Encontrando seu fluxo de trabalho

Fluxos de trabalho e dor E bem-vindo ao módulo final do Mastering Git. Até agora neste treinamento, fizemos uma suposição implícita. Assumimos que você estava trabalhando sozinho em seu próprio repositório Git e nunca estava compartilhando coisas com outros repositórios. Às vezes, prestamos atenção ao mundo exterior, como onde dissemos regra de ouro, não mudamos as coisas que você compartilhou com outras pessoas, mas principalmente ignoramos outras pessoas e outros repositórios como se eles não existissem. Mas é claro, o ponto principal do Git para a maioria das pessoas é que você o usa para compartilhar informações com outros desenvolvedores. E você tem muitas opções para compartilhar as informações. De fato, o conjunto de ferramentas Git é tão poderoso e flexível que você tem muitas opções às vezes, e isso pode causar algumas dores de cabeça quando várias pessoas se reúnem para trabalhar em um projeto compartilhado. Para evitar essas dores de cabeça, você e sua equipe precisam tomar algumas decisões sobre como trabalhar juntos e três coisas em particular. Primeiro, você deve decidir que tipo de modelo de distribuição adotar. Quantos repositórios você possui e como eles interagem? Você talvez tenha um repositório compartilhado que seja visível para todos os desenvolvedores? Talvez muitos repositórios compartilhados. Todos os desenvolvedores podem enviar suas confirmações para o repositório compartilhado ou alguns desenvolvedores têm acesso somente leitura e assim por diante? Além disso, você precisa tomar decisões em alguns ramos. Para quais ramos você tem no seu projeto e para que os usa? Qual ramo você usa para integrar seu trabalho ao trabalho de outros desenvolvedores? De qual ramo você usa para compactar um release? E similar. E, finalmente, você precisa definir restrições mais gerais para todos que contribuem para o projeto. Essas são todas as regras adicionais que realmente não pertencem ao modelo de distribuição ou ao modelo de ramificação, como por exemplo, quando você recebe um monte de novas confirmações de um controle remoto, você deve mesclar essas confirmações no seu próprio repositório ou deve você os rebase? Você pode enviar seu código para um controle remoto se as tarefas forem interrompidas ou deve garantir que o código esteja estável e funcionando antes de enviar? E tantas outras decisões possíveis. Portanto, modelo de distribuição, modelo de ramificação e restrição. Essas três coisas juntas definem o que você pode chamar de fluxo de trabalho distribuído, e é sobre isso que trata este módulo de trabalho, fluxos de trabalho distribuídos. Como você usa o Git na prática em um projeto do mundo real? Este módulo é um pouco diferente do restante do treinamento. Os fluxos de trabalho são mais um tópico social do que um tópico técnico; portanto, você não terá muitas informações técnicas aqui. O que você obterá é um vocabulário para discutir fluxos de trabalho com outras pessoas, como um conjunto de padrões. De certa forma, você pode pensar neste módulo como padrões de fluxo de trabalho do Git. Para entender este módulo, você precisa saber algumas coisas sobre o Git distribuído. Não muito, apenas o básico. Em particular, espero que você saiba o que é controle remoto e esteja familiarizado com o conceito de enviar para um controle remoto e puxar de um controle remoto. Se você já trabalhou em um projeto Git distribuído, mesmo em seu próprio projeto no GitHub, já possui esse conhecimento. Se você não tiver certeza sobre esses tópicos, poderá atualizá-los assistindo ao último módulo do treinamento Como o Git Funciona. Vamos começar a falar sobre fluxos de trabalho.

Modelos de distribuição A primeira coisa que você deve decidir ao configurar um fluxo de trabalho Git, a decisão que influencia a maioria das outras decisões na linha, é o seu modelo de distribuição. O modelo de distribuição mais simples é provavelmente o modelo ponto a ponto. Imagine que você é um desenvolvedor em uma equipe de três. Todos esses repositórios contêm o mesmo projeto, é claro, foram originalmente clonados um do outro, mas divergem à medida que os desenvolvedores se comprometem com suas próprias cópias do repositório. O truque aqui é que cada desenvolvedor pode ver os repositórios de outros desenvolvedores como controles remotos. Portanto, se você é Nick neste diagrama e sabe que Ralph tem alguns novos commits que deseja, basta sair e inserir essas alterações no seu repositório, e se você sabe que há coisas novas no repositório de Jane, então você também pode fazer essas mudanças e os outros desenvolvedores podem fazer o mesmo, para que as mudanças se espalhem assim. Agora, é verdade que o modelo ponto a ponto é simples, mas não é necessariamente fácil, especificamente porque nenhum repo é mais importante que os outros. Essa é a definição de ponto a ponto, certo? E isso pode se tornar um desafio quando você deseja fazer algo tão simples quanto liberar o projeto, por exemplo. Então você precisa decidir de qual repositório liberar, e cada repositório pode conter coisas ligeiramente diferentes, por isso pode ser difícil decidir qual é o certo, por assim dizer, e se isso pode ser difícil para três desenvolvedores, pode ser muito mais difícil com quatro ou cinco. Por esse motivo, a menos que seu projeto seja literalmente duas pessoas em uma sala, você provavelmente desejará tomar algumas decisões adicionais antecipadamente. Em particular, você pode decidir que um dos acordos é especial. É abençoado, como eles dizem. Esse repositório abençoado é acessível a todos no projeto, tanto para puxar quanto para empurrar, para que você possa confirmar dados em seu próprio repositório local, mas também possui um controle remoto que está apontando para o repositório abençoado e pode enviar dados para esse controle remoto . A maioria dos desenvolvedores chama essa origem remota, por convenção, porque esse é o nome padrão que o Git atribui ao controle remoto do qual você clona o projeto. O repositório abençoado geralmente é um repositório simples, o que, em termos do Git, significa que é apenas o repositório sem uma área de trabalho ou um índice. Ninguém está trabalhando diretamente nessa máquina. É usado apenas para compartilhar dados e, talvez, para hospedar a máquina de compilação, o sistema que executa os testes de unidade, pacotes, versões e assim por diante. Então, deixe-me colocar um robô aqui para significar que este é o repositório de máquinas de construção compartilhada, não um repositório em que um humano está trabalhando dia após dia. Agora, neste modelo, você não acessa os repositórios de seus colegas de equipe e vê os dados deles, como no modelo ponto a ponto. Em vez disso, você se preocupa apenas com os dados no repositório abençoado. Esse é o estado oficial do projeto, por assim dizer. Portanto, todo mundo extrai seus dados do repositório abençoado, e o desenvolvedor que possui novos commit, como Ralph neste caso, deve enviar esses commit para o repositório abençoado, para que o restante da equipe possa obtê-los. Tudo é centralizado. Na verdade, você pode simplesmente chamar o repo abençoado de servidor, e o modelo em si é frequentemente chamado de modelo centralizado. Essencialmente, o modelo centralizado é o mesmo que as pessoas usam com sistemas de versão não distribuídos, como o subversion ou o Team Formation Server. Atualmente, a maioria das empresas substitui esses sistemas de versão. Eles mudam para o Git por vários motivos, mas não necessariamente mudam o modelo de distribuição. Muitos deles ainda usam um modelo centralizado por pares. Há ainda outro modelo que é uma reviravolta no modelo centralizado, em que você ainda possui repositórios de desenvolvedores e ainda possui um repositório abençoado central, mas a maioria dos desenvolvedores não pode gravar diretamente no repositório abençoado. Eles não podem empurrá-lo, apenas podem puxá-lo. Somente uma pessoa, ou algumas pessoas no projeto, têm esse poder de levar as coisas ao repo abençoado. Eles geralmente são chamados de mantenedores. Neste exemplo aqui, apenas Nick pode enviar por push para o repo abençoado, e os outros desenvolvedores, aqueles que não têm acesso por push ao repo abençoado, às vezes são chamados de colaboradores. Fiz as setas cinza para significar que são conexões somente leitura ou somente puxar. Portanto, a equipe ainda usa o repositório abençoado como um hub de comunicação e, quando um commit está no repositório abençoado, é oficial, e todo mundo o fará eventualmente. Agora, digamos que Jane tem novos compromissos para contribuir, mas ela não é a mantenedora. Ela não tem o poder de avançar para o repositório central, então como ela pode contribuir com esses compromissos para o projeto? O truque aqui é que o mantenedor também pode ver os colaboradores para reposicionar, para que Jane possa dizer oi Nick, puxe minhas alterações, veja se você gosta delas e fique à vontade para empurrá-las para o repo abençoado. Talvez Jane possa simplesmente ir até a mesa de Nick e contar a ele sobre as mudanças, ou talvez Jane e Nick nem compartilhem o mesmo escritório, então ela deve avisá-lo de alguma outra maneira, por exemplo, enviando uma mensagem dizendo: olha, eu tenho novos commits no meu repositório que você pode querer usar. Essa mensagem pode ser um email, por exemplo, ou pode ser gerenciada por algum tipo de serviço, como o GitHub, e isso geralmente é chamado de solicitação de recebimento. Agora, Nick recebeu uma solicitação de recebimento, para que ele saiba das mudanças de Jane. Se ele estiver de acordo com essas mudanças, ele poderá colocá-las em seu próprio repositório, resolver qualquer conflito de mesclagem que ele possa ter e enviá-las para o repositório abençoado. E agora as mudanças de Jane foram feitas parte oficial do o projeto, e outros colaboradores como Ralph também podem retirá-los. A solicitação de recebimento é a mecânica definidora desse modelo, para que você possa chamá-lo de modelo de solicitação de recebimento. O Git possui alguns recursos que facilitam a preparação de uma solicitação pull. Na verdade, existe um comando pull de solicitação do git, mas na verdade o envio da solicitação de pull é um recurso do Git. Você precisa usar correio antiquado ou algum outro meio de fazer isso. E essa é uma das razões pelas quais serviços como o GitHub são tão populares, porque automatizam o envio de solicitações pull, tanto que a solicitação pull se tornou o recurso definidor do GitHub, e o modelo pull request se tornou o mais popular. modelo para desenvolvimento de código aberto. A vantagem mais significativa do modelo de solicitação de recebimento é que ele ajuda a gerenciar a confiança. Você pode obter contribuições de fontes confiáveis, como Nick, neste caso, e também de fontes menos confiáveis, como Ralph e Jane, e isso é necessário em projetos de código aberto, é claro, porque, em código aberto, qualquer pessoa na Internet pode ser um colaborador e você provavelmente não confia na Internet inteira para enviar diretamente para o repositório do seu projeto, para que possa ter alguns mantenedores confiáveis ​​e qualquer número de colaboradores, mas esse conceito de gerenciamento de confiança também é útil para projetos de código fechado. Sempre que você tiver um projeto em que não deseja conceder acesso push ao repositório principal para toda a equipe de desenvolvimento. Nesse caso, você pode nomear um mantenedor interno, talvez chamá-lo de gerente de integração, e os desenvolvedores devem solicitar que essa pessoa faça suas alterações. Uma última coisa sobre o modelo de solicitação de recebimento antes de prosseguirmos. Se você observar os projetos no GitHub, eles usam esse modelo em uma variação um pouco mais complexa. A complexidade adicional aqui é que, no GitHub e em outros serviços como o GitHub, todos esses repositórios estão realmente na nuvem; portanto, os desenvolvedores não estão trabalhando diretamente nesses repositórios. Em vez disso, cada desenvolvedor tem dois repositórios, o repositório particular em seu próprio computador e o repositório público na nuvem. Portanto, o conceito é praticamente o mesmo que o modelo básico de solicitação de puxar, mas a mecânica de empurrar e puxar e coisas do gênero estão um pouco mais envolvidas. Se você quiser uma explicação detalhada desse tipo específico de modelo de solicitação de recebimento, poderá encontrar um no último módulo do treinamento Como o Git Funciona, para que eu não me repita aqui. Em vez disso, vou falar sobre um último modelo de distribuição. Este é como o modelo de solicitação de recebimento, mas leva as coisas um passo adiante. No modelo de solicitação de recebimento, você tem um repositório abençoado para todo o projeto com um ou mais mantenedores que podem acessá-lo. Aqui eu mostro no repo abençoado e que Jane é a mantenedora. Nesse outro modelo, no entanto, o projeto é dividido em subprojetos, e cada subprojeto tem seu próprio repositório abençoado e seu próprio mantenedor, ou mantenedores, e existem outros colaboradores em geral. Devo admitir que estou reutilizando os gráficos para os colaboradores um pouco aqui, mas tudo bem. Você sabe o que eles dizem sobre descolados, todos parecendo iguais. Também mudei um pouco o estilo deste diagrama. Não desenhei nenhuma flecha para evitar torná-la muito confusa. Em vez disso, lembre-se de que, neste modelo, qualquer pessoa pode extrair dados de praticamente qualquer lugar, mas apenas os mantenedores podem enviar dados para os repositórios abençoados. Então, aqui temos três níveis. Você pode ter ainda mais se quiser ter vários níveis de subprojetos, e há solicitações de recebimento, é claro. Contribuintes regulares geralmente enviam PRs para subprojetos, e os mantenedores do subprojeto enviam PRs para o mantenedor do projeto principal, e todo mundo puxa dados para seus próprios repositórios, para que os dados se espalhem para cima em resposta às solicitações de recebimento, e para baixo à medida que as pessoas o puxam. dos níveis superiores. Às vezes, esse modelo é usado em projetos muito grandes que são grandes demais para uma única equipe de mantenedores. O exemplo clássico é o kernel do Linux. No Linux, os mantenedores do subprojeto são chamados de tenentes, e o mantenedor global é chamado de ditador benevolente; portanto, você pode chamar esse modelo de modelo de ditador e tenentes. Também é popular em grandes empresas que têm grandes projetos ou, às vezes, realmente gostam de hierarquias. Então, para recapitular, vimos quatro modelos de distribuição, o modelo ponto a ponto, sem repo abençoado, sem controle centralizado. É puramente distribuído. O modelo centralizado é como o gerenciamento de configuração tradicional e não distribuído. Você tem um repositório central abençoado, um servidor e todos enviam para esse servidor. O modelo de solicitação de recebimento ainda possui um repositório centralizado, mas a maioria das pessoas só pode obter dele. Alguém que tem o direito de enviar por push para o repositório central deve enviar os motores de outros repositórios em resposta a algum tipo de solicitação de recebimento. E o modelo mais complicado é o ditador e os tenentes, onde você tem vários subprojetos, cada um é como um projeto de modelo de solicitação pull e, em seguida, um nível mais alto, ou talvez até vários níveis mais altos de integração, porque o subprojeto também é baseado em solicitações de recebimento. Um ponto importante para fechar. Lembre-se de que esses são padrões. Eles não são receitas de como estruturar seu próprio projeto. Portanto, você pode querer ter uma abordagem mista em seu próprio projeto. Muitos projetos fazem isso. Talvez a maioria dos desenvolvedores possa precisar de um projeto para trabalhar em um modelo centralizado, exceto a equipe de desenvolvimento offshore que segue o modelo de solicitação de recebimento e envia solicitações de recebimento aos desenvolvedores internos. Ou talvez você trabalhe centralizado, exceto que às vezes os desenvolvedores estão trabalhando juntos em um recurso e, em seguida, optam por sincronizar seus repositórios como se estivessem trabalhando em um modelo ponto a ponto. Isso é bom. Você não precisa seguir um modelo religiosamente. Use o que funciona para você. Esses nomes que eu dei a você são apenas rótulos. Eles são úteis para conceituar e discutir suas opções; portanto, por exemplo, você pode facilmente dizer que estamos usando um modelo centralizado aqui, e seu colega de equipe pode responder, não, vamos usar um modelo de solicitação de recebimento. Esse é o objetivo dos padrões.

Modelos de ramificação Logo após o modelo de distribuição, o segundo elemento importante para garantir o fluxo de trabalho do Git é algum tipo de política para gerenciar filiais. Todo projeto tem essa política, seja explícita ou não, então vamos ver alguns padrões comuns para ramificações. Primeiro, deixe-me fazer uma distinção entre ramos estáveis ​​e instáveis. Uma ramificação é estável quando a ponta da ramificação sempre contém uma versão funcional do projeto. Ou seja, os testes são verdes, não há bugs conhecidos e assim por diante. Aqui, você pode empacotar o que estiver em um ramo estável e reutilizá-lo. Usarei essa marca de seleção verde para significar a versão de trabalho do projeto. Portanto, mesmo que alguém adicione novas confirmações à ramificação, por exemplo, pressionando-as, a ponta da ramificação ainda contém um sistema em funcionamento. Em um ramo instável, você não necessariamente tem isso. Quando alguém envia mais uma confirmação ao ramo, a ponta do ramo pode estar funcionando ou pode estar quebrada. Não há garantia. Agora que temos uma definição de ramificação estável e ramificação instável, voltemos às ramificações típicas de um projeto. Quase todo projeto tem um ramo principal que você usa para juntar tudo. Geralmente, esse é o ramo que as pessoas consideram o ramo mais importante do sistema. As pessoas podem trabalhar em outros ramos, mas os outros ramos tendem a se ramificar nesse ramo principal. Eles tendem a permanecer razoavelmente alinhados com o ramo principal e acabam voltando ao ramo principal. Estou usando cores diferentes para os commits aqui para fazer com que os segmentos dos ramos se estendam um do outro. As cores não têm nenhum significado específico. Esse ramo tão importante, que eu desenhei em vermelho na foto, geralmente é chamado de mestre, mas seja como for. Você pode chamá-lo de ramo de integração porque é o lugar onde as coisas se reúnem e geralmente o lugar onde você resolve os conflitos que você pode ter quando as coisas se reúnem. As pessoas também o chamam de ramo principal, ou a linha principal, às vezes, ou ramo de desenvolvimento, ou apenas o ramo principal, mas temos que escolher um nome, então chamarei de ramo de integração. Agora, o ramo de integração é estável ou instável? Bem, isso depende do projeto. Na maioria dos casos, quanto mais estável for o ramo de integração, melhor. Afinal, este é o ramo central do projeto. Principalmente todo mundo está trabalhando nisso, e ninguém gosta de trabalhar em uma base de código instável. Por outro lado, é difícil manter uma ramificação estável quando você está constantemente integrando coisas novas sobre ela; portanto, na prática, a maioria dos projetos parece uma ramificação de integração praticamente estável. Na verdade, é para isso que servem as máquinas de construção. O principal trabalho de uma máquina de compilação na Jenkins, ou algum outro tipo de sistema de compilação automatizado, é verificar o que estiver no ramo de integração, provavelmente executar os testes e informar se a compilação atual está funcionando ou quebrada . Outra pergunta importante na maioria dos projetos é de qual ramo você libera seu software? Em algum momento, você precisa implantar o software em um servidor da Web, ou talvez empacotá-lo e distribuí-lo em uma loja de aplicativos ou qualquer que seja o seu método de distribuição. Alguns projetos fazem isso a partir do ramo de integração, colocando principalmente uma marca em confirmações específicas para marcar os pontos de onde são liberados. Outros projetos preferem ter uma ramificação separada para liberações, uma ramificação de liberação. Qual o sentido de ter uma ramificação separada para lançamentos? Bem, existem algumas vantagens, mas a mais óbvia é que você pode manter o código nesse ramo mais estável do que no ramo de integração. Por exemplo, você pode mesclar a ramificação de integração na ramificação de liberação somente depois de verificar se é estável. Essencialmente, um ramo de versão separado fornece um buffer para manter as alterações liberáveis ​​separadas das alterações ainda não totalmente liberáveis. Eu falei sobre um ramo de lançamento, mas em alguns projetos você precisa manter vários lançamentos ao mesmo tempo. e, nesse caso, você pode ter várias ramificações de liberação. Talvez eles se ramifiquem do ramo de integração no momento em que o lançamento acontece e depois prossigam. Se você precisar adicionar documentação específica ou corrigir um bug ou a versão 1.1, mas não a versão 1.2, bem, você terá um ramo de versão específico para trabalhar. Há outro tipo de filial que é um grampo de muitos e muitos projetos. Digamos que você tenha dois desenvolvedores trabalhando em dois recursos diferentes. Em alguns projetos, ambos iriam diretamente para o ramo de integração. Estou usando duas cores diferentes para os dois recursos aqui. Com essa maneira de trabalhar, você pode ter uma integração muito frequente, que eu gosto em geral, mas às vezes pode ser difícil fazer o certo e, além disso, sua história fica difícil de entender porque os commits pertencentes a diferentes recursos podem ser todos interligado, como neste caso, e com essa abordagem, você também precisa viver com recursos semi-desenvolvidos no ramo de integração na maior parte do tempo. Por exemplo, agora, talvez o recurso verde esteja pronto, mas o recurso azul está ainda em andamento, você terá metade do recurso azul no ramo de integração. Uma alternativa possível é criar uma nova ramificação para cada recurso. Assim, por exemplo, um ramo para o recurso A e um para o recurso B. E quando as pessoas começam a avançar para esses ramos, elas divergem e progridem em paralelo até eventualmente voltarem para o ramo de integração. Nesse caso, estamos integrando ao mesclar, mas você também pode refazer o processo, se desejar. E depois disso, você pode excluir esses ramos, se desejar, ou deixá-los lá para referência futura. Esses ramos são chamados ramos de recursos. Algumas pessoas preferem o nome tópico ramos, o que também é bom. Usarei ramificações de recursos aqui apenas porque parece ser um pouco mais comum. Muitos projetos usam ramificações de recursos. Já vimos alguns estilos de ramificações, e você pode pensar que, depois de ter uma ramificação de integração, talvez uma ou mais ramificações de lançamento e várias ramificações de recursos, você tem tudo o que precisa. Mas há um problema adicional que pode resultar em mais ramificações para alguns projetos. Imagine que você tenha uma situação como essa com alguns ramos. Deixe-me dar nomes a eles, para que seja mais fácil referenciá-los. Não importa quais ramificações elas são exatamente, o que importa é que essas ramificações estão divergindo há algum tempo, mas a ramificação 1 tem uma confirmação, essa confirmação vermelha aqui, que você também deseja ter na ramificação 2. Você não não quero nenhum dos outros commit das ramificações 1 e 2, apenas esta confirmada. Então, como você faz isso? Como você copia um único commit de um ramo para o outro? Bem, o Git tem um comando especial que é exatamente isso, e é chamado de escolha de cereja. Você pode escolher uma única confirmação, ou algumas confirmações específicas, de uma ramificação e copiá-las em cima de outra ramificação. Problema resolvido, exceto por um detalhe. Uma escolha de cereja é como uma pequena recuperação. É a reformulação de um commit específico, e os subprojetos não gostam de reformulações. Eles querem usar a mesclagem em todos os lugares. Portanto, as escolhas de cereja não são uma opção para projetos baseados em mesclagem. Como você ainda pode ter o mesmo commit em dois ramos separados sem escolher a cereja? Bem, há outra solução para o problema. Quando você deseja que o mesmo commit seja compartilhado por dois ramos separados, basta ter um terceiro ramo e colocar esse commit compartilhado no terceiro ramo. E então você mescla o terceiro ramo no primeiro e no segundo ramo, assim. E é isso. Agora você tem um lugar para colocar confirmações compartilhadas e acaba de mesclar, sem rebotes. Esta é realmente uma situação comum, especialmente quando você tem uma correção de bug. Deixe-me mudar os nomes desses ramos. Você tem um ramo de lançamento e um ramo de integração, e eles estão divergindo. Você acabou de encontrar um bug desagradável em sua versão mais recente e deseja corrigi-lo imediatamente e preparar outra versão. Mas você também deseja que a mesma correção de bug esteja no ramo de integração, para que você também tenha o bug corrigido na próxima versão. Bem, basta colocar a correção de bug em outro ramo chamado, digamos, hotfix, e mesclar o hotfix no ramo de lançamento e de integração. Agora, essa correção está no histórico dos dois ramos. Melhor dos dois mundos. Você manteve as ramificações separadas e ainda compartilhou dados entre elas. Também existem outros casos de uso para o ramo de acoplamento como este, mas as correções de bugs são comuns, portanto, não se surpreenda se você vir projetos que possuem os chamados ramos de hotfix para esse tipo de coisa. Então vamos ver. Falamos sobre a diferença entre ramificações estáveis ​​e instáveis ​​e, em seguida, sobre diferentes padrões de ramificação, integração, versão, recurso e ramificações de hotfix. Pode haver mais, você provavelmente criaria alguns ramos especializados. Mas esses são comuns o suficiente para merecerem ser chamados de padrões. Você os vê surgindo várias vezes em vários projetos diferentes. E é isso sobre modelos de ramificação. Vejamos o terceiro e o último componente de um fluxo de trabalho distribuído.

Restrições O terceiro e o último elemento de um fluxo de trabalho distribuído, após o modelo de distribuição e o modelo de ramificação, é um conjunto de restrições. Estou usando restrições como um nome completo para todas as coisas que claramente não pertencem ao modelo de distribuição ou ao modelo de ramificação. Além disso, comportamento, se desejar. Todo projeto em que trabalhei tinha suas próprias regras, às vezes muito específicas, e às vezes surpreendentes, então não posso mostrar nenhum padrão específico aqui, porque existem muitas restrições possíveis, mas posso apresentar alguns exemplos de restrições que eu já vi em projetos da vida real. Eu já mencionei um dos exemplos mais comuns, a escolha entre mesclagens e rebotes. Em alguns projetos, as pessoas preferem sempre mesclar, portanto, se você acabou de trabalhar em uma ramificação de recursos, por exemplo, você as mescla na ramificação de integração. Por outro lado, alguns projetos refazerão toda a ramificação de recursos em cima da ramificação de integração, e as duas políticas têm trocas diferentes, que geralmente são sutis e, bem, estão fora de tópico aqui, na verdade. Eu falei sobre essas compensações no treinamento Como o Git Funciona. Mais uma vez, não quero me repetir. Meu argumento aqui é que depende de você usar mesclagem ou rebase, mas você provavelmente deve tentar ser consistente com isso, para que provavelmente não queira metade da equipe usando mesclagem e a outra metade usando rebase. Isso seria confuso, portanto, essa geralmente é uma decisão para todo o projeto. Você provavelmente quer que seja uma construção oficial. Outra questão importante para alguns projetos é quem pode fazer o que com quais filiais? Talvez seja esperado que alguns desenvolvedores se comprometam apenas com alguns ramos e não com outros, ou talvez seu projeto esteja usando tags no ramo de lançamento para marcar um novo lançamento, para que apenas o responsável pelas liberações possa marcar o ramo de lançamento. Outros desenvolvedores devem evitar fazer isso, mesmo que tecnicamente eles possam fazer isso. Aqui está outro exemplo. Este não é necessariamente comum, mas é interessante. Eu estava trabalhando em um projeto com muitos desenvolvedores fazendo integração contínua, e a build-machine foi executada nos testes de unidade no ramo de integração. De vez em quando, os testes podiam ser interrompidos, em vermelho; portanto, tínhamos uma regra em vigor que dizia que, se você perceber que a compilação é vermelha, pare de empurrar para o servidor. Aguarde até a pessoa que interrompeu a construção perceber que há um problema para que ela possa consertar a construção e torná-la verde novamente. Caso contrário, se você continuar pressionando, tornará o trabalho dessa pessoa realmente difícil. Você mudará o código sob seus pés e acionará mais compilações. Então, essa foi a restrição. Não empurre quando a construção estiver vermelha. Espere que ele fique verde novamente. A parte interessante é que às vezes as pessoas se esqueciam de verificar a máquina de construção antes de empurrar, então elas violavam essa restrição, mesmo que não quisessem. Então a equipe fez algo inteligente. Eles usaram o recurso Git chamado ganchos. É basicamente uma maneira de executar um script sempre que um evento específico acontece, então eles tinham esse gancho no servidor que iniciou esse script sempre que alguém iniciou um push no servidor, e o script verificou a máquina de compilação e, se a compilação era vermelha, o script lhe deu um aviso e a oportunidade de abortar o envio. Eu acho que é um bom exemplo de como ajudá-lo a respeitar a restrição com alguma automação simples. Aqui está outra decisão que algumas equipes optam por regular. Que tipo de histórico ou refatoração você deve fazer nas ramificações de recursos antes de mesclá-las na ramificação de integração? Alguns projetos gostam de compactar o recurso inteiro em um único commit, por exemplo, enquanto outros preferem manter um commit pequeno e granular no histórico da linha principal, e assim por diante. Não quero continuar reclamando aqui, porque isso pode sugerir que essas são todas as áreas em que você deve tomar decisões. Na verdade, em seu projeto, você pode não se importar com esses assuntos e, em vez disso, pode se preocupar com outras coisas que eu nem pensaria. Eu só queria criar um monte de exemplos. O que estou dizendo é que os fluxos de trabalho mais distribuídos têm algumas restrições e existem muitas restrições diversas; portanto, é bom que você torne explícita a restrição do projeto onde descreve seu fluxo de trabalho. Portanto, agora discutimos os elementos que constituem um fluxo de trabalho Git, um modelo de distribuição, um modelo de ramificação e um conjunto de restrições. Como combinamos esses elementos em um fluxo de trabalho? Para responder a isso, vamos primeiro examinar um exemplo popular da vida real de um fluxo de trabalho distribuído.

Precisamos conversar sobre o GitFlow Se você procurar fluxos de trabalho do Git no Google, é provável que você encontre esta postagem no blog entre os primeiros resultados. É um post de Vincent Driessen, um desenvolvedor holandês. Ele o publicou em 2010. Naquela época, o Git começava a se espalhar por toda a comunidade de software e muitas pessoas ficaram impressionadas com todas as possibilidades que o Git oferece; portanto, procuravam conselhos sobre como trabalhar com o Git, e muitas delas acharam esse conselho nesta postagem do blog. Ele descreve o fluxo de trabalho que Vincent e sua equipe usaram. É uma maneira muito específica de trabalhar que ficou conhecida como GitFlow. Não precisamos descrevê-lo em detalhes aqui, mas vale a pena dar uma olhada rápida nele. Em poucas palavras, o GitFlow é baseado em um modelo de distribuição centralizado. Todo desenvolvedor pode enviar para um repositório abençoado central. Mas o GitFlow também incentiva os desenvolvedores a trocar dados diretamente quando apropriado, portanto esse é um modelo de distribuição misto, centralizado com elementos ponto a ponto. O núcleo do GitFlow é seu modelo de ramificação, que é muito detalhado. Ele define um número de ramificações. Para entender as ramificações do GitFlow, você pode particioná-las amplamente em dois grupos, as ramificações instáveis, usadas para o trabalho de desenvolvimento, e as ramificações estáveis, usadas principalmente para lançamentos. Na maioria das ramificações instáveis, você tem uma ramificação de integração, denominada develop, e as ramificações de recursos, uma por recurso. Na segunda categoria, os ramos estáveis, você tem outro ramo de integração chamado mestre. Isso é diferente de desenvolver porque o desenvolvimento não é estável em geral, enquanto o mestre deve ser estável. Portanto, você só mescla o desenvolvimento ao mestre quando sabe que possui um sistema em funcionamento. E então você tem as ramificações da versão, uma separada para cada versão. E você também tem uma ramificação de hotfix. E depois existem as restrições. Muitos deles. Eles definem principalmente quais ramificações podem ramificar de quais outras ramificações e quais ramificações podem ser mescladas em quais outras ramificações. A propósito, eu digo mesclar, porque está sempre mesclando, nunca rebatendo. O GitFlow acredita em manter uma história verdadeira e rastreável. Você não deve mudar essa história rebatizando. E há mais regras sobre o que marcar e quando, e assim por diante. O GitFlow também fornece as convenções de nomenclatura que você deve usar para algumas de suas ramificações. Todas essas restrições significam que o GitFlow está bem definido e essa precisão tem algumas vantagens definidas. Por exemplo, você pode encontrar as extensões Git na Internet que fornecem comandos específicos para o GitFlow. Operações como criar uma ramificação de recurso ou mesclar um hotfix estão bem definidas o suficiente para que possam ser automatizadas. No mínimo, a precisão contribui para uma documentação muito boa. Essas são outras razões pelas quais o GitFlow se tornou popular. De qualquer forma, é muito popular. Tornou-se o fluxo de trabalho do Git para muitas pessoas, e muitas equipes apenas adotaram como estão. Agora, tenho uma opinião bastante forte sobre isso. Em uma frase, sugiro que você não use o GitFlow e comece a usá-lo. Agora, por favor, não pense que estou desrespeitando o GitFlow aqui. Eu não acho que haja algo errado com o GitFlow. É um fluxo de trabalho tão bom quanto você pode ter. E também inclui muitas práticas sólidas de controle de versão. O que eu recomendo é apenas pegar e usar qualquer fluxo de trabalho pré-enlatado, e muitas equipes fazem exatamente isso com o GitFlow, especialmente em grandes empresas. Eles acessam a Internet, encontram o GitFlow e o tornam obrigatório para todos. Mas existem muitos projetos para os quais o GitFlow não é a solução ideal. Por exemplo, projetos que podem evitar todas essas regras complexas de ramificação. Imagine que seu produto é um aplicativo da Web e não um aplicativo empacotado; portanto, você mantém apenas uma versão de produção por vez. Talvez você não precise de várias ramificações de lançamento. Você pode se safar com um. Ou talvez algumas das regras do GitFlow possam ser contraproducentes para você. Talvez você esteja em um projeto de ponta que faz implantação contínua. Ele implanta a produção toda vez que alguém integra um recurso estável. E todas essas ramificações introduziram muito na direção da integração à implantação. Algumas das restrições no GitFlow podem até incentivar comportamentos prejudiciais em projetos que não se encaixam nelas. Por exemplo, o GitFlow exige ramificações de recursos, e você deve mesclar uma ramificação de recursos ao desenvolvimento depois que o recurso for concluído. Porém, alguns grandes projetos herdados podem ter dezenas de recursos em desenvolvimento a qualquer momento, e cada recurso pode levar meses para ser implementado e tende a conter muito código. Portanto, quando esse processo é concluído e você finalmente mescla um recurso no ramo de desenvolvimento, tudo de uma vez, em uma única grande mesclagem, que pode causar grandes conflitos para outras pessoas na equipe. Em tais projetos, eu encorajaria as pessoas a se integrarem com mais frequência. Eu não sou um grande fã de ramos de recursos para dizer a verdade. Geralmente, prefiro fazer uma integração mais frequente quando posso. Mas algumas equipes respeitam essa restrição do GitFlow à letra e acabam tendo ramificações de recursos de folhas muito longas. Bem, você entendeu a minha deriva. O GitFlow é ótimo. É um fluxo de trabalho muito bem projetado, mas isso não o torna o fluxo de trabalho perfeito para qualquer projeto. Quando falamos sobre fluxos de trabalho, falamos sobre pessoas, a maneira como elas funcionam e, quando se trata disso, um tamanho definitivamente não serve para todos. Contexto é tudo, então, mais uma vez, não use apenas o GitFlow. Então, o que você deve fazer?

Crescendo um fluxo de trabalho Como você cria seu próprio fluxo de trabalho? Fazer isso é mais arte do que ciência, portanto não há regras rígidas. Sem receitas. Mas posso lhe dar uma orientação importante aqui, que é evitar a tentação de apenas sentar e projetar um fluxo de trabalho. Essa abordagem tende a gerar um fluxo de trabalho super-projetado e super-complicado que ainda não trata dos problemas específicos que seu projeto pode ter no futuro. Mesmo se você for muito esperto, ainda é difícil prever todas as situações possíveis em que seu projeto entrará, e mesmo se você for bastante experiente, isso ajuda, mas ainda não é suficiente porque cada projeto é diferente, geralmente de maneiras sutis, para que suas experiências passadas possam realmente enganá-lo. Portanto, em vez de criar um fluxo de trabalho, você deve se esforçar para aumentar seu fluxo de trabalho. Comece pequeno. Estou pensando muito pequeno aqui. Algo assim. Os detalhes realmente não importam aqui. Este é apenas um exemplo do que quero dizer com pequeno. Modelo de distribuição, o modelo centralizado neste caso, mas escolha o que for mais adequado ao tamanho do seu projeto. Um modelo de ramificação simples, sem complexidades, a menos que você saiba que essas complexidades são necessárias e um punhado de restrições. Três restrições, como neste exemplo, podem ser suficientes. Você sempre pode adicionar mais restrições posteriormente. Talvez você precise de mais do que isso se tiver um grande projeto ou se trabalha em uma organização tradicional e muito estruturada. Mas, de qualquer forma, tente ficar do lado pequeno. Você pode achar que seu fluxo de trabalho muito simples é quase tudo o que você precisa, na verdade, e quando se mostra insuficiente e você descobre que precisa de algo mais complexo, basta adicionar essa complexidade à medida que avança. Você pode adicionar restrições, adicionar novas ramificações, incorporar idéias de outros modelos de distribuição. É isso que quero dizer quando digo aumentar o seu fluxo de trabalho. Faça evoluir para se adequar ao seu projeto. Adicione apenas as regras em resposta a problemas reais que se tornam visíveis e sempre esteja pronto para remover regras que não estão tendo um impacto positivo no projeto. Um cavalheiro chamado Dee Hock disse algo que se relaciona com o que estou sugerindo aqui. Aqui está essa citação. Se você deseja um comportamento complexo e inteligente, crie princípios simples, porque princípios simples dão às pessoas o espaço necessário para serem flexíveis e inteligentes, para resolver os problemas inesperados que sempre surgem em um ambiente complexo. Por outro lado, regras e regulamentos complexos criam burocracia, eles criam um ambiente inadequado para a solução de problemas. Dee Hock foi o fundador e CEO da empresa de cartões de crédito Visa, então provou em primeira mão o quão poderosa essa ideia pode ser. Muitos de nós nunca resolverão o tipo de problemas complexos que ele resolveu, mas ainda tento manter essa citação em mente quando sou tentado a ser inteligente e adicionar regras e procedimentos às minhas equipes em minha própria maneira de trabalhar. Mantenha simples. Você raramente se arrependerá de fazer isso.

**GitFlow - os monorepos do Google e Facebook ou o monotree**

Em ambos os casos, os monorepos do Google e Facebook ou o monotree da kernel do Linux. Nenhum é o nosso caso comum, meu ou seu. Pra maioria esmagadora dos projetos, **o GitFlow do GitHub**, com um repositório principal, diversos forks, e pull requests que voltam pro repositório principal, e um único lugar pra registrar Issues, ainda é o caso de uso mais produtivo.

O fluxo de trabalho num GitHub envolve fazer forks dos repositórios, discutir Issues na interface gráfica do repositório, e receber colaborações via pull requests vindos de outros forks. O fluxo de trabalho da kernel do Linux envolve dezenas de repositórios distribuídos que são forks do repositório do Linux, com discussões acontecendo simultaneamente em dezenas de mailing lists, e esses e-mails servindo como canal de distribuição de dezenas de patches de código que são automatizados de diversas maneiras diferentes por cada empresa ou mantenedor. Uma das primeiras coisas que é diferente no fluxo é que no GitHub só tem como fazer pull request de volta pro repositório de onde se fez fork. Na kernel você quer distribuir os patches pra múltiplos repositórios de uma só vez. Além de pode discutir o assunto em paralelo entre diversos repositórios ao mesmo tempo.

É por essas e várias outras razões que, hoje, não tem como suportar o fluxo de trabalho da kernel no GitHub ou GitLab ou qualquer um que siga esse mesmo modelo e tenha essa escala. O fluxo deles segue uma árvore, onde o principal é a raíz e ele vai expandindo numa única direção pra cada galho. No caso do Linux é uma rede, um mesh, onde não existe direção definida, cada nó da rede pode se comunicar com qualquer outro nó da rede, sem um que coordena ou bloqueia essa comunicação. O Git foi originalmente feito com essa estrutura totalmente distribuída em mente. Hoje em dia nós simplificamos de uma forma mais centralizada, como num GitHub, pra ficar mais usável pra maioria dos mortais.

Em resumo, sim, se o GitHub ou GitLab resolverem criar uma estrutura de projeto de múltiplas árvores do mesmo repositório separados pra mantenedores diferentes com uma forma mais simples de unificar a cooperação entre repositórios, tanto do ponto de vista de Issues e Pull Requests distribuídos, talvez, bem talvez, um projeto como Linux poderia viver nessas plataformas. Mas seria um tanto de funcionalidade extra pra atender um nicho muito pequeno de projetos. Pouquíssimos projetos tem os requerimentos da kernel do Linux, por isso não vejo isso acontecendo tão cedo.

Pros colaboradores do kernel, tirando alguns subgrupos que podem trabalhar mais isoladamente; digamos, equipes de drivers ou algo assim, a grande maioria ainda vai ser mais produtiva com simples mailing lists e distribuindo patches mesmo, sem ficar na dependência de nenhuma plataforma controlando a comunicação.

E falando em sujeira, vamos falar rapidamente sobre monorepos. Esse é um daqueles assuntos que arrisca ser controverso, mas vou tentar simplificar pra vocês. Se você nunca ouviu falar dessa palavra, se não me engano começou com o Twitter. Mas o que interessa é o seguinte. Digamos que você é uma empresa de tecnologia gigante, mais especificamente uma com poucos produtos muito populares e centenas de desenvolvedores internos. Como num Twitter. Pior ainda, digamos que esse produto é composto por dezenas de bibliotecas e microserviços. Imagine o pesadelo logístico.

Você tem dezenas de pull requests aparecendo toda hora em dezenas de repositórios de código. Alguns deles podem acabar quebrando as integrações entre os diversos componentes. Num exemplo hipotético simples, digamos que num dos repositórios você atualize uma biblioteca de ... parsing de JSON por conta de uma melhoria de performance. Mas digamos que em outro repositório você precise voltar uma versão pra trás dessa biblioteca porque a versão nova causa um erro de compatibilidade com seu código. E agora imagine um terceiro repositório que importa o código desses dois repositórios, cada um dependendo da mesma biblioteca em versões diferentes. Fodeu.

Agora imagine situações como essa acontecendo na mão de centenas de desenvolvedores, toda hora, todos os dias. Imagine a dificuldade que é no final empacotar tudo junto pra gerar um deploy coerente. Milhões de linhas de código, divididos em dezenas ou centenas de repositórios, recebendo modificações o dia inteiro pela mão de centenas de desenvolvedores, sendo vários desses times distribuídos geograficamente e separados até por fusos horários diferentes, e no final precisa empacotar uma versão de produção sem bugs. Ah, e isso tem que acontecer com bastante frequência, tipo uma vez por semana ou mesmo uma vez por dia. Ah sim, e o deploy acontece em centenas de máquinas espalhadas em dezenas de data centers pelo mundo.

E pra piorar imagine 20 anos de histórico de código acumulado no caso de alguém como Google. Falando em Google, no caso dele estamos falando de mais de um bilhão de linhas de código. Mais de 25 mil engenheiros. E quase 20 anos de histórico num mono repositório gigante com todo o código da empresa que hoje já passou dos 80 terabytes de tamanho. Só de código! Esse é o tipo de desafio que empresas com Twitter, Google, Facebook, Netflix, e outros gigantes enfrentam todos os dias.

A kernel do Linux é um dos projetos open source mais antigos em atividade hoje, e com maior quantidade de código, e com a maior quantidade de voluntários colaborando ao redor do mundo. Dezenas de empresas, de dezenas de setores, dependem do Linux. A kernel em si é tecnicamente um monorepo. Mais corretamente ela é um monotree. Diferente de projetos dentro de um Google, onde a maior parte dos repositórios é centralizada e todo mundo clona de um único lugar. No caso da kernel, existe o repositório do Linus, só que o único desenvolvedor que trabalha nesse repositório é o próprio Linus.

Cada distro como Ubuntu, Fedora ou outros tem seus próprios repositórios principais, que não necessariamente clonam direto do Linus. Cada fabricante de hardware que desenvolve drivers tem repositórios diferentes. Mais importante, diversos produtos estão trabalhando em versões diferentes da kernel. Por exemplo, na própria Canonical que faz o Ubuntu, você tem o Ubuntu LTS que é o Long Term Service e o Ubuntu com versões mais novas dos componentes. A idéia do LTS é que ele atualize o menos possível pra não ficar quebrando compatibilidade o tempo todo.

Porém, digamos que seja descoberto um problema grave de segurança num driver na kernel nova. E esse bug afeta várias versões da kernel pra trás. O bug é descoberto no Fedora primeiro. Então a correção é feita no repositório deles. O patch da correção começa a ser distribuída em mailing lists. Cada empresa ou comunidade com um repositório tem seu mailing list. A forma de distribuir pra tanta gente é literalmente usar o recurso de CC ou carbon copy de e-mail, e assim distribuir o patch pra diversos lugares diferentes.