3
Gerenciamento

O gerenciamento de projeto envolve a realização de muitas tarefas tais como: compilação, execução dos testes, geração dos artefatos finais, controle de versões, disponibilização do produto para o cliente. Fazer qualquer uma dessas tarefas manualmente, além de possibilitar o aparecimento de erros, é uma completa perda de tempo. Por isso, é fundamental um mecanismo automatizado para auxiliar no cumprimento dessas tarefas.

3.1. *Build* Automatizado

Uma das práticas que mais claramente define o conceito de "evitar o desperdício de tempo" é o *build* automatizado. Em XP, a construção do produto final – incluindo a execução de todos os testes – deve ser feita de forma rápida e automatizada.

A automação do *build* é uma prática relativamente recente. A engenharia de software tradicional foi fundada em uma época em que os computadores eram extraordinariamente caros. Nesta época, encontrar e corrigir problemas na compilação dos lotes de programas era muito devagar. Até a metade dos anos 80, programas grandes demoravam muitas horas para serem compilados (McBreen, 2002). Nos últimos anos, no entanto, esta situação mudou. O poder computacional aumentou bastante e tornou-se cada vez mais barato, ao ponto de a maioria dos programas poderem ser recompilados em poucos segundos. É raro, até mesmo para aplicações muito grandes, demorar mais do que dez minutos para terminar o *build* (McBreen, 2002).

De forma simples, a automação do *build* pode ser descrita como o ato de automatizar (por meio de ferramenta ou script) o processo de compilar o código-fonte na sua forma binária. Com um processo de *build* manual, uma pessoa efetua múltiplas tarefas que são normalmente entediantes e passíveis de erro. O *build* automatizado envolve a automação de uma variedade de tarefas que o desenvolvedor de software precisa fazer nas suas atividades diárias, incluindo o

empacotamento do código binário, execução dos testes e o *deployment*[15] em um servidor de produção. O objetivo deste tipo de automação é criar um procedimento de um único passo para transformar código-fonte em um sistema funcional. Isto é feito para poupar tempo e diminuir a quantidade de erros.

O propósito de uma ferramenta de *build* é minimizar o tempo necessário para construir um programa usando as versões atuais do seu código-fonte. Para cada arquivo alvo do seu projeto, especificam-se as dependências e como construí-lo. Ferramentas de *build* também eliminam erros relacionados com a inconsistência de estado em que o código-fonte pode estar; estas garantem que o código será trazido para um estado consistente (McConnell, 2004).

Esta prática pode ser dividida em três categorias: 1) automação conduzida, aquela que é feita por cada desenvolvedor em sua estação de trabalho – na linha de comando ou usando uma IDE[16] – para verificar se o projeto em que está trabalhando está sendo construído corretamente; 2) automação agendada, feita por uma ferramenta que executa o *build* em determinados momentos; ou 3) automação disparada, quando o *build* é feito de acordo com determinadas condições (*commit*[17] no sistema de controle de versões, por exemplo).

Builds automatizados são muito mais valiosos do que *builds* que precisam de intervenção humana. No decorrer do projeto, os prazos vão expirando e a pressão aumenta. *Builds* manuais tendem a ser feitos com menos freqüência e com menos precisão, resultando em mais erros e maior apreensão. As práticas de XP tentam reduzir essa apreensão. O uso de *builds* automatizados torna banal esta

---

[15] O *deployment* de software é toda atividade necessária para que um sistema seja disponibilizado para uso (isso inclui compilação, construção, instalação e ativação, por exemplo).

[16] IDE (do inglês *Integrated Development Environment*) ou Ambiente de Desenvolvimento Integrado é um programa de computador que reúne características e ferramentas de apoio ao desenvolvimento de software com o objetivo de agilizar este processo.

[17] No contexto de ciência da computação, *commit* significa a idéia de tornar um conjunto de alterações temporárias em modificações permanentes. Em um sistema de controle de versões, o *commit* significa enviar as alterações feitas em uma estação de trabalho para um repositório de versões de arquivos.

atividade. “Algum erro foi cometido? É só executar o *build* e verificar” (Beck & Andres, 2004).

Se o processo de *build* não for automatizado, partes importantes do sistema podem ser construídas e testadas inadequadamente. Por isso, esta é a primeira prática que deve ser empregada em qualquer tipo de projeto, pois é realmente indispensável. A automação do *build* também é obrigatória para o funcionamento da integração contínua do sistema. Segundo a definição proposta em (Beck & Andres, 2004), *builds* automatizados devem construir o sistema inteiro e testá-lo em dez minutos. Se demorar muito mais do que isso, o *build* não será feito com a freqüência necessária, limitando a oportunidade de obtenção de feedback.

Mais do que simplesmente compilar o código, algumas ferramentas de *build* vão além e permitem a gerência do projeto em vários sentidos. Algumas ferramentas integram funcionalidades de verificação do código de acordo com um estilo, geram relatórios com métricas e informações sobre o projeto, gerenciam as dependências e produzem versões do programa.

Toda a configuração feita para automatizar o *build* também serve como documentação. Na configuração ficam informações como a localização das classes de teste, qual o tipo de empacotamento que deve ser feito para um determinado projeto, onde o artefato final deve ser instalado e outras. Basta olhar o *build* e todas essas informações podem ser obtidas.

3.2.Ferramentas

Para que a automação do *build* seja alcançada, é preciso que uma ferramenta seja utilizada e configurada para executar as devidas tarefas. A ferramenta escolhida deve ser rica o suficiente ao ponto de permitir, entre outras funcionalidades, a produção dos artefatos, a execução dos testes, o *deployment* em um servidor remoto, a geração de versões dos artefatos, a gerência das dependências do projeto e a geração de documentação.

Como a ferramenta de *build* será responsável pela realização de muitas tarefas fundamentais, mais cedo ou mais tarde o bom funcionamento do processo de desenvolvimento estará totalmente dependente dela. Por isso, é muito importante definir bem qual a ferramenta será empregada, pois esta será crucial

para que as outras práticas funcionem. Uma restrição na ferramenta pode impedir a utilização de outras práticas.

Além de ser completa naquilo que faz, é importante que a ferramenta de *build* se integre bem com o ambiente de desenvolvimento (IDE), mas que ao mesmo tempo não seja dependente deste. Caso contrário, muita perda de tempo pode ocorrer.

Para o desenvolvimento Java, duas ferramentas são muito utilizadas para automação: Ant e Maven.

3.2.1.Ant[18]

O Ant é uma ferramenta que possibilita a automatização do processo de *build*. Neste sentido, o Ant é muito parecido com a ferramenta Make[19], mas ao contrário do Make, o Ant foi projetado especificamente para o desenvolvimento Java. Um arquivo XML conhecido como *buildfile* especifica quais as tarefas devem ser realizadas pelo Ant na hora de construir um projeto. Por meio de classes Java, as tarefas do Ant são implementadas e podem realizar qualquer operação permitida na linguagem Java. A API[20] do Ant é aberta e feita para ser estendida; se for necessário, qualquer pessoa pode escrever um novo tipo de tarefa (Burke & Coyner, 2003).

O ciclo de construção e distribuição também deve ser automatizado para não incorporar erros do operador. Escrever um script de *build* serve como documentação do processo de construção do sistema. Possuir estes dados torna-se crítico quando um desenvolvedor sai de uma empresa. Usando o Ant, a empresa retém o conhecimento necessário para distribuir o sistema, uma vez que o ciclo de construção e distribuição está automatizado por um script Ant e não esquecido na cabeça do desenvolvedor que foi embora (Hightower et al., 2004).

---

[18] http://ant.apache.org/

[19] O Make é um programa de computador concebido para compilar automaticamente o código fonte de um programa. Este utiliza instruções contidas num arquivo chamado "Makefile" e é capaz de resolver as dependências do programa que se pretende compilar.

[20] *Application Programming Interface* – ou simplesmente API – é um conjunto de rotinas e padrões estabelecidos por um software para utilização de suas funcionalidades. De modo geral, a API é composta por uma série de funções acessíveis somente por programação e que permitem utilizar características do software menos evidentes ao usuário tradicional.

Outra vantagem de usar o Ant é o formato do *buildfile*. Por ser escrito em XML, é fácil de ser lido e compreendido. Dessa forma, a configuração da automação do processo de *build* e *deploy* torna-se uma documentação deste processo. Onde os artefatos finais devem ser instalados? Onde está o código-fonte? As classes de teste devem fazer parte do produto final? Basta olhar o script e ver a resposta.

A seguir, um exemplo de um *buildfile* simples do Ant:

```
<project>
    <target name="limpar">
        <delete dir="classes"/>
    </target>
    <target name="compilar">
        <mkdir dir="classes"/>
        <javac srcdir="src" destdir="classes"/>
    </target>
    <target name="empacotar" depend="compile">
        <mkdir dir="build"/>
        <jar destfile="build/HelloWorld.jar" basedir="classes"/>
    </target>
</project>
```

Neste exemplo são definidos três objetivos: `limpar`, `compilar` e `empacotar`. Com esse *script* é possível compilar o código-fonte de uma aplicação e gerar um arquivo HelloWorld.jar simplesmente executando o comando `ant empacotar`.

O Ant é popular porque é fácil de aprender e estender. Além disso, muitas IDEs e ferramentas de desenvolvimento suportam o Ant, como por exemplo o Eclipse, Netbeans e JBuilder. Muitos projetos *open source* também utilizam Ant. Por isso, o Ant tornou-se muito utilizado para a automação de projetos Java.

"Uma boa ferramenta de *build* como o Ant é decisiva para o sucesso no uso das práticas de XP. Não se pode esperar que uma equipe de programadores faça *refactoring* do código constantemente, execute os testes de unidade e integre suas modificações sem um ambiente de *build* previsível e veloz" (Burke & Coyner, 2003).

3.2.2.Maven[21]

Além de controlar o processo de construção de um software, o Maven oferece uma abordagem abrangente para gerenciar projetos de software. Desde a compilação até a distribuição, incluindo documentação e colaboração entre a

[21] http://maven.apache.org/

equipe, o Maven oferece as abstrações necessárias para encorajar o reuso e diminuir muito do trabalho para fazer os *builds* de um projeto (Massol & Van Zyl, 2006).

O Maven é baseado em quatro princípios fundamentais: 1) convenção ao invés de configuração; 2) execução declarativa, de acordo com o modelo de projeto definido; 3) reuso da lógica de *build* por meio de herança; e 4) organização coerente das dependências, por meio de repositórios locais e remotos.

O objetivo desta ferramenta sempre foi tornar mais fácil para os desenvolvedores a tarefa de seguir métodos ágeis, tornando um pouco difícil escapar de algum deles (por exemplo, para que um *build* seja bem sucedido, não basta que o código compile corretamente, todos os testes também precisam passar). Outros objetivos do Maven são:

- Tornar o processo de *build* mais simples.
- Oferecer um sistema de *build* uniforme.
- Fornecer informações sobre a qualidade do projeto.
- Proporcionar um guia claro sobre o processo de desenvolvimento de software.
- Apresentar orientações para a completa aplicação de práticas de teste.
- Possibilitar uma visualização coerente das informações de um projeto.
- Permitir uma migração transparente para novas funcionalidades.

O Maven consegue satisfazer esses objetivos e continua sendo melhorado graças a sua arquitetura baseada em *plug-ins*. Alguns benefícios proporcionados por esta ferramenta são:

- ***Layout dos projetos bem definido***: A maioria dos projetos adere ao layout básico de um projeto, o que torna muito mais fácil para o desenvolvedor navegar e procurar por itens em qualquer projeto. Projetos que seguem a estrutura padrão de diretórios precisam de menos configurações.
- ***Teste de unidade embutido***: O *plug-in* do Maven para o JUnit oferece as funcionalidades necessárias para que todos os testes de unidade de um projeto sejam executados.
- ***Visualização do código e dos relatórios***: O Maven possui vários relatórios prontos para serem gerados de forma a ajudar a equipe a focar no projeto e

os problemas relacionados com ele. Todos estes relatórios podem ser integrados na construção de um site com documentação sobre o projeto.

- ***Integração com outras tecnologias ágeis***: O Maven inclui *plug-ins* para cobertura de código, controle de versão, formatação de código, verificação de violação de estilo de código e rastreamento de *bugs*.

Por causa do conjunto de padrões, do formato de repositório para controle de dependências e do componente de software usado para gerenciar e descrever projetos, qualquer pessoa que conheça a estrutura de um projeto que utiliza o Maven terá muita facilidade para trabalhar em qualquer outro projeto controlado por esta ferramenta. Essa é uma das grandes vantagens em relação ao Ant. Portanto, o uso dessa ferramenta é altamente recomendado para qualquer tipo de projeto.

O Maven fornece uma linguagem comum para descrição de projetos. Sistemas que seguem essa abordagem declarativa tendem a ser mais transparentes, mais reusáveis e mais fáceis de serem mantidos e compreendidos. Logo, se você pode construir um projeto usando o Maven, você é capaz de construir qualquer outro projeto que use o Maven; se você pode aplicar um *plug-in* de testes para um projeto, então é possível aplicar para todos os projetos. Você descreve o seu projeto usando um modelo do Maven e ganha acesso a particularidades e boas práticas de uma comunidade inteira (Massol & Van Zyl, 2006).

A seguir, um arquivo POM (sigla para *Project Object Model*), modelo para descrição de projetos do Maven:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>meu.grupo</groupId>
    <artifactId>projeto-exemplo</artifactId>
    <version>1.0</version>
    <name>Projeto de Exemplo</name>
    <packaging>jar</packaging>
    <dependencies>
          <dependency>
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.0</version>
                <scope>test</scope>
          </dependency>
    </dependencies>
</project>
```

Neste arquivo são definidas informações como: o nome do artefato (*artifactId*), o nome do projeto (*name*), o tipo de artefato que deve ser gerado (*packaging*) e as dependências do projeto (*dependencies*), entre outras

informações. Seguindo a estrutura proposta pelo Maven, com um arquivo simples como esse, é possível compilar o código-fonte, executar todos os testes, gerar o artefato final (um arquivo JAR) e gerar uma documentação básica sobre o projeto.

3.3.Precauções

Sem sombra de dúvidas, o *build* automatizado é o coração do funcionamento de um projeto. Por isso, é importante que todos os programadores tenham um conhecimento razoável sobre a ferramenta de *build* que estão utilizando. Uma dica é começar por essa prática antes de qualquer outra. Com o *build* automatizado, menos tempo será desperdiçado.

E se a equipe não tiver tempo para aprender a usar uma nova ferramenta? Então ela terá que ter tempo para os problemas. Se a equipe for pequena e o número de projetos grande (cinco desenvolvedores trabalhando em quatro projetos, por exemplo), o problema será ainda maior, pois praticamente cada desenvolvedor terá que criar o seu arquivo de *build*, aumentando a possibilidade de todos passarem repetidas vezes pelo mesmo problema.

Enquanto a equipe inteira não entender como usar a ferramenta de *build* como uma parte integrada do ambiente de desenvolvimento de software, ela não estará pronta para iniciar a primeira etapa do desenvolvimento (McBreen, 2002). Se a equipe não tiver tempo suficiente para se familiarizar com as ferramentas, será muito fácil voltar para a forma como era feito anteriormente – sem práticas de XP – o que pode facilmente significar a ruína de um projeto (McBreen, 2002).

Algumas boas práticas com relação ao uso da ferramenta de *build* também devem ser levadas em consideração:

- O arquivo de *build* deve ficar no diretório raiz do projeto. Algumas ferramentas de *build* permitem que o arquivo de configuração de um projeto fique em uma pasta dentro ou fora do projeto. Essa informação faz parte da descrição do projeto, por isso deve estar dentro dele. É aconselhável que o arquivo de *build* fique no diretório raiz porque, assim, quando for necessário construir um projeto, basta ir para o seu diretório raiz e executar o comando especificado. Além disso, qualquer pessoa que obtenha o projeto já saberá qual o tipo de ferramenta está sendo utilizada apenas pelo nome do arquivo de *build*.

- Usar convenções e um estilo consistente para estruturação do projeto e configuração do arquivo de *build*. A liberdade que algumas ferramentas oferecem para a definição de ações pode dar mais trabalho. Quando se tem mais de um projeto, a mesma ação pode ser executada de forma diferente em cada um deles. Por exemplo, a ação que limpa os arquivos gerados pode ser executada de várias formas diferentes: `clean`, `clear` ou `limpa`. Com o uso do Maven este tipo de problema não existe. Devido aos padrões definidos, cada etapa do ciclo de construção tem um nome pré-determinado.
- Não se repetir (do inglês *Don't Repeat Yourself*). Esse é uma das frases adotadas pela turma de XP que implica em não permitir duplicação. No caso dos *builds*, isso significa não ficar repetindo as mesmas configurações em vários projetos. Algumas ferramentas, como o Maven, por exemplo, permitem que a lógica de *build* seja reaproveitada por meio de herança das configurações.
- Comentar os scripts de *build*, oferecendo ajuda para pessoas que não estão familiarizadas com o projeto. Algumas ferramentas permitem que meta-dados sejam associados às tarefas que estão sendo realizadas. Escreva mensagens que ajudem a esclarecer qual tipo de erro pode estar acontecendo.
- Gerenciar as dependências apropriadamente com o uso de uma ferramenta como o Maven.
- Utilizar propriedades específicas do usuário para permitir que as configurações definidas por padrão sejam sobrescritas, como, por exemplo, localização de diretórios e conexões com o banco de dados.
- Manter o *build* independente de intervenção humana. O *build* deve ser completamente automatizado. A única tarefa que pode ser manual é a inicialização do *build*, que pode ser feita por um desenvolvedor ou por uma ferramenta de integração.
- Conservar a configuração do *build* em um sistema de controle de versões, junto com o código. Para construir um projeto em uma determinada etapa, deve-se utilizar as configurações daquele momento. Novas configurações podem ser incompatíveis com o estado do projeto em uma fase anterior.

Além disso, as configurações de *build*, assim como o código, vão evoluindo com o decorrer do projeto. Isso significa que erros no *build* também podem aparecer. Nestes casos, é importante possuir maneiras de voltar atrás, para uma versão que funcione.

O *build* automatizado promove um ganho muito grande de tempo, além de atenuar a preocupação com as tarefas de construção e distribuição de sistemas. Se for decidido que mais nenhuma prática de XP será adotada, certamente a automação do *build* não será descartada. Apenas com o uso desta prática, muito tempo que antes era desperdiçado ao fazer um *build* ou *deployment* manual poderá ser empregado em tarefas mais nobres.