# Java Productivity Primer

## Twelve guidelines to boost your productivity with a software factory

October 2008


Cabinet d'Architectes en Systèmes d'Information

# Table of Contents

# Summary

Development teams want Java to help them achieve better productivity. This paper provides a set of guidelines to help you be more productive with Java when developing enterprise applications. It includes a quick introduction to modern development principles and guides you through the fundamentals of organizing your activities around a productive software factory, i.e. a set of tools that automates repetitive tasks and helps implement an efficient development workflow. Although experts may already be familiar with some or all of these guidelines, they will serve as a good way to start discussions and share knowledge.

The authors, a team of software architects at OCTO Technology, are responsible for designing and supporting such a software factory inside a large financial institution. This paper is the fruit of our hands-on experience, the lessons we have learned in our quest for productivity and feedback from the OCTO software architect community.

# Introduction

It can be challenging to adopt new architectural patterns (e.g. Web-based applications) and develop enterprise-grade software using the ever-evolving Java platform. The Java ecosystem itself, whose great vivacity comes at the price of great complexity, is often baffling for newcomers to the platform. Some are accustomed to the productivity provided by classic client/server development environments like PowerBuilder or VB, and have a hard time figuring out how to attain, let alone surpass, this level of productivity in Java.

**So, you want to get productive with Java?**

Clearly, the challenge of productivity, widely acknowledged by Java users and vendors, is one of the most worked-on issues in the Java community. For instance, productivity is the primary focus of the new Java Enterprise Edition (JEE 5 [1]) effort led by SUN and has led to large-scale changes being made to the Java language itself, starting with Java 5 [2], and the adoption of better programming models coming from the lightweight containers movement. Still, Java enterprise development continues to be a substantial undertaking and presents many hidden traps and dead ends.

Organizations often have legacy systems, integrated in a number of different ways, and many data flows and repositories of various levels of quality; all forming layers and layers of software that we need to interact with in our Java projects.

Faced with these challenges, the Java development community is adopting productivity-enhancing practices, combining efficient, agile development tools and processes into what are called software factories. This document will introduce you to these forms of development. It is structured along twelve practical guidelines sprinkled with easy-to-use advice.

**What is productivity, anyway?**

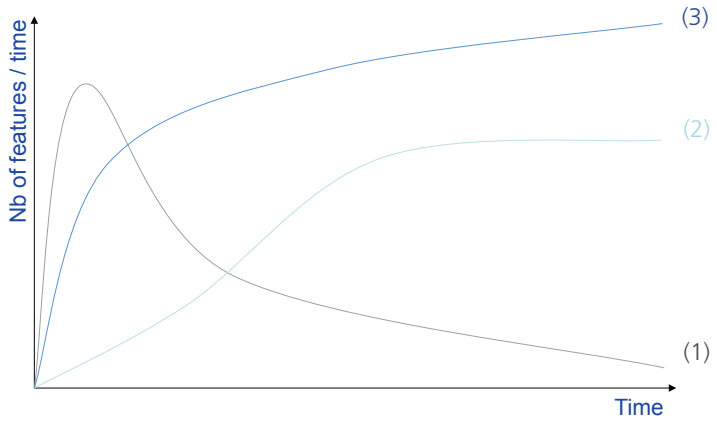
Although productivity is hard to define, we think that a good indicator for measuring productivity is the number of error-prone features added to the software during a fixed amount of time.

This definition includes 3 things :

- A "number of features": the simple part; if you cannot add the features that have been requested, your productivity is nil.
- "Error-prone": A feature that doesn't work is worthless so you don't count it. You may want to count a delivered feature with just a non-blocking bug. But be aware that this will probably have a negative impact on your future productivity, for at least two reasons. First, this bug may cause problems when adding the next feature. Second, when you have dozens of bugs to fix, you can't work on real features.
- "During a fixed amount of time": Here, we want to choose an amount of time that is small enough to let us take measurements often. Remember that if you have only a few measurements, this indicator means very little because it is very difficult to compare it across projects with different team sizes, different project durations, etc. Plus, you want to be able to detect the effect on your productivity early, whether positive or negative; this too advocates for a small amount of time, from one week to one month.

**Keep an eye on productivity over time.**

While you should measure often, it is also important that you look at productivity over the long term, as you want it to be as high as possible, all the time. So, it is very likely that you will go through periods with very low productivity because you are modifying existing code to ensure that your future productivity will be as high as possible.



In this graph, we show three extreme profiles of productivity that could be measured on projects.

(1) The first shows projects that jump right into code, using the simplest tools available to do what they have to do. Unfortunately, as they grow they reach the limits of these tools so productivity drops quickly. Since it wasn't expected, it's difficult to reverse this tendency.

(2) The second shows projects that try to anticipate everything. A great deal of time is wasted choosing the best possible tool, choosing the best logical architecture, and checking that everything looks as it should. Initially, productivity is very poor, but that's alright because "these good choices will help us to increase our productivity". Although this might be true in some cases, in most cases, they result in overly complex systems, and productivity debts are never paid back.

These two caricatured profiles offer different advantages: the first has high productivity at the beginning, and the second one has better productivity after some time. We could argue which approach is best, but we would miss the essential point. In fact, they both suffer from the same problem: productivity goes down! That is what is really essential.

What you're really looking for is high productivity very early on that **keeps increasing over time**. This characteristic is represented by the third line in our figure.(3)

The first twelve guidelines we present will help you to set up conditions that will help you keep this characteristic in your projects.

# Productivity Guidelines

## Guideline 01 – Adopt a productive Java platform that's tailored for your project(s).

Our experience makes us quite optimistic about the possibility of productive Java development, and you will read a lot about that in the coming chapters. However, we will start by sharing some bad news with you – a painful realization we all have to come to: we can't just blindly buy some existing Java platform, with all its bells and whistles, and use it as is, trusting that "it is the software vendor's job give us a best of breed, productive platform".

This just doesn't work. Keep in mind that, generally speaking, the primary goal of software vendors is to sell you as much software as possible, not to improve your productivity. You have to step in and lead the productivity effort!

One pillar of this effort is choosing a productive Java platform that will work well in your context. This is important because there isn't just "one" Java platform out there, but an amazingly rich industrial ecosystem made of thousands of tools and libraries provided by many vendors and organizations.

J2EE 1.4 [3], a "standard" enterprise Java platform that used to be promoted by SUN and various vendors, suffered from many productivity problems, which have been discussed abundantly in the development literature . This situation gave rise to a number of alternative frameworks with much better productivity profiles. Indeed, SUN has incorporated many of their characteristics and features in newer versions of its reference platforms, like the recently released JEE 5 (the successor of J2EE 1.4).

However, JEE 5 is still lagging with respect to development productivity and only defines a generic Java platform, not the actual platform that should be instantiated for your project(s). You'll very likely use only parts of JEE 5, combined with additional third-party or in-house frameworks and tools.

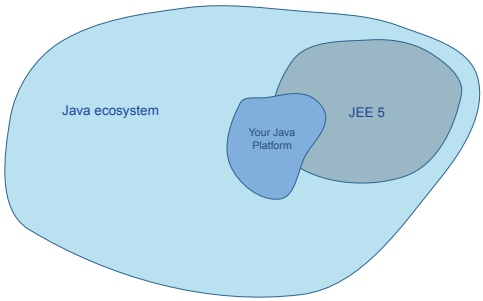



**Figure 1 – Choosing the right set of technologies**

Competing Java tools and frameworks have not matured to the point where they can be considered more or less equivalent. Far from it: bad architectural and technological choices can seriously hurt your productivity and the quality of your software. This is why it's important to choose and put together the right set of Java technologies.

## Tips & Tricks

When people ask us "*What is the best Java architecture?*", "*What's the best framework?*" or "*Which platform should I start my project with?*" we are often stuck because there is no one-size-fits-all solution. It depends on so many things like team experience, team size and legacy system that we just can't answer that question without specific information on the context. However, we already have some pieces of the equation:

- We need to be productive right now.
- A solution that's effective for big projects with dozens of developers over several years is probably overkill for a small project with one or two developers over a few weeks.
- We know that a project we start now will likely run into the same problems we've already had in the past on others projects.
- We don't know when we will face one particular problem, if ever.

Our answer to this question is to start with a simple platform so that you can be as productive as possible. But don't choose the shiniest solution. Just start with the platform that raises the first most important obstacle you have met and that is scable enough to allow you solve upcoming problems. Improvements will be made day after day.

Don't pay the highest price for everything that might happen, just pay the right price at the right time.

Below is an example of a Java platform we've used on several Web projects. It shows some of the components we have found to be especially useful:

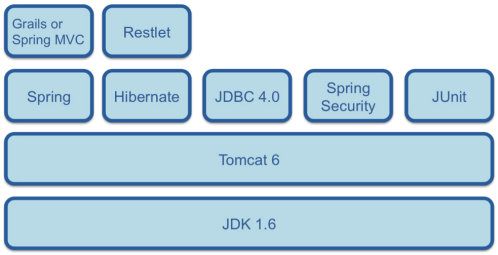



**Figure 2 – Example of a state-of-the-art, low-cost Java platform**

At the time of writing, the **JDK** (Java Development Kit) 1.6 is the latest and best Sun JDK release. It provides a standard Java development and run-time platform, and includes recent advances in Java that greatly improve productivity. Some of these advances come from evolutions of the Java language itself (e.g. ***properties***, introduced with Java 5, which greatly reduce the need for XML configuration files). Others come from new or improved Java frameworks and better performance [21].

**Tomcat 6** is a high-quality open-source application server. It provides a Servlet/JSP engine and includes a Web server.

**Spring [4]** is an open-source framework that helps implement recurring patterns typical of enterprise applications (e.g. component configuration, object lifecycle, data access, transaction management, etc.). In recent years, Spring has established itself as a popular alternative to J2EE by providing more productive programming models and a simpler, more enjoyable development framework. Spring also strongly promotes the testability of its application components. Later in this book, we will discuss how testing can (somewhat surprisingly) become a key productivity enabler.

**Hibernate [5]** can boost the productivity of developments related to database interaction. As a sophisticated object-relational mapping framework, it helps automatically create and synchronize an object graph in memory from/with the corresponding data in the relational database. Hibernate inspired and conforms to JPA (Java Persistence API), SUN's latest "standard" object persistence specification.

**JDBC** is the low-level API for interacting with Java databases, using the SQL language. For some time, it has been considered tedious to use JDBC directly; a situation which led to the development of object-relational mapping frameworks like Hibernate. However, **JDBC 4.0 [22]** is

a major evolution of the JDBC programming model and is more productive than previous versions. In some situations, it makes sense to develop with it instead of the higher-level Hibernate stack.

**Spring Security [23]** (formerly Acegi) and JUnit [17] are productive open-source frameworks for dealing with security and unit testing, respectively.

For developing Web user interfaces (generating HTML, managing user navigation) we often recommend using Spring MVC or Grails.

**Spring MVC [24],** an optional Spring component, is a powerful request-based MVC framework and a nice improvement over the classic Struts.

**Grails [25]** is in a league of its own. It is based on the Groovy programming language, a higher-level dynamic language that runs inside the Java platform and can seamlessly interact with Java objects. Grails can allow spectacular productivity gains, provided it fits with the development team's culture and preferences.

Finally, for implementing Web services, we really like the Restlet framework [26]. Note that in this document, we use the term Web services to refer to services that can be accessed by other applications over HTTP. The Restlet framework is built for the REST architectural style, which is HTTP's native architectural style. It lets you leverage the Web's intrinsic qualities (scalability, loose coupling, etc.) much better than RPC- and/or SOAP-oriented frameworks (e.g. JEE's JAX-RPC/JAX-WS). For more on this, see [27].

## Key Points

- There is no *one true* Java platform but a complete ecosystem from which you will build a runtime and development platform tailored to your needs.
- Be wary of software vendor pre-built platforms and architectural blueprints. They are often bloated and will be inefficient if used as is, without consideration of your real needs and other contextual matters.
- Recent advances in Java development can make you much more productive. Monitoring evolutions taking place in the Java space and updating your platform appropriately will help you leverage these new technical and organizational approaches.
- Choose a simple and productive platform that you know provides answers to your problems: pay the right price at the right time.

## Guideline 02 – Set up a software factory adapted to the size and organization of your development team

As you well know, a project needs a technical infrastructure to allow developers to do things like:

- Design and produce code
- Coordinate modifications to the shared code base with the rest of the team
- Manage the build process (this includes dealing with packages dependency and other niceties)
- Follow project advancement: what is done and runs ok, what is missing or unfinished
- Address bugs and other issues in a traceable way
- Test...

As teams often learn the hard way, installing an IDE on every developer's workstation and starting coding just doesn't cut it. A software factory is a set of development tools, models and processes whose goal is to industrialize your developments by identifying discrete actions (such as those listed above) and helping you perform them in a productive, highly structured and automated manner.

**Does a software factory relegate developers to mere factory workers on an assembly line?**

Not at all. On the contrary, the software factory described in this document takes care of most of the tedious, repetitive, error-prone work, and actually frees developers by automating it. Developers can then concentrate on more interesting and demanding tasks, innovate, solve real problems, have more fun in their programming activities, and, hopefully, create good software.

Your software factory should cover the various needs of your Java project(s), from programming (providing an IDE and a set of selected frameworks and code samples) to integrating your code with other team members, testing, tracking issues and so on. Furthermore, we recommend adopting a modular approach if you need to tailor your factory to various projects and contexts.

Setting up a complete software factory can be time consuming, and we've seen some projects, ours included, that spent more time on setting up and tuning their software factory than on actual coding. In the end, they didn't save any time. So, once again, we recommend you set

it up incrementally, one or two elements at a time: **set up the smallest factory you need to be the most productive and commit to continuous improvement in order to be the most productive throughout the project life.**

For instance, you could start by putting in place a shared source code repository, then a continuous integration server and then progress further with an issue-tracking tool, and so on (we will review these tools later in this paper). This should avoid overwhelming your team with too much stuff at once and let you progressively adopt new tools and practices. Along the way, you can also refine your factory, as your mastery improves.

## Key Points

- Software development with Java entails many different and complex tasks.
- Automation is the key for sustained productivity.
- Industrialize your developments by setting up a software factory.
- Set up the smallest software factory for your needs, to be most productive quickly, and commit to continuous improvement.

## Guideline 03 – For small projects, set up a small software factory

Your factory setup will vary depending on your team's organization. Having just one developer (or just one pair, if you practice pair programming) ensures the best level of productivity, since all the tasks related to synchronization with other developers vanish. In such a case, you will install a minimal software factory on the developer workstation.

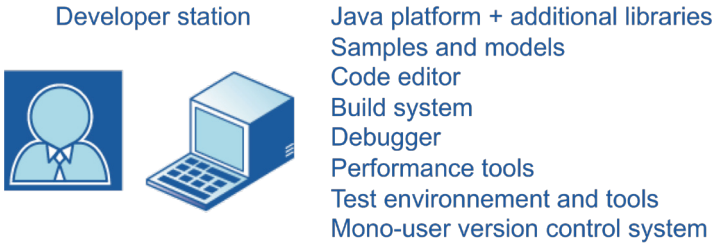



**Figure 3 – Example of a minimal software factory**

Typically, an integrated tool provides several components of the software factory. For instance, we often use Eclipse [16], an integrated development environment that provides a code editor, a debugger and a local build system. Eclipse is one of the leading Java IDEs and benefits from a wide industry support. It can be customized and extended through a number of plug-ins [7] (there are literally hundreds of them), if you have special needs. (See Guideline 9 for further discussion on IDEs.)

Depending on the developer's preferences, a version control system can be used or versions and branches can be managed "manually" without a dedicated tool (e.g. just making copies of the project's files). Plus, Eclipse itself will help by automatically maintaining a history of all the modifications made on each source file on the development station. This can be used for things like returning to a previous state of your code, or visualizing a file's modification history. Note that if the project is going to be actively developed over a long period (several months) or if the developer is likely to be replaced during development (or for further maintenance), you would be better off using a real version control system, such as Subversion. This will help you ensure better traceability of the project's history, which is useful when switching developers or trying to understand code created a long time ago.

**With two developers,** you can still opt for installing a minimal factory on each developer's workstation and have them coordinate together more or less manually. This can be a reasonable option when the developers are in the same room, the project is going to be done over a short period of time (a few weeks) or can be easily split into relatively independent parts. However, a more sophisticated software factory setup (like the one described in the following guideline) will also work well in such context.

**For projects with three or more developers,** you will need to further industrialize your development process in order to be productive. In such cases, you will be using the more extensive software factory setup described in the next guidelines. Note that, in any case, having the whole development team in the same room would be very good for productivity.

At OCTO, we like Subversion, so we use it whenever we can, even for very small projects with one developer. For example, as we are writing this document, it is also on our Subversion repository.

## Key Points

- Projects with only one developer (or one pair) are potentially more productive since no synchronization process is required.
- For such projects, a scaled-down software factory setup will provide optimal productivity.
- Projects with two developers can use a minimal factory or a more advanced one, depending on the developers' preferences and project characteristics (duration and team turnover).

## Guideline 04 – For larger projects, set up a software factory on a shared server

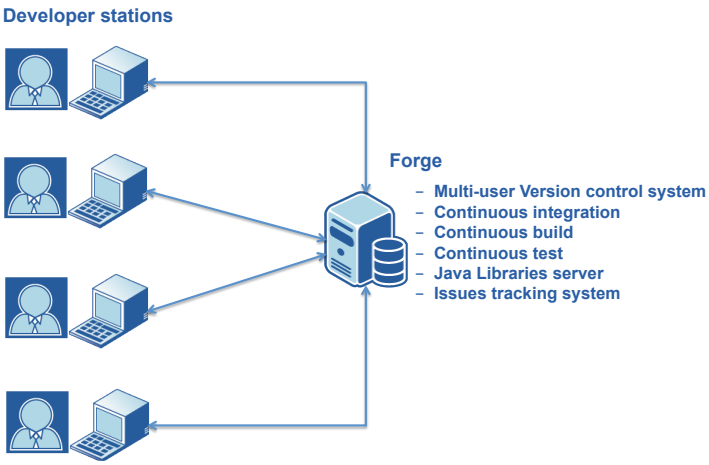



**Figure 4 – Software factory with a shared forge**

During development, each developer workstation should contain:

- A productive set of development tools (code editor, debugger, build system, etc.)
- All the source code and other resources the developer needs to build and test his developments. Typically, it will contain a copy of all of the project's source code and resources.
- All the libraries needed by the developer. This includes the standard Java environment as well as all necessary additional frameworks.
- The infrastructure required for testing the code. For instance, if the code interacts with a database, the development workstation should either contain a test database or a connection to a remote test database. If the code needs to run in an application server, one should be available on the workstation, etc.
- A set of "client" tools to seamlessly interact with the forge

Under this setup, every developer can work independently. When a developer feels that a chunk of code is ready, he puts it in the integration server hosted on the forge. Likewise, when he wants to update his local code base with the latest contributions from his colleagues, he synchronizes his local code repository with the integration server's.

## Tips & Tricks

Once every developer synchronizes code, a very time consuming task can begin: repairing other people's mistakes. There is nothing more frustrating than synchronizing your code, finding that a test doesn't pass anymore, spending time on it and then realizing that you're not involved in that regression. Nothing is more frustrating, except maybe updating non-compiling code because you can't work with non-compiling code. In some teams, we use two simple rules to prevent such situations:

- Anyone committing code that doesn't pass tests has to buy coffee for the whole team.
- Anyone committing code that no longer compiles has to pay for lunch.

While that hasn't solved all the problems, it has solved most of them and made the last ones much more challenging.

Like a development workstation, the forge is a complete environment where the project can be built and tested. The main difference is that the integration server hosted by the forge contains the result of integrating all developer contributions. The various versions and branches of the project's source code and resources are stored and managed by the version control system, which is also part of the forge. For testing, the forge contains controlled test environments that are cleanly isolated and more representative of the future execution contexts than the individual development environments found on each developer's workstation.

Hmm… This looks nice and shiny, but it is sure is a pain to put in place!

Setting up a complete development infrastructure conforming to the best practices in software development can be a daunting task. So, to avoid spending days of manual installation on each development workstation when your team is composed of several developers, you will want to package all the components of your factory and automate installation. First and foremost, you should do that for the «client» side of your factory (i.e. the set of components you want to install on development workstations). Further down the development industrialization path, you might also want to do it for the server-side part of the factory (i.e. the forge), although this isn't as important because your development projects will usually share the same forge (same physical server and software installation).

Various technologies are available to automate factory installation. For example, on Windows, we use NSIS (Nullsoft Scriptable Install System [19]).

Test environments in the forge should be as close as possible to the target deployment environment. This might mean you need to include specific resources in the hosted forge (e.g. a database, a particular ERP system, etc.) or configure the forge with connections to existing external resources (for instance, if your application needs to access an Oracle database you already manage, the forge can be configured to connect to it).

Typically, the forge should also take care of regularly backing up your code and other project resources.

## Key Points

- Developers have to be able to develop concurrently and integrate and test easily on a daily basis.
- For projects with three or more developers, split the factory between developers' stations and a shared server for continuous integration and tests.
- Automate the installation of the "client" side of the factory when repeated installations start to get painful.

## Guideline 05 – Don't waste your time : automate your build process.

From the developer's perspective, building software (i.e. transforming the source code and associated resources into an executable system) can be done locally on his station or remotely on the integration server hosted by the forge. In either case, building is often a complex process, consisting of several steps such as:

- Locating and retrieving source elements that have been modified since the last build, as well as elements that depend on them
- Processing these elements with various tools (preprocessors, translators, compilers, linkers, ...)
- Producing packaged, ready to deploy, executable artifacts (e.g. executable applications)

As you can see, manually managing build chains can be a deadly blow to your productivity. You will want to automate the build process as much as possible. Furthermore, you will benefit from using a build tool that minimizes the amount of information you need to provide to define your build process. Such smart tools can handle dependency relationships between build resources and only ask for information they can not get by themselves (e.g. specifying that a new file is now part of the project, specifying that a new version of the compiler should now be used, etc.).

**Sure, it's easy to say "do this" and "do that", but less so to actually do it. Would you mind explaining how we're supposed to automate this build process?**

The key is to set up a system that knows as much as possible about the different aspects of your project and that can then automatically feed and coordinate your build tools in order to produce executable packages. Since each project is different and uses a variety of tools, this automated build system should be highly extensible to adapt to your context, while maximizing the reuse of common build logic between projects. For example, we often use Maven 2 [8], which is just such a build system. Maven uses the concept of *Project Object Model* (POM), an XML-based description of the various elements of your project, including dependencies. Just provide Maven with your POM, and it will orchestrate builds and create executables.

## Tips & Tricks

Typically, building directly with Maven is done on the integration server only. On development workstations, Eclipse, if used, will do the build using its own build system. Fortunately, there is a way to configure the Eclipse's build system with the information in your Maven POM. The mvn `eclipse:eclipse` command will create or update your Eclipse project from your POM. That way, you will avoid duplicating efforts.

If Eclipse can't perform your local build, that's a symptom of a problem in the build configuration. In such a case, don't try to bypass Eclipse (for instance, by using Maven for your local build). Instead, fix your build configuration and make it work with Eclipse. Remember that Maven should be transparent for developers.

In the next chapters, you will be advised to build and test your components and/or your whole project very often (typically, several times each hour). This is doable only if your build process is very fast. Executing unit tests on the component you are working on should typically take a few seconds and building and testing the whole project should be done within a few minutes. That way, building and testing your code won't disrupt your programming velocity and you will be able to do it as often as you want.

## Key Points

- Industrialize your build process using a smart build system such as Maven 2.
- Ensure that your local build process is really fast.

## Guideline 06 – Manage team development gracefully: use a version control system.

To be productive at coding, you must be freed from as much burden and constraints as possible. In particular, you need to be able to develop in parallel with other team members, sometimes on the same source code. Plus, you need to be able to freely try out different solutions and revert them if necessary.

Another common need is to manage multiple versions of the project. At a given point in your software's lifecycle, you might have one or more versions already deployed and used in various places, by various users. You will have to develop the next version while bugs are being fixed on previous ones. This raises numerous questions. For instance:

- How do you store and keep track of the various versions of your source code and other resources?
- How can multiple developers work concurrently on the same source code in a given version?
- Should you duplicate source files?
- How and when do you merge your modifications?
- How do you even know that concurrent modifications have been made?
- What if some modifications temporarily break another part of the project? Should you fork the source code into multiple branches?
- How can you confidently experiment with various implementation designs while being assured that you will be able to easily get back to a previous state of the project if you want to discard the changes you have made?
- Etc.

These problems happen again and again, so instead of reinventing the wheel you'll appreciate a system that solves most of them for you.

Deployed in part on the forge and in part on the development workstations, a version control system is first and foremost the shared repository for code and other project artifacts. It can ensure integrity of its content by making logically atomic interactions with developers (e.g. checking code in, checking code out). It can manage multiple versions and branches of the project and lets developers view differences in the source code of different versions/branches. It will also detect potential conflicts when multiple developers are modifying the same piece of code at the same time.

These capabilities are fundamental for analyzing the code base and for merging (e.g. integrating modifications made concurrently by multiple team members on same source file). Using file comparison algorithms that go a long way towards automating merging, a smart version control system will help you in this process, minimizing the number of cases where your attention is required.

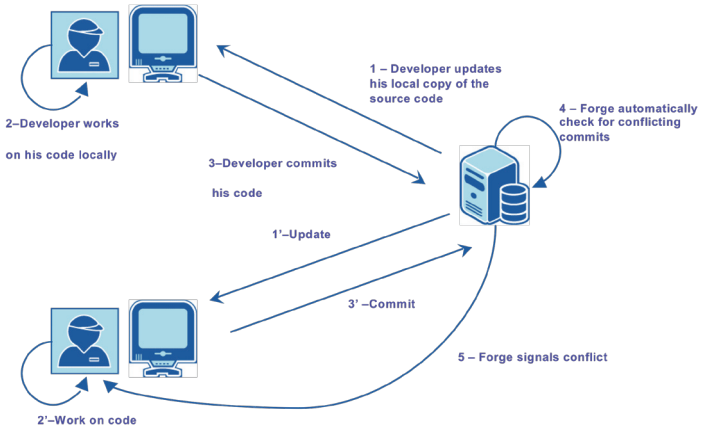



**Figure 5 – Team development using the version control system**

From the developer's point of view, the main steps working with a version control system are:

- **Updating:** getting the project's files on the developer workstation in their most recent state from the shared repository
- **Modifying:** working on the code
- **Committing:** putting modified code back on the shared repository

The version control system checks for potential conflicts when code is committed. A conflict exists when two developers modify the same piece of code, inside the same version of the project at the same time. Since each developer works on a local copy of the code, the conflict is detected at commit, when the code is put back in the shared repository on the forge. The first developer to commit will see his submission automatically accepted. The second developer will be informed of the conflict when trying to commit his code and will need to solve it by merging his modifications with those of the first developer.

**Updating often seems like a good idea, but what if another developer has committed modifications on code I myself have modified locally on my developer workstation? My modifications won't be erased, will they?**

No, they won't. This is just another case of conflicting modifications, this time happening on your local copy of the project. In this kind of situation, the version control system will kick in and inform you of the conflict. As usual, you will be presented the conflicting modifications and given the possibility of merging them. It is much easier to merge code that has not diverged too much. So prefer lots of little updates and/or little commits over less frequent "giant" merges.

## Tips & Tricks

Try to minimize the need to deal with conflicting modifications. To that end, keep your local copy of the code (i.e. the copy that is on your developer workstation) as fresh as possible by updating it from the forge often (e.g. many times a day). That way, you will get the latest modifications committed by other developers and thus work on the most current code base. In the same spirit, commit your code to the forge as soon as you consider it solid enough to share with others (in particular, make sure it passes your local tests correctly). The sooner other developers pick up your code, the less likely the risk of concurrent modifications. In a good software factory, updating or committing should be as simple as clicking a button. Do it early and do it often.

In some projects, we push everybody to act selfishly, we tell each developer to be selfish: "Let others deal with the painful work of resolving conflicts. Just synchronize more often than them". In a short period of time, conflicts become rare.

Having a productive version control system is utterly important because developers interact with it very often. It should be well integrated with the programming environment to make interactions seamless. Our team often uses *Subversion [9]*, a popular open-source, version control system that integrates well with other development tools.

## Tips & Tricks

Powerful graphical tools are available to easily visualize and/or merge conflicting modifications. They present conflicting versions of the code side by side, show differences and let you specify the desired final result. For example, the Eclipse IDE includes such a tool. To use it efficiently in conjunction with Subversion, we have found Subclipse [20] to be a very useful tool. Subclipse is an open source plug-in for Eclipse that provides support for Subversion.

When you commit something, you are given the opportunity to add a comment to your commit. This comment will then appear whenever you or other developers look at the commit history (for example, from Eclipse's history viewer). **Therefore, it is important to always provide a comment when committing.** The comment should be informative enough to let your fellow developers know the purpose of the additions/modifications made in the committed code.

We suggest you ensure that all the files that are needed to produce the deployable artifacts of your project (e.g. your executables) are stored in your version control system. This will make it much easier to automate the various processes (building, testing, deploying, etc.) throughout your project's lifecycle. An exception to this rule applies to existing, compiled Java libraries you are using in your project (e.g. third-party libraries). You will put them in *a Java libraries server*. For example, our team uses *Archiva [10]*, a repository manager that is well suited for such artifacts.

## Key Points

- In addition to managing multiple versions and branches, a version control system lets you synchronize code between developers, automatically detecting and helping to fix any conflicting modification.
- Synchronize your local code repository with the rest of the team often.
- Commit as soon as you can.
- Never commit without comments.
- Use the graphical code merging facilities to quickly resolve conflicting modifications.

## Guideline 07 – Avoid the tunnel effect: set up a continuous integration and test process.

Let's start with a typical story. Faced with a new project, a development team started by decomposing the target application into various modules and assigning a number of them to each developer. The plan was to integrate all the modules together when completed and then to ship the project. But after a few weeks of development, the team realized that the project was turning into a black hole. Each developer was busy working on his own parts but there were no visibility of the state of the project as a whole.

This situation is easy to explain. When a developer thinks a module is ready, how can he make sure it really is? How can he determine whether his module will work when put together with the other parts of the project? How can the team avoid being stuck in the dark until the end of the project, when everyone tries to integrate the various modules, hoping everything works as planned (but knowing from experience that it never does without a lot of suffering)?

To address these problems, the team decided to do things incrementally and planned several intermediate integration deadlines, where the developers would put the various finished parts together to see how they worked as an integrated system. To their dismay, they then realized how time consuming it was to manually integrate and test the project at various steps of its development… Shipping dates slipped, morale went down and the project quickly followed…

Fortunately, such situations can be avoided and productivity restored by setting up a continuous integration and testing process. Continuous integration and testing mean that the forge is configured to automatically integrate, build, execute and test the latest version of the project on a regular basis; for example once or twice a day, or every time a developer commits new code to the forge.

Modules are compiled and series of automated tests are performed. In the end, a report is automatically generated that shows what works and what doesn't, and what errors were encountered during build and test.

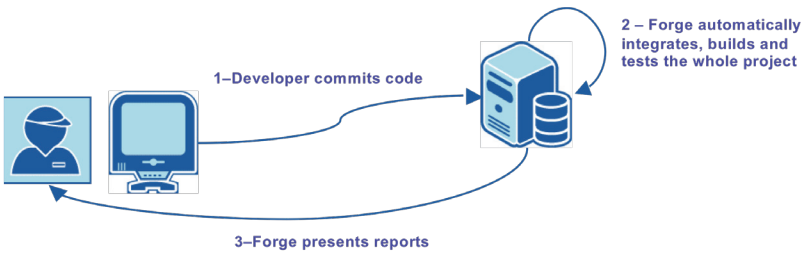



**Figure 6 – Continuous integration and testing**

The reports can also contain any other metrics you might want to automatically extract from the build and test processes. Common examples include:

- Code metrics
- Potential coding error to investigate
- Level of code duplication
- Conformance to coding guidelines
- Test coverage
- Performance metrics ...

### Tips & Tricks

When starting with metrics like coding rules, a common mistake is to add all rules at the same time. This usually means hundreds or thousands of errors. This is very discouraging and looks like a mountain you'll never be able to climb. Another problem with this approach is that really important rules are next to some less important ones.

We prefer a smoother way. We start with only one rule, the most important one for us and explain to everybody why this particular rule is important. Then, reports show where the rule is not applied, and everybody can comply with the rule. Since there is only one rule, the number of violations is usually restricted. When there are no more violations, we add another rule, and when both rules are not violated, we add the next one, and so on.

Continuous Integration is a very handy tool, allowing team members to follow the progress of the project as a whole, after automatic integration of the various pieces that compose it. It lets you identify issues or regression bugs early on and plan for further actions. It exposes problematic interactions between subparts of your project that you thought were independent. It also makes it easy and compelling to show the project's stakeholders the parts of your applications that work in order to get their feedback often, so you can develop in an iterative, incremental manner, in full collaboration with your software's future users.

## Tips & Tricks

So now you have a Continuous Integration tool. That's great, but do you really use Continuous Integration? Tools don't rsolve every problem. Christian Blavier, an Octo Architect, helps answer this question on our Blog (in French):

http://blog.octo.com/index.php/2008/04/30/111-faites-vous-vraiment-de-l-integration-continue

Besides, continuous integration helps a lot in reducing risk on your project. Martin Fowler, one of the historical proponents of this approach, provides great insight on this point :

> *The trouble with deferred integration is that it's very hard to predict how long it will take to do, and worse it's very hard to see how far you are through the process. The result is that you put yourself into a complete blind spot right at one of tensest parts of a project.*
>
> *Continuous Integration completely finesses this problem. There's no long integration, you completely eliminate the blind spot. At all times, you know where you are, what works, what doesn't, the outstanding bugs you have in your system.*
>
> *Continuous Integrations doesn't get rid of bugs, but it does make them dramatically easier to find and remove. In this respect it's rather like self-testing code. If you introduce a bug and detect it quickly it's far easier to get rid of. Since you've only changed a small bit of the system, you don't have far to look. Since that bit of the system is the bit you just worked on, it's fresh in your memory - again making it easier to find the bug. You can also use diff debugging - comparing the current version of the system to an earlier one that didn't have the bug.*
>
> *Bugs are also cumulative. The more bugs you have, the harder it is to remove each one. This is partly because you get bug interactions, where failures show as the result of multiple faults - making each fault harder to find. It's also psychological - people have less energy to find and get rid of bugs when there are many of them.*
>
> *As a result, projects with Continuous Integration tend to have dramatically fewer bugs, both in production and in process. However, I should stress that the degree of this benefit is directly tied to how good your test suite is. You should find that it's not too difficult to build a test suite that makes a noticeable difference. Usually, however, it takes a while before a team really gets to the low level of bugs that they have the potential to reach. Getting there means constantly working on and improving your tests.*
>
> **Martin Fowler, Continuous Integration [11]**

Continuous integration and testing is the cornerstone of a productive software development process. Therefore, a software factory must include the architecture and tools for setting continuous integration and testing in motion. There are a number of tools for configuring and automatically executing such a process. **We found Hudson [12]** to be our tool of choice. Other tools such as **Luntbuild [13]** or **Bamboo [14]** are also worth considering.

Below is a more detailed view of a typical continuous integration process:

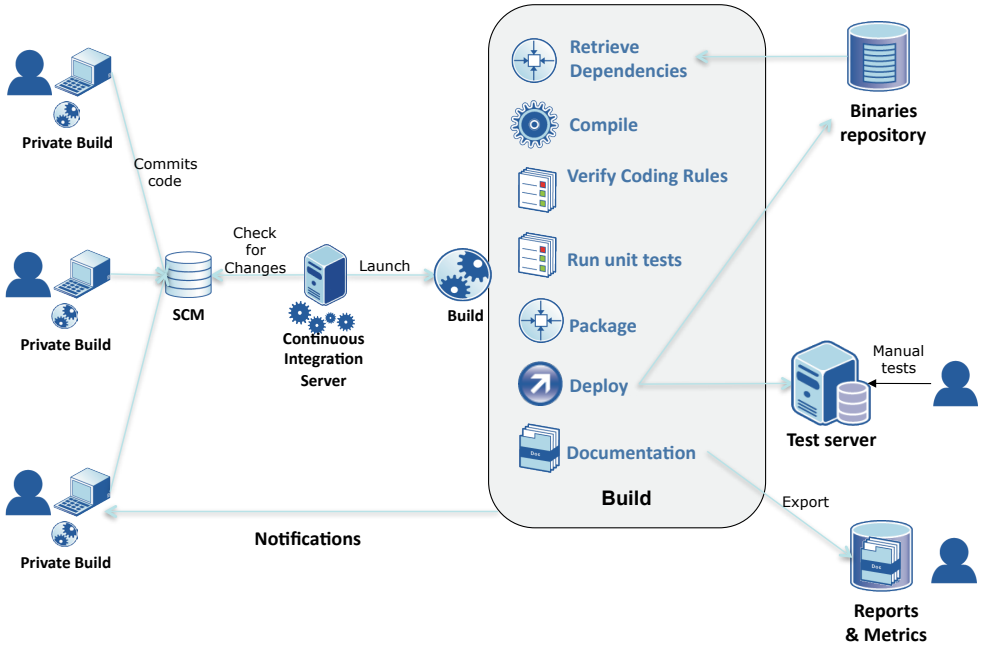



**Figure 7 – Example of a continuous integration system**

**Why are integration tests done on the forge and not on the developer stations (before committing code)?**

This isn't mandatory. Indeed, if you have a complete copy of the project on your station, access to all the resources needed by the integration tests (databases, files, security tokens, etc.), and enough processing power, you should conduct integration tests on your station. That way, you'll only commit code when integration tests are OK and consequently avoid impacting other developers with bugged code. In practice, however, it's often hard to do complete integration tests on developer's stations, because of environment constraints. Besides, even if you can do integration tests on developer stations, you should still have them automatically performed on the forge as well. This will ensure that you get visibility, through tests reports, of the most recent committed version of the project, taking into account any merge that took place at commit time.

There are a number of options available on the market today for creating your forge. We have tested a lot of stuff over the years and here is a set of tools we have found to be efficient in building a state-of-the-art, low-cost forge [2] :

| Name | Type | Main functions |
|---|---|---|
| Subversion | Version control system | Stores source code, manages versioning and merging |
| Archiva | Binaries repository | Stores and manages Java libraries and builds artifacts |
| Maven 2 | Build tool | Builds the project: compiles, packages, launches tests, deploys, etc. |
| Hudson | Continuous integration engine | Orchestrates and schedules the execution of tasks in the factory to carry on the continuous integration and automatic testing process |
| JIRA | Issue tracking system | Manages information related to issues |

[2] Full disclosure: one of the authors is a member of the management committees for the Maven and Archiva project at the Apache Software Foundation.

## Key Points

- Continuous integration and testing take place during development and involve regularly compiling the project components, integrating them and testing the resulting system.
- Continuous integration and testing help project management: they improve visibility of project advancement and help find bugs early.
- Continuous integration and testing are only practicable if automated.

## Guideline 08 – Use an issue tracking system.

This type of tool helps team members coordinate together by signaling issues (bug, feature requests, suggestions, etc.), assigning tasks, transferring them between colleagues and monitoring progress. It will quickly become a central tool of collaboration and organization for your team. It is also efficient to make it available to future users, along with the latest preliminary version of the application you are developing. This is a great tool for getting their feedback and bug reports, in addition to the workshops you may organize with them regularly. Once your project is deployed, the issue tracking system can also be used to track and manage issues raised by your applications' end-users.

As shown in the previous chapter, we usually use *JIRA [15]*, a powerful and productive issue tracking system. JIRA is deployed on the forge and used by members of the project team (developers, business analysts, etc.) through a Web interface.

### Tips & Tricks

When creating a new entry in your issue-tracking tool, use a title that clearly summarizes what is at hand. For instance, say you found a bug in some part of the application and want to make other members of the team aware of it. Here are various titles you could use for the entry.

| | |
|---|---|
| `My first entry!` | **Bad** |
| `Bug` | **Still bad** |
| `Bug in report window` | **Better** |
| `Bug: report rating is inexact in report window` | **Best** |

The last one is clearly more informative and will instantly let your coworkers know what the issue is about. When a development project is in full swing, lots of entries might be created and worked on every day. Providing informative titles will help your team use this tool productively.

### Key Points

- An issue-tracking tool can become the centerpiece of a smooth team collaboration process and can also be used to get feedback from future users.

## Guideline 09 – Use a productive integrated development environment.

An integrated development environment (IDE) makes your workflow intuitive and hassle free by supporting your various programming activities in one tool as much as possible. Our team found Eclipse [16] to be a good IDE. It offers many productivity enhancing features such as a very good code completion system, a number of tools for browsing code and project resources, code refactoring capabilities, and much more.

Eclipse and other state-of-the-art Java IDEs such as NetBeans and IntelliJ IDEA can be intimidating at first glance. Often, developers are not aware of many of the great productivity features hidden inside these tools. This is why we strongly advise you to spend some time exploring these features.

### Key Points

- Eclipse provides many useful tools and shortcuts. It's worth taking some time to discover them. Below are some examples of such productivity enhancers.

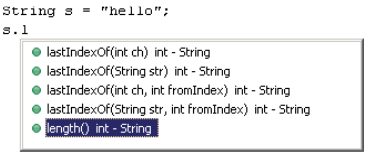
**You said Eclipse is a good tool, but what about IntelliJ? It's good too, maybe even better than Eclipse.**

Well, you're right; IntelliJ is a good tool, as are some others like NetBeans. In fact, we regularly have IDE wars on our mailing lists at Octo. Some like IntelliJ, others love Eclipse and yet others prefer TextMate when coding on their Mac.

All in all, this isn't really important. What is important is the final code, the final application. So you can use another IDE. In fact, you can have developers on the same projects using different IDEs. Just ensure that all tests pass and that your software is built correctly. Tools we discuss help you in this quest.

## Tips & Tricks

The *Content Assist system* is one of Eclipse's most important productivity features. It offers coding assistance by providing code completion suggestions as well as other niceties, like hints about expected types when entering actual method parameters.




It can be configured to kick in automatically while you are typing. You can also summon it by pressing **Ctrl + Space**, then using the arrow keys to navigate in the list of suggestions. In most cases, it is smart enough to determine the type of object you are manipulating in your code and thus to provide you with pertinent suggestions. In addition, to make you code much faster, it can also be used as a learning tool since it lets you easily explore the API provided by the Java objects you are using.

Content Assist can be finely configured to best suit your coding habits. Select the **Windows > Preferences**… menu item, and then open the **Java > Editor > Content Assist** node to change the default configuration.

Note that *Content Assist* is not just for Java code. It is also available for various types of content such as JavaScript, XML, JSP, HTML, etc.

## Guideline 10 – Put tests at the center of your developments

**Wait a minute! Testing is for ensuring software quality, right? But it takes a lot of time to design and implement test suites, even when the actual testing process is done automatically. So, how is that supposed to help with my productivity?**

It's true that testing is commonly associated with software quality (i.e. ensuring that the software works correctly). However, in this document, we will discuss it primarily from the productivity angle. And yes, at first glance, it might seem that, while important for quality, testing hinders productivity. In fact, as modern, agile software development methodologies have demonstrated, adopting the test practices outlined in this section will become one of your best productivity enablers…

Putting tests at the center of development means developing not only the actual components of your application (classes, methods, services, stored procedure, etc.), but also the code that tests these components. In fact, you might say that components without associated automated tests are not complete. When the design of a component you are working on starts to stabilize, take the time to write unit tests that can be automatically executed on your component. Until then, don't consider your component as complete or really functional. You can even choose to write your tests before starting programming the code that is going to be tested (this is called the "Test first" approach).

Your software factory should include frameworks (such as *JUnit [17]*) allowing you to develop your tests along with your application code. In fact, this testing methodology should be integrated throughout your development cycle. You must be able to launch your tests on your development station easily from your IDE, and immediately visualize results.

Moreover, tests should be done automatically on the forge by the continuous integration and test process, providing you with precious reports.

### Tips & Tricks

Your brain never stops working and if you let it switch from the current task, it will be hard to re-focus. Your brain may switch when the current task doesn't need much attention. For instance, after launching a unit test on a component you've just modified, you have to wait for the test to finish in order to know whether the code you're working on passes the test successfully. During this period, you are in danger of losing focus. To avoid that, try to keep the time taken to test the code you are working on to less than 3 seconds.

Your test suites will become one of the most valuable assets of your project, because they can be applied automatically whenever you want, letting you know if your application code works. This means that you can code with confidence. If you do something that introduces a regression in your application, you will know because it will immediately be caught by your extensive automated test suite. Thus, you can quickly and freely modify your code, refactor, improve, iterate – in other words, program – because you now have a safety net. And the more developers or complex code you have, the more this will hold true.

Kent Beck, who has formalized XP (Extreme Programming) principles, brilliantly explains this approach of testing:

> *Programming when you have the tests is more fun than programming when you don't. You code with so much more confidence. You never have to entertain those nagging thoughts of «Well, this is the right thing to do right now, but I wonder what I broke.» Push the button. Run all the tests. If the light turns green, you are ready to go to the next thing with renewed confidence.*
>
> *Programming and testing together is also faster than just programming. I didn't expect this effect when I started, but I certainly noticed it and have heard it reported by lots of other people. You might gain productivity for half an hour by not testing. Once you have gotten used to testing, though, you will quickly notice the difference in productivity. The gain in productivity comes from a reduction in the time spent debugging—you no longer spend an hour looking for a bug, you find it in minutes. Sometimes you just can't get a test to work. Then you likely have a much bigger problem, and you need to step back and make sure your tests are right, or whether the design needs refinement.*
>
> **Kent Beck, Extreme Programming Explained [18]**

This will not only boost your immediate productivity, but also render your software adaptable. Indeed, when a new function has to be implemented, or any other technical changes have to be made, you'll be able to do it instead of saying "Sorry, this software is too brittle. If we touch it somewhere, we might break it elsewhere, and we'd have no way of knowing".

> *Tests keep the program alive longer (if the tests are run and maintained). When you have the tests, you can make more changes longer than you can without the tests. If you keep writing the tests, your confidence in the system increases over time.*
>
> **Kent Beck, Extreme Programming Explained [18]**

## Key Points

- Testing is not just about software quality. It is also one of the most effective productivity enablers, both short-term and long-term.
- You should consider testing an integral part of your day-to-day development activity.
- Systematically implement automated unit tests for your software components.
- Try to keep unit tests very fast. Most of them should take less than a second to run.
- Consider automating higher-level test phases (e.g. functional testing) and running them as part of the test process done by the forge.

## Guideline 11 – Deploy with ease: externalize dependencies to external resources.

During its lifetime, your application will run in several environments (e.g. development station, integration server, homologation, various deployment environment, etc.). Obviously, if your code or other files in your application package (e.g. your *.war*) contain hardcoded references to external, environment-dependent resources, you'll have to modify and rebuild it each time you want to deploy on a different environment, or to maintain multiple versions in parallel. For instance, if the modification occurs in your source code, you'll have to go through a complete build to regenerate an updated application package. If the modification is made in a configuration file inside your application package, you might be able to do it without going through a complete build cycle, but you'll still have to open your package and carefully modify its content.

In addition to being detrimental to your productivity, this also defeats somewhat the whole purpose of having homologation and UAT (User Acceptance Testing) environments. Indeed, to minimize risks of accidental regressions, the application you deploy in production should be as close as possible to the tested and validated application. In some organizations, this policy is enforced by making sure the application package that gets the green light from UAT is not modified before going in production (for instance, this can be done using MD5 checksums).

Moreover, depending on the environment, the need to modify the content of the application package can create organizational challenges. Indeed, knowledge of build tools is usually restricted to the development team whereas knowledge of the production environment (file system layout, resources locations, passwords etc.) is usually restricted to the production team. Consequently, the two teams would have to work together to deploy the application.

There is another problem with hard coding: external resources used by your application tend to change over time. For instance, a database password may change, or a web service may migrate from one server to another, using another URL as its endpoint. If the password or the web service URL are in your application package, maybe buried somewhere in your code, you will have to modify the application, rebuild it and redeploy it. On the contrary, if such external references are defined outside your application package, **you can modify them without touching it.** Of course, there must be a way to make your application aware of the modified configuration. Indeed, JNDI, a standard extension to the Java platform, provides several mechanisms to specify environment dependent parameters to your application at launch time. One possibility is to set environment-dependent values in **the application server configuration file.** For example, the following XML fragment in a Tomcat configuration defines a connection chain for a database:

```
<?xml version='1.0' encoding='utf-8'?>
<Context displayName=»My Web Application» docBase=»/usr/local/myWebApp/
myWebApp-1.0.war» path=»/myWebApp»>
  <ResourceParams name=»jdbc/myDataSource»>
    <parameter>
      <name>url</name>
      <value>jdbc:oracle:thin:@myDBServer:MyTNSNAME</value>
    </parameter>
....
  </ResourceParams>
</Context>
```

**This XML fragment is outside of the application package (i.e. outside of the .war)**

In the application, you can look up this connection chain using the JNDI API, or, alternatively, you can ask the Spring framework to automatically inject the connection chain inside one of your objects, as shown in the example below:

```
<bean id=»DataSource» class=»org.springframework.jndi.JndiObjectFactory-
Bean»>
    <property name=»jndiName» value=»java:/comp/env/jdbc/myDataSource»/>
</bean>
```

**This XML fragment is outside of the application package (i.e. outside of the .war)**

As shown in this example, you should not hardcode the connection chain in the application package. Instead, you should reference it using a logical name (`java:/comp/env/jdbc/wspds`). The actual value of the database connection chain (`jdbc:oracle:thin:@16.40.64.62:1521:SID`) is defined outside of the application package, by providing it to the application server. This can be in a file, as in the example, or using the application server administration console, or by other means, depending on your application server.

## Tips & Tricks

This approach is not restricted to environment-dependent values or references to external resources. It can be used for any technical parameter that you identify as being likely to change during the application's lifetime. Decoupling your executable from such parameters will often let you avoid costly development/validation/deployment cycles.

## Key Points

- Don't hardcode environment-dependent values inside your application package.
- Instead, use an indirection layer and look them up at run-time through a logical name.
- Define the binding between the logical name and the actual value outside your application package.

## Guideline 12 – Be productivity aware

A good practice for improving productivity is to deal with it explicitly. Make it a subject of reflection and discussion within, and between, project teams, instead of something implicit that is never directly addressed in its own right. Setting up team discussions about productivity will put the focus on the subject and help you quickly identify and address issues. Furthermore, it will build a collective intelligence by promoting discovery, discussion and exchange of productive practices.

Here are a few operational guidelines we have found valuable in achieving this goal:

- **Assign time** at your team meetings to discuss productivity explicitly (issues, opportunities, practices, etc.).
- **Exchange productivity tips with your colleagues:** useful keyboard shortcuts, code refactoring practices, new features in development tools, etc. Building a culture around productivity is rewarding, both in terms of achieving project goals and developing highly valuable skills.
- **Don't accept discomfort.** Just don't. When you feel that something in your development workflow hinders your productivity (should it be a tedious, repetitive task, an annoying delay, or anything that makes your development painful), don't accept it. Instead, fix the problem and get back to a productive and enjoyable workflow… Because you're worth it! If you need help, contact us and we'll work with you to identify the problem and solve it.Conformance to coding guidelines
- **Adapt development practices and technical recommendations to your team's skills and culture.** A perfect, ideal, state-of-the-art environment that the team wouldn't be able to benefit from or to master would be detrimental. For instance, while we generally recommend using an object-relational mapping system when developing interactions with a database, it requires a certain level of experience that might not match your team's skills or culture. Using the JDBC API and gradually moving to Hibernate might be better for you, or you might prefer to use another object-relational mapping system you already know well, or go for an object-oriented database, etc. Also, keep in mind that some technologies and practices won't make sense to your team up front but will need to be introduced gradually.
- **Develop a deep understanding of the technology you are using and the associated programming techniques.** Don't blindly reproduce code and patterns you have been shown or have access to: doing so will come back to haunt you. Every line of code you produce, every Java instruction should make sense to you, be justified, and be fully under your control. Without such an understanding of your own production, you won't be able to debug or maintain your own code.

## Key Points

- Make productivity an explicit matter.
- Create a culture of productivity.
- Don't accept discomfort.
- Adapt practices to your skills and culture.
- Develop a deep understanding and command of the technology.

# References


[1] **Java Enterprise Edition v. 5**
http://java.sun.com/javaee/technologies/javaee5.jsp

[2] **The Java Language Specification, Third Edition**
http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html

[3] **Java 2 Platform, Enterprise Edition v. 1.4**
http://java.sun.com/j2ee/1.4/

[4] **Spring Framework**
http://www.springframework.org

[5] **Hibernate – Relational Persistence for Java and .NET**
http://www.hibernate.org

[6] **ITEC Testing Standard & Methodology**
http://secnet.fr.world.socgen/qualitec/index.php?id=202

[7] **Eclipse Plugin Central**
http://www.eclipseplugincentral.com

[8] **Maven – A software project management and comprehension tool**
http://maven.apache.org

[9] **Subversion – An open source version control system**
http://subversion.tigris.org

[10] **Archiva, the build artifact repository manager**
http://maven.apache.org/archiva/

[11] **Continuous Integration**
Martin Fowler, http://martinfowler.com/articles/continuousIntegration.html

[12] **Hudson : extensible continuous integration engine**
http://hudson.dev.java.net

[13] **LuntBuild – Automate and manage your builds**
http://luntbuild.javaforge.com

[14] **Bamboo, Continuous Integration Server**
http://www.atlassian.com/software/bamboo/

[15] **JIRA: Bug Tracking, Issue Tracking, & Project Management**
http://www.atlassian.com/software/jira/

[16] **Eclipse – An open development platform**
http://www.eclipse.org

[17] **JUnit**
http://www.junit.org

[18] **Extreme Programming Explained: Embrace Change**
Kent Beck and Cynthia Andres, Addison-Wesley Professional.

[19] **NSIS (Nullsoft Scriptable Install System)**
http://nsis.sourceforge.net

[20] **Subclipse**
http://subclipse.tigris.org

[21] **Java SE 6 Performance White Paper**
http://java.sun.com/performance/reference/whitepapers/6_performance.html

[22] **JDBC 4.0 Specification**
http://jcp.org/aboutJava/communityprocess/final/jsr221/index.html

[23] **Spring Security (Acegi)**
http://static.springframework.org/spring-security/site/index.html

[24] **Spring MVC**
http://static.springframework.org/spring/docs/2.0.x/reference/mvc.html

[25] **Grails**
http://grails.codehaus.org

[26] **Restlet**
http://www.restlet.org

[27] **Services Web : Choix Architecturaux**
OCTO Technology. http://www.octo.com/com/download/cahier_servicesweb.pdf

# About OCTO Technology

Founded in 1998, OCTO is a French-based consultant company focusing on information systems' architecture and agile methodologies for software development. OCTO has established a pioneering approach in advanced technology and software development, and for every project provides workable solutions responding to our clients' business needs. Communication, respect and knowledge sharing are key values of the company, shared and transmitted by its co-founders, and visible throughout our many publications. OCTO is continuously building a community of excellence in Agile Software Development, Enterprise Integration and Security, Test Driven Development, Java, .NET, and Open Source. Our customers benefit from the collective experience gained through the various projects our architects have worked on.

## About the Authors

The authors of this white paper are software architects at OCTO Technology : Philippe Mougin, Guillaume Duquesnay, Arnaud Héritier et Benoît Lafontaine.

Special thanks to Olivier Malassi, Laurent Brisse, Damien Joguet and Nelly Grellier for all their support during the redaction of this document.

Thanks to Roxana Gharagozlou for the visual quality of the finish product.

For more information about OCTO technology, visit www.octo.com

or contact publications@octo.com.

## Acknowledgments

Special thanks to Société Générale Corporate & Investment Banking, and more particularly to M. Tesici and M. Cosenza for authorizing the publication of this white book whose contents are inspired by a mission at Société Générale Corporate & Investment Banking in 2007.

# Also by OCTO Technology

In our collection: « Une Politique pour le Système d'Information »


Une Politique pour le Système d'Information

**Descartes - Wittgenstein - (XML)**


Une Politique pour le Système d'Information

**Gestion des identités**

OCTO Technology is continuously publishing its knowledge and approach in white paper :

- Services Web : Choix Architecturaux (2007)
- Architecture Orientée Services (SOA), Une politique de l'interopérabilité (2005)
- Architecture de Systèmes d'Information - Gouvernance de la Donnée (2004)
- Architecture d'Applications - la solution .NET (2003)
- Architectures de Systèmes d'Information (2002)
- Le Livre Blanc de la Sécurité (2001)
- IRM, Gestion de la Relation Client sur internet (2000)
- EAI, Intégration des Applications d'Entreprise (1999)
- Les Serveurs d'Applications (1999)

OCTO Technology is also the author of three IT books published by « Editions Eyrolle » :


« Le projet e-CRM - Relation client et Internet » (2002)


« Les Serveurs d'Applications » (2000)


« Intégration d'Applications - l'EAI au cœur du e-business » (2000)


OCTO Technology - 50, Avenue des Champs-Elysées - 75008 PARIS
Tél : [33] 1 58 56 10 00 - Fax : [33] 1 58 56 10 01 - info@octo.com - www.octo.com