UNIVERSIDADE FEDERAL DE PERNAMBUCO


GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Período: 2004.2

# GERÊNCIA DE CONFIGURAÇÃO DE SOFTWARE

# PADRÕES E FERRAMENTAS

Aluno: Alexsandro José de Melo farias

Orientador: André Luís de Medeiros Santos

Sumário

1. Introdução ..... 3
1.1 Objetivo ..... 3
1.2 Padrões ..... 3
2. Padrões em GCS ..... 5
2.1 Mainline ..... 8
2.2 Active Development Line ..... 11
2.3 Private Workspace ..... 15
2.4 Repository ..... 16
2.5 Private System Build ..... 17
2.6 Integration Build ..... 21
2.7 Third Party Codeline ..... 24
2.8 Task Commit Level ..... 27
2.9 Codeline Policy ..... 28
2.10 Smoke Test ..... 30
2.11 Unit Testing ..... 32
2.12 Regression Test ..... 33
2.13 Private Versions ..... 34
2.14 Release line ..... 34
2.15 Release-Prep Code Line ..... 37
2.16 Task Branch ..... 39
3. Suporte das ferramentas aos padrões de GCS ..... 41
3.1 Innovative Repository Manager ..... 43
3.2 VSS - Visual Source Safe ..... 45
3.3 CVS— Concurrent Versions System ..... 46
3.4 Perforce ..... 48
3.5 BitKeeper ..... 49
3.6 AccuRev ..... 51
3.7 ClearCase LT ..... 52
3.8 ClearCase – UCM ..... 54
3.9 CM Synergy ..... 55
3.10 StarTeam ..... 56
3.11 PVCS Dimensions ..... 57
3.12 PVCS Version Manager ..... 59
3.13 MKSource Integrity ..... 60
4. Considerações finais ..... 62
5. Bibliografia ..... 63

## 1. Introdução

Neste trabalho serão descritos temas relacionados à linguagens de padrões, padrões de uma forma geral, padrões de gerência de configuração de software (GCS) e ferramentas de GCS que podem ajudar a entender e melhorar o processo de desenvolvimento de software de equipes / organizações.

Visando um melhor entendimento deste trabalho, a seção 1.1 relata o objetivo da pesquisa, a seção 1.2 introduz o conceito de linguagem de padrões e padrões de uma forma geral. O capítulo 2 descreve os padrões de GCS e fornece informações necessárias para saber em quais situações devemos usá-los. Já o capítulo 3 apresenta o resultado da pesquisa sobre as ferramentas de GCS versus os padrões. Finalmente, o capítulo 4 faz as considerações finais sobre o trabalho.

### 1.1 Objetivo

O objetivo principal consiste em levantar uma pesquisa com as ferramentas de GCS mais utilizadas pelas organizações para avaliar a aderência / implementação das mesmas em relação aos mais usados padrões de gerência de configuração de software. Através do resultado dessa pesquisa, empresas poderão adquirir ferramentas de GCS que atendam melhor as suas necessidades, além de explorar o potencial das mesmas de forma mais efetiva.

### 1.2 Padrões

Um conjunto de padrões que pertencem a um mesmo contexto, formam uma linguagem de padrões. Dessa forma existem linguagens relacionadas aos seguintes contextos: software, arquitetura, equipes, tecnologia de desenvolvimento, entre outros.

Uma simples definição de padrão é a seguinte: “padrão é uma solução para um problema num contexto.” Cada padrão de uma linguagem de padrão

complementa o outro na mesma linguagem. Isto significa que padrões se agrupam com outros padrões para formar uma linguagem para um determinado contexto.

A idéia de padrões e linguagem de padrões é originalmente do trabalho que o arquiteto Christopher Alexander fez em construir uma arquitetura para descrever qualidades para bons projetos arquiteturais. Nos anos 70 ele iniciou usando linguagem de padrões para descrever eventos e formas que aparecem em cidades, países, e edifícios no mundo.

Alexander fala sobre um padrão como alguma coisa que “descreve um problema que ocorre inúmeras vezes em nosso ambiente, e que descreve o conjunto de soluções para aquele problema, de tal maneira que se possa usar esta solução um milhão de vezes e sempre obter o mesmo resultado. Alexander define um padrão como” uma regra que descreve o que temos que fazer para gerar a entidade que ela define” (Alexander 1979).

Padrões, segundo Alexander, são mais do que apenas soluções. Eles são boas soluções. Segundo, Alexander, “a linguagem de padrões é um sistema que permite a seus usuários criar uma infinita variedade de combinações de padrões que nós chamados de edifícios, jardins e cidades” (Alexander 1979). Alexander documentou padrões que existiam em lugares das cidades e edifícios. Por exemplo, um dos padrões de Alexander é *HALF PRIVATE OFFICE*, que descreve como alcançar o equilíbrio certo entre privacidade e conexão para trabalho em escritório.

Partindo da idéia inicial de Alexander sobre catalogar padrões na escala humana, podemos aplicar a mesma idéia para catalogar padrões no contexto de desenvolvimento de software. Desenvolvimento de software envolve pessoas trabalhando juntas para desenvolver artefatos. As técnicas usadas para construir esses artefatos são freqüentemente similares a outras que foram utilizadas no passado. Essas técnicas podem ser catalogadas e agrupadas por contexto formando várias linguagens de padrões para o desenvolvimento de software.

O primeiro maior trabalho em padrões de software foi o livro *Design Patterns* (Gamma et al.1995). Este livro cataloga algumas técnicas chaves em projeto orientado a objeto com muito boa descrição sobre quando e como implementá-las. *Design Patterns* não capturam todo o poder de padrões porque

cada padrão é muito isolado, e é necessário ainda de um bom entendimento sobre sistemas de software para juntá-los e construir o sistema.

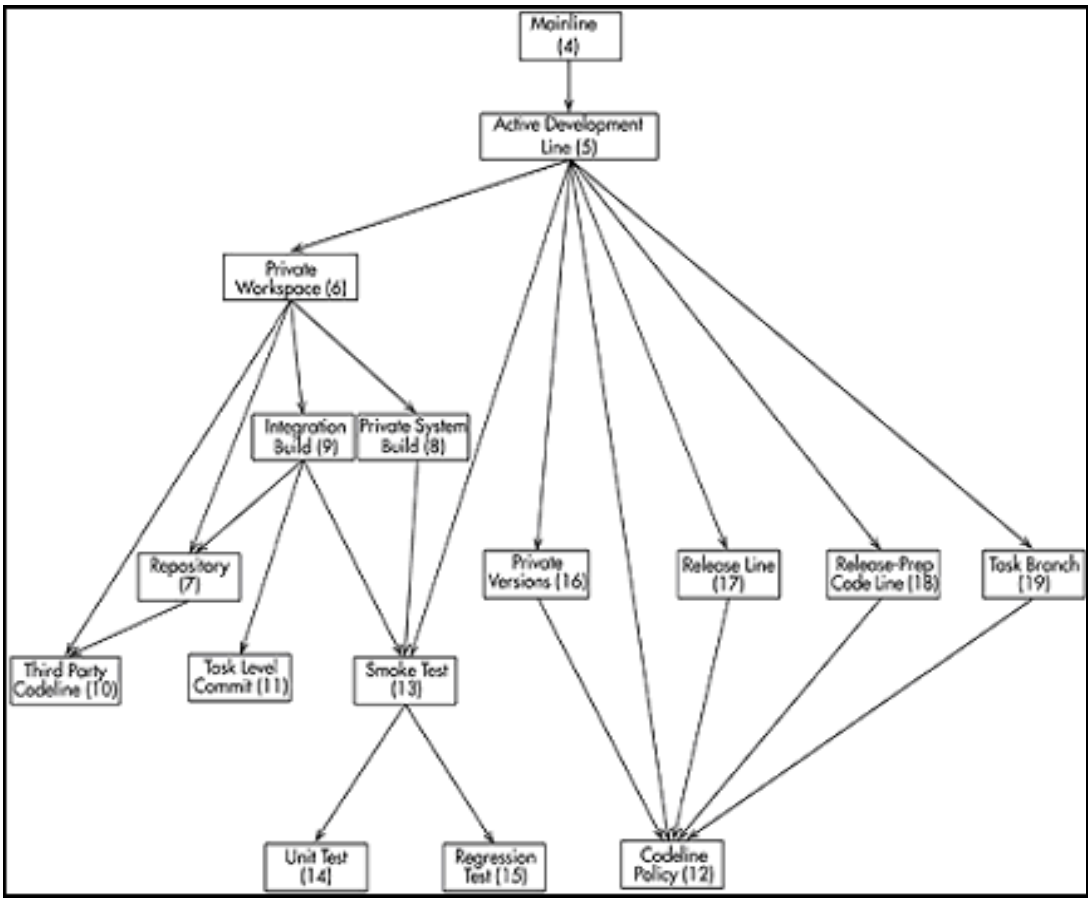
Neste trabalho será descrita uma linguagem de padrões no contexto de gerência de configuração de software. Nos próximos capítulos serão abordados cada padrão que pertence a essa linguagem.

## 2. Padrões em GCS

As organizações estão cada vez mais se preocupando com seus processos de gerência de configuração de software (GCS) visando maior competitividade, qualidade, produtividade e manutenibilidade. Para obter isso, é importante ter um bom entendimento sobre padrões de gerência de configuração de software, saber quando é necessário usar ou não um determinado padrão e saber quais ferramentas suportam os padrões usados na organização.

Padrões são independentes de ferramentas. Algumas ferramentas suportam padrões explicitamente e outras menos diretamente. Por exemplo: algumas ferramentas não suportam *branching* e *merging* tão bem quanto outras. Claramente essas não são práticas executadas rotineiramente sem apoio de ferramentas, mas em algumas situações, quando o suporte a *braching* é rudimentar, é melhor evitá-los.A figura 1 mostra os padrões da linguagem discutida nesse trabalho:

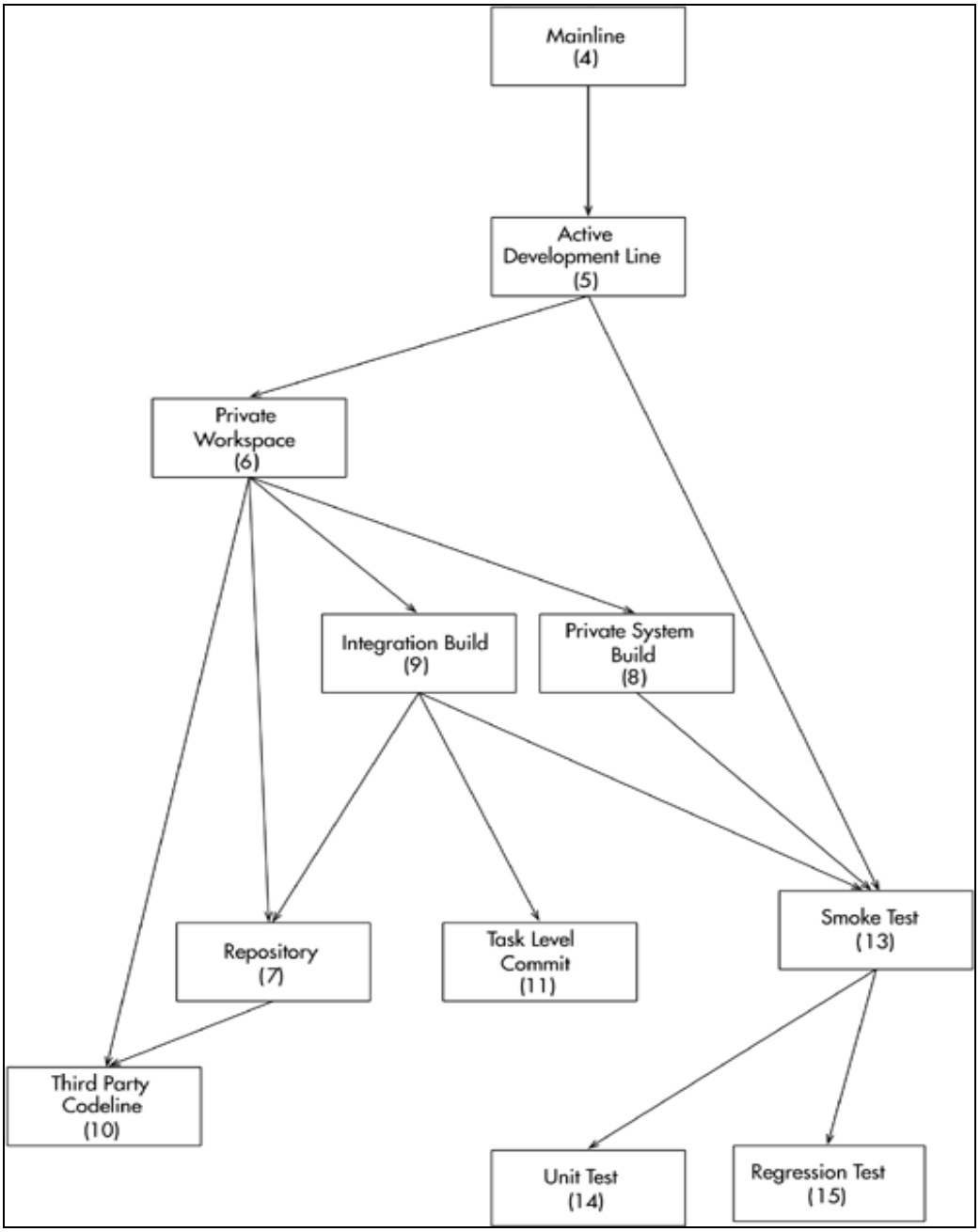
**Figura 1. Padrões**




Na figura acima, uma seta de um padrão (A) para um outro padrão (B) significa que o padrão A está no contexto do padrão B. Em outras palavras, significa que A precisa de B para ser completo.

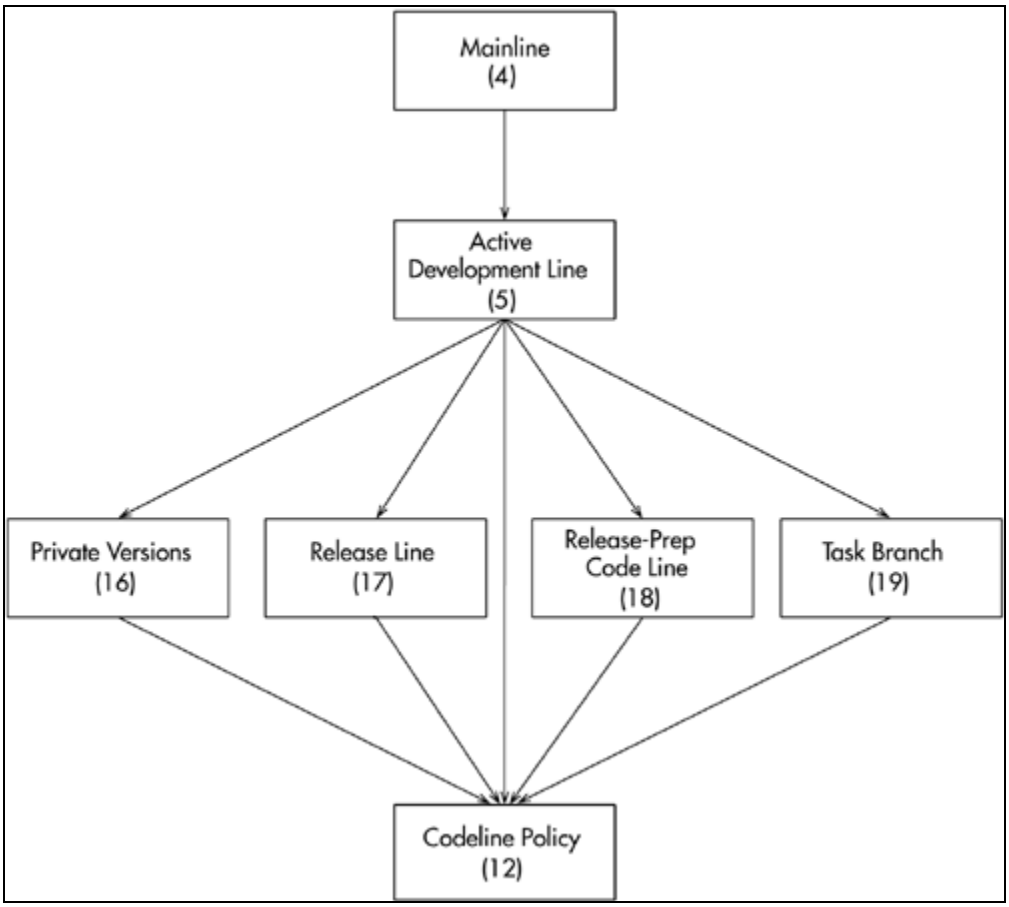
Os padrões são  usados para estabelecer um ambiente de desenvolvimento ativo que busca um ponto de equilíbrio entre velocidade e produtividade, tal que seja possível produzir um bom produto rapidamente. Depois dos dois primeiros padrões, na figura 1,os padrões são agrupados em dois tipos: *padrões relacionados a codeline* (figura 2) e *padrões relacionados a área de trabalho* (figura 3).

**Figura 2. Padrões relacionados à área de trabalho**




Os *padrões relacionados a codeline* ajudam a organizar o código fonte e outros artefatos em uma apropriada seção em termos de estrutura e tempo.

**Figura 3. Padrões relacionados a codeline**




## 2.1 Mainline

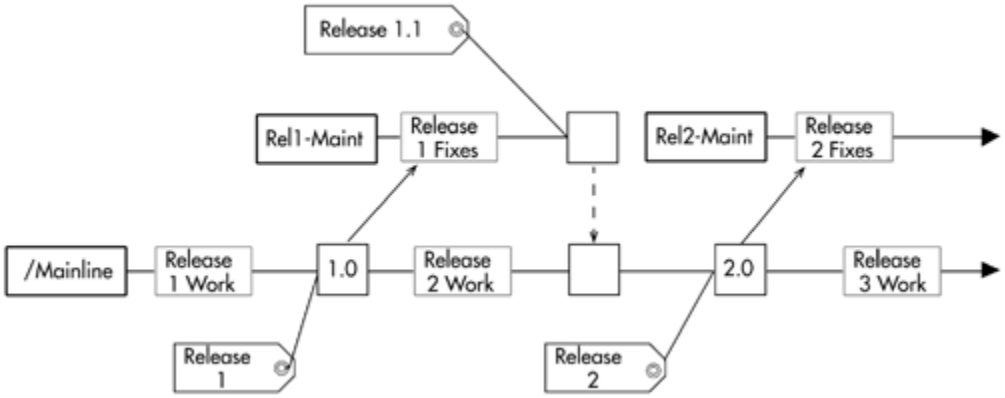
Quando estamos desenvolvendo um software com a colaboração de vários grupos de pessoas, temos que integrar freqüentemente esforços de desenvolvimento em paralelo dessas pessoas. Para isso as ferramentas de controle de versão geralmente oferecem as facilidades de branching e merging. Podemos usar branchs para isolar esforços paralelos, mas isso pode ter um custo relacionado aos esforços dos merges. O padrão *mainline* mostra como gerenciar a codeline para minimizar esforços com integração que branching e merging requerem.

Se não usamos branchs, perdemos a isolação que ele oferece e cada membro da equipe precisa disputar a aquisição do artefato para realizarem suas

tarefas. E aquele que consegue o artefato, seqüestra-o para realizar suas mudanças enquanto os demais candidatos a adquirí-lo esperam na fila. Por outro lado, se usamos branchs aumentamos o desenvolvimento concorrente entre os membros da equipe, mas em muitos casos, os códigos precisam trabalhar juntos de qualquer forma e precisamos alocar recursos para integrar a mudanças paralelas. Dessa forma, precisamos balancear a liberdade que branching fornece com o custo que encontraremos quando precisamos re-sincronizar as várias atividades da equipe para encontrar um ponto de equilíbrio.Queremos maximizar a concorrência em nossas codelines enquanto minimizamos problemas causados pela integração.

A razão para trabalhar com uma *mainline* é para ter uma codeline central que seja base para subbranches e para acomodar seus merges. A *mainline* para um projeto geralmente inicia com o código base do release anterior, ou se estamos iniciando um novo projeto, temos somente uma codeline que é a própria *mainline*.Trabalhar com uma mainline não significa "não fazer branch", significa que todas as atividades desenvolvidas sobre os artefatos são refletidas sempre sobre uma simples codeline em um determinado momento. Devemos realizar a maioria dos nossos trabalhos na *mainline* e usar ferramentas de desenvolvimento e de testes para garantir a integridade/qualidade dos artefatos.Não devemos criar um branch, a menos que tenhamos uma forte razão para isso. Favoreça branchs que não precisem realizar merges freqüentemente – por exemplo, branchs que representam um release ou versão do produto. Branching pode ser uma poderosa ferramenta, mas como uma ferramenta, ele precisa ser tratado com seriedade e bom entendimento. Quando precisamos criar uma codeline para uma nova versão do sistema, ao invés de criar um branch da codeline do release anterior, devemos fazer merge dessa codeline com a mainline e desta criar o branch para a nova versão. Esse modelo de branching está ilustrado na figura abaixo:

**Figure 4 – Modelo de branching**




Trabalhar com uma *mainline* oferece as seguintes vantagens:

- Reduz esforços de merging e sincronização, pois requerem poucas propagações transitivas de mudanças.
- Maior poder de coesão por centralizar as mudanças em uma única codeline, ao invés de deixá-las espalhadas / fragmentadas pelos branchs.

Existem ainda várias razões para trabalhar com branchs. Poderíamos, por exemplo, criar um branch no final do ciclo de uma release para isolar as mudanças que visam estabilizar o release das novas funcionalidades alocadas para o próximo release. Isto possibilita consertar erros no release corrente sem introduzir novas características e outras mudanças pertencentes ao próximo release. Branchs devem ser criados em determinadas situações como:

- Releases dos clientes. Isto possibilita corrigir erros no release sem expor para o cliente novas funcionalidades que estão sendo desenvolvidas na *mainline*.
- Longos esforços paralelos onde múltiplas pessoas estão trabalhando. Se este trabalho pode tornar a *mainline* muito instável, devemos criar um *Task Branch* para isolar temporariamente essas mudanças da *mainline*
- Integração. Quando criamos branchs para as versões dos clientes, geralmente congelamos a mainline para estabilizar o release atual e só depois criamos o branch. Para evitar isso, podemos criar um branch de

integração onde os desenvolvedores trabalharão para estabilizar a versão corrente, enquanto o produto continua a evoluir na mainline.

Como podemos observar, existem várias situações onde podemos usar branchs, mas antes de usar temos que ter certeza de que o trabalho realmente requer um branch, pois temos consciência do esforço requerido para fazer merges depois.

Para implementar o padrão *mainline* podemos seguir os passos abaixo:

1. Criar uma codeline (ex.: */mainline*) com todos os artefatos correntes pertencendo a mesma.
2. Check - in todas as mudanças para essa codeline.
3. Executar necessários pré-check-in procedimentos de teste para manter a *mainline* usável e íntegra.

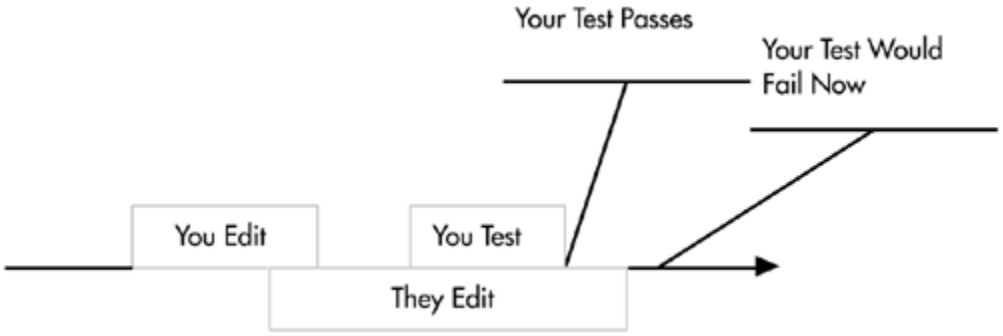
Na próxima seção veremos como gerenciar a *mainline* para mantê-la usável quando muitas pessoas estão trabalhando na mesma. Para isso estudaremos o padrão *Active Development Line*.

## 2.2 Active Development Line

Na seção anterior foi estudado o padrão *Mainline*, que consiste de uma codeline central onde todo o trabalho das releases do produto ocorre diretamente sobre ela ou indiretamente através dos merges. Quando estamos trabalhando com um ambiente de desenvolvimento bastante dinâmico, muitas pessoas estão mudando os artefatos, principalmente o código fonte e algumas dessas mudanças podem desestabilizar o sistema, ou conflitar com outras mudanças. Este padrão ajuda a balancear estabilidade e progresso no desenvolvimento do produto sobre a *mainline*.Podemos dizer, também, que o padrão *Mainline* precisa do padrão *Active Development Line* para ser executado de forma completa ou eficiente.

Para aumentar a performance de um desenvolvimento em equipe desejamos aumentar a quantidade de membros da equipe trabalhando na suas atividades de forma concorrente e também aumentar a comunicação entre esses membros para diminuir conflitos entre as suas atividades.Essa comunicação é implementada através dos pontos de sincronização onde os trabalhos de todos os membros serão integrados e suas dependências serão gerenciadas para evitar deadlock e blocking. Deadlocks são causados por dependências múltiplas e blocks ocorrem quando alguém reserva um artefato para editá-lo e passa muito tempo para liberá-lo para as outras pessoas. Quando alguém check-in uma mudança para a mainline, esta pessoa pode causar atrasos a toda a equipe a sua mudança impacta o trabalho dos demais. Então seria necessário antes de executar o check-in testar as mudanças com a última versão dos artefatos da mainline para observar qualquer incompatibilidade. Mas se os testes consumirem muito esforço, isso causará atrasos no progresso do projeto.Queremos que as pessoas executem simples procedimento antes do check-in do código fonte para mainline. Esses procedimentos podem ser, por exemplo, um build ou algum nível de teste. Mesmo realizando esses procedimentos, o problema de concorrência não é resolvido, pois duas mudanças podem ser testadas individualmente, mas quando integradas desestabilizar a mainline. E por mais exaustivo que sejam os seus testes, sobre uma mudança, sempre existe a probabilidade de uma segunda mudança não compatível ser submetida e os testes da primeira falharem, como ilustra a figura 5, abaixo:

**Figura 5. Falha dos testes pré-check-in**



Podemos prevenir este tipo de situação se somente uma pessoa por vez testar e check-in suas mudanças, mas isso pode diminuir a performance do progresso. A figura 6 mostra uma codeline muito estável, mas de baixa performance:

**Figura 6 – Uma codeline estável, mas lenta**


Podemos também fazer mudanças na estrutura da codeline para manter partes da sua árvore estável, criando branchs em vários pontos, mas isso adiciona complexidade e requer esforços para merges.

No outro extremo, podemos deixar a codeline livre para executar todas as tarefas sem nenhum procedimento pré-check-in, mas não usável ou com alto grau de instabilidade, como mostra a figura 7:

**Figura 7 - Uma codeline muito ativa, mas não usável**


É possível obter uma codeline estável, mas com atrasos de processo e sincronização e nem sempre isso é vantajoso. Queremos balancear para achar um ponto de equilíbrio onde tenhamos uma codeline que com um nível de estabilidade / performance toleráveis.

Uma *Active development line* (linha de desenvolvimento ativa) sofrerá freqüentes mudanças, algumas serão bem testadas e outras fracamente testadas como ilustra a figura 8, abaixo:

**Figura 8 – Uma codeline balanceada**


A parte difícil dessa solução é saber o quanto “bom” precisam ser os testes sobre a codeline. Para nos ajudar, podemos fazer a seguinte análise:

- Quem usa a codeline?
- Qual é o ciclo de release?
- Quais mecanismos de testes temos a disposição?
- Qual é o custo real para um ciclo onde a codeline fica instável?

Por exemplo, se estamos no desenvolvimento de um novo sistema ou estamos adicionando muitas novas funcionalidades ao produto, podemos focar mais em velocidade (realizar menos testes). Já se a codeline serve de base para o trabalho de outras pessoas, mais validação seria mais apropriada (realizar bons testes). Se dispusermos de boas ferramentas para fazer teste unitário e de regressão, podemos executá-los como um procedimento pós-check-in, desta forma os erros não “viverão” por muito tempo e podemos enfatizar em dar mais *velocidade* ao check-in. Se não dispomos de uma boa infra-estrutura de testes, precisamos ser mais cuidadosos até melhorarmos a mesma.

Temos também que ter cuidado para não tornar o processo de check-in difícil de ser executado. Por exemplo, se tivermos um processo de pré-check-in que consuma muito tempo, corremos o risco dos desenvolvedores realizarem check-in menos freqüentemente e de tarefas de alta granularidade. Isto causa, respectivamente, o aumento da probabilidade de conflitos e a diminuição da possibilidade de *roll-back* uma mudança problemática. Para evitar esses

problemas, temos que estabelecer uma política para saber quanto teste é necessário antes de fazer um check-in. Por exemplo, poderíamos antes do check-in executar localmente os scripts de testes de partes mais críticas do sistema.Ou ainda, baseado na análise de impacto da mudança, executar os scripts das possíveis partes do sistema afetadas direta ou indiretamente pela mudança.Os testes mais exaustivos poderiam ser executados em uma área de integração de forma periódica.

### 2.3 Private Workspace

Em *Active Development Line*, as pessoas fazem freqüentes mudanças no código e para isso precisam das últimas versões dos arquivos inclusive das suas dependências. Se todas as pessoas mudarem de uma forma descontrolada, surgirão vários problemas de deadlocks e locks chegando ao caos. Então este padrão descreve como conciliar a tensão de obter sempre as últimas versões dos arquivos e ao mesmo tempo permitir que as dependências sejam editadas sem causar nenhum problema aos dependentes. Desenvolvedores, por exemplo, precisam de um lugar onde eles possam trabalhar em seu código de forma isolada até que eles finalizem as suas tarefas.

Cada membro da equipe pode está apto a usar um *private workspace* (área de trabalho privada) populada com uma consistente versão do software. Nessa área de trabalho, o proprietário pode ter total controle para atualizá-la, recriar uma versão passada, editar os arquivos, executar testes dos procedimentos pré-check-in, após ter finalizado as suas tarefas, e finalmente, publicar suas mudanças (check-in).

Uma área de trabalho privada para um desenvolvedor, por exemplo, poder ser constituída da seguinte forma:

- Código fonte
- Algum componente de build

- Ferramentas de desenvolvimento
- Arquivos de dados e configuração de testes
- Scripts de builds
- Informação identificando os arquivos e as versões de que compõem o sistema

Abaixo seguem alguns passos que envolvem a criação / manipulação das áreas de trabalho privadas:

1. Criar uma área de trabalho local e popule a mesma com a última versão dos artefatos da codeline que onde o trabalho de ser executado. Se a área já estiver sido criada, devemos atualizá-las com as últimas versões. Se a codeline for um novo branch ou label, crie uma nova área de trabalho para aquele branch.
2. Realizar as mudanças localmente. Editar os componentes alvos da mudança
3. Fazer um build privado para atualizar qualquer dependência
4. Realizar teste unitário para a mudança
5. Atualizar a área de trabalho com as últimas versões de todos os componentes que não tenham sido mudados
6. Fazer um build novamente e executar um *Smoke test*.

## 2.4 Repository

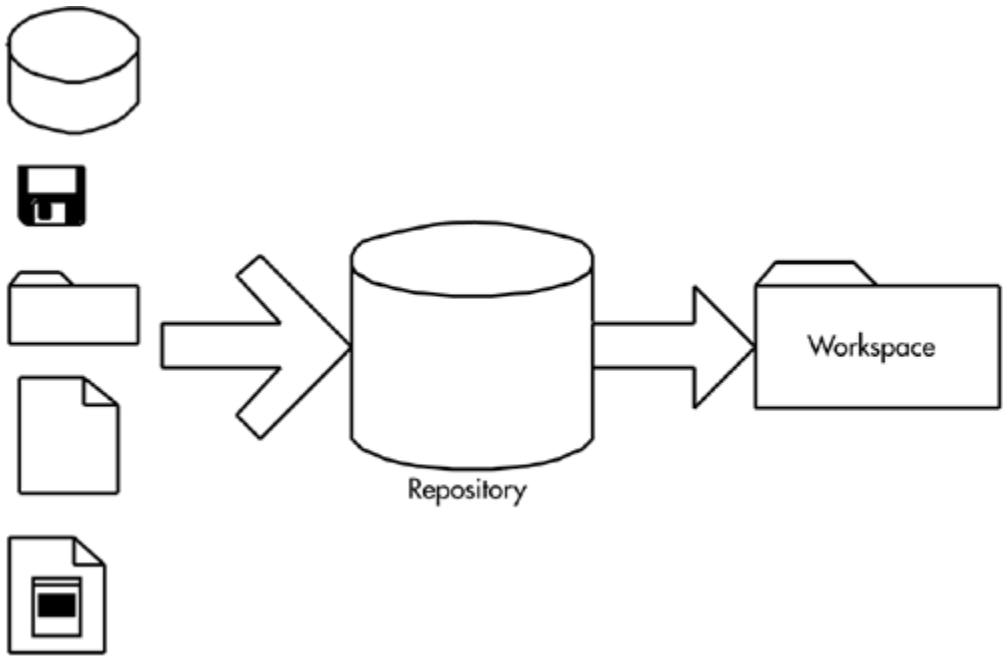
Para criar uma área de trabalho local precisamos dos corretos arquivos e das corretas versões destes que compõem uma versão do sistema. Este padrão mostra como construir uma área de trabalho local facilmente e com a configuração necessária.

Quando vamos realizar uma mudança sobre o sistema precisamos de uma área onde possamos acomodar as corretas versões dos arquivos do sistema para

realizar a mudança, e testa-la de forma isolada. Queremos obter os elementos da área de trabalho facilmente tal que possamos confiantemente criar um ambiente que permita-nos fazer nosso trabalho com a versão correta do software, por exemplo, quando vamos corrigir um erro de uma versão anterior do software.

A criação manual de uma área de trabalho gasta muito tempo. E podem levar a erros de configuração por não pegarem a versão correta de um componente. Dessa forma, precisamos de um mecanismo simples e repetível para criar uma área de trabalho. Poderíamos criar áreas de trabalho que contivessem artefatos de alguma versão identificável do produto, como artefatos de uma determinada release, por exemplo. O mecanismo poderia também facilitar a necessidade de atualização da área de trabalho quando uma nova release fosse criada, a figura 9 mostra esse mecanismo (Repository):

**Figura 9 - Populando a área de trabalho do repositório**

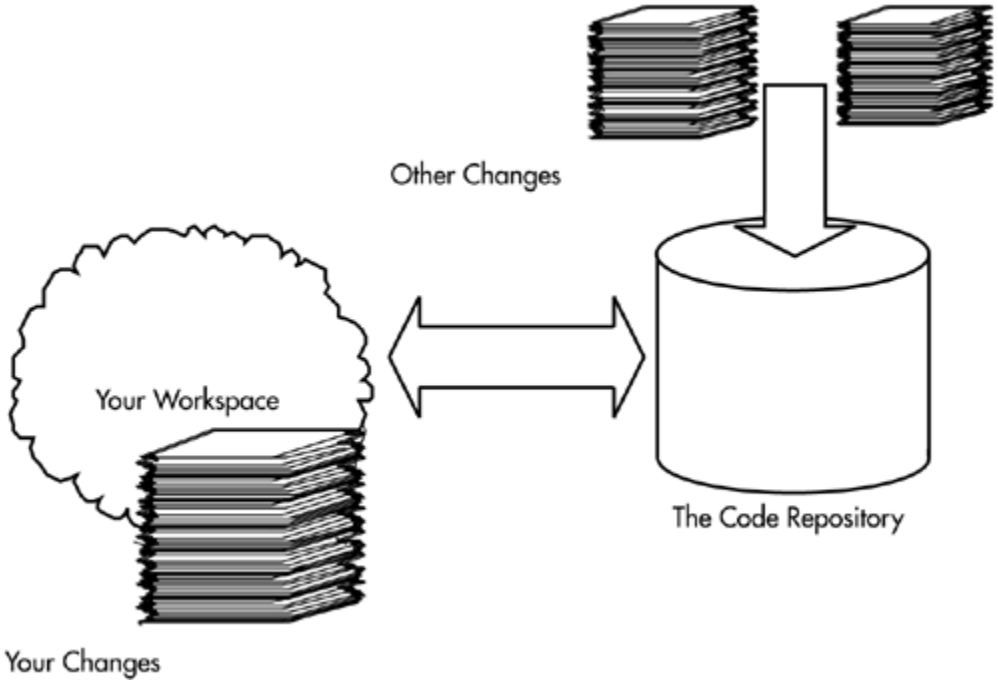



## 2.5 Private System Build

Um *Private Workspace* permite ao proprietário trabalhar de forma isolada das mudanças realizadas pelas demais pessoas. Mas esses trabalhos precisam ser sincronizados / integrados em um determinado momento para compor o sistema de forma íntegra. Para isso é necessário realizar builds locais no sistema consistentemente e incluindo as novas mudanças.Este padrão explica como podemos verificar se as mudanças sendo desenvolvidas nas áreas de trabalhos estão consistentes com a atual configuração da codeline, antes de publicá-las (check-in).

Em uma equipe de desenvolvimento com política liberal de codelines, mudanças acontecem muito rapidamente. O código existente é mudado, novos módulos são adicionados a codeline, e as dependências podem mudar.O único verdadeiro teste para avaliar a compatibilidade das mudanças é através da realização de builds centralizados da área de trabalho de integração que acomoda todas as mudanças de todos os membros da equipe. Os builds privados eliminam alguns possíveis erros sobre as mudanças realizadas na área de trabalho, mas não garantem que apareçam erros no builds noturnos quando todas as mudanças são integradas, como ilustra a figura 10, abaixo:

**Figura 10 – O build integra mudanças de todos os membros**




Às vezes os builds de precheck-in podem funcionar corretamente, mas os builds noturnos podem falhar. Ou a cópia do sistema obtida da ferramenta de controle de versão funciona bem, mas a instalação do produto através do instalador construído no build noturno não funciona. Algumas vezes o problema neste caso é que o produto instalado não incorporou um novo arquivo ou recurso que foi adicionado ao controle de versão. Algumas empresas mantêm uma lista dos arquivos que compõem um build, mas muitas vezes os desenvolvedores adicionam novos arquivos e não atualizam a lista, causando esses tipos de erros.

Para realizar um razoável teste dos efeitos das mudanças, é necessário build todas as partes do código afetadas pelas mesmas.

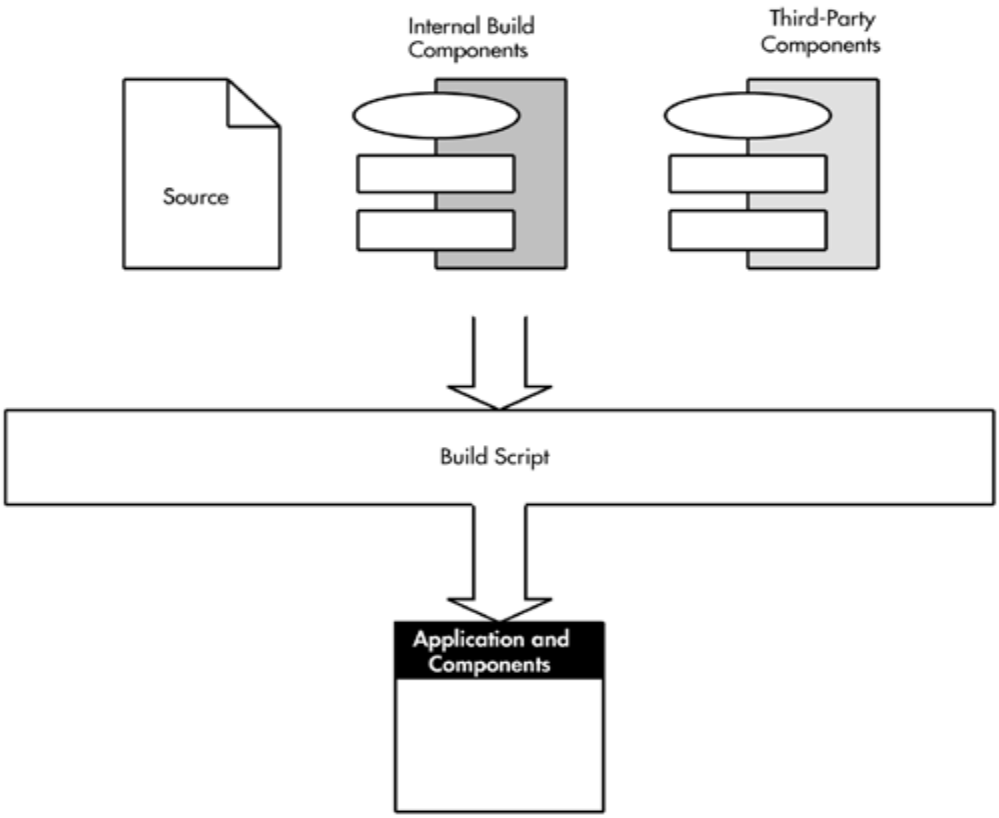
O private system build (build privado de sistema) poderia ter os seguintes atributos:

- Ser o mais parecido possível do build de integração que testa todas as mudanças

- Incluir todas as dependências
- Incluir todos os componentes de terceiros

A arquitetura do sistema ajudará a determinar qual conjunto de componentes é suficiente para realizar o build. Uma arquitetura com bons encapsulamentos facilita a construção do build. A figura 11 ilustra isso:

**Figura 11 – Componentes do private system build**



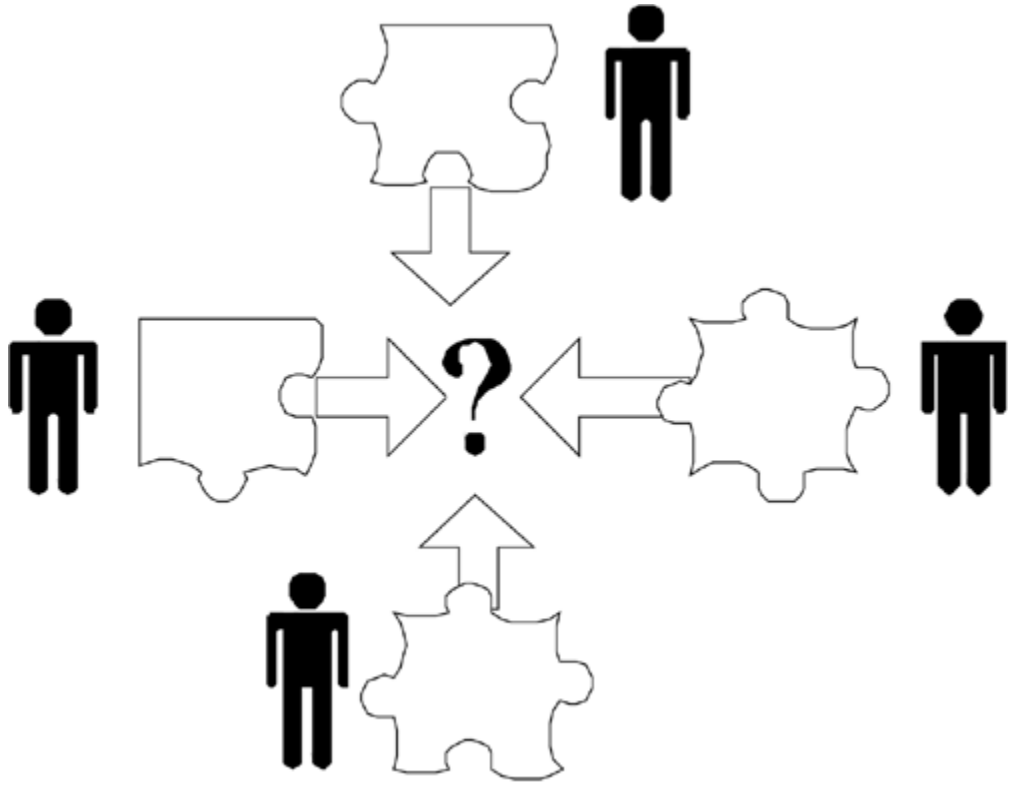

O private system build não garante que o software falhe funcionalmente. Para verificar se as funcionalidades do sistema funcionam bem, precisamos realizar um Smoke Test. Se o sistema é muito grande, esse teste pode não ser viável, pois pode consumir muito tempo e talvez seja melhor esperar para fazê-lo no build de integração.

## 2.6 Integration Build

Todos os desenvolvedores trabalham em suas próprias áreas de trabalho e em muitas dessas áreas são realizados trabalhos sobre mudanças independentes que precisam funcionar juntas e todo o sistema precisa funcionar bem. Este padrão fornece mecanismos para ajudar a garantir que o código do sistema sempre funciona (build).

Como muitos desenvolvedores estão realizando mudanças sobre o código, não é possível para um único desenvolvedor garantir que 100% do sistema funciona depois que as mudanças são integradas na mainline. O melhor que poderia ser feito seria cada desenvolvedor verificar se cada parte do sistema funciona, mas isso iria consumir bastante tempo. Por outro lado, quando o build do sistema falha, bastante tempo também é gasto, pois todos os desenvolvedores são afetados e pode não ser fácil encontrar o problema.Como a figura 12 mostra, integração pode ser comparada a um quebra – cabeça:

**Figure 12 - Integração pode ser um quebra - cabeça**



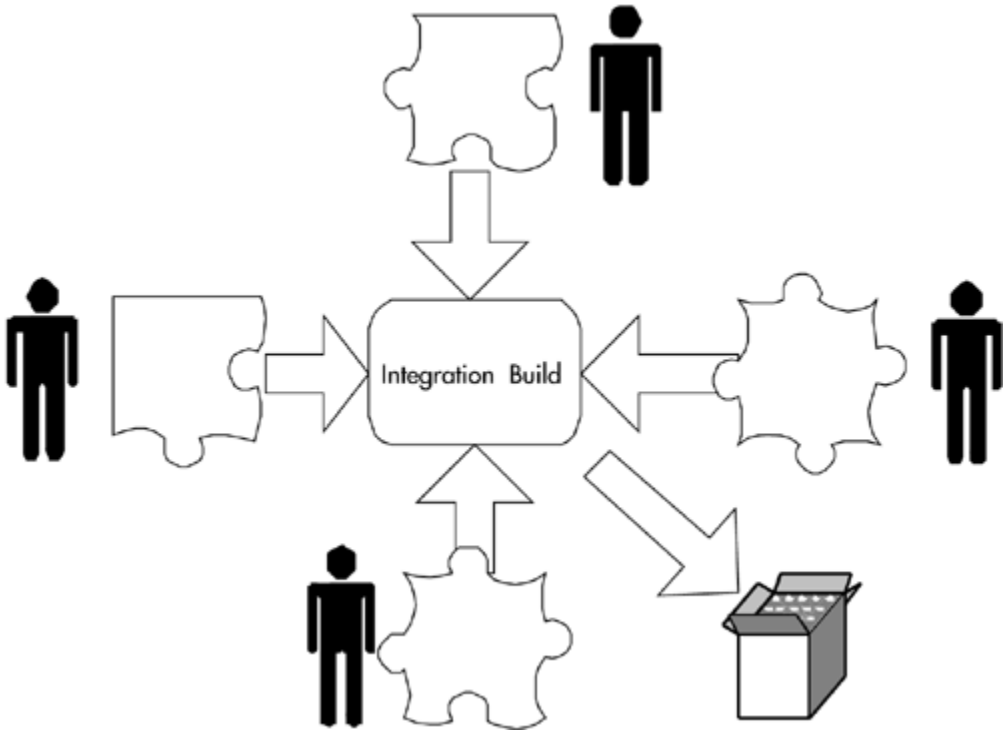
Um completo, centralizado build pode identificar vários problemas na integração, mas não pode atuar de forma preventiva, pois as mudanças já foram publicadas e o problema já existe.

O processo de build poderia ser:

- Reproduzível
- Tão rigoroso quanto um build de release
- Automatizado.
- Possuir um mecanismo de comunicação para reportar erros e inconsistências. Quanto mais rápido os erros sejam identificados, mais rápido eles serão corrigidos. A notificação também torna mais fácil rastrear as mudanças que causaram as falhas no build

O processo de build de integração verificará se as peças do quebra-cabeça são compatíveis como ilustra a figura 13, abaixo:

**Figura 13 - O processo de build de integração**




Para realizar o build, podemos criar uma área de trabalho de interação que contenha todos os componentes sendo integrados. Podemos determinar a freqüência de realização dos builds baseados nos seguintes fatores:

- Quanto tempo leva para se fazer o build?
- Com que freqüência as mudanças acontecem?

Se o sistema consome muito tempo para realizar um build ou se a freqüência de mudanças é baixa, podemos fazer builds diariamente, por exemplo. Se o sistema consome pouco tempo realizar o build, podemos realizá-lo a cada check-in. Embora isso possa parecer bastante custoso, pode ser mais fácil determinar a seqüência de mudanças que ocasionaram a falha do build. Se os builds falham sempre pelo mesmo motivo, poderíamos adicionar procedimentos pré-check-in para identificar esses problemas.

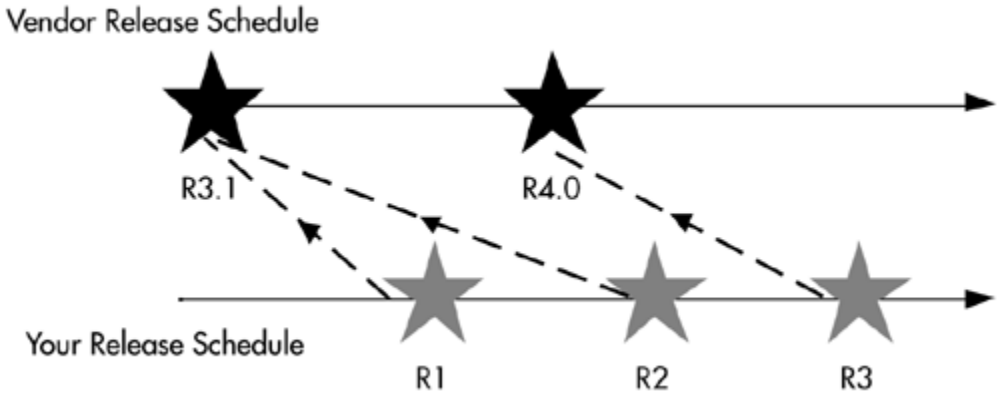
Mesmo que o sistema funcione no build de integração, isso não garante que o mesmo não possua erros funcionais. Para garantir isso, podemos executar um Smoke Test e ou um teste de regressão após o build de integração e promover a qualidade do build.

## 2.7 Third Party Codeline

A codeline está associada com um conjunto de componentes externos que são entregues ao cliente junto com o produto. Podemos customizar alguns desses componentes quando necessário. Isso gerará várias versões desses componentes e essas precisam ser associadas a cada release do produto. Quando criamos uma área de trabalho ou quando empacotamos o build para distribuição, precisamos das versões corretas desses componentes associados ao release.Este padrão mostra como rastrear componentes de terceiros (externos) da mesma forma que rastreamos o próprio código do sistema.

A essência de controle de versão do código fonte e do gerenciamento de release é para identificação dos componentes que juntos reconstroem uma determinada versão do produto. Precisamos reconstruir versões passadas do produto para verificar erros, então é necessário saber quais versões dos componentes externos estão relacionadas aquela versão do software. Quando os componentes são todos internos, podemos simplesmente rotulá-los quando fizermos o release do sistema. Mas ciclos de releases de componentes externos são diferentes dos ciclos de release do sistema, como mostra a figura 14, abaixo:

**Figura 14 – Releases do sistema e releases dos componentes externos são diferentes**



Precisamos identificar rapidamente quais versões dos componentes externos estão associados com quais versões do produto. Algumas empresas mantêm uma lista com essas informações.Mas usar uma lista pode causar problemas durante o desenvolvimento, pois desenvolvedores podem esquecer de consultar ou de atualizar a lista e trabalhar com versões inconsistentes dos componentes.Algumas vezes, por exemplo, os componentes adquiridos precisam ser adaptados para o produto ou podem precisar de uma correção de um erro se temos acesso ao código fonte do componente. Mesmo para componentes binários, os quais não podemos adaptar, precisamos ainda associar os seus releases aos releases do produto.

Devemos usar o sistema de controle de versão para controlar ambas as versões dos componentes externos e as versões do produto. Podemos usar diferentes branchs para as versões dos componentes externos e de suas versões customizadas. Quando recebemos uma nova versão desses componentes podemos fazer merge dessas com as versões customizadas. O sistema de controle de versão manterá as informações para saber quais versões dos componentes internos e externos são relacionadas.

O código fonte dos componentes de terceiros são aqueles fornecidos por alguém externos a organização. "Plug-ins" e extensões para um framework externo não são código fonte de terceiros e podem ser tratados como código do produto. Por exemplo, uma biblioteca que faz parsing em XML, pode ser um código fonte dos componentes de terceiros.

Para controlar versão dos componentes de terceiros podemos seguir os passos abaixo:

1. Criar uma codeline para esses componentes
2. Adicionar os componentes ao apropriado diretório da codeline.
3. Rotular os componentes com as suas versões.
4. Imediatamente criar uma branch dessa codeline. Todos os projetos que usarem esta versão dos componentes usarão diretamente do branch, onde

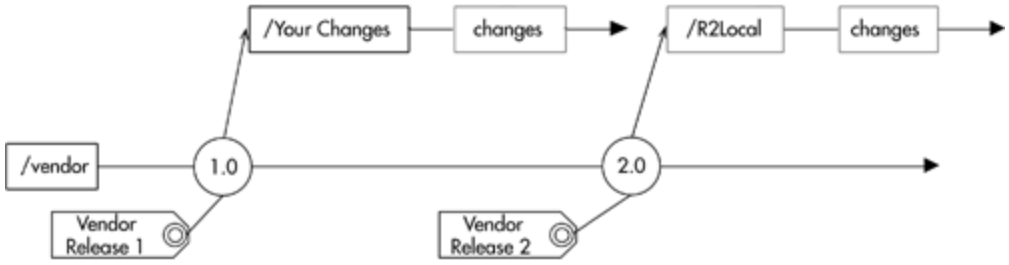
será possível customizá-las, quando possuímos o código fonte do componente.

5. Quando uma nova versão dos componentes for adquirida, adicionar ela a mainline da codeline inicial.
6. Criar outro branch com essas novas versões e mesclar as mudanças relevantes do branch anterior no novo branch.

OBS.: Se os componentes são estáveis ou nós nunca os customizamos, não é necessário criar um branch. O custo de branch neste caso é pequeno, e dá mais flexibilidade para mudar os componentes depois, se necessário.

A figura 15, mostra a codeline dos componentes de terceiros:

**Figura 15 – Codeline dos componentes de terceiros**




Dessa forma, podemos facilmente reproduzir versões anteriores dos releases do produto incluindo as versões corretas dos componentes de terceiros. Customizações diferentes podem ser facilmente isoladas e reproduzidas de tal forma que podemos identificar o que mudou para um determinado release. Diferenças entre releases dos componentes de terceiros podem ser facilmente isoladas e reproduzidas para auditar o que mudou de release para release. Podemos ainda, quando fomos liberar uma versão do produto, rotular os componentes de terceiros com o mesmo rótulo que definimos para o release do produto. Isso mantém a rastreabilidade dos releases do produto sobre os releases dos componentes de terceiros e facilita quando formos reproduzir um build anterior, pois para incluir as versões corretas dos componentes, basta realizar um “check-out” pelo rótulo do release do produto para uma área de trabalho.

## 2.8 Task Commit Level

Um build de integração é mais fácil de debugar se nós sabemos quais mudanças foram realizadas sobre o sistema naquele build.Este padrão discute como balancear as necessidades para estabilidade, velocidade e atomicidade.

Muitas mudanças são realizadas sobre o código do produto, tais como correção de erros, melhorias e novas funcionalidades. Essas mudanças precisam ser rastreadas para que possamos saber quais características foram corrigidas, melhoradas ou adicionadas a cada build / release do sistema. O histórico das versões da ferramenta de controle de versão poderia refletir como os arquivos mudam em relação as suas funcionalidades.

Para adicionar uma funcionalidade ou corrigir um erro, poderíamos realizar mudanças em vários arquivos do código fonte. E cada mudança introduz uma potencial instabilidade na mainline. Então, seria interessante poder remover (roll-back) uma mudança se ela causar um problema ao sistema. Para isso temos que saber tudo o que foi afetado pela mudança. E quanto maior for o impacto da mudança sobre os arquivos, maior será o trabalho para desfazê-la. Por isso temos que avaliar a granularidade de uma mudança.Essa tarefa nem sempre é trivial, pois algumas vezes para corrigir um erro precisamos alterar apenas um arquivo, e realizamos um check-in atômico. Outras vezes, uma mudança envolve vários componentes de vários sistemas que devem todos ser publicado (check-in) ao mesmo tempo, mas muitas vezes isso não acontece e dificulta bastante a remoção de uma mudança. Quando a granularidade da mudança é alta fazemos vários check-in por unidades de trabalho e perdemos em performance devido ao tempo gasto com vários check-ins. Se por outro lado, submetemos várias unidades de trabalho em um único check-in, perdemos a habilidade de voltar (roll-back) as mudanças.

Desejamos manter a codeline estável e associar mudanças com as funcionalidades do sistema e que cada check-in reflita o trabalho de uma tarefa consistente. Para conseguirmos isso devemos avaliar a complexidade da tarefa,

quantos arquivos ou componentes precisam ser atualizados para implementá-la, e qual o risco de implementar / não – implementar a tarefa.

Exemplos dessas tarefas são:

- Correção de um erro de relatório
- Mudança para usar uma nova API para todo o sistema
- Melhoria no cadastro de uma atividade
- Uma nova funcionalidade

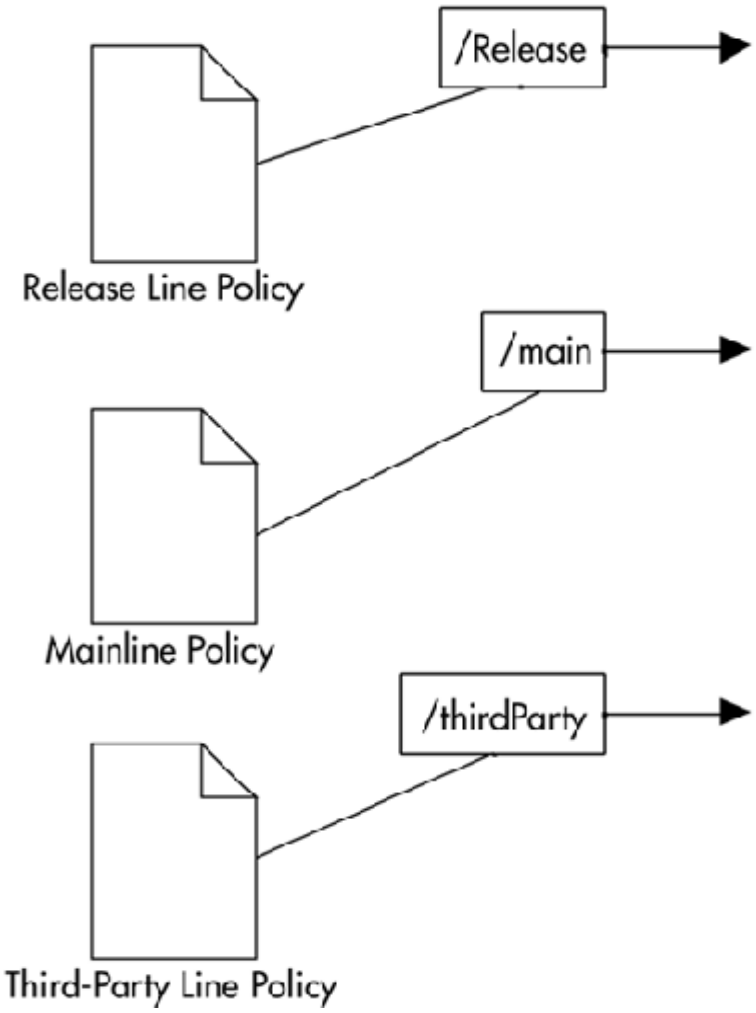
## 2.9 Codeline Policy

Quando temos várias codeline, desenvolvedores precisam saber como tratar cada uma. Uma *Release Line* pode ter regras de restrição para quando e como deve-se fazer check-in das mudanças, já uma *Active Development Line* poderia ter regras mais flexíveis. Este padrão descreve como estabelecer as regras para cada codeline de acordo com o seu propósito.

Cada codeline tem um propósito diferente; uma codeline pode proposta para consertar erros em um determinado release; uma outra codeline pode ser usada para migrar o código para uma outra plataforma; já outra pode ser para o desenvolvimento dia-a-dia. Desenvolvedores precisam saber quais codeline eles poderiam usar e quais políticas devem ser seguidas nessas codelines.

Podemos identificar diferentes codeline por seus nomes. O nome de uma codeline pode ser relacionado a algum substantivo que se relacione com o seu propósito. Por exemplo, um release line pode ser muito restrito ou menos restrito, dependendo da estratégia e visão da organização. E pode ser difícil encontrar uma boa e não ambígua convenção de nomes.  A figura 15 mostra como relaciona cada codeline a sua política.

**Figura 16 - Cada codeline precisa de diferentes políticas**




Uma vez que seja decidido o grau de estabilidade que uma codeline precisa ter e como manter essa estabilidade através do processo, é necessário informar aos desenvolvedores as políticas e forçá-los a seguí-las.

A política deve ser breve e poderia incluir os seguintes itens:

- O tipo de trabalho encapsulado pela codeline, tais como: desenvolvimento, manutenção, ou um específico release, função, ou subsistema.
- Quando e como os arquivos podem ser checked in, checked out, branched, e merged
- Restrições de acesso aos arquivos

- Relações de Importação/exportação entre codeline
- A duração do trabalho ou condição para “aposentar” a codeline
- A carga de atividade esperada e a freqüência de integração

OBS.: Uma boa política deve ter de um a três parágrafos, com no máximo uma página.Deve ser especificado só o que é essencial. Se a ferramenta de controle de versão permitir associar comentários na criação dos branch, a política pode ser informada nesses comentários. Isso facilita o acesso para os desenvolvedores saberem a política da codeline que estão trabalhando.

Abaixo seguem alguns exemplos de políticas de codelines:

- Codeline de desenvolvimento — assim que finalizadas, as mudanças podem ser publicadas (checked-in); componentes afetados precisam ser recompilados e as referencias atualizadas.
- Codeline de release — é necessário fazer um build privado e realizar testes de regressão antes do check-in; check-ins são limitados à correção de erros; nenhuma nova funcionalidade ou melhoria podem ser checked in; depois do check-in, o branch é congelado até ser testado e auditado.
- Mainline — todos os componentes precisam compilar e passar no teste de regressão; e somente completas e testadas tarefas podem ser checked in.

A política pode ser forçada a ser seguida usando algum mecanismo que a ferramenta de controle de versão suporte, tais como gatilhos. Auditorias também podem ser feitas sobre a política.

### 2.10 Smoke Test

Um *Integration Build* ou um *Private System Build* são usados para verificar problemas de integração em build - time. Mas, mesmo o código funcionando (build), precisamos ainda verificar se as mudanças não geram erros em run - time. Esta verificação é essencial para manter a *Active Development Line*. Este padrão

está relacionado às decisões necessárias que precisam ser feitas para validar um build.

Uma primeira decisão é saber quais testes devem ser executados antes do check-in. Podemos escrever testes para as partes mais críticas do sistema ou aquelas que apresentam erros com mais freqüência, mas é difícil desenvolver testes completos.Poderíamos também escrever testes para testar todas as partes do sistema, mas isso pode consumir bastante tempo, e atrasar o progresso do projeto. Testes pré-check-in demorados, encoraja aos desenvolvedores fazer grandes mudanças num mesmo check-in e perder a capacidade de remover mudanças.  O escopo dos testes não pode ser exaustivo. Poderia testar as funcionalidades básicas. Idealmente, ele poderia ser automatizado tal que o custo para realizá-lo seja baixo. O *smoke test* não precisa ser tão profundo quanto o teste de integração. Uma suíte de testes unitários pode ser uma base para os *smoke tests.*

Executar um *smoke test* com cada build não remove a responsabilidade dos desenvolvedores testar suas mudanças antes de submetê-las ao repositório. Desenvolvedores poderiam executar um *smoke test* manualmente antes de check-in uma mudança. Um *smoke test* é mais usados em correções de erros e verificar as iterações entre as existentes e as novas funcionalidades. Quando novas funcionalidades ao sistema, o smoke test deve ser atualizado para testar essa nova funcionalidade.

Um smoke test poderia:

- Ser rápido de executar, onde "rápido" é relativo a cada situação
- Ser suportado por alguma ferramenta de teste
- Cobrir todos os cenários a serem testados
- Ser acessível tanto pelos desenvolvedores quanto pelos testadores
- Mais caixa – preta que caixa - branca

## 2.11 Unit Testing

Algumas vezes um *Smoke Test* não é bastante para testar uma mudança em detalhes quando estamos trabalhando em um módulo, especialmente quando estamos trabalhando em um código novo. Este padrão mostra como testar de forma detalhada as mudanças para garantir a qualidade das codelines.

Verificar se uma classe, modulo, ou função ainda trabalha depois que é realizada uma mudança é um procedimento básico que ajuda a manter a estabilidade do desenvolvimento do software. É também mais fácil entender o que pode dá errado no nível unitário do que no nível de sistema. Por outro lado, testar em pequenas escalas pode ser tedioso.

Bons testes unitários têm as seguintes propriedades:

- É automatizado.
- Tem uma boa granularidade. Algum método relevante de alguma classe poderia ser testado. Não é necessário escrever testes para verificar métodos triviais como “sets” e “gets” de uma classe. Resumindo, devemos testar o que pode provocar erros.
- É isolado. Um teste unitário não interage com outros testes, pois a falha de um teste pode ocasionar a falha de outros testes.
- Devem ser atualizados quando o alvo do teste muda.
- Deve ser simples de executar.

Os testes unitários poderiam ser executados:

- Enquanto as mudanças estão sendo codificadas
- Apenas antes de check-in uma mudança e depois de atualizar a área de trabalho.

Os testes unitários também podem ser executados quando estamos tentando encontrar um problema com um *smoke test* ou um teste de regressão, ou em respostas a um problema solicitado pelos usuários.

## 2.12 Regression Test

Um Smoke Test é rápido mais não exaustivo. Pata ele ser efetivo, é necessário fazer um desgastante trabalho de exaustivos testes também. Se quisermos estabelecer um release candidato, por exemplo, precisamos garantir que o código está robusto. Este padrão explica como gerar builds que não são piores do que o último build.Em outras palavras, explica como garantir que o produto não é pior depois das mudanças do que ele era antes.

Sistemas de software são complexos e quando um erro é corrigido existe uma substancial probabilidade de inserir outros erros. Por outro lado, sem mudar, não é possível evoluir o produto e o tamanho do impacto da mudança é difícil de medir, especialmente em termos de como uma unidade de código interage com o resto do sistema.

Podemos executar testes de regressão no sistema se quisermos garantir a estabilidade da codeline. Isso pode ser antes de liberar um release, ou antes, de realizar uma mudança de alto risco. Um teste de regressão é um teste caixa – preta que testas as novas e as antigas funcionalidades do sistema. Ele pode identificar uma falha no nível de sistema, mas não pode necessariamente identificar a sua causa.Teste de regressão é projetado para garantir que o software não está “caminhando para traz” (ou regredindo). Sempre devemos executar os mesmos testes a cada ciclo de regressão. Devemos adicionar mais testes a suíte de teste de regressão quando encontramos mais condições ou itens problemáticos; se um problema aconteceu uma vez, ele pode acontecer novamente; só devemos remover casos de testes por razões muito fortes.

Como testes de regressão consomem muito tempo, talvez não seja viável executá-lo a cada check-in (a menos que tenhamos recursos para isso). Mas existem vantagens de possuir um procedimento automatizado de executar testes de regressão depois de cada mudança para identificar o ponto no qual o sistema regrediu em check-in – time. Podemos também executar testes de regressão a cada build noturno.

## 2.13 Private Versions

Algumas vezes, os desenvolvedores querem avaliar rapidamente uma mudança complexa que pode “quebrar” o sistema em uma *Active Development Line*. Eles querem experimentar essa mudança sem publicá-las e ao mesmo tempo se beneficiar do sistema de controle de versão para controlar as versões intermediárias da mudança. Este padrão descreve como manter rastreabilidade local sem afetar a codeline do produto.

O sistema de controle de versão fornece a habilidade de reverter o sistema a estados anteriores quando remove mudanças. Quando o desenvolvedor está explorando implementações, ele pode não querer publicar suas mudanças até que ele tenha certeza da sua escolha. Mas, ele também pode querer gerar versões intermediárias da sua implementação para poder revertê-las se, por exemplo, as implementações durarem vários dias. O desenvolvedor poderia realizar todas as mudanças na sua área de trabalho e não publicar trabalhos intermediários. Mas se ele faz isso ele perde  a habilidade de reverter mudanças. Ele pode também usar uma técnica “ad hoc” de salvar cada estado do trabalho em um pasta local separada. Infelizmente nenhum desses métodos é adequado.

Existem muitas maneiras de resolver implementar esse padrão. Uma delas é ter um repositório privado para cada desenvolvedor — por exemplo, um repositório local do CVS ou um específico branch para o desenvolvedor. E o desenvolvedor realiza os check-ins das versões intermediárias para o repositório privado ou para o branch privado e quando tiver concluído a mudança publicar a mesma para a codeline ativa.Algumas ferramentas implementam o padrão *private versions*, através de níveis de promoção, onde só publicado para toda a equipe o que é recomendado.

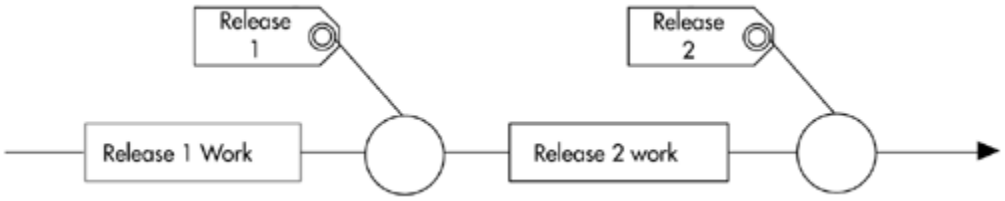
## 2.14 Release line

Durante o ciclo de desenvolvimento / manutenção do produto, várias versões são distribuídas e é necessário controlar as mudanças sobre essas versões já em produção e ao mesmo tempo continuar evoluindo o produto para o próximo release. Este padrão mostra como desenvolver mudanças sobre os releases em produção sem interferir no desenvolvimento do release corrente.

Cada versão do produto em produção precisa evoluir independentemente do desenvolvimento corrente. Para muitas empresas seria ideal que os clientes atualizassem suas versões para a versão mais recente do produto, com todos os erros corrigidos. Mas a realidade é que em muitas circunstâncias é necessário corrigir um erro numa versão anterior do produto. Numa correção de um erro urgente, por exemplo, poderia a última versão do produto ainda não está totalmente implementada e o cliente não poder esperar até que a mesma seja finalizada.Nesse cenário temos que conduzir o desenvolvimento do próximo release e ao mesmo atender (em curto prazo) às solicitações de erros e melhorias dos clientes.

È necessário identificar qual código fonte pertence a cada release e qual código está na mainline para o próximo release. Uma forma de identificar cada releases é através de rótulos aplicados a mainline a cada liberação de cada release e continuar desenvolvendo o próximo release a partir dessas versões rotuladas. A figura 16 ilustra esta técnica. Mas, esta forma não permite realizar mudanças sobre um release de forma independente da mainline.

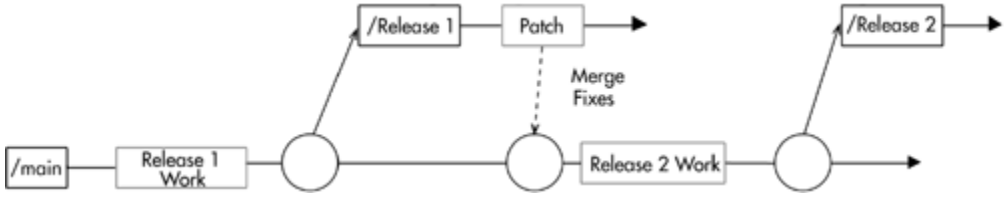
**Figura 17 - Realizando todas a mudanças na mainline**




Para obter independências entre os releases é necessário criar um branch para cada release quando cada estes são finalizados. E caso exista alguma

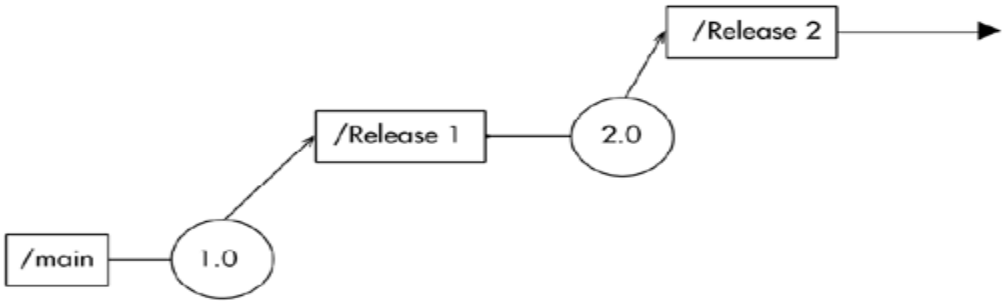
mudança (como correção de erros) que devem ser realizadas tanto nos branchs quanto na mainline, é necessário mesclar as mudanças (merge) ou duplicar o trabalho. A figura 18 mostra essa situação:

**Figura 18 – Um branch para cada release**




Em alguns casos podemos ter cada cliente com uma versão / release do produto e temos que rastrear múltiplos releases que são derivados de outros releases. Isso pode ser modelado pela estrutura em escada de branchs como mostra a figura 19:

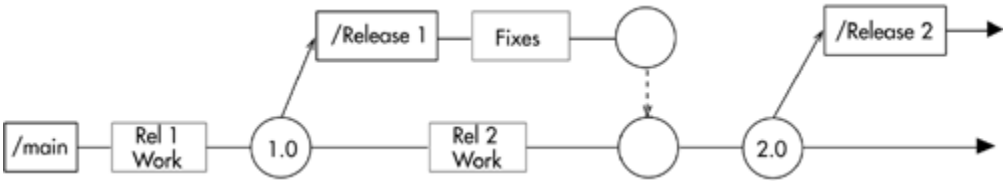
**Figura 19 – Estrutura em escada de branchs dependentes**




O problema desse modelo é que não se sabe o que é comum entre os releases e a manutenção (correção de erros e melhorias) num determinado release pode ser incompatível com alguma funcionalidade já implementada no próximo release.

Ao invés de tentar acomodar mudanças de manutenção e mudanças para o próximo release na mesma codeline, devemos separar manutenção e desenvolvimento em codelines separadas. Todas a correções e melhorias serão implementadas na codeline de manutenção, e os esforces para o próximo release serão alocados na codeline de desenvolvimento. È necessário verificar se as mudanças para a codeline de manutenção estão sendo regularmente propagadas para codeline de desenvolvimento. A figura 19 mostra esta estrutura:

**Figure 20 – Codelines de manutenção (Release lines) e de desenvolvimento (mainline)**




É necessário também propagar correções de erros da mainline para as releases lines quando possível. Desta forma, mesmo o código da mainline tendo progredido, ainda pode-se realizar mudanças sobre o código das releases line de forma independente. Quando o próximo release for finalizado, rotule a mainline com o nome do release e crie uma branch para o mesmo. Erros para o release serão corrigidos no branch correspondente e depois propagados para mainline antes do próximo release. O código de uma release line será “aposentado” quando a empresa não dá mais suporte a versão corresponde a da release line.

## 2.15 Release-Prep Code Line

Em algum milestone do processo de desenvolvimento de um produto, nos deparamos com a seguinte situação: como estabilizar o release corrente e ao mesmo tempo (sem congelar o código fonte) permitir o desenvolvimento de novas

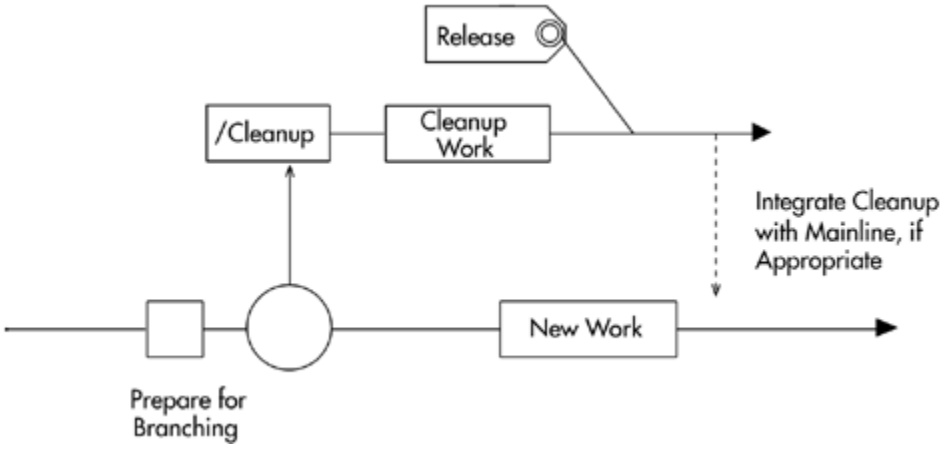
mudanças para o próximo release. Este padrão mostra como implementar este tipo de situação.

Antes de um release ser liberado para entrar em produção, existe freqüentemente um esforço para estabilizá-lo e deixá-lo pronto para ser entregue aos clientes. Existem também correções dos erros de “última hora” e detalhes dos “últimos minutos”, relacionados ao empacotamento, disponibilização e instalação do release. Durante esse período de estabilização, é aconselhável não realizar algum novo trabalho na codeline como, por exemplo, adicionar novas funcionalidades para o próximo release, pois novos problemas podem ser introduzidos. Nessa fase poderiam existir também restrições de check-in e políticas de qualidade que garantissem a integridade da codeline.

Uma solução é congelar as atividades na codeline até que as atividades de estabilização do release sejam finalizadas e o release fique pronto para ser distribuído. Se tudo ocorrer corretamente, isso pode durar só um dia ou menos. No entanto, essa estabilização pode durar semanas ou meses e envolver grande parte da equipe. Mas nessa situação, muitos recursos são desperdiçados, pois parte da equipe que não está envolvida na estabilização não podem desenvolver novas implementações para construir o próximo release.

A solução defendida por este padrão, é a seguinte: criar um branch da mainline em estabilização e nele, implementar as atividades para estabilizar o release, enquanto novas atividades para o próximo release serão implementas na mainline. Por outro lado, se esse branch for criado de forma prematura, existirá uma grande demanda de merges para serem mesclados do branch (codeline atual), após a estabilização, para a mainline propagando, dessa forma, a correção dos erros do release atual para a próxima versão. A figura 21 ilustra esse padrão:

**Figura 21 - Release-Prep Code Line**




Quanto mais estabilizado estiver a release corrente, no momento da criação do branch, menos merges serão realizados entre ele e a mainline.Por outro lado, se o branch só for criado quando o release estiver num grau avançado de estabilização, a mainline ficará congelada durante esse período e esta é a situação que deve ser evitada. Deve-se encontrar um ponto de equilíbrio para criação do branch, onde o tempo de congelamento e a quantidade de esforces com merges sejam aceitáveis.

Trabalhar de forma concorrente em dois releases permite adicionar solicitações de mudanças na mais apropriada codeline.Poe exemplo: correções de erros críticos e melhorias na release corrente, podem ser implementadas e entregues sem imediatamente impedir o desenvolvimento do próximo release. "Patches" dos releases podem ser periodicamente liberados sem grande impacto no desenvolvimento das próximas versões. Políticas podem definir quando e como as mudanças dos “patches” serão propagadas para a mainline e para os outros branchs.

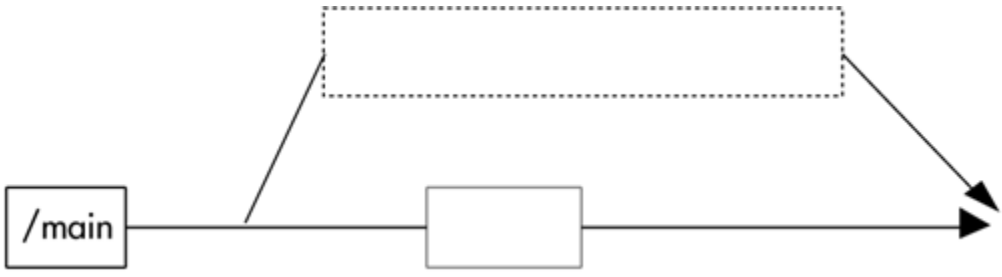
**2.16 Task Branch**

Alguma implementação de uma solicitação de mudança pode durar muito tempo, e passos intermediários podem comprometer a consistência / integridade da mainline na qual está sendo desenvolvido o release corrente. Este padrão

descreve como conciliar o desenvolvimento de tarefas que consomem muito tempo com a garantia de uma mainline ativa (*Active Development Line*).

O sistema de controle de versão geralmente é usado como um mecanismo para manter toda a equipe do projeto consciente do que cada membro está fazendo (comunicação). Sobre condições normais, cada membro freqüentemente executa check-in das suas mudanças nos artefatos. No entanto, algumas dessas mudanças podem desestabilizar a codeline ativa. Por exemplo, um trabalho de troca da camada de persistência (acesso à base de dados) de um sistema não pode facilmente ser feito em estágios, e ao mesmo tempo não é salutar esperar uma semana ou mais para finalizar um conjunto de mudanças e finalmente publicá-las (check-in), pois outros membros que estão trabalhando na mesma tarefa, precisam sincronizar seus trabalhos. A figura 22 ilustra essa situação:

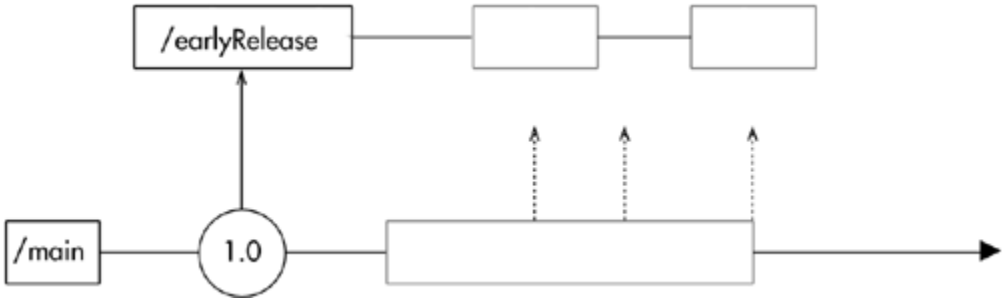
**Figura 22 – Algumas tarefas são temporariamente isoladas**




Um outro exemplo de situação onde um pequeno grupo de desenvolvedores está trabalhando em uma tarefa que pode causar conflito é o seguinte: o fechamento de um release está se aproximando, mas uma parte da equipe está trabalhando em uma nova funcionalidade para o próximo release. È necessário que essa parte da equipe compartilhe suas mudanças através de alguma ferramenta de controle de versão, mas as mudanças não podem afetar o desenvolvimento do release corrente. Se for somente uma pequena parte da equipe que está trabalhando nessas mudanças, o “overhead’ de criar um branch (*Release Line*) pode ser muito grande, porque todos os demais membros

trabalharão no branch e a mainline precisa ser sincronizada com ele, resultando em um esforço demasiado para os merges. Isso é ilustrado na figura 23:

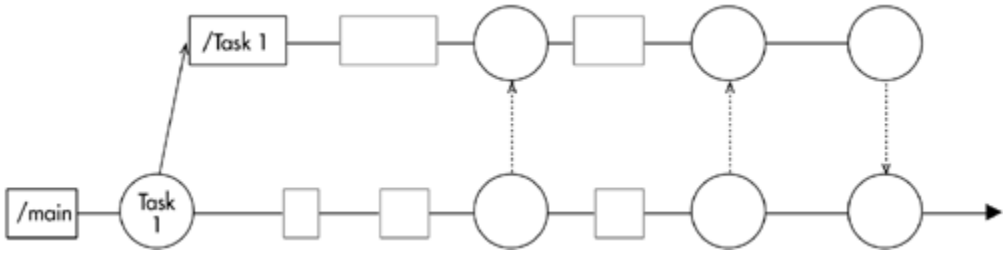
**Figura 23. Criar uma release line prematura é um problema**




A solução defendida por este padrão é criar um branch para cada tarefa que adicione mudanças significantes à codeline. O padrão *Task Branch* funciona como um mecanismo que busca minimizar os riscos de implementar a tarefa e "quebrar" a integridade da codeline. O check-in para o *Task Branch* ativo será realizado quando a tarefa estiver finalizada e todas as mudanças sobre os arquivos estiverem sido testadas. Quando o *Task Branch* for finalizado, deverá realizar-se merges com a(s) apropriada(s) codeline (como uma transação).

É importante também integrar mudanças da codeline ativa na *Task Branch* freqüentemente para que a integração final da *Task Branch* na codeline seja tão "suave" quanto possível (poucos merges conflitantes). A figura 24 ilustra o padrão.

**Figura 24. Task Branch**




## 3. Suporte das ferramentas aos padrões de GCS

Este capítulo descreve como os padrões de GCS abordados no capítulo 2 são implementados pelas mais utilizadas ferramentas de GCS utilizadas mundialmente. Objetivo é catalogar um conjunto de ferramentas que suportam ou não os padrões utilizados pelas organizações. Isso irá ajudar tanto na aquisição dessas ferramentas quanto no uso mais efetivo das mesmas. A tabela 1 lista as ferramentas, e para cada uma delas, o seu fabricante e seu endereço eletrônico onde podem ser encontradas mais informações sobre elas.

**Tabela 1. Lista das ferramentas avaliadas**

| Ferramenta | Fabricante | Endereço eletrônico |
|---|---|---|
| Repository Manager | Innovative | http://www.innovative.inf.br/produtos/ |
| VSS—Visual Source Safe | Microsoft | http://msdn.microsoft.com/ssafe/ |
| CVS— Concurrent Versions System | Open Source | http://www.cvshome.org/ |
| Perforce | Perforce Software | http://www.perforce.com/ |
| BitKeeper | BitMover Inc. | http://www.bitkeeper.com/ |
| AccuRev | AccuRev Inc. | http://www.accurev.com/ |
| ClearCase | Rational Software | http://www.rational.com/products/clearcase/ |
| UCM—Unified Change Management | Rational Software | http://www.rational.com/products/clearcase/ |
| CM Synergy | Telelogic | http://www.telelogic.com/products/synergy/ |
| StarTeam | Starbase | http://www.starbase.com/products/starteam/ |
| PVCS Dimensions | Merant PVCS | http://www.merant.com/pvcs |
| PVCS Version Manager | Merant PVCS | http://www.merant.com/pvcs |

| | | |
|---|---|---|
| MKSource Integrity | MKS Inc. | http://www.mks.com/products/sie/ |

### 3.1 Innovative Repository Manager

O *Innovative Repository Manager (RepositoryM)* é uma ferramenta recente no mercado, mas que possui características tão poderosas quanto as ferramentas mais respeitadas do mercado. Uma das suas principais características é o gerenciamento de múltiplos repositórios, permitindo inclusive que um projeto pertença a mais de um servidor. Outra característica muito poderosa é o suporte a definição de políticas e aos processos de gerenciamento de mudança integrado com o controle de versão e ao gerenciamento do projeto.

A ferramenta é orientada a tarefa, e através dessa abstração é possível avaliar o impacto de uma mudança sobre os requisitos, sobre os artefatos, sobre outras mudanças e sobre o cronograma do projeto. Essa rastreabilidade é mantida através de redes de dependência construídas visualmente na ferramenta.

A ferramenta permite ainda construir *tamplates* de árvores de diretório que podem ser reusados por todos os projetos.O suporte a branch e merge é visual e tão poderoso quanto o CVS.

A avaliação dos padrões versus o RepositoryM é mostrada na tabela 2.

**Tabela 2. RepositoryM versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | Implementação | **Comentário** |
|---|---|---|
| Repository | Repository | Suporta múltiplos repositórios fisicamente distribuídos |
| Development Workspace | Working Folder | Podem ser criados vários *Working Folder*, para cada projeto e um *Working Folder* pode acomodar arquivos pertencentes a repositórios diferentes. |

| | | |
|---|---|---|
| Codeline | Branch | Existe também uma aba chamada *Branch View,* onde todos os branchs podem ser acessados, manipulados. |
| Codeline Policy | Control Level | Podem ser definidas políticas pré e pós cada uma das operações. Como por exemplo, bloquear um arquivo após o *checkin*, ou disparar um gatilho para uma ferramenta de build automático ou teste. Ou ainda, só permitir checkout de um arquivo se for através de uma autorização formal (solicitação de mudança / tarefa). |
| Private Versions | Checkin / Delivery | Permite controlar versão do arquivo com checkin, mas somente liberá-lo quando finalizar a tarefa sendo realizada sobre o mesmo (Delivery). Isso evita que outras pessoas utilizem versões de arquivos com tarefas incompletas / inconsistentes. |
| Change Task | Task | Engloba esforço, custo, prazo, responsável e arquivos atualizados. É base para o release notes automático gerado pela ferramenta. |
| Workspace Update | Update | |
| Task-Level Commit | Delivery | Publica as mudanças e finaliza as tarefas (em execução) associadas. |
| Task Branch | Branch | |
| Label | Baseline / TAG | As baselines são exibidas em uma seção da ferramenta e podem ser promovidas de acordo com os níveis de promoção configurados pelo usuário. |
| Third Party Codeline | Project ou Branch | As ferramentas de terceiros podem ser tratadas como um projeto e compartilhada com os demais |

| | | |
|---|---|---|
| | | projetos que as utilizam. Existem também os *links*. |

### 3.2 VSS - Visual Source Safe

VSS é uma das mais usadas ferramentas de controle de versão integrada aos ambientes de desenvolvimento da Microsoft (IDE) tal como Visual C++. VSS é considerada uma ferramenta de baixa capacidade no suporte a branching e desenvolvimento paralelo. Não é recomendado para projetos que regularmente requerem múltiplas codelines. Um repositório em VSS corresponde a uma base de dados acomoda um ou mais projetos (funciona como uma mainline).

Um *development workspace* é criado em VSS quando associamos um projeto com um diretório de trabalho. O diretório de trabalho é populado através da operação GET que cria localmente uma cópia das últimas versões dos arquivos com acesso READ-ONLY. A operação CHECKOUT é usada para copiar uma cópia com permissões de escrita para o diretório de trabalho. Mudanças do diretório de trabalho podem ser publicadas para a codeline através da operação CHECKIN.

A avaliação dos padrões versus o VSS é mostrada na tabela 3.

Tabela 3. VSS versus Padrões de GCS

| **Conceitos / Padrões de GCS** | **Implementação** | **Comentário** |
|---|---|---|
| Repository | Master project | Suporta múltiplos projetos |
| Development | Working | Área local de trabalho |

| | | |
|---|---|---|
| Workspace | directory | |
| Codeline | Share and branch | Branchs são limitados, não suportam eficientemente branchs em mais de dois níveis de profundidade. Os arquivos pertencentes a um Branch são cópias (não links) e podem evoluir independentemente da codeline da qual ele foi originado. |
| Codeline Policy | **Não suporta** | |
| Private Versions | **Não suporta** | |
| Change Task | **Não suporta** | |
| Workspace Update | Get project | |
| Task-Level Commit | Check-in project | |
| Task Branch | **Não suporta** | |
| Label | Label project | |
| Third Party Codeline | Label / Branch | Não há uma implementação direta deste padrão, mas pode ser implementado através de branchs e/ou Labels. |

### 3.3 CVS— Concurrent Versions System

CVS é uma ferramenta open source e é uma das mais conhecidas e usadas ferramentas de controle de versão atualmente.

Os principais comandos usados são: **tag**, **checkout**, **update**, e **commit**. A opção "-R" pode ser usada nos comandos **update**, **commit**, and **tag** para aplicá-

los recursivamente a todos os arquivos do “working directory”. Branchs para uma codeline ou um task branch são criados usando o comando **tag** com a opção "-b". O comando **update** sincroniza a área de trabalho do desenvolvedor com as últimas versões da codeline.

A avaliação dos padrões versus o CVS é mostrada na tabela 4.

**Tabela 4. CVS versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | **Implementação** | **Comentário** |
|---|---|---|
| Repository | Repository | Também conhecido como CVSROOT |
| Development Workspace | Working directory ou working copies | Criado através do comando cvs checkout |
| Codeline | Branch | Criado através do comando cvs tag –b |
| Codeline Policy | **Não suporta** | |
| Private Versions | *Repositórios locais / Branchs* | Implementação não eficiente, pois na maioria dos casos é necessário fazer merges com a codeline ativa. |
| Change Task | **Não suporta** | |
| Workspace Update | Update | cvs update |
| Task-Level Commit | Commit | cvs commit |
| Task Branch | *Branch* | |
| Label | Tag | |
| Third Party | Vendor branch | Como um caso especial, CVS foi |

| | | |
|---|---|---|
| Codeline | | especialmente projetado para suportar o conceito de um “vendor branch” (um Third Party Codeline). O comando cvs import foi projetado para esse propósito. |

### 3.4 Perforce

Perforce é muito usada, simples, mas poderosa ferramenta de controle de versão com excelente suporte a branching e merging. Ela se auto classifica como “a mais rápida ferramenta de controle de versão”. E se orgulha sobre sua performance em operações TCP/IP cliente-servidor.

Um repositório em Perforce é chamado de *Depot*. O Perforce server gerencia centralmente o aceso dos clientes conectados ao depot.

Um Perforce workspace é chamado de *client workspace* e é configurado através de um arquivo de especificação, o *client spec* para criar uma visão dos artefatos do *depot*. Uma codeline é chamada de branch em Perforce. Perforce rastreia as mudanças entre branchs pai e filhos, tal que ela sabe quando um arquivo já tem sido mesclado ou não.

Perforce usa um modelo atômico de transação, onde uma operação ocorre em todos os arquivos pertencentes a operação ou não ocorre em nenhum deles. Dessa forma, não é possível ocorrer um incompleto update ou commit.

Perforce também oferece um sistema de notificação e configuração de gatilhos para disparar antes ou após suas operações.

A avaliação dos padrões versus Perforce é mostrada na tabela 5.

**Tabela 5. Perforce versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | **Implementação** | **Comentário** |
|---|---|---|
| Repository | Depot | |

| | | |
|---|---|---|
| Development Workspace | Client workspace and client spec | *client spec* informa quais arquivos e versões destes compõe o workspace. |
| Codeline | Branch | |
| Codeline Policy | **Não suporta** | |
| Private Versions | *Depot/branch* | Implementação ineficiente |
| Change Task | *Changelist* | |
| Workspace Update | Sync | |
| Task-Level Commit | Submit | |
| Task Branch | *Branch* | |
| Label | Label | |
| Third Party Codeline | *Branch* | |

### 3.5 BitKeeper

BitKeeper se auto considera com um sistema de GCS distribuído e de excelente performance e corretude. Ao contrário da grande maioria das outras ferramentas, BitKeeper não usa o modelo cliente-servidor, usa o modelo replicado de operações peer-to-peer que permite realizar as operações estando desconectado do “master repository”. BitKeeper também oferece o uso de gatilhos(triggers).

A chave para entender o modelo operacional de BitKeeper é considerar cada workspace como um repositório.Mudanças, na forma de “change – sets”, são realizadas em cada workspace, e desenvolvedores podem propagá-las entre os demais workspaces usando asa operações **push** e **pull**. Todas “change-sets” são atômicas em BitKeeper.

Devido a este simples, mas poderoso modelo de workspaces distribuídos como repositórios para transmitir e receber change-sets, ambos codelines e task branchs podem ser representados como workspace em BitKeeper. Esses workspaces operam como um branch.

A avaliação dos padrões versus BitKeeper é mostrada na tabela 6.

**Tabela 6. BitKeeper versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | **Implementação** | **Comentário** |
|---|---|---|
| Repository | Master repository | |
| Development Workspace | Developer repository | |
| Codeline | Integration repository | |
| Codeline Policy | **Não suporta** | |
| Private Versions | *Developer repository* | Cada wokspace funciona como uma repositório e desta forma é possível controlar versão e ao mesmo tempo manter um codeline ativa no *Integration Repository*. |
| Change Task | *Change-set* | |
| Workspace Update | Pull e resolve | |
| Task-Level Commit | Commit e push | |
| Task Branch | *Developer* | |

| | | |
|---|---|---|
| | *repository* | |
| Label | tag | |
| Third Party Codeline | Developer repository | Pode funcionar como um branch específico para as ferramentas de terceiros. |

### 3.6 AccuRev

AccuRev é uma ferramenta recente, mas que apresenta várias características comuns as ferramentas apresentadas nas seções anteriores. Por exemplo, como Perforce, AccuRev usa o termo "depot" para se referir a um repositório, além de se orgulhar da sua performance em relação as velocidade das operações cliente-servidor.

Uma das características diferentes de AccuRev é que ela não apenas versiona os seus dados, mas também todos os seus metadados (exemplo: versões de labels).

A idéia central para entender AccuRev é entender o simples, mas poderoso conceito de um stream. Esses streams podem ser usados como codelines, workspaces, e labels. Um AccuRev stream é um conjunto lógico de arquivos e versões de arquivos no depot. Um AccuRev workspace é simplesmente um lugar no qual são realizados trabalhos sobre um stream.Streams podem ser estáticos ou dinâmicos. Streams estáticos podem não ter seu conteúdo alterado, funcionam como um label ou baseline. Streams dinâmicas podem ter seu conteúdo mudado no workspace.

A avaliação dos padrões versus AccuRev é mostrada na tabela 7.

**Tabela 7. AccuRev versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | **Implementação** | **Comentário** |
|---|---|---|
| Repository | Depot | |

| | | |
|---|---|---|
| Development Workspace | Workspace | |
| Codeline | Stream | |
| Codeline Policy | **Não suporta** | |
| Private Versions | *Stream* | Implementação ineficiente. |
| Change Task | *Transaction* | A operação ocorre sobre todos os arquivos associados a Change Task ou a nenhum. |
| Workspace Update | Update | |
| Task-Level Commit | Promote | |
| Task Branch | *Workspace stream* | |
| Label | Real version | |
| Third Party Codeline | *Stream* | Pode-se criar um Stream específico para as ferramentas de terceiros. |

### 3.7 ClearCase LT

ClearCase está entre as mais populares e mais sofisticadas ferramentas de controle de versão do mercado. Assim como Perforce, ClearCase é bastante poderosa em relação ao suporte de desenvolvimento paralelo e branching. ClearCase tem uma versão básica (ClearCase LT), e uma versão *multisite* com suporte ao UCM (Unified Change Management), ClearCase - UCM.

ClearCase é umas das mais parametrizadas ferramenta do mercado, pois, por exemplo, possibilita customização de triggers para cada uma das suas operações. Uma características que somente ClearCase tem, é o controle de versões de diretórios tão bem quanto de arquivos.

A avaliação dos padrões versus ClearCase LT é mostrada na tabela 8.

**Tabela 8. ClearCase LT versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | **Implementação** | Comentário |
|---|---|---|
| Repository | Versioned Object Base (VOB) | |
| Development Workspace | View, config spec, e view profile | cleartool mkview, cleartool edcs,<br><br>make branch com view profile |
| Codeline | Project branch | Ferramentas / comandos relacionados: cleartool mkbrtype, mkbranch config spec rules, view profiles |
| Codeline Policy | *gatilhos* | |
| Private Versions | *Check-in / delivery(verificar)* | |
| Change Task | **Não suporta** | |
| Workspace Update | findmerge | cleartool findmerge, MergeManager |
| Task-Level Commit | findmerge | cleartool findmerge, MergeManager |
| Task Branch | *Private branch* | cleartool mkbrtype, make private branch |
| Label | Label | cleartool mklbtype, cleartool mklabel |
| Third Party Codeline | *Branch* | Podem ser implementados com Branch. |

### 3.8 ClearCase – UCM

O ClearCase UCM adiciona conceitos de gerência de configuração de alto nível de abstração, como baseline, atividade e stream. Um projeto no Clearcase UCM pode ser mapeado como uma codeline.Cada projeto pode ter um stream de integração (onde mudanças são mescladas dentro da codeline) e um ou mais streams de desenvolvimento. Cada stream pode está associado a várias atividades. E finalmente, cada atividade pode ser inicializada, finalizada e associada às versões dos arquivos alterados na execução da atividade.

A avaliação dos padrões versus ClearCase - UCM é mostrada na tabela 9.

**Tabela 9. ClearCase - UCM versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | **Implementação** | **Comentário** |
|---|---|---|
| Repository | Versioned Object Base (VOB) | |
| Development Workspace | Development stream | Comando associado: cleartool mkstream |
| Codeline | Project | Comando associado: cleartool mkproject |
| Codeline Policy | *Project policy settings* | |
| Private Versions | *Development stream* | |
| Change Task | *Activity* | Comando associado: cleartool mkactivity |
| Workspace Update | rebase | Comando associado: cleartool Rebase |

| | | |
|---|---|---|
| Task-Level Commit | Deliver | Comando associado: cleartool deliver |
| Task Branch | *Activity* | Uma atividade em um *development stream* |
| Label | Baseline | Comando associado: cleartool mkbl |
| Third Party Codeline | *Development stream* | *Development stream* específico para as ferramentas de terceiros. |

### 3.9 CM Synergy

CM Synergy era conhecida como “Continuus” e é uma poderosa ferramenta de GCS centrada em processo que usa tarefas, projetos, e diretórios com um workflow autamente configurável.

CM Synergy usa o termo "repository" e "database" indiferentemente. Um database suporta um ou mais projetos. Todos os arquivos para um particular sistema são geralmente associados a um projeto.

Workspaces em CM Synergy são chamados “work areas” ou “working projects”. Um working project é criado quando alguém executa um check-out do projeto inteiro para dentro de uma “work area”. A work área é um diretório físico onde os componentes serão armazenados localmente.

A unidade lógica de mudança é a tarefa. CM Synergy não precisa criar um branch para cada tarefa. Desenvolvedores simplesmente criam seu workspace e trabalham nas tarefas alocadas a eles naquele workspace. Desta forma, *project* em CM Synergy corresponde a uma codeline.

A avaliação dos padrões versus CM Synergy é mostrada na tabela 10.

**Tabela 10. CM Synergy versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | **Implementação** | **Comentário** |
|---|---|---|
| Repository | Project | Também chamado de *database* |

| | | |
|---|---|---|
| Development Workspace | Work area | |
| Codeline | Project object | |
| Codeline Policy | *Project template* | |
| Private Versions | *Project object* | Implementação ineficiente. |
| Change Task | *Task* | |
| Workspace Update | Update members | |
| Task-Level Commit | Complete task | Também chamado de *check-in task* |
| Task Branch | *Task* | |
| Label | Baseline | |
| Third Party Codeline | *Project object* | Específico para as ferramentas de terceiros. |

### 3.10 StarTeam

StarTeam é uma ferramenta orientada a processo, relativamente nova, mas já bem conhecida. Usa uma base de dados centralizada como repositório. Os clientes windows, UNIX, e Web acessam os diretórios/arquivos gerenciados pelo repositório. Um StarTeam view é um conjunto de pastas de trabalho e de arquivos de um projeto que representa ambos os workspaces e as codelines. Todas as atividades são realizadas sobre os arquivos dessa view.

Uma característica diferente de Starteam é a habilidade para combinar em uma mesma visão da ferramenta: arquivos, solicitações de mudança, requisitos, tarefas, e tópicos. Dessa forma, semelhante ao *Innovative Repository Manager*, ela permite rastrear e reproduzir mudanças sobre os arquivos, e relacionar

logicamente arquivos com as solicitações de mudança, com as tarefas, e com os tópicos. Os tópicos são as conversas entre os membros da equipe.

A avaliação dos padrões versus StarTeam é mostrada na tabela 11.

**Tabela 11. StarTeam versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | **Implementação** | **Comentário** |
|---|---|---|
| Repository | Repository | Working files/folders |
| Development Workspace | Working files/folders | |
| Codeline | Project view | Existe também uma branching view |
| Codeline Policy | *Project template* | |
| Private Versions | *Project view* | Implementação ineficiente. |
| Change Task | *Task* | |
| Workspace Update | Update Status | Usa view compare/merge para tratar conflitos. |
| Task-Level Commit | Merge | |
| Task Branch | *Branching view* | Existe também process rules, project view, e reference view |
| Label | Label | Existe também view label e revision label |
| Third Party Codeline | *Project view* | Pode-se criar uma visão específica para as ferramentas de terceiros. Implementação ineficiente. |

### 3.11 PVCS Dimensions

PVCS Dimensions é uma ferramenta bastante completa que automatiza não só o gerenciamento de versão, mas também o gerenciamento de mudança e o gerenciamento de processo.

Um workspace em PVCS corresponde a um *workset*. Os usuários podem compartilhar um *workset* ou podem ter permissão para criar um *workset* privado para suas mudanças.

As solicitações de mudança são publicadas (commit) num *workset* quando passam do estado de implementação para o estado de verificação. Codelines são criadas através de branchs. E os Labels correspondem as“baselines” que podem ser criadas para um produto.

A avaliação dos padrões versus PVCS Dimensions é mostrada na tabela 12.

**Tabela 12. PVCS Dimensions versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | **Implementação** | **Comentário** |
|---|---|---|
| Repository | Base database | |
| Development Workspace | Private workset | |
| Codeline | Named branch | Também pode ser um *workset* |
| Codeline Policy | *Control plan* | Nesse plano são configurados os processos/regras da equipe de desenvolvimento |
| Private Versions | *Branch ou Private workset* | Implementação ineficiente. |
| Change Task | *Work package* | |
| Workspace Update | Não suporta (verificar) | Updates the workset with the latest versions from the codeline (also see |

| | | |
|---|---|---|
| | | check-in) |
| Task-Level Commit | Promote action | Promove uma solicitação de mudança do estado de implementação para o estado de verificação |
| Task Branch | *Work package* | |
| Label | Baseline | |
| Third Party Codeline | *Branch* | Pode-se criar um branch específico para as ferramentas de terceiros. |

### 3.12 PVCS Version Manager

PVCS VM já existe a bastante tempo no mercado e é uma das mais usadas ferramentas de GCS. Ela é bem conhecida pela sua capacidade de definição de vários modelos de níveis de promoção. Que permite aos usuários cadastrar tarefas, *worksets*, projetos, e acompanhar sua evolução desde o desenvolvimento inicial das solicitações até o nível desejado de estabilidade / qualidade.

A avaliação dos padrões versus PVCS Version Manager é mostrada na tabela 13.

**Tabela 13. PVCS Version Manager versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | **Implementação** | **Comentário** |
|---|---|---|
| Repository | Project database | |
| Development Workspace | Workspace | |
| Codeline | Branch | |

| | | |
|---|---|---|
| Codeline Policy | *Promotion groups* | Existe também a opção “configuration options” onde as políticas podem ser configuradas por projeto |
| Private Versions | *Branch ou Project database* | |
| Change Task | **Não Suporta** | |
| Workspace Update | *Get* | Não trata bem conflitos causados pelo paralelismo |
| Task-Level Commit | Check-in | |
| Task Branch | **Não Suporta** | |
| Label | Version label | Existe também “baselining “ |
| Third Party Codeline | *Branch* | Pode-se criar um branch específico para as ferramentas de terceitos. |

### 3.13 MKSource Integrity

Source Integrity (SI) funciona de forma muito similar ao PVCS Version Manager. SI *projects* são conjuntos de arquivos agrupados por um mesmo escopo. *Projects* podem ser particionados em *subprojects*. Um workspace é chamado de *sandbox*. Codelines e task branches são chamados de *development paths*, e labels são chamados de *project checkpoints*.

Uma das características mais poderosas de SI são os pacotes de mudanças, que podem ser formados por uma simples solicitação de mudança ou por uma coleção das mesmas que podem ser adicionados/ removidos entre versões do projeto.

Um *sandbox* é criado e associado as últimas versões dos arquivos do projeto. Os arquivos são  bloqueados na operação c*heck-out*. O sandbox pode ser atualizado com o comando *resync* e todas a mudanças realizadas no sandbox podem ser agrupadas em um único pacote, de desejar. Eventualmente, um comando checkin publica as mudanças do *sandbox* para toda a equipe.

A avaliação dos padrões versus MKSource Integrity é mostrada na tabela 14.

**Tabela 14. MKSource Integrity versus Padrões de GCS**

| **Conceitos / Padrões de GCS** | **Implementação** | **Comentário** |
|---|---|---|
| Repository | Top-level project | |
| Development Workspace | Sandbox | |
| Codeline | Development path | |
| Codeline Policy | **Não suporta** | |
| Private Versions | *Development path* | Pode-se criar um Development path privado para cada desenvolvedor. Isso, no entanto, não é muito eficiente, pois pode demandar muitos merges posteriormente. |
| Change Task | *Change package* | |
| Workspace Update | Resync | |
| Task-Level | Check-in | |

| | | |
|---|---|---|
| Commit | | |
| Task Branch | *Change package* | |
| Label | Project checkpoint | |
| Third Party Codeline | *Development path* | *Path* específico para as ferramentas de terceiros. |

## 4. Considerações finais

O presente trabalho apresentou uma pesquisa realizada com 13 ferramentas de gerência de configuração de software onde se procurou avaliar a aderência de cada uma delas em relação aos padrões de GCS também apresentados no capítulo 2.

A pesquisa foi baseada nos livros citados na bibliografia e nas documentações/artigos sobre as ferramentas. Alguns livros foram adquiridos e os artigos foram coletados através da internet.

As seções sobre os padrões foram descritas de forma resumida, pois o foco principal era conhecer objetivamente cada um deles para poder entender a avaliação das ferramentas. Maiores informações sobre cada padrão apresentado poderão ser encontradas através da bibliografia apresentada no capítulo 5.

Sobre a seção das ferramentas também foram descritas de forma resumida cada uma delas, pois o objetivo maior era a montagem das tabelas relacionando os padrões com a forma de suporte/implementação de cada uma das ferramentas. Qualquer informação adicional sobre as mesmas poderá ser encontrada nos endereços eletrônicos listados na tabela 1.

Os padrões de GCS funcionam como uma *linguagem padrão* que pode ser tomada como parâmetro tanto para a aquisição de uma ferramenta de GCS, como

também para usá-las de forma mais eficiente e explorar todo o seu potencial ou o quanto for necessário. Dessa forma, as empresas precisam inicialmente entender as suas necessidades, selecionar os padrões que atendem as mesmas e consultar o presente trabalho para analisar quais ferramentas suportam os padrões selecionados. Depois podem acessar o endereço eletrônico das mesmas para obter mais detalhes técnicos e/ou comerciais.

## 5. Bibliografia


- Anne Mette Jonassen Hass - Configuration Management Principles and Practice
- Susan Dart, Spectrum of Functionality in Configuration Management Systems
- Ivica Crnkovic, Ulf Asklung, Anita Persson Dahlqvist - Implementing andIntegrating Product Data Management and Software Configuration Management
- Brown, William J., Hays W. McCormick, and Scott W. Thomas. 1999. *Antipatterns and Patterns in Software Configuration Management*. New York: Wiley.
- Dart, Susan. 1992. The Past, Present, and Future of Configuration Management: Software Engineering Institute.
- Grinter, Rebecca. 1995. Using a Configuration Management Tool to Coordinate Software Development. Paper read at ACM Conference on Organizational Computing Systems, August 13 - 16 1995, at Milpitas, CA.
- Leon, Alexis. 2000. *A Guide to Software Configuration Management*. Norwood, MA: Artech House.
- Mikkelsen, Tim, and Suzanne Pherigo. 1997. *Practical Software Configuration Management: The Latenight Developer's Handbook*. Upper Saddle River, NJ: Prentice Hall PTR.

- Vance, Stephen. *Advanced Scm Branching Strategies* 1998 [cited. Available from http://svance.solidspeed.net/steve/perforce/Branching_Strategies.html.
- White, Brian. 2000. *Software Configuration Management Strategies and Rational Clearcase: A Practical Introduction*, *Addison-Wesley Object Technology Series*. Boston, MA: Addison-Wesley.Whitgift, David. 1991.
- *Methods and Tools for Software Configuration Management*, *Wiley Series in Software Engineering Practice*. Chicester, England: Wiley.
- The Configuration Management Yellow Pages http://www.cmtoday.com/yp/configuration_management.html
- CM Crossroads - Online Community and Resource Center for CM Professionals http://www.cmcrossroads.com/
- UCM Central - Unified Configuration Management http://www.ucmcentral.com
- Berczuk, Steve. 1996b. Configuration Management Patterns. Paper read at Third Annual Conference on Pattern Languages of Programs, at Monticello, IL.
- Berczuk, Steve, and Brad Appleton. 2000. Getting Ready to Work: Patterns for a Developer's Workspace Paper read at Pattern Languages of Programs, at onticello, IL.