Introdução das premissas dos Controles de Versão

(grande parte) por Tableless

Parece ser óbvio, mas muitos desenvolvedores desdenham de ter controle total sobre o código gerado no desenvolvimento. Muitos, por trabalharem sozinhos, entendem que não precisam manter certo nível de organização do seu código exatamente porque talvez, somente eles, o verão.

Quem nunca ficou horas tentando debugar um código que você mesmo fez, mas não lembrava o que uma determinada função fazia… Esse código pode ter sido feito por você de madrugada, quando você estava pingando de sono, ou pior, bêbado… Vai saber…

Controlar seu código fonte deve ser uma premissa. Um princípio. Se você acha que o undo, do seu editor predileto, salva sua vida, imagina ter um undo do seu projeto inteiro. Imagine você ter a história de edição de cada um dos arquivos do seu projeto.

Os princípios abaixo servem para qualquer tipo de controle de versão que conhecemos hoje: GIT, SVN, Mercurial, etc.

Branchs e trunks

A última revisão, a mais atual, normalmente é chamada de HEAD. Existem momentos onde teremos vários HEADs porque o projeto tem vários BRANCHES.

Um branch é uma cópia do projeto. Cada branch pode ser editado e evoluído de forma independente.

Imagine que exista a versão de produção, que é aquela que está no ar. É a versão que o usuário está utilizando. É importante que nós não modifiquemos esta versão porque alguém pode commitar algo errado, quebrar tudo e o usuário reclamar. Por isso nós precisamos de outro ambiente, e é aí que criamos um branch de desenvolvimento, onde possamos fazer o que quisermos ali, sem afetar o branch principal.

É muito útil criarmos novos trunks (tronco) quando precisarmos realizar modificações drásticas de novas features, resolver bugs ou modificar layout.

Exemplo: imagine que você esteja trabalhando em uma nova feature, em um branch específico. O cliente liga e diz que encontrou um bug, originado de uma modificação feita na semana passada. Você sai do branch da nova feature e cria um novo. Este novo tem a cópia do código do branch de produção, que é o que o cliente está vendo. Você resolve o bug neste novo branch e então você move as modificações deste para o de produção. Isso tudo sem ter que subir as modificações incompletas da nova feature que você estava trabalhando anteriormente.

Versionamento

Versionamento é uma das características mais básicas do controle de código. Quando comecei a desenvolver, normalmente eu trabalhava guardando pastas com nomes do tipo nomeDoProjeto-v1, nomeDoProjeto-v2, etc… Estas pastas guardavam apenas as versões com grandes modificações de projeto… algo como mudança geral de layout, uma feature importante ou algo do tipo. O problema são as pequenas edições.

Quando juntamos as pequenas modificações, temos uma grande modificação. É muito provável que se você perder essa versão, você não se lembre das pequenas modificações feitas, e isso é um problema grande.

Toda vez que o desenvolvedor termina uma determinada tarefa, ele geral um commit, que por sua vez gera um ponto de referência, uma versão daqueles arquivos modificados.

Cada commit gera uma versão do seu código. A graça é que se você tem versões do código, você praticamente tem um undo gigante do seu projeto.

Se depois de uma determinada modificação seu projeto começou a dar problema, você consegue entender com a história do seu versionamento – o famoso log – a partir de onde exatamente o problema foi iniciado.

Logs

Quando cada revisão é commitada, o desenvolvedor pode adicionar uma mensagem explicando o que ele fez naquela tarefa. É importante que essa mensagem seja clara e rica em detalhes, porém, apenas o suficiente para que você mesmo ou outras pessoas envolvidas entendam o que aquela submissão significa.

Diffs

Diffs são sensacionais. Imagine que você queira comparar duas versões de um mesmo arquivo para descobrir quais as linhas modificadas. Fazer um diff entre as versões do arquivo te possibilita encontrar exatamente onde está o erro, e o melhor, quando juntamos com o Log, podemos saber exatamente o porque de a pessoa ter incluído aquela linha ou aquele código, e ainda mais, quem fez aquela alteração.

Rollback

Mantendo uma história de commits, você tem pontos de volta na história do arquivo, possibilitando a facilidade de voltarmos para antes de alguma modificação. Você pode voltar a partir de um commit, por exemplo. Se o sistema ou o arquivo passou a dar problema depois de uma determinada revisão, é simples você voltar o arquivo para a versão daquele determinado commit.

Edição multiusuário e merge

Lembro-me de equipes inteiras perdendo arquivos e partes de código porque dois membros abriam o mesmo arquivo para editar. Obviamente nunca dava certo. O que faziam era criar mecanismos ineficientes para avisar os outros que determinado arquivo estava sendo usado. O trabalho em equipe remotamente nunca era possível. Com um controle de versão isso muda. Qualquer um pode editar qualquer arquivo a qualquer hora.

O sistema de controle de versão compara as modificações feitas no arquivo pelos dois (ou mais) desenvolvedores e junta os códigos automaticamente.

E quando os desenvolvedores modificam a mesma linha de código? Aí o controle de versão gera um conflito, esse conflito deve ser resolvido pelo desenvolvedor que executou o segundo commit. Funciona assim:

1. Dois desenvolvedores abrem o mesmo arquivo para editar;
2. Quando os desenvolvedores terminam, eles geram um commit. Se os desenvolvedores fizeram edições em partes diferentes do arquivo, por exemplo, um no topo do arquivo e o outro no final, o sistema junta os códigos fazendo um merge, e criando um arquivo atualizado com as duas revisões.
3. Se sistema percebe que os desenvolvedores fizeram as modificações na mesma linha, o sistema gera um conflito.
4. O conflito é resolvido pelo segundo desenvolvedor que commitar o código, ou seja, o cara commitou a modificação dele primeiro que você. Quando você commitar a sua modificação, o controle de versão avisará que há um conflito em uma parte do seu código. O conflito é resolvido mostrando algo mais ou menos assim:

```
p {
<<<<<<< HEAD:estilo.css
color: red;
=======
color: blue;
>>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:estilo.css
}
```

A primeira linha mostra a sua linha. A segunda linha mostra a modificação do outro dev. Aquele hash maluco no final é a identificação do commit do outro dev.

Bom, nesses casos você precisa ver qual código é o correto, e isso, às vezes, inclui perguntar para o outro dev o que ele fez.