METHODS & TOOLS

Global knowledge source for software development professionals **ISSN 1023-4918**

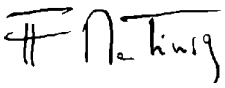
Summer 2004 (Volume 12 - number 2)

What Motivate a Developer? Besides Money... ;-)

In a recent issue, I wrote an editorial about the price of a developer. The implicit thesis was that the money that you earn developing software is a sign of appreciation that represents the value of your work. This money is an important motivation factor. In this issue, there is an article of Stéphane Croisier about collaborative source software. The point developed in this article is that this is not fair that people can benefit from open source software without giving a monetary contribution to the developers. I think that this topic that could lead to an interesting debate on the Methods & Tools Forum. (www.methodsandtools.com/forum/index.html). More interesting is the question: why so many developers participate to open source projects if there is no money to be made?

Doing some research on the Web, I found results of surveys targeting the open source development community. What can be found in these documents is that the most important motivations are linked to the community concept: being part of a community or giving back something after using open source software. A second important group of motivating factors is that this is an opportunity to learn and develop new skills. There is also an ideological conscience of people belonging to the open source development community. They are proud to produce free software, but this does not provide the greatest source of motivation.

I think that these findings provide a clear message for software development managers. Create a working environment where your developers will be proud to be part of the team and put them in a situation where they can develop their skills and learn something new. You will create conditions to improve your development productivity... and developers will not have to look for open source projects to be motivated. This seems a simple thing, even if it is not always obvious to create such environment. This is perhaps why these motivating work conditions are not widely present in software development departments around the world.



Inside

Parallel Development Strategies for Software Configuration Management

Tom Bret, Confluence Systems Ltd
in association with MKS Inc (www.mks.com)

Abstract

Software project managers routinely face the challenge of developing parallel configurations of software assets. If not identified and planned for in advance, the complexity introduced by parallel development can derail even an otherwise well-managed project.

This article describes business situations where parallel development is necessary and examines strategies for configuration management in each situation. "Patterns" are identified which allow different projects with similar characteristics to be managed in a repeatable way, and make it possible to carry forward a body of knowledge and experience from one project to another. Patterns may be combined or nested to deal with complex scenarios. Parallel development potentially needs to be managed at multiple levels within a software asset repository, recognising that different products and components may have their own independent development lifecycles.

What is Parallel Development?

Many software projects, especially in their early stages, follow a strictly linear development progression in which each successive version of the software is derived from, and increments, the previous version. The configuration management of such a project is straightforward, since in general there is just one "latest and greatest" version of the software, which forms the context in which all development work takes place.

Parallel development occurs when there is a need for separate development paths to diverge from a common starting point, so that there is no longer a single "latest and greatest" version, but instead two or more concurrent "latest" configurations (often called ***variants***) where new development is carried on. Also implicit is the potential need for the divergent development paths to converge again. This means that any strategy for development branching should also take into account the process for merging.

Why Parallel Development?

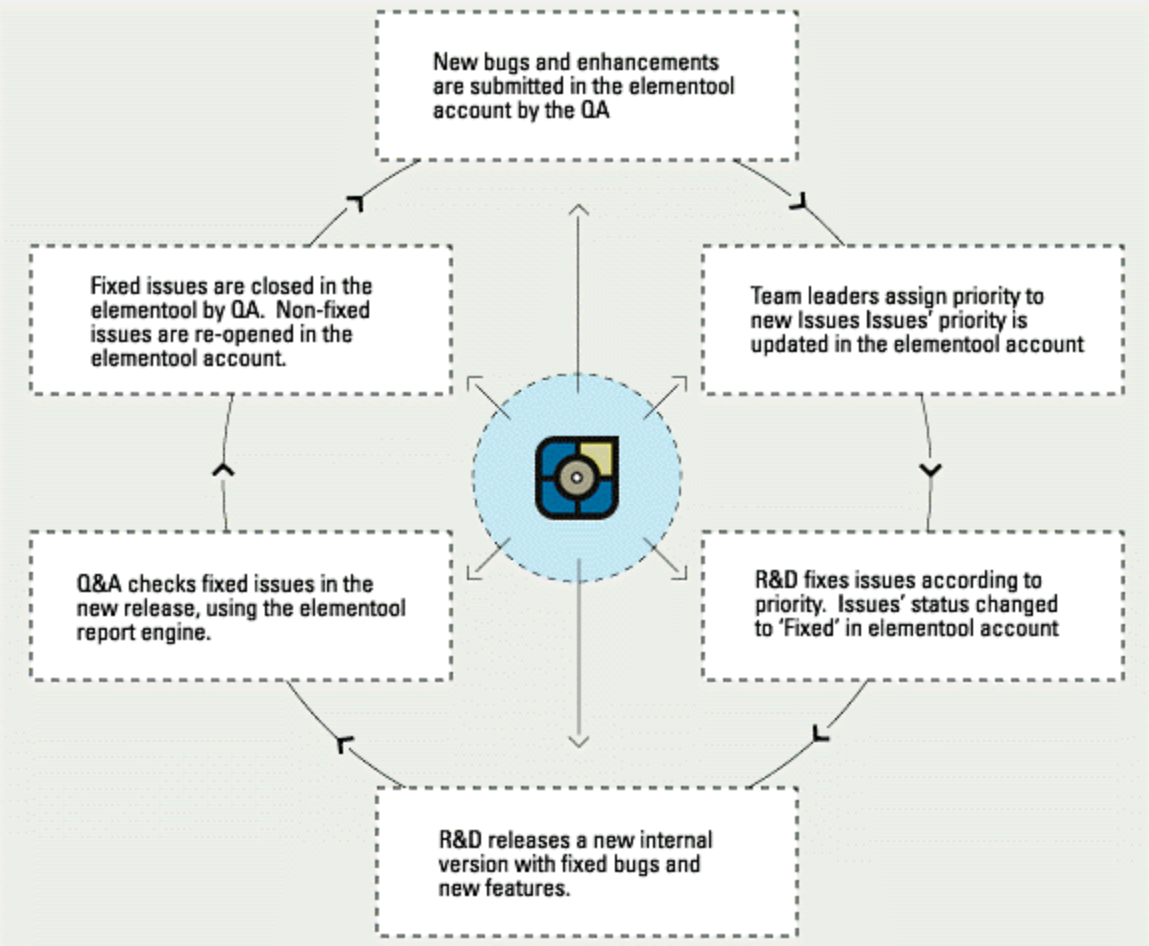
There are many reasons why parallel development may be necessary for a project. These include:

- Release preparation
- Post-release maintenance (segregated from new development)
- Tailored or customer-specific software
- Segregation of work by different development teams or individuals
- Segregation of work on different features
- Deployment of different software variants into different environments

Effective Bug Tracking

Bugs are part of every product development process. How do you track the bugs you find during product development and after? Bugs that are found but not properly tracked might slip away and be discovered by your customers. Elementool's bug-tracking tool enables you to track and save your bugs so nothing gets lost. Elementool enables you to track new bugs, prioritize and assign bugs to your team members, generate bug reports, customizing the account according to your special needs and more.

www.elementool.com/?m



Join for free: www.elementool.com/?m

Macro and Micro Parallelism

Parallel development can occur at the level of a single file or other configuration item (micro level) or at the level of an overall project configuration (macro level).

Consider the case cited above regarding "segregation of work by different development teams or individuals". If different developers need to make changes to the same configuration item for different purposes, this can often be handled by short-term branching at the micro level. The second and subsequent developers to complete their work would be responsible for merging to maintain the integrity of all changes to that configuration item, but still within a single overall development context.

In the case of changes involving more than a small number of configuration items and/or timescales longer than a few hours, parallelism at the micro level would rapidly become unworkable due to the large number of ad-hoc merges; teams or individuals waiting for completion of each other's merges; and confusion over the merge status and correct "latest" version of each configuration item. In this situation it makes much more sense for each team to work with its own different "latest" configuration of an entire project (macro-level parallelism). Behind the scenes, branching of individual configuration items still takes place, but a good software configuration management (SCM) tool should handle this (more or less) transparently to the user. Merging would be deferred until a suitable development milestone, when a batch merge of all the desired changes may be performed to result in a single "latest and greatest" version once more, forming the basis for the next development cycle.

Brad Appleton [1] refers to this as "file-oriented" and "project-oriented" branching, and summarises as follows:

> "Most VC [version control] tools supporting branches do so at the granularity of a lone file or element. … This is called *file-oriented branching*."
>
> "But branching is most conceptually powerful when viewed from a project-wide or system-wide perspective; the resultant version tree reflects the evolution of an entire project or system. We call this *project-oriented branching*. "

Fundamental for the capability of a tool to support macro-level branching is the capability to save and reproduce successive versions of the entire project configuration on any branch, known as *baselines* or *checkpoints*.

The remainder of this article will deal with macro-level branching and the business situations to which it can be applied, using a modern SCM tool with good support for macro or project-oriented branching.

Frequently Experienced Problems

The assertion, made earlier, that "the complexity introduced by parallel development can derail even an otherwise well-managed project" is backed up by anecdotal evidence of problems experienced on software projects where branching and merging has been poorly planned or poorly controlled.

- Project granularity: failure to allow for a separate lifecycle of sub-components *Product released with immature component code, or component changes interfere with product release cycle*

- Failure to segregate maintenance fixes from new feature development
  New features "leak" into production code when not fully QA'd and/or not paid for by customers
- Failure to segregate custom code from base product
  Custom features accidentally released when not paid for and/or in violation of confidentiality
- Over-complex branching model
  Excessive amount of effort spent in merging and maintaining branches; developer confusion about which branch they should work on
- Failure to track compatible sets of changes and apply them consistently during merge
  Broken build or non-working features after merge
- Failure to anticipate the need for parallel development and to plan early
  Inappropriate tool selection and/or repository design becomes "locked in" in the early stages of the project and only becomes a problem when it is too late to remedy without excessive cost, effort and/or delay

Careful and appropriate design of the branching model is critical to success – as expressed by Mario Moreira [2]:

> "A branch and merge strategy should be ***designed*** prior to creating branches to first ensure that branching is needed and secondly to ensure that the branch structure under consideration will work for the project."

Tool selection can also be critical, since development teams may be forced into one or more of the above problem situations by constraints of the configuration management tools they are using. Tool selection is typically done early in the development project lifecycle, and it is important to think ahead, since branching requirements may typically only arise later in the lifecycle: for example, once post-release maintenance comes into play after the first production release.

Introducing SCM "Patterns"

A key factor to help achieve a successful design is the use of ***patterns***. A pattern is an established engineering solution for a certain class of problems: for example, in civil engineering, the "river crossing" class of problems may lend itself to solution patterns such as "bridge", "ferry" or "tunnel". The engineer will be guided to a choice of pattern by parameters such as width, depth and flow speed of the river, type and density of the expected traffic, and so on. In both choosing a pattern and subsequent execution of the solution, the engineer can draw on a body of prior experience with these patterns.

Most readers will be familiar with the well-established concept of patterns in software engineering. The use of patterns is becoming established in Software Configuration Management [3] though less mature than in software systems analysis and design, it is still a very useful and instructive approach.

We now discuss various patterns that can be applied in different situations where parallel development occurs. It is worth noting that these patterns may – indeed should, be adapted and/or combined (with each other or with other SCM patterns) to achieve the best solution fit with a particular set of development project constraints and requirements.

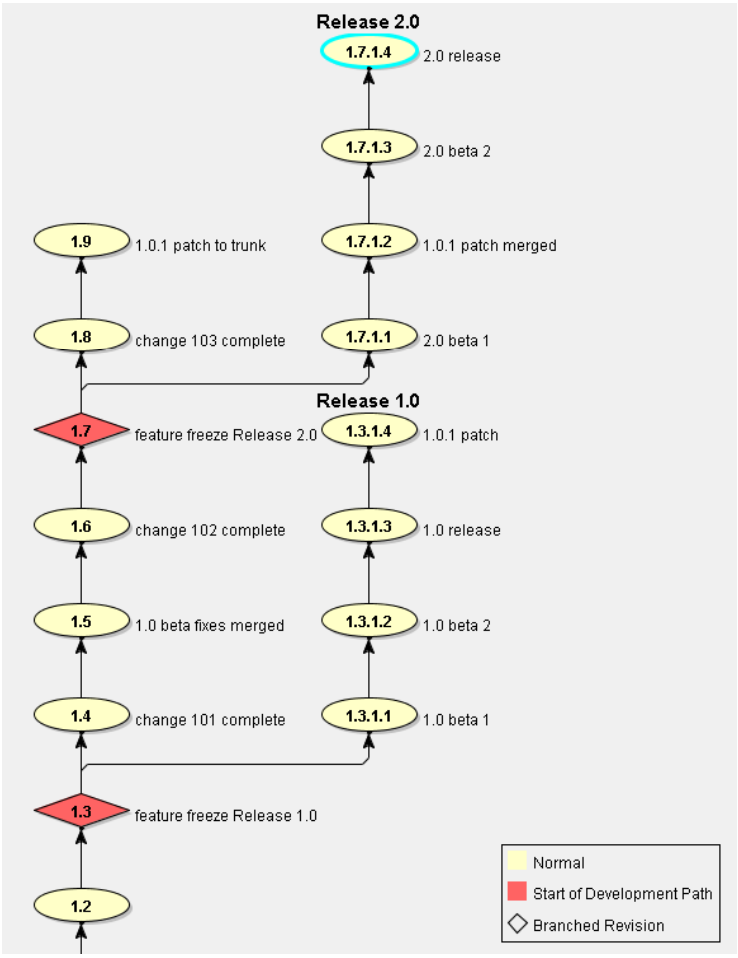
The treatment of patterns in this article is somewhat simplified compared with Berczuk & Appleton [3], [4]; where appropriate, cross-references are given to the Berczuk & Appleton nomenclature [4].

Pattern: Branch per Release

[Berczuk & Appleton: *Release Prep Codeline*]

Probably the most frequently encountered motivation for parallel development, mentioned several times already in this discussion, is the need to segregate post-release maintenance from ongoing development targeted on future releases. Taking this a logical step further, we could (or indeed should) segregate the Pre-release fixes in the same way. The optimum point at which to diverge a release configuration from the ongoing "mainline" of development would be at the point of "feature freeze", when the decision is taken not to include any more features in the release being planned. From this point onward it becomes important that no further feature-enhancement code leaks into the release and that only changes necessary to fix existing features are applied to that configuration.

A convenient simplification is to continue post-release maintenance on the same development path as the release preparation (rather than using a separate release branch) – the successive release candidates, releases and patch releases simply being incremental checkpoints along the same development path.



In the "project history" diagram above, we see that all the main feature changes are implemented on the project "trunk". (This corresponds to the Berczuk & Appleton *Mainline* pattern.) A *checkpoint* has been taken on completion of each change and labelled with the change number (change 101, 102, 103).

In parallel with this, fixes during release preparation and post-release maintenance have been made on release branches that diverge from the trunk at each feature freeze checkpoint. Each checkpoint on the branches would correspond to a release candidate, beta or patch build. (The illustration has been simplified and in real life there would probably be many more checkpoints, fixes and changes!)

A criticism sometimes levelled at this pattern is that fixes need to be made in more than one place: any fixes made on a release branch are likely also to be needed on the trunk (mainline), and possibly on other release branches if multiple releases are active. This is true but is mitigated by the following factors:

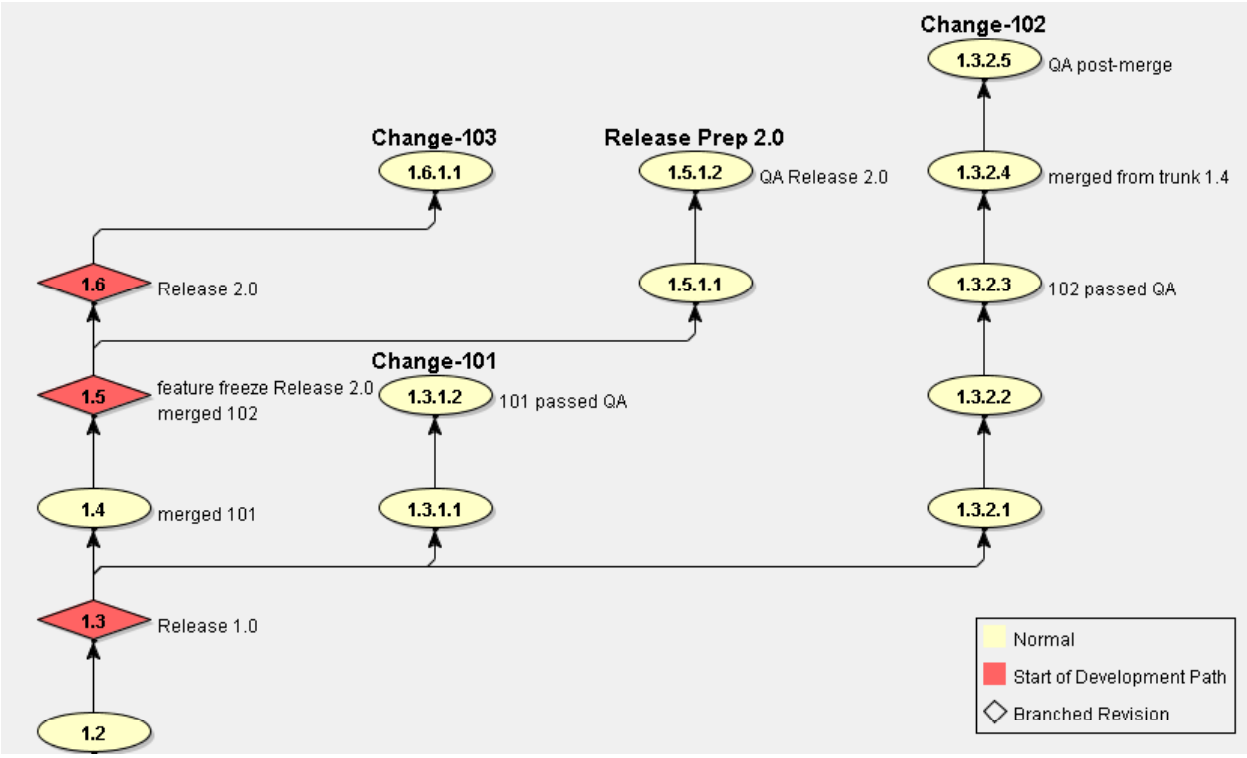
- The fixes which need to be merged will generally be relatively small code changes and therefore easy to merge
- The (generally much larger) feature changes do not need to be merged at all
- In practice, some of the fixes will have been rendered obsolete by other changes on the trunk and will not be required

Usually it is better (both for quality and productivity) to design for smaller merges, since merging can be a time-consuming process and bears the risk of introducing errors.

Pattern: Branch per Change

[Berczuk & Appleton: *Task Branch*]

The motivation for using a "Branch per Change" pattern has already been discussed above under "Macro and Micro Parallelism". To summarise, when different development teams are engaged in large and complex changes that may involve conflicting changes to the same configuration items, it may be desirable to isolate them from each other using different development paths. This means they are each able to check-in interim versions of their code without adversely affecting each other (as might occur if they were both using the same "mainline"). Merging back to the trunk, or mainline, is deferred until each change is complete and the merge is "batched", meaning that all the related code changes are merged at the same time so that the mainline remains consistent. A good SCM tool should have the capability to track which code changes are related to each other and also to initiate the batch merge of those changes.



This time in the project history diagram we see, diverging from the "Release 1.0" checkpoint on the trunk, two separate branches for "Change-101" and "Change-102". On completion of each change, the code is merged back to the trunk, resulting in a new checkpoint ("merged 101" etc). After feature freeze and use of a release preparation branch, the cycle restarts from "Release 2.0" and "Change-103".

The main difficulty with using this pattern is that the amount of code to be merged is much larger than with "Branch per Release"; this is because each merge involves the entire code for some new feature, rather than just a bug fix where the code changes are often quite small.

Also, if this pattern is rigidly applied – for example, by restricting developer access on the trunk so that developers are forced to work on branches – then it imposes a large procedural overhead on even a very small and simple change. This could have an adverse effect on productivity (due to the disproportionate amount of time spent performing merges and verifying the merge results) as well as causing a proliferation of branches that could lead to confusion. Arguably, for most

development teams, it would be preferable to create task branches only for changes of long duration or which are known to conflict with other changes already in progress, and to allow simpler changes to be done on a single shared "mainline".

Anti-Pattern: Good Code on Trunk

If a pattern describes a known engineering solution to be emulated, an "anti-pattern" could be defined as describing a known engineering pitfall to be avoided! Consider the – superficially quite convincing – concept of only allowing "good code" (i.e. passed by some QA process) on the project trunk. This is often found in conjunction with the "Branch per Change" pattern and in fact is illustrated in the "Branch per Change" diagram, above.

The problems here are:

- The excessive merge workload and the effect this has on developer productivity;
- Over-restrictive process interfering with progress of the development team;
- Additional QA overhead; and
- The fact that merge outputs are generally not reliable anyway, certainly requiring verification and often necessitating rework.

Looking again at the diagram, we see that Change-101 has been completed first and may be merged straightforwardly back to the trunk. Change-102, when completed, cannot be merged without *either* losing the Change-101 updates *or* introducing non-validated merge results onto the trunk. To comply with the "Good Code on Trunk" principle, the only option is first to merge the Change-101 code to the Change-102 branch, repeat the QA process (verifying that both the Change-101 and Change-102 features still work) and then to perform a second merge onto the trunk. Worse still, no other changes may be merged to the trunk while this second merge is pending, because this would invalidate the QA for Change-102, so the project progress could be held up for possibly several days while this is ongoing. If we imagine a large project with say fifteen parallel change branches instead of two, we can see that the extra effort and complexity would become overwhelming.

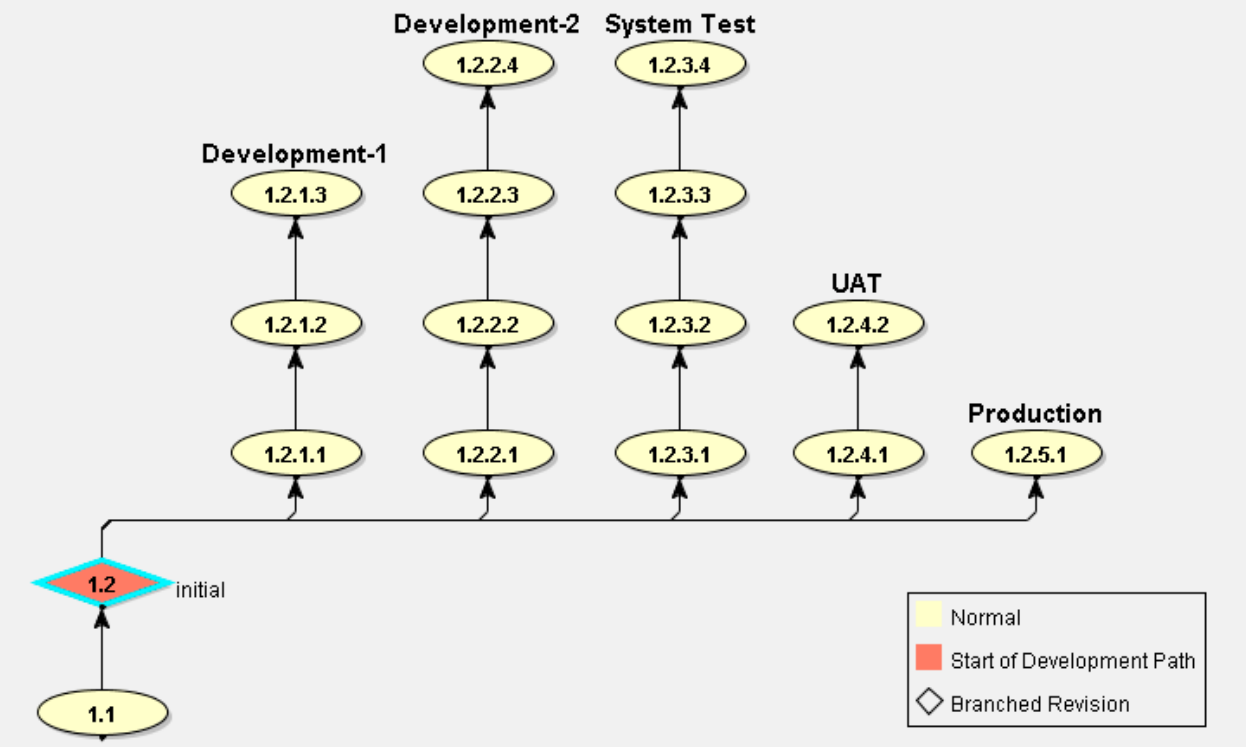
Pattern: Branch per Environment

[No direct equivalent in Berczuk & Appleton]

A situation often encountered, especially with software systems being developed for "in-house" use, is the requirement to migrate software into successive different environments. Typically this starts with a development environment (sometimes several different ones, allowing for segregation between work by different teams) and moves on into system test, UAT (user acceptance test) and production environments.

By modelling each environment with a parallel "development" branch in the software repository, it is possible to harness the CM tool to assist with:

- Packaging and deployment of migrations between environments;
- Status accounting of current configuration in each environment;
- Access control to different environments; and
- Traceability of changes as they move through the environments (for instance, so that testers which fixes and features are available for test).



Composite and Nested Patterns

The patterns presented here are abstractions and simplifications of situations encountered in real-life software development projects, and represent only a fraction of the possible patterns that could be identified. When designing a configuration management implementation, it is important to maintain flexibility to match the needs of the project, to use patterns for guidance but not to be constrained by them, and to adapt or combine them as appropriate. The following are some examples of how this might be approached.

Branch per Release nested at component level. A code repository is organised as a number of products each made up from some combination of components. Products and components have their own independent release cycles and each is controlled using a Branch per Release pattern. Different products (or successive releases of the same product) may utilise different releases of the same component.

Branch per Release adapted for tailored customer code. A similar structure to the above could be used to manage the delivery of tailored versions of a core product. In this case the core product would be treated as a component (or set of components), and customer-specific code would also be segregated into components. A "container" project for each customer would not hold any actual code, but just a set of pointers to the correct versions of the core and custom components. (This depends on the use of an SCM tool with appropriate support for component inclusion or "sharing".) The design objective here should be to avoid a proliferation of customer-specific branches within the actual components themselves.

Composite of Branch per Release with Branch per Environment. The development of a software system follows a Branch per Release pattern and each release is deployed into successive environments using a Branch per Environment pattern. Each release branch spawns its own sub-tree of environment branches.

Composite of Branch per Environment with Branch per Change. A Branch per Environment pattern is used, but additionally a Branch per Change pattern is superimposed on the development environment. Development is not a single branch, but a sub-tree of change branches.

Conclusions

It is hoped that this article has encouraged software project managers and SCM practitioners to take a fresh look at the issues surrounding parallel development, and the engineering solutions by which they can be addressed. Careful analysis of the project requirements at an early stage, coupled with appropriate tool selection and a disciplined development process, can avoid many of the difficulties which have traditionally beset branching and merging within software repositories.

To conclude - some key points to keep in mind:

- Plan early
- Use patterns
- Get the right tools
- Merge little & often
- Batch merges of related changes
- Avoid excessive complexity
- Be creative!

References

1. Brad Appleton (Motorola Network Solutions Group) et al
   Streamed Lines: Branching Patterns for Parallel Software Development
   www.cmcrossroads.com/bradapp/acme/branching/#StreamedLines

2. Mario Moreira (Fidelity Investments Systems Company)
   ABCs of a Branching and Merging Strategy
   CM Crossroads News, November 2003
   www.cmcrossroads.com/newsletter/articles/mmnov03.pdf

3. Steve Berczuk with Brad Appleton
   Software Configuration Management Patterns: Effective Teamwork, Practical Integration
   Addison-Wesley, 2002

4. Steve Berczuk with Brad Appleton
   Software Configuration Management Patterns Reference Card
   www.scmpatterns.com/book/refcard.html

Architecting Integration - Enterprise Integration Patterns

Mehran Nikoo, info@dthomas.co.uk
Dunstan Thomas Consulting http://consulting.dthomas.co.uk.

Architecting integration solutions is a complex task because there are so many conflicting drivers and different possible solutions. Whether the architecture was in fact a good choice is usually not known until many months or even years later, when unavoidable changes put the original architecture to the test. Unfortunately, there is no set solution for enterprise integration solutions. The following article looks at enterprise integration patterns and the various challenges facing those who take up the architecting challenge.

Enterprise Integration Patterns

Over time, enterprises have developed multiple applications and have purchased packaged software, including ERP applications from vendors like SAP, PeopleSoft, JD Edwards and Oracle. Not all of these solutions have been designed as part of the enterprise architecture of the organisation, either because well defined enterprise architecture did not exist at the time or it was not possible to fit that specific application into the enterprise architecture.

Enterprise Application Integration (EAI) solutions are traditionally known to come with technical challenges like diversity of platforms; various transport protocols; and 'closed' applications. Software solutions like Microsoft BizTalk Server address most of those challenges by using standard protocols. Although these technical issues still need to be addressed, there are products out there that can help to address those issues. But not all of the EAI challenges are technical issues.

This introduces more challenges to the EAI solutions:

(a) Functionality Overlap

In some cases the same job can be done in two applications. For example in order to see the status of an order, an organisation may be able to use either sales or delivery tracking application. The problem is that none of them does the job completely. The customer is either transferred to another department, or told that "Our computers are very slow today". In the meantime the operator is launching another package to get a complete order status. And they may come back to the customer to ask their account number or name again because they didn't write it down the first time!

In an EAI project, there is normally no change to any of the existing applications so addressing the functionality overlap issue is a challenge.

(b) Data Duplication

For example, customer information may be held in accounting software as well as part of the CRM package, and this information can become inconsistent. This means customers may receive calls or correspondence offering to cross-sell services that they may already have.

Applications may have different interpretations of the same concept. For example in one application the term 'property' may be used to refer to a building, whereas in another application the same term may refer to a car as well. So when it comes to integrating applications, there may be inconsistencies in the semantics of terminology.

Since these challenges are related to semantics, they need to be tackled using appropriate patterns, and not products. This introduces us to 'Enterprise Integration Patterns'. These patterns provide the best practices to solve common Enterprise Integration problems. So let's have a quick look at some of the patterns that can be used to solve the problems mentioned above.

Problem: Functionality Overlap
Solution: Shared Business Function, Service-Oriented Architecture

'Shared Business Function' pattern means defining a clear boundary for shared functionality, implementing it and then reusing it in multiple applications. Most of the time this shared functionality will be part of one of the existing systems, like the delivery tracking functionality in an order processing system.

Shared Business Functions can also be considered as 'services'. So a Service-Oriented Architecture (SOA) can be used to publish the Shared Business Function so that it can be used from other applications.

Problem: Data Duplication
Solution: Data Replication, Information Portal

To address the 'Data Duplication' challenge, either Data Replication or Information Portal pattern can be used. Data Replication pattern involves replicating data from one application to the other, so that they are in a synchronised and consistent state.

When it comes to data synchronisation, there are various options available based on consistency and transactional requirements. Information Portal suggests that information needs to be aggregated from multiple sources (different applications) and then displays the result back to the user.

Problem: Different Semantics
Solution: Canonical Data Model

As mentioned earlier, different applications may have different interpretations for a single term. This makes the process of inter-connecting applications very complex because every application needs to have a data adapter to translate the terms when it wants to talk to other applications. So the total number of adapters needed for an enterprise grows exponentially when adding new applications to the Enterprise Architecture.

To address this challenge, the "Canonical Data Model" pattern can be used, which talks about creating a canonical, enterprise-wide data model. This model doesn't have to be implemented by all of the applications. The purpose of creating such a model is to have a conceptual view of the data within an enterprise, and then have a mapping between the canonical data model and the data model used within any specific application. This helps to define the data adapters. Also there is no need to create a separate data adapter for inter-connecting every two single applications. An adapter should be created to transform the canonical data model to the data model used by any other application, and vice versa. Using this pattern will reduce the number of needed adapters and there will be a linear relationship between the number of needed adapters and the number of applications in the Enterprise Architecture.

Could web services be the new EAI?

One of the primary success factors for any EAI solution is how well it is agreed upon (and used) among various organisations and software vendors. Web Services are based on well established and widely used standards and specifications like XML and HTTP. Almost all Internet users are using HTTP as their transport protocol, while XML has become very popular because of its flexibility.
An additional benefit of XML and HTTP is that they are platform-independent. This means you can have a Web Service implemented using Microsoft .NET while the consumer is based on J2EE. Although the publisher and consumer of the service are targeted at a specific platform, the interface (which is XML over HTTP) is platform independent.

Barriers to the adoption of Web Services

Although Web Services are based on XML, the specification for Web Services does not outline what XML documents should look like. This is positive as it provides flexibility. But problems arise when the publisher and consumer of the service have different understandings of the contents of a message.

Standards are emerging from bodies such as OASIS, who defines and publishes schemas for different types of messages (such as Auto Repair, Legal XML and Tax XML). But more needs to be done in order for the true benefit of Web Services to be realised.

The second barrier is created by legacy applications. Almost every company has existing applications in place and most have not been designed with a service-oriented mindset. Software designers did not generally consider the fact that this application may be used by other applications as well as human users. So another challenge is to build 'wrappers' around existing applications and services so that they can be consumed by Web Services clients.

This can get more complicated when building the Web Service interface for a mainframe application that is running batch schedules and cannot be used interactively. This introduces the idea of using asynchronous clients for Web Services so that the client does not have to wait for the response and can carry on with other tasks in the meantime.

This can be taken even further by looking at Web Services as a way of message passing, and not making remote calls to objects. This means that you send a message to a service but do not expect an instant response. The only thing you get back is a receipt for your message. In the future you will receive the response to your message and you can use a tracking mechanism (like using a unique identifier) to correlate those messages together.

The third issue relates to managing aspects like security and transactions, which is usually done using proprietary mechanisms. For example if you are using Windows system, you use the Distributed Transaction Coordinator (DTC) service to manage distributed transactions.

When it comes to managing security you rely on built-in authentication and authorisation mechanism that is built into the operating system. The question is what to do when systems are disconnected, asynchronous and talking to each other using protocols like HTTP and XML?

The best solution so far to this problem is the enhancements that have been made to the specifications of Web Services like WS-Transactions and WS-Security. The positive aspect of Web Services is that it is based on Simple Object Access Protocol (SOAP) and SOAP was designed with extensibility requirements in mind. Basically none of these enhancements make

any modifications to original protocols, and initial specifications are now published as standards.

Technical Issues

SOAP provides a way of encapsulating calls to remote objects in XML format. Since the adoption of SOAP, new concepts like XML Schemas have been introduced. XML Schemas replaced the built-in encoding mechanism for the data, leaving developers with two ways of representing the data in SOAP messages: Encoded (which uses the built-in mechanism) and Literal (which uses XML Schemas to define data elements).

The trend towards Service-Oriented Architectures forced industry participants to look at those interactions as message passing and not making remote calls to objects. As a result, today there are two different styles for Web Services (Document-Oriented and Remote Procedure Call).

From an implementation perspective these various styles and encoding mechanisms have much in common and the differences are minimal, but the difference is enough to break the compatibility between the consumer and publisher of the Web Service. Today software vendors and standard bodies are making recommendations for organisations to use the Document-Oriented style, (which uses a message passing concept rather than making remote calls to objects) with a Literal encoding type (which uses XML Schemas to define data).

Another technical challenge is availability of good development environments for developing and consuming Web Services. Currently there are various products from software vendors such as Sun and Microsoft to support developers, but because of the fast developments in Web Services specifications, they have to make frequent updates to their products.

Microsoft has implemented and released some of these enhancements as add-on tools to .NET framework and Visual Studio.NET, and developers will see major changes when these improvements are baked into the environment in “Indigo”, code-name for the next generation of Web Services tools and concepts from Microsoft.

A Web Service is the best solution to implement a SOA as far as the user base is concerned. There is a reliance on HTTP, which is the most accessible protocol on the Internet. By looking back at characteristics of SOA-based solutions web services satisfy most of the requirements. But they compromise the following factors for the others:

- Performance: XML, which is the building block of web services is an overhead when compared to efficient native mechanisms to represent data. So web services can be easily challenged because of their performance.
- Reliability (Transactional Integrity): If providing a service that should be part of a transaction, how can you maintain the integrity of transactions over the Internet using HTTP? For example in the Microsoft world there is the option of using Distributed Transaction Coordinator (DTC) to achieve this.
- Security: How can users be authenticated over the Internet and authorised to use the service?

Another negative point about Web Services is the level of support that developers get from existing IDEs. Web Services allows IntelliSense feature to provide assistance to developers. Also compilers can detect any errors in the code at compile time, and not the run time.

So unless all of the above issues are resolved, there is a very good argument why web services should not be used everywhere. There should be very good reasons to implement a web service

interface for software. Even if this exists there may be a need to provide a more efficient way of connecting to the service if many internal clients in the organisation will be using it. For example the service can be consumed using COM+ inside the organisation, and provide a web service interface for clients that cannot use COM+ interface because they are behind a firewall or running on another platform.

Preparing for SOA

There are some negative points for web services, but they are resolvable. For instance performance issues can be addressed by introduction of faster processors.

And although a decision may be taken not to move to a service oriented architecture yet, preparation is needed for it anyway. Here is a list of things to consider when architecting software:

- Interact with the servers using a chunky interface rather than a chatty one. In other words, try to combine different method calls as a single method call as much as possible. If sending a message over the Internet, every byte counts and it is better to avoid the overhead associated with extra bytes in the header and footer of each message, as well as the time required for name resolution and routing.

- Try not to keep too much state on servers. Delegate this task to the clients instead. To keep all that data, a very big warehouse is needed, and this costs money. So it is better to ask the clients to keep those documents themselves. The alternative is to store the state in a database or another machine which is dedicated to this task.

- Have very well defined interfaces for services. When moving to the web services world, publish the interfaces to clients.

- Join the asynchronous world. Many of the servers serve the clients in an asynchronous fashion. So it is better for the clients to use the same approach. It is not a good idea to block the execution of the client for receiving a response when the server has not even started the processing. Try to change the view of a request/response paradigm to a message based system where a client sends a message to the server, and the server will send a message back to the client sometime later.

- Think of an alternative way of authentication and authorisation. Security mechanisms are different in web services world. Do not rely on platform-dependent security mechanisms since this will be of no use when you are interoperating with another platform. There is a security extension to the web services basic profile called WS-Security, which specifies the way security can be implemented in the web services world.

- Choose a platform that keeps existing versions of the same component in parallel. This way there can be a separate interface for each version and this allows the provision of service to various clients using different versions of the service.

Component technologies like COM+, CORBA and RMI have been a success since there are so many systems out there using these technologies. Introduction of web services addresses some issues which are not addressed by those technologies, but web services technology is not replacing any of those technologies in the short term. Consider using them in parallel to get the best of both worlds.

.NET and SOA

Currently .NET has the broadest support for web services profiles. Web services basic profile consists of XML Schema 1.0, SOAP 1.1, WSDL 1.1 and UDDI 1.0. This profile is supported by both .NET 1.0 and J2EE 1.4. Microsoft released Web Services Enhancements 1.0 (WSE) in 2002, which adds the following profiles to the list of supported profiles in .NET:

- WS-Security: Describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication.

- WS-Routing: WS-Routing is a simple, stateless, SOAP-based protocol for routing SOAP messages in an asynchronous manner over a variety of transports like TCP, UDP, and HTTP.

- DIME: Direct Internet Message Encapsulation (DIME) is a lightweight, binary message format that can be used to encapsulate one or more application-defined payloads of arbitrary type and size into a single message construct.

- WS-Attachments: Defines an abstract model for SOAP attachments and based on this model defines a mechanism for encapsulating a SOAP message and zero or more attachments in a DIME message.

WS-Security and WS-Routing are published specifications whereas DIME and WS-Attachments are IETF Internet drafts at the moment while the work is in progress.

In addition to the web services .NET Remoting can also be used to implement SOA-based software. It allows the clients to connect to a server, without knowing about details of implementation of the service. The location of the server should be specified using a configuration file on the client. It should use the side-by-side versioning feature of .NET, which allows the publication of multiple versions of the same component at the same time.

Specific binary format for messages results in a better performance as compared to web services and HTTP can still be used, so that firewalls do not become a concern.

The major drawback for using Remoting is that it is platform dependent, meaning that both client and the server need to have .NET framework installed.

Visual Studio .NET is a great time saver for developers. As mentioned before, having an environment that provides IntelliSense when consuming a web service can be a great step forward. Visual Studio .NET 2003 allows developers to locate a web service by searching the local machine, UDDI servers on a local network or global UDDI directory. Once the reference is added to the web service of the project, .NET builds a proxy for the web service and the web service can be used as if it was a local class on the machine.

Another benefit of .NET is its built-in serialisation mechanism for classes like DataSets that are usually used to transport data. A DataSet can easily be returned from a web service without being concerned how to serialise and de-serialise the data. It also allows customisation of the serialisation process if improved performance is needed.

Windows Server 2003 has built-in support for Web Services. By native support Microsoft does not mean just an upgrade for IIS to run Web Services. Windows Server 2003 benefits from a device driver names HTTP.SYS which serves HTTP requests. Since device drivers run in kernel

mode rather than user mode, it will greatly improve performance. Windows Server 2003 has a built-in UDDI server too, which means that you can register and publish your web services on your own UDDI server.

Although standards are emerging to help developers 'plug and play' software components in a service orientated architecture, there are still complex choices to make when building new systems. However, by adopting some of the best practice processes outlined in this article, developers can avoid building up problems for the future.

Do you want to discuss the content of this article? Visit the Methods & Tools Forum:
http://www.methodsandtools.com/forum/index.html

Collaborative Source Software
Combining the best of open source and proprietary software

Stéphane Croisier, scroisier2@jahia.com
Jahia Ltd - www.jahia.org
CollaborativeSource.org - www.collaborativesource.org

Introduction

There are two worlds in software today: the closed source world of traditional software vendors and the open source (For convenience we use "open-source" or "free software" - note the lower casing- to encompass Open Source, Free Software, and similar endeavors) world of Linux, JBoss, MySQL and others. Neither the open source nor closed source models are new: both have been around almost from the beginning of computing. But neither has managed to take over the other.

More and more customers are today disappointed by closed source proprietary software but they do not want to rely part or whole of their IT strategy on some pure open source projects managed by virtual people. They want real software vendors with long term dedicated and trained developers, real product release cycle management, strong support agreements and code warranties.

After having carefully studied both approaches in order to license our own software, we decided to create a new paradigm which combines the best of both worlds. In this article I will present it, how we came to it and our experience after some years of practice.

1. Current business concerns with open source software

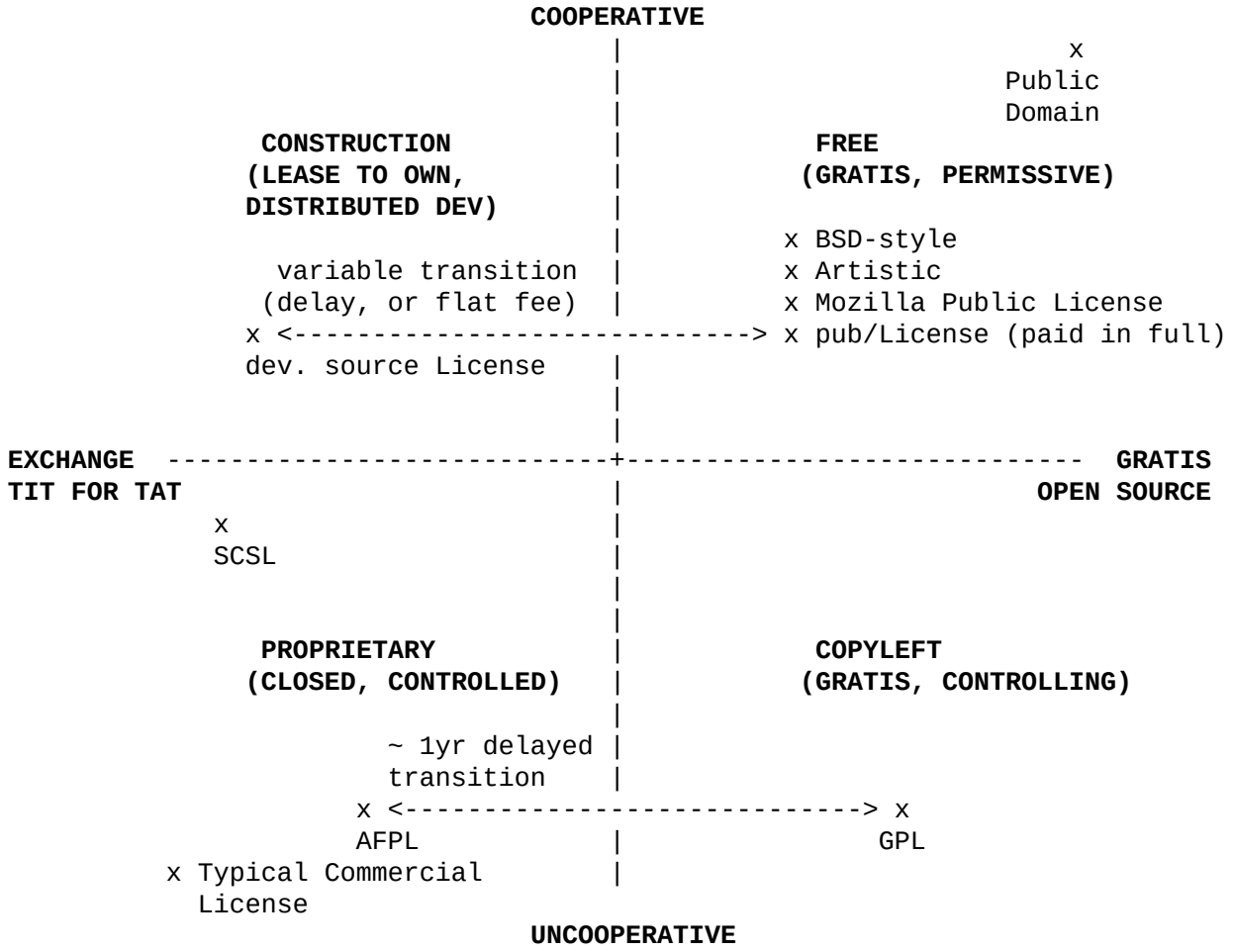
A great deal has been written recently about the advantages of open source software. Some writers have predicted that it will overthrow the commercial software world. In its present state I find this improbable. Open source software has too many inherent limitations despite the fact that the strengths of free software are still very real. These are discussed below.

Analyzing Open Source software

Open Source software delivers real advantages but has also certain weaknesses especially in its possible underlying commercial business models.

Firstly, one has to understand the world of software license. Stig Hackvän has dressed a basic chart of the licensing universe that is quite explicit (©Stig Hackvän - **http://devlinux.org/devLicense.html**)

However this matrix is not representative of the size of each market. Most of the ready to use enterprise software is still located in the bottom-left quadrant. Why? Why are there so few ERP, CRM, CMS, Portal… or other similar types of enterprise software in the open source world compared to the proprietary offerings and why do open source projects mainly stick to the lightweight product corner and have difficulties competing with mid-range or high-end proprietary products? And finally why there is no existing license today positioned right in the middle of the quadrant allowing best of all worlds?



Software licensing universe

To understand it, you have to analyze one of the key criteria for an open source project which is usually proportionally inverse to the success of the project: “the economic of respect”. The more an open source project is successful, the less this criteria will be respected.

The economics of respect is only relevant for a small fringe of the population

The hacker community is based on a gift culture. The term hacker is taken in the meaning of a computer enthusiast, i.e., a person who enjoys learning programming languages and computer systems and can often be considered an expert on the subject. The term can be either complimentary or derogatory, the negative connotation refers to individuals who gain unauthorized access to computer systems for the purpose of stealing and corrupting data. Hackers, themselves, maintain that the proper term for such individuals is cracker. Its inhabitants gain respect not by amassing property but by giving it away. And the word "community" is important as well. It means a particular sub-culture. Humans evolved to cope with a community of a few hundred people at most, not hundreds of millions. We solve this problem by creating mini-communities and seeking respect within those. Hackerdom is a cluster of these mini-communities and hackers seek respect within them by contributing code "for free". When the group felt that the person had "earned" the merit to be part of the development community, she was granted direct access to the code repository, thus increasing the group and increasing the ability of the group to develop the program and maintain/ develop it more effectively. This principle is called "meritocracy": literally, govern of merit and is used in most open source projects.

Then "respect" acts as a form of value. The analogy with money is quite close. Like money the supply of respect is limited, and members of the community value it. However there is one big difference: respect cannot be traded between communities. I might gain respect amongst the community of hackers, but that won't get me free credit from a shopkeeper. In contrast money is freely exchangeable between communities. If I want something from a shop I have only to walk in with money. This is the great limitation of gift cultures: they can only barter with the outside world. To do more you need money.

This limitation means that money has a divisive effect on gift cultures. A member of a gift culture is tied to that culture: the amount of respect he has earned is in the heads of other members of the culture and cannot be transferred elsewhere. But if the same individual obtains money instead of respect then that money can be taken elsewhere and even traded for respect in another gift culture if desired.

The lack of mobility in respect matters in another important way. This lies behind the greatest failure of the free software effort: until recently its products could only be used by other hackers. Documentation was sparse and poorly maintained, and user interfaces were arcane and forbidding. The reason is simple: hackers don't see interest in easy-to-use packages, and so would give little respect to anyone who wrote one. If you want the respect of hackers then you are much better advised to write something that they want. The rest of the world needs easy-to-use software to handle routine administrative jobs, but the money behind that need does not buy respect in the free software community.

There is nothing like a free beer

Furthermore for the rest of the world there is almost no such thing as free software. Of course everybody agrees when Mr. Richard Stallman, one of the founders of the Free Software Foundation, mentions: "Think free speech, not free beer". However due to the inherent clauses which needed to be assessed by the open source world in order to keep this “speech freedom”, the segregation between the free speech and the free beer became nearly impossible to apply in practice: with an open source license, everybody automatically gets the freedom to use, modify or copy the program from a royalty free manner. The software author can then only charge a fee on the distribution of his software. But if he plans to resell the “distribution” of his program, any of his customers also get the right to duplicate and redistribute his work for free. With the increasing popularity of the Internet today, this is now becoming even easier to freely “redistribute” software all around the world in a few clicks. Software distribution is then not an entry barrier any more. So finally open source software equals free beer in most of the cases.

Then commercial companies based on open source software needed to find other ways to generate some revenue streams. The most well known commercial open source business models today are for example based on:

- Selling some support agreements and a wide range of value added professional services on top of a widely spread open source project (e.g. JBoss Inc)
- Proposing some proprietary modules or add-ons on top of an open source framework (e.g. Covalent) or commercializing a distribution with improved installers, tested drivers… (e.g. Linux Distributions)
- Proposing a yearly subscription fee to get some enhanced documentation, some automatic upgrade programs… (e.g. Redhat)
- Dual licensing the software under one commercial and one open source license (e.g. MySQL AB).

Most of these business models still face one critical problem. They are giving away toothpaste while reselling toothbrushes. However they still need to cover both production costs. Everybody agrees that if a programmer writes some software then it necessarily is at some cost to somebody, even if that cost is only the payment that the programmer could have had for writing something commercial (known to economists as an "opportunity cost"). The problem is that, to go on with the comparison, the producers also publicly disclose their toothpaste formula and agree to let anyone freely copy and reuse it at will. Anyone may then easily understand that such a mechanism is not a reliable formula from an economical point of view. Any “competitor” may just take your toothpaste formula for free, decide to do not involve himself in your “R&D community of gift” in order to share with you the underlying maintenance and improvements costs and just focus on the commercial lucrative aspects: that’s to say selling the toothbrushes. The “competitor” is then easily in order to provide more affordable toothbrushes than the original producer may never do.

Hang on a minute. If this is so, why do so many people write such good software for free?

- Some of the incremental progress in free software can be attributed to the value of existing free software. If someone finds free software that almost fits their requirements then a few extra features can often be added for less than the cost of switching to a commercial package which could handle them.
- Some software is written by government employees, and is therefore automatically placed in the public domain. The cost of developing the code is then indirectly financed by your taxes.

- Increasingly, companies are releasing the source code of their non-strategical and non-tactical legacy IT asset in the hope that other programmers will extend and maintain them for free.
- A software company may decide to commoditize a certain technology in order to gain some market shares and be in a better situation to leverage other products or services. However this practice may only be applied by large software vendors which can leverage cross-selling opportunities in term of hardware, software or services

Open source programs assume de facto today that a certain number of end-users will never involve themselves into your community of gift. Such users like what you are doing, they want to use it but even if they wanted to they do not have the level of skills, experience or even just time to enter and involve themselves into your sub-community. This trend is even worse if the software company open sources a ready to use program as most of the end users won't be any more hackers but common anonymous end-users. They become what we commonly name a technology free-rider. Their only interest is then not to gain some kind of "respects" but to simply freely reuse the work other did and, as good capitalists, to maximize their profits.

While your open source project is limited to a certain sub-community which can naturally respect each other member and who does not really take care if third party people can take benefits from your resources or if your project is indirectly financed by your taxes or by a company which agrees to finance your open source driven effort for one reason (extending the company market shares) or the other (trying to share the development costs), a classical open source model may be perfectly adequate. For other more traditional software vendors especially SMEs which focus on one or a limited set of programs (without any cross-selling opportunities) and that have to deal on a daily basis with the liberal system, they will face a major problem: they can not tolerate technology free-riders. There is no longer a win-win relationship here. As the program targets a larger audience the number of technology free riders increases, creating a feeling of being abused by the system for the software vendor. This certainly explains why so many software programs stay proprietary.

Solution: enforcing a real quid pro quo paradigm

- In summary we may say that the key open source advantages are:
- Free access to source code
- Unlimited right to modify, copy and redistribute the program
- Free use of the software

And the main open source weaknesses are:

- No possible direct license revenue streams leading to some absurd business models (technical documentation only available against payment, stable releases only disclosed to yearly subscribers, dual commercial license available to remove any possible viral effect on your code modifications...)
- A growing number of "technology free riders" who will never contribute to the program despite their increasing usage of open source technology.
- Most of the time, a lack of a competitive and reliable "software vendor" in charge of managing, maintaining, supporting the software and of coordinating the community of development on a dedicated and long term manner.

The major problem current open source licenses are facing is that they do not really enforce any quid pro quo paradigm ("something for something"). The open source culture is however heavily and strongly based on the gift culture we described above. For some open source projects this is perhaps not important to formalize it, as it is mainly restricted to a limited hackers sub-community who de facto respect each other. It is then clear for most of the interested parties that they will have to involve themselves in order to see the program evolved and maintained. For others, mainly ready to use finished software, the large majority of the end-users are technology free riders who do not ambition or even consider entering in your gift community and simply evaluate their own market advantages of benefiting from a gratis offering versus some commercial ones. There is no more sense of collaboration here, just some pure economic rationale.

Observing this for our own programs, we decided to "extract" the key benefits of open source and community based programs and to create a new licensing model which would respect most of the open source fundamentals, but which would enforce a strong quid pro quo paradigm where each user would have to contribute in cash or in kind to the project pro rata his usage of the technology.

2. Collaborative source software

Making open-source software commercial without some business tricks and workarounds (cf. the commercial open source business models mentioned above) while keeping the community and openness aspects requires two things: first that payment is collected in proportion to the benefit gained from the software, and secondly that active users in the community can compensate their license charges in proportion to their code contributions.

This means:

- Being as close as possible to the Open Source Definition (OSD) and allowing free access to the source code, free possible redistribution and free rights to modify the code
- Being able to introduce some limited charges based only on certain limited type of execution of the program
- Allowing payment in cash or in kind (contribute or pay paradigm)

That is what we named **Collaborative Source Software (CSS)** and that we defined in a general Collaborative Source Definition (http://www.collaborativesource.org/jahia/page417.html). The whole mechanism aims to enforce a real quid pro quo mechanism. The rest is similar to the open source definition (http://www.opensource.org/docs/definition.php).

The underlying philosophy of this approach is the following: “We are very happy to share our program for no payment with those who agree to contribute to the project (enhance, debug, document, translate,...), but we also think it is fair to charge a fee from those who are not ready to involve themselves in our community of gift.”

Allowing Value Added Execution-Based Payment

The origin of the strength of open source software is its freedom. Users are free to modify open source software and to distribute modified copies. Charges can be levied for the act of copying the software, but not for the information itself. CSS preserves these freedoms. Almost all the requirements for use of the trademarked term "Open Source Software" can be satisfied by a CSS license. The only freedom of conventional open source software which CSS modifies is the

freedom to use the software without charge: open source software requires that the software be available for any use without restriction, while CSS requires that certain uses carry a charge.

The access to the source code remains of course free. There should be no charge (other than the usual copying expenses) for obtaining, reading, modifying or compiling the source code. It will also be permitted to execute the software for experimental purposes, for example to determine if it is suitable for the job, or to test some modified copies.

Instead, the charges will be imposed only for certain type of execution which adds value. In a commercial setting this is fairly easy to define: the use of the software to assist in any way with the operation of the company is adding value, and therefore should incur a charge. In a domestic setting the value is defined by the purpose of a particular execution: running a game program provides entertainment, which is valuable, and using a word processor to write a letter saves time and increases the quality of the result.

There are many possible frameworks for charging, most of which have also been tried by the closed-source software companies. For example:

- Per-copy charges. A fee is charged for each computer with a copy of the software installed.
- Seat charges. A fee is charged for each user authorized to run the software.
- Concurrent user charges. The software can be used by anyone anywhere, but may only be used by a limited number of people at any one time.
- Per-invocation charges. A fee is charged for each time the software is used.

Accepting payment in kind

However allowing value added execution payment is not enough to create a community of gift. This is no different than any other shared source or developer source initiatives. To enforce a real quid pro quo paradigm you also need to let users finance their license royalties in cash or … in kind. Otherwise speaking you need to agree to “value” contributors’ work by providing to them some free license’s credits.

Thanks to this system, instead of just being able to contribute to an open source project if a user wants to, he will now **HAVE TO** involve himself or he will be "taxed" as a technology free rider by some license royalties according to the value he gets from the program. We call this simple paradigm the "**No Value Added Tax**".

Such a mechanism helps the original authors create and maintain a stronger community of active developers in the long run as it provides clear incentives to all the end-users to contribute. A software author does not have to wait that new “volunteers” spontaneously decides to assign some of their time in order to help. The original author now has a leverage to push them to involve themselves in the community.

This also provides some good arguments for the IT managers to allocate resources on your project. The price of your program is not free anymore. It has a certain market value and an IT manager will need to compensate it in one manner (in cash) or another (in contributions). The passive “wait and see” attitude we commonly see from a lot of open source users is now banned.

Finally, in the spirit of Open Source Software, modified copies of CSS packages must of course be permitted. This ensures that the authors of a package do not have a monopoly on improved versions. However the end-users shall still pay the run-time license fees for the original package

as well as any license fee imposed for the modification. The viral effect is no more on code (copyleft) but on contributions. A reseller or a distributor may then decide to freely repackage your work for research and test purposes. However when the final end user executes the software, he will have to pay to the various concerned intermediaries royalties in cash or in kind pro rata his use of the technology.

Valuation issues

Of course when trying to convert a user contribution into some license credits, the software author will face a valuation problem. This is not a trivial task and if he wants to go into the details, it may rapidly become quite complex. Indeed basing the valuation mechanism on a standard time = money equation may raise several problems: for example a junior developer will not have the same efficiency as a senior one, so the rate applied should then not be the same. In a similar manner you may say, from a marketing point of view, that a small killer extension may be worth more than a complex but not heavily used one. You will also face the problem of globalization: is it normal that certain users in certain countries with the same level of skills may need to work 10 times more in order to get the same license value. All these points should be correctly assessed but are not directly defined by the CSS definition. This is left to the responsibility of the program's author(s).

From our experience the valuation issue should not become an administrative burden. The real underlying objective is to enforce a quid pro quo paradigm, not to scientifically know if the value of the contribution is worth ABC or XYZ to the precise cent. Valuing a development is performed every day by system integrators all around the earth. The same best practices may then be easily applied to your CSS project except that the rules are this time inverted: the software author will act as the customer requiring a "quote" from the future licensee.

Synthesis

So, in summary the CSS licensing policy is taxing only to the one who attempts to get an unfair benefit of other peoples' work. The choice of payment type is completely up to the user. The quid pro quo principle is the best way to ensure the availability of high-quality, rapidly evolving software while keeping full and free access to the whole source code. Thanks to the community of contributors, users have battle-tested and well-integrated software. Thanks to the paying customers, the editor can afford to hire great developers and have them work full-time on developing and maintaining the product.

3. Advantages of the Collaborative Source License

As mentioning before, Collaborative source software solves a lot of business issues. You now have a real win-win deal for each actor. No one may feel left out in the blue.

From a customer point of view:

- Customers will have all the benefits of a program managed and supported on the long run by one or several **professional software editor(s).** This includes a dedicated full time R&D team, some level of free support and assistance, patches and bug fixes that could be developed rapidly,...
- Similar to open source software, customers will be able to contribute new generic features that they needed to the original project. This means that **the code will be maintained, upgraded and/or debugged afterwards by the whole community** and not only by the customer or the software editor.

- Thanks to the "**contribute or pay**" paradigm, customers will not increase their overall project TCO (Total Cost of Ownership). If they want to extend the program for one reason or another, they can use their license budget either to recruit or assign internally some development resources or to mandate an existing project contributor.
- This is the fundamental principle of Collaborative Source Development: thanks to this philosophy, an active contributor will pay only part of his royalty or, according to the value of his contribution, no royalty at all, reducing the cost of ownership in such a case to the cost of a standard open source project, that is to say zero.

From a developer point of view

- Free and unlimited access to the source code for all the latest stable releases and to the new version under development. No delayed open sourcing. No kind of hidden license cost based on some "yearly community subscriptions" required to get access to the latest stable builds or to the development documentation. Developers immediately get free access to the whole development resources as the license is only limited to certain types of value added executions (e.g. production server only).
- Similar to any classical open source project, developers are able to adapt the software to their own needs without the usual constraints imposed by proprietary code. Then if it's important for the company, you don't have to wait for the editor to develop the needed updates or patches.

From a software editor point of view

- The software editor is now able to finance his company from a reliable and competitive manner thanks to revenues coming directly from license and maintenance royalties. He can then stop inventing crazy business model, often difficult to explain to customers (let's think about the dual licensing model based on a "viral effect on derivative work" which may impact your code modifications if you take the free version of the program).
- As there may be some limitation on the software sales, the software editor may also improve control of the network of authorized resellers and vendors. This avoids having companies without a real knowledge of your product being able to resell any kind of services while using your product’s notoriety. This also allow the software editor to better control OEM, VARs or other type of software vendors that will embed part or whole of your technology for their own benefits.
- As the main architect of the technology and thanks to the mechanism of payment in kind, the software editor may also leverage cross-selling opportunities by also offering a wide range of professional services. Of course, in opposition to proprietary software business models, the editor no longer as the exclusivity of developing new features. However, in practice, customers will often mandate the software editor to enhance the program thanks to his larger expertise on the technology.

From a partner / system integrators point of view

- The partner (e.g. a system integrator) has full access to the source code. He can then solve problems for his customers (bug fixing code, adding some new features) without having to rely on the software editor in term of pricing, delay...
- The partner may get some license sales commissions as it is usually the case in the industry to cover his marketing and promotion costs
- The partner can finally propose a wider range of services to his customers without increasing the overall project TCO. The end-customer may mandate his regular system

integrator in order to develop a new generic extension which he will contribute back to the community once developed in exchange of a software license. This process creates a win-win situation for every actor: the customer gets the new extension he wanted, the partner can resell more value added development services without increasing the overall project pricing, the editor and the community finally gets a better program.

4. Feedback from experimenting collaborative sourcing

We will try to introduce in this section some best practices that certain software editors experienced while releasing their programs under a collaborative source license schema these last years.

Impact on software management

Managing a CSS project is quite close to managing an open source project. If a software author wants to really leverage the "collaborative sourcing" effect on his program, he will need to organize his developments to best leverage contributions from the community. This often requires significant adjustments for software vendors generally used to centrally control all their software release cycle management to have to move to a community based system. Thus agile methodologies such as XP are usually clearly more adequate. Such flexible methodologies may be more easily adjusted to virtual community members who will all follow their own vision, milestones and deadlines.

However, in opposition to open source projects where it is sometimes difficult to refuse volunteer contributions over the pretext that it is not documented enough or that it does not include certain test units, CSS allows you to better formalize the quality, milestones and deadlines of the "outsourced" developments. Rules are reversed. The original authors now act as a customer in case of a payment in kind. The follow-up, tracking and reviewing process of external contributions are then considerably eased by the fact that there are now some "contractual" agreements between the actors.

Finally the Technical Committee will play a role even more critical than for a similar open source committee. Not only shall it lead the technical vision, it will also be in charge of accepting, valuing, tracking and reviewing all the external contributions. But correctly applied, this will enforce a stronger quality to your project while considerably reducing your R&D budgets.

Impact on organization

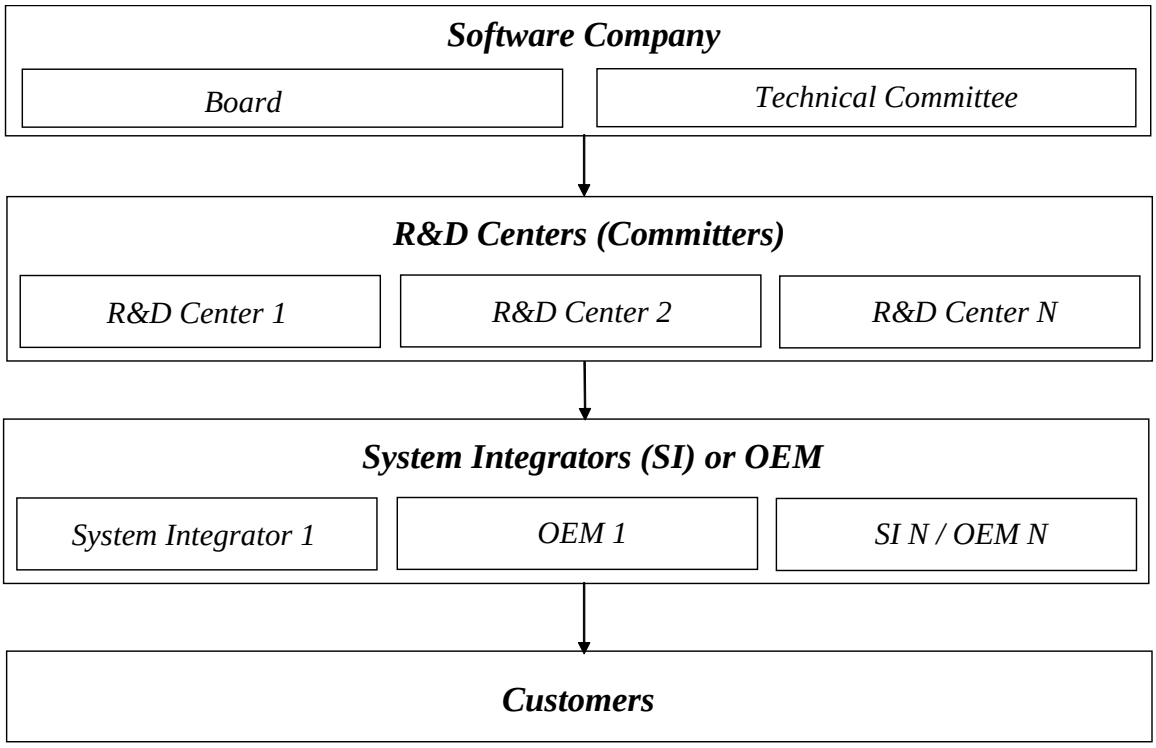
Thanks to the introduction of limited execution charges, the software company can now get some real license revenue streams that can be directly allocated to maintain and improve the program or to cover the direct administrative operating costs.

When allocating license revenues, two main approaches are possible:

- The classical software vendor approach. The original authors keep the license revenues internally in order to finance their own dedicated R&D team and optionally to make some benefits.
- The collaborative approach. The software organization acts as a shell (it may be either a commercial company or a non profit foundation) in order to secure the IT assets and redistribute the license revenues to key contributors. Usually contributors could be divided in two major categories: developers who extend the software on a periodical basis when they

need it and committers that are assigned on the long run on the project. In practice, the former tends to be employed by system integrators which allocate their IT resources according to the requests and needs of their customers, the latter tends to be small or mid sized software vendors which aim to leverage their sales thanks to the software.

This lead in practice to two main categories of partners: the System Integrators which integrates and occasionally develop on the software and the R&D Centers which acts in cooptation as the de facto software editor, the whole thing being managed and coordinated by a virtual organization.



The collaborative approach is of course certainly more community-centric as A) it creates competition among the various R&D Centers B) it provides warranties to the customers against any possible bankruptcy of one of the R&D Center and insures that other dedicated teams will still be driving the software on the long run.

Finally in a full virtual community based organization where there is no more one or a small number of lead software vendors but a multitude of independent authors, you may decide to allocate the revenues pro-rata the contributions realized. If you have created a commercial company to centrally manage your software asset, you may even think about providing shares to all the key contributors according to their respective efforts.

Of course CSS does not stipulate how the company revenues or shares shall be distributed among the contributors. This is up to the original author to decide how he wants to manage it according to the legal structure he puts in place and his product strategy. The only mandatory condition is to let users pay part or whole their license fee in kind.

Impact on partners

Thanks to the clear segregation among license revenues and professional services revenues (from support agreements to customization services), CSS also avoids conflicts as they may today arise between existing system integrators and other commercial open source business players. With a collaborative source program, the software vendor or the ones acting as the de-facto software vendors (the R&D Centers) may keep a partner neutral attitude and will not try to bypass their local system integrators partners by trying to resell their own range of professional services. In this business model each rule in the industry is clearly respected.

The limits of collaborative source software

The main limit created by a CSS program is obviously caused by the viral effect on contributions (and indirectly on royalties). An organization may not simply modify and enhance the software and then redistribute it in an unlimited royalty free manner to all its subsidiaries. Like for any traditional proprietary software, there is now some charges pro-rata the use of the technology. However the original authors may propose some kind of “Organization Wide License” that could be easily dealt in the spirit of a collaborative source win-win approach between all parties.

End-users shall also be able to predict the long-term cost of owning the software if they have to make some rational choices based on value for money. So in a CSS program, the fee for a particular version may only be increased in line with general price inflation. The practice of "stiffing" clients who need to add users or upgrade hardware after becoming locked in to a particular package is therefore banned.

Finally several users also raised the risk that a CSS editor may abruptly reject the collaborative source approach once a large community of users is locked to his software. Even if this risk is not limited to CSS project only (you may exactly face the same risk with MySQL AB for example), it is true that it may become more difficult to fork a CSS program rather than an OSS program. To compensate this threat and in order to get the same advantage than a classical open source program, authors can add a backward open source compliant clause that will force them to release the last version of their software into a classical open source license if they cease to support a CSS license.

All these clauses (and certainly new ones as we are still experiencing the model) should avoid a software vendor being able to abuse “the community” he created around his project.

Synthesis

CSS software could be easily adjusted to any type of programs. According to our experience it best applies to ready to use enterprise software developed by some small and mid-sized ISVs and whom value are more than 1’000 USD. If your program aims to the large public and only costs a few dozens or hundreds of dollars, the administrative costs of valuing contributions in kind will be too high compared to the community benefits you will obtain. Then a classical open source approach or even a developer source approach (source available only after having paid some royalties) would be more adequate. On the other side large software vendors may prefer leveraging some of their open source investments by choosing a cross-selling (open-source-proprietary extensions-services) strategy. However such a practice is often not possible for a small or mid-range ISV.

Summary of the main various software licenses

	Open Source		Collaborative Source License	Developer Source License	Proprietary License
	Copyleft license (e,g, GPL)	Permissive License (e.g. BSD)			
Source Code Access	Yes	Yes	Yes	Limited (e.g. under NDA)	No
Warranty of code availability in time	Yes	Yes	Yes	No	No
Code Modifications Permitted	Yes	Yes	Yes	Limited	No
Community based working	Yes	Yes	Yes	No	No
Derived Works Possible	Yes (but only open source derivatives compliant with the license)	Yes	Yes	OEM Agreement	OEM Agreement (based on a proprietary API)
Free redistribution	Yes	Yes	Yes	No	No
Free use	Yes	Yes	Limited	No	No
Royalties	0	0	0-∞	0-∞	0-∞
Payment	N/A	N/A	In cash or in kind	In cash	In cash

Conclusion

The sudden realization of the importance of open source software to the world economy has had a mixed reception from both capitalists and free software hackers. Hackers are afraid that capitalists will destroy the gift culture that they have created, and capitalists are unwilling to trust hackers with their future. Neither side really understands or sympathizes with the world-view of the other. Hopefully the two sides can find common ground in the principle of giving value in return for value. Trade is the basis of capitalism, and I hope that in this article I have shown how it is already the basis for open software as well, in the form of respect. By arranging for both money and respect to flow in the merit stages we can combine the advantages of open-source software with capitalist economics and motivation. This combination should conquer the world.

Do you want to discuss the content of this article? Visit the Methods & Tools Forum: http://www.methodsandtools.com/forum/index.html

Click on ad to reach advertiser web site

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1023-4918
Free subscription on : http://www.methodsandtools.com/forms/submt.php