JAXenter: *Hello Mr. Dockter! What was - and is - the main focus in the development of the Gradle build system?*

Hans Dockter: We believe that project automation is an essential component of successful software development. The requirements of project automation have increased significantly in recent years, and now go far beyond the traditional build. We want our tool to make it is as easy as possible to meet these requirements.

We are convinced that the declarative approach is the key to easy-to-maintain and extensible Enterprise builds. With the word 'declarative,' we mean that the user only needs to determine the 'what' and not the 'how.' An example of a declarative element is the Maven packaging and dependency element. Users can define whether the project is, for example, a Java or Web project ('what,') and which dependencies it has ('what.') The 'how' (compiling, copying resources, retrieving of dependencies, etc ...) is then figured out by Maven. This is not to be confused with build-by-convention, and also works for a custom layout. Then, one only has to give a few more specifications about the 'what'.

Gradle takes Maven's declarative approach a step further. With Gradle:

There are declarative elements on different abstraction levels.

- It is possible to customise the behaviour of the standard elements, or add your own language elements (via plugins or directly in the build script.)
- All the features are integrated into the build-by-convention approach.
- The declarative elements form a language. The users can use this language to model their project at a high level of abstraction. Often, the build master creates the model for the entire department or organisation, and makes it available as a plugin.
- Gradle also offers a plugin that implements a standard Maven layout. However, this layout is not privileged and you can extend this plugin (for example, have the integration tests in the same project.)

Declarative elements can be very powerful. Gradle, for example, provides the source-set-element for describing the 'what' of a logical group of sources (such as location, class paths, etc....) The user need only declare such an element, to automatically receive the appropriate actions for it (compile, resource handling, ...)

The declarative layer is based on a powerful imperative layer, which can be accessed directly as required.

Gradle allows the user to structure their build-logic appropriately. It does not enforce indirections if they are unnecessary, but can allow them in a high degree if required. With Gradle, you get a build system where it is possible to apply the common design principles of software development. We believe this is very important for non-trivial builds. Simple builds are also easy to implement in Gradle. Gradle scales very well to your particular requirements.

In our experience, performance is often a major problem for builds. This is particularly true of

builds that automate as much as possible. Gradle scales very well in terms of performance. There are huge enterprise builds which use Gradle productively (the largest has nearly 700 sub-projects). Gradle's features, such as the incremental build and the parallel execution of tests, can often reduce the average build time dramatically.

Gradle welcomes the unexpected. We are always surprised and amazed at the unexpected needs of our users, and pleased that Gradle provides a way to implement them. Gradle's extensible build language is one of the factors that facilitates this. Another, is the huge amount of powerful extension points into which one can plug in custom code to make the necessary adjustments. This applies to the configuration of the build, in addition to its execution. To take just one example: with Gradle it is very easy to expand a CI build so that it sends an email to the author when a unit test in the Package *com.mycomp.db* lasts longer than one second. You can also expand Gradle with your own extension points (via plugins or init scripts.)

JAXenter: *In your opinion, what are Gradle's strengths compared to other build systems, in particular Maven and Ant?*

Hans Dockter: Gradle offers many features that make every-day build processes easier. These include a generic function to skip certain aspects of a build, without the aid of a Build-Master. In Ant, the developer has to use Skip Properties to miss out build aspects; in Maven he needs to adjust the plugins – if this is possible.

Gradle also offers our beloved Camel Task Execution for abbreviating task names when running in the console. Gradle has a very dynamic output, which includes a counter for the executed tests. There is a 'stand alone' GUI, which can be accessed with `gradle --gui`. There is a wrapper for building projects with Gradle, without the need to manually install Gradle. This wrapper also means zero-administration for Gradle on the client side. I could go on!

The original motto of Gradle was: "Make the impossible possible, make the possible easy and make the easy elegant.” This motto summarises the contrasting characteristics of Gradle, and Ant and Maven:

Ant
Ant is a pure, imperative build system, which means Ant has the flexibility to make the impossible possible. However, often Ant doesn't make development easy and elegant, which frequently results in builds that are difficult to maintain and extend. Ant lacks the declarative element and the conventions. Ant does not model the problem space. The Ant-model consists of Task, Target, Project, Resources and Properties. It does not understand source directories, project dependencies, etc. ... This, the user has to model himself - and with the limited language elements that Ant provides. Gradle offers a rich, extensible model to model software projects - without losing the flexibility of Ant. This flexibility can be accessed when you need it; otherwise, it rests in the background of the declarative build description.

We consider Ant not primarily as a competitive build system, but as ageing friend and helper.

This is especially true when it comes to the treasure of existing Ant tasks. These are First Class Citizens in Gradle. You can also easily wrap them in a Gradle-plugin, and get Convention over Configuration. You can load an Ant-build in Gradle-runtime and integrate it with Gradle. You can define dependencies of Gradle tasks to Ant targets and vice versa. You can also extend Ant targets in Gradle. This way you can very gently migrate - in whole or in part – from Ant to Gradle.

Maven

Even if a number of requirements can be easily realised with Maven, far too often it's very difficult to solve relatively easy problems with Maven. Project Automation is an important and critical component of successful software development but frequently we keep encountering things that are not automated, because it is too difficult to integrate with the build systems in use. This particularly applies to the requirements which cannot be anticipated by the build system. The implementation of such requirements often require workarounds, which result in high maintenance costs. I would also like to quote Erich Gamma:

Frameworkitis is the disease where a framework wants to do too much for you, or to do it in a way that you don't want, but can't change. It's fun to get all this functionality for free, but it hurts when the free functionality gets in the way. But, you are tied into that framework. To get the desired behaviour you start to fight against the framework. And at this point you often start to lose, because it's difficult to bend the framework in a direction that it didn't anticipate. Toolkits do not attempt to take control for you, and therefore they do not suffer from frameworkitis. Erich Gamma

This quotation corresponds to our intensive experience with Maven, and we do not anticipate anything fundamentally changing with Maven 3. This is not an issue with XML. Gradle follows the path recommended by Erich Gamma and offers small, optional frameworks based on toolkits.

Gradle extends the declarative approach of Maven, making it suitable for the full range of company-specific requirements (see above.)

Gradle offers very powerful support for multi-project builds. Project dependencies are First Class Citizens, in contrast to Maven, where dependencies are modelled in the same way as normal external dependencies. This allows for many optimisations (eg, partial builds,) and makes this information available to the users. Gradle doesn't use inheritance to define the common features of sub-projects. Instead, there is 'configuration injection' where a user can inject shared configurations into any group of subprojects. The Maven Mix In functionality for poms (which is postponed to Maven 3.1) is heading in the same direction, but is not as powerful.

There is one final aspect that often leads to considerable debate. Many Maven users like the very restrictive, unchangeable Presettings of Maven. In their opinion, they result in builds that are easy to understand, and easy to maintain. We believe that the opposite is often the case. What happens when you have to squeeze complex requirements into a simplified model? First, it is laborious. Often, it is so complex that it cannot be done. The result: the build rests as it is. But, what happens when you squeeze? Then, it often begins to become 'smelly' (in the words of Kent

Beck.) You have to tear apart what belongs together (Shotgun Surgery,) or you get unnecessary indirections (such as Lazy Project.) The worst thing is that the developer is forced to violate a fundamental principle of domain-driven design: make explicit the implicit. Such a build may at first glance look simple, but in reality it is often a maintainability monster.

We believe that it is important to have a clearly structured process for the build within an organisation. Gradle supports this more than any other build tool that we know of. But, the organisation must be able to create these processes themselves. Gradle offers a set of standard processes, but these are not privileged, and they are extensible. You can also design your own processes or use a mix.

JAXenter: *What can Gradle learn from other build systems?*

Hans Dockter: We have obviously learned from Ant and Maven, and much of this is incorporated into Gradle. But now, this process is more or less complete. In the future, Gradle will learn mostly from build systems related to other platforms and languages. These often provide very powerful features that we don't have in the Java world – features many people are unaware of. One example of such a feature is a powerful, generic incremental build. We have implemented this feature in Gradle 0.9. Another interesting feature are the rule-based approaches which are applied to input / output patterns. These help to define transformations, for example files which match a certain pattern. Rake and SCons are two build systems we're taking a closer look at. Basically, we think a 'Rich Model' for the input and output is very important for implementing efficient solutions for parallel and distributed builds.

And finally, there are other nice things we want to offer soon, such as incremental testing.

JAXenter: *Thank you very much!*

Hans Dockter is the founder and project lead of the Gradle build system and the CEO of Gradle Inc, a company that provides training, support and consulting for Gradle and all forms of enterprise software project automation in general. Hans has 13 years of experience as a software developer, team leader, architect, trainer, and technical mentor. Hans is a thought leader in the field of project automation and has successfully been in charge of numerous large-scale enterprise builds. He is also an advocate of Domain Driven Design, having taught classes and delivered presentations on this topic together with Eric Evans. In the earlier days, Hans was also a committer for the JBoss project and founded the JBoss-IDE.

This is an article from JAXenter. It can also be found here with similar questions being asked for Ant and Maven.