
	P1	P2	P3	P4	P5	Total	
Cenário 1	0	5	128	1	0	134	42
Cenário 2	43	113	1.335	16	22	1.529	377

O objetivo destes cenários é avaliar a influência do tamanho do espaço de busca do problema na qualidade da solução gerada pelo método proposto e no respectivo tempo de resposta, em comparação com os procedimentos informais que nortearam os esquemas de agendamento das tarefas de correção realizadas. Tendo em vista que o aumento no número de tarefas de correção de bugs implica em um esforço ainda maior para construir um bom esquema de agendamento de execução destas tarefas, acredita-se que a solução proposta pelo método heurístico pode ser mais apropriada nos casos onde um espaço de busca muito grande compromete a qualidade das soluções propostas por métodos informais. Esta hipótese deve ser avaliada durante os testes através das análises obtidas em cada um destes cenários.

5.4. Avaliação da Solução Proposta com os Dados do Eclipse

O procedimento de avaliação da solução proposta neste trabalho consistiu na execução do algoritmo genético descrito no Capítulo 4 considerando os cenários de dados apresentados na Seção 5.3 e a configuração padrão do algoritmo genético descrita na Tabela 5.6. Esta configuração foi inspirada nas configurações que implicaram nos melhores resultados nos trabalhos de Gonçalves et al. (2008) e Xu & Bean (2007).

Tabela 5.6. Configuração Padrão Utilizada no Procedimento de Avaliação

Tamanho da Polução	50 cromossomos
Critério de Parada (G)	200 gerações
Percentil de Seleção dos Melhores Indivíduos (T%)	20%
Percentil de Geração de Novos Indivíduos (I%)	79%
Percentil de Recombinação (C%)	70%

Em geral, os procedimentos de avaliação de novos métodos de otimização propostos tendem a compará-los com outros métodos previamente aplicados ao mesmo problema, considerando um conjunto de instâncias de referência para as quais são conhecidos as melhores soluções e os respectivos tempos de execução. O objetivo desta avaliação é verificar se o novo método é capaz de aprimorar a melhor solução conhecida ou reduzir o tempo necessário para se alcançar tal solução ao longo da busca.

Neste trabalho foi abordado um novo problema de otimização sem nenhum referencial histórico de outros métodos para comparação, dificultando o processo de avaliação do método proposto. Sabe-se somente que o problema em questão é recorrente na maioria das organizações e métodos manuais são mais frequentemente utilizados. Assim, optou-se por comparar cronogramas de execução das tarefas de correção bugs gerados com as seguintes linhas de base: (i) agendamento efetivamente realizado pela equipe de desenvolvimento do Eclipse (procedimento manual); (ii) o agendamento gerado a partir de uma especialização do método de busca local inspirado no trabalho de Antoniol et al (2005) que abordou um problema semelhante; (iii) e o agendamento sugerido pelo método proposto nesta dissertação.

A técnica de busca local foi baseada na seguinte estratégia:

- Utilizando o esquema de codificação de soluções definido para o algoritmo genético apresentado no Capítulo 4, uma solução inicial é gerada aleatoriamente e representada sob a forma de um vetor V com N_b posições, onde cada elemento $v \in V$ contém um número real dentro do intervalo [0,1];
- Calcula-se o custo da solução corrente;
- O conjunto vizinhança da solução corrente é definido como sendo o conjunto de soluções geradas a partir da troca dos valores armazenados em duas posições quaisquer i e j do vetor V, onde i < j;
- A busca inicia-se atribuindo o índice i=1 e variando o índice j de 2 até N_b ou até que o primeiro melhor vizinho seja encontrado (custo da solução vizinha menor do que o custo da solução corrente);
- Se alguma melhoria for alcançada, então a solução corrente assume a configuração do primeiro melhor vizinho e um novo procedimento de busca é iniciado a partir do conjunto vizinhança da nova solução corrente;
- Caso nenhuma nova melhoria seja atingida, o procedimento de busca é encerrado e a solução final recebe as configurações da solução corrente.

A Tabela 5.7 apresenta os resultados obtidos durantes os testes, considerando os cenários descritos na Seção 5.2. Os indicadores de desempenho utilizados na análise comparativa entre os métodos foram: (i) a medida de custo da função objetivo definida na Equação 4.3; (ii) o tempo de execução do método; (iii) tempo médio de correção das tarefas em cada esquema de agendamento; e (iv) o número de correções concluídas em até um mês de trabalho.

Tabela 5.7. Resultados Obtidos nos Testes para a Configuração Padrão

Esquema de Agendamento	Custo	Tempo de Execução do Método	Tempo Médio de Correção (dias)	Número de Bugs Corrigidos Após 1 Mês						(%) das Tarefas
				P1	P2	P3	P4	P5	Total	
Cenário 1										
Agendamento Realizado	24.781	NA	60,9	-	1	45	-	-	46	34,30%
Busca Local	11.654	47 seg	28,8	-	4	86	1	-	91	67,90%
Algoritmo Genético	6.935	27 seg	17,17	-	5	105	1	-	111	82,80%
Cenário 2										
Agendamento Realizado	403.805	NA	83,51	5	12	267	4	2	290	19,00%
Busca Local	204.316	24 min	44,29	37	77	784	8	8	914	59,80%
Algoritmo Genético	143.329	45 min	30,87	38	84	905	7	10	1.044	68,30%

De posse dos resultados obtidos após os testes nos cenários, observa-se que a qualidade da solução gerada pelo algoritmo genético proposto foi significativamente superior aos esquemas de agendamento efetivamente realizados e aos esquemas propostos pelo método de busca local em ambos os cenários. As soluções propostas pelo algoritmo genético indicam uma redução de 70% no tempo médio de término das tarefas de correção (maximizando a taxa de ocupação da equipe de testes e liberando o mais breve possível a equipe de desenvolvimento para atuar em outras frentes de trabalho), um aumento no número de correções que podem ser incorporadas na próxima versão mensal do software e a priorização na resolução dos bugs mais críticos.

Com relação ao tempo de execução dos métodos, para o Cenário 1, que possui um espaço de busca menor, o menor tempo de resposta foi obtido com o algoritmo genético, enquanto que para o Cenário 2, que mapeia um espaço de busca maior, a busca local implicou em um tempo de processamento 46% menor. O método de busca local norteia a busca para uma solução que é um ótimo local, mas que pode estar distante do ótimo global. Isto pode ser constatado pela diferença na qualidade da solução gerada entre o algoritmo genético e a busca local. Em virtude da presença dos indivíduos mutantes, o algoritmo genético consegue mapear um espaço de busca de maior diversidade, fugindo do ótimo local.

5.5. Análise de Sensibilidade

O estudo que será descrito nesta Seção tem por finalidade analisar o grau de influência de cada um dos parâmetros do algoritmo genético definidos na Tabela 5.6. Esta análise consiste em selecionar cada um dos parâmetros, variar o valor do mesmo

mantendo os valores dos demais fixos e observar a influência na qualidade da solução gerada e no tempo de resposta.

Um dos pontos que devem ser considerados durante a análise de sensibilidade consiste na avaliação da relação custo x benefício das configurações. Por custo, entende-se como sendo o custo computacional, isto é, o tempo de processamento do método. Quanto maior o tempo de processamento, maior o custo. A avaliação do benefício está relacionada à medida de qualidade da solução que, no contexto deste trabalho, corresponde ao valor calculado pela função objetivo Custo apresentada na Equação 4.3. Quanto menor o valor da função objetivo, maior será o benefício ofertado pela solução proposta pelo método.

Para avaliar a relação custo x benefício entre as mudanças nas configurações do algoritmo genético, foi definido o indicador de desempenho ID, calculado pela Equação 5.1. $Custo_i$ e $Tempo_i$ são respectivamente o custo (medida da função objetivo) e o tempo de execução do algoritmo na configuração i, enquanto que $Custo_0$ e $Tempo_0$ correspondem ao custo e ao tempo de execução do algoritmo na configuração padrão apresentada na Tabela 5.6. Um ID_i maior do que 1 significa dizer que a configuração i apresenta resultados melhores do que a configuração padrão.

$$ID_i = \frac{\left(Custo_0 / Custo_i \right)}{\left(Tempo_i / Tempo_0 \right)} \tag{5.1}$$

Cada uma das próximas sub-seções apresentará um quadro comparativo entre diversas configurações do algoritmo genético geradas a partir da variação de um determnado parâmetro. Nas tabelas a seguir, o item grifado em amarelo reproduz os resultados obtidos pela configuração padrão utilizada nos testes da Seção 5.4.

5.5.1. Tamanho da População

O parâmetro “Tamanho da População” variou na seguinte escala: 50 (padrão), 100, 150 e 200 indivíduos. Os resultados obtidos para cada um dos cenários estão apresentados na Tabela 5.8. Considerando o aumento no tamanho da população, foi observada uma melhoria na qualidade da solução proposta acompanhado de um aumento no tempo de execução do algoritmo. Contudo, o indicador de desempenho ID demonstra que nenhuma das configurações alternativas apresentou uma relação custo x benefício superior à configuração original.

Tabela 5.8. Resultados Obtidos na Análise do Tamanho da População

Configuração	Custo	Tempo de Execução do Método	Tempo Médio de Correção (dias)	(%) das Tarefas Concluídas Após 1 Mês	ID
Cenário 1					
Pop. 50 indivíduos	6.935	27 seg	17,17	82,84%	1
Pop. 100 indivíduos	6.797	54 seg	16,83	81,30%	0,47
Pop. 150 indivíduos	6.685	129 seg	16,56	83,60%	0,2
Pop. 200 indivíduos	6.627	186 seg	16,42	83,60%	0,14
Cenário 2					
Pop. 50 indivíduos	143.329	0:45 hr	30,87	68,30%	1
Pop. 100 indivíduos	132.450	1:16 hr	28,49	69,50%	0,64
Pop. 150 indivíduos	117.504	2:50 hr	21,49	71,10%	0,32
Pop. 200 indivíduos	112.439	3:51 hr	18,41	72,80%	0,25

Observa-se que no Cenário 2, que contempla um espaço de busca maior, os indicadores de desempenho implicaram em quedas menos acentuadas quando comparados com os indicadores obtidos no Cenário 1. Além disso, a melhoria no custo foi mais acentuada no Cenário 2. Estes resultados indicam que quanto maior for o espaço de busca do problema, mais sensível será a qualidade da solução gerada pelo método ao aumento do tamanho da população.

5.5.2. Critério de Parada

O parâmetro “Critério de Parada” variou na seguinte escala: 100, 200 (padrão), 300 e 400 iterações. Os resultados obtidos para cada um dos cenários estão apresentados na Tabela 5.9. Considerando o aumento no tamanho da população, foi observada uma melhoria na qualidade da solução proposta acompanhado de um aumento no tempo de execução do algoritmo. A configuração de 100 iterações apresentou a melhor relação custo x benefício (1,91).

Tabela 5.9. Resultados Obtidos na Análise do Critério de Parada

Configuração	Custo	Tempo de Execução do Método	Tempo Médio de Correção (dias)	(%) das Tarefas Concluídas Após 1 Mês	ID
Cenário 1					
100 iterações	6.984	14 seg	17,29	82,84%	1,91
200 iterações	6.935	27 seg	17,17	82,84%	1
300 iterações	6.886	41 seg	17,06	83,58%	0,66
400 iterações	6.847	55 seg	16,96	83,58%	0,5
Cenário 2					
100 iterações	147.917	00:23 hr	31,86	67,56%	1,89
200 iterações	143.329	00:45 hr	30,87	68,30%	1
300 iterações	141.732	01:10 hr	30,53	68,67%	0,65
400 iterações	139.415	01:33 hr	30,02	69,06%	0,5

Em comparação com a análise de sensibilidade do tamanho da população, nota-se que a melhoria nos resultados é menos acentuada quando o número de iterações do algoritmo é estendido mantendo-se fixas as demais configurações. É possível imaginar que neste caso os resultados tendem a caminhar para algum mínimo local, tendo em vista que a amplitude dos custos obtidos é baixa e a melhoria tende a ficar menos sensível a cada novo ciclo de 100 iterações.

Por outro lado, cabe ressaltar que o algoritmo genético rapidamente atingiu bons resultados, conforme observado na configuração de 100 iterações, quando comparado com a técnica de agendamento realizada e o método de busca local testados na Seção 5.4. No contexto da Engenharia de Software, o desafio nem sempre consiste em buscar a configuração ótima, mas aquela capaz de aumentar a eficiência do processo de resolução de bugs, isto é, encontrar rapidamente uma configuração capaz de aprimorar significativamente os resultados dos esquemas informais de agendamento das tarefas de correção de bugs.

5.5.3. Geração da População

Este procedimento consiste em variar os parâmetros para geração da população futura a cada nova iteração: Percentil de Seleção dos Melhores Indivíduos (T%), Percentil de Geração de Novos Indivíduos (I%) e Percentil de Mutação (M%). A Tabela 5.10 descreve as configurações analisadas.

Tabela 5.10. Configurações da Análise dos Parâmetros da Geração da População

Configuração	T(%)	I(%)	M (%)
Configuração 1	10%	89%	1%
Configuração 2	20%	79%	1%
Configuração 3	30%	69%	1%
Configuração 4	20%	80%	0%
Configuração 5	20%	75%	5%

A Tabela 5.11 apresenta os resultados desta análise. No cenário 1, as configurações 1, 2 e 3 apresentaram relações custo x benefício semelhantes, enquanto que no cenário 2 a Configuração 5 (que apresentou o pior desempenho no cenário 1) apresentou a solução com melhor relação custo x benefício. Em ambos os cenários, a Configuração Padrão sugerida por Gonçalves et al. (2008) e Xu & Bean (2007) esteve entre as configurações de melhor desempenho.

Tabela 5.11. Resultados Obtidos na Análise da Geração da População

Configuração	Custo	Tempo de Execução do Método	Tempo Médio de Correção (dias)	(%) das Tarefas Concluídas Após 1 Mês	ID
Cenário 1					
Configuração 1	6.817	27 seg	16,9	82,08%	1,01
Configuração 2	6.935	27 seg	17,17	82,84%	1
Configuração 3	6.908	27 seg	17,09	81,34%	1
Configuração 4	7.332	26 seg	18,14	79,85%	0,98
Configuração 5	6.976	36 seg	17,25	83,58%	0,74
Cenário 2					
Configuração 1	151.916	00:46 hr	32,81	67,04%	0,92
Configuração 2	143.329	00:45 hr	30,87	68,30%	1
Configuração 3	150.112	00:53 hr	32,32	66,84%	0,81
Configuração 4	160.467	01:19 hr	34,52	65,47%	0,73
Configuração 5	139.553	00:41 hr	30,1	68,08%	1,12

Um ponto importante a ser analisado é a influência da mutação na qualidade da solução gerada. De acordo com Bean (1994), a finalidade dos indivíduos mutantes é permitir a diversidade da população ao longo das sucessivas gerações, evitando a convergência do algoritmo para um ótimo local. Com base nos resultados, observa-se que em ambos os cenários a solução de pior qualidade foi obtida pela Configuração 4 onde há ausência de indivíduos mutantes. Isto remete à hipótese de uma convergência precoce para um ótimo local nesta configuração, que é reforçada pelos gráficos apresentados nas figuras 5.1 e 5.2. Observe nestes gráficos que a curva da Configuração 4 atinge rapidamente um valor mínimo, mantendo-se constante a partir deste momento nas próximas iterações.

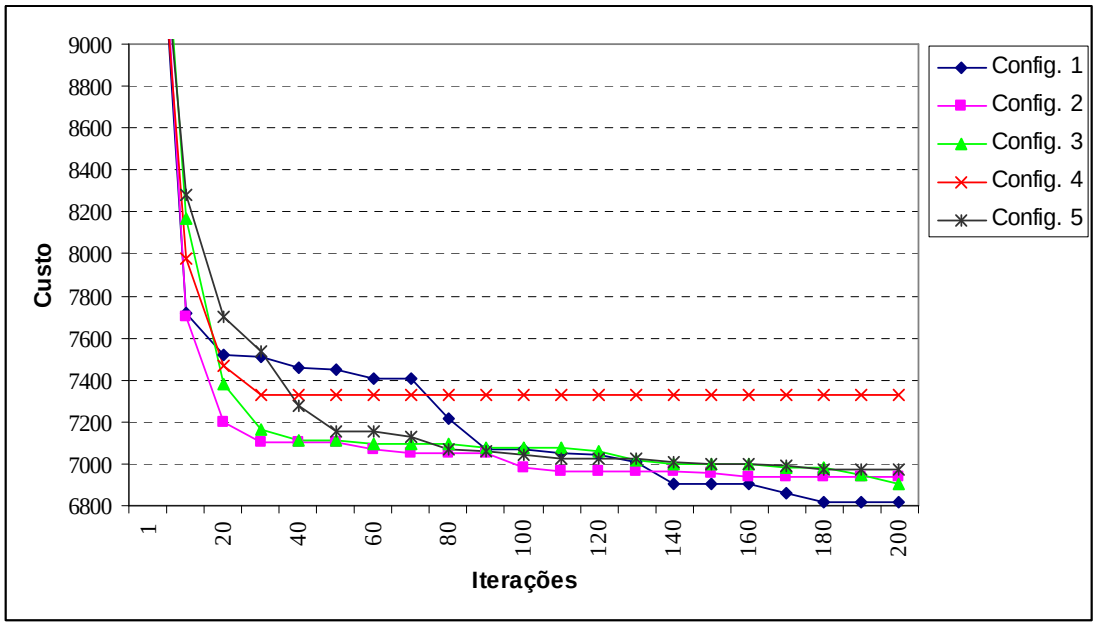

Figura 5.1. Evolução dos Resultados ao Longo das Iterações no Cenário 1

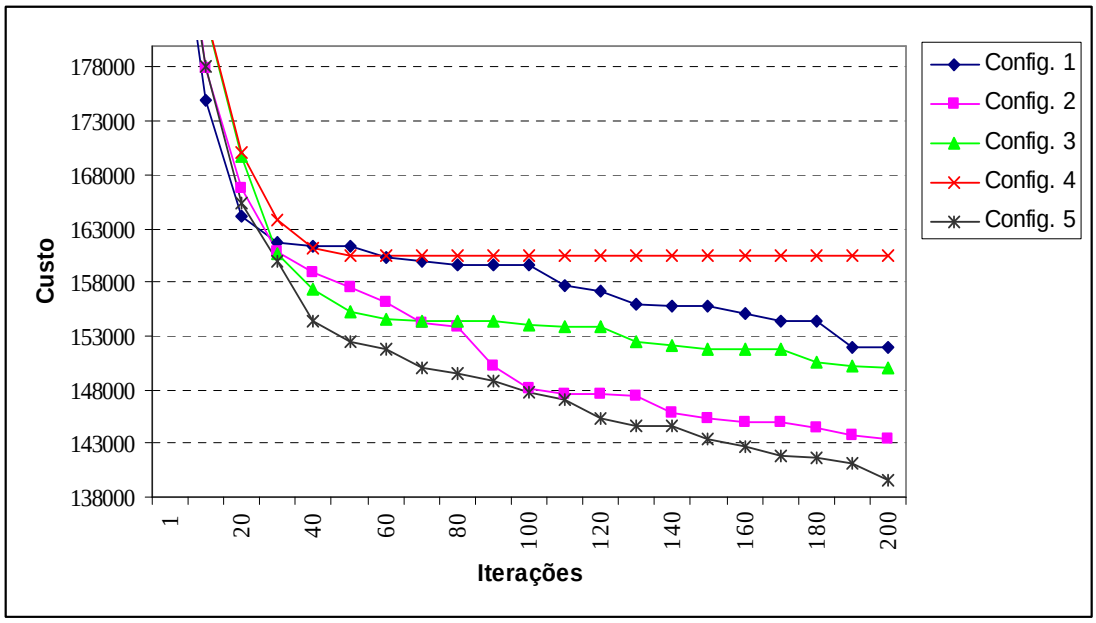

Figura 5.2. Evolução dos Resultados ao Longo das Iterações no Cenário 2

5.5.4. Percentil de Recombinação Genética

O parâmetro Percentil de Recombinação Genética (*crossover*) variou na seguinte escala: 60%, 70% (padrão), 80% e 90%. Os resultados obtidos para cada um dos cenários de dados estão apresentados na Tabela 5.12.

Tabela 5.12. Resultados Obtidos na Análise da Recombinação Genética

Configuração	Custo	Tempo de Execução do Método	Tempo Médio de Correção (dias)	(%) das Tarefas Concluídas Após 1 Mês	ID
Cenário 1					
% Crossover = 60%	7.208	27 seg	17,83	80,60%	0,96
% Crossover = 70%	6.935	27 seg	17,17	82,84%	1
% Crossover = 80%	7.286	27 seg	18,04	80,60%	0,95
% Crossover = 90%	7.354	27 seg	18,19	79,10%	0,94
Cenário 2					
% Crossover = 60%	141.261	00:45 hr	30,42	68,08%	1,01
% Crossover = 70%	143.329	00:45 hr	30,87	68,30%	1
% Crossover = 80%	151.837	00:45 hr	32,66	67,30%	0,94
% Crossover = 90%	157.109	00:45 hr	33,91	66,45%	0,91

Conforme observado na Tabela 5.12, os tempos de execução do algoritmo foram idênticos em todas as configurações. Assim, observa-se que o parâmetro do Percentil de Recombinação Genética não influi no tempo de resposta do algoritmo. Outro item a ser destacado é o desempenho da Configuração Padrão. No Cenário 1, a

Configuração Padrão foi superior às demais, enquanto que no Cenário 2 ficou como o segundo melhor resultado.

5.6. Conclusão

Ao longo deste capítulo foram descritas análises realizadas para avaliar a eficácia da solução proposta, considerando uma amostra dos relatórios de bugs registrados para os softwares do projeto Eclipse. Foram criados dois cenários de teste de tamanhos distintos, que foram submetidos ao método de otimização proposto.

Os esquemas de agendamento sugeridos pelo método proposto apresentaram resultados expressivamente superiores quando comparados com os esquemas de agendamento efetivamente realizados e o esquema sugerido pela técnica de busca local, em ambos os cenários. Os três objetivos do esquema de agendamento definidos no Capítulo 4 (reduzir o tempo médio de correção dos bugs, maximizar o número de bugs corrigidos e incorporados na próxima versão do software e priorizar a correção dos bugs mais críticos) foram alcançados pelos esquemas de agendamento sugeridos pelo método proposto.

Foram realizadas análises de sensibilidade nos parâmetros do algoritmo genético, que consistiram em analisar isoladamente cada um dos parâmetros com a finalidade de avaliar a influência deste na qualidade da solução sugerida e no tempo de resposta do algoritmo. Cabe ressaltar que, em todas as análises de sensibilidade conduzidas, a configuração padrão proposta por Gonçalves et al. (2008) e Xu & Bean (2007) esteve entre as configurações que apresentaram as melhores relações custo x benefício.

Na análise de sensibilidade do tamanho da população, observou-se que um aumento no número de indivíduos melhora a qualidade da solução gerada, mas também aumenta exponencialmente o tempo de resposta do algoritmo. A influência na qualidade das soluções geradas foi mais sensível no Cenário 2, que possui um espaço de busca maior.

Na análise de sensibilidade do critério de parada, observou-se que um aumento no número de iterações do algoritmo genético também melhora a qualidade da solução gerada, mas aumenta linearmente o tempo de resposta do algoritmo. Notou-se que em ambos os cenários, boas soluções foram rapidamente encontradas utilizando a configuração com apenas 100 iterações.

Considerando a análise de sensibilidade da geração da população, o percentual de indivíduos mutantes na nova população foi o parâmetro que mais influenciou a

qualidade da solução. Constatou-se que a ausência de indivíduos mutantes pode implicar em uma convergência precoce para um ótimo local durante o processamento do algoritmo genético.

Na análise do percentil de recombinação genética (*crossover*), foi constatado que este parâmetro não exerce influência no tempo de resposta do algoritmo. Os resultados também indicam que variações neste parâmetro próximas ao valor estabelecido pela configuração padrão tem maiores probabilidades de aprimorar a qualidade da solução gerada.

CAPÍTULO 6

CONCLUSÕES

6.1. Considerações Finais

A definição do agendamento das tarefas de correção de bugs consiste na execução das seguintes atividades: (i) identificar a lista de tarefas de correção de bugs pendentes; (ii) analisar a disponibilidade e a capacitação dos desenvolvedores da organização; (iii) definir qual desenvolvedor será alocado para realizar cada uma das tarefas de correção de bugs; e (iv) sequenciar a execução das tarefas, levando em consideração a prioridade e o esforço estimado. O esquema de agendamento resultante deste processo desempenha um papel fundamental no ciclo de vida do software, visto que definirá quais correções de bugs serão incorporadas em suas versões subsequentes.

Neste contexto, é importante prover um esquema de agendamento capaz de maximizar a entrega de correções de bugs no próximo *release* (onde o maior interessado é o fornecedor do software), aprimorando a qualidade da nova versão do software, e antecipar a execução das correções dos bugs mais críticos sob o ponto de vista do cliente. Ocorre que em algumas situações estes objetivos podem ser conflitantes. Para ilustrar esta situação, imagine um cenário contemplando um software com *release* mensal e um conjunto de 21 bugs pendentes, onde 20 bugs são de baixa prioridade, com esforço estimado para correção de 1 homem-dia, e 1 bug de alta prioridade, com esforço estimado para correção de 20 homem-dia.

Se as tarefas de correção dos bugs mais simples forem priorizadas, ao final do primeiro mês teríamos 95,24% do total das correções realizadas, atingindo o objetivo de maximizar o número de bugs corrigidos no próximo *release*, mas negligenciando o interesse do cliente de priorizar a correção do bug mais crítico. Por outro lado, caso o bug mais crítico fosse priorizado, os membros das equipes de qualidade e integração, responsáveis respectivamente pelos processos de testes das correções de bugs e incorporação das correções de bugs ao próximo *release*, teriam que esperar cerca de 20 dias para o cumprimento de suas tarefas, implicando em uma ociosidade

indesejada de membros da equipe. Assim, é possível perceber a importância de buscar um esquema de agendamento das tarefas de correção de bugs capaz de conciliar os interesses do fornecedor do software e do cliente.

O problema acima foi modelado sob a ótica de otimização combinatória e definido como o problema de distribuição e sequenciamento de correções de bugs (PDSCB). O modelo do problema apresentado neste trabalho foi inspirado em modelos de problemas de otimização clássicos, que foram adaptados para o contexto do processo de resolução de bugs. Foram identificadas semelhanças entre o PDSCB e outros problemas de otimização combinatória nas áreas de escalonamento de processos e de planejamento de cronogramas de projetos. Dada a complexidade deste problema, que tem características de um problema NP-Difícil, faz-se necessário um método heurístico para buscar uma solução aproximada.

Desta forma, foi desenvolvida uma técnica de otimização baseada no uso de um algoritmo genético combinado com uma heurística construtiva para buscar soluções aproximadas para o PDSCB. A escolha da metaheurística do algoritmo genético está sustentada na aderência desta técnica com os problemas de construção de cronogramas para projetos e de escalonamento de processos. A heurística construtiva foi utilizada em conjunto com o algoritmo genético para aprimorar o processo de seleção das soluções candidatas a cada iteração.

A proposta de solução apresentada neste trabalho também sugere uma interface de integração entre o BTS, ferramenta amplamente utilizada pelas organizações para gerir o processo de resolução de bugs, e o método heurístico. Foi descrito o procedimento que captura de informações relevantes do BTS, agrega-as às informações complementares fornecidas pelo gerente de projeto e transforma estas informações consolidadas em instâncias do PDSCB, que são submetidas à técnica de otimização a fim de buscar um esquema de agendamento próximo à solução ótima.

Foram conduzidos testes considerando uma amostra dos relatórios de bugs extraídos do Eclipse Bugzilla, para avaliar a eficácia da proposta de solução apresentada nesta dissertação. Também foram realizadas análises de sensibilidade dos parâmetros do algoritmo genético, cujos testes avaliaram diferentes parametrizações do algoritmo genético e os respectivos impactos na qualidade e no tempo de resposta do método.

Os resultados dos testes indicaram que os esquemas de agendamento sugeridos pelo método proposto apresentaram resultados expressivamente superiores quando comparados com os esquemas de agendamento efetivamente realizados e os esquemas sugeridos pela técnica de busca local. Com relação as análises de sensibilidade nos parâmetros do algoritmo genético, em todas as análises conduzidas

a parametrização da configuração padrão esteve entre as configurações que apresentaram as melhores relações custo x benefício.

De posse dos resultados observados, cabe ressaltar a potencialidade dos benefícios ofertados pela solução proposta. A técnica de otimização é capaz de sugerir bons esquemas de agendamento das tarefas de correção de bugs em poucos minutos, considerando um grande espaço de busca. O planejamento da execução das tarefas de correção de bugs apoiado pelo método proposto provê ganhos substanciais para as organizações, como a redução dos custos de manutenção corretiva do software e o alinhamento das novas versões do software com as necessidades dos clientes.

Além dos benefícios citados acima, o modelo definido para o PDSCB é compatível com a modelagem dos dados manipulados pelo BTS, viabilizando a integração entre o método proposto e o BTS. Assim, é possível imaginar a incorporação deste ferramental no BTS, constituindo um instrumento diferenciado para apoiar as atividades de planejamento dos gerentes de projeto.

6.2. Contribuições

Podemos destacar como principais contribuições deste trabalho:

- Identificação de um novo problema de otimização no contexto da Engenharia de Software Baseada em Busca (*Search Based Software Engineering - SBSE*);
- Definição e formalização do problema de distribuição e sequenciamento de tarefas de correção de bug;
- Definição do procedimento de integração entre as variáveis de decisão do problema e as informações mantidas no BTS;
- Definição, formalização e implementação da técnica de otimização baseada no uso de um algoritmo genético combinado a uma heurística construtiva;
- Definição, planejamento, execução e análise dos resultados dos testes para avaliar a viabilidade da abordagem proposta;
- Definição, planejamento, execução e análise dos resultados da análise de sensibilidade dos parâmetros do algoritmo genético.

6.3. Limitações e Perspectivas Futuras do Trabalho

O modelo proposto neste trabalho é uma simplificação de uma realidade bastante complexa, inerente ao contexto do processo de resolução de bugs. Os itens a seguir descrevem algumas questões que podem comprometer a confiabilidade da abordagem proposta.

- A função objetivo consiste do cálculo do somatório ponderado dos tempos de término das tarefas. A definição dos pesos das prioridades é um processo arbitrário a ser realizado pelo gerente do projeto. A base de dados do BTS pode reunir informações de diferentes aplicações, direcionadas para vários clientes. A diversidade do contexto das aplicações (por exemplo, sistemas de informação, sistemas de tempo real e sistemas especialistas) e da relevância dos clientes que identificaram os bugs, devem ser levadas em consideração pelo gerente de projeto na definição dos pesos das prioridades das correções de bugs;

- O modelo assume que sempre que um desenvolvedor inicia uma correção de bug, o desenvolvedor executa a tarefa até o seu término, sem interrupções. Contudo, nem sempre a sequência de tarefas é seguida rigorosamente pelos desenvolvedores. Às vezes, problemas podem ocorrer, impedindo a continuidade das tarefas (por exemplo, a execução da tarefa depende de um recurso que pode não estar disponível no momento), e o desenvolvedor pode adiar a atividade corrente e seguir para uma próxima;

- O cálculo do tempo de término de cada correção de bug depende do sequenciamento das correções e do esforço estimado para cada correção. Embora o cálculo da estimativa de esforço das correções de bug esteja fora do escopo deste trabalho, é importante ressaltar que variações altas nas estimativas informadas nos relatórios de bug podem comprometer a qualidade dos esquemas de distribuição e sequenciamento das tarefas;

- Segundo o modelo do problema, cada tarefa de correção de bugs é classificada de acordo com um tipo do bug e somente desenvolvedores capazes de corrigir bugs desta classe podem assumir as tarefas de correção. Uma vez que a avaliação da capacidade dos desenvolvedores consiste da análise do histórico de correções de bugs realizadas, que está armazenado na base de dados BTS, novos desenvolvedores seriam ignorados pelos

procedimentos de seleção, pois estes não possuem dados históricos associados. Neste caso, faz-se necessária a intervenção manual do gerente de projeto, permitindo a atribuição de correções de bugs aos novos desenvolvedores ou ajustes na matriz SK para inclusão dos novos desenvolvedores.

Apesar das limitações citadas, a solução proposta constitui-se de um ferramental com potenciais benefícios. O objetivo deste trabalho não é definir um modelo que contemple todas as características de uma realidade bastante complexa, mas sim abordar os principais aspectos de um problema relevante e recorrente na maioria das organizações e prover um ferramental capaz de sugerir bons esquemas de agendamento das tarefas de correção de bugs em um período de tempo relativamente curto, agregando benefícios substanciais aos gerentes de projetos de software.

Outras possíveis evoluções do trabalho podem ser citadas:

- A proposta de solução foi avaliada considerando amostras de relatórios de bugs extraídos do projeto de software livre Eclipse, visto que o repositório de dados da comunidade do Eclipse é capaz de prover uma quantidade razoável de dados públicos. Contudo, os desenvolvedores da comunidade Eclipse trabalham como voluntários, implicando em algumas características específicas deste contexto, tais como tarefas com longa duração e um conjunto grande de desenvolvedores disponíveis. A realidade da maioria das organizações não reflete estas características. Assim, é importante conduzir a avaliação da solução proposta considerando dados obtidos a partir de projetos da indústria de software;

- O conjunto das classes de bugs corresponde ao conjunto definido por todas as associações possíveis entre os atributos PRODUCT e COMPONENT dos relatórios de bug. Uma alternativa mais elegante consistiria na aplicação de técnicas de mineração de textos (text mining) na base de dados do BTS, identificando as classes de bugs a partir de técnicas de agrupamento (clustering);

- As versões mais recentes do BTS permitem a definição de regras de precedência entre as tarefas de correção de bugs. Nestes sistemas, é possível criar um conjunto R de restrições de precedência onde cada elemento $r_{ij} \in R$ define que a tarefa de correção de bug $b_j \in B$ deverá ser

iniciada somente após a conclusão da tarefa $b_i \in B$. Estas regras poderiam ser acomodadas em uma nova versão do modelo do problema.

Referências Bibliográficas

Alba, Henrique; Chicano, Francisco, J., “Software project management with GAs”, Information Sciences: an International Journal, Volume 177, Issue 11, 2007.

Antoniol, Giulio; Di Penta, Massimiliano; Harman, Mark, “Search-based Techniques Applied to Optimization of Project Planning for a Massive Maintenance Project”, Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 240 – 249, 2005.

Anvik, John; Hiew, Lyndon; Murphy, C., Gail, “Coping with an open bug repository”, OOPSLA workshop on Eclipse technology eXchange, San Diego, California, pp. 35 – 39, 2005.

Anvik, John; Hiew, Lyndon; Murphy, C., Gail, “Who Should Fix This Bug?”, Proceedings of the 28th international conference on Software engineering, Shanghai, China, pp. 361-370, 2006.

Anvik, John; Murphy, C., Gail, “Determining Implementation Expertise from Bug Reports”, 4th International Workshop on Mining Software Repositories, Minneapolis, Minnesota, May 2007, pp. 9-16.

Bean, C., James. “Genetic algorithms and random keys for sequencing and optimization”, ORSA Journal of Computing, vol. 6, pp. 154-160, 1994.

Beck, Kent, “Extreme Programming Explained”, 1 ed., Addison Wesley, 1999.

Bettenburg, Nicolas; Just , Sascha; Schröter, Adrian; Weiss, Cathrin; Premraj, Rahul; Zimmermann, Thomas; “What makes a good bug report?”, 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Atlanta, Georgia, Pages: 308-318, 2008.

Boehm, B.; Basili, V., “Software Defect Reduction Top 10 List,” IEEE Computer, vol. 34(1): 135-137, January 2001.

Bruno, J.; Coffman Jr., E.G.; Sethi, R., “Scheduling Independent Tasks to Reduce Mean Finishing Time”, Communications of the ACM, 17:382-387, 1974.

Clarke, J.; Dolado, J.; Harman, M.; Hierons, R.; Jones, B; Lumkin, M.; Mitchell, B.; Mancoridis, S.; Rees, K.; Roper, M.; Shepperd, M., “Reformulating software engineering as a search problem”, IEEE Proc. Software 150 (3) 161–175, 2003.

Colorni, Alberto; Dorigo, Marco; Maniezzo, Vittorio, “Distributed Optimization by Ant Colonies”, Europen Conference on Artificial Life, Paris, France, Elsevier Publishing, 134–142, 1991.

D'Ambros, M.; Lanza, M.; Pinzger, M., "A Bug's Life" Visualizing a Bug Database", 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, pp. 113-120, 2007.

Dreo, Johann ; Petrowski, Alain; Siarry, Patrick; Taillard, Eric, "Metaheuristics for Hard Optimization: Methods and Case Studies", 1 ed., Springer, 2005.

Elmasri, Ramez; Navathe, Shamkant, "Sistemas de Banco de Dados", 4 ed., Addison Wesley, 2005.

Fowler, Martin, "Refactoring: Improving the Design of Existing Code", 1 ed., Addison Wesley, 1999.

M., R., Garey; D., S., Johnson. "Computers and Intractability: A Guide to the Theory of NP-completeness", Freeman, San Francisco, 1979.

Gonçalves, F., J.; Mendes, M., J., J; Resende, C., G., M. "A Genetic Algorithm for the Resource Constrained Multi-Project Scheduling Problem", European Journal of Operational Research, 1171-1190, 2008.

Harman, Mark; Jones, F., Bryan. "Search based software engineering", Information and Software Technology, 43(14):833–839, 2001.

Harman, Mark; Tratt, Laurence, "Pareto optimal search based refactoring at the design level", 9th annual conference on Genetic and evolutionary computation, 2007.

Harman, Mark; Hierons, M., Robert; Proctor, Mark, "A New Representation And Crossover Operator For Search-based Optimization Of Software Modularization", Genetic and Evolutionary Computation Conference, pp. 1351 – 1358, 2002.

Harman, Mark ; Mansouri , S., Afshin; Zhang, Yuanyuan, "Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications", Disponível em http://www.dcs.kcl.ac.uk/technical-reports/papers/TR-09-03.pdf, 2009.

Harman, Mark. "The Current State and Future of Search Based Software Engineering", Proceedings of International Conference on Software Engineering / Future of Software Engineering 2007 (ICSE/FOSE '07), pp. 342–357, Minneapolis, Minnesota, USA. IEEE Computer Society, 2007.

Hart, Emma; Ross, Peter; Corn, David. "Evolutionary Scheduling: A Review", Genetic Programming and Evolvable Machines, Volume 6, Number 2 / June, 2005.

Holland, H., J. "Adaptation in Natural and Artificial Systems", University of Michigan Press, 1975.

Hooimeijer, Pieter; Weimer, Westley, "Modeling bug report quality", Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, Atlanta, Georgia, USA, pp. 34-43, 2007

Hoos, Holger, Stüzle, Thomas, “Stochastic Local Search - Foundations and Applications”, 2 ed., San Francisco, Elsevier, 2005.

Jones, F. , B.; Eyres, E., D.; Sthamer, H., “A strategy for using genetic algorithms to automate branch and fault-based testing”, Computer Journal, Vol. 41(2), pp. 98-107, 1998.

Just, Sascha; Premraj, Rahul; Zimmermann, Thomas, “Towards the next generation of bug tracking systems”, IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 82-85, 2008.

Karner, Cem, "Liability for bad software and support." Software Support Professionals Association Executive Briefing, San Diego, CA, October 3, 1996.

Khan, Usman; Bate, Iain, “WCET Analysis of Modern Processors using Multi-Criteria Optimisation”, Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE '09), Cumberland Lodge Windsor UK, 13-15 May, 2009.

Kirkpatrick, S. ; Gelatt , D. , C. ; Vecchi, P., M., “Optimization by Simulated Annealing”, Science, pp. 865 – 892, 1983.

Kolisch, Rainer; Padman, Rema. “An Integrated Survey of Project Scheduling”, OMEGA International Journal of Management Science, pp. 2141-2153 , 1997.

Kotonya, Gerald; Sommerville, Ian, “Requirements Engineering : Processes and Techniques”, 1 ed., John Wiley, 1998.

Netto, Fernando; Barros, Marcio; Alvim, Adriana, “A Hybrid Heuristic Approach for Scheduling Bug Fix Tasks to Software”, Proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE '09), Cumberland Lodge Windsor UK, pp. 28-29, 13-15 May, 2009.

Panjer, D., Lucas, “Predicting Eclipse Bug Lifetimes”, Proceedings of the Fourth International Workshop on Mining Software Repositories, pp. 29, 2007.

Papadimitriou, H., Christos; Steiglitz, Kenneth, “Combinatorial Optimization: Algorithms and Complexity”, 1 ed., Prentice Hall, 1998.

Pressman, Roger S., “Engenharia de Software”, 6 ed., McGraw-Hill, 2006.

E. S. Raymond. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O’Reilly and Associates, Oct 1999.

Reeves, R., Colin, “Modern heuristic techniques for combinatorial problems”, 1 ed., John Wiley & Sons, Inc, 1993.

Glass, Robert, “Defining Quality Intuitively”, IEEE Software, May 1998, p. 103-104, 107, 1998.

Sagrado, José; Águila, Isabel, “Ant Colony Optimization for Requirement selection in Incremental Software development”, Proceedings of the 1st International

Symposium on Search Based Software Engineering (SSBSE '09), Cumberland Lodge Windsor UK, pp. 30, 13-15 May, 2009.

Serrano, Nicolas; Ciordia, Ismael, “Bugzilla, ITracker, and Other Bug Trackers”, IEEE Software, March/April 2005, vol. 22 issue. 2, 2005.

Shaw, Mary; Garlan, David, “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall, 1996.

Skutella, Martin, “Approximation and Randomizing in Scheduling”, Berlin, 1998.

Smith, E., W., “Various optimizers for single–stage production”, Naval Research and Logistics Quarterly 3, 59 – 66, 1956.

Tracey, Nigel ; Clark, A., John; Mander, Keith, “Automated Program Flaw Finding using Simulated Annealing”, Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98), pp. 73-81, Clearwater Beach Florida USA, 2-4 March, 1998.

Wang, Xiaoyin; Zhang, Lu; Xie, Tao; Anvik, John; Sun, Jiasu, “An approach to detecting duplicate bug reports using natural language and execution information”, International Conference on Software Engineering, Leipzig, Germany. pp. 461-470, 2008.

Weiss, Cathrin; Premraj, Rahul; Zimmermann, Thomas; Zeller, Andreas. “How Long will it Take to Fix This Bug?”, Proceedings of the Fourth International Workshop on Mining Software Repositories, pp. 1, 2007.

Xu, Shubin; Bean, C., James. “A Genetic Algorithm for Scheduling Parallel Non-identical Batch Processing Machines”, Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Scheduling, Honolulu, HI, pp. 143-150 , 2007