

Gleiph Ghiotto Lima de Menezes

OURIÇO: UMA ABORDAGEM PARA MANUTENÇÃO DA CONSISTÊNCIA EM
REPOSITÓRIOS DE GERÊNCIA DE CONFIGURAÇÃO

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Engenharia de Software.

Orientador: Prof. Dr. Leonardo Gresta Paulino Murta

Niterói

2011

Gleiph Ghiotto Lima De Menezes

OURIÇO: UMA ABORDAGEM PARA MANUTENÇÃO DA CONSISTÊNCIA EM
REPOSITÓRIOS DE GERÊNCIA DE CONFIGURAÇÃO

Dissertação apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Mestre. Área de Concentração: Engenharia de Software.

Aprovada em agosto de 2011.

BANCA EXAMINADORA

Prof. Dr. Leonardo Gresta Paulino Murta – Orientador
UFF

Prof. Dr. Alexandre Plastino de Carvalho
UFF

Prof. Dr. Marco Túlio de Oliveira Valente
UFMG

Niterói
2011

A Deus, aos meus pais e ao Leonardo Gresta Paulino Murta.

AGRADECIMENTOS

A Deus, que me manteve de pé e deu forças para enfrentar todas as adversidades.

Ao meu Orientador Leonardo Gresta Paulino Murta, que me incentivou e fez com que o Ouriço se tornasse uma realidade.

Aos meus amigos Heliomar Kann e Daniel Castellani, que juntos tornamos realidade o Oceano.

Aos meus colegas e ex-colegas de república Bruno Costa, Claudio Gusmão, Emanuel Antunes e Thiago Brañas, pelo apoio e por se tornarem minha família durante o período do mestrado.

A Alessandreia Oliveira e Heliomar Kann pelo apoio nas revisões da dissertação.

A todos os professores e funcionários do Instituto de Computação, que tive um contato direto ou indireto.

A meus pais, Duclerk Cardoso e Vilma Lúcia, que me apoiaram e entenderam a necessidade de estar ausente durante o tempo do mestrado.

A CAPES e a FAPERJ pelo apoio financeiro.

“O sucesso é um professor perverso. Ele seduz as pessoas inteligentes e as faz pensar que jamais vão cair.” Bill Gates

RESUMO

Gerência de Configuração é a disciplina tradicionalmente responsável por controlar a evolução de software. Entretanto, seu ciclo baseado em *check-out*, modificações e *check-in* pode se tornar contraproducente e propenso a erros, ainda que apoiada por abordagens manuais de verificação. Deste modo, este trabalho propõe uma abordagem que amplifica o ciclo tradicional de Gerência de Configuração através de tarefas assíncronas, automáticas e incrementais. Tarefas essas que verificam se os artefatos de software enviados para o repositório possuem conflitos físicos, sintáticos ou semânticos, antes que cheguem de fato no repositório. De acordo com os resultados obtidos em experimentos preliminares, é possível dizer que esta abordagem foi capaz de encontrar artefatos quebrados sem que o fluxo de trabalho do desenvolvedor seja significativamente afetado. Foi encontrada uma taxa de até 76,85% de *check-ins* inconsistentes, sendo que apenas 1,17% deles sofreram atraso devido às verificações adicionadas pela abordagem proposta.

Palavras-chave: Integração Contínua, Percepção, Gerência de Configuração, Repositório e Verificação.

ABSTRACT

Configuration Management is a discipline traditionally responsible for controlling the software evolution. However, its work cycle, based on check-out, changes, and check-in, is usually supported by manual verifications, which are, in most cases, counterproductive and error prone. Thus, this work proposes an approach that amplifies the traditional Configuration Management work cycle with asynchronous, automatic, and incremental tasks to verify software artifacts physically, syntactically, and semantically in order to find broken artifacts and, consequently, contribute for maintaining the consistency of Configuration Management repositories. Our preliminary results show that our approach was able to identify broken artifacts without a significant delay in the flow of check-ins. We found rates of up to 76.85% of inconsistent check-ins, with a delay of just 1.17% due to additional verifications introduced by the proposed approach.

Keywords: Continuous Integration, awareness, Configuration Management, Repository, and Verification.

LISTA DE ILUSTRAÇÕES

Figura 1. Cenário de quebra sintática onde o desenvolvedor renomeia um método (a) enquanto outro desenvolvedor faz referência a esse método(b).....	18
Figura 2. Cenário de quebra semântica onde um desenvolvedor realiza uma correção (a) e outro altera o comportamento de um método(b).....	19
Figura 3. Esquema de um SCV centralizado.....	26
Figura 4. Esquema de um SCV distribuído.....	26
Figura 5. Ciclo básico de Gerência de Configuração.....	27
Figura 6. Exemplo de conflito físico.....	28
Figura 7. Execução do <i>diff</i> no Git.....	30
Figura 8. Listando arquivos de um repositório CVS recursivamente.....	31
Figura 9. Exibindo <i>log</i> do Subversion.....	31
Figura 10. Compilação no Ant.....	36
Figura 11. Execução de testes no Maven.....	37
Figura 12. Execução da etapa de empacotamento no Maven.....	37
Figura 13. <i>Makefile</i> para compilação de dois arquivos.....	39
Figura 14. Estrutura de um <i>buildfile</i> para compilação de um projeto.....	40
Figura 15. Esquema do arquivo de configuração do Maven.....	41
Figura 16. Interface para exibição de uma base de código compartilhada.....	47
Figura 17. Interface de usuário do <i>Palantír</i>	49
Figura 18. Ilustração da IC Automática, adaptado de DUVALL et al. (2007).....	52
Figura 19. Painel para acompanhamento de projetos do Hudson.....	53
Figura 20. Ciclo tradicional de GC – <i>check-out</i> , <i>update</i> e <i>check-in</i>	59
Figura 21. Ciclo executado pela abordagem Ouriço.....	60
Figura 22. Atividades e comandos que os desenvolvedores executam durante o ciclo de desenvolvimento com GC.....	62
Figura 23. Comando de <i>check-out</i> do Ouriço.....	70
Figura 24. Comando de <i>check-in</i> do Ouriço.....	72
Figura 25. Aplicação do <i>update</i> tradicional na abordagem Ouriço.....	74
Figura 26. <i>Update</i> da abordagem Ouriço.....	75
Figura 27. Diagrama de classe da configuração inicial dos cenários.....	77
Figura 28. Espaço de trabalho obtido.....	78
Figura 29. Maria altera o nome de um método.....	78

Figura 30. João altera o nome de um método (a) e refatora a classe <i>LeiDoGasIdeal</i> (b).	79
Figura 31. Maria altera o nome de um método.	80
Figura 32. Configuração original.	81
Figura 33. João altera o nome de um método (a) e ajusta a classe <i>Testes</i> (b).	81
Figura 34. Maria adiciona uma classe.	82
Figura 35. Configuração de Maria.	82
Figura 36. Configuração obtida por João e Maria.	83
Figura 37. João altera a ação de um método.	83
Figura 38. Maria insere um artefato de software.	84
Figura 39. Diagrama do banco de dados do Ouriço e Oceano.	88
Figura 40. Página inicial do Oceano.	89
Figura 41. Tela para cadastro de um item de configuração.	90
Figura 42. Tela para cadastro de um projeto.	91
Figura 43. Tela para cadastro do um usuário Oceano.	92
Figura 44. Ligação de um usuário Oceano com um projeto.	93
Figura 45. Casos de uso da relação projeto-usuário.	93
Figura 46. Interface desktop para realização de <i>check-out</i>	95
Figura 47. Interface de linha de comando para realização de <i>check-out</i>	95
Figura 48. Interface de linha de comando nativa do Subversion.	96
Figura 49. Interface de <i>check-in</i> do TortoiseSVN.	97
Figura 50. Comando do <i>Maven</i> para compilar um projeto.	98
Figura 51. Comando do <i>Maven</i> para executar os testes de um projeto.	98
Figura 52. Junção para criação da configuração candidata.	99
Figura 53. Junção do <i>update</i>	100
Figura 54. Interface gráfica do módulo de <i>update</i> do Ouriço.	101
Figura 55. Interface de linha de comando do módulo de <i>update</i> do Ouriço.	101
Figura 56. Interface de configuração de um projeto.	102
Figura 57. Interface para configuração do servidor.	103
Figura 58. Painel de acompanhamento de <i>autobranches</i>	105
Figura 59. Interface para o detalhamento de uma <i>autobranch</i>	106
Figura 60. Interface para detalhes de um defeito.	106
Figura 61. Comando <i>svnsync initialize</i>	109
Figura 62. Comando <i>svnsync sync</i>	109

Figura 63. Ciclo para obtenção do repositório.....	111
Figura 64. Comando de <i>log</i> do Subversion.	111
Figura 65. Ciclo base para extração de dados do experimento.....	116
Figura 66. <i>Check-ins</i> atrasados.	117

LISTA DE TABELAS

Tabela 1. Gatilhos executados pelo Git (Chacon 2009).	32
Tabela 2. Gatilhos Subversion (Berlin and Rooney 2006).	33
Tabela 3. Caracterização dos projetos.	112
Tabela 4. Resultados dos experimentos levando em consideração o tempo.	118
Tabela 5. Resultados do experimento levando em consideração as configurações....	119
Tabela 6. Informações do tempo de execução das verificações.	120
Tabela 7. Informações sobre os <i>check-ins</i>	120

LISTA DE ABREVIATURAS E SIGLAS

GC – Gerência de Configuração

GEMS - Grupo de Evolução e Manutenção de Software

IC – Integração Contínua

JPA – *Java Persistence API*

JSF – *Java Server Faces*

SCM – Sistema de Controle de Modificações

SCV – Sistema de Controle de Versão

SGC – Sistema de Gerenciamento de Construção

SIC – Servidor de Integração Contínua

SUMÁRIO

Capítulo 1 – Introdução	17
1.1 Motivação	17
1.2 Objetivos	20
1.3 Organização	21
Capítulo 2 – Gerência de Configuração	23
2.1 Introdução	23
2.2 Sistemas de Controle de Versões	24
2.2.1 Interação entre repositório e espaço de trabalho	25
2.2.2 Gerenciando o repositório	29
2.2.3 Extração de dados do repositório	30
2.2.4 Estendendo SCV	31
2.2.5 Boas Práticas	33
2.3 Sistemas de Gerenciamento de Construção	35
2.3.1 Tarefas dos SGC	35
2.3.2 Automatizando tarefas com SGC	38
2.3.3 Exemplos de SGC	38
2.4 Considerações finais	41
Capítulo 3 – Manutenção da integridade de repositórios	43
3.1 Introdução	43
3.2 Abordagens baseadas em Percepção	44
3.2.1 FASTDash	46
3.2.2 Palantír	47
3.3 Integração Contínua	49
3.3.1 Integração Contínua manual	50
3.3.2 Integração Contínua automática	51
3.3.3 Práticas de Integração Contínua	53

3.4 Abordagens para manutenção da integridade de repositórios	55
3.4.1 Safe-Commit.....	56
3.4.2 Repoguard.....	56
3.5 Considerações Finais	57
Capítulo 4 – Ouriço	59
4.1 Introdução	59
4.2 Visão geral	60
4.3 Filtros	62
4.4 Políticas.....	64
4.4.1 Política permissiva.....	64
4.4.2 Política moderada	65
4.4.3 Política restritiva	65
4.4.4 Política dinâmica	66
4.5 Autobranches	67
4.6 Ciclo de trabalho do Ouriço.....	68
4.6.1 Check-out.....	69
4.6.2 Check-in.....	71
4.6.3 Update.....	74
4.7 Cenários de Utilização do Ouriço.....	76
4.7.1 Cenário ideal.....	77
4.7.2 Cenário com problema de primeiro nível	79
4.7.3 Cenário com problema sintático de segundo nível	80
4.7.4 Cenário com problema semântico de segundo nível	83
4.8 Considerações finais	84
Capítulo 5 – Implementação e utilização do Ouriço	86
5.1 Introdução	86
5.2 Visão geral	87

5.3 Oceano	88
5.3.1 Cadastro de item de configuração.....	90
5.3.2 Cadastro de projeto	91
5.3.3 Cadastro de usuários	91
5.3.4 Associação entre usuários e projetos	92
5.4 Ouriço	94
5.4.1 Check-out.....	94
5.4.2 Check-in.....	96
5.4.3 Update.....	100
5.4.4 Web.....	101
5.5 Considerações finais	106
Capítulo 6 – Avaliação Experimental do Ouriço.....	108
6.1 Introdução	108
6.2 Obtenção de repositórios	109
6.3 Projetos utilizados.....	112
6.4 Objetivos.....	114
6.5 Execução dos experimentos.....	115
6.6 Resultados.....	117
6.7 Análise de resultados	121
6.7.1 Eficácia do Ouriço na identificação de defeitos	121
6.7.2 A influência do Ouriço no ciclo de desenvolvimento do projeto	123
6.8 Considerações finais	125
Capítulo 7 – Conclusão.....	127
7.1 Contribuições.....	127
7.2 Limitações.....	127
7.2.1 Todos podem realizar <i>check-in</i>	127
7.2.2 Experimentos não levam em consideração o desenvolvedor	128

7.2.3 A identificação de quebras semânticas depende dos testes criados pelo desenvolvedor.....	129
7.3 Trabalhos futuros	129
7.3.1 Inserção de verificação estática	130
7.3.2 Integração do Ouriço com o Peixe-Espada.....	130
7.3.3 Realizar verificações paralelas em configurações distintas.....	131
7.3.4 Estudo de escalabilidade da abordagem	131

CAPÍTULO 1 – INTRODUÇÃO

1.1 MOTIVAÇÃO

De acordo com Bersoff (*apud* Pressman 2000) “não importa onde o desenvolvedor está no ciclo de desenvolvimento de um sistema, o sistema vai mudar, e o desejo de mudar vai permanecer durante todo o seu ciclo de vida”. Geralmente, o software evolui para atender às necessidades do usuário. Entretanto, se a evolução não é planejada e nem controlada, o software tende a degenerar após algum tempo, resultando em dificuldades para realização de futuras evoluções. Este cenário pode afetar a qualidade do produto e diminuir a produtividade da equipe de desenvolvimento.

Gerência de Configuração (GC) é uma disciplina tradicionalmente utilizada para auxiliar na evolução de software (Dart 1991). Esta disciplina faz uso de Sistemas de Controle de Versão (SCV) para gerenciar modificações em artefatos de software (e.g., código fonte, documentação, etc.), provendo um acesso controlado ao repositório. Além disso, os SCV permitem que os desenvolvedores obtenham versões dos artefatos de software referentes a diferentes momentos desenvolvimento (Duvall et al. 2007).

Durante a interação com um SCV existe controle de concorrência sobre as transações, que funciona como um mecanismo para controle de consistência. O acesso ao repositório pode ser realizado de duas formas distintas: (1) pessimista e (2) otimista. A primeira forma tem o trabalho baseado em *locks*, que serão descritos com mais detalhes no Capítulo 2. A segunda forma, otimista, não é baseada em *locks*, o que permite que diferentes desenvolvedores trabalhem sobre o mesmo conjunto de artefatos em paralelo. Nesse último caso o SCV fica responsável por realizar a combinação das transações de maneira atômica. No restante deste trabalho, é considerada a política otimista como a padrão.

O ciclo de trabalho dos SCV é basicamente composto por três passos, que são executados pelo desenvolvedor. O primeiro passo é o comando de *check-out*, que é responsável por obter artefatos de software de um repositório e enviá-los para um espaço de trabalho na máquina do desenvolvedor. O segundo passo é a realização de modificações, que podem ser adição, remoção ou edição dos artefatos de software previamente obtidos. Finalmente, o terceiro passo é a execução do comando de *check-in*, que é utilizado para enviar as modificações realizadas no espaço de trabalho do desenvolvedor para o repositório, tornando-as visíveis para todos os demais desenvolvedores. Alternativo ao ciclo padrão, o comando de *update* pode ser executado para atualizar o espaço de trabalho do desenvolvedor com modificações presentes no repositório. Durante o comando de *check-in*, o SCV é capaz

de identificar conflitos físicos (i.e., regiões dos artefatos editadas em paralelo por diferentes desenvolvedores). Entretanto, outros problemas não são verificados, permitindo que artefatos que não compilam (i.e., sintaticamente quebrados) ou que não passam nos testes (i.e., semanticamente quebrados) cheguem ao repositório. É válido ressaltar que os testes utilizados por essa abordagem devem ser automatizados, ou seja, testes que podem ser executados de maneira automática.

Quebras sintáticas e semânticas podem ocorrer quando o SCV não é utilizado com cuidado. Se um desenvolvedor compartilha artefatos quebrados, outros desenvolvedores podem realizar *check-out* e obter esses artefatos quebrados, devido à inconsistência do repositório. A quebra sintática (Figura 1) ocorre quando, por exemplo, um método inexistente é chamado, resultando em um erro de compilação. Esse cenário pode ocorrer quando dois desenvolvedores, por exemplo, João e Maria, realizam *check-out* de um repositório e realizam as seguintes ações: João altera o nome de um método, de *celsiusParaKelvin* para *transformacao*, e realiza *check-in* (Figura 1.a); Paralelamente, Maria adiciona uma nova classe que utiliza o método removido por João e, em seguida, realiza *check-in* (Figura 1.b). Após essas modificações terem sido enviadas para o repositório, se um desenvolvedor realizar *check-out* do repositório, obterá uma configuração que não compila, pois o repositório está inconsistente.

```

1 public class Transformacoes {
2
3     public static final int K = 273;
4
5     public static double celsiusParaKelvin(double tCelsius){
6     public static double transformacao(double tCelsius){
7         return tCelsius + K;
8     }
9 }

```

(a)

```

1 public class LeiDoGasIdeal {
2
3     public final double R = 8.314472;
4
5     public double volume(double p, double n, double tCelsius){
6         double tKelvin = Transformacoes.celsiusParaKelvin(tCelsius);
7         return n * R * tKelvin / p;
8     }
9 }

```

(b)

Figura 1. Cenário de quebra sintática onde o desenvolvedor renomeia um método (a) enquanto outro desenvolvedor faz referência a esse método(b).

Outro problema acontece quando os artefatos de software estão semanticamente quebrados. Esse cenário ocorre quando dois desenvolvedores, por exemplo, Arnaldo e Willian, realizam *check-out* de um repositório e executam as seguintes ações: Arnaldo altera o nome de um método, de *celciusParaKelvin* para *transformacao*, corrigindo o *check-in* de Maria, e realiza *check-in* dessa configuração (Figura 2.a); Paralelamente, Willian altera o comportamento do método *transformacao* (Figura 2.b), que passa a transformar a temperatura de Celsius para Fahrenheit, e realiza *check-in*, enviando sua configuração para o repositório. Após essas alterações, se algum desenvolvedor realizar *check-out* obterá uma configuração que não passa pelos testes, devido a uma quebra da regra de negócios, dado que o método *volume* espera a temperatura em *Kelvin* e não em *Fahrenheit*.

```

1 public class LeiDoGasIdeal {
2
3     public final double R = 8.314472;
4
5     public double volume(double p, double n, double tCelsius){
6         double tKelvin = Transformacoes.celciusParaKelvin(tCelsius);
7         double tKelvin = Transformacoes.transformacao(tCelsius);
8         return n * R * tKelvin / p;
9     }
10 }

```

(a)

```

1 public class Transformacoes {
2
3     private static final int K = 273; //Celsius
4
5     public static double transformacao(double tCelsius){
6         return tcelsius + K;
7         return 5 * (tCelsius - 32) / 9;
8     }
9 }

```

(b)

Figura 2. Cenário de quebra semântica onde um desenvolvedor realiza uma correção (a) e outro altera o comportamento de um método(b).

Algumas abordagens identificam esses tipos de quebras. Um exemplo é a Integração Contínua – IC (Duvall et al. 2007), que é uma prática onde membros de uma equipe de desenvolvimento integram o seu trabalho constantemente (i.e., no mínimo uma vez ao dia). Cada integração é verificada por uma construção do software (do inglês, *build*) que inclui compilação, execução de testes e, alternativamente, execução de verificações estáticas. Essa prática apresenta como objetivo principal identificar os problemas e reportar para os desenvolvedores o mais rápido possível.

Outro grupo de abordagens para identificação de problemas em repositórios de GC são as abordagens baseadas em percepção (do inglês, *awareness*). O conceito de percepção está fortemente ligado em manter um desenvolvedor ciente se as contribuições realizadas por ele são relevantes ou não para o restante da equipe de desenvolvimento e vice-versa (Dourish and Bellotti 1992). Portanto, estas abordagens realizam verificações no espaço de trabalho do desenvolvedor e quando algum indício de inconsistência é identificado, o desenvolvedor é alertado antes mesmo de realizar *check-in*.

Apesar de essas abordagens de integração contínua e percepção terem capacidade de identificar problemas em artefatos de software, a consistência do repositório não pode ser garantida. Com a utilização de IC o desenvolvedor é alertado que o repositório está inconsistente somente após os artefatos terem chegado ao repositório, o que a caracteriza como uma abordagem reativa. Por outro lado, as abordagens de percepção identificam artefatos inconsistentes de maneira proativa, isto é, durante o período em que o desenvolvedor está realizando suas modificações. Entretanto, não existe nenhum mecanismo para impedir que os artefatos quebrados cheguem ao repositório. Além disso, essas abordagens normalmente são intrusivas no ambiente de desenvolvimento de software, obrigando a adoção de IDEs (*Integrated Development Environment*) ou *plug-ins* específicos, que nem sempre estão disponíveis ou são adotados por todos os membros da equipe.

1.2 OBJETIVOS

Devido ao fraco apoio existente para prevenir a entrada de artefatos quebrados no repositório (Duvall et al. 2007) e à natureza evolutiva do software, é proposta a abordagem Ouriço. Tal abordagem objetiva identificar artefatos inconsistentes antes que cheguem ao repositório e responder às seguintes questões:

Questão 1: O Ouriço é capaz de identificar artefatos inconsistentes antes que estes alcancem o repositório?

Questão 2: A utilização de tal abordagem resulta em algum atraso no ciclo de desenvolvimento de software?

Para responder essas questões foi desenvolvido o Ouriço que atua, principalmente, durante o comando de *check-in*. A abordagem Ouriço visa identificar artefatos inconsistentes antes que estes possam corromper o repositório. Vale ressaltar que essa abordagem visa identificar problemas em repositórios de SCV. Para isso, ela aplica verificações sobre os artefatos enviados no *check-in*. Essas verificações, compostas por filtros, são aplicadas de acordo com a necessidade de cada projeto, e quando algum problema é identificado uma

notificação é enviada para o desenvolvedor. Além disso, quando um problema é identificado os artefatos são rejeitados e o ciclo de *check-in* é imediatamente interrompido.

A abordagem Ouriço foi avaliada sobre quatro projetos que possuem características distintas. Esses experimentos foram executados em projetos reais por meio da reconstrução de seus históricos de desenvolvimento. Ou seja, todo o processo de desenvolvimento foi reproduzido através de uma ferramenta, discutida no Capítulo 6, que é capaz de repetir os *check-outs* e *check-ins* realizados pelos desenvolvedores. Portanto, além de responder às questões levantadas nessa seção, este trabalho apresenta os seguintes objetivos:

- Definir e implementar uma abordagem para prevenção de artefatos quebrados no repositório;
- Avaliar a abordagem proposta em projetos reais; e
- Identificar padrões a partir dos projetos avaliados que utilizam alguma abordagem de verificação de artefatos.

1.3 ORGANIZAÇÃO

O restante deste trabalho está organizado em 6 capítulos, além deste capítulo de introdução. O Capítulo 2 apresenta uma visão geral de GC e seus sistemas auxiliares. Neste capítulo são discutidos os SCV, Sistemas de Controle de Modificações (SCM) e Sistemas de Gerenciamento de Construção (SGC), sendo o primeiro e o último tratados com maior detalhe por serem mais relevantes para este trabalho.

No Capítulo 3 são discutidos trabalhos que apresentam abordagens similares ao Ouriço. Nesse capítulo é apresentada a Integração Contínua, realizada de forma manual e automática. Além disso, são apresentadas abordagens baseadas em percepção e na identificação de conflitos em repositórios de CG.

No Capítulo 4 a abordagem Ouriço é descrita, levando em consideração o ciclo proposto para o Ouriço confrontado com o ciclo tradicional de GC. Nesse capítulo foram descritos os comandos de *check-out*, *check-in*, *update*, adaptados para o Ouriço e, também, as tarefas de apoio necessárias, como, por exemplo, verificações e criação de um local protegido no repositório.

No Capítulo 5 são descritas as características de implementação do Ouriço. Nesse capítulo estão detalhados os módulos que foram implementados, as interfaces disponibilizadas para interação do usuário com a ferramenta e também os mecanismos de acompanhamento desta abordagem.

No Capítulo 6 é descrito o experimento realizado para extração de indicadores que avaliam a eficácia do Ouriço na identificação de defeitos. Nesse capítulo também é discutido se existem indícios sobre a interferência negativa da abordagem sobre o ciclo de trabalho do desenvolvedor.

Finalmente, no Capítulo 7 são discutidas as contribuições deste trabalho para o contexto de desenvolvimento de software, eventuais limitações da abordagem e os trabalhos futuros.

CAPÍTULO 2 – GERÊNCIA DE CONFIGURAÇÃO

2.1 INTRODUÇÃO

GC é uma disciplina da Engenharia de Software para o controle da evolução de sistemas de software (Dart 1991). Para controlar essa evolução, GC é apoiada por três classes de sistemas: Sistemas de Controle de Versão (SCV), Sistemas de Controle de Modificações (SCM) e Sistemas de Gerenciamento de Construção (SGC). Esses sistemas auxiliam em tarefas de GC desempenhadas ao longo do ciclo de vida de um projeto.

A GC surgiu nos anos 50 com o foco voltado para aplicações militares e aeroespaciais, tendo como principal objetivo controlar as modificações de documentos relacionados à produção de aviões de guerra e naves espaciais (Leon 2000, Hass 2003, Estublier et al. 2005). Entretanto, a partir dos anos 80, a CG passou a ser utilizada em organizações de desenvolvimento de software não militares (Leon 2000).

Ainda nos anos 80 foi publicado o Padrão IEEE 828 (IEEE 2005), um documento que auxilia na elaboração do plano de GC. Segundo tal norma, GC é “uma disciplina formal da engenharia de software que provê métodos e ferramentas para identificar e controlar o software ao longo do seu desenvolvimento e uso. As atividades de GC incluem: (1) a identificação e estabelecimento de *baselines*¹; (2) a revisão, aprovação e controle das modificações; (3) o rastreamento e relatório de mudanças; (4) as auditorias² e revisões do produto de software em desenvolvimento; e (5) a gestão de *releases*³ do software e atividades de entrega”. Para executar essas tarefas os interessados (i.e., desenvolvedores, gerentes de projetos, etc.) podem utilizar os sistemas listados anteriormente.

A primeira classe de sistemas, os SCVs, é caracterizada por disponibilizar um local para armazenamento de artefatos de software, chamado repositório, e apresentar um suporte para o compartilhamento de tais artefatos entre os usuários (Duvall et al. 2007). Para gerenciar estes artefatos, os SCVs oferecem uma interface, através de uma série de comandos (e.g., *check-in*,

¹ *Baseline* é o termo que representa um conjunto de artefatos de software revisados e aceitos que servem de base para futuros desenvolvimentos (IEEE 1990).

² Auditoria é uma verificação independente de funcionalidades ou conjunto de funcionalidades do produto para avaliar a conformidade com as especificações, normas, acordos contratuais ou outros critérios (IEEE 1990).

³ *Release* é uma versão particular de um item de configuração (i.e., artefato de software) que é disponibilizado para um propósito específico (IEEE 2005).

check-out, *update* entre outros), pela qual seus usuários podem obter artefatos, extrair informações e refletir suas contribuições sobre o repositório.

A segunda classe de sistemas, os SCMs, é capaz de prover um canal de comunicação que auxilia no desenvolvimento de software. Geralmente, os SCMs estão associados a requisições de manutenções, sejam elas corretivas, evolutivas, adaptativas ou preventivas. Tradicionalmente o SCM é visto como um canal de comunicação entre os desenvolvedores e o usuário final, que identifica falhas de software e as reportam ao fornecedor do software para que as correções sejam providenciadas. Os SCMs também podem ser utilizadas como guias de desenvolvimento, através de tickets, em que os desenvolvedores de uma equipe podem selecionar novas tarefas, de forma controlada, evitando que mais de um desenvolvedor realize a mesma tarefa de maneira sobreposta, evitando o retrabalho.

A terceira classe de sistemas, SGC, é composta por ferramentas capazes de realizar construções (do inglês, *building*) de softwares complexos através de comandos que simplificam tais tarefas. De acordo com [Lehman e Ramil \(2001\)](#), o software tende crescer e se tornar cada vez mais complexo, e devido a essa característica a construção dos softwares também se torna mais complexa e custosa para ser realizada da maneira tradicional. Uma alternativa para minimizar a complexidade de construção de software é o uso de SGC, que são sistemas capazes de realizar tarefas como compilação, execução de testes e outros, através de comandos que facilitam a execução dessas tarefas.

As próximas seções detalham os SCV e o SGC, devido à necessidade do entendimento dos conceitos aplicados a abordagem proposta. Deste modo, este capítulo está organizado com as seguintes seções: Seção 2.2 são discutidos os SCV; Na Seção 2.3 são discutidos os SGC; e, finalmente, na Seção 2.4 são feitas as considerações finais. Para maiores detalhes sobre SCM podem ser consultados em [Murphy \(2007\)](#), [Barnson et al. \(2009\)](#), [Murta \(2009b\)](#) e [Redmine \(2011\)](#).

2.2 SISTEMAS DE CONTROLE DE VERSÕES

O SCV possibilita o armazenamento, recuperação e comparação de versões de arquivos. Tal sistema permite ao usuário versionar artefatos de software e manter sobre eles um controle da sua evolução. Este controle pode ser realizado através dos comandos identificados anteriormente. Tais comandos tornam esse tipo de sistema capaz de permitir trabalho em paralelo sobre um grupo de artefatos.

A utilização de SCV auxilia no gerenciamento de artefatos de forma automática evitando trabalhos manuais e contraproducentes para a manutenção desses artefatos. O

gerenciamento de artefatos de forma manual, além de não fornecer nenhum suporte ferramental, é contraprodutivo devido a uma série de efeitos colaterais. Dentre esses efeitos colaterais é possível citar (Murta 2009a):

- Perda de artefatos - usuários de mesmos artefatos não possuem apoio para comparar documentos e realizar junção destes documentos, resultando em perda de códigos, modelos e outros, devido à sobreposição de arquivos com conteúdos dessincronizados, por exemplo.
- Não é escalável - à medida que o número de usuários e o tamanho dos projetos crescem a complexidade de gerenciar as modificações cresce.
- Histórico inconsistente - o histórico controlado de maneira manual transfere a complexidade de gerenciar informações, relacionadas à configuração e outras características, para o usuário final que pode cometer erros.

Por outro lado, na abordagem automática alguns comandos são utilizados para apoiar o desenvolvimento de software de forma mais formal e controlada. Os SCVs possibilitam obter artefatos a partir de repositórios, adicionar novos artefatos no repositório, comparar artefatos e outros. Além disso, diversos efeitos colaterais (e.g., a perda de código) são evitados pelos controles realizados pelos SCVs. Mais detalhes sobre repositórios e seus comandos são discutidos no restante deste capítulo.

2.2.1 INTERAÇÃO ENTRE REPOSITÓRIO E ESPAÇO DE TRABALHO

SCV são capazes de armazenar artefatos de software. O ponto principal de um SCV é o repositório, local onde residem os artefatos de software (Berczuk 2003). Além disso, prove controle de versão para os artefatos presentes nele (Dart 1991).

Os repositórios de GC podem ser centralizados ou distribuídos. Repositórios de GC centralizados, como exibidos na Figura 3, são aqueles que possuem um repositório único onde o desenvolvedor pode obter configurações e realizar contribuições. Além disso, este é o modelo padrão para SCV (Chacon 2009). Alguns exemplos de SCV com repositório centralizado são: CVS (Fogel 2003), Subversion (Collins-Sussman et al. 2004), Perforce (Wingerd 2005) e ClearCase (Bellagio 2005). Por outro lado, os repositórios distribuídos, como exibidos na Figura 4, são caracterizados pela presença de vários repositórios, locais ou remotos, em que os desenvolvedores podem obter artefatos. Neste tipo de repositório, o desenvolvedor tem a opção de obter o repositório inteiro, através do comando *clone*, realizar *check-ins* localmente e, posteriormente, integrar as alterações no repositório remoto, através

do comando *push*. Alguns exemplos de SCV distribuídos são: Git (Chacon 2009), Mercurial (O'Sullivan 2009).

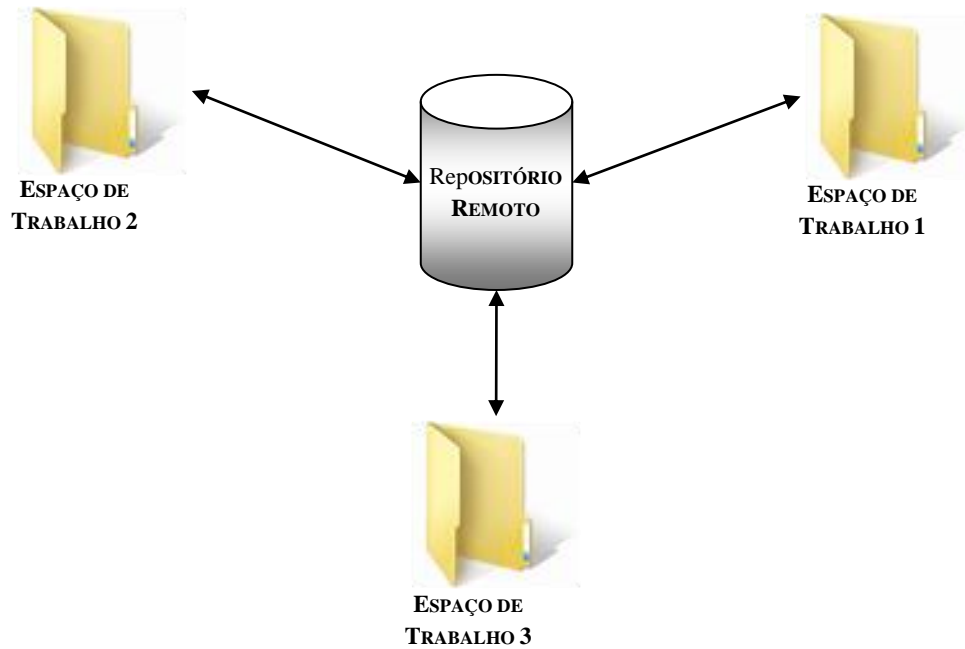


Figura 3. Esquema de um SCV centralizado.

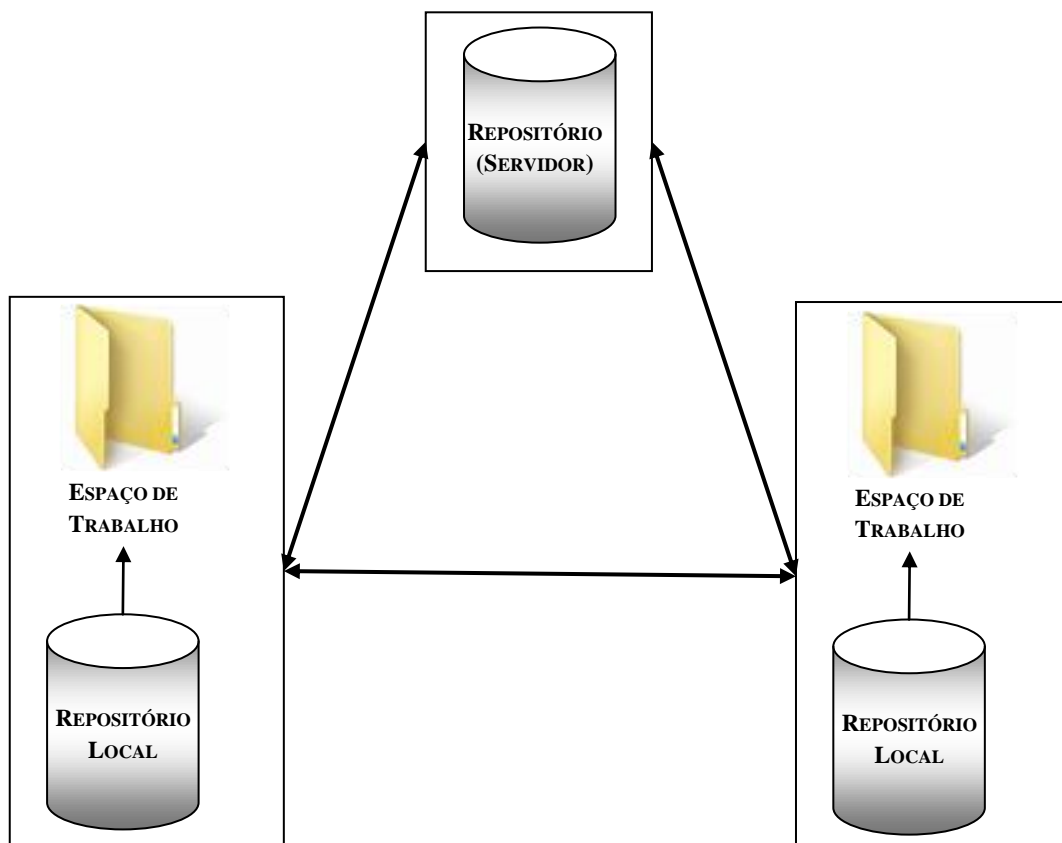


Figura 4. Esquema de um SCV distribuído.

Outro conceito importante utilizado é o de espaço de trabalho (do inglês, *workspace*). Segundo Dart (1991), o espaço de trabalho é projetado para impedir que desenvolvedores interfiram no trabalho dos outros. Deste modo, é possível prover uma forma de desenvolvimento onde os desenvolvedores que utilizam um SCV possam executar suas tarefas (e.g., codificação, compilação e execução de testes) de maneira isolada (Estublier 2000, Berczuk 2003).

Durante um ciclo de GC, o fluxo de dados do repositório para o espaço de trabalho, e vice-versa, é realizado através de comandos disponibilizados pelos SCV. O ciclo de GC geralmente segue os seguintes passos: (1) *check-out*, comando responsável por obter configurações de um repositório para um espaço de trabalho; (2) modificações, onde são concretizadas as contribuições de um usuário por adição, remoção ou edição de artefatos de software no espaço de trabalho; e (3) *check-in*, comando que reflete no repositório as alterações realizadas no espaço de trabalho. Alternativamente, o comando de *update* (4) pode ser executado para refletir no espaço de trabalho as alterações que ocorreram no repositório. Esse ciclo de GC é ilustrado na Figura 5.

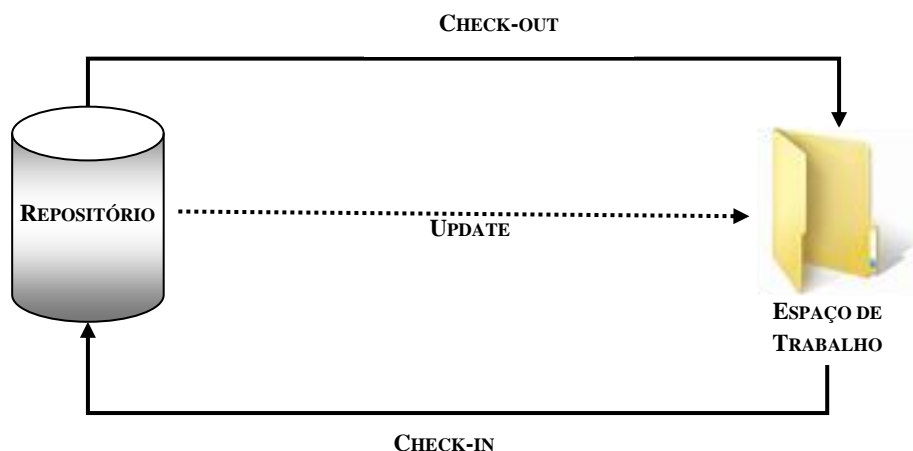


Figura 5. Ciclo básico de Gerência de Configuração.

Durante o *check-in* são usualmente realizadas verificações sobre os artefatos que entrarão no repositório. No intervalo do *check-out* até o *check-in*, diversos *check-ins* podem ter entrado no repositório de GC. Portanto, durante a realização do *check-in* é feita uma busca por conflitos físicos que podem ocorrer em casos onde dois ou mais desenvolvedores editam a mesma área, de forma paralela, de artefatos presentes no repositório. Este exemplo é ilustrado na Figura 6.

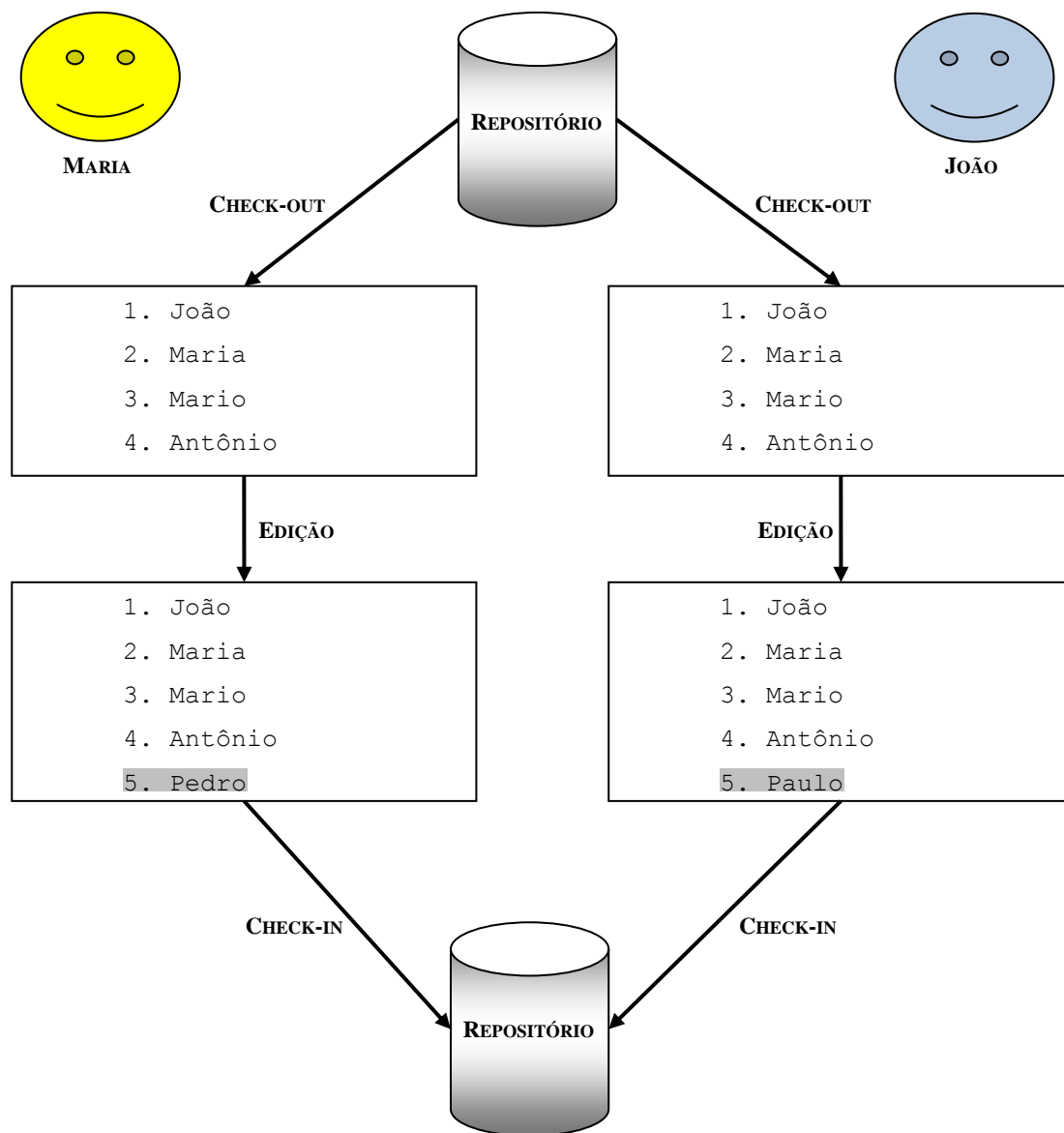


Figura 6. Exemplo de conflito físico.

Na Figura 6 dois desenvolvedores, Maria e João, editam o mesmo artefato em paralelo gerando conflito físico. Neste exemplo é apresentado um artefato, que possui a lista de desenvolvedores de um projeto de software. A edição feita por Maria, lado esquerdo da Figura 6, consiste em adicionar na linha 5 o nome de Pedro na lista de desenvolvedores do projeto. Paralelamente, João edita a mesma lista em seu espaço de trabalho, como exibido no lado direito da Figura 6, adicionando o nome de Paulo, também na linha 5. No momento de realizar *check-in*, supondo que Maria realize o *check-in* primeiro, a sua modificação vai entrar no repositório sem conflito. Porém, no momento em que João for realizar *check-in* um conflito físico será identificado e o *check-in* não entrará no repositório até que tal conflito seja resolvido.

Outro comando que possui particularidades é o de *check-out*. Durante os *check-outs* duas políticas para gerenciamento dos artefatos de software podem ser aplicadas (Prudêncio et al. 2009): (1) pessimista ou (2) otimista. A política pessimista é baseada em *check-outs* reservados, método que bloqueia os artefatos e impede desenvolvimento em paralelo. Deste modo, os desenvolvedores evitam gastar tempo com junções que não possuem apoio de ferramentas, podendo se tornar contraproducentes e desgastantes. Já a política otimista não se baseia em *check-outs* reservados, permitindo o desenvolvimento paralelo sobre os artefatos de software. Portanto, a política otimista é aconselhada para artefatos de software que possuem alta concorrência e fácil junção (Prudêncio et al. 2009).

2.2.2 GERENCIANDO O REPOSITÓRIO

Um repositório de SCV é usualmente dividido em três compartimentos lógicos para o armazenamento de artefatos de software. O primeiro compartimento, chamado de linha principal de desenvolvimento (do inglês, *mainline*), é o local onde a maior parte do desenvolvimento é realizada. O segundo compartimento, chamado de ramos (do inglês, *branches*), é um compartimento auxiliar, composto por várias linhas de desenvolvimento, onde pode ser realizado desenvolvimento de forma paralela. Este compartimento ainda possibilita o desenvolvimento de forma isolada por diversas restrições de projeto (e.g., evitar *check-ins* com artefatos que não estão aderentes a um determinado padrão na linha principal). Ainda nesse compartimento é possível realizar desenvolvimentos que demandem um longo tempo e, por isso, é desejável realizá-lo em outra linha. O terceiro compartimento, chamado de etiqueta (do inglês, *tag*), é o local onde configurações específicas, como *releases* e *baselines*, são armazenadas e recuperadas com maior facilidade.

Durante o ciclo de desenvolvimento de software novas configurações são geradas devido a manutenções que podem ser corretivas, evolutivas, preventivas ou adaptativas (Pressman 2000). Estas manutenções são realizadas via alterações (i.e., adição, remoção ou edição) em artefatos que podem ser identificadas via um comando, chamado de *diff*, que identifica áreas de artefatos que foram inseridas ou removidas. Essas modificações podem ser calculadas com relação a diferentes linhas de desenvolvimento de um repositório e, também, de uma linha de desenvolvimento em relação a um espaço de trabalho. Um exemplo desse comando é mostrado na Figura 7, que ilustra inclusão, feita por Maria, do nome de Pedro na lista representada na Figura 6. Nesse exemplo o *diff* é feito entre a configuração presente no espaço de trabalho e a configuração que está no repositório. O resultado indica que a linha “5. Pedro” foi adicionada, o que é representado pelo sinal de +. É importante ressaltar que o sinal

de + nem sempre indica algo que foi adicionado, pois ele é o sinal responsável por identificar a alteração de um desenvolvedor específico e não necessariamente para identificar adições no repositório.

```
$ git dif
diff -git a/lista b/lista
index 2917105..453ad1f 100644
--- a/lista
+++ b/lista
...
4. Antônio
+5. Pedro
```

Figura 7. Execução do *diff* no Git.

Quando as modificações identificadas pelo *diff* são desejadas pelo desenvolvedor em outra linha de desenvolvimento o comando de junção (do inglês, *merge*) pode ser executado. O comando de junção pode ser entendido como uma associação de *diff* e *apply*, ou seja, é um comando que identifica as modificações em uma linha de desenvolvimento e as aplica sobre uma configuração. Deste modo, quando um desenvolvedor realiza contribuições em um determinado ramo, isolado da linha principal de desenvolvimento, e deseja refleti-las na linha principal de desenvolvimento, um comando de junção pode ser executado enviando todas as contribuições do ramo para a linha principal.

2.2.3 EXTRAÇÃO DE DADOS DO REPOSITÓRIO

Os SCVs auxiliam na manutenção de artefatos de software e na recuperação do histórico de desenvolvimento. Tais sistemas possibilitam que o desenvolvedor acompanhe o ciclo de desenvolvimento de software de maneira autônoma. Desse modo, o desenvolvedor é auxiliado a responder perguntas como: quem alterou uma linha de desenvolvimento ou um artefato do repositório, quando foi feita a alteração e até o motivo da alteração.

Durante o ciclo de desenvolvimento de um projeto, podem existir interessados em listar artefatos que estão presentes no repositório sem a necessidade de executar um *check-out*. Uma possível causa pode ser a necessidade de saber quais artefatos estão presentes em uma determinada linha de desenvolvimento e, consequentemente, acompanhar o desenvolvimento de um projeto sem consumir tantos recursos de rede como um *check-out* poderia gastar. Para realizar esta tarefa os SCVs disponibilizam comandos que listam arquivos de repositório. Um exemplo para o CVS é ilustrado na Figura 8. E importante ressaltar que a sintaxe desse

comando só é válida da versão 1.12.8 em diante e, também, é assumido que a variável de ambiente CVSROOT está definida.

```
$ cvs ls -R
lista.txt
```

Figura 8. Listando arquivos de um repositório CVS recursivamente.

Informações sobre o histórico de desenvolvimento de um projeto também podem ser obtidas via comandos de SCV. Este comando, geralmente chamado de *log*, fornece informações como: quem executou *check-in* em uma determinada revisão, quando este *check-in* foi realizado e, possivelmente, o motivo da alteração.. Esse comando exibe como resultado a lista dos *check-ins*, autor, data e mensagem que descreve o objetivo do *check-in*. A Figura 9 mostra o comando *log* para o Subversion.

```
$ svn log
-----
R2 | Maria | 2010-09-29 07:13:30 -0200 (Wed, 29 Sep 2010) | 2 lines
Adicionando Pedro
-----
R1 | João | 2010-09-29 05:14:30 -0200 (Wed, 29 Sep 2010) | 2 lines
...
```

Figura 9. Exibindo *log* do Subversion.

2.2.4 ESTENDENDO SCV

Os SCVs apresentam mecanismos cujo objetivo é executar processamentos baseado em eventos disparados no repositório. Estes eventos podem estar ligados à execução de comandos, como *check-in*, *lock* e *update*, e outras ações disparadas no repositório a partir da interação com o desenvolvedor. Tal mecanismo, chamado de gatilho⁴ (do inglês, *hooks*), tem como exemplo tradicional a notificação de usuários sempre que um *check-in* ocorre. Neste caso, é utilizado um gatilho que é disparado após o *check-in* ocorrer de fato. Nessa seção são discutidos alguns gatilhos que são disparados pelo Git e outros pelo Subversion.

O Git é um SCV distribuído composto por 10 tipos de gatilhos que são disparados por diferentes tipos de eventos. Esses eventos são: *check-in*, *update*, *merge*, *receive* e *rebase*. Esses gatilhos são listados na Tabela 1.

⁴ Gatilhos são processos executados quando um evento em particular ocorre (Berlin and Rooney 2006).

Tabela 1. Gatilhos executados pelo Git (Chacon 2009).

Nome	Descrição
<i>pre-commit</i>	Gatilho executado antes do <i>check-in</i> (<i>commit</i>). Este gatilho pode abortar o <i>check-in</i> no caso de um <i>time-out</i> na conexão do cliente com o servidor.
<i>prepare-commit-msg</i>	Gatilho executado antes da mensagem de <i>commit</i> ser armazenada.
<i>commit-msg</i>	Gatilho que trata a mensagem de <i>commit</i> antes que os artefatos cheguem ao repositório.
<i>post-commit</i>	Gatilho executado após a conclusão do <i>commit</i> .
<i>pre-rebase</i>	Gatilho executado antes de aplicar o comando <i>rebase</i> ⁵ em alguma configuração.
<i>post-checkout</i>	Gatilho executado após o comando de <i>check-out</i> ser executado com sucesso.
<i>post-merge</i>	Gatilho executado após o comando de junção ser executado com sucesso.
<i>pre-receive</i>	Gatilho executado quando um cliente executa o comando <i>push</i> ⁶ .
<i>post-receive</i>	Gatilho executado após o comando de <i>push</i> ser completamente executado.

O Subversion é um SCV com repositório centralizado (Berlin and Rooney 2006) e é composto por 9 gatilhos que são disparados por eventos como: *lock*, *unlock*, *check-in* e alteração em propriedade da revisão. Esses gatilhos estão descritos na Tabela 2.

⁵ O comando *rebase* do Git responsável pela reintegração de um ramo a uma linha de desenvolvimento (Chacon 2009).

⁶ O comando *push* do Git responsável por enviar contribuições feitas no repositório local para um repositório remoto (Chacon 2009).

Tabela 2. Gatilhos Subversion (Berlin and Rooney 2006).

Nome	Descrição
<i>start-commit</i>	Gatilho executado antes da criação da transação do <i>check-in (commit)</i> . Este gatilho pode abortar o <i>check-in</i> no caso de um <i>time-out</i> na conexão do cliente com o servidor.
<i>pre-commit</i>	Gatilho executado depois da criação da transação, mas antes do <i>check-in (commit)</i> ser concluído.
<i>post-commit</i>	Gatilho executado após o <i>check-in (commit)</i> ser completado.
<i>pre-lock</i>	Gatilho executado antes de um <i>lock</i> exclusivo ser criado.
<i>post-lock</i>	Gatilho executado após a operação de <i>lock</i> ser concluída.
<i>pre-unlock</i>	Gatilho executado antes de uma operação de <i>lock</i> ser destruída.
<i>post-unlock</i>	Gatilho executado após uma operação de <i>unlock</i> ser concluída.
<i>pre-revprop-change</i>	Gatilho executado antes da propriedade de uma revisão ser alterada.
<i>post-revprop-change</i>	Gatilho executado após a propriedade de uma revisão ser alterada.

2.2.5 BOAS PRÁTICAS

SCV são ferramentas que possuem grandes utilidades no controle da evolução de artefatos de software. Contudo, se essa utilização for realizada de forma desordenada muitas das vantagens dos SCV não serão alcançadas.

SCV recebem contribuições através do comando de *check-in* e, portanto, este comando precisa ser executado seguindo padrões que garantam melhores resultados em repositórios de GC. Dessa forma, a utilização de boas práticas para a inclusão de artefatos é necessária para a obtenção de melhores resultados com um SCV. Algumas dessas boas práticas consistem na manutenção de artefatos sem conflitos, que tenham funções bem definidas e descrições consistentes. Com esses cuidados, o SCV é capaz de oferecer um histórico mais detalhado e manter o repositório íntegro.

Segundo Berlin e Roney (2006) quanto maior for o intervalo entre o *check-out* e o *check-in*, maior será o risco para o desenvolvedor. O primeiro motivo para essa afirmação é que o repositório apresenta uma estrutura mais segura que o espaço de trabalho, tanto de hardware quanto de software. Isso possibilita a recuperação dos artefatos em decorrência de um sinistro, por exemplo. A segunda razão é que uma vez que as contribuições estão no repositório, o desenvolvedor não precisa se preocupar com conflitos físicos entre suas configurações e as dos demais desenvolvedores. Portanto, a utilização de uma política que estimule a realização de *check-ins* cedo e sempre (do inglês, *commit early, commit often*) é desejável (Berlin and Rooney 2006).

Nesse sentido, outra boa prática descrita por BERLIN e RONEY (2006) é realizar modificações pequenas e atômicas. Juntamente com a responsabilidade de execução de *check-ins* mais cedo, outra responsabilidade que surge é a manutenção de um repositório íntegro.

Como já foi discutido anteriormente, o repositório é o local onde os desenvolvedores compartilham artefatos de software. Portanto, com modificações menores (i.e., alterando menos artefatos) e atômicas (i.e., alterando o mínimo de funcionalidades possível) o risco de conflitos físicos, problemas de compilação e falhas em testes ocorrerem também é menor. Todavia, caso algum desses problemas ocorra, a facilidade de identificação e correção é maior, pelo fato das alterações serem pequenas e atômicas, minimizando o espaço para buscar contribuições que apresentem problemas.

Uma maneira de conviver com estes defeitos e manter a linha principal de desenvolvimento consistente é realizar o desenvolvimento em ramos. Neste método de desenvolvimento o usuário realiza todo o desenvolvimento em ramos, que estão isolados da linha principal de desenvolvimento. Quando as alterações estão maduras o suficiente e, possivelmente, já foram verificadas uma junção é realizada para a linha principal de desenvolvimento. Essa tática de desenvolvimento é marcada pelo isolamento entre as linhas de desenvolvimento e possivelmente resultará em conflitos de junção. Quando uma junção resulta em conflitos o desenvolvedor será acionado para que a melhor decisão seja tomada, para resolver os conflitos identificados. Contudo, é importante ressaltar que os artefatos chegarão à linha principal com maior maturidade e possivelmente livre de defeitos.

Outra boa prática em SCV é a boa utilização de gatilhos. Os SCVs não possuem habilidades de realizar tarefas como compilação, execução de testes, envio de e-mail e outras tarefas devido ao seu propósito de apenas gerenciar artefatos. Entretanto, fornecem meios de extensão que tornam possível executar tais tarefas. Portanto, com a utilização bem feita de gatilhos é possível notificar desenvolvedores quando um *check-in* é realizado e inclusive executar, de maneira automática, tarefas de verificação sobre os artefatos presentes no repositório. Deste modo, a responsabilidade de execução de tarefas pode ser garantida, pois sempre que um determinado evento (e.g., *check-in*) for realizado no repositório todas as tarefas, descritas no gatilho, serão executadas. Além disso, a responsabilidade de execução sai de um humano, que pode falhar ou burlar os passos de um determinado processo.

Finalmente, o desenvolvedor deve executar testes sobre a configuração que está sendo enviada para o repositório. O repositório é um meio de compartilhamento que os desenvolvedores podem trabalhar de maneira paralela, portanto é importante que todas as linhas de desenvolvimento estejam sempre compilando e passando pelas baterias de testes desenvolvidas para os artefatos que a habitam. Isso é necessário, pois caso uma linha de desenvolvimento esteja inconsistente, os desenvolvedores poderão perder em produtividade devido à necessidade de corrigir defeitos que outros desenvolvedores inseriram no repositório.

Um repositório inconsistente pode resultar em retrabalho em casos como: a correção de um defeito por mais de um desenvolvedor.

2.3 SISTEMAS DE GERENCIAMENTO DE CONSTRUÇÃO

SGCs são capazes de criar objetos derivados (i.e., código compilado e os executáveis) a partir de artefatos de software e outros objetos derivados (Conradi and Westfechtel 1998, Asklund and Bendix 2002). Além disso, esses tipos de sistemas devem manter os arquivos gerados atualizados e, preferencialmente evitar construções desnecessárias.

A construção de sistemas é um processo composto por diversas tarefas que pode compreender compilação, execução de testes, entre outras. Durante o ciclo de execução dessas tarefas alguns defeitos podem ser identificados. Portanto, esses sistemas podem auxiliar na identificação de defeitos presentes em uma configuração.

No restante desta subseção são descritas tarefas que podem ser realizadas por SGC e ressaltados alguns ganhos obtidos com esse tipo de ferramenta. Estes ganhos podem estar relacionados à redução de tempo, queda da complexidade de construção de sistemas e automatização de tarefas realizadas durante o ciclo de desenvolvimento de software.

2.3.1 TAREFAS DOS SGC

Conforme descrito anteriormente, os SGCs são capazes de realizar tarefas como compilação, execução de testes e, além disso, podem fazer inspeção em artefatos de software, implantação, entre outras tarefas. Adicionalmente, os SGC podem reduzir o tempo em que tarefas, como a compilação e execução de testes são executadas.

Esses sistemas são capazes de executar essas tarefas sobre diversas linguagens de programação como C/C++, Java e Ruby. O primeiro SGC, chamado Make (Feldman 1979, Mecklenburg 2005), foi criado em 1979 com o intuito de ser apenas um sistema para construção de softwares escritos em C (Feldman 1979). Posteriormente, esse sistema passou a ter outras habilidades como o reaproveitamento de objetos construídos anteriormente em construções futuras e, deste modo, evitar reconstrução desnecessárias. Representando uma evolução do Make, nos anos 2000 surgiram o Ant (Holzner 2005) e, em 2004, o Maven (Sonatype 2008) que já incorporam outras evoluções como a utilização de um arquivo xml que os tornam independente de plataforma. Esses SGC são utilizados principalmente por desenvolvedores que utilizam a linguagem Java. Alternativamente, existe o Rake (Fowler, M. 2005), que trata de projetos que utilizam a linguagem Ruby.

SGC podem identificar artefatos que já foram compilados anteriormente resultando em construções posteriores mais rápidas (Estublier 2000). Na primeira vez em que um projeto é construído, todos os artefatos que o compõe serão transformados em artefatos derivados. Entretanto, quando já existem objetos derivados de uma configuração que está sendo construída, somente os artefatos que foram alterados ou incluídos, após a compilação anterior, serão transformados em objetos derivados. Esta capacidade dos SGC resulta em compilações mais rápidas, o que é desejável no contexto de software. O comando de compilação pode ser chamado de vários SGC. A título de exemplo, na Figura 10 está representada a chamada deste comando no Ant.

```
$ ant compile
Buildfile: /home/marapao/NetbeansProjects/Ant/build.xml
Compile:
[mkdir] Created dir: /home/marapao/NetbeansProjects/Ant/build/classes
[javac] Compiling 1 source file to /home/marapao/NetbeansProjects/A...
BUILD SUCCESSFUL
```

Figura 10. Compilação no Ant.

SGC possuem a habilidade de executar testes sobre artefatos de software, resultando em projetos construídos de forma mais segura (Sonatype 2008). Essa habilidade é possível através de *frameworks*, generalizados como XUnit (e.g., JUnit⁷ e NUnit⁸), que oferecem suporte a diversas linguagens, tornando possível a verificação das regras de negócios do software durante o ciclo de construção de um projeto. Deste modo, caso alguma regra de negócio seja quebrada, um desenvolvedor poderá corrigir o defeito que ocasionou a falha de um teste, por exemplo. O comando para execução de testes pode ser chamado de vários SGC. Contudo, para exemplificar, na Figura 11 está representada a chamada deste comando no Maven.

⁷ O JUnit é um *framework* criação de testes sobre artefatos escritos em Java (Massol and Husted 2003).

⁸ O NUnit é um *framework* para criação de testes sobre artefatos escritos em C#. (Hamilton 2004)

```

$ mvn test
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building svn
[INFO]    task-segment: [test]
[INFO] -----
...
-----
T E S T S
-----
Running br.uff.ic.gems.test.svn.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed:
0.033 sec
Results:

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
...

```

Figura 11. Execução de testes no Maven.

Outra etapa que é usualmente executada pelos SGC é o empacotamento, que tem como resultado o executável do projeto ou uma biblioteca, por exemplo. No caso da linguagem Java esse executável pode ser um JAR, gerado com ou sem dependência (e.g., bibliotecas, outros projetos e etc.) de acordo com a necessidade do desenvolvedor ou SGC utilizado. Quanto à obtenção de dependências, existem SGC, como o Maven, capazes de obter as dependências para um dado projeto que no momento de empacotamento serão anexados ao pacote gerado por essa etapa. O comando para empacotamento pode ser chamado de vários SGC. Contudo, para exemplificar, na Figura 12 está representada a chamada desse comando no Maven.

```

$ mvn package
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Oceano-Core
[INFO]    task-segment: [package]
[INFO] -----
[INFO] [resources:resources {execution: default-resources}]
...
[INFO] [jar:jar {execution: default-jar}]
[INFO] -----
[INFO] BUILD SUCCESSFUL

```

Figura 12. Execução da etapa de empacotamento no Maven.

2.3.2 AUTOMATIZANDO TAREFAS COM SGC

Conforme descrito anteriormente, SGC são capazes de realizar tarefas que são comuns no ciclo de desenvolvimento de software, através de comandos. Estes comandos são capazes de compilar, testar e, também, verificar características estáticas de artefatos de software (e.g., número máximo de linhas de código em artefatos de software). Tais comandos removem a complexidade da maneira tradicional de execução. Além disso, esses comandos podem ser aplicados diversas vezes para executar ações sobre um projeto. Essas ações podem ser: criação de *releases*, execução de auditorias funcionais, entre outras.

Durante a criação de uma *release*, é desejável que algumas verificações, como execução de testes automatizados, sejam realizadas sobre os artefatos que a compõe. SGC são essenciais para a realização destas tarefas uma vez que provêem um meio para as suas execuções e, conseqüentemente, retiram essa responsabilidade de um humano deixando-o livre para realização de tarefas que não podem ser automatizadas. Portanto, para a verificação das regras de negócio, um desenvolvedor pode criar um procedimento que descreve uma sequência lógica de passos para construção do projeto e execução dos testes. Esses passos podem ser executados quando, por exemplo, uma *release* for criada.

A automatização de análises estáticas também é possível via a associação de SGC e ferramentas como Findbugs (Hovemeyer and Pugh 2004) e PMD (PMD 2011). Para alguns SGC, como Ant e Maven, existem maneiras de integração que podem ser configuradas via arquivos especiais, discutidos posteriormente, que tornam a tarefa de verificação estática automática e, conseqüentemente, simples para a sua inclusão em outras tarefas.

Além da automatização de tarefas de desenvolvimento de software, os SGC são úteis para reduzir a complexidade das construções de projetos de software que tendem a aumentar continuamente. Lehman e Ramil (2001) afirmam que os sistemas evoluem para atender às demandas do mundo real. Este crescimento leva a sistemas mais complexos que resultam no aumento da complexidade de gerenciá-los e construí-los. Com a utilização dos SGC, essa tarefa tende a se tornar mais fácil, dado que, no pior caso, o procedimento será criado apenas uma vez pelo desenvolvedor e construções futuras serão realizadas com base no procedimento criado.

2.3.3 EXEMPLOS DE SGC

Os SGCs vêm evoluindo ao longo dos anos se tornando cada vez mais robustos e automáticos. Alguns exemplos de SCGs, que serão abordados nesta seção, são: Make, Ant e Maven.

O Make é um sistema criado para construir outros programas. No desenvolvimento de software os artefatos de software são constantemente alterados e, portanto, é necessário um meio pelo qual seja possível manter os executáveis sempre atualizados com os artefatos. Com Make, através de arquivos geralmente denominados *makefile*, é possível executar essa tarefa de forma simples e demandando menor esforço para realização da construção, quando comparada a execução manual (Feldman 1979).

O *makefile* segue uma estrutura padrão composta por objetivos, pré-requisitos e comandos. Os objetivos são objetos ou arquivo que precisam ser construídos como, por exemplo, um executável. Os pré-requisitos são representados por dependências que devem existir para que o objetivo seja construído com sucesso. Finalmente, os comandos são executados para que o objetivo seja atingido a partir dos pré-requisitos listados. É importante ressaltar que a construção de cada arquivo segue um objetivo diferente. Portanto, para projetos complexos, o Make ainda é complexo mesmo que apresente alguns recursos como, por exemplo, compilar apenas artefatos de software alterados após a última construção.

Para realizar tarefas por meio do Make é necessário escrever um *makefile* com tantos objetivos quanto forem necessários, como o exemplo da Figura 13. Nesse exemplo é possível visualizar quatro objetivos que são: (1) obtenção do arquivo *lexer.c* a partir da execução do comando flex com a opção `-t` sobre o arquivo *lexer.l*, (2) obtenção do arquivo *lexer.o* a partir da execução do compilador (gcc) sobre o arquivo *lexer.c*, (3) obtenção do arquivo *count_words.o* a partir da execução de compilação no arquivo *count_words.c* e (4) obtenção do executável *count_words* a partir da execução do compilador sobre os objetos *count_words.o* e *lexer.o* com a opção `-lfl`, dizendo que os objetos precisam ser agregados ao executável.

```
(4)count_words: count_words.o lexer.o -lfl
    gcc count_words.o lexer.o -lfl -ocount_words
(3)count_words.o: count_words.c
    gcc -c count_words.c
(2)lexer.o: lexer.c
    gcc -c lexer.c
(1)lexer.c: lexer.l
    flex -t lexer.l > lexer.c
```

Figura 13. Makefile para compilação de dois arquivos.

Essa aplicação possui apenas dois arquivos base, resultando na execução de 4 comandos. Portanto, o *makefile* e, conseqüentemente, o Make possuem um reflexo do número de artefatos base que são necessários para a geração dos executáveis. Dessa forma, quanto maior o número de artefatos base, mais complexo o *makefile* pode se tornar.

Por outro lado, o Ant concentra o esforço para construção no *buildfile*, arquivo de configuração do Ant. O *buildfile*, geralmente chamado de *build.xml*, fica na raiz do projeto, embora não seja necessário. A sua estrutura apresenta mais recursos para execução de tarefas que são configuradas com mais facilidade, se comparada ao *makefile*. No exemplo exibido na Figura 14, é ilustrada a compilação de um projeto (em negrito). De acordo com este exemplo, é possível identificar que a compilação de um projeto é menos complexa que a apresentada anteriormente (Figura 13). Essa queda da complexidade se justifica, pois a complexidade de compilação do projeto não aumenta simplesmente com o aumento do número de artefatos de software.

```
<project name="Projeto" default="dist" basedir=". ">
  <description>
    Exemplo simples de um arquivo de construção
  </description>
  <property name="src" location="local do código fonte"/>
  <property name="build" location="local do executável"/>
  <property name="dist" location="diretório do projeto"/>

  <target name="init">
    <tstamp/>
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
    description="compilar código">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>
</project>
```

Figura 14. Estrutura de um *buildfile* para compilação de um projeto.

Por fim, o Maven possui organizações padrões de projetos que os desenvolvedores devem seguir, levando em consideração o tipo de projeto, o que torna possível a execução de tarefas, como a de compilação e execução de testes, sem nenhuma configuração prévia. Portanto, quando se compara um arquivo *build.xml* com o *pom.xml* (o arquivo de configuração do Maven) a diferença identificada é enorme, pois ao contrário do arquivo do Ant (i.e., *build.xml*) o *pom.xml* apresenta dados como quais são as dependências do projeto, mas a descrição de como compilar e como realizar outras tarefas são desnecessárias.

Outra característica do Maven é a capacidade de obter dependências de repositórios remotos para executar a construção de projetos de forma automática. Pelo *pom.xml*, presente na raiz de projetos que seguem a estrutura do Maven, é possível inserir dependências (e.g., JAR's, WAR's e outros) que são obtidas durante o processo de construção de projetos de software. Deste modo, o processo de construção é facilitado, uma vez que a responsabilidade

de obtenção destes artefatos fica a cargo do SGC e não do desenvolvedor. Um esquema simplificado para exemplificar a estrutura geral do *pom.xml* é mostrado na Figura 15.

```
<Project>
  <name>Meu Projeto</name>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Figura 15. Esquema do arquivo de configuração do Maven.

2.4 CONSIDERAÇÕES FINAIS

GC é uma disciplina necessária para acompanhar a evolução de sistemas. A GC propicia o acompanhamento sistemático através de suas tarefas, descritas na IEEE Std 828 (IEEE 2005). Estas tarefas podem ser: auditoria; análise de impacto; entre outras. Portanto, a utilização dos SCV, SCM e SGC de forma ordenada pode ser eficaz para garantia da evolução de software, na realização das tarefas projetadas.

Conforme discutido, os SCV podem oferecer meios de compartilhamento propiciando o desenvolvimento de projetos de maneira distribuída. Além disso, esses sistemas armazenam o histórico de desenvolvimento de software quando boas práticas são seguidas. Entretanto, possuem limitações quanto à identificação de artefatos que podem deixar o repositório inconsistente. Deste modo, artefatos que não compilam e/ou não passam nos testes podem alcançar o repositório sem que nenhuma verificação, de compilação e/ou das regras de negócios, seja executada.

Os SGC são sistemas capazes de construir projetos e também identificar falhas durante o ciclo de construção. SGC são sistemas capazes de realizar tarefas que, a princípio, seriam realizadas pelo desenvolvedor, propiciando que este realize tarefas voltadas para o desenvolvimento de projetos e não gaste tempo com tarefas que podem ser automatizadas com esses sistemas. Porém, este tipo de sistema está tipicamente desvinculado dos SCV.

Os sistemas de GC são independentes e a integração deles pode resultar em benefícios relacionados à consistência de artefatos presentes em um repositório. Além disso, os sistemas de CG se complementam, suprimindo características que apenas um sistema não é capaz

desempenhar. Portanto, a integração desses sistemas pode resultar no aumento da robustez do repositório, através de suas características complementares.

CAPÍTULO 3 – MANUTENÇÃO DA INTEGRIDADE DE REPOSITÓRIOS

3.1 INTRODUÇÃO

Conforme descrito no Capítulo 2, os sistemas de CG atuam de forma isolada, possibilitando que artefatos quebrados (i.e., artefatos que não compilam, não passam em bateria de testes, etc.) alcancem o repositório tornando-o inconsistente. A manutenção de um repositório, disponibilizado por um SCV, deve ser feita de maneira cuidadosa e seguindo as boas práticas discutidas na Seção 2.2.5. Porém, essa tarefa não precisa ser exclusiva dos desenvolvedores, possibilitando o surgimento de abordagens automáticas para verificação da qualidade de artefatos presentes em repositórios e em espaços de trabalho.

As duas principais quebras que podem ser encontradas em repositórios de SCV são (Mens 2002): (1) quebra sintática e (2) quebra semântica. A **quebra sintática** é caracterizada pela existência de artefatos que não compilam no repositório. A **quebra semântica** é caracterizada pela existência de artefatos de software que não passam em testes, ou seja, quebram alguma regra de negócio. Essas quebras deixam o repositório inconsistente, exigindo que os desenvolvedores façam correções desnecessárias antes de realizarem as manutenções previstas.

Portanto, a verificação de artefatos é importante. Foram encontradas abordagens que visam manter os desenvolvedores cientes de artefatos que podem tornar o repositório inconsistente (i.e., abordagens de percepção) e abordagens para identificar artefatos quebrados presentes no repositório (i.e., IC). Essas abordagens são importantes, pois a manutenção de um repositório consistente, ou seja, um repositório que privilegie a produtividade do desenvolvedor, evitará correções desnecessárias antes da execução de tarefas relacionadas ao desenvolvimento de software.

Visando manter o desenvolvedor ciente de que alterações em seu espaço de trabalho podem resultar em problemas futuros são utilizadas abordagens baseadas em percepção (do inglês, *awareness*). Nestas abordagens são realizadas verificações e quando alguma irregularidade (e.g., conflito físico e quebra sintática ou semântica) é encontrada, o usuário é alertado para que alguma atitude preventiva seja adotada para evitar conflitos ou o envio de artefatos quebrados para o repositório.

Outra abordagem que visa verificar artefatos e alertar os usuários sobre o estado corrente do repositório é a Integração Contínua (IC). Esta abordagem atua em uma etapa

posterior à abordagem de percepção. Além disso, esta abordagem pode ser realizada de duas formas principais: (1) manualmente ou (2) automaticamente, sendo a última apoiada por servidores de integração contínua.

No restante deste capítulo são discutidas com maior profundidade abordagens de percepção, integração contínua e as abordagens para manutenção da integridade de repositórios. Portanto, na Seção 3.2 são discutidas abordagens baseadas em percepção dando enfoque no que é esperado de tais abordagens e, também, na descrição de algumas destas abordagens. Na Seção 3.3, é discutida a IC, manual e automática, e suas práticas. Na Seção 3.4 são discutidas abordagens para manutenção da integridade de repositórios, que têm um caráter proativo na identificação de problemas. Finalmente, na Seção 3.5, são feitas as considerações finais deste capítulo.

3.2 ABORDAGENS BASEADAS EM PERCEPÇÃO

Segundo Dourish e Bellotti (1992), percepção “é o entendimento da atividade dos outros interessados (e.g., desenvolvedores, gerentes de projeto e outros) que proveem um contexto para sua própria atividade”. O conceito de percepção está fortemente ligado em manter um desenvolvedor ciente de que as contribuições realizadas por ele são relevantes para a equipe de desenvolvimento como um todo. Além disso, percepção viabiliza a avaliação de atividades com relação a objetivos e progressos alcançados, tornando possível o gerenciamento de trabalho colaborativo.

No contexto de desenvolvimento de software, esse conceito apresenta uma importância muito grande devido ao aumento da complexidade dos softwares e a necessidade de desenvolvimento com desenvolvedores distribuídos geograficamente (Biehl et al. 2007). Sistemas complexos precisam ser desenvolvidos de maneira colaborativa (Dewan and Hegde 2007), portanto existe a necessidade de ferramentas para coordenação de projetos como um método de evitar problemas futuros como: conflitos físicos ou artefatos quebrados em repositórios de GC.

Utilizando apenas SCV, artefatos sintaticamente ou semanticamente quebrados podem chegar facilmente ao repositório sem que sejam identificados, resultando na detecção de defeitos de forma tardia (Sarma et al. 2007). Artefatos quebrados são criados no espaço de trabalho e introduzidos no repositório no momento em que o comando de *check-in* é executado. Por isso, as abordagens de percepção, para identificação de conflitos e quebras, visam identificar futuros problemas em tempo de desenvolvimento, ou seja, quando os artefatos ainda estão no espaço de trabalho do desenvolvedor.

A comunidade de GC reconheceu recentemente o potencial de abordagens baseadas em percepção (Sarma et al. 2008) por apresentarem um caráter proativo na detecção de erros. A intenção dessa frente é manter o usuário informado continuamente sobre alterações realizadas em outros espaços de trabalho e, deste modo, antecipar possíveis problemas, que podem ocorrer no momento em que os desenvolvedores enviam as suas contribuições para o repositório. Deste modo, percepção alerta aos interessados que uma situação não planejada está prestes a acontecer, aumentando o poder de decisão do desenvolvedor que só fará uma contribuição quebrada caso deseje assumir o risco.

Alguns pontos de que tornam este tipo de abordagem interessante são descritos a seguir (Dewan and Hegde 2007):

- **Detecção de conflitos físicos antecipados:** Os conflitos podem ser capturados enquanto o desenvolvedor está implementando suas tarefas no espaço de trabalho e, deste modo, pode tomar providências para solução destes conflitos antes que de fato ocorram.
- **Notificação de problemas baseados em dependências:** O sistema pode usar informações sobre dependências entre elementos dos programas em versões que sofreram *check-out* para notificar desenvolvedores sobre conflitos físicos e quebras sintáticas ou semânticas.

A resolução de conflitos e quebras em etapas de desenvolvimento seguintes à criação de um artefato de software é uma atividade complicada. Com a detecção de conflitos e quebras durante o desenvolvimento de artefatos de software, o desenvolvedor não necessita voltar ao contexto de desenvolvimento anterior. Ou seja, entender completamente os problemas identificados e encontrar uma combinação de artefatos de software que faça sentido para solucionar tais problemas. Solução esta que pode envolver outros membros da equipe devido a sua complexidade (Sarma et al. 2007).

Para embasar o Ouriço, foram identificadas algumas abordagens de percepção que apresentam características distintas quanto à identificação de irregularidades. Para identificar tais abordagens foram realizadas buscas em fontes como ACM e IEEE com as seguintes palavras chave: “*awareness repository*”, “*awareness workspace*” e “*awareness configuration management*”. Dessa pesquisa foram identificados diversos resultados que ainda passaram por uma filtragem para descartar trabalhos que não possuem relação com o tema deste trabalho. A partir dos resultados anteriores foram realizadas buscas de artigos que eram referenciados ou referenciavam tais trabalhos. Ao final da análises foram identificados os seguintes trabalhos: Dourish e Bellotti (1992), Sarma et al. (2003), Ganoe et al. (2004), Ripley et al. (2004), Biehl

et al. (2007), Dewan e Hegde (2007), Sarma et al. (2007), Al-ani et al. (2008), Panjer et al. (2008), Sarma et al. (2008), Brun et al. (2010), Schmeelk (2010) e Servant et al. (2010).

Dos trabalhos identificados dois chamaram mais atenção pelos diferentes *feedbacks* apresentados e conflitos que são capazes de identificar, tais trabalhos são descritos no decorrer desta seção. O primeiro, FASTDash (Biehl et al. 2007), foi escolhido por ser uma abordagem que possui mais recursos visuais para identificação de possíveis conflitos e o segundo, Palantír (Sarma et al. 2003), foi escolhido por ser um trabalho que apresenta maior cobertura na identificação de conflitos físicos e quebras sintáticas ou semânticas.

3.2.1 FASTDASH

FASTDash (Biehl et al. 2007) é uma ferramenta para visualização das tarefas realizadas durante o desenvolvimento de software, mantendo os membros de uma ou mais equipes cientes das atividades dos outros membros. Com esse sistema, um desenvolvedor pode saber quem realizou *check-out*, quais artefatos foram alterados por um desenvolvedor e, também, quais classes ou métodos foram alterados. As alterações podem ser comentadas, permitindo que desenvolvedores possam dar informações adicionais como o motivo da alteração, por exemplo.

FASTDash prove um alto grau de ciência, de possíveis conflitos, aos desenvolvedores. Com a identificação de quais artefatos de software (i.e., métodos e classes) que estão sendo alterados, o desenvolvedor é alertado sobre a possibilidade de ocorrer conflitos. Isso ocorre, graças a capacidade do FASTDash de identificar conflitos físicos com maior precisão. Deste modo, tais conflitos podem ser solucionados no momento de desenvolvimento, ou seja, sem que o desenvolvedor perca o contexto das alterações.

FASTDash é implementado para dois SCV, um sistema interno utilizado na *Microsoft* e o Team Foundation Server (David et al. 2006). Adicionalmente, possui uma interface que facilita o trabalho do desenvolvedor e possibilita a visualização de artefatos que sofreram *check-out*, quais artefatos estão sendo editados e quem está editando, por exemplo. Uma representação da interface do FASTDash pode ser vista na Figura 16. Nessa figura é possível visualizar um ambiente onde três desenvolvedores, identificados pelas suas fotografias, estão trabalhando sobre o mesmo projeto.

Com esta abordagem é possível identificar futuros conflitos físicos que podem surgir no momento em que alguma interação (e.g., *check-in*, *update* e *merge*) com o repositório é realizada. A antecipação de conflitos pode evitar trabalhos posteriores que podem ser custosos e que, possivelmente, envolveriam vários membros de uma equipe. Contudo, os conflitos

físicos são apenas uma das classes de problemas que podem deixar repositórios inconsistentes. Portanto, abordagens que possibilitem a antecipação de mais problemas, como quebras sintática e semântica, podem contribuir ainda mais para desenvolvimento de software.

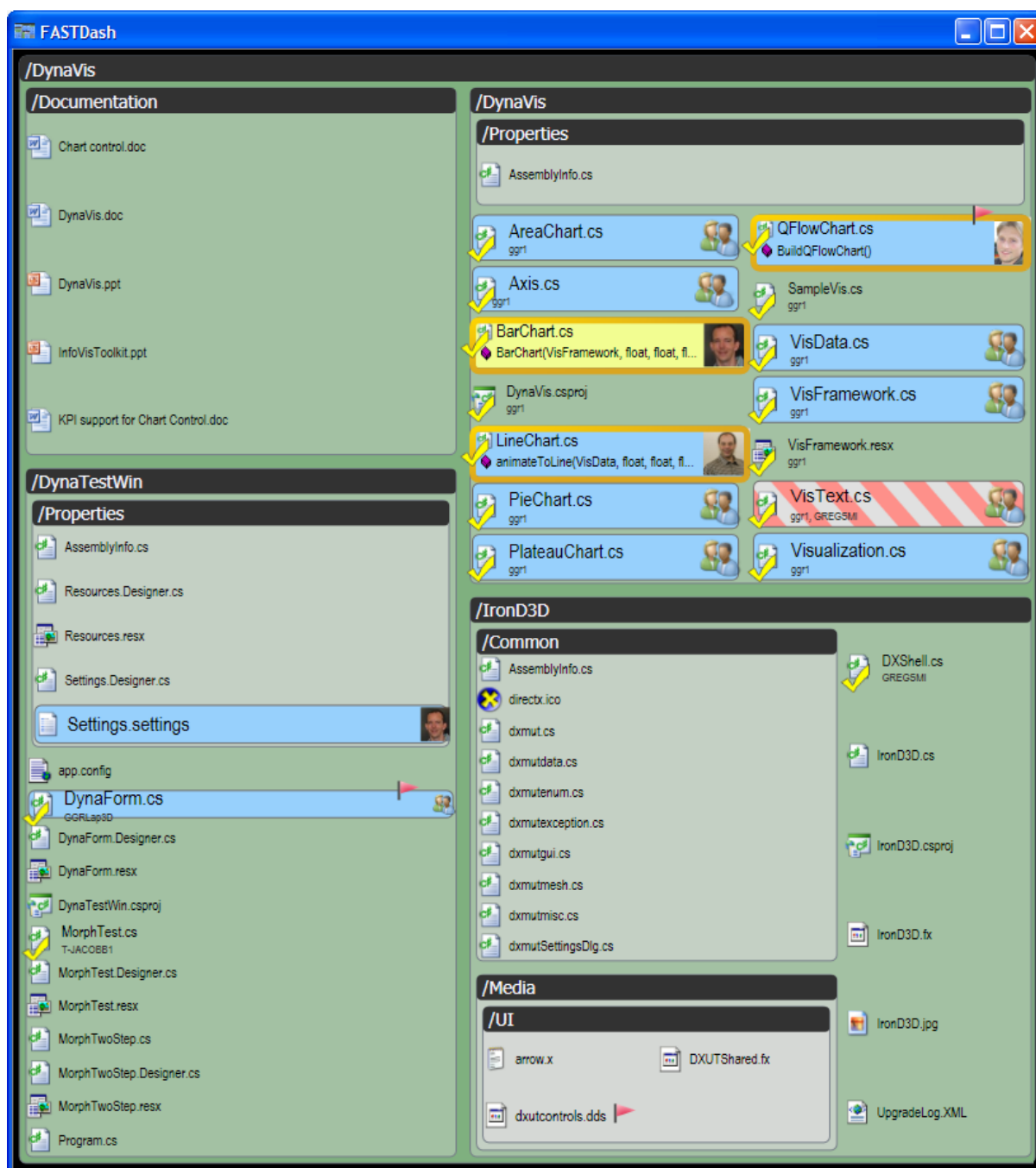


Figura 16. Interface para exibição de uma base de código compartilhada.

3.2.2 PALANTÍR

Palantír (Sarma et al. 2003) é uma ferramenta de GC para compartilhar alterações realizadas por desenvolvedores em seu espaço de trabalho com os demais desenvolvedores.

Essa abordagem tem por objetivo quebrar com o lado negativo do isolamento (i.e., desenvolvedores não sabem o que está sendo alterado por outros desenvolvedores em seus respectivos espaços de trabalho) e, deste modo, os desenvolvedores podem identificar problemas futuros durante a fase de desenvolvimento.

Esta abordagem apresenta suporte a dois SCV: o RCS (Tichy 1985) e o CVS (Fogel 2003). Este projeto foi implementado com módulos que atendem a apenas um SCV. Portanto, essa implementação apresenta dois pacotes para obtenção de dados do espaço de trabalho. Tais pacotes foram projetados para não terem mais funcionalidades que apenas emitir eventos relatando as operações realizadas por diferentes desenvolvedores.

As alterações ou ações realizadas em espaços de trabalho são capturadas através de eventos do *Palantir*, que seguem um mesmo fluxo padrão de tratamento. O tratamento dos eventos é realizado pelo *Palantir* da seguinte forma: (1) a ação é capturada, (2) a ação é interpretada, (3) uma verificação é feita para ver se a ação interessa ao *Palantir*, (4) se relevante, o tratamento apropriado para aquela ação é dado e (5) o alerta é construído e emitido.

Esta implementação tem o objetivo de ser não intrusiva, i.e., visa não introduzir novos passos além dos que já são executados normalmente pelo SCV; escalável, i.e., com o aumento do número de espaços de trabalho, o *Palantir* continua mostrando somente informações relevantes; flexível, capacidade de tratar diferentemente diferentes SCV; e configurável, para que cada desenvolvedor possa ser alertado da maneira que achar mais pertinente.

Além disso, tal implementação só é capaz de identificar conflitos físicos. Entretanto algumas melhorias foram feitas e, segundo SARMA, A. et al. (2008), o *Palantir* pode identificar quebras sintáticas e semânticas. Na Figura 17 é possível visualizar a interface de usuário para a IDE Eclipse. Nessa interface é possível visualizar artefatos que foram alterados no *package Explorer* (caixa na parte superior esquerda) . Também é possível visualizar o impacto dessas alterações, por arquivo, na caixa localizada na parte inferior da Figura 17.

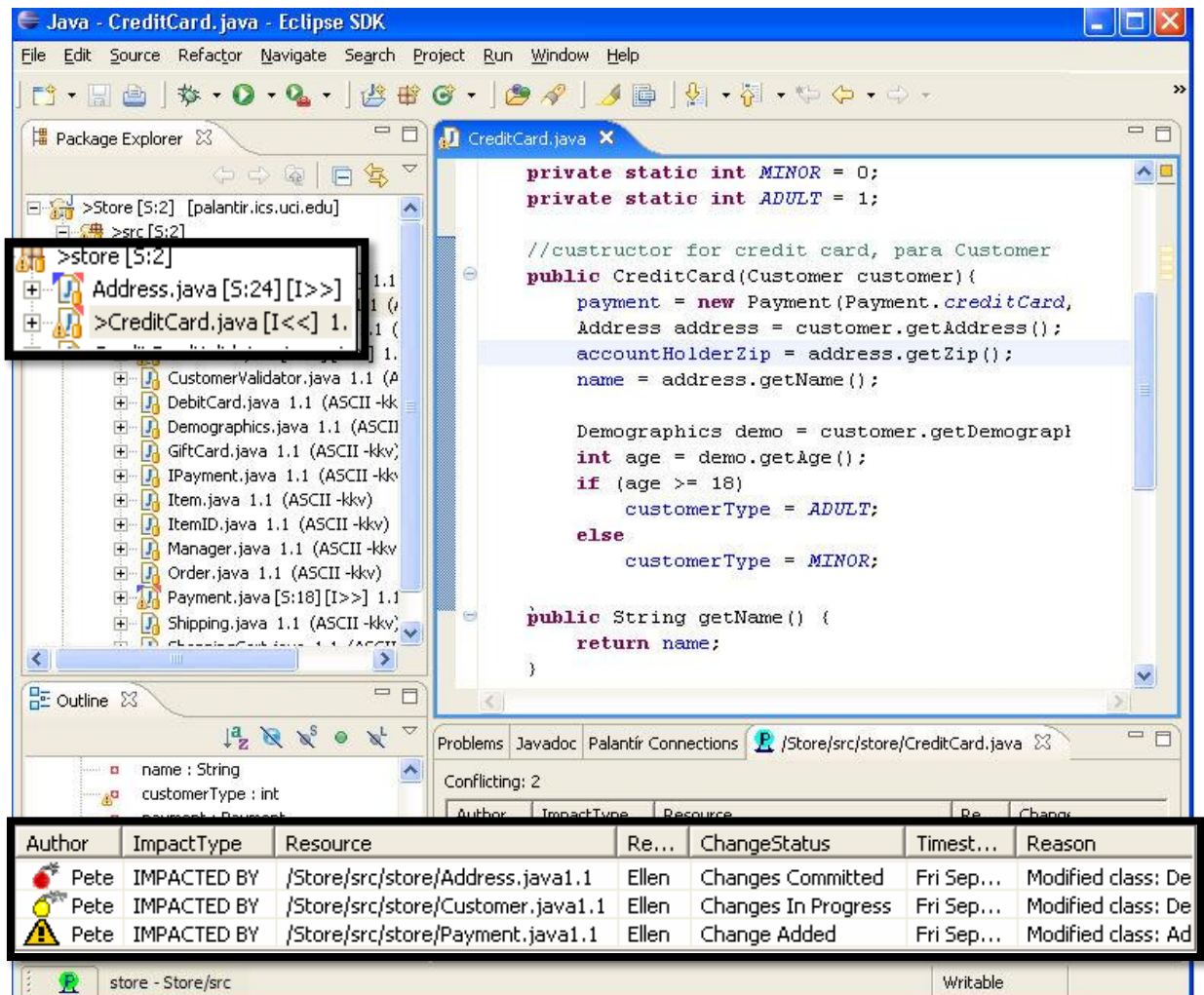


Figura 17. Interface de usuário do Palantir.

3.3 INTEGRAÇÃO CONTÍNUA

Segundo FOWLER (Fowler 2010), IC é uma prática de desenvolvimento de software onde membros de uma equipe de desenvolvimento integram o seu trabalho constantemente (i.e., pelo menos uma vez ao dia), levando a várias integrações por dia. Cada verificação é acompanhada por um processo de construção, que inclui compilação e testes, para detectar falhas o mais rápido possível.

Os usuários de IC acreditam que o tempo para solução de problemas em repositórios de GC está diretamente relacionado ao tempo que um desenvolvedor demora a encontrá-lo. Portanto, um rápido *feedback* é a solução para resolução de problemas (Fowler 2010). Com a integração de artefatos sendo realizada pelo menos uma vez ao dia, por desenvolvedores, várias construções serão realizadas no projeto em desenvolvimento. Deste modo, caso algum defeito exista, estará sempre à prova e possivelmente será capturado via execução das tarefas de construção do projeto.

Quando alguma irregularidade é encontrada nos artefatos de software, a correção dos problemas deve ser realizada imediatamente (Duvall et al. 2007). Conforme discutido no parágrafo anterior, o usuário de IC acredita no rápido *feedback* para correção de defeitos. Portanto, quando um defeito é identificado devido a, por exemplo, alguma falha nos testes, a solução deve ser imediata para que o risco de propagação do defeito seja minimizado. Essa propagação pode resultar em retrabalho para os desenvolvedores.

A IC pode ser executada de duas principais formas (Duvall et al. 2007): (1) manualmente e (2) automaticamente. Na forma manual, o trabalho é serializado e dificilmente pode ser escalável de acordo com o crescimento das equipes. Entretanto pode render bons resultados quando a consistência do repositório é primordial para a equipe de desenvolvimento. Na forma automática, as integrações podem ser facilmente escaláveis, pois esta abordagem é auxiliada pelos Servidores de Integração Contínua (SIC), responsáveis pela automatização do processo de integração e construção de projetos de software.

No restante desta seção são abordados assuntos como: formas de aplicar de IC, SIC e, também, algumas boas práticas que podem levar a melhores resultados. A aplicação dessas boas práticas juntamente com a utilização desta abordagem pode trazer retornos em termos de tempo, qualidade dos resultados e outros. Além disso, são discutidas algumas limitações que essa abordagem possui.

3.3.1 INTEGRAÇÃO CONTÍNUA MANUAL

Na IC manual o processo de integração é realizado individualmente, possibilitando que apenas um desenvolvedor realize *check-in* no repositório durante o intervalo de integração. Na integração manual são realizadas tarefas como: compilação, execução de testes, verificações estáticas e demais tarefas que sejam de interesse da equipe de desenvolvimento. Deste modo, essa abordagem pode ser contraprodutiva em casos onde a equipe de desenvolvimento é grande ou tenha tendência de crescer, devido ao gargalo humano.

A IC manual pode ser executada de diferentes formas, mas para exemplificar é utilizado o exemplo de TELES (2006), que é composto pelos seguintes passos:

1. Assegurar que o projeto compila e todos os testes automatizados executam com sucesso.
2. Conquistar a vez de integrar.
3. Criar *backup* do projeto na estação de trabalho.
4. Fazer *update* do projeto.

5. Assegurar que o projeto continua compilando e todos os testes executam com sucesso.
6. Realizar *check-in* do projeto.
7. Apagar o diretório do projeto da máquina de integração e realizar *check-out*.
8. Assegurar que o projeto continua compilando e todos os testes executam com sucesso.

Neste exemplo, foram dados alguns passos a serem seguidos e é assumido um cenário onde apenas uma estação de trabalho pode realizar integração por vez (passo 2). Os passos 1, 5 e 8 visam realizar verificações sintáticas e semânticas dos artefatos que estão sendo enviados ao repositório de GC. Estas verificações são feitas durante o envio de artefatos para o repositório e também quando já estão presentes no repositório. Como a estação de trabalho está sendo utilizada para realizar integração, é preciso garantir que eventuais falhas não interfiram no trabalho (passo 3), que as atualizações feitas no repositório reflitam no espaço de trabalho do desenvolvedor (passo 4) e que sejam verificadas (passo 5). Após verificar, as contribuições são enviadas para o repositório (passo 6), por medida de segurança o espaço de trabalho é apagado e uma nova cópia é requisitada do repositório (passo 7) sendo realizada uma nova verificação sobre os artefatos (passo 8). Com o ciclo concluído com sucesso, outro desenvolvedor pode iniciar o processo de integração.

Este meio de integração é bastante efetivo para evitar construções quebradas, devido à presença de artefatos inconsistentes no repositório. Entretanto pode se tornar inviável para grandes equipes de desenvolvimento (Duvall et al. 2007). Esta abordagem permite que apenas um desenvolvedor realize *check-in* de cada vez, resultando em *check-ins* serializados, ou seja, *check-ins* seguidos, mas não paralelos. Com essa prática é possível garantir maior confiança nos artefatos presentes no repositório. Contudo, pode atrapalhar o trabalho colaborativo, que é algo necessário em grandes equipes de desenvolvimento que cada vez mais lidam com o desenvolvimento de softwares complexos que devem ser desenvolvidos em curto espaço de tempo.

3.3.2 INTEGRAÇÃO CONTÍNUA AUTOMÁTICA

A IC automática é auxiliada por SIC, que são responsáveis por verificar os artefatos quando são enviados para o repositório. Na Figura 18 é ilustrado um esquema de como é realizada a IC de forma automática. Neste esquema o desenvolvedor realiza *check-out* e faz as alterações necessárias para completar uma tarefa e, em seguida, realiza *check-in* ou *commit* com as suas contribuições para o repositório. Neste momento, o Servidor de IC realiza a

construção do projeto, através de um script de construção, e, finalmente, um *feedback* reportando a situação do repositório é gerado e enviado aos interessados.

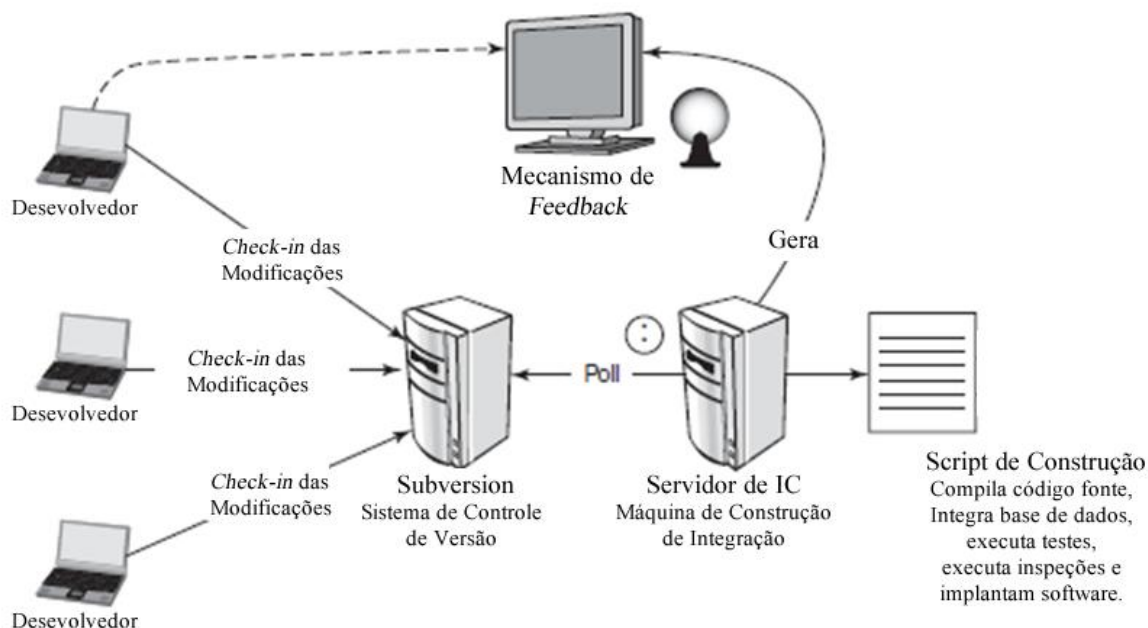


Figura 18. Ilustração da IC Automática, adaptado de DUVALL et al. (2007).

Comparado com a forma anterior, a IC Automática possui a vantagem de ser escalável e, deste modo, oferecer maior suporte ao trabalho colaborativo. Com a utilização de Servidores de IC, a responsabilidade de realizar construções da integração é retirada dos desenvolvedores. Portanto, os desenvolvedores podem realizar *check-in* sem a necessidade de conquistar a vez de integrar. Esse fator é fundamental para que os *check-ins* continuem sendo verificados sem a necessidade de um desenvolvedor realizar a construção e identificar problemas, resultando na eliminação do gargalo humano.

A verificação automática, no entanto, pode ser uma alternativa arriscada devido à possibilidade de artefatos quebrados poderem alcançar o repositório e torná-lo inconsistente. Com a forma automática de IC, os artefatos são enviados diretamente para o repositório sem que verificações sintáticas ou semânticas sejam realizadas, pois SCV não possuem habilidade de encontrar esse tipo de quebra. Portanto, os usuários de IC devem corrigir os defeitos identificados assim que notificados para tentar evitar tarefas indesejáveis, como a correção de artefatos defeituosos, impossibilitem outros desenvolvedores de prosseguirem com suas tarefas. Entretanto, nem sempre as correções são realizadas imediatamente, possibilitando que o repositório permaneça inconsistente durante longos períodos.

Vários Servidores de IC são disponibilizados para que equipes de desenvolvimento possam utilizar IC de forma automática. Alguns dos Servidores de IC mais populares são Hudson (Sun Microsystems 2011), CruiseControl (CruiseControl development team 2010) e Continuum (Apache Foundation 2010). Na Figura 19 é possível visualizar o painel do Hudson, que mostra uma lista de projetos e seus respectivos estados. Por exemplo, é possível identificar que o projeto *japex* está quebrado.

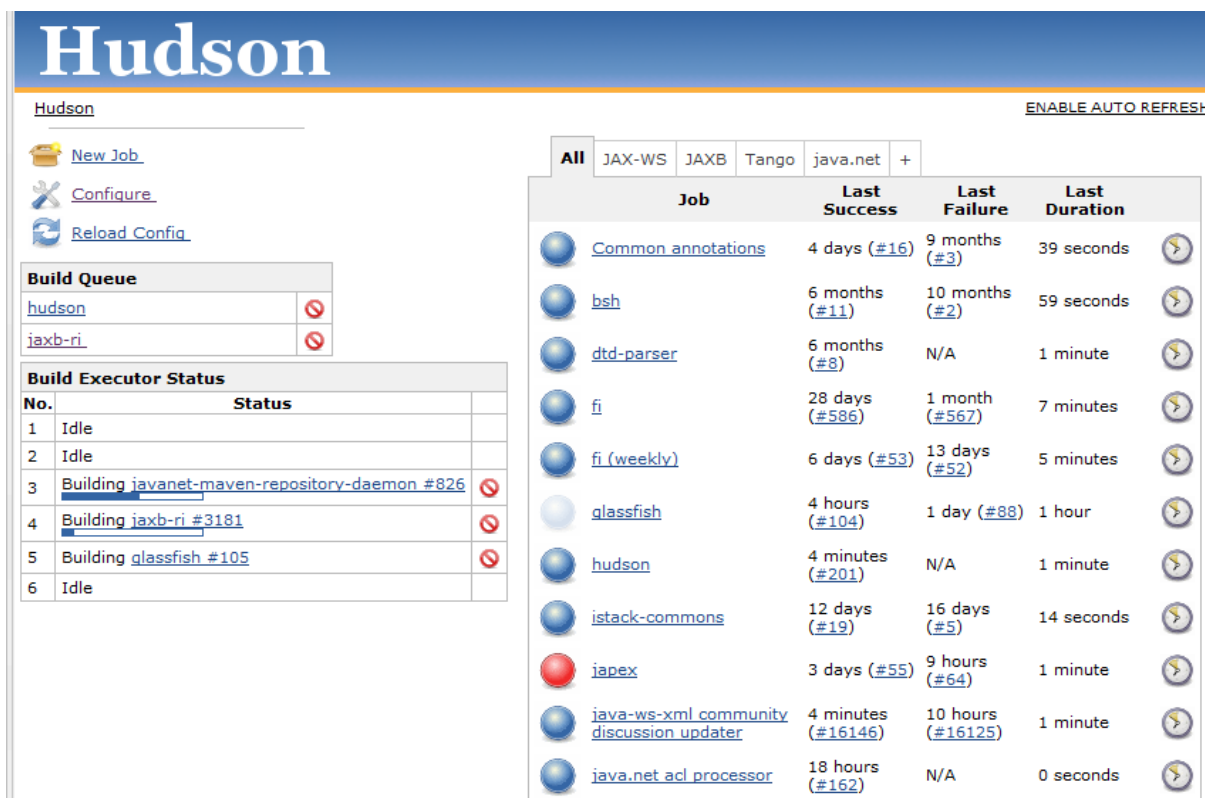


Figura 19. Painel para acompanhamento de projetos do Hudson.

3.3.3 PRÁTICAS DE INTEGRAÇÃO CONTÍNUA

FOWLER (2010) definiu um conjunto de 10 práticas para que melhores resultados sejam obtidos com a utilização de IC. Estas práticas estão voltadas para a manutenção dos artefatos de software, manutenção de um tempo regular para verificação de artefatos, visualização do estado da linha principal e obtenção de *feedbacks* rápido via Servidores de IC.

A **primeira prática** está relacionada ao armazenamento dos artefatos de software em um repositório único. Um dos fundamentos básicos de IC é que o projeto deve ser construído a partir de um comando, como se fosse um botão (Duvall et al. 2007). A manutenção dos artefatos em um único repositório pode auxiliar isso, pois com a combinação do comando de *check-out* e o comando de construção o projeto pode ser gerado sem maiores dificuldades.

A **segunda prática** listada por (Fowler 2010) é para automatizar o processo de construção. Com a utilização dos SGC é possível realizar construções de maneira automática, evitando construções complexas, que podem envolver diversas tarefas que podem ser automatizadas, e remover complexidades desnecessárias que podem levar a erros. Além disso, com a redução da complexidade, via automatização, muito tempo pode ser economizado.

A **terceira prática** se refere ao processo de construção ser autotestável. Com a utilização dos *frameworks* de teste, generalizados como XUnit, é possível que a construção, além de realizar compilação, também, execute testes sobre artefatos, identificando possíveis defeitos. Embora os testes não cubram 100% dos defeitos presentes em um projeto, muitos defeitos podem ser identificados via execução de testes.

A **quarta prática** aborda a periodicidade com que o *check-in* é realizado na linha principal: pelo menos uma vez ao dia. IC tem como um dos trunfos a comunicação entre membros de equipes, que podem saber de alterações feitas nos artefatos quando elas chegam ao repositório. Portanto, a realização do *check-in* é primordial para o bom funcionamento e obtenção de *feedbacks*.

A **quinta prática** diz que todo *check-in* na linha principal de desenvolvimento deve ser construído na máquina de integração. Durante o período de desenvolvimento, algumas configurações podem ser realizadas pelo desenvolvedor e deixar de serem automatizadas no *script* de construção, resultando em problemas no momento de construir o projeto em outra máquina. Portanto, o *check-in* não pode ser considerado completo até que a construção na máquina de integração seja realizada com sucesso.

A **sexta prática** está relacionada ao tempo de execução da construção, que deve ser o mais rápido possível. Para acelerar o processo de construção, as tarefas da construção devem ser mantidas executando rapidamente. Portanto, a execução do conjunto de testes deve ser mantida de forma que não ultrapasse um limite (e.g., 10 minutos). Para manter esse tempo ideal, geralmente, alguns testes são deixados de lado, resultando na execução de, por exemplo, testes de unidade e na não execução de testes de sistema, regressão, entre outros. Estes testes que não são executados continuamente (i.e., a cada *check-in*) podem ser executados de forma agendada (e.g., uma vez ao dia), resultando em melhores tempos de construção durante o *check-in*.

A **sétima prática** diz que o teste deve ser realizado, também, em um ambiente clone do de produção. O objetivo desta prática é realizar testes fora de um ambiente controlado (i.e., ambiente de desenvolvimento) onde o resultado de testes pode ser diferente dos demais

ambientes, ou seja, apresentar falhas que no ambiente de desenvolvimento não seria possível de identificar.

A **oitava prática** diz que o último executável do projeto deve estar em um local de fácil acesso para que todos possam utilizar. Esta prática é importante para que os interessados (e.g., desenvolvedores, gerentes de projeto, clientes, entre outros) possam fazer demonstrações, realizar testes exploratórios ou apenas visualizar o que mudou no software com relação a última construção ou ainda em um determinado período de tempo, caso possua as ultimas construções.

A **nona prática** está relacionada ao *feedback*, que é um dos principais objetivos de IC. Portanto, esta prática diz que todos os interessados devem ter acesso ao que está acontecendo com o projeto. IC é baseada na comunicação, portanto os interessados devem ter acesso à situação na qual se encontra um projeto, seja por meio de um painel demonstrativo, por e-mail, SMS ou qualquer outro meio. O importante é que todos possam ter acesso ao estado do projeto e que essa comunicação seja eficiente.

Finalmente, a **décima prática** está relacionada à realização de *deploy* de maneira automática. Com a utilização de IC é possível que existam múltiplos ambientes para produção, testes e outros, e os projetos serão implantados diversas vezes ao dia nesses ambientes. Portanto, a automatização da implantação de projetos, através de *script*, é uma boa prática, pois resulta na aceleração da identificação de defeitos e redução de erros no processo de implantação de projetos.

O conjunto de práticas descritas por Fowler (2010) pode levar ao melhor uso de IC e, deste modo, extrair melhores resultados e reduzir riscos, como a manutenção de um repositório inconsistente sem que nenhum alerta seja feito sobre o estado dos artefatos que compõe o repositório.

3.4 ABORDAGENS PARA MANUTENÇÃO DA INTEGRIDADE DE REPOSITÓRIOS

Além de percepção e IC, foram identificadas outras abordagens que realizam verificação em artefatos de software, mas que não se encaixam completamente no perfil das duas abordagens anteriores. Estes trabalhos, *Safe-commit* (Wloka et al. 2009) e *Repoguard* (Legenhausen et al. 2009), são descritos nesta seção. Vale ressaltar que o *Safe-commit* foi identificado através da pesquisa feita para o conjunto de abordagens de percepção e o *Repoguard* através de buscas por ferramentas similares à abordagem proposta nesse documento.

3.4.1 SAFE-COMMIT

O *Safe-Commit* (Wloka et al. 2009) é uma abordagem que auxilia o desenvolvedor a realizar verificações no espaço de trabalho e identificar modificações que podem ser enviadas para o repositório sem torná-lo inconsistente. Tal abordagem que divide as alterações em pequenas mudanças, chamadas de mudanças atômicas, e analisa se a inserção dessas mudanças resulta em algum problema, como, por exemplo, uma quebra sintática ou semântica.

Esta abordagem faz uso de políticas para decidir se uma mudança é bem vinda ao repositório ou, ainda, se deve ser rejeitada. Com a utilização das políticas permissiva, moderada e pessimista, os artefatos de software candidatos a serem enviados para o repositório recebem tratamentos diferentes. Tais tratamentos vão desde um menos severo (política permissiva), onde os artefatos modificados não precisam estar cobertos por testes; até um tratamento mais severo (política restritiva), onde os artefatos necessitam estar cobertos e passarem testes. No último caso o desenvolvedor tem maior confiança nos artefatos, pois serão semanticamente verificados.

Safe-Commit é uma abordagem que apresenta um foco diferenciado das demais, devido ao objetivo de identificar modificações que poderiam estar presentes no repositório e que não foram enviadas por insegurança do desenvolvedor. Portanto, com a utilização dessa abordagem, os desenvolvedores podem enviar modificações com maior segurança ao repositório e antecipar o compartilhamento de alterações com os demais membros da equipe, privilegiando o trabalho colaborativo.

3.4.2 REPOGUARD

O *RepoGuard* é uma ferramenta que visa combinar ferramentas de GC que atualmente “vivem em mundos separados”, i.e., sem muita interação (Legenhausen et al. 2009). Tal abordagem visa colocar para trabalhar em conjunto os SGC, SCM e SCV, permitindo que verificações sejam realizadas em artefatos antes que estes alcancem o repositório.

O *RepoGuard* é baseado em duas partes principais (Legenhausen et al. 2009): *checks* e os *handlers*. Essas partes devem ser configuradas pelos usuários que definirão quais tarefas serão realizadas para processar uma transação realizada no SCV. Quando uma transação é identificada pelos *hooks* de SCV, os *checks* realizam a validação do conteúdo da transação e os *handlers* fazem o tratamento dos resultados e/ou geram a saída dos *checks*.

Esta abordagem dá uma autonomia grande para os desenvolvedores que podem realizar as tarefas de verificação que acharem necessárias, entretanto alguns problemas podem ser

apresentados durante verificações muito longas (i.e., verificações que envolvem testes muito longos resultando em muito tempo para executá-los). Para verificar artefatos antes que estes cheguem ao repositório, esta abordagem utiliza um *hook pre-commit*, como discutidos no Capítulo 2, que, no caso do Subversion (Berlin and Rooney 2006), possui limitações quanto à conexão com a rede (i.e., no caso de *time-out* a transação não terá o resultado esperado).

3.5 CONSIDERAÇÕES FINAIS

As abordagens de verificação discutidas neste capítulo possuem características distintas quanto ao momento da verificação e, também, do ponto de vista de objetivos. Abordagens baseadas em percepção atuam no momento em que artefatos estão sendo desenvolvidos, portanto, o desenvolvedor possui maiores recursos para tomar decisões que possam evitar futuros problemas. Por outro lado, existem momentos em que o desenvolvedor poderia não desejar possuir este recurso ativado, pois o processo de desenvolvimento, em alguns casos, pode ficar comprometido com o número de informações gerado por esse tipo de sistema (Brun et al. 2010). Um exemplo desse cenário é quando o desenvolvedor está realizando suas tarefas e diversos alertas aparecem na tela dispersando-o da tarefa de desenvolvimento (Brun et al. 2010). Além disso, a abordagem de percepção não possui mecanismos que rejeitam a inclusão de artefatos que deixariam o repositório inconsistente, tornando-se um informante passivo, que não proíbe a entrada de artefatos quebrados no repositório. Deste modo, a utilização de abordagens de percepção juntamente com alguma abordagem capaz de rejeitar contribuições quebradas poderia contribuir para a manutenção de repositórios íntegros.

Outra abordagem discutida nesse capítulo foi a IC, que realiza verificações quando artefatos já estão presentes no repositório. A utilização de IC é uma prática bastante comum em empresas de desenvolvimento de software que utilizam uma metodologia de desenvolvimento ágil. Tal prática fornece informações relevantes sobre o estado de um repositório. Entretanto, é possível que esse *feedback* seja dado de maneira reativa. Ou seja, no momento em que outros desenvolvedores já estão de posse dos artefatos, que podem conter defeitos. Sem essa informação, os desenvolvedores podem realizar manutenções sobre artefatos inconsistentes ou ainda realizar retrabalho, pois mais de um desenvolvedor pode corrigir um mesmo defeito.

As abordagens para manutenção da integridade de repositório realizam verificações em outros momentos e dão informações relevantes para o desenvolvimento. Entretanto, essas abordagens possuem algumas limitações quanto à conexão de rede devido à necessidade de execução das tarefas de verificação durante o período de transferência dos artefatos, ou seja,

de maneira síncrona; e também não evitam que artefatos quebrados cheguem ao repositório, no caso do *Safe-Commit*. Deste modo, nessa abordagem é feita apenas uma identificação das modificações que podem ser enviadas ao repositório sem torná-lo inconsistente. Portanto, uma abordagem que identifique artefatos quebrados antes que eles cheguem ao repositório e tenha o poder de rejeitá-los, auxiliaria os desenvolvedores de software a manter repositório íntegro. A necessidade de uma abordagem com esse perfil é reforçada pela afirmação de Duvall et al. (2007), que destaca a falta de apoio ferramental para identificar e impedir que artefatos quebrados cheguem ao repositório.

CAPÍTULO 4 – OURIÇO

4.1 INTRODUÇÃO

A proteção de repositórios contra artefatos quebrados é uma tarefa que atualmente dispõe de pouco apoio ferramental (Duvall et al. 2007). Tal afirmação motiva o surgimento de abordagens que desempenhem este papel. A abordagem Ouriço surgiu dessa necessidade e, deste modo, auxilia os desenvolvedores a realizarem verificações em artefatos de software que entrarão em um repositório de GC e, quando necessário, rejeita tais configurações que podem tornar o repositório inconsistente. Quando uma configuração é rejeitada o desenvolvedor pode enviá-la após corrigir os erros identificados.

Essa abordagem estende o ciclo básico de GC para realizar tarefas de verificação sintática e/ou semântica e, deste modo, garantir confiança sobre os artefatos presentes no repositório de GC. O ciclo básico de GC é composto geralmente por *check-out*, modificações e *check-in*; ciclo este que cuida desde a obtenção de artefatos de software de um repositório até a inclusão das contribuições no mesmo. Além disso, tal ciclo pode ser auxiliado pelo comando de *update*. Durante a execução do comando de *check-in* os SCV são capazes de identificar apenas conflitos de origem física e, conseqüentemente, quebras sintáticas e semânticas chegam ao repositório sem que nenhuma verificação deste caráter seja realizada (Figura 20). Devido a essa deficiência, artefatos quebrados podem alcançar e permanecer em um repositório de GC até que sejam identificados posteriormente pelo próprio desenvolvedor ou por algum processo de auditoria.

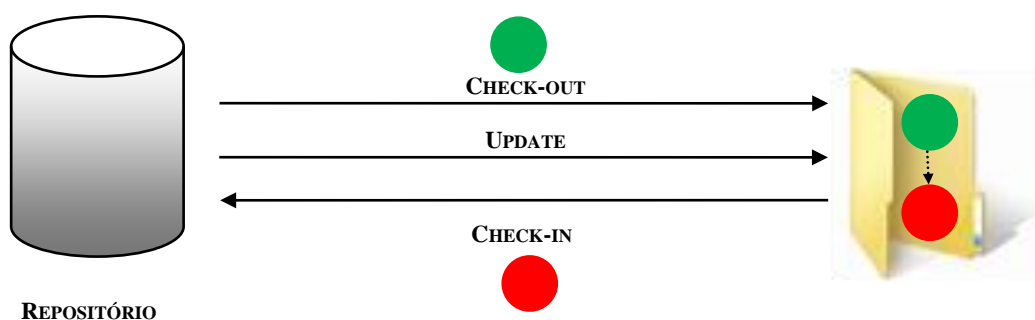


Figura 20. Ciclo tradicional de GC – *check-out*, *update* e *check-in*.

Para evitar tal cenário, a abordagem Ouriço é proposta e definida neste Capítulo. Tal Capítulo apresenta a seguinte divisão: Na Seção 4.2 é dada uma visão geral do Ouriço ressaltando o objetivo e mecanismos que viabilizam a verificação de artefatos; Na Seção 4.3 são descritos os filtros que são utilizados para identificar artefatos quebrados, que estão sendo

enviados para o repositório; Na Seção 4.4 são descritas as políticas permissiva, moderada e restritiva; A Seção 4.5 aborda os *autobranches*, que representam o meio de proteção do repositório; A Seção 4.6 apresenta o ciclo de desenvolvimento proposto pelo Ouriço e, também, são discutidas formas para resolução de problemas identificados pela abordagem; Na Seção 4.7 são discutidos cenários de utilização do Ouriço, onde existem ou não problemas a serem identificados; e, finalmente, na Seção 4.8 são feitas as considerações finais da abordagem.

4.2 VISÃO GERAL

Para evitar que um ou mais artefatos quebrados alcancem o repositório de GC, a abordagem proposta utiliza um mecanismo, chamado de *autobranch*, que viabiliza a verificação e, quando necessário, a rejeição desse tipo de artefato. Para viabilizar a rejeição de artefatos quebrados durante a execução do *check-in*, tarefas de verificação, de primeiro e segundo nível, são executadas para garantir a consistência dos artefatos antes que sejam integrados no repositório. Ou seja, se uma configuração apresenta algum artefato quebrado, ela é rejeitada; caso contrário, a configuração é enviada para o repositório que permanecerá consistente. Este esquema é ilustrado na Figura 21.

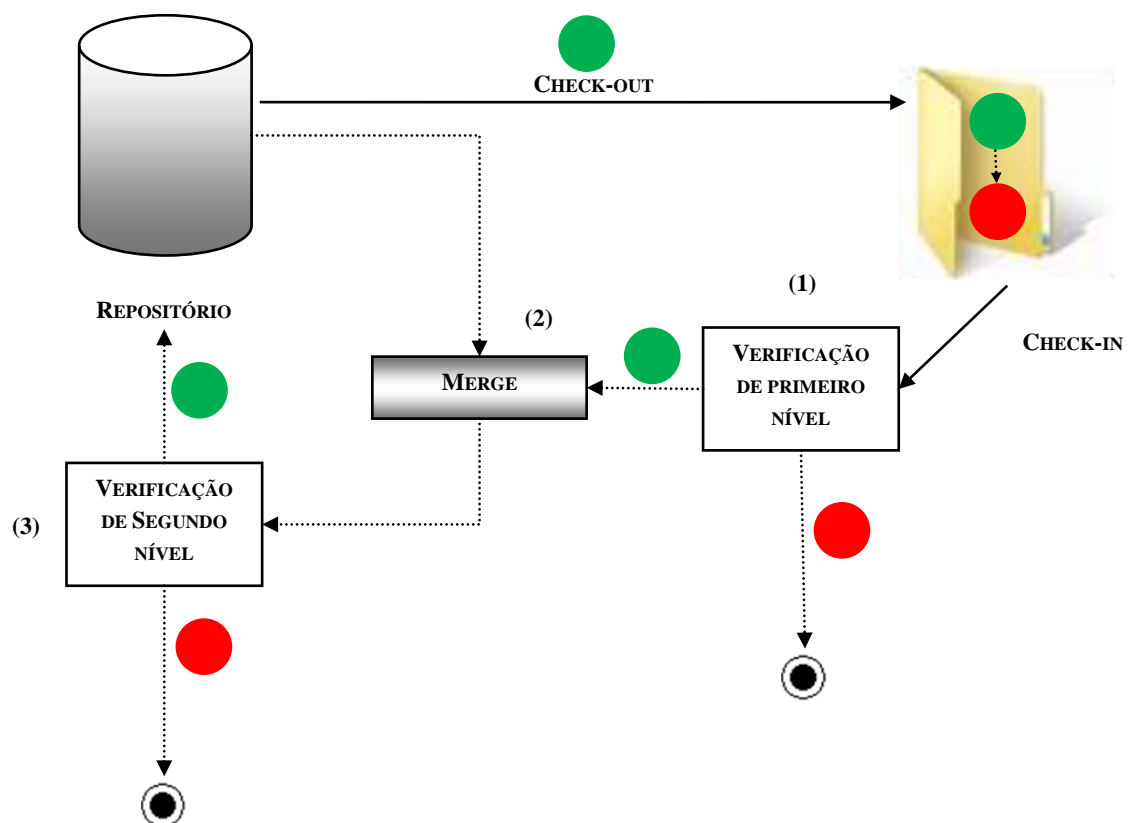


Figura 21. Ciclo executado pela abordagem Ouriço.

No ciclo da abordagem Ouriço é possível identificar que os artefatos passam por duas verificações, sendo uma de primeiro nível e outra de segundo nível. A verificação de primeiro nível ocorre na configuração que é denominada configuração do desenvolvedor. Esse nome é dado justamente por se tratar da configuração que o desenvolvedor envia para o repositório. Já a verificação de segundo nível atua sobre a configuração candidata. Essa configuração recebe esse nome, pois essa configuração é o resultado da junção da configuração do desenvolvedor com a configuração corrente do repositório. Isto é, a configuração resultante no repositório após o *check-in* ser concluído com sucesso.

As verificações realizadas sobre os artefatos de software podem ser configuráveis para que linhas de desenvolvimentos distintas possam receber tratamentos diferenciados (IEEE 2005). O Ouriço é composto basicamente por três políticas: a permissiva (política menos severa), a moderada e a restritiva (política mais severa). Cada linha de desenvolvimento pode receber um tratamento diferenciado, através da escolha de políticas diferentes, por motivos como: restrição de prazo (i.e., tempo máximo que uma tarefa pode ser executada); existência ou não de testes, entre outros. Um cenário de utilização destas políticas é o caso em que os ramos recebem um tratamento menos severo, enquanto a linha principal recebe um tratamento mais severo, por ser o local onde a maioria dos desenvolvedores realizam contribuições e, portanto, necessita maior grau de confiança nos artefatos.

As políticas utilizadas pelo Ouriço são compostas por filtros que possibilitam a detecção de problemas específicos nos artefatos de software. Esta abordagem é composta por três filtros: (1) físico, (2) sintático e (3) semântico. Estes filtros são chamados por políticas quando a execução de alguma dessas verificações é necessária para a política em questão. Os filtros que compõem cada política são descritos com maiores detalhes na Seção 4.3, com exemplos de tipos de problemas que cada filtro é capaz de identificar.

A abordagem Ouriço tem o objetivo de ser não intrusiva, portanto os desenvolvedores continuam tendo a mesma percepção do ciclo de GC, que também é composto por *check-out*, modificações e *check-in*. Além disso, o Ouriço trata o comando de *update*, que é utilizado em situações onde é desejável obter novidades do repositório (Figura 22). Durante a execução do *check-out* o Ouriço realiza tarefas como criação de *autobranches* (i.e., camada que tem o objetivo de proteger o repositório e viabilizar verificações) e, se necessário, verificações são executadas nos artefatos que estão sendo enviados ao desenvolvedor. Essas verificações do *check-out* são realizadas com o objetivo de calibrar a política dinâmica, discutida na subseção 4.4.4. Durante a execução do *check-in* o Ouriço realiza uma verificação mais detalhada sobre

a configuração candidata, que será integrada ao repositório, caso as verificações forem executadas com sucesso.

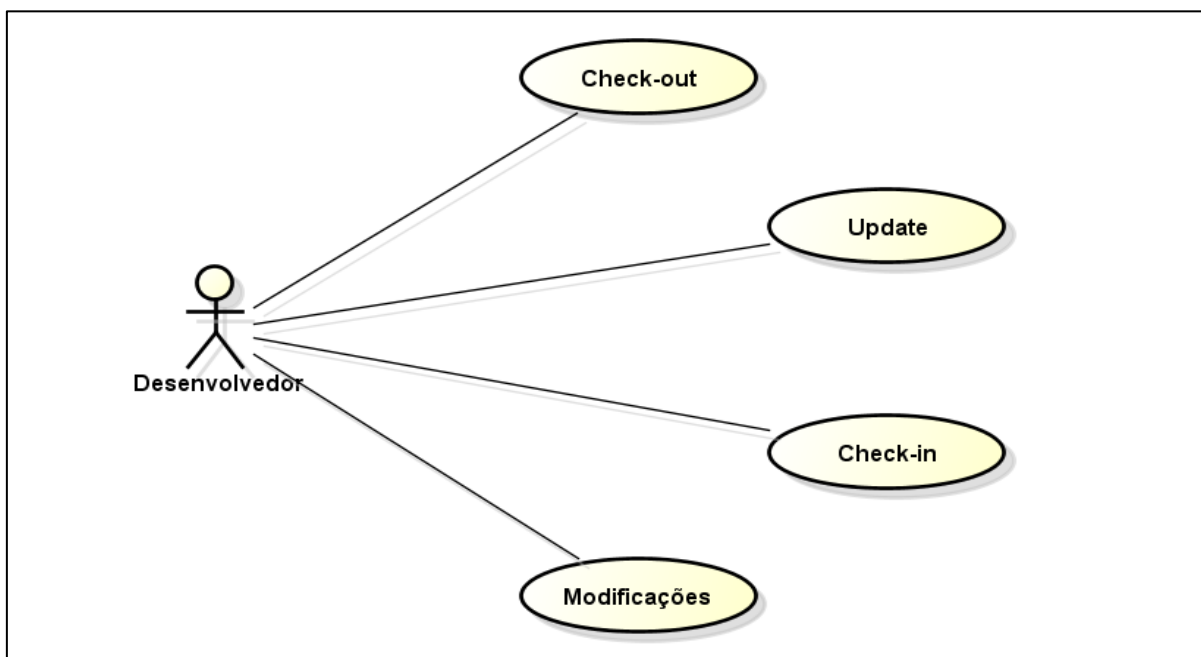


Figura 22. Atividades e comandos que os desenvolvedores executam durante o ciclo de desenvolvimento com GC.

Portanto, esta abordagem apresenta como objetivo ampliar o ciclo tradicional de GC com tarefas automáticas e incrementais para verificar quebras sintáticas e/ou semânticas em artefatos de software e, conseqüentemente, contribuir para a manutenção de um repositório consistente e confiável.

No restante deste capítulo os conceitos, até então introduzidos, são descritos com maiores detalhes visando dar um melhor entendimento da diferença do ciclo tradicional de GC para o ciclo de trabalho do Ouriço. Esse ciclo incorpora verificações e outras tarefas inexistentes no ciclo tradicional de GC.

4.3 FILTROS

A abordagem Ouriço executa verificações para identificar problemas específicos em artefatos de software. Conforme foi discutido no Capítulo 3, os dois problemas principais que podem ocorrer em repositórios de GC são as quebras sintática e semântica. Além disso, conflitos físicos podem também ocorrer, devido à edição em paralelo da mesma área de um artefato de software. Estes problemas podem ser identificados por meio dos filtros físico, sintático e semântico, que são descritos nesta seção.

O filtro físico é usado para identificar conflitos físicos. O filtro físico pode identificar possíveis conflitos que ocorrem entre a configuração do desenvolvedor, que está sofrendo

check-in, e a configuração corrente do repositório de GC. Este conflito é identificado quando mais de um desenvolvedor edita a mesma região de um artefato de software e tentam enviar estas configurações para o repositório. Para identificar tais conflitos, esta abordagem utiliza as ferramentas dos SCV, como a de junção, para identificar tais conflitos e, posteriormente, reportá-los ao desenvolvedor.

O filtro sintático é utilizado para identificar quebras sintáticas. Tal filtro é capaz de identificar situações onde uma configuração não compila, para isso são utilizados os SGC. O resultado deste filtro, em caso de sucesso ou falha, é registrado de forma que possa ser recuperado e utilizado posteriormente para, por exemplo, reportá-lo ao desenvolvedor. Exemplos deste cenário são ilustrados na Seção 4.7, pela utilização de outros conceitos que são abordados no restante deste capítulo.

O filtro semântico é utilizado para identificar quebras semânticas. O filtro semântico pode identificar, por exemplo, situações onde as regras de negócio não são implementadas corretamente, resultando em casos de testes quebrados. Para identificar quebras em regras de negócio, o filtro semântico executa testes sobre a configuração que está sendo verificada, através de um SGC. O resultado da aplicação do filtro semântico, em caso de sucesso ou falha, é registrado de forma que possa ser recuperado e utilizado posteriormente. Exemplos deste cenário são ilustrados na Seção 4.7.

O objetivo desses filtros é identificar problemas (i.e., conflitos físicos e quebras sintáticas ou semânticas) que posteriormente serão enviados aos desenvolvedores, por meio de uma notificação. Esta notificação é composta pela natureza da quebra (i.e., física, sintática ou semântica) e usa descrição (e.g., *class not found*).

Filtros podem ser classificados em duas categorias, bloqueante ou informativo, dependendo do seu comportamento quando um problema é encontrado. Os Filtros bloqueantes param o ciclo de trabalho quando algum problema é encontrado e, deste modo, evita que o repositório fique em estado inconsistente. Este tipo de filtro pode ser aplicado quando artefatos sintaticamente quebrados não são permitidos no repositório, por exemplo.

Por outro lado, os filtros informativos não param o ciclo de trabalho quando algum problema é identificado. Estes filtros são utilizados quando artefatos são aceitos no repositório de GC independente de estarem ou não quebrados, mas o desenvolvedor deseja ter ciência sobre a situação do repositório ao final do *check-in*. Portanto, quando algum problema é encontrado, o ciclo de trabalho prossegue, mas o desenvolvedor será avisado que existe algum artefato quebrado no repositório.

4.4 POLÍTICAS

O Ouriço tem como um dos objetivos realizar verificações em diferentes níveis de controle para diferentes compartimentos lógicos do repositório (IEEE 2005). Para realizar essas verificações, o uso de políticas foi adotado. Estas políticas foram classificadas como: permissiva, moderada, restritiva e dinâmica. Além disso, essas políticas são compostas por filtros bloqueantes e, com o objetivo de prover maior informação sobre os artefatos de software presentes no repositório, podem ser estendidas por filtros informativos.

4.4.1 POLÍTICA PERMISSIVA

A política permissiva possui um nível de controle menos severo. Tal política aplica um ciclo de verificação muito similar ao aplicado pelos SCV tradicionais. Por default, a política permissiva identifica apenas conflitos físicos, via execução do filtro físico bloqueante discutido anteriormente. Portanto, somente artefatos que não possuem conflitos físicos serão aceitos na linha de desenvolvimento do repositório controlado por esta política. No caso de alguma configuração apresentar tal problema, o desenvolvedor será alertado sobre a sua causa e a configuração será rejeitada. Após corrigi-lo, o desenvolvedor poderá realizar *check-in* desta configuração para o repositório e a configuração será rejeitada ou integrada ao repositório.

Para viabilizar a identificação de outros problemas sem finalizar o ciclo do Ouriço, a política permissiva pode ser estendida por filtros informativos, sintático e semântico. Tais filtros darão uma percepção da situação do repositório mesmo que este permita a existência de artefatos quebrados. Deste modo, uma maior confiança poderá ser depositada à linha de desenvolvimento do repositório quando nenhum problema de natureza sintática ou semântica for identificado por esses filtros, supondo que o projeto tenha testes.

Esta política pode ser utilizada por equipes que estejam começando a utilizar o Ouriço e, também, em casos onde não sejam necessárias ou possíveis as verificações sintática e semântica. Para utilizar esta abordagem, é desejável que o projeto possua testes para viabilizar a verificação das regras de negócio e possibilitar a identificação de quebras semânticas. Entretanto, é possível que para determinadas equipes a produção de casos de teste e a manutenção dos artefatos compilando ainda sejam tarefas negligenciadas. Portanto, esta política pode ser utilizada para que a equipe vá se adequando gradativamente à abordagem proposta e posteriormente desfrutar das vantagens que o Ouriço pode oferecer, através de políticas mais poderosas.

4.4.2 POLÍTICA MODERADA

A política moderada apresenta um nível de controle superior à política permissiva. Esta política aplica um ciclo de verificação que já estende as verificações realizadas pelos SCV tradicionais. Isto é, por padrão, a política moderada pode identificar conflitos físicos e, também, quebras sintáticas através dos filtros físico e sintático do tipo bloqueante. Deste modo, somente artefatos que não resultem em conflitos físicos e podem ser compilados serão aceitos na linha de desenvolvimento do repositório protegida por essa política. No caso que algum artefato da configuração em questão apresente algum problema relacionado a esses filtros, o desenvolvedor será alertado sobre a causa do mesmo e a configuração será rejeitada. Após corrigi-lo, o desenvolvedor poderá realizar *check-in* desta configuração para o repositório. Tal configuração será rejeitada ou integrada ao repositório.

A política moderada pode ser estendida com a utilização do filtro informativo para identificação de quebras semântica. Este filtro dará uma percepção da situação do repositório, mesmo que seja permitida a existência de artefatos semanticamente quebrados. Deste modo, uma maior confiança poderá ser depositada à linha de desenvolvimento do repositório, quando nenhum problema de natureza semântica for identificado por esse filtro, supondo que o projeto tenha testes.

Esta política pode ser utilizada por equipes que não tenham necessidade ou estejam impossibilitadas de executar a verificação semântica. Como ressaltado na política anterior, para utilizar o Ouriço é desejável que o projeto possua casos de testes que garantam que as regras de negócio do sistema estejam sendo seguidas. Entretanto, muitas equipes não possuem tais testes. Embora esta política ainda não seja a mais severa, ela apresenta maior proteção para a linha de desenvolvimento que está sendo protegida e, deste modo, pode-se garantir ainda mais confiança aos artefatos presentes nela.

4.4.3 POLÍTICA RESTRITIVA

A política restritiva apresenta o nível de controle mais severo desta abordagem. Essa política aplica uma verificação que estende as verificações realizadas pelos SCV tradicionais. Por padrão, a política restritiva pode identificar conflitos físicos e, também, quebras através dos filtros sintático e semântico do tipo bloqueante. Deste modo, somente artefatos que não resultem em conflitos físicos, sejam compiláveis e passem pelos testes serão aceitos na linha de desenvolvimento do repositório que é protegida por essa política. No caso em que algum artefato da configuração apresentar qualquer problema relacionado a esses filtros, o desenvolvedor será alertado sobre a sua causa e a configuração será rejeitada. Ao ser alertado

o desenvolvedor poderá corrigir os problemas relatados e enviar a configuração para ser novamente verificada e, possivelmente, integrada.

A política restritiva utiliza todos os filtros que a abordagem dispõe e, portanto não pode ser estendida. Para o uso desta política é necessário que o projeto tenha casos de teste que cubram suas regras de negócio. Além disso, esta política exige maior maturidade da equipe de desenvolvimento que a utiliza, dado que é necessária uma prática de criação de testes paralela ao desenvolvimento de novas funcionalidades. Com a criação de testes, durante o comando de *check-in* serão verificadas características sintáticas e semânticas, resultando em artefatos mais confiáveis. Essa política garante maior confiança em artefatos presentes em linhas de desenvolvimento protegidas pela abordagem Ouriço. Com a execução de verificações física, sintática e semântica, esta política garante que um desenvolvedor realize *check-out* de uma linha de desenvolvimento protegida e possa executar suas tarefas normalmente sem se preocupar com quebras. Quebras essas que em alguns casos precisavam ser retiradas antes de iniciar tarefas evolutivas do projeto, conforme discutido no Capítulo 3.

4.4.4 POLÍTICA DINÂMICA

A política dinâmica configurada de acordo com o estado da configuração obtida do repositório. Durando o processo de *check-out*, detalhado na Seção 4.6.1, a configuração que está sendo obtida do repositório é verificada sintaticamente e semanticamente para que esta política seja calibrada conforme uma das políticas anteriores. A principal diferença desta política para as anteriores é que o desenvolvedor deverá devolver os artefatos de software da maneira que recebeu ou ainda mais consistentes.

O processo de verificação, realizado durante o *check-out*, é composto por verificações sintáticas e semânticas, que definirão qual política será escolhida. Se a configuração não passar por nenhuma das duas verificações executadas, a política será calibrada conforme a política permissiva. Como já discutido, a política permissiva aceita contribuições com quebras sintáticas e semânticas na linha de desenvolvimentos que está sendo protegida pela abordagem. Se a configuração passar apenas pelo filtro sintático, a política será calibrada conforme a política moderada, que aceita contribuições que não passam pelos testes. Finalmente, se a configuração passar pelos filtros sintático e semântico, a política será calibrada conforme a política restritiva, que verifica se uma configuração está sintaticamente ou semanticamente quebrada, rejeitando-a no caso em que alguma verificação falhe.

A política dinâmica é interessante para que a confiança em uma linha de desenvolvimento cresça, de acordo com a maturidade que a equipe vai adquirindo. Esta

política exige que uma contribuição passe pelos mesmos filtros de quando foi obtida. Entretanto, podem existir casos em que a configuração seja enviada corrigindo defeitos encontrados anteriormente e, conseqüentemente, em um próximo *check-out* passe por um filtro que não passava no *check-out* anterior. Portanto, com a utilização desta política, a confiança pode crescer até chegar ao nível de confiança da política restritiva, que verifica os artefatos sintaticamente e semanticamente.

4.5 AUTOBRANCHES

Conforme discutido anteriormente, o repositório de GC é dividido nos seguintes compartimentos: *mainline*, *branches* e *tags*. Esta divisão auxilia a manutenção de um repositório de modo mais organizado, facilitando o desenvolvedor a encontrar artefatos de software e *releases*. Portanto, essa organização também pode auxiliar no processo de desenvolvimento de software.

Para suportar verificações de artefatos de software, o Ouriço propõe uma nova divisão lógica composta pelos seguintes compartimentos: *mainline*, *branches*, *tags* e *autobranches*. O compartimento dos *autobranches* evita que artefatos quebrados cheguem à linha de desenvolvimento protegida pela abordagem Ouriço. *Autobranches* são ramos protegidos que são criados de maneira automática quando um *check-out* é realizado e permitem que alterações sejam realizadas sem serem refletidas nos artefatos originais. A criação de *autobranches* é uma tarefa barata. Por exemplo, no Subversion a criação de um ramo, forma que é implementada o *autobranch*, é realizada com tempo constante, pois não existe a necessidade de duplicar os arquivos e diretórios presentes no repositório (Collins-Sussman et al. 2004). Deste modo, os *autobranches* atuam protegendo as linhas de desenvolvimento de artefatos quebrados até que sejam verificadas e integradas ao repositório.

Autobranches que passarem pelas verificações de uma política do Ouriço são integrados à linha de desenvolvimento que os originou. Após as verificações serem executadas com sucesso em uma configuração, ela estará apta a ser integrada ao seu local de origem com maior confiança. Caso alguma das verificações falhe, a configuração que está sendo enviada ao repositório será rejeitada e a integração ao repositório será interrompida. O processo de integração será discutido com maiores detalhes na Seção 4.6.2.

Uma alternativa a utilização dos *autobranches* pode ser a utilização do mecanismo de extensão presente na maioria dos SCV, conhecidos como gatilho *pré-commit* (discutido no Capítulo 2). Esses gatilhos atuam executando tarefas antes que o *check-in* seja finalizado e, deste modo, realizam verificações em artefatos de software. Esses gatilhos podem ser

utilizados em SCV como, por exemplo, CVS (Fogel 2003), Subversion (Collins-Sussman et al. 2004) e Git (Chacon 2009). No entanto, algumas limitações podem ser destacadas e levadas em conta nessa decisão. A primeira limita o tempo que uma tarefa gasta para ser executada, devido ao risco de acontecer um *time-out* na conexão, quando a tarefa é longa. Adicionalmente, em alguns SCV o gatilho executado antes do *check-in* possui a habilidade apenas de ler os artefatos de software, mas não pode modificar a transação (Berlin and Rooney 2006).

Portanto, verificações que gastam muito tempo para serem executadas e precisam alterar transações não são suportadas por tal mecanismo. Quando uma configuração apresenta artefatos quebrados, o gatilho *pre-commit* não poderá alterar a transação e nem evitar que artefatos quebrados cheguem ao repositório. Por isso, o Ouriço adota os *autobranches* para substituir os gatilhos.

4.6 CICLO DE TRABALHO DO OURIÇO

Neste capítulo foram discutidos alguns conceitos que são utilizados no ciclo de trabalho do Ouriço que, basicamente, é composto pelos comandos de *check-out*, *check-in* e *update*. Tal ciclo acompanha o trabalho dos desenvolvedores que farão modificação em uma configuração, que posteriormente será enviada para o repositório de GC, e, com a utilização da abordagem Ouriço será verificada segundo alguma das políticas do Ouriço.

Durante o ciclo de trabalho do Ouriço algumas verificações serão realizadas, de acordo com a política escolhida, nos artefatos manipulados por esta abordagem. Quando o comando de *check-out* é executado, tarefas como criação de *autobranches*, povoamento dos *autobranches* e verificações serão executadas para manter o repositório confiável e íntegro. Posteriormente, no comando de *check-in*, os artefatos passarão por verificações mais criteriosas, de primeiro e segundo nível, e tarefas para integração dos artefatos ao repositório. Essas verificações e a integração são executadas para manter o conteúdo do repositório consistente. Finalmente, no comando de *update* serão executadas tarefas para que modificações presentes no repositório, mas que ainda não foram integradas ao espaço de trabalho, sejam integradas e disponibilizadas aos desenvolvedores. Além disso, o comando de *update* pode ser utilizado para solucionar problemas identificados pelo Ouriço.

No restante desta seção são abordados, com mais detalhes, os comandos que compõe esta abordagem: *check-out* (Seção 4.6.1), *check-in* (Seção 4.6.1) e *update* (Seção 4.6.3), passando por criação de *autobranches* e, também, verificações feitas nos artefatos enviados e retirados de um repositório.

4.6.1 CHECK-OUT

O comando de *check-out* é tradicionalmente utilizado para obter uma configuração de um repositório de GC. Quando um *check-out* tradicional é executado, uma cópia de uma configuração do repositório é enviada para o desenvolvedor. Este poderá modificá-la e enviar de volta ao repositório uma configuração que pode conter ou não artefatos quebrados. No caso em que a configuração tem artefatos quebrados, os artefatos entrarão no repositório sem que verificações, sintática e semântica, sejam executadas. Para evitar esse caso, o Ouriço executa tarefas que estendem esse comando e, deste modo, oferece mecanismos que viabilizam a verificação de artefatos de software antes que estes sejam integrados ao repositório. O que pode tornar o repositório inconsistente e gerar retrabalho.

A execução do *check-out* é feita em duas etapas (Figura 23): (1) criação de um *autobranch* baseado na configuração que o desenvolvedor requisitar e (2) *check-out* do *autobranch* criado para o espaço de trabalho do desenvolvedor. Além disso, quando necessário, o *check-out* também realiza verificações (3) baseadas na política escolhida ou ainda calibra a configuração da política dinâmica, quando esta política é previamente selecionada. O passo 3 é executado em todas as políticas para que no momento do *check-in* as verificações sejam realizadas em menos tempo, devido às habilidades do SGC.

A criação do *autobranch* consiste na criação de um ramo com a configuração que o desenvolvedor requisitou. O *autobranch* sofrerá alterações, que refletem modificações realizadas no espaço de trabalho, sem que estas atinjam os artefatos originais e torne o repositório inconsistente. Para a criação de um *autobranch* o Ouriço identifica o próximo *autobranch* válido (e.g., através de uma sequência de números inteiros) e o cria baseado na configuração alvo do desenvolvedor.

Em seguida, uma cópia deste *autobranch* é enviada ao espaço de trabalho do desenvolvedor. Quando o desenvolvedor recebe esta configuração, pode continuar o seu trabalho normalmente, como na abordagem tradicional, realizando as modificações necessárias para completar sua tarefa. Vale ressaltar que a criação do *autobranch* é imprescindível para que as verificações do *check-in* ocorram antes que os artefatos atinjam a linha de desenvolvimento protegida pelo Ouriço.

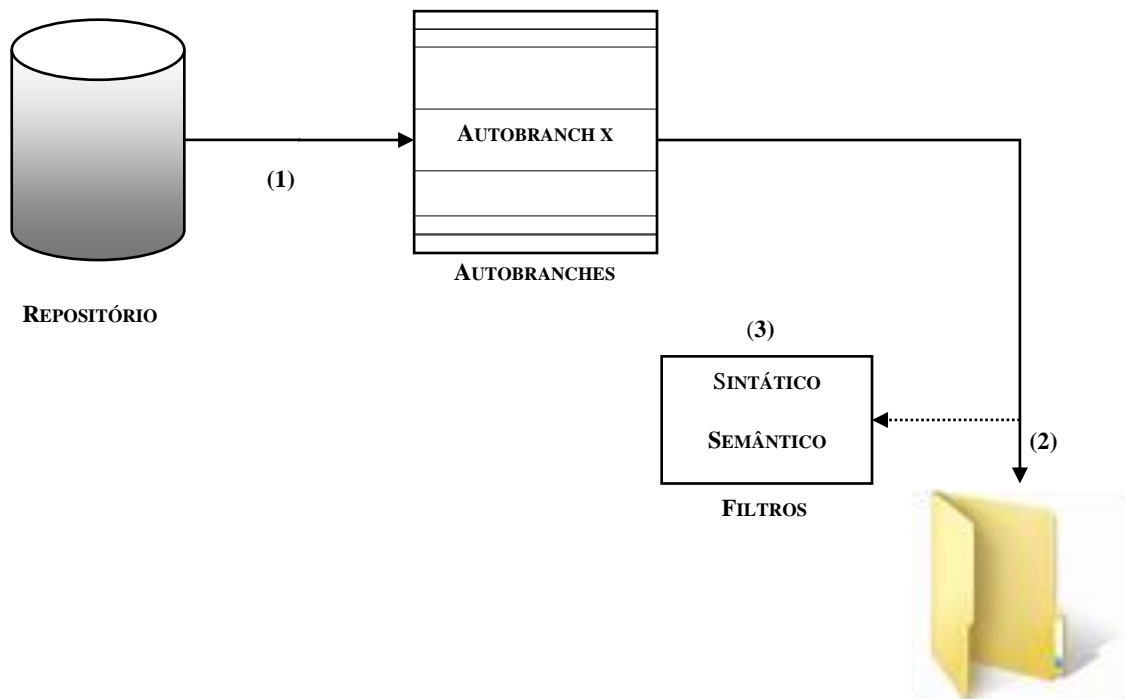


Figura 23. Comando de *check-out* do Ouriço.

Ainda durante o comando de *check-out*, uma verificação é realizada para calibrar a política dinâmica. Conforme discutido anteriormente, a política dinâmica não possui filtros pré-definidos, devido ao seu objetivo de garantir que o desenvolvedor possa enviar artefatos para o repositório, conforme foram obtidos. Tal política é calibrada durante o processamento do comando de *check-out*. Para definir os filtros que serão utilizados os seguintes passos são executados:

- Aplicação do filtro sintático sobre a configuração desejada.
- Aplicação do filtro semântico sobre a configuração desejada.
- Se a configuração não passar por nenhum filtro, calibrar política conforme a política permissiva.
- Se a configuração passar apenas pelo filtro sintático, calibrar a política conforme a política moderada.
- Se a configuração passar pelos filtros sintático e semântico, calibrar a política conforme a política restritiva.

Além de calibrar a política dinâmica, as verificações realizadas no *check-out* aceleram a verificação de artefatos durante o comando de *check-in*. Isto é possível graças à habilidade dos SGC de aproveitar artefatos que já foram construídos anteriormente, pois as verificações serão sempre realizadas no mesmo espaço de trabalho. Deste modo, durante o *check-in* serão

construídos somente os artefatos que foram modificados pelo usuário e não toda a configuração. Além disso, outro ponto relevante é que as dependências também serão obtidas anteriormente, durante o comando de *check-out*, gerando ainda mais eficiência durante o *check-in*. Com relação à verificação do *check-out*, é importante ressaltar que serão aplicados apenas os filtros bloqueantes propostos para a política adotada, excetuando o filtro físico. Logo, para o caso da política moderada, apenas o filtro sintático será aplicado, ou seja, a verificação do *check-out* será composta apenas por compilação.

Conforme discutido anteriormente, a abordagem proposta visa ser não intrusiva, portanto mantém o comando de *check-out* sem alterações, levando em conta a visão do desenvolvedor. Da ótica do desenvolvedor o comportamento externo do comando de *check-out* é mantido para conservar a familiaridade dos desenvolvedores com esse tipo de sistema. Deste modo, é esperada a redução de possíveis resistências que possam surgir devido a alguma alteração no ciclo de trabalho dos desenvolvedores.

4.6.2 CHECK-IN

O comando de *check-in* é tradicionalmente executado para enviar modificações feitas no espaço de trabalho do desenvolvedor para o repositório de GC. Durante a execução do comando de *check-in* tradicional, as contribuições enviadas pelo desenvolvedor não são sintaticamente ou semanticamente verificadas. Deste modo, artefatos defeituosos podem entrar no repositório tornando-o inconsistente.

A não verificação dos artefatos pode levar o repositório a um estado inconsistente e dar origem a retrabalho. Para evitar esses efeitos colaterais o comando de *check-in* do Ouriço foi criado. Tal comando é composto basicamente por 5 passos que realizam verificações, junções e integração de artefatos ao repositório. Esses passos são os seguintes (Figura 24):

1. *Check-in* tradicional;
2. Verificação de primeiro nível;
3. Obtenção da configuração candidata;
4. Verificação de segundo nível;
5. Integração da contribuição do desenvolvedor ao repositório.

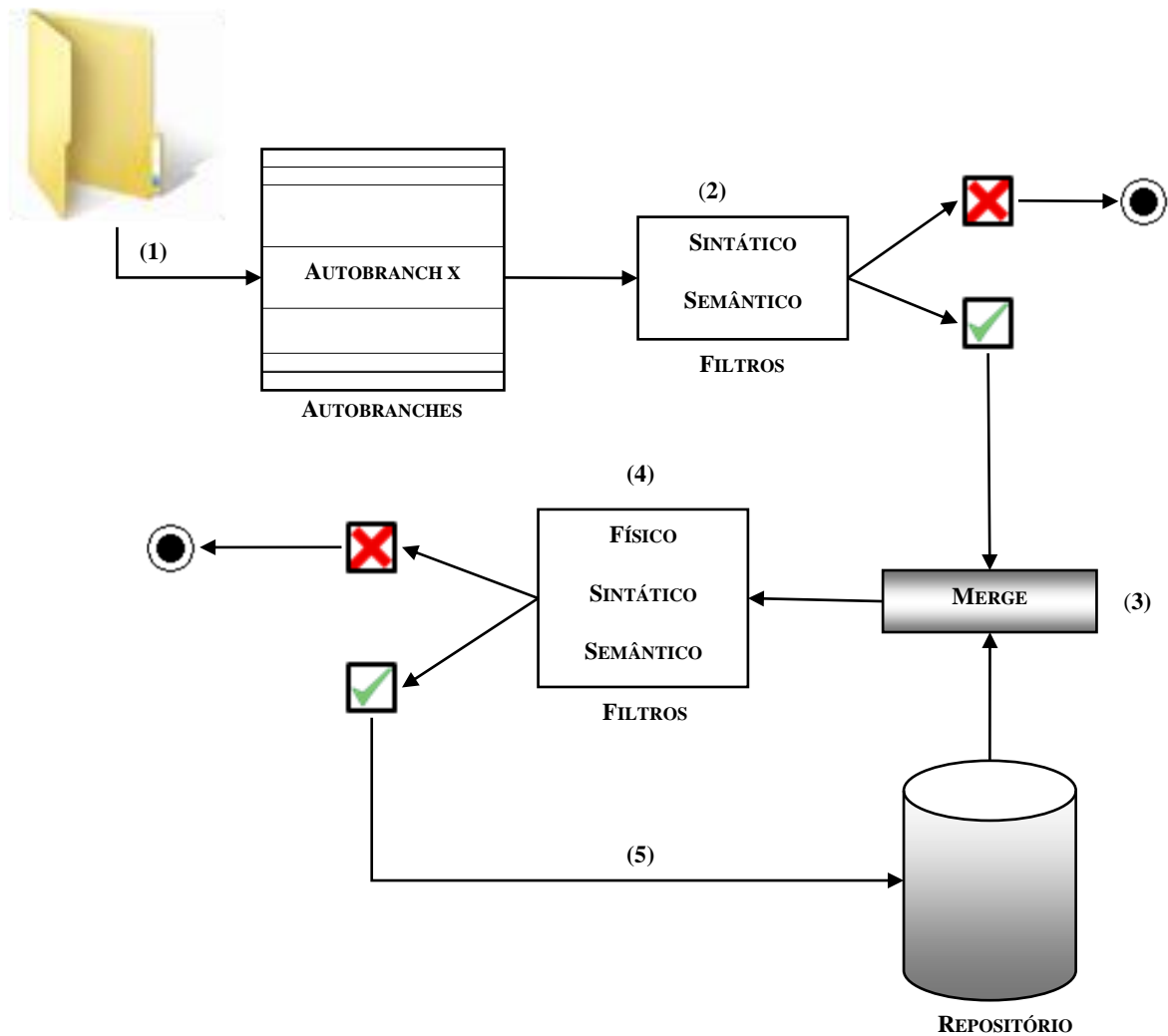


Figura 24. Comando de *check-in* do Ouriço.

O comando de *check-in* pode ser interrompido se qualquer irregularidade for identificada. Conforme discutido anteriormente, esta abordagem visa encontrar irregularidades em artefatos de software, impedi-los de chegar ao repositório de GC e afetar os artefatos originais. Portanto, quando um defeito é encontrado o comando de *check-in* é imediatamente interrompido, em qualquer passo, e uma notificação explicando o problema identificado e o tipo (i.e., conflito físico, quebra sintática ou quebra semântica) é enviada ao desenvolvedor.

A verificação de primeiro nível é aplicada sobre a configuração do desenvolvedor, de acordo com a política previamente definida para proteger o repositório. A verificação de primeiro nível pode identificar artefatos sintaticamente e semanticamente quebrados. Por isso, esse passo aplica o filtro sintático quando a política moderada ou restritiva é adotada. No caso da política dinâmica, o filtro sintático só será aplicado se tal política for previamente calibrada para executar esse filtro. Além disso, a verificação de primeiro nível aplica o filtro semântico

para o caso em que a política restritiva ou dinâmica (calibrada com o filtro semântico) for adotada.

Outro passo que vale ressaltar é a obtenção da configuração candidata. Este passo cria a configuração candidata que é uma projeção da configuração final após a integração da contribuição do desenvolvedor no repositório. Para obter a configuração candidata é feita uma junção entre a configuração do desenvolvedor e a configuração corrente do repositório. A configuração resultante é chamada de configuração candidata que será verificada posteriormente, visando garantir a integridade do repositório.

De posse da configuração candidata, a verificação de segundo nível pode identificar conflitos que ocorreram durante a junção, caracterizando um conflito físico. Esta verificação é executada em qualquer uma das políticas descritas por essa abordagem. Além da verificação física, para as políticas moderada, restritiva e dinâmica (calibrada com o filtro sintático), a verificação de segundo nível aplica o filtro sintático, identificando, deste modo, artefatos quebrados que seriam incluídos no repositório, mas que, no entanto, não compilavam. Finalmente, a verificação de segundo nível pode também identificar artefatos semanticamente quebrados na configuração candidata para as políticas restritiva e dinâmica (calibrada com o filtro semântico).

Quando as verificações de primeiro e segundo nível são executadas com sucesso, a configuração do desenvolvedor é integrada (passo 5) pois está apta a entrar no repositório sem deixá-lo inconsistente. Esta integração segue os seguintes passos:

1. *Check-out* da configuração corrente do repositório para um espaço de trabalho;
2. Reflexão das alterações realizadas no *autobranch* para o espaço de trabalho criado no passo anterior; e
3. *Check-in* do espaço de trabalho resultante.

Além de se preocupar com a consistência do repositório, tal abordagem também se preocupa com o tempo que o desenvolvedor pode ficar ocioso esperando por verificações que, em alguns casos, podem ser demoradas e executadas de maneira automática. As verificações realizadas por esta abordagem serão executadas de maneira assíncrona em relação ao desenvolvedor. Com as verificações sendo executadas de maneira assíncrona, o desenvolvedor fica livre para se dedicar a outras tarefas, enquanto o Ouriço executa as tarefas de verificação, que em alguns casos podem ser longas e desgastantes para o desenvolvedor. Um exemplo de verificação demorada é a compilação do OpenOffice Ximian que leva aproximadamente 11 horas (Linux Reviews 2011).

Como a verificação é realizada de maneira assíncrona, o Ouriço disponibiliza um painel em que o desenvolvedor pode acompanhar o andamento das verificações. Por essa característica assíncrona, nenhuma interação direta acontece entre o desenvolvedor e o Ouriço durante o ciclo de verificação, exceto em caso de falha. Nesse painel é possível acompanhar o desenvolvimento de cada *autobranch* com relação ao *check-out*, verificação de primeiro nível, verificação de segundo nível e integração ao repositório. Além disso, o painel oferece o detalhamento das falhas que ocorreram durante o ciclo de trabalho do Ouriço para auxiliar na identificação dos defeitos.

4.6.3 UPDATE

O comando de *update* é utilizado tradicionalmente para trazer contribuições que estão no repositório, mas que ainda não foram incorporadas no espaço de trabalho do desenvolvedor. Este comando é tradicionalmente executado quando um desenvolvedor quer integrar as contribuições feitas por outros desenvolvedores no seu espaço de trabalho.

O comando de *update* tradicional seguiria os seguintes passos se aplicado no Ouriço (Figura 25): (1) análise do *autobranch* para detectar as modificações presentes no repositório, mas que ainda não foram incorporadas ao espaço de trabalho, e (2) integração das modificações no espaço de trabalho do desenvolvedor. Este comando resultaria na incorporação de todas as modificações realizadas no *autobranch* para o espaço de trabalho do desenvolvedor. No entanto, este não seria o resultado desejado.

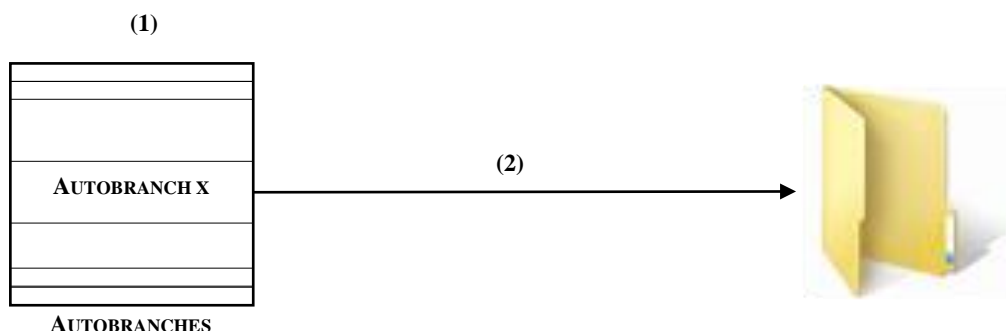


Figura 25. Aplicação do *update* tradicional na abordagem Ouriço.

Conforme visto anteriormente, o comando de *check-out* cria um *autobranch* e realiza *check-out* deste *autobranch*, que é de uso exclusivo do desenvolvedor que realizou *check-out*. Portanto, quando um *update* tradicional for executado nenhuma das modificações realizadas pelos outros desenvolvedores será incorporada ao espaço de trabalho que ele está utilizando e o resultado será o antigo espaço de trabalho.

Deste modo, o comando de *update* do Ouriço foi reestruturado para obter as informações do repositório original, deixando o *autobranch* somente para receber as alterações executadas no espaço de trabalho. O comando de *update* do Ouriço é composto pelos seguintes passos (Figura 26): (1) análise da linha de desenvolvimento que originou o *autobranch* para identificar as modificações e (2) reflexão destas modificações no espaço de trabalho do desenvolvedor.



Figura 26. Update da abordagem Ouriço.

Durante o comando de *check-out*, o Ouriço guarda uma relação entre a linha de desenvolvimento que originou *autobranch* e o próprio *autobranch*. Deste modo, é possível identificar qual linha de desenvolvimento originou os artefatos de software, que o desenvolvedor possui em seu espaço de trabalho. Assim sendo, no momento que o Ouriço for obter as modificações, que serão incorporadas no espaço de trabalho, a linha de desenvolvimento que deu origem aos artefatos do espaço de trabalho do desenvolvedor será utilizada como referência.

O comando de *update* pode ser também entendido como a junção de dois comandos: *diff* e *apply*. O primeiro comando visa identificar diferenças entre duas configurações. No caso do *update* do Ouriço o comando de *diff* identifica artefatos ou regiões de artefatos que estão no repositório, mas diferem da configuração presente no espaço de trabalho do desenvolvedor. O segundo comando aplica as modificações identificadas no comando *diff* sobre uma configuração. No caso do *update* descrito por esta abordagem, o destino é o espaço de trabalho do desenvolvedor. Portanto, após a execução do *update* do Ouriço, o espaço de trabalho receberá as modificações realizadas no repositório, mas que ainda não estavam disponíveis para os desenvolvedores.

Conforme discutido na Seção 4.6.2, o Ouriço identifica problemas durante a execução do comando de *check-in* e quando algum problema é identificado, a configuração deve ser corrigida. Os problemas podem ser identificados na configuração do desenvolvedor, via

verificação de primeiro nível, ou na configuração candidata, via verificação de segundo nível. Cada um desses problemas deve ser tratado de maneira específica (i.e., utilizando ou não o comando de *update*) para o desenvolvedor posteriormente reenviar a configuração final ao repositório. Configuração esta que será novamente verificada pelo comando de *check-in*.

Os problemas que ocorrem na configuração do desenvolvedor são identificados pela verificação de primeiro nível serão resolvidos diretamente no espaço de trabalho do desenvolvedor, sem a necessidade da execução do comando de *update*. Quando um problema é verificado durante a verificação de primeiro nível, uma notificação é enviada ao desenvolvedor identificando o problema e a sua descrição. Por exemplo, uma falha sintática na verificação de primeiro nível será reportada ao desenvolvedor como uma “falha sintática de primeiro nível”, seguida da descrição do problema que pode ser a falta de ponto e vírgula (;) no final de um comando qualquer. Portanto, o desenvolvedor pode colocar o ponto e vírgula e, posteriormente, realizar o *check-in* da configuração que possivelmente será aceita.

Os problemas que ocorrem na configuração candidata são identificados pela verificação de segundo nível e necessitam da execução do comando de *update* para que sejam refletidos no espaço de trabalho e possam ser corrigidos. Quando um problema é identificado somente pela verificação de segundo nível, é sinal que durante a criação da configuração candidata problemas foram gerados, podendo eles ser de origem física, sintática ou semântica. Portanto, é necessário que esses problemas sejam replicados para o espaço de trabalho, local onde o desenvolvedor terá possibilidade de corrigi-los. Após realizar as modificações necessárias para corrigir a configuração candidata, o desenvolvedor poderá executar novamente o *check-in*, que fará a verificação da configuração candidata com as correções incorporadas.

4.7 CENÁRIOS DE UTILIZAÇÃO DO OURIÇO

A abordagem Ouriço é utilizada para capturar artefatos inconsistentes antes que estes atinjam o repositório, através de verificações de primeiro e segundo nível. Portanto vários cenários, originados da aplicação desta abordagem, podem ser discutidos para o melhor entendimento desta abordagem. Nesta seção são discutidos os seguintes cenários: (1) Cenário ideal, onde todas as verificações são executadas com sucesso; (2) Cenário com problema de primeiro nível; (3) Cenário com problema sintático de segundo nível; e, (4) Cenário com problema semântico de segundo nível. Os cenários aqui apresentados são baseados nos exemplos apresentados no Capítulo 3 e são discutidos levando em conta o ciclo de trabalho da abordagem Ouriço utilizando a política restritiva. Vale notar que os cenários são

intencionalmente simples, para fins didáticos. Experimentos da utilização do Ouriço em ambiente real são discutidos posteriormente, no Capítulo 6.

Com o objetivo de facilitar a visualização das classes utilizadas nestes cenários foi construído um diagrama de classes (Figura 27). Nesse diagrama é possível visualizar as classes: *Transformacoes*, responsável por realizar transformações ou conversões de temperaturas, onde $tCelsius$ é a temperatura em Celsius e K é uma constante utilizada para transformar Celsius em Kelvin; *LeiDoGasIdeal*, uma classe responsável por calcular o volume segundo a Lei do Gás Ideal, onde R é a constante dos gases perfeitos, p é a pressão, n é o número de gases em Mol e $tKelvin$ é a temperatura em Kelvin; e, a classe *Testes*, que é um caso de teste que verifica o cálculo do volume conforme a Lei do Gás Ideal.

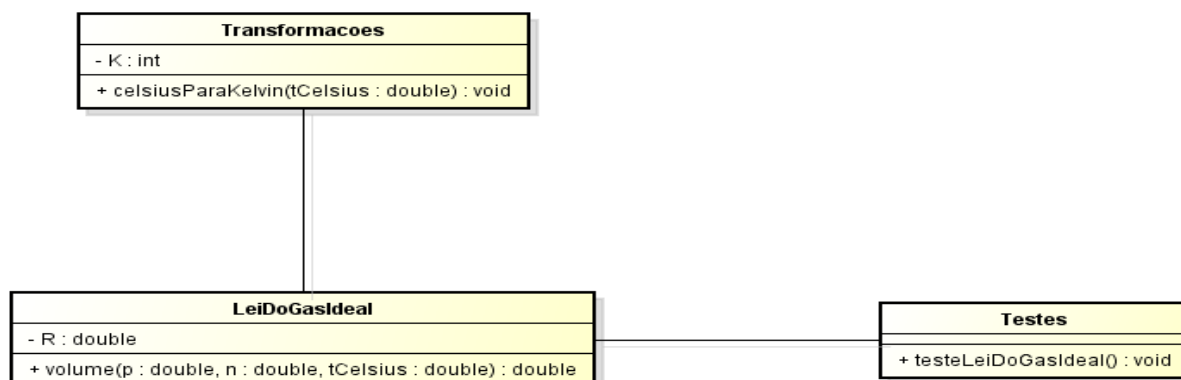


Figura 27. Diagrama de classe da configuração inicial dos cenários.

4.7.1 CENÁRIO IDEAL

No cenário ideal nenhum problema ocorrerá durante a execução de todo o ciclo. Portanto, suponha que João e Maria tenham realizado *check-out* dos artefatos apresentados na Figura 28. Após obter a configuração ilustrada na Figura 28, Maria altera o nome do método que realiza os testes de *gasIdeal* para *testeLeiDoGasIdeal* e realiza *check-in* (Figura 29). Em paralelo as alterações de Maria, João altera o nome do método *celsiusParaKelvin* da classe *Transformacoes* para *transformacao*, refatora a classe *LeiDoGasIdeal* e realiza *check-in* (Figura 30).

Quando Maria realizou *check-in*, a configuração dela passou por verificações de primeiro e segundo nível. Na verificação de primeiro nível foi levada em conta a configuração que ela estava enviando para o repositório. Esta configuração era composta pelos artefatos representados na Figura 28.a, Figura 28.b e Figura 29. Quando esta configuração foi verificada sintaticamente, nenhuma falha de compilação foi identificada, e o mesmo aconteceu para a verificação semântica. Na verificação de segundo nível, foi levada em

consideração a mesma configuração anterior, caso nenhum *check-in* tenha entrado no repositório entre o período de *check-out* e *check-in* de Maria. Portanto, nenhum problema foi encontrado e a configuração de Maria foi integrada ao repositório.

```
1 public class Transformacoes {
2     public static final int K = 273;
3     public static double celsiusParaKelvin(double tCelsius){
4         return tCelsius + K;
5     }
6 }
7 }
```

(a)

```
1 public class LeiDoGasIdeal {
2     public final double R = 8.314472;
3     public double volume(double p, double n, double tCelsius){
4         double tKelvin = Transformacoes.celsiusParaKelvin(tCelsius);
5         return n * R * tKelvin / p;
6     }
7 }
```

(b)

```
1 public class Testes {
2     @Teste
3     public void gasIdeal(){
4         LeiDoGasIdeal leiDoGasIdeal = new LeiDoGasIdeal();
5         double oraculo = 9744.561184;
6         double delta = 0.000001;
7         assertEquals(oraculo, leiDoGasIdeal.volume(25, 100, 20), delta);
8     }
9 }
```

(c)

Figura 28. Espaço de trabalho obtido.

```
1 public class Testes {
2     @Teste
3     public void testeLeiDoGasIdeal(){
4         LeiDoGasIdeal leiDoGasIdeal = new LeiDoGasIdeal();
5         double oraculo = 9744.561184;
6         double delta = 0.000001;
7         assertEquals(oraculo, leiDoGasIdeal.volume(25, 100, 20), delta);
8     }
9 }
```

Figura 29. Maria altera o nome de um método.

```

1 public class Transformacoes {
2     public static final int K = 273;
3     public static double transformacao(double tCelsius){
4         return tCelsius + K;
5     }
6 }
7 }

```

(a)

```

1 public class LeiDoGasIdeal {
2     public final double R = 8.314472;
3     public double volume(double p, double n, double tCelsius){
4         double tKelvin = Transformacoes.transformacao(tCelsius);
5         return n * R * tKelvin / p;
6     }
7 }

```

(b)

Figura 30. João altera o nome de um método (a) e refatora a classe *LeiDoGasIdeal* (b).

Analogamente, a configuração de João passou por verificações de primeiro e segundo nível. A configuração de João que passou pela verificação de primeiro nível era composta pelos artefatos representados na Figura 28.c, Figura 30.a e Figura 30.b, configuração obtida através da junção do comando de *check-in*. Portanto, durante sua verificação, nenhum problema sintático ou semântico pôde ser identificado. Diferentemente do ciclo de Maria, a configuração verificada no segundo nível foi diferente da do primeiro nível, pois contou com a contribuição anterior de Maria, e, neste caso, a verificação de segundo nível foi aplicada sobre a configuração composta pelos artefatos ilustrados na Figura 29, Figura 30.a e Figura 30.b. Durante a verificação de segundo nível, nenhuma irregularidade foi encontrada e a configuração de João foi integrada ao repositório.

4.7.2 CENÁRIO COM PROBLEMA DE PRIMEIRO NÍVEL

No cenário com problemas no primeiro nível, artefatos sintaticamente ou semanticamente quebrados são enviados para o repositório. Entretanto, com a utilização do Ouriço configurado com a política restritiva, tais artefatos serão rejeitados. Para este cenário, suponha que Maria tenha obtido a configuração ilustrada na Figura 28 diretamente do repositório. Em seguida, que ela tenha alterado o nome do método *celsiusParaKelvin* da classe *Transformacoes* para *transformacao* (Figura 31) e tenha realizado *check-in*.

```

1 public class Transformacoes {
2     public static final int K = 273;
3     public static double transformacao(double tCelsius){
4         return tCelsius + K;
5     }
6 }
7 }

```

Figura 31. Maria altera o nome de um método.

Durante a verificação de primeiro nível, a configuração verificada era composta pelos artefatos representados na Figura 28.b, Figura 28.c e Figura 31. Diferentemente do cenário ideal, a configuração deste cenário apresentava um problema de compilação, pois quando o artefato da Figura 28.b for compilado, o método *celsiusParaKelvin* da classe *Transformacoes* não existia mais, devido a alteração de Maria (Figura 31). Portanto, a verificação foi imediatamente interrompida e uma notificação foi enviada ao desenvolvedor. Esta notificação era composta de um assunto (e.g., “Falha na verificação sintática de primeiro nível”) e de uma descrição desta falha (e.g., o método chamado na linha 4 da classe *LeiDoGasIdeal* não existe). Diante dessa informação, é esperado do desenvolvedor que alguma providência seja tomada para a correção do defeito identificado. Após corrigir este defeito o desenvolvedor poderá realizar *check-in* novamente e, se nenhuma outra falha for identificada, a verificação de primeiro nível será concluída com sucesso.

4.7.3 CENÁRIO COM PROBLEMA SINTÁTICO DE SEGUNDO NÍVEL

Outro cenário que pode acontecer é uma quebra na verificação de segundo nível. Uma quebra de segundo nível ocorre quando a configuração candidata possui algum defeito que a impeça de compilar ou passar pelos testes. Para ilustrar essa situação, suponha que Maria e João tenham feito *check-out* da configuração apresentada na Figura 32.

Após realizar *check-out*, João modifica o artefato da Figura 32 alterando o nome do método *celsiusParaKelvin* para *transformacao*, ajusta a referência na classe *Teste*, e, em seguida, realiza *check-in* de sua configuração (Figura 33). Durante o *check-in*, a configuração de João passa pela verificação de primeiro nível, que não identifica nenhum problema sintático ou semântico. Essa configuração passou, pois quando tal configuração passou pelo processo de compilação nenhum problema foi identificado. Além disso, nenhum problema semântico foi identificado, pois o caso de teste *testeCelsiusParaKelvin* foi executado com sucesso, devido a 23° C corresponder a 296 K ($tCelsius + K = 23 + 273 = 296$). Portanto, a verificação de primeiro nível é concluída com sucesso. O mesmo ocorre com a verificação de segundo nível, caso nenhuma contribuição tenha entrado no repositório entre o intervalo de *check-out* e *check-in* de João, devido à configuração de João ser a mesma que a configuração

candidata. Deste modo, a contribuição de João é integrada ao repositório, que está disponível para que outros desenvolvedores possam obter essa configuração e modificá-la de acordo com a tarefa a ser desenvolvida.

```
1 public class Transformacoes {
2     public static final int K = 273;
3     public static double celsiusParaKelvin(double tCelsius){
4         return tCelsius + K;
5     }
6 }
7 }
```

(a)

```
1 public class Testes {
2     @Teste
3     public void testeCelsiusParaKelvin(){
4         double oraculo = 296;
5         double delta = 0.000001;
6         assertEquals(oraculo, Transformacoes.celsiusParaKelvin(23), delta);
7     }
8 }
```

(b)

Figura 32. Configuração original.

```
1 public class Transformacoes {
2     public static final int K = 273;
3     public static double tranformacao(double tCelsius){
4         return tCelsius + K;
5     }
6 }
7 }
```

(a)

```
1 public class Testes {
2     @Teste
3     public void testeCelsiusParaKelvin(){
4         double oraculo = 296;
5         double delta = 0.000001;
6         assertEquals(oraculo, Transformacoes.tranformacao(23), delta);
7     }
8 }
```

(b)

Figura 33. João altera o nome de um método (a) e ajusta a classe *Testes* (b).

Em paralelo, Maria, que não sabe das alterações de João, adiciona uma classe (Figura 34), que utiliza o método alterado por João, resultando na configuração composta pelos seguintes artefatos: Figura 32.a, Figura 32.b e Figura 34. Durante a verificação de primeiro nível, nenhum problema sintático ou semântico é encontrado, pois a configuração compila corretamente e passa nos testes. Portanto, a verificação de segundo nível é realizada com sucesso na configuração candidata.

```

1 public class LeiDoGasIdeal {
2     public final double R = 8.314472;
3     public double volume(double p, double n, double tCelsius){
4         double tKelvin = Transformacoes.celsiusParaKelvin(tCelsius);
5         return n * R * tKelvin / p;
6     }
7 }

```

Figura 34. Maria adiciona uma classe.

A configuração candidata, gerada pela junção da configuração do desenvolvedor com a configuração corrente do repositório, é composta pelos artefatos de software representados na Figura 33.a, Figura 33.b e Figura 34. Essa configuração é avaliada pela verificação de segundo nível por verificações física, sintática e semântica. Durante a verificação sintática desta configuração, um problema é identificado, pois o método *volume* da classe *LeiDoGasIdeal* chama o método *celsiusParaKelvin*, que não existe mais. Portanto, a verificação sintática apresenta uma falha, resultando na rejeição da configuração de Maria e, deste modo, uma notificação é enviada a Maria descrevendo o problema identificado.

Maria, ao receber a notificação dos problemas identificados, executa o comando de *update*, trazendo as modificações do repositório para o espaço de trabalho. Em seguida, Maria faz a correção no artefato, alterando o método chamado pelo método *volume* da classe *LeiDoGasIdeal*, de *celsiusParaKelvin* para *transformacao* (Figura 35). Posteriormente, faz o *check-in* novamente.

```

1 public class LeiDoGasIdeal {
2     public final double R = 8.314472;
3     public double volume(double p, double n, double tCelsius){
4         double tKelvin = Transformacoes.transformacao(tCelsius);
5         return n * R * tKelvin / p;
6     }
7 }

```

Figura 35. Configuração de Maria.

Durante o processamento do *check-in*, a verificação de primeiro nível atua sobre a configuração formada pelos artefatos representados na Figura 33.a, Figura 33.b e Figura 35. Tal verificação não encontra nenhum problema. Caso nenhum *check-in* tenha ocorrido no intervalo entre o *update* e o *check-in* de Maria, a configuração candidata seria a mesma que a configuração verificada no primeiro nível. Deste modo, nenhum problema também seria identificado na verificação de segundo nível.

4.7.4 CENÁRIO COM PROBLEMA SEMÂNTICO DE SEGUNDO NÍVEL

Outra verificação de segundo nível é a verificação semântica, que é abordada nesse cenário. Para isso, suponha que Maria e João tenham feito *check-out* da configuração ilustrada na Figura 36.

```
1 public class Transformacoes {
2     public static final int K = 273;
3     public static double transformacao(double tCelsius){
4         return tCelsius + K;
5     }
6 }
7 }
```

(a)

```
1 public class LeiDoGasIdeal {
2     public final double R = 8.314472;
3     public double volume(double p, double n, double tCelsius){
4         double tKelvin = Transformacoes.transformacao(tCelsius);
5         return n * R * tKelvin / p;
6     }
7 }
```

(b)

Figura 36. Configuração obtida por João e Maria.

Após realizar *check-out*, João altera a ação do método *transformacao* da classe *Transformacoes*, obtendo o resultado final ilustrado na Figura 37. Essa alteração reflete no comportamento do método *transformacao* que passou a transformar uma temperatura de graus Celsius para Fahrenheit, em vez de transformar de Celsius para Kelvin. Após realizar estas alterações o *check-in* é realizado por João.

```
1 public class Transformacoes {
2     public static double transformacao(double tCelsius){
3         return 5 * (tCelsius - 32) / 9;
4     }
5 }
```

Figura 37. João altera a ação de um método.

Durante o processamento do *check-in* as verificações de primeiro e segundo nível são realizadas com sucesso, pois a configuração candidata é igual à configuração do João (Figura 36.a, Figura 37). Portanto, esta configuração é integrada no repositório.

Em paralelo à alteração de João, Maria cria um caso de teste (Figura 38) e o incorpora em sua configuração. O caso de teste inserido verifica o comportamento do método *volume*, presente na classe *LeiDoGasIdeal*. Após adicionar esse artefato em sua configuração, Maria realiza *check-in*, que reflete suas modificações no repositório.

```

1 public class Testes {
2     @Teste
3     public void gasIdeal() {
4         LeiDoGasIdeal leiDoGasIdeal = new LeiDoGasIdeal();
5         double oraculo = 9744.561184;
6         double delta = 0.000001;
7         assertEquals(oraculo, leiDoGasIdeal.volume(25, 100, 20), delta);
8     }
9 }

```

Figura 38. Maria insere um artefato de software.

Durante a verificação de primeiro nível na configuração de Maria (i.e., configuração formada pelos artefatos da Figura 36.a, Figura 36.b e Figura 38), nenhum problema é identificado. Por outro lado, durante a verificação de segundo nível uma quebra semântica é identificada. A configuração candidata é composta pelos artefatos da Figura 37, Figura 36.b e Figura 38. Portanto, essa é a configuração verificada sintática e semanticamente na verificação de segundo nível. Durante a verificação sintática, nenhuma quebra é encontrada, pois esta configuração compila sem apresentar nenhuma falha. Todavia, durante a verificação semântica uma quebra é identificada, pois o método *transformacao* foi alterado para converter uma temperatura de Celsius para Fahrenheit e não mais de Celsius para Kelvin. Deste modo, uma notificação é enviada ao desenvolvedor descrevendo o problema identificado e é esperado que o desenvolvedor corrija o defeito, de maneira análoga ao problema sintático de segundo nível, e realize *check-in* novamente.

4.8 CONSIDERAÇÕES FINAIS

Com a GC tradicional, artefatos quebrados podem atingir o repositório sem que nenhuma verificação de caráter sintático ou semântico seja realizada, resultando em repositórios inconsistentes. O ciclo tradicional de GC, composto por *check-out*, modificações e *check-in*, executa apenas verificações físicas sobre os artefatos de software que são enviados para o repositório. Portanto, artefatos sintaticamente ou semanticamente quebrados podem habitar o repositório sem que sejam identificados por ferramentas que compõe o ciclo de trabalho tradicional de GC.

Para evitar que artefatos quebrados cheguem ao repositório foi proposta a abordagem Ouriço, que é capaz de identificar e rejeitar artefatos inconsistentes durante o comando de *check-in*. Conforme descrito neste capítulo, a abordagem proposta consiste em uma extensão de comandos usualmente executados no ciclo tradicional de GC, incluindo habilidades que não existem originalmente. Habilidades estas que permitem verificar uma modificação antes

que atinja os artefatos originais, que são compartilhados e utilizados pelos demais desenvolvedores de uma equipe.

Junto a esse propósito, houve uma preocupação constante em manter o desempenho já existente em operações de *check-in*, evitando que a verificação adicione uma grande espera e influencie negativamente na produtividade da equipe. Para isso, o Ouriço trabalha em paralelo ao desenvolvedor, antecipando etapas de compilação logo após o *check-out*, enquanto o desenvolvedor faz as suas modificações. Além disso, a abordagem foi projetada para trabalhar de forma assíncrona, liberando o desenvolvedor durante o processamento da verificação, logo após o *check-in*.

CAPÍTULO 5 – IMPLEMENTAÇÃO E UTILIZAÇÃO DO OURIÇO

5.1 INTRODUÇÃO

Esta abordagem foi implementada em um ambiente, denominado Oceano, que reúne abordagens desenvolvidas pelo GEMS (Grupo de Evolução e Manutenção de Software). O Oceano é um ambiente integrado para desenvolvimento de abordagens ligadas a GC. Atualmente esse ambiente possui três abordagens, além do Ouriço. A primeira abordagem, denominada Peixe-espada, utiliza o tempo ocioso das máquinas para refatorar código fonte visando a melhoria dos atributos de qualidade do software em questão. A segunda abordagem, denominada Ostra, apresenta um apoio para que gerentes de projeto e equipes de desenvolvimento possam tomar decisões mais precisas, baseada em mineração de indicadores e métricas. A terceira abordagem, denominada Polvo, realiza verificações em ramos e estima a dificuldade para integração.

Neste capítulo é detalhada a implementação do Ouriço, que é o resultado deste trabalho de mestrado. Conforme discutido anteriormente, o Ouriço é uma abordagem para manutenção da consistência de repositório de GC. Ou seja, ele realiza verificações em artefatos de software e quando algum defeito, que não é permitido no repositório, é identificado, a configuração enviada pelo desenvolvedor é rejeitada.

Para viabilizar a concretização do Ouriço foram tomadas algumas decisões de projeto. Essas decisões foram utilizar o Subversion e o Maven como SCV e SGC, respectivamente. A utilização de Subversion e Maven foi escolhida por ser uma combinação bastante utilizada por empresas de desenvolvimento de software e, também, por projetos *open source*. Tal escolha permitiu que os experimentos, discutidos no Capítulo 6, fossem realizados com maior facilidade, devido à disponibilidade de projetos que seguem esse padrão e, conseqüentemente, podem ser utilizados nos experimentos.

O restante desse capítulo está organizado com as seguintes seções: Na Seção 5.2 é dada uma visão geral do Ouriço e do Oceano. Na Seção 5.3, o Oceano é descrito levando em consideração as suas terminologias que são: item de configuração, projeto, usuário e, também, a associação entre um projeto e um usuário. Na Seção 5.4, é apresentado o Ouriço e as implementações dos comandos de *check-out*, *check-in* e *update*. Além disso, é apresentado o painel web, meio que o usuário pode acompanhar as tarefas do Ouriço. Finalmente, na Seção 5.5, são feitas as considerações finais relativas à implementação.

5.2 VISÃO GERAL

Para desenvolver esta abordagem foram utilizadas tecnologias que auxiliaram na produção de uma implementação robusta. Para a camada de banco de dados, foi utilizada a especificação JPA 1.0 (Keith and Schincariol 2009) para modelos de persistência na camada de acesso a dados utilizando o padrão de projetos DAO (Nock 2003); para a camada de visualização, foi utilizado o JSF 1.2 (Burns and Schalk 2009) integrado com *Facelets* (Aranda and Wadia 2008) e *RichFaces* (Katz 2008); para criar a implementação das regras de negócios em uma camada interna, foi utilizado o *Application Service* (Alur et al. 2003); e para modularizar interesses ortogonais como acessar ao banco de dados e buscas genéricas, foi utilizado o *AspectJ* (Laddad 2009). Além disso, a implementação atual utiliza o *PostgreSQL* (Smith 2010), está implantada no servidor de aplicações *Glassfish* (Heffelfinger 2007) e utiliza o *SVNKit* (TMate Software 2011) para executar as ações do Subversion.

Este projeto está dividido em módulos que são de uso geral do Oceano e, também, em módulos de uso específico da abordagem Ouriço. O ambiente Oceano, descrito na Seção 5.3, está dividido em dois módulos: o primeiro é o oceano-core, um módulo onde se concentra a maioria das suas funcionalidades, e o segundo o oceano-web, módulo onde estão concentradas interfaces para cadastro de usuário, projetos e outros. Por outro lado, o Ouriço foi dividido em quatro módulos: (1) *check-out*, (2) *check-in*, (3) *update* e (4) *web*.

O banco de dados do Oceano e do Ouriço possui a modelagem representada na Figura 39. Neste diagrama é possível identificar do lado direito (em tom mais claro) as entidades do Oceano e no lado esquerdo (em tom mais escuro) as entidades do Ouriço. Do lado do Ouriço é possível visualizar as seguintes entidades: *ConfiguracaoVerificacao*, que é responsável pelas configurações utilizadas durante a verificação do Ouriço; *Estado*, que é responsável por armazenar os estados pelos quais um *autobranch* passou; *Checkout*, que é responsável por controlar os *check-outs* realizados pelo Ouriço; *VerificacaoPosCheckout*, que controla se um *autobranch* foi verificado após a realização do *Check-out*; e, finalmente, *ProjectConfiguration*, que guarda as configurações de um projeto, como, por exemplo, a política adotada, entre outros. Do lado do Oceano é possível visualizar as seguintes entidades: *SoftwareProject*, responsável por armazenar dados de um projeto de software; *ConfigurationItem*, que armazena os dados de um item de configuração; *ProjectUser*, que é responsável por fazer a ligação entre um usuário do Oceano e um projeto; *Repository*, que define o SCV utilizado para o projeto; e, finalmente, *OceanoUser*, que é responsável por

armazenar os usuários do Oceano. Essas entidades são discutidas, com mais detalhes, ao longo deste capítulo quando uma interface ou serviço for utilizá-las.

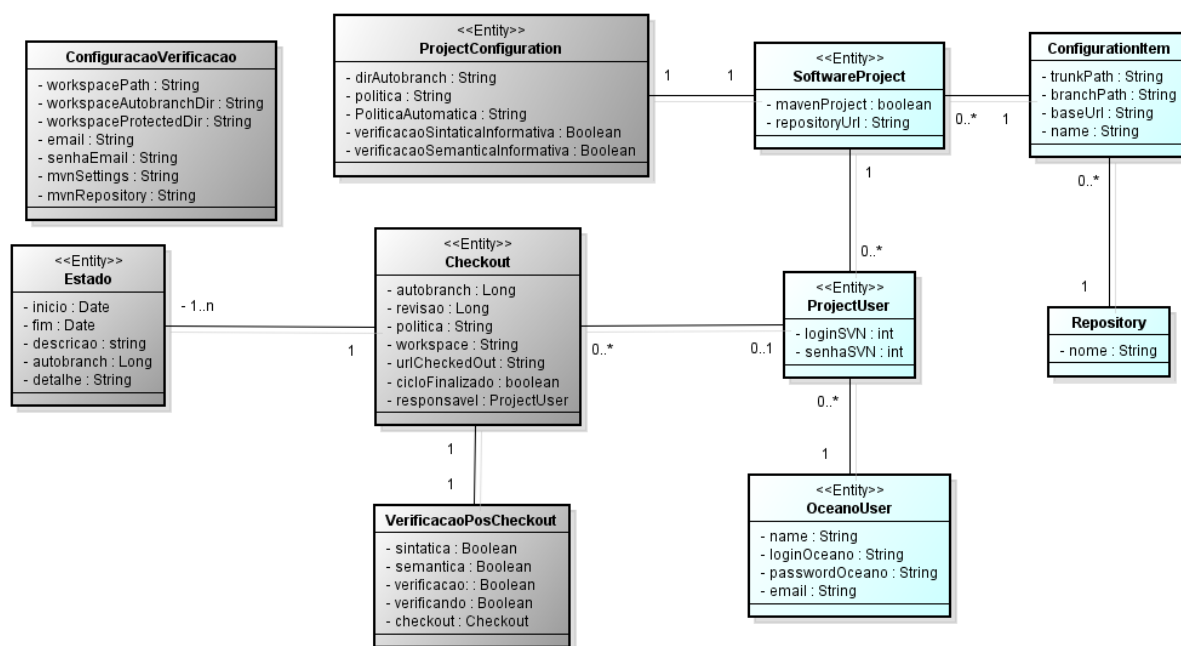


Figura 39. Diagrama do banco de dados do Ouriço e Oceano.

5.3 OCEANO

O Oceano consiste de uma infraestrutura pela qual é possível configurar projetos que posteriormente serão utilizados por cada uma das abordagens contidas nele. Uma interface web foi disponibilizada para registrar projetos, itens de configuração, usuário e vínculo entre usuários e projetos, que são utilizados para o funcionamento do Oceano.

Conforme discutido na visão geral, o Oceano atualmente é composto de 4 abordagens, que são (Figura 40): Ostra, Ouriço, Peixe-espada e Polvo. Cada um desses projetos possui objetivos distintos que são respectivamente: apresentar um apoio para que gerentes de projeto e equipes de desenvolvimento possam tomar decisões mais precisas, baseada em mineração de indicadores e métricas; evitar que o repositório fique inconsistente; utilizar o tempo ocioso das máquinas para refatorar código fonte visando a melhoria de atributos de qualidade deste; e realizar verificações em ramos e estimar a dificuldade para integração. Entretanto, este documento se limita ao conteúdo do Oceano e do Ouriço, que são de interesse da abordagem proposta neste trabalho.

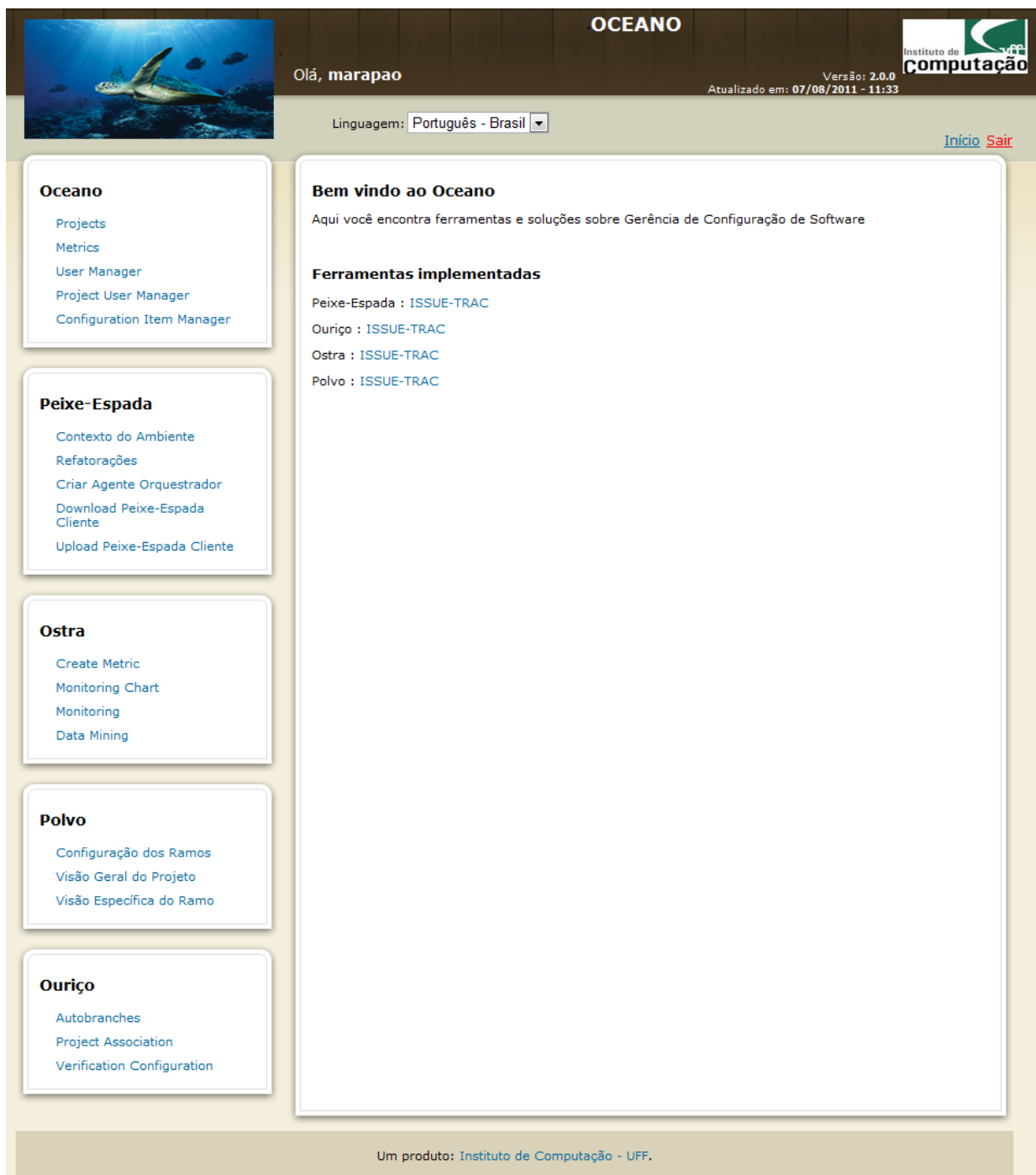


Figura 40. Página inicial do Oceano.

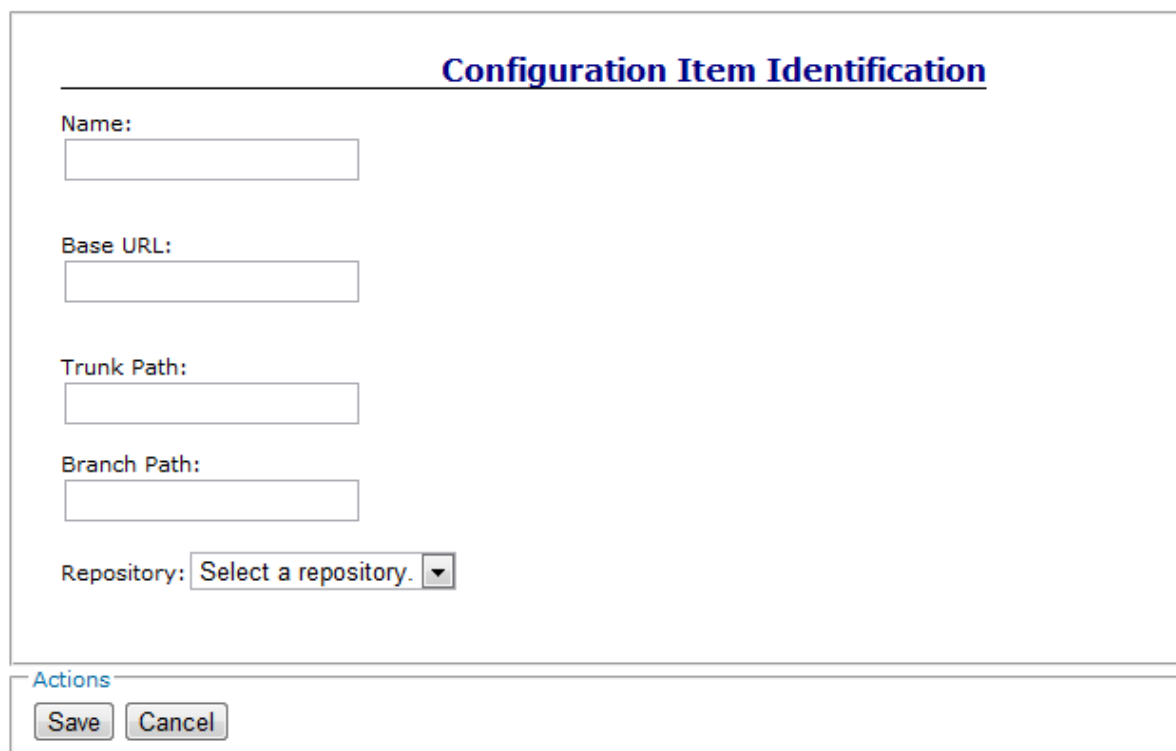
No oceano é possível realizar toda a parte administrativa de organização dos projetos e suas dependências. Portanto, o restante desta seção está dividido da seguinte forma: na subseção 5.3.1, é discutido o cadastro de itens de configuração; na subseção 5.3.2, é abordado o cadastro de projetos; na subseção 5.3.3 é ilustrado o cadastro de usuários do Oceano; e, finalmente, na subseção 5.3.4 é discutido como realizar a associação entre um projeto e um usuário oceano.

5.3.1 CADASTRO DE ITEM DE CONFIGURAÇÃO

Para o Oceano, um item de configuração é um módulo do repositório, que consiste usualmente de um local que contém *trunk* (nome convencional dado a linha principal de desenvolvimento do Subversion), *branches* e *tags*. Para cadastrar um item de configuração uma interface (Figura 41) é disponibilizada e são necessárias as seguintes informações:

- Nome (do inglês, *name*), que é um identificador do item de configuração;
- URL base (do inglês, *base URL*), a URL até chegar ao item de configuração cadastrado como, por exemplo, o local onde é armazenado o módulo de *update* do Ouriço - <https://gems.ic.uff.br/svn/oceano/ourico/update/>;
- Caminho do *trunk* (do inglês, *trunk path*), que é o local onde está a linha principal de desenvolvimento no repositório;
- Caminho dos *branches* (do inglês, *branch path*), que representa o local onde serão criados os ramos; e
- Repositório (do inglês, *repository*), que indica o SCV que será utilizado (campo criado para possíveis extensões de SCV no Oceano que atualmente suporta apenas o Subversion).

Configuration Item Manager



The screenshot displays a web interface titled "Configuration Item Manager". The main section is titled "Configuration Item Identification" and contains five input fields: "Name:", "Base URL:", "Trunk Path:", "Branch Path:", and "Repository:". The "Repository:" field is a dropdown menu with the text "Select a repository." and a downward arrow. At the bottom of the form, there is a section labeled "Actions" containing two buttons: "Save" and "Cancel".

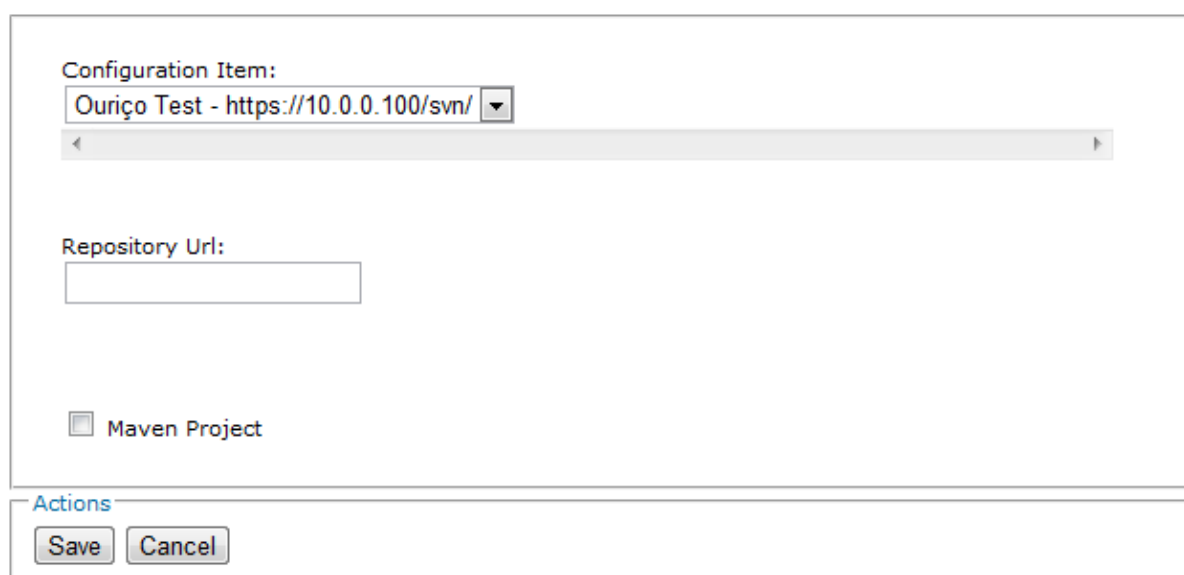
Figura 41. Tela para cadastro de um item de configuração.

5.3.2 CADASTRO DE PROJETO

Para o Oceano, um projeto representa uma das linhas de desenvolvimento que um item de configuração possui, por exemplo, o trunk. Para cadastrar um projeto é necessário um cadastro prévio do item de configuração no qual o projeto em questão estará baseado. Portanto, para cadastrar um projeto, o desenvolvedor ou gerente de projeto executa os seguintes passos (Figura 42):

- Selecionar um item de configuração (do inglês, *configuration item*);
- Inserir a URL inteira de um projeto (do inglês, *repository Url*) como `https://gems.ic.uff.br/svn/oceano/ourico/update/trunk/`, por exemplo.
- Definir se o projeto obedece à estrutura do *Maven* (do inglês, *Maven project*) ou não.

Project Manager



The screenshot shows a web form titled "Project Manager". It contains the following elements:

- A label "Configuration Item:" followed by a dropdown menu showing "Ourico Test - https://10.0.0.100/svn/" with a small downward arrow.
- A horizontal scrollbar below the dropdown.
- A label "Repository Url:" followed by an empty text input field.
- A checkbox labeled "Maven Project" which is currently unchecked.
- A section titled "Actions" at the bottom, containing two buttons: "Save" and "Cancel".

Figura 42. Tela para cadastro de um projeto.

5.3.3 CADASTRO DE USUÁRIOS

Além de detalhes de projeto, outra utilidade do Oceano é realizar o controle de usuários, que podem ter acesso a seus serviços. Para cadastrar um usuário do Oceano, o gerente ou desenvolvedor deve fornecer as seguintes informações (Figura 43):

- Nome, que consiste no nome completo do usuário;
- E-mail, que é utilizado para notificar o usuário sobre alguma ação do Oceano ou seus subprojetos.
- Nome de usuário, que consiste em um nome reduzido utilizado para fazer *login*.

- Senha, que possibilita acessar a interface web do Oceano e também permite a validação do acesso aos serviços dos subprojetos do Oceano.

User Manager

Personal Data

Name:

E-mail:

Oceano Data

Username:

Password:

Password Confirmation:

Actions

Figura 43. Tela para cadastro de um usuário Oceano.

5.3.4 ASSOCIAÇÃO ENTRE USUÁRIOS E PROJETOS

O Oceano não possui papéis de usuário definidos, devido a decisões de implementações. Entretanto, possui um mecanismo para controle de usuários que permite ou não um usuário acessar um projeto. A definição de papéis (e.g., administrador, usuário simples e outros) não foi implementada para essa versão do Oceano, embora a restrição a projetos seja desejável. Portanto, para restringir acesso de usuário foi criada uma associação projeto-usuário (*ProjectUser* na Figura 39) cujo objetivo é liberar acesso a usuários que têm competência para ter acesso ou ainda alterar tais projetos (Figura 44).

Para gerenciar os vínculos entre usuários e projetos, o desenvolvedor ou gerente de projeto deve executar os seguintes passos (Figura 45):

1. Selecionar um usuário Oceano, previamente cadastrado.
2. Selecionar uma das seguintes opções para um projeto: editar, remover ou adicionar uma relação entre um usuário e um projeto (Supondo que adição a seja selecionada).

3. O gerente deve incluir *login* e senha que usuário Oceano utiliza para ter acesso ao SCV referente ao projeto previamente selecionado;
4. Finalmente, salvar suas alterações.

Project User Manager

Users:
 ▼

Project: Ouriço Test - https://10.0.0.100/svn/trunk

Login:

Password:

Password Confirmation:

Actions

Projetos

OceanoUser	Project	Login	Link	Delete
marapao	Ouriço Test - https://10.0.0.100/svn/trunk	marapao	Change	Delete

Figura 44. Ligação de um usuário Oceano com um projeto.

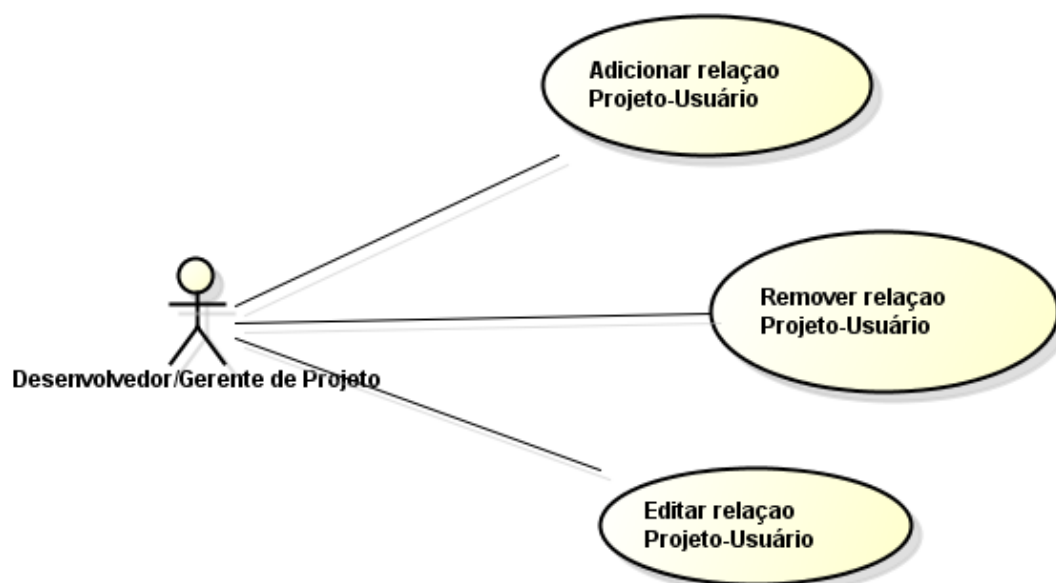


Figura 45. Casos de uso da relação projeto-usuário.

Após salvar as alterações o usuário Oceano poderá atuar nos projetos que tem permissão, como acontece no caso do Ouriço. Quando um usuário não tem acesso a um projeto específico, tarefas que envolvem obtenção de artefatos e, também, modificações no repositório via Ouriço não são permitidas. Portanto, para que um desenvolvedor qualquer tenha acesso a serviços do Ouriço, esse cadastro é fundamental.

5.4 OURIÇO

Conforme discutido no Capítulo 4, a intenção do Ouriço é evitar que artefatos inconsistentes cheguem ao repositório e, consequentemente, o torne inconsistente. Para alcançar tal objetivo, o Ouriço foi implementado obedecendo a uma divisão nos seguintes módulos: *check-out*, *check-in*, *update* e o módulo web, local onde está implementado o painel de acompanhamento da prova de conceito, proposta por esta abordagem.

No restante dessa seção são discutidos os módulos do Ouriço e detalhes de implementação. Essa seção está dividida da seguinte forma: a subseção 5.4.1 trata do comando de *check-out*; a subseção 5.4.2 trata do comando de *check-in*; a subseção 5.4.3 trata do comando de *update*; e, finalmente, a subseção 5.4.4, trata da parte web do Ouriço.

5.4.1 CHECK-OUT

O módulo de *check-out* é o responsável por implementar o comando que obtém artefatos de software de um repositório e cria o mecanismo para a proteção do repositório. Conforme discutido no Capítulo 4, o *check-out* do Ouriço estende os comandos tradicionais em GC através da criação de *autobranches* e verificações, quando necessárias, impedindo que artefatos com conflitos físicos ou quebrados cheguem ao repositório.

Para a criação de um *autobranch*, é requisitado o próximo *autobranch* válido que será preenchido com a configuração desejada pelo desenvolvedor. A estratégia para nomeação dos ramos adotada pelo Ouriço é baseada em uma sequência de inteiros. Portanto, quando um *check-out* é realizado, uma busca pelo maior *autobranch* é efetuada no banco de dados e, posteriormente, um ramo com o nome igual ao valor do “maior *autobranch* encontrado” + 1 é criado dentro do compartimento do repositório reservado para os *autobranches*. Essa criação é realizada com o comando de *copy* do Subversion, tendo como origem o endereço especificado pelo desenvolvedor e destino o endereço do *autobranch*, baseado no nome definido. Deste modo, um *autobranch* com a mesma configuração que o desenvolvedor requisitou é criado.

Após o *autobranch* ser criado, uma cópia é enviada para o desenvolvedor e este poderá realizar modificações nos artefatos sem que os artefatos originais sejam atingidos. Para

viabilizar isso, após a criação do *autobranch*, um comando de *check-out* tradicional é realizado do *autobranch* para o espaço de trabalho do desenvolvedor.

Os artefatos originais só sofrerão alteração após todas as verificações, durante o *check-in*, serem realizadas com sucesso e os artefatos serem integrados ao local de origem do repositório, onde o desenvolvedor havia requisitado originalmente o *check-out*. Esse passo será discutido com mais detalhes na Seção 5.4.2.

O módulo de *check-out* possui duas interfaces para interação com o desenvolvedor: uma gráfica (Figura 46) e outra em linha de comando (Figura 47). Para realização do *check-out*, são necessários os seguintes dados:

- *Login* do SCV, que é cadastrado no momento em que uma associação entre um projeto e um usuário é realizada.
- Senha do SCV, que é cadastrada no momento em que um vínculo entre um projeto e um usuário é realizado.
- URL repositório, que se refere ao projeto cadastrado.
- Caminho do espaço de trabalho, local onde o desenvolvedor deseja obter os artefatos requisitados durante o *check-out*.

Figura 46. Interface desktop para realização de *check-out*.

```
$ java -jar OuricoCLI -command co -username username -password password -
url http://localhost/svn -workspace workspace
The check-out from url http://localhost/svn was successfully performed.
```

Figura 47. Interface de linha de comando para realização de *check-out*.

A implementação do Ouriço possui estas duas versões devido à motivação inicial de ser uma abordagem não intrusiva, ou seja, não mudar o estilo de interação do desenvolvedor com o SCV. Portanto, as duas interfaces que são tradicionalmente oferecidas para os desenvolvedores realizem *check-out* são oferecidas também pela ferramenta que implementa a abordagem Ouriço.

5.4.2 CHECK-IN

O comando de *check-in* é utilizado para enviar modificações feitas localmente para que fiquem acessíveis para toda a equipe no repositório. No intervalo entre o *check-out* e o *check-in* o desenvolvedor realiza modificações nos artefatos de software para que uma tarefa corretiva, evolutiva ou de qualquer outra natureza seja concluída. Após estas modificações serem concluídas, é necessário disponibilizá-las para o restante da equipe de desenvolvimento, através do comando de *check-in* do Ouriço. Portanto, após o pós-processamento do *check-in* ser concluído, os demais membros da equipe poderão utilizar a configuração resultante para executar suas tarefas que podem depender ou não desta tarefa. Vale ressaltar que com a utilização desta abordagem os companheiros de equipe poderão depositar maior confiança nos artefatos do repositório que foram previamente verificados.

O Ouriço não disponibiliza nenhuma interface especial para realização do *check-in* em sua implementação, pois ele atua após a realização de tal comando através de um gatilho *post-commit*. O desenvolvedor que utilizar a abordagem Ouriço poderá realizar *check-in* com a ferramenta que achar mais conveniente. Estas ferramentas podem ser a própria interface de linha de comando disponibilizada pelo Subversion (Figura 48) ou ainda implementações com interface gráfica como, por exemplo, o TortoiseSVN (Figura 49). Após o *check-in* ser concluído uma tarefa de pós-processamento será realizada para garantir a consistência dos artefatos que foram enviados para o repositório.

```
$svn ci workspace --username "username" --password "password" -m "adição  
do metodo transformação"  
Sending      workspace/file1  
Sending      workspace/file2  
...  
Sending      workspace/fileN  
Transmitting file data..  
Committed revision 22.
```

Figura 48. Interface de linha de comando nativa do Subversion.

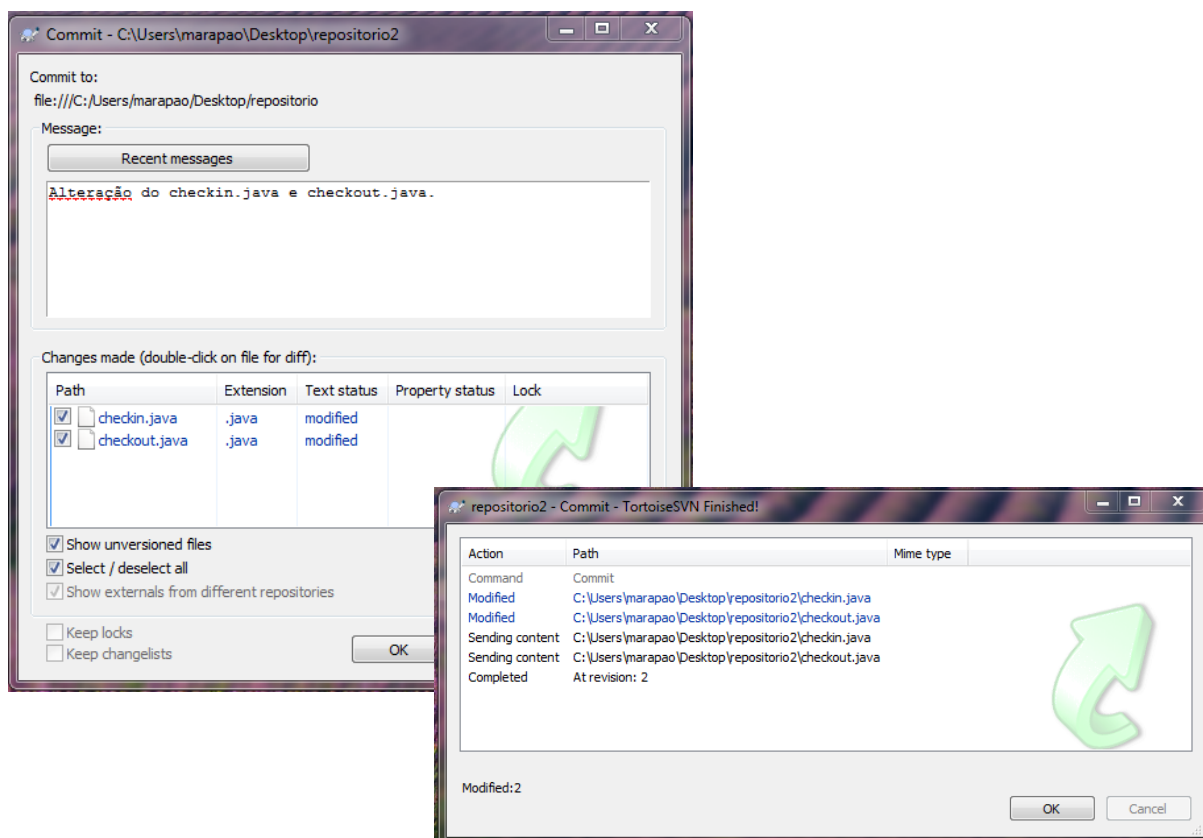


Figura 49. Interface de *check-in* do TortoiseSVN.

No pós-processamento do *check-in*, os artefatos do *autobranch* são verificados e, quando pertinente, são integrados ao seu local de origem no repositório. Após a realização do *check-in* tradicional, os artefatos enviados pelo desenvolvedor são depositados no *autobranch* que os originou. Este, no entanto, não é o resultado esperado para um comando de *check-in*. Logo, para enviar estes artefatos para o local original, um pós-processamento do *check-in* é realizado através de um *hook post-commit*, que foi discutido no Capítulo 2. Esse pós-processamento realiza verificações dos artefatos e os integra ao local de origem no repositório.

Para realizar as verificações sintática e semântica, a abordagem proposta utiliza o *Maven* através de chamadas dos comandos *compile* (Figura 50) e *test* (Figura 51), respectivamente. Para execução desses comandos é utilizado o espaço de trabalho criado no momento do *check-out*, durante a verificação pós *check-out*. Os comandos citados anteriormente realizam tarefas de compilação e execução de testes sobre um projeto e seus subprojetos, caso existam. Além disso, tais comandos são chamados internamente pelo Ouriço e, portanto, as representações da Figura 50 e da Figura 51 são mostradas para fins didáticos.

```

$ mvn compile -f pathWorkspace/pom.xml
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building svn
[INFO]    task-segment: [compile]
[INFO] -----
...
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] -----
[INFO] BUILD SUCCESSFUL

```

Figura 50. Comando do *Maven* para compilar um projeto.

```

mvn test -f pathWorkspace/pom.xml
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building svn
[INFO]    task-segment: [test]
[INFO] -----
...
-----
T E S T S
-----
Running br.uff.ic.gems.test.svn.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.033 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
...

```

Figura 51. Comando do *Maven* para executar os testes de um projeto.

A verificação de primeiro nível é a primeira a ser aplicada sobre a configuração do desenvolvedor. Conforme discutido no Capítulo 4, a verificação de primeiro nível segue uma política que é definida por algum membro da equipe de desenvolvimento (*e.g.*, desenvolvedor, gerente e etc.). Portanto, durante a verificação de primeiro nível uma busca será realizada na tabela *projectConfiguration* do banco de dados do Ouriço, cujo modelo está representado na Figura 39, para recuperar tal informações. De posse da política, são aplicados os filtros necessários para que os erros, cobertos pela política, possam ser identificados. Quando algum erro é encontrado na configuração do desenvolvedor, as seguintes ações são executadas:

- O ciclo é imediatamente finalizado;
- O responsável é notificado via e-mail;
- O estado atual é registrado no painel de controle do Ouriço, presente no módulo web. Este painel será discutido posteriormente na Seção 5.4.4.

Alternativamente, quando nenhum erro é encontrado na configuração do desenvolvedor, a configuração candidata é gerada. Para gerar essa configuração, uma junção que traz todas as modificações realizadas na linha de desenvolvimento original, recuperada do banco de dados do Ouriço, é realizado e seu resultado é aplicado sobre a configuração do desenvolvedor (Figura 52). A sintaxe do comando de junção exige que o comando seja executado de dentro do espaço de trabalho que a configuração do desenvolvedor está. É importante ressaltar que o desenvolvedor não executa esse comando, pois eles são executados internamente pelo Ouriço através da chamada de métodos do SVNKit.

```
$ espacoDeTrabalho> svn merge pathLinhaDeDesenvolvimentoOriginal
--- Merging r4 through r26 into `.`:
C pathWorkspaceDesenvolvedor/file1
U .
```

Figura 52. Junção para criação da configuração candidata.

Outra informação importante é que durante a criação da configuração candidata alguns conflitos de origem física podem ocorrer e serem identificados por este passo. Entretanto, por questões didáticas, tal tarefa está sendo atribuída à verificação de segundo nível, que ocorre logo após a criação da configuração candidata.

A verificação de segundo nível é aplicada sobre a configuração candidata. A verificação de segundo nível trata a configuração candidata de forma similar à verificação de primeiro nível, ou seja, ela recupera a política e aplica os filtros necessários sobre a configuração em questão. A diferença está na capacidade de identificar conflitos físicos, que quando encontrados também resultam na interrupção do comando de *check-in*, notificação do responsável e registro do estado atual no módulo web do Ouriço. Alternativamente quando nenhum erro é encontrado na configuração candidata, os artefatos da configuração do desenvolvedor são integrados ao repositório.

Durante a integração, os artefatos verificados são enviados para o seu local de origem no repositório. Esta integração é realizada obedecendo aos seguintes passos:

1. *Check-out* da linha de desenvolvimento protegida pela abordagem Ouriço para o workspaceProtegido;

2. *Merge* com a opção *reintegrate* do *autobranch*, que será integrado com o *workspaceProtegido*; e
3. *Check-in* do *workspaceProtegido* no repositório.

Ao final desses três passos, as modificações realizadas pelo desenvolvedor passam a ficar disponíveis para o restante da equipe de desenvolvimento.

5.4.3 UPDATE

O comando de *update* é responsável por refletir no espaço de trabalho do desenvolvedor as modificações que estão no repositório, mas ainda não estão disponíveis para o desenvolvedor. Conforme discutido no Capítulo 4, o *update* tradicional não tem o comportamento desejado com a utilização desta abordagem, pois as modificações viriam do *autobranch*, que é um local do repositório onde somente o desenvolvedor que realizou *check-out* pode alterar. Portanto, esta implementação estende o comando tradicional para que as modificações venham do local que originou o *autobranch* e, deste modo, gere o resultado desejado pelo desenvolvedor que executar o comando de *update*, dada que a criação de um *autobranch* deve ser imperceptível. Além disso, o comando de *update* é importante para solucionar as falhas identificadas durante a verificação de segundo nível.

Durante a execução do *update* é necessário recuperar a linha de desenvolvimento que deu origem ao *autobranch*. Durante a execução do *check-out*, uma das informações que são armazenadas é o projeto que originou um determinado *autobranch*. Portanto, para cada *autobranch* gerado existe uma relação com uma linha de desenvolvimento e esta relação é recuperada do banco de dados durante o comando de *update*.

De posse do endereço do projeto original, é possível refletir as modificações realizadas no repositório para o espaço de trabalho do desenvolvedor. Essa reflexão é realizada utilizando o comando de junção do Subversion. Tal comando é aplicado para realizar a junção com a semântica ilustrada na Figura 53. Para a execução do comando com a sintaxe da Figura 53, é assumido que o comando está sendo chamado de dentro do espaço de trabalho que se deseja atualizar. Ainda com relação ao comando de junção, é importante ressaltar que ele é chamado internamente pelo Ouriço através do SVNKit.

```
$ EspacoDeTrabalho> svn merge enderecoOriginalDoProjeto
--- Merging r4 through r26 into '.'
C pathWorkspaceDesenvolvedor/file1
U .
```

Figura 53. Junção do *update*.

A implementação do Ouriço possui, também, duas interfaces para realização do *update*: uma gráfica (Desktop) e outra da linha de comando. Para execução deste comando, é necessário que o desenvolvedor forneça:

- Login do SCV, que é cadastrado no momento em que uma associação entre um projeto e um usuário é realizada.
- Senha do SCV, que é cadastrada no momento em que uma associação entre um projeto e um usuário é realizada.
- Endereço do espaço de trabalho, em que o desenvolvedor deseja refletir as modificações realizadas na linha de desenvolvimento que deu origem à configuração presente no espaço de trabalho em questão.

Figura 54. Interface gráfica do módulo de *update* do Ouriço.

```
$ java -jar OuricoCLI -command update -username username -password
password-workspace workspace
The workspace is now up-to-date.
```

Figura 55. Interface de linha de comando do módulo de *update* do Ouriço.

5.4.4 WEB

O módulo web do Ouriço é responsável por disponibilizar informações sobre o andamento do seu ciclo e, também, pela configuração das verificações que serão realizadas pelo protótipo do Ouriço. O módulo web do Ouriço está dividido em duas partes: configuração e acompanhamento do ciclo.

Na parte de configuração é possível configurar projetos (Figura 56) e também informações relacionadas ao servidor que executará as verificações sobre os artefatos. Para configurar um projeto, é necessário que o desenvolvedor:

- Selecione o nome projeto a ser configurado;
- Forneça o diretório onde serão armazenados os *autobranches*;
- Selecione a política adotada e os filtros informativos que serão aplicados. No caso da política dinâmica não é disponibilizado o cadastro de filtros informativos, pois os filtros são definidos durante o *check-out*.

Project Association

The screenshot shows a web interface titled "Settings" for "Project Association". It contains three main configuration sections: "Project:" with a dropdown menu showing "externo - http://192.168.0.101/svn/trunk"; "Autobranch Directory:" with a text input field containing "autobranches"; and "Policy:" with a dropdown menu showing "Restritiva". Below these fields is an "Actions" section with "Save" and "Cancel" buttons.

Figura 56. Interface de configuração de um projeto.

O módulo web ainda permite que um desenvolvedor ou gerente de projetos configure o servidor que executará as verificações através da interface ilustrada na Figura 57. Nesta configuração são inseridos dados utilizados pelo Maven, configurações de envio de e-mail e, também, locais para armazenamento de *autobranches* e artefatos protegidos. Tais locais são utilizados para realizar as construções de primeiro e segundo nível.

Verification Configuration

Maven

Settings.xml Path:

Maven repository Path:

Email

E-mail:

Password:

Password Confirmation:

Server

Workspace:

Autobranch Directory:

Protected Directory:

Actions

Figura 57. Interface para configuração do servidor.

Esta configuração é realizada pelo Ouriço e para isso o desenvolvedor deve fornecer:

- Caminho do *settings.xml* do Maven, arquivo de configuração que permite, entre outras coisas, acessar repositórios externos através de *login* e senha descritos nele.
- Caminho para um repositório Maven, onde serão armazenadas as dependências, baixadas durante a construção, e também os executáveis gerados durante a construção de um projeto.
- E-mail padrão para envio de notificações do Ouriço.
- Senha do e-mail para que o Ouriço possa se autenticar e enviar o e-mail.

- Espaço de trabalho, que servirá de base para que os artefatos sejam armazenados e verificados no servidor.
- Diretório dos *autobranches*, criado dentro do espaço de trabalho do servidor e que armazenará os *check-outs* realizados em *autobranches* para que posteriormente sejam realizadas as verificações na configuração do desenvolvedor. Essa opção é disponibilizada para que o desenvolvedor organize as configurações da maneira que achar pertinente.
- Diretório protegido, criado dentro do espaço de trabalho do servidor e que armazenará os *check-outs* realizados de projetos protegidos pela abordagem. Neste local serão realizadas as verificações na configuração candidata. Assim como o diretório dos *autobranches*, tal opção é disponibilizada para que o desenvolvedor organize as configurações da maneira que achar pertinente.

Além das configurações, o Ouriço também oferece uma interface pela qual é possível acompanhar o andamento do ciclo dos *autobranches*. Durante o ciclo de trabalho do Ouriço, algumas tarefas podem estar em andamento, outras sendo realizadas ou concluídas. No caso de uma tarefa em andamento, ela pode estar sendo verificada ou ter alguma falha identificada. Neste último caso, o desenvolvedor deve remover o defeito que deu origem a essa falha e enviar a configuração para o repositório novamente.

Para acompanhar o ciclo de desenvolvimento, o protótipo fornece um painel de controle, como ilustrado na Figura 58. Nesse painel, é possível visualizar os *autobranches* criados, a configuração que deu origem a este *autobranch*, o usuário que realizou *check-out*, a URL do repositório que sofreu *check-out* e o estado atual do *autobranch*.

O detalhamento das tarefas realizadas em um *autobranch* também é possível de ser visualizado a partir desse painel. Quando um desenvolvedor sente a necessidade de obter mais informações das tarefas que foram executadas em um *autobranch*, ele pode clicar no hiperlink com o número do *autobranch*. Após clicar, uma nova página (Figura 59) é exibida com as tarefas que foram executadas e o intervalo de tempo dedicado a cada uma das tarefas. Neste caso, está sendo mostrado o *autobranch* 12, que foi integrado com sucesso após três falhas sintáticas em verificações de primeiro nível.

Autobranches

Below are the created autobranches.

Autobranches

Autobranch	Revision	User	Repository URL	Current State
1	4	marapao	https://10.0.0.100/svn/trunk	Checked-out
2	5	marapao	https://10.0.0.100/svn/trunk	Checked-out
3	6	marapao	https://10.0.0.100/svn/trunk	Checked-out
4	9	marapao	https://10.0.0.100/svn/trunk	Checked-out
5	11	marapao	https://10.0.0.100/svn/trunk	Integration successfully performed
6	20	marapao	https://10.0.0.100/svn/trunk	Integration successfully performed
7	24	marapao	https://10.0.0.100/svn/trunk	Integration successfully performed
8	28	marapao	https://10.0.0.100/svn/trunk	Integration successfully performed
9	33	marapao	https://10.0.0.100/svn/trunk	First level syntactic verification failed.
10	34	marapao	https://10.0.0.100/svn/trunk	First level semantic verification failed.
11	4	marapao	http://localhost/svn/trunk	Integration successfully performed
12	9	marapao	http://localhost/svn/trunk	Integration successfully performed
13	16	marapao	http://localhost/svn/trunk	Checked-out
14	18	marapao	http://localhost/svn/trunk	Integration successfully performed
15	23	marapao	http://192.168.0.101/svn/trunk	Integration successfully performed

Figura 58. Painel de acompanhamento de autobranches.

Quando um erro é identificado, é possível exibir detalhes deste erro clicando o hiperlink “*detail*”. A Figura 60 apresenta a primeira ocorrência de quebra sintática durante a verificação de primeiro nível do *autobranch* 12. Nela é possível identificar que um artefato possui um trecho fora do padrão que, neste caso, é uma variável que é utilizada sem ser declarada no arquivo App.java.

Autobranch 12 cycle

Start Time	End Time	Description
2011-06-11 21:48:33.997	2011-06-11 21:48:46.101	Checked-out
2011-06-11 21:49:44.1	2011-06-11 21:49:44.593	First level syntactic verification failed. (Detail)
2011-06-11 21:50:36.1	2011-06-11 21:50:36.612	First level syntactic verification failed. (Detail)
2011-06-11 21:51:39.1	2011-06-11 21:51:39.766	First level syntactic verification failed. (Detail)
2011-06-11 21:52:34.1	2011-06-11 21:52:34.699	First level syntactic verification successfully performed.
2011-06-11 21:52:34.704	2011-06-11 21:52:35.383	First level semantic verification successfully performed.
2011-06-11 21:52:35.386	2011-06-11 21:52:36.112	Second level physic verification successfully performed
2011-06-11 21:52:36.115	2011-06-11 21:52:36.333	Second level syntactic verification successfully performed
2011-06-11 21:52:36.337	2011-06-11 21:52:37.008	Second level semantic verification successfully performed.
2011-06-11 21:52:37.011	2011-06-11 21:52:41.105	Integration successfully performed

Figura 59. Interface para o detalhamento de uma *autobranch*.

Autobranch 12 details

First level syntactic verification failed.

First level syntactic verification failed. Build for project: br.uff.ic.gems.test:svn:jar:1.0-SNAPSHOT failed during execution of mojo: org.apache.maven.plugins:maven-compiler-plugin:2.0.2:compile /src/main/java/br/uff/ic/gems/test/svn/App.java:[14,0] not a statement

Figura 60. Interface para detalhes de um defeito.

5.5 CONSIDERAÇÕES FINAIS

O Ouriço apresenta uma implementação que auxilia a abordagem a não ser muito intrusiva, ou seja, não alterar muito o ciclo de trabalho do desenvolvedor. A implementação do Ouriço foi feita de modo que um desenvolvedor tenha ferramentas de apoio muito próximas das ferramentas disponibilizadas por SCV tradicionais. Essa decisão foi tomada para que a resistência à utilização da ferramenta e a curva de aprendizado sejam minimizadas.

Como as verificações do Ouriço são realizadas de maneira assíncrona, o painel de controle é um forte aliado do desenvolvedor. Um usuário do Ouriço, em determinados projetos, principalmente nos que possuem verificações mais demoradas, pode ficar sem saber em que passo um *autobranch* está ou ainda em que fase de desenvolvimento se encontra um projeto. No entanto, pelo painel de controle do Ouriço é possível identificar quais *autobranches* estão em estado de verificação, quais estão sendo modificados e, ainda, quais já foram integrados ao repositório.

A implementação atual do Ouriço suporta projetos que utilizam Subversion e Maven, mas pode ser estendida para outros SCV e SGC. O Ouriço possui interfaces para os SCV e os SGC. Portanto, para expandi-lo será necessário um esforço de implementação para automatizar as tarefas do SGC e do SCV, como compilação, execução de testes, criação de *autobranches*, realização de *check-in*, entre outros.

CAPÍTULO 6 – AVALIAÇÃO EXPERIMENTAL DO OURIÇO

6.1 INTRODUÇÃO

Como visto anteriormente, o Ouriço tem o objetivo de auxiliar os desenvolvedores a manterem um repositório íntegro, com pouca influencia no ciclo de trabalho do desenvolvedor. Para isso, são executadas verificações cujo objetivo é identificar artefatos que possam gerar conflitos físicos, sintáticos e/ou semânticos. Contudo, essas verificações podem levar um longo tempo para serem concluídas e, deste modo, resultar em atrasos no ciclo de desenvolvimento. Portanto, neste capítulo são descritos experimentos realizados com o intuito de avaliar:

1. A eficácia do Ouriço na identificação de defeitos
2. A influência do Ouriço no ciclo de desenvolvimento do projeto.

O experimento foi conduzido em um ambiente controlado e, portanto, é caracterizado como um experimento *in-vitro* (Travassos and Barros 2003). O experimento do Ouriço foi realizado a partir da exploração de repositórios Subversion que armazenam projetos que seguem a estrutura definida pelo *Maven*, conforme as restrições da implementação discutida no Capítulo 5. O ambiente para realização dos experimentos foi composto por um computador com processador Intel® Core™ Quad CPU Q9400, memória RAM de 4096 MB e Sistema Operacional Ubuntu 10.04. Neste ambiente estavam em execução o Oceano e o Ouriço, além de outros softwares necessários para execução dos projetos.

A decisão de realizar os experimentos em um ambiente controlado está ligada à facilidade para obtenção de repositórios de projetos *open-source*, comparando à dificuldade de inserção desta abordagem em uma instituição de desenvolvimento de software. Para adquirir os repositórios desses projetos foram encontradas algumas dificuldades, devido às limitações da abordagem.

No restante deste capítulo são discutidas as etapas para obtenção de repositórios, execução e aquisição de resultados dos experimentos conduzidos. Tal Capítulo está dividido da seguinte forma: Na Seção 6.2 são explicados os meios para obtenção de repositórios utilizados nos experimentos; Na Seção 6.3 são descritos os projetos escolhidos para avaliar a abordagem; Na Seção 6.4 são descritos os objetivos do experimento; Na Seção 6.5 são abordados os passos para execução do experimento propriamente dito; Na Seção 6.6 são apresentados os resultados; na Seção 6.7 é feita a análise dos resultados; e, finalmente, na Seção 6.8 são feitas as considerações finais, fechando esse capítulo.

6.2 OBTENÇÃO DE REPOSITÓRIOS

Para obter um repositório remoto do Subversion, tradicionalmente é utilizado o comando *svnsync* (Collins-Sussman et al. 2004, Berlin and Rooney 2006). Esse comando é capaz de copiar todo o histórico de desenvolvimento de um repositório remoto para um repositório local. Quando esse comando é concluído, o repositório local fica disponível para que outras tarefas sejam executadas com o seu conteúdo, sem que as modificações afetem o repositório remoto.

A sincronização dos repositórios é dividida em duas etapas: (1) inicialização e (2) transferência de dados do repositório remoto para o repositório local. Para execução da primeira etapa foi utilizado o comando *svnsync* com a opção *initialize* ou *init* (Figura 61). Tal comando realiza a ligação dos repositórios, da origem (<https://svn.codehaus.org/mojo/trunk/mojo/>) para o destino (`file:///home/mojo`), e copia as propriedades e conteúdo da revisão 0 para o repositório de destino.

```
$ svnsync initialize https://svn.codehaus.org/mojo/trunk/mojo/  
file:///home/mojo  
Copied properties for revision 0.
```

Figura 61. Comando *svnsync initialize*.

Para execução da segunda etapa, foi utilizado o comando *svnsync* com a opção *synchronize* ou simplesmente *sync* (Figura 62). Após realizar a ligação, no passo anterior, é necessário continuar com a obtenção do conteúdo do repositório para que no final todas as configurações do repositório remoto estejam no repositório local. Para isso o comando *svnsync sync* é executado e todas as demais revisões são extraídas do repositório remoto e enviadas para o repositório local, que no final da execução deste comando se torna um clone do repositório remoto.

```
$ svnsync sync file:///home/mojo  
Committed revision 1.  
Copied properties for revision 1.  
Committed revision 2.  
Copied properties for revision 2.  
...  
Committed revision n.  
Copied properties for revision n.
```

Figura 62. Comando *svnsync sync*.

O comando *svnsync* é a forma mais tradicional e cômoda para obter repositórios Subversion completos, quando não se tem acesso remoto as instalações do repositório. Entretanto, nem sempre é possível utilizá-lo devido a limitações impostas pelos **servidores** que hospedam tais repositórios. Um exemplo dessas limitações acontece no repositório ASF da Apache, que identifica quando uma máquina está acessando o repositório durante um longo tempo e bloqueia a conexão.

Para repositórios que não podem ser obtidos através do *svnsync*, uma ferramenta, baseada na ferramenta de experimentação da abordagem de Murta et al. (2007), foi desenvolvida. Esta ferramenta executa *check-outs* sucessivos de um repositório de origem, remoto, e realiza *check-ins* em um repositório de destino, local, através da execução dos seguintes passos (Figura 63):

1. Exporta uma revisão do repositório de origem (remoto) para um espaço de trabalho temporário;
2. Executa *check-out* de uma revisão do repositório de destino (local) para o espaço de trabalho de estudo;
3. Combina o espaços de trabalho de estudo, obtido no passo 2, com o espaço de trabalho temporário, obtido no passos 1, aplicando o resultado no espaço de trabalho de estudo; e
4. Executa *check-in* do espaço de trabalho de estudo. É importante ressaltar que a mensagem do *check-in* é o número da versão correspondente à configuração enviada para o repositório, o que possibilita a recuperação do histórico posteriormente.

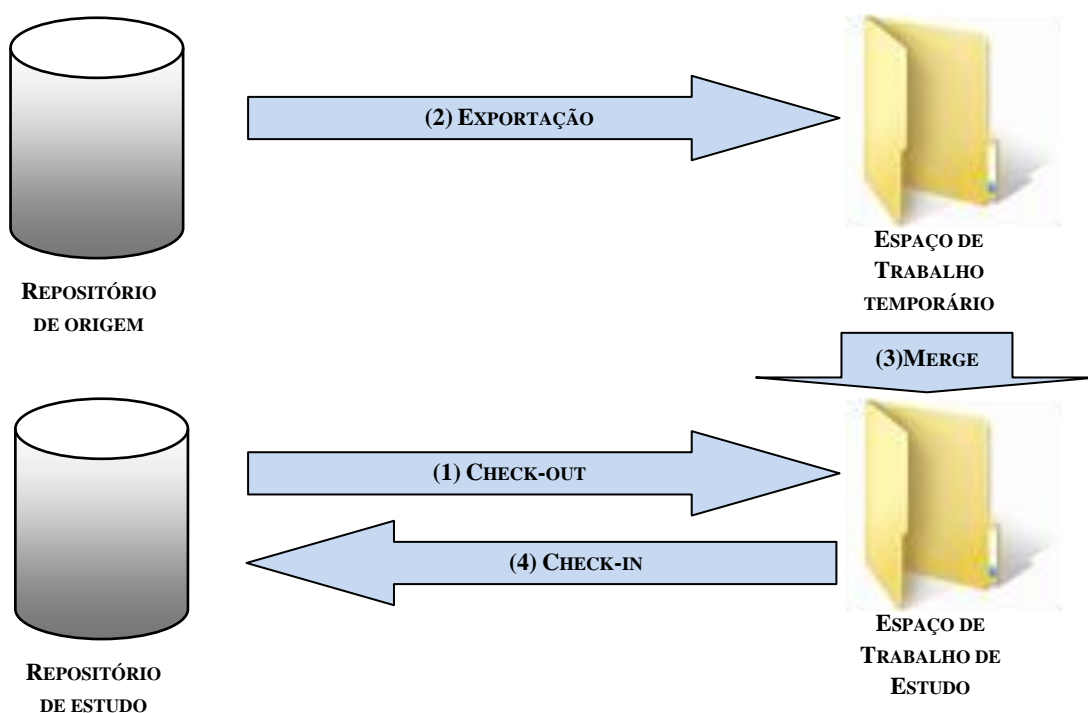


Figura 63. Ciclo para obtenção do repositório.

Os passos descritos na Figura 63 são aplicados em todas as revisões que impactam no projeto alvo. Para descobrir quais são estas revisões, o comando *svn log* (Figura 64) é executado com o endereço do projeto que está sendo copiado, passado como parâmetro. De posse do *log* gerado pelo comando anterior, um tratamento manual é feito para extrair todas as revisões resultantes deste comando. Com as revisões obtidas no passo anterior, o ciclo da Figura 63 é aplicado até que todas as revisões sejam transferidas para o repositório de estudo. O resultado dessa operação é um repositório que possui todas as revisões de um projeto alvo, mas não possui as propriedades de cada revisão como, por exemplo, data de *check-in*, mensagem, autor e outras propriedades.

```

$ svn log https://svn.apache.org/repos/asf/jakarta/bcel/trunk
-----
r1136412 | sebb | 2011-06-16 09:51:03 -0300 (Thu, 16 Jun 2011) | 1 line
Tab police
-----
r1136411 | sebb | 2011-06-16 09:50:55 -0300 (Thu, 16 Jun 2011) | 1 line
  
```

Figura 64. Comando de *log* do Subversion.

6.3 PROJETOS UTILIZADOS

Para realização dos experimentos foram selecionados projetos com perfis variados quanto a tamanho, número de desenvolvedores e objetivo da aplicação. Deste modo, chegou-se a um conjunto de projetos que são *plugins* de ferramentas utilizadas no mercado e, também, projetos independentes com portes diferenciados. Esses projetos estão caracterizados na Tabela 3, que foi montada com base em estatísticas extraídas com a ferramenta StatSVN⁹.

Tabela 3. Caracterização dos projetos.

Projeto	Desenvolvedores	Linhas de Código	Data Inicial (mês/ano)	Última Modificação ¹⁰ (mês/ano)
Native Maven Plugin	12	16.200	Setembro/2005	Janeiro/2011
SQL Maven Plugin	12	3.400	Setembro/2006	Maio/2011
BCEL	13	88.100	Outubro/2001	Junho/2011
Checkstyle ¹¹	12	213.620	Outubro/2001	Junho/2011

Os repositórios utilizados nesse experimento foram obtidos através do comando *svnsync*, que faz uma cópia de todo histórico de desenvolvimento de um projeto. Dessas cópias nem todas as configuração são utilizadas, pois durante o ciclo de vida de um projeto, várias tecnologias e ferramentas podem ter sido utilizadas. Portanto, como o Ouriço assume que algumas condições sejam satisfeitas pelo projeto alvo, uma etapa de seleção de configurações é executada. Essa verificação observa se: (1) o repositório do projeto é o *Subversion*, (2) o projeto utiliza o SGC *Maven*, (3) utiliza a linguagem Java e (4) possui conjuntos de testes, para aplicação da verificação semântica.

Na primeira etapa foi realizada uma busca por projetos *open source* que fossem implementados em Java. Durante a execução dessa etapa foram identificados diversos projetos que atendiam a esta restrição, mas que ainda precisariam passar pelos outros critérios para serem utilizados como objetos deste experimento.

⁹ <http://www.statsvn.org/index.html>.

¹⁰ A coluna “última modificação” na Tabela 3 representa a última modificação feita até a coleta do repositório.

¹¹ O repositório do projeto Checkstyle não permitiu a aplicação da ferramenta StatSVN, devido a problemas para obtenção do log. Portanto, Suas informações foram obtidas através do site <http://www.ohloh.net/p/checkstyle> e de buscas no repositório.

De posse destes projetos, foi realizada uma segunda etapa de análise, que visava identificar quais projetos estavam armazenados em repositórios Subversion. Desta etapa é importante ressaltar que o repositório ASF da Apache possui uma série de projetos que atendem o perfil desejado pela abordagem Ouriço. Além do ASF, existem outros repositórios como o da *codehaus*, que possuem uma série de *plugins* do Maven, e o repositório Subversion do *CheckStyle*.

Destes repositórios foram encontrados diversos projetos que utilizavam o Maven ou não durante o seu período de desenvolvimento. Conforme discutido anteriormente, o *Maven* é um SGC que possui uma estrutura própria que possibilita a construção de projetos através de um comando. Os projetos que seguem tal estrutura são caracterizados pela presença de um arquivo denominado *pom.xml*. Portanto, de posse dos projetos, foi realizada uma seleção manual de todas as revisões dos repositórios que possuem tal arquivo e, conseqüentemente, podem ser utilizadas como objeto do experimento da abordagem Ouriço.

Finalmente, uma etapa para identificar projetos que possuíam conjuntos de teste foi realizada. Esta etapa consistiu do *check-out* de revisões aleatórias do repositório e execução da tarefa de construção do projeto. Durante a construção foram identificados projetos que possuíam testes através das saídas geradas pelo *Maven*.

Essa fase de seleção das configurações é uma etapa manual que pode demandar muito esforço. Encontrar projetos com o perfil definido pela abordagem Ouriço é uma tarefa que pode demandar bastante tempo, devido às restrições que foram impostas pela implementação atual da abordagem. Esta dificuldade foi aumentada devido à análise manual dos projetos para que posteriormente a abordagem Ouriço fosse aplicada sobre as configurações resultantes. No final dessa busca foram obtidos os projetos representados na Tabela 3.

O primeiro dos projetos (Tabela 3) é o *Native Maven Plugin*, um *plugin* do Maven responsável por realizar construção de projetos que utilizam as linguagens C e C++. Este projeto possui uma equipe de desenvolvimento composta por 12 desenvolvedores e aproximadamente 16.200 linhas de código. Esse projeto foi retirado do repositório da *codehaus* armazenado no seguinte endereço: <https://svn.codehaus.org/mojo/trunk/mojo/maven-native>. Outra informação relevante é que o repositório do projeto foi criado em setembro de 2005 e teve o último *check-in* em janeiro de 2011.

O segundo projeto é o *SQL Maven Plugin*, outro *plugin* do *Maven* responsável por executar instruções SQL. Tal projeto possui cerca de 3.400 linhas de código e está sendo desenvolvido por 12 desenvolvedores que trabalham em seus 25 arquivos. Esse projeto foi

retirado do repositório da *codehaus* armazenado no seguinte endereço: <https://svn.codehaus.org/mojo/trunk/mojo/sql-maven-plugin>. O projeto em questão teve o seu primeiro *check-in* em setembro de 2006 e teve o último em maio de 2011.

O terceiro projeto é o BCEL (*Byte Code Engineering Library*), que visa dar aos usuários a possibilidade de criar e manipular arquivos binários da linguagem Java. Este projeto possui cerca de 88.100 linhas de código, mas já teve um pico de cerca de 110.000 linhas e possui uma equipe formada por 13 desenvolvedores. Esse projeto foi retirado do repositório ASF, da *apache*, que está armazenado no seguinte endereço: <https://svn.apache.org/repos/asf/jakarta/bcel/trunk>. Esse projeto teve início em outubro de 2001 e seu último *check-in* foi realizado em junho de 2011.

O quarto projeto é o *Checkstyle*, uma ferramenta utilizada para realizar verificações estáticas em artefatos de software que contém 213.620 linhas de código. O repositório que abriga este projeto começou a ter registros a partir de junho de 2001 e o último *check-in* ocorreu em setembro de 2010. Atualmente, o projeto *Checkstyle* foi migrado para um repositório Mercurial e, portanto, nos experimentos executados no Ouriço revisões do novo SCV foram descartadas. O repositório de onde foram extraídas as revisões analisadas está no seguinte endereço: <https://checkstyle.svn.sourceforge.net/svnroot/checkstyle/trunk>. As demais informações como, por exemplo, número de desenvolvedores e outros não puderam ser extraídas via StatSVN, devido a uma falha para obtenção do *log*. Portanto, foi feita a coleta de dados diretamente do site do projeto.

6.4 OBJETIVOS

Conforme descrito anteriormente, este experimento visa obter indicadores relacionados a (1) eficácia do Ouriço na identificação de defeitos e (2) se a utilização do Ouriço influencia no ciclo de desenvolvimento do projeto sob análise. Para avaliar esses dois aspectos foram extraídas métricas que indicam a quantidade de defeitos identificados nos repositórios dos projetos, descritos na Seção 6.3, e o tempo que foi necessário para realização das verificações.

A eficácia do Ouriço na identificação de defeitos é medida através do número de configurações que o Ouriço é capaz de identificar como quebradas, seja sintática ou semanticamente. Portanto, um dos objetivos deste experimento é obter o número de revisões quebradas que a abordagem Ouriço conseguiria ter identificado se estivesse em uso durante o desenvolvimento do projeto em questão, assim como o tempo que este permaneceria inconsistente.

Outro objetivo deste experimento é verificar se a aplicação da abordagem influencia no ciclo de trabalho de desenvolvimento de software. Para verificar esse fator, o tempo gasto para fazer a verificação de cada revisão foi coletado e comparado com o intervalo de realização dos *check-ins*. Deste modo, é possível identificar o número de *check-in* que seriam postergados devido à verificação do Ouriço.

6.5 EXECUÇÃO DOS EXPERIMENTOS

A obtenção dos repositórios é uma fase que foi adicionada para reduzir as interferências externas, como a de rede, no momento de execução dos experimentos. Conforme discutido no Capítulo 4, o Ouriço trabalha sobre um repositório para que linhas de desenvolvimento possam ser protegidas e, por conseguinte, a consistência do repositório possa ser mantida. Essa consistência é alcançada através de verificações, em dois níveis, das configurações que são enviadas para o repositório via comando de *check-in*. Portanto, o experimento do Ouriço baseia-se na execução do protótipo do Ouriço sobre todas as revisões do repositório sob estudo que atendam às restrições impostas por esta abordagem.

A aplicação da abordagem sobre as revisões selecionadas foi realizada através de uma ferramenta similar à de obtenção de repositórios (Figura 63). Para avaliar a abordagem Ouriço foi necessário:

- Repositório de Origem, que contém as revisões do projeto a ser verificado pela abordagem Ouriço;
- Repositório de Estudo, utilizado para criação dos *autobranches* e reprodução do ciclo de desenvolvimento do repositório de origem.

Como em alguns casos o repositório de origem não possui o histórico de desenvolvimento, por ser copiado na base de *check-out* e *check-in*, uma ferramenta foi desenvolvida para automatizar a obtenção desses dados. Conforme discutido anteriormente, a ferramenta de obtenção de repositório passa como mensagem para o *check-in* a revisão original da configuração no repositório remoto. Essa informação é utilizada para recuperar o histórico de desenvolvimento do projeto.

A ferramenta criada é capaz de recuperar dados do banco de dados do Oceano\Ouriço, do repositório remoto e agrupá-los resultando em uma tabela que posteriormente é tratada para extração de informações mais estruturadas. Durante o tratamento dos resultados do Ouriço a ferramenta em questão realiza os seguintes passos (Figura 65): (1) recuperar os tempos utilizados para tratar cada *autobranch*, nas verificações de primeiro e segundo nível e, também, o tempo total, que inclui a integração; (2) recuperar do repositório o momento de

check-in de cada configuração; e, finalmente, (3) tratar os dados, obtidos nos passos 1 e 2, para fornecer informações como a diferença entre o tempo de verificação e o intervalo entre *check-ins*. Esses passos são detalhados a seguir.

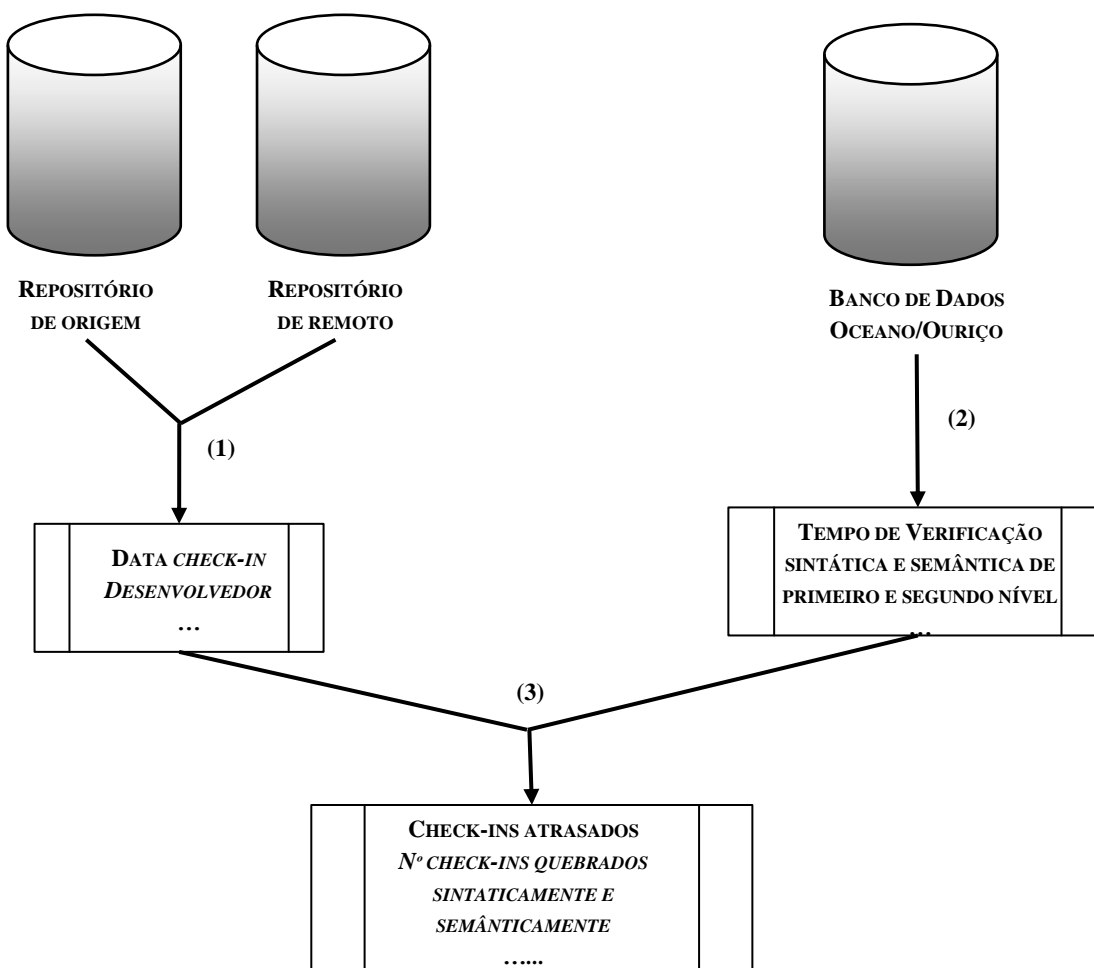


Figura 65. Ciclo base para extração de dados do experimento.

A partir dos passos anteriores foi possível obter o número de *check-ins* que sofreram atraso por causa da utilização do Ouriço e também a quantidade de *check-ins* sintaticamente e semanticamente quebrados que seriam identificados pela abordagem proposta. Na Figura 66 é ilustrado o cenário onde é possível obter indicadores do primeiro caso, listado anteriormente. Nessa figura é ilustrada, no eixo 1, a linha de desenvolvimento real do projeto e nos eixos restantes, de 2 a 6, o tempo de verificação de cada configuração. De posse desse gráfico é possível identificar que dois *check-ins* sofreram atraso devido ao tempo de verificação exceder o intervalo entre dois *check-ins* consecutivos. É importante ressaltar que na ferramenta desenvolvida o apoio gráfico não foi implementado. Ele foi incluído nesse documento para facilitar o entendimento.

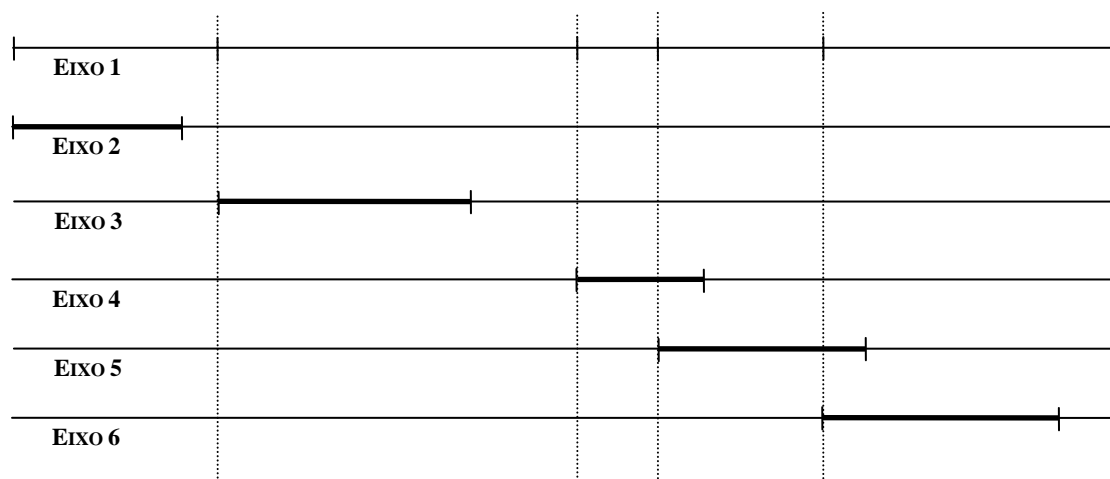


Figura 66. Check-ins atrasados.

6.6 RESULTADOS

Foram realizados experimentos em quatro projetos, descritos na Seção 6.2, e seus resultados foram organizados segundo os objetivos destes experimentos. Conforme discutido anteriormente, os objetivos deste experimento estão ligados à avaliação de: (1) a eficácia do Ouriço na identificação de defeitos e (2) se a sua utilização influencia no ciclo de desenvolvimento do projeto.

Na Tabela 4 estão os resultados dos experimentos levando em conta todo o período de desenvolvimento das configurações dos projetos analisados e, deste período, quanto tempo o projeto esteve inconsistente ou consistente. Tal tabela está organizada com os seguintes campos:

- Projeto, que identifica o projeto cujos resultados pertencem;
- Períodos de Desenvolvimento, que representa quantas horas do projeto foram analisadas nas revisões selecionadas para um projeto;
- Período Inconsistente, que relata durante quantas horas o repositório esteve sintaticamente ou semanticamente inconsistente;
- Período Consistente, que identifica o intervalo de tempo que o repositório esteve consistente, ou seja, com ausência de defeitos.

É importante ressaltar que o período inconsistente e o período consistente são a soma dos intervalos que um projeto permaneceu em um determinado estado (i.e., consistente ou inconsistente). Portanto, os valores descritos não são contínuos.

Na Tabela 4 ainda é possível ver os resultados em porcentagem, ou seja, a porcentagem do período de desenvolvimento que o repositório esteve inconsistente ou não. A extração das porcentagens foi realizada da seguinte forma:

- Período Sintaticamente Inconsistente (%) = Período Sintaticamente Inconsistente (horas) / Período de Desenvolvimento (horas);
- Período Semanticamente Inconsistente (%) = Período Semanticamente Inconsistente (horas) / Período de Desenvolvimento (horas);
- Período Consistente (%) = Período Consistente (horas) / Período de Desenvolvimento (horas).

Tabela 4. Resultados dos experimentos levando em consideração o tempo.

Projeto	Período de desenvolvimento em horas	Período Inconsistente em horas		Período consistente em horas
		Sintaticamente	Semanticamente	
BCEL	6.951,96	664,25 (9,5%)	0 (0%)	6.287,70 (90,5%)
SQL Maven Plugin	44.386,16	4.510,90 (10,16%)	426,97 (0,96%)	39.448,28 (88,88%)
Native Maven Plugin	48.656,44	35.190,81 (72,33%)	3,02 (0%)	13.462,61 (27,67%)
Checkstyle	33.026,03	312,76 (0,95%)	0 (0%)	32.713,28 (99,05%)

Na Tabela 5 os resultados estão organizados levando-se em conta o número de revisões que foram analisadas em cada projeto. Nessa tabela é possível identificar quantas revisões foram analisadas e o número de revisões que apresentam inconsistência seja ela sintática ou semântica. A Tabela 5 está organizada da seguinte forma:

- Projeto, que identifica o projeto cujos resultados pertencem;
- Configurações estudadas, que indica o número de revisões que foram analisadas durante o experimento, para cada projeto;
- *Check-ins* Inconsistentes, que se refere ao número que *check-in* quebrados que foram enviados para o repositório, resultando em um repositório sintaticamente ou semanticamente inconsistente.
- *Check-ins* Consistentes, que se refere ao número de *check-ins* enviados para o repositório sem que nenhuma quebra tenha sido identificada.

Tabela 5. Resultados do experimento levando em consideração as configurações.

Projeto	Configurações Estudadas	Check-ins Inconsistentes		Check-ins Consistentes
		Sintaticamente	Semanticamente	
BCEL	30 ¹²	3 (10%)	0 (0%)	27 (90%)
SQL Maven Plugin	135	20 (14,81%)	8 (5,93%)	107 (79,26)
Native Maven Plugin	298	226 (75,84%)	3 (1,01%)	69 (23,15%)
Checkstyle	309	43 (13,92%)	0 (0%)	266 (86,08%)

Além dos valores inteiros na Tabela 5, é possível identificar a porcentagem de revisões que possui uma característica como, por exemplo, estar inconsistente. Esses valores foram extraídos da maneira descrita a seguir:

- $Check-ins \text{ Sintaticamente Inconsistente } (\%) = \frac{Check-ins \text{ Sintaticamente Inconsistente}}{Configurações \text{ Estudadas}}$;
- $Check-ins \text{ Semanticamente Inconsistente } (\%) = \frac{Check-ins \text{ Semanticamente Inconsistente}}{Configurações \text{ Estudadas}}$;
- $Check-ins \text{ Consistentes } (\%) = \frac{Check-ins \text{ Consistentes}}{Configurações \text{ Estudadas}}$.

Para responder se a utilização do Ouriço influencia no ciclo de desenvolvimento de um projeto, outras variáveis do experimento foram coletadas. Na Tabela 6 é possível observar médias e desvios padrão (DP) do tempo gasto pelo Ouriço nas verificações dos projetos em três situações. A primeira situação ocorre quando a revisão está sintaticamente quebrada, pois a configuração não compila. Nesta situação o tempo médio de execução é sempre menor que nas demais. A segunda situação ocorre quando uma configuração não passa pelo conjunto de testes, que verificam as regras de negócio do projeto em questão. Esta situação, no geral, apresenta um tempo médio superior ao da primeira, mas inferior ao da terceira situação. E, finalmente, a terceira situação acontece quando o projeto é verificado em todos os níveis e a configuração é incorporada o repositório de estudo.

¹² O BCEL possui poucas revisões estudadas devido à sua migração recente para a estrutura Maven.

Tabela 6. Informações do tempo de execução das verificações.

Projeto	Tempo de Verificação em segundos					
	Sintaticamente Quebrado		Semanticamente Quebrado		Check-in Realizado com Sucesso	
	Média	DP	Média	DP	Média	DP
BCEL	4,68	5,04	-	-	28,17	48,14
SQL Maven Plugin	0,37	0,88	2,89	1,40	9,85	1,95
Native Maven Plugin	6,25	4,10	12,52	0,57	45,78	13,71
Checkstyle	0,65	0,58	-	-	30,12	21,21

Para complementar as informações da Tabela 6, a Tabela 7 mostra dados que possibilitam, de fato, saber quantos *check-ins* seriam atrasados devido à utilização da abordagem. Na Tabela 7 é possível visualizar dados como:

- Tempo médio entre *check-ins*, que mostra o intervalo médio que o repositório recebe contribuições de desenvolvedores. Além disso, é possível visualizar, também, o desvio padrão para melhor entendimento dos resultados.
- Número de *check-ins* sintaticamente quebrados, que atrasaram algum *check-in*.
- Número de *check-ins* semanticamente quebrados, que atrasaram algum *check-in*.
- Número de *check-ins* concluídos com sucesso, que atrasaram algum *check-in*. É importante ressaltar que esse caso, em especial, é o que deve ser minimizado, pois representa a criação de uma nova configuração válida para os demais membros da equipe de desenvolvimento.

Tabela 7. Informações sobre os *check-ins*.

Projeto	Intervalo entre <i>Check-ins</i> em segundos		<i>Check-ins</i> Atrasados – CA		
	Média	D.P.	Sintaticamente quebrado	Semanticamente quebrado	Integrado com Sucesso
BCEL	834.235,22 (9,66 dias)	2.744.817,99 (32,12 dias)	0	0	2 (6,67%)
SQL Maven Plugin	1.183.631,00 (13,70 dias)	2.900.287,31 (33,57 dias)	0	0	0
Native Maven Plugin	587.795,94 (6,80 dias)	2.180.780,40 (25,24 dias)	1 (0,34%)	0	6 (2,01%)
Checkstyle	384.769,32 (4,45 dias)	1.136.792,81 (13,16 dias)	0	0	0

Estes resultados foram obtidos da ferramenta de obtenção de dados, discutida anteriormente, e de uma etapa posterior para estruturação dos resultados. A ferramenta implementada para extração de dados fornece dados brutos que posteriormente foram tratados para que uma informação mais estruturada, ou seja, médias e desvio padrão, fossem obtidas. Esse tratamento foi feito através de ferramentas como Excel e o Calc do OpenOffice.

6.7 ANÁLISE DE RESULTADOS

Com base nos resultados apresentados na Seção 6.6 uma análise dos objetivos destacados na Seção 6.4 pôde ser realizada. Foram utilizados, no total, quatro projetos que possuem características distintas como o número de revisões analisadas, número de desenvolvedores, número de linhas e objetivos.

6.7.1 EFICÁCIA DO OURIÇO NA IDENTIFICAÇÃO DE DEFEITOS

De acordo com a Tabela 4, é possível identificar que todos os projetos analisados possuem um período em que o repositório ficou inconsistente. No primeiro projeto, BCEL, o repositório permanece inconsistente durante o período de 664,25 horas, ou seja, durante 9,5% das horas de desenvolvimento analisadas no projeto. Portanto, é possível concluir que o desenvolvedor que realizar *check-out* durante essas 664,25 horas deve corrigir erros antes de prosseguir com as tarefas planejadas. Algo similar acontece com o projeto *SQL Maven Plugin* cujo seu repositório permanece inconsistente durante 10,16% do tempo, equivalente a 4.510,90 horas de desenvolvimento do projeto. Dos projetos utilizados nos experimentos do Ouriço, estes dois representam o caso médio, considerando o tempo como parâmetro para avaliação. Vale ressaltar que os tempos discutidos nessa seção representam o somatório de todos os intervalos, ou seja, o BCEL, por exemplo, não ficou durante 664,25 horas consecutivas inconsistente.

Além disso, de acordo com a Tabela 4, é possível identificar que o projeto *Checkstyle* apresentou o melhor desempenho dos projetos analisados. Foram coletadas informações do *Checkstyle* referente a 33.026,03 horas de desenvolvimento e apenas 312,76 horas apresentam falha sintática, o que representa 0,95% do tempo de desenvolvimento. Portanto, esse projeto possui uma característica de estar na maioria do tempo com o repositório consistente. A manutenção da consistência do repositório do *Checkstyle* pode ser atribuída à utilização de IC para verificar o desenvolvimento do projeto em questão. Conforme discutido no site¹³ do

¹³ <http://checkstyle.sourceforge.net/integration.html>

Checkstyle, o Hudson é utilizado para verificar a configuração resultante no repositório e notificar o desenvolvedor em caso de falha.

Finalmente, por meio da Tabela 4 é possível identificar que o *Native Maven Plugin* representa o pior caso dentre os projetos analisados. Os artefatos analisados deste projeto correspondem a um período de desenvolvimento igual a 48.656,44 horas, algo em torno de 5 anos e meio de desenvolvimento. Durante esse período, o projeto permaneceu quebrado, sintaticamente ou semanticamente, por 35.193,83 horas, algo em torno de 72,33% do período de desenvolvimento do projeto. Através do resultado apresentado é possível inferir que um desenvolvedor que atua neste tipo de projeto quase não possui a chance de trabalhar, pois durante aproximadamente 4 anos de projeto o desenvolvedor necessita realizar correções antes de iniciar as suas tarefas. Esses 4 anos representam o somatório de todos os períodos de inconsistência, ou seja, o projeto não ficou quebrado durante 4 anos consecutivos. Tal valor é bastante elevado, mesmo considerando situações onde são esperados *check-ins* quebrados como: em refatorações grandes e definição da arquitetura inicial.

Para dar mais suporte aos resultados apresentados para o projeto *Native Maven Plugin*, foi realizada uma busca no site de desenvolvimento deste *plugin* e foi constatado que este projeto utiliza IC. Esta informação gerou uma confusão inicial, no sentido de como um projeto pode utilizar IC e apresentar uma taxa tão elevada de *check-ins* quebrados. Com esta dúvida e uma análise mais profunda foi possível identificar que a partir da 223ª configuração analisada o número de *check-ins* quebrados foi bastante reduzido, ou seja, foram realizados apenas 6 *check-ins* quebrados em um total de 75. Este resultado significa uma queda superior a 9 vezes, caindo de 72,33% para apenas 8%, que provavelmente foi obtido através da implantação de IC. É válido ressaltar que no site deste projeto não existe a informação de quando foi implantado IC e, portanto, não é possível afirmar com certeza o período de implantação, pois foi baseado apenas em inferências a partir dos resultados obtidos.

Uma análise similar a da Tabela 4 é apresentada na Tabela 5, mas com o parâmetro número de configurações sendo utilizados para efeito de comparação. A utilização de configurações no lugar de tempo dá uma ideia menos temporal, dado que os *check-ins* não possuem necessariamente uma simetria de tempo. Sendo assim, através dessa análise é possível identificar quantas vezes um desenvolvedor enviou uma configuração que possui alguma inconsistência para o repositório e, consequentemente, quantas vezes o Ouriço rejeitaria uma contribuição, por julgá-la inconsistente.

Analisando desta perspectiva, número de *check-ins*, é possível notar que o projeto BCEL é o que possui o melhor desempenho. No projeto BCEL foram encontrados apenas três

casos em que ocorreram quebras sintáticas, o que representa 10% dos casos analisados. Os resultados deste projeto não servem para caracterizar o projeto como um todo, devido à sua pequena amostra (30 revisões). Entretanto, possui indícios de que poucos *check-ins* com artefatos quebrados são enviados para o repositório, dado que 90% dos *check-ins* possuem artefatos livres de quebras.

Outro resultado interessante é que quando essa inversão de perspectiva é feita o *Checkstyle* passa a ter um desempenho mediano, semelhante ao do *SQL Maven Plugin*. Com base nesta afirmação é possível inferir que uma das prioridades no projeto *Checkstyle* é a remoção de um defeito quando ele é identificado, algo que se assemelha às práticas de IC. Conforme discutido anteriormente, o *Checkstyle* utiliza o Hudson como servidor de IC e embora não exista nenhuma informação no site de desenvolvimento desse projeto, é possível inferir que existe uma política para remoção de defeitos do repositório. Remoção esta que representa uma das práticas da IC.

Ainda com relação à Tabela 5, é possível identificar que o *Native Maven Plugin* continua tendo o pior desempenho dos projetos analisados. Este resultado demonstra, mais uma vez, a necessidade da aplicação de abordagens para contenção de artefatos quebrados, evitando, deste modo, que o repositório possua artefatos inconsistentes, dado que 75,28% das revisões presentes no repositório apresentam artefatos sintaticamente ou semanticamente quebrados.

6.7.2 A INFLUÊNCIA DO OURIÇO NO CICLO DE DESENVOLVIMENTO DO PROJETO

Conforme descrito na Seção 6.7.1, o Ouriço apresentou um desempenho favorável na identificação de erros e poderia ser adotado para apoiar a manutenção da consistência de repositórios. Entretanto, a análise feita até então não teve como propósito verificar se o Ouriço gera algum efeito colateral no ciclo de desenvolvimento do projeto, atrasando *check-ins* consistentes. Para verificar esse aspecto, é necessária a análise da Tabela 6 e da Tabela 7.

Na Tabela 6 é possível identificar o tempo de verificação dos projetos em três situações diferentes. A primeira situação acontece quando a revisão está sintaticamente quebrada, ou seja, o *Maven* não pode executar a tarefa de compilação devido a algum defeito presente em pelo menos um de seus artefatos. Nesta situação é possível citar o exemplo do BCEL que gasta cerca de 4,68 segundos para ser verificado, contudo um desvio padrão bem elevado pode ser notado. Esse desvio padrão ocorre devido a diferenças no trato dos artefatos pelo compilador e também devido à evolução do projeto de configuração para configuração. A

segunda situação ocorre quando uma revisão é compilada, mas não passa pelo conjunto de testes propostos para ela. O *SQL Maven Plugin*, nesta situação, executa tal tarefa em aproximadamente 2,8 segundos com o desvio padrão de 1,4 segundos. Finalmente, a terceira situação ocorre quando o *check-in* é integrado ao repositório de estudo com sucesso, resultando na execução completa do ciclo do Ouriço e, portanto, em um tempo elevado de execução. Este tipo de verificação é realizada com tempo médio de 30,12 segundos considerando o *Checkstyle*, que possui um desvio padrão de 21,21 segundos. Este desvio padrão acontece devido às revisões serem constantemente incrementadas, através do crescimento natural do software, acompanhado pelo aumento do número de testes, resultando em tempos distintos para execução da mesma tarefa.

Os dados da Tabela 6 demonstram que a verificação do Ouriço possui um desempenho bom, mas este desempenho não foi confrontado com uma situação real de desenvolvimento. Conforme discutido anteriormente, durante o *check-out* o projeto é construído para que durante as verificações do *check-in* somente os artefatos modificados sejam construídos, elevando o desempenho do Ouriço na verificação dos artefatos. Embora o desempenho seja alavancado, devido a este artifício do Ouriço, uma comparação com o contexto real de desenvolvimento precisa ser realizado. Esta comparação é possível através da Tabela 7.

Na Tabela 7 é possível identificar o intervalo médio entre *check-ins* de um determinado projeto e também quantos *check-ins* foram atrasados devido à implantação do Ouriço no contexto de desenvolvimento. Conforme indicado, os projetos têm intervalos médios entre *check-ins* de 4,45 a 13,7 dias, algo que está fora de comparação com a realidade dos tempos de verificação da abordagem Ouriço, apresentados na faixa de segundos. Entretanto, através da Tabela 7 é possível identificar que os atrasos ocorrem devido à alta dispersão dos tempos em torno da média, ou seja, o desvio padrão elevado.

Os atrasos resultantes da implantação da abordagem Ouriço não representariam um conjunto predominante. Dos 4 projetos analisado, apenas dois, BCEL e *Native Maven Plugin*, sofreram atrasos devido ao tempo de verificação superar o intervalo entre os *check-ins*. No BCEL foram encontrados 2 atrasos que ocorreram quando o ciclo foi concluído com sucesso, situação que demanda maior tempo de verificação. Estes 2 *check-ins* atrasados representam 6,67% dos *check-ins* analisados para este projeto. Tal atraso foi gerado devido a uma diminuição do intervalo de *check-in*, acompanhado de um aumento do tempo de verificação (e.g., no caso do BCEL uma verificação demorou 265 segundos, enquanto o intervalo entre dois *check-ins* sucessivos específicos foi de 97 segundos). Com uma relevância menor, considerando-se o número de revisões analisadas, o *Native Maven Plugin* sofreu atraso em 7

revisões que representa 2,35% das revisões deste projeto. Esses atrasos foram gerados devido a uma brusca redução dos intervalos de *check-in*, que nesses casos não foram acompanhados pelo aumento do tempo de verificação.

O número de atrasos é considerado baixo, pois quando comparado ao número que *check-ins* que foram verificados, representam apenas 1,17%. Foram identificados 9 atrasos (dois do BCEL e 7 do *Native Maven Plugin*) dentre 772 *check-ins* verificados. Portanto, a abordagem Ouriço, para o espaço amostral analisado, atrasa o ciclo de trabalho do desenvolvedor em 1,17% dos casos estudados. É importante ressaltar que o atraso está sempre na casa dos segundos resultando na possibilidade de nem ser percebido pelo desenvolvedor que está utilizando a abordagem.

6.8 CONSIDERAÇÕES FINAIS

O experimento do Ouriço foi baseado em dois pilares: (1) se a abordagem realmente impede a entrada de artefatos quebrados no repositório e (2) se a utilização da abordagem influencia no ciclo de desenvolvimento de um projeto. Os estudos realizados mostraram que o Ouriço é capaz de identificar configurações que apresentam artefatos quebrados em 39,25% dos casos estudados, sendo que existem projetos com um pico de até 76,85% das configurações inconsistentes. Outro fator relevante é que a abordagem Ouriço possui uma influência insignificante sobre o ciclo de desenvolvimento, pois apenas 1,17% das configurações estudadas sofreram atraso quando comparadas com o ciclo de desenvolvimento real, com pico de 6,67% no projeto BCEL.

Durante a seleção dos projetos e a execução dos experimentos foram identificados alguns desafios que impediram que este experimento fosse aplicado em projetos maiores. Muitos dos projetos disponibilizados em repositórios remotos e que têm livre acesso possuem passos de configuração manual devido a, por exemplo, bibliotecas que não podem ser obtidas ou não estão disponíveis em repositórios do *Maven*. Este fator torna a execução do experimento inviável devido à necessidade de execução de passos manuais para realização do experimento em todas as configurações. Portanto, projetos com esse perfil foram descartados.

A utilização do Ouriço em um ambiente real de desenvolvimento ainda não foi realizada, mas existe a expectativa que em breve este experimento possa ser realizado. Uma experimentação em um ambiente real de desenvolvimento auxiliaria na obtenção de resultados em um contexto real e também no amadurecimento da ferramenta. Esse amadurecimento pode ser atingido, pois esta implementação pode conter limitações perceptíveis apenas em ambientes de desenvolvimento real.

Com os experimentos descritos nesse capítulo foi possível responder as questões propostas, entretanto não é possível generalizar os resultados devido a algumas limitações que podem ameaçar os estudos realizados. A primeira ameaça é o número de configurações analisadas durante os experimentos, que apresenta sempre um número bem reduzido, ou seja, algo em torno de 300 configurações. A segunda limitação está relacionada à falta de informação disponibilizada no site dos projetos utilizados, que não permite confirmar inferências obtidas a partir dos experimentos. A terceira limitação está relacionada à infraestrutura utilizada para executar os experimentos, pois não é possível garantir que o ambiente se comportou de maneira estável. Um exemplo é se a capacidade de *download* da rede manteve estável, pois esta característica tem impacto direto no tempo de verificação.

CAPÍTULO 7 – CONCLUSÃO

7.1 CONTRIBUIÇÕES

Este trabalho apresentou a definição de uma abordagem para a manutenção de consistência de repositórios de GC que foi chamada de Ouriço. Para isso foram realizados passos para embasamento teórico, definição, implementação e análise da abordagem. Portanto, as principais contribuições deste trabalho são:

- Definição e implementação do Ouriço – abordagem para manutenção da consistência de repositórios de GC;
- Definição e implementação de um ciclo de trabalho de GC que realiza verificações em artefatos que são candidatos a entrarem no repositório, sem atrasar o ciclo de desenvolvimento realizado pelo desenvolvedor;
- Definição e implementação de um painel de controle onde é possível identificar os desenvolvedores que realizaram *check-out* e, também, em qual parte do ciclo de trabalho do Ouriço eles estão;
- Viabilização do projeto Oceano como plataforma para implementações acadêmicas relacionadas à GC.
- Experimentação do Ouriço em quatro projetos reais (*Native Maven Plugin*, *SQL Maven Plugin*, *BCEL* e *Checkstyle*).

7.2 LIMITAÇÕES

Apesar de o Ouriço atingir os seus objetivos, esta abordagem possui algumas limitações devido a requisitos que ficaram fora do Ouriço, embora contribuam para que a abordagem seja mais robusta. Estas limitações são discutidas a seguir.

7.2.1 TODOS PODEM REALIZAR *CHECK-IN*

Uma das limitações que a abordagem Ouriço possui é que todos os usuários com permissão de escrita no repositório de GC podem realizar o *check-in* tradicional, mesmo que não tenham a autorização via Ouriço. O ciclo de trabalho do Ouriço deve ser realizado através de suas ferramentas de *check-out*, *check-in* e *update*. Entretanto, quando um desenvolvedor possui acesso a um repositório os comandos de *check-out*, *check-in* e *update* tradicionais podem ser realizados sem que o Ouriço controle o conteúdo das configurações. Deste modo, o desenvolvedor pode enviar configurações para o repositório sem a realização das verificações propostas nessa abordagem.

Quando os artefatos não são verificados pelo Ouriço, nenhuma garantia pode ser dada ao conteúdo do repositório. Como o Ouriço realiza verificações sintáticas e semânticas nos projetos, ou linhas de desenvolvimento, presentes no repositório, a qualidade dos artefatos já foi verificada. Portanto, o desenvolvedor vai adquirir artefatos cuja qualidade está garantida pela abordagem Ouriço. Entretanto, quando o ciclo tradicional de GC é realizado, nenhuma garantia pode ser dada para a configuração obtida após o comando de *check-out*. É válido ressaltar que existem contextos onde essa liberdade é esperada. Um exemplo pode ser em casos onde uma *release* emergencial é necessária e, deste modo, o ciclo do Ouriço não pode ser executado, pois representaria um gargalo.

Uma maneira de controlar esse acesso ao repositório é criar somente um usuário que o Ouriço utilizará para realizar *check-in* no repositório em questão. Com a utilização de apenas um usuário, com permissão de escrita no repositório, todos os *check-ins* devem passar pelas verificações do Ouriço para serem integradas ao repositório. Ou seja, nenhum usuário além de, por exemplo, o gerente de projetos, teria acesso para escrever no repositório e, deste modo, apenas o Ouriço poderia realizar *check-ins*. Com essa restrição é possível garantir que o conteúdo do repositório foi verificado pelo Ouriço e, consequentemente, que o repositório estará em estado consistente.

7.2.2 EXPERIMENTOS NÃO LEVAM EM CONSIDERAÇÃO O DESENVOLVEDOR

Os experimentos realizados nesse trabalho não levaram em consideração a opinião dos desenvolvedores dos projetos analisados. Conforme descrito no Capítulo 6, o Ouriço foi executado em quatro projetos *open source* cujo acesso aos desenvolvedores não foi explorado. Portanto, não foi possível implantar a abordagem no ambiente de desenvolvimento de tais projetos e nem verificar a importância dos resultados obtidos para as equipes de desenvolvimento.

Com a execução de experimentos que tenham interação com os desenvolvedores seria possível identificar se a abordagem apresenta algum caráter intrusivo, pontos de melhoria na implementação atual, entre outras. Além disso, os resultados obtidos poderiam ser validados pelas equipes de desenvolvimento e possíveis melhorias poderiam ser incorporadas em futuras implementações do Ouriço.

Para contornar essa limitação serão propostos trabalhos futuros que incluem a execução do Ouriço em ambientes reais de desenvolvimentos ou em projetos desenvolvidos por alunos do GEMS. Esses experimentos vão levar em consideração aspectos de usabilidade e

identificação de limitações em diversas situações como, por exemplo, no caso criação de *releases* emergenciais.

7.2.3 A IDENTIFICAÇÃO DE QUEBRAS SEMÂNTICAS DEPENDE DOS TESTES CRIADOS PELO DESENVOLVEDOR

Conforme discutido, o Ouriço possui quatro opções de políticas que realizam verificações sintáticas e semânticas de artefatos de software na configuração do desenvolvedor e na configuração candidata. Das políticas do Ouriço, apenas as políticas restritiva e dinâmica podem realizar, por default, verificações semânticas, que são baseadas no conjunto de testes desenvolvidos durante o ciclo de desenvolvimento do projeto.

A qualidade das verificações da abordagem Ouriço está diretamente ligada à cobertura dos testes desenvolvidos. Tal abordagem não possui nenhum passo para criação de casos de teste, que são utilizados durante a verificação do Ouriço, e, portanto, essa é uma de suas limitações. Esta limitação pode ser determinante em diversas situações como, por exemplo, em casos onde a equipe de desenvolvimento possui tarefas com prioridade mais elevada que a escrita dos casos de teste como, por exemplo, correção de defeitos sem a manutenção dos testes e implementação de novas funcionalidades.

Tal limitação pode ser contornada com a inserção de abordagens que criem casos de teste para serem executados sobre a configuração do desenvolvedor e configuração candidata. Um exemplo desse tipo de abordagem é o Saferefactor (Soares et al. 2009), que cria casos de testes sobre uma configuração para verificar se uma refatoração foi realizada com sucesso. Para tanto, os seguintes passos são seguidos: (1) Identifica mudanças no software, (2) gera testes de unidade, (3) aplica os testes nas configurações original e alvo, (4) aplica as refatorações que não alteraram o comportamento do software.

7.3 TRABALHOS FUTUROS

O Ouriço possui diversas funcionalidades que foram desenvolvidas conforme o escopo definido para o projeto, entretanto outras funcionalidades foram identificadas sem que fossem implementadas nesta primeira versão. Essas funcionalidades identificadas são descritas nesta seção para que futuramente sejam implementadas e incorporadas ao Ouriço, resultando em maior grau de confiança sobre os artefatos presentes no repositório e, também, sobre os resultados apresentados por esta abordagem.

7.3.1 INSERÇÃO DE VERIFICAÇÃO ESTÁTICA

A verificação estática consiste na verificação de padrões sem que o código seja executado. Esta verificação pode realizar a análise de textos, inseridos nos artefatos de software; verificação do número de linhas por artefato de software de um projeto; e, também, verificação da qualidade da documentação gerada durante o processo de desenvolvimento de software. É importante ressaltar que apenas verificações que podem ser realizadas de maneira automática podem ser inseridas como um filtro de verificações da abordagem proposta.

O Ouriço é baseado em quatro políticas que executam os filtros físico, sintático e semântico. Tais filtros podem identificar respectivamente: conflitos físicos, erros de compilação e falhas durante a execução de testes. No entanto, a verificação estática pode ser incorporada nas verificações de artefatos enviados para o repositório. Esta verificação pode ser realizada por ferramentas como, por exemplo, o *Findbugs* (Hovemeyer and Pugh 2007).

7.3.2 INTEGRAÇÃO DO OURIÇO COM O PEIXE-ESPADA

Durante o mestrado foram desenvolvidas no ambiente Oceano abordagens que futuramente constituirão a GC Contínua. A GC Contínua se difere da GC tradicional por promover a proatividade nas suas funções. Como as abordagens que compõem o Oceano foram dissertações que caracterizam projetos distintos, uma integração entre as abordagens não foi explorada. Uma dessas combinações pode ser o Ouriço com o Peixe-espada.

Conforme discutido anteriormente, o Peixe-espada é uma abordagem capaz de realizar refatorações automáticas em projetos de software. Estas refatorações, no entanto, possuem um processo de verificação automática, que está limitado à compilação e aos testes criados pelos desenvolvedores. Deste modo, refatorações que alteram ações não cobertas por tais testes podem gerar um resultado indesejado, dado que tal abordagem não possui o objetivo de criar testes de forma automática.

Essa verificação pode ser realizada através da associação do Peixe-espada, Ouriço e uma abordagem para criação automática de testes, como o *Saferefactor* (Soares et al. 2009). Neste contexto, o Peixe-espada realizaria refatorações em projetos e, em seguida, *check-in* da configuração resultante em um ramo reservado. Com a integração, o Ouriço no momento de *check-in*, executaria as verificações sintática e semântica na configuração resultante, utilizando os testes automáticos criados pelo *Saferefactor*, por exemplo. Se ambas as verificações fossem realizadas com sucesso, a configuração seria automaticamente integrada ao repositório.

7.3.3 REALIZAR VERIFICAÇÕES PARALELAS EM CONFIGURAÇÕES DISTINTAS

O Ouriço realiza suas verificações de maneira assíncrona ao desenvolvedor. Embora o Ouriço possua uma verificação assíncrona, características paralelas de verificação não são exploradas. No entanto, a exploração dessas características pode resultar na remoção de alguns dos atrasos identificados nos experimentos. Para realizar essa paralelização, alguns pontos devem ser levados em consideração como, por exemplo, a manutenção da ordem dos *check-ins*.

Um dos pontos que pode ser facilmente paralelizado é a verificação de primeiro nível. A verificação de primeiro nível só depende da configuração do desenvolvedor, ou seja, a configuração que o desenvolvedor realizou *check-in*. Portanto, essa verificação pode ser realizada de maneira paralela sem que nenhum prejuízo ao ciclo do comando de *check-in* seja gerado e, conseqüentemente, todas as verificações de primeiro nível podem ser realizadas simultaneamente. Essa paralelização pode resultar em ganhos em alguns pontos da construção como: aquisição de dependência de maneira paralela, construção de várias configurações de desenvolvedores de maneira paralela e quando estas possuísem algum problema, os desenvolvedores seriam alertados, sem que as configurações que chegaram primeiro fossem totalmente verificadas (i.e., sem depender do término das verificações de primeiro e segundo nível para as demais configurações).

7.3.4 ESTUDO DE ESCALABILIDADE DA ABORDAGEM

A construção de sistemas pode ser uma tarefa altamente custosa diante da vasta quantidade de tecnologias e ferramentas utilizadas no desenvolvimento de software. A abordagem Ouriço atua na construção de configurações que incluem compilação e execução de testes. Essas construções podem ser demoradas, dada a diversidade de projetos que podem estar sobre o controle da abordagem Ouriço, e comprometer a escalabilidade da abordagem quando projetos se tornarem grandes e complexos.

Deste modo, a realização de construções mais rápidas pode auxiliar na manutenção de uma abordagem escalável. Uma possível alternativa é o versionamento de objetos derivados. Essa alternativa tende a ser eficiente devido à habilidade dos SGC de reconstruir apenas objetos derivados que tiveram o seu código-fonte alterado e, deste modo, possuem alguma alteração que não está inclusa nos objetos derivados gerados anteriormente. Outra alternativa é identificar os testes que foram afetados pelas áreas de código-fonte alteradas na configuração corrente e reexecutar somente tais testes. Essas habilidades em conjunto

possibilitariam a execução de construções de maneira mais rápida e, possivelmente, resultariam em maior escababilidade da abordagem Ouriço.

REFERÊNCIAS

Al-Ani, B. and Trainer, E. and Ripley, R. and Sarma, A. and van der Hoek, A. and Redmiles, D., (2008), "Continuous coordination within the context of cooperative and human aspects of software engineering", *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, p. 1–4.

Alur, D. and Malks, D. and Crupi, J., (2003), *Core J2EE Patterns: Best Practices and Design Strategies*. 2 ed. Prentice Hall.

Apache Foundation, (2010), Continuum. Disponível em: <"http://continuum.apache.org/">. Acesso em: 16 abr 2010.

Aranda, B. and Wadia, Z., (2008), *Facelets Essentials: Guide to JavaServer(TM) Faces View Definition Framework*. 1 ed. Apress.

Asklund, U. and Bendix, L., (2002), "A Study of Configuration Management in Open Source Software Projects", *IEE Proceedings - Software*, v. 149, n. 1, p. 40-46.

Barnson, M. P. and Weissman, T. and Hernandez, T. and Lawrence, D. and Endico, D. and Steenhagen, J. and Miller, D., (2009), *The Bugzilla Guide - 3.2.2 Release*, Mozilla Foundation.

Bellagio, D., (2005), *Software configuration management strategies and IBM Rational ClearCase : a practical introduction*. 2 ed. Upper Saddle River NJ, IBM Press.

Berczuk, S., (2003), "Pragmatic Software Configuration Management", *IEEE Software*, v. 20 (mar.), p. 15–17.

Berlin, D. and Rooney, G., (2006), *Practical Subversion*. 2 ed. Apress.

Biehl, J. T. and Czerwinski, M. and Smith, G. and Robertson, G. G., (2007), "FASTDash: a visual dashboard for fostering awareness in software teams". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*, p. 1313–1322

Brun, Y. and Holmes, R. and Ernst, M. D. and Notkin, D., (2010), *Speculative identification of merge conflicts and non-conflicts*, Citeseer.

Burns, E. and Schalk, C., (2009), *JavaServer Faces 2.0, The Complete Reference*. 1 ed. McGraw-Hill Osborne Media.

Chacon, S., (2009), *Pro Git*. 1 ed. Berkeley, Calif. :, Apress,.

Collins-Sussman, B. and Fitzpatrick, B. W. and Pilato, C. M., (2004), *Version control with subversion*. O'Reilly Media, Inc.

Conradi, R. and Westfechtel, B., (1998), "Version models for software configuration management", *ACM Computing Surveys (CSUR)*, v. 30 (jun.), p. 232–282.

CruiseControl development team, (2010), CruiseControl. Disponível em: <"http://cruisecontrol.sourceforge.net/">. Acesso em: 9 abr 2011.

Dart, S., (1991), "Concepts in configuration management systems". , p. 1–18, New York, NY, USA.

David, J.-L. and Gousset, M. and Gunvaldson, E., (2006), *Professional Team Foundation Server*. Wrox.

Dewan, P. and Hegde, R., (2007), "Semi-synchronous conflict detection and resolution in asynchronous software development", *ECSCW 2007*, p. 159–178.

Dourish, P. and Bellotti, V., (1992), "Awareness and coordination in shared workspaces", *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, p. 107–114.

Duvall, P. M. and Matyas, S. and Glover, A., (2007), *Continuous Integration: Improving Software Quality and Reducing Risk*. 1 ed. Addison-Wesley.

Estublier, J., (2000), "Software configuration management: a roadmap". In: *Proceedings of the Conference on The Future of Software Engineering*, p. 279–289, New York, NY, USA.

Estublier, J. and Leblang, D. and van der Hoek, A. and Conradi, R. and Clemm, G. and Tichy, W. and Wiborg-Weber, D., (2005), "Impact of Software Engineering Research on the Practice of Software Configuration Management", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 14, n. 4, p. 1-48.

Feldman, S. I., (1979), "Make - a program for maintaining computer programs", v. 9, p. 255--265.

Fogel, K., (2003), *Open source development with CVS*. 3 ed. Scottsdale (Ariz.), Paraglyh Press.

Fowler, M., (2005), "Using the Rake build language"

Fowler, M., (2010), Continuous Integration. Disponível em: <"<http://martinfowler.com/articles/continuousIntegration.html>">. Acesso em: 8 fev 2011.

Ganoe, C. H. and Convertino, G. and Carroll, J. M., (2004), "The BRIDGE awareness workspace: tools supporting activity awareness for collaborative project work". In: *Proceedings of the third Nordic conference on Human-computer interaction*, p. 453–454, Tampere, Finland.

Hamilton, B., (2004), *NUnit Pocket Reference (Pocket Reference*. 1 ed. O'Reilly Media.

Hass, A. M. J., (2003), *Configuration Management Principles and Practices*. Boston, MA, Pearson Education, Inc.

Heffelfinger, D., (2007), *Java EE 5 Development using GlassFish Application Server: The complete guide to installing and configuring the GlassFish Application Server and ... 5 applications to be deployed to this server*. 1 ed. Packt Publishing.

Holzner, S., (2005), *Ant: the definitive guide*. 2 ed. Beijing;;Sebastopol CA;; O'Reilly.

Hovemeyer, D. and Pugh, W., (2007), "Finding more null pointer bugs, but not too many". In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, p. 9–14

Hovemeyer, D. and Pugh, W., (2004), "Finding bugs is easy", *ACM SIGPLAN Notices*, v. 39 (dez.), p. 92–106.

IEEE, (1990), *IEEE standard glossary of software engineering terminology*. New York N.Y. USA, Institute of Electrical and Electronics Engineers.

IEEE, (2005), *IEEE Std 828 - standard for software configuration management plans*. New York, N.Y. :, Institute of Electrical and Electronics Engineers,.

- Katz, M., (2008), *Practical RichFaces*. 1 ed. Apress.
- Keith, M. and Schincariol, M., (2009), *Pro JPA 2: Mastering the Java(TM) Persistence API*. 1 ed. Apress.
- Laddad, R., (2009), *Aspectj in Action: Enterprise AOP with Spring Applications*. Second Edition ed. Manning Publications.
- Legenhausen, M. and Pielicke, S. and Rühmkorf, J. and Wendel, H. and Schreiber, A., (2009), "RepoGuard: A Framework for Integration of Development Tools with Source Code Repositories". In: *Global Software Engineering, International Conference on*, p. 328-331, Los Alamitos, CA, USA.
- Lehman, M. M. and Ramil, J. F., (2001), "Rules and Tools for Software Evolution Planning and Management", *Annals of Software Engineering*, v. 11 (nov.), p. 15–44.
- Leon, A., (2000), *A Guide to Software Configuration Management*. Norwood, MA, Artech House Publishers.
- Linux Reviews, (2011), Compile time stats. Disponível em: <"<http://linuxreviews.org/gentoo/compiletimes/>">. Acesso em: 31 jan 2011.
- Massol, V. and Husted, T., (2003), *JUnit in Action*. Manning Publications.
- Mecklenburg, R., (2005), *Managing projects with GNU make*. 3 ed. Beijing ;;Cambridge [Mass.], O'Reilly.
- Mens, T., (2002), "A State-of-the-Art Survey on Software Merging", *IEEE Trans. Softw. Eng.*, v. 28, n. 5, p. 449-462.
- Murphy, D., (2007), *Managing software development with Trac and Subversion: simple project management for software development*. Birmingham UK, Packt Publishing.
- Murta, L. G. P., (2009a), Gerência de Configuração: Introdução. Disponível em: <"<http://www.ic.uff.br/~leomurta/courses/2009.1/gc/aula2.pdf>">.
- Murta, L. G. P. and Oliveira, H. L. R. and Dantas, C. R. and Lopes, L. G. B. and Werner, C. M. L., (2007), "Odyssey-SCM: An integrated software configuration management infrastructure for UML models", *Science of Computer Programming*, v. 65, n. 3, p. 249-274.
- Murta, L. G. P., (2009b), *Gerência de Configuração no Desenvolvimento baseado em Componentes*. Tese de Doutorado, UFRJ
- Nock, C., (2003), *Data Access Patterns: Database Interactions in Object-Oriented Applications*. 1 ed. Addison-Wesley Professional.
- O'Sullivan, B., (2009), *Mercurial: the definitive guide*. 1 ed. Sebastopol CA, O'Reilly Media Inc.
- Panjer, L. D. and Damian, D. and Storey, M.-A., (2008), "Cooperation and coordination concerns in a distributed software development project". In: *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*, p. 77–80, Leipzig, Germany.
- PMD, (2011), PMD. Disponível em: <"<http://pmd.sourceforge.net/>">. Acesso em: 1 maio 2011.
- Pressman, R., (2000), *Software engineering: a practitioner's approach*. 5 ed. Boston Mass., McGraw Hill.
- Prudêncio, J. G. and Murta, L. and Werner, C., (2009), "On the Selection of

Concurrency Control Policies for Configuration Management". In: *2009 XXIII Brazilian Symposium on Software Engineering 2009 XXIII Brazilian Symposium on Software Engineering (SBES)*, p. 155-164, Fortaleza, Ceara, Brazil.

Redmine, (2011), Redmine. Disponível em: <"http://www.redmine.org/">. Acesso em: 8 ago 2011.

Ripley, R. M. and Yasui, R. Y. and Sarma, A. and van der Hoek, A., (2004), "Workspace awareness in application development". In: *Proceedings of the 2004 OOPSLA Workshop on Eclipse technology eXchange*, p. 17–21, Vancouver, British Columbia, Canada.

Sarma, A. and Bortis, G. and Van Der Hoek, A., (2007), "Towards supporting awareness of indirect conflicts across software configuration management workspaces". In: *Proceedings of the Twenty-Second IEEE/ACM international conference on Automated software engineering*, p. 94–103

Sarma, A. and Noroozi, Z. and Van Der Hoek, A., (2003), "Palantír: raising awareness among configuration management workspaces", /, p. 444.

Sarma, A. and Redmiles, D. and van der Hoek, A., (2008), "Empirical evidence of the benefits of workspace awareness in software configuration management", *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, p. 113–123.

Schmeelk, S., (2010), "Towards a unified fault-detection benchmark". In: *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, p. 61–64, Toronto, Ontario, Canada.

Servant, F. and Jones, J. A. and van der Hoek, A., (2010), "CASI: preventing indirect conflicts through a live visualization". In: *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, p. 39–46, Cape Town, South Africa.

Smith, G., (2010), *PostgreSQL 9.0 High Performance*. Packt Publishing.

Soares, G. and Cavalcanti, D. and Gheyi, R. and Massoni, T. and Serey, D. and Cornélio, M., (2009), "Saferefactor-tool for checking refactoring safety", *Tools Session at SBES*, p. 49–54.

Sonatype, (2008), *Maven : the definitive guide*. 1 ed. Sebastopol Calif, Oreilly.

Sun Microsystems, (2011), Hudson Continuous Integration. Disponível em: <"http://hudson-ci.org/">. Acesso em: 9 abr 2011.

Teles, V. M., (2006), *Integração Contínua*. Disponível em: <"http://improveit.com.br/xp/praticas/integracao">. Acesso em: 26 ago 2009.

Tichy, W. F., (1985), "RCS, a system for version control", *Software—Practice & Experience*, v. 15 (jul.), p. 637–654.

TMate Software, (2011), SVNKit. Disponível em: <"http://svnkit.com/">. Acesso em: 15 jun 2011.

Travassos, G. H. and Barros, M. O., (2003), "Contributions of In Virtuo and In Silico Experiments for the Future of Empirical Studies in Software Engineering". In: *Workshop on Empirical Studies in Software Engineering (WSESE)*, p. 109-121, Rome, Italy.

Wingerd, L., (2005), *Practical Perforce*. Sebastopol Calif. ;;Cambridge, O'Reilly.

Wloka, J. and Ryder, B. and Tip, F. and Ren, X., (2009), "Safe-commit analysis to

facilitate team software development". In: *Proceedings of the 31st International Conference on Software Engineering*, p. 507–517, Washington, DC, USA.