Organizing and Building Software

Patterns for effective management of large and complex code bases

By Ralph Thiim (thiim@slb.com), Lise Hvatum (hvatum1@slb.com)

Abstract

Organizations which feature a distributed workforce developing software products with large and complex code bases require efficient management of the software artifacts and the developer environments to ensure the quality of the software products over time. The authors present three patterns that are part of a software management pattern language and which have proven successful for over 25 years within a large multi-national technology organization. First, a rule-based environment provides consistency to all developer environments, ensuring code compatibility as developers work locally, integrating and building their code against a shared baseline. Second, component baselines provide a structure to the shared software resembling a software development kit, delivering a full-product baseline that is only built when there are changes, isolating and reducing risk between full-product builds. Third, sparse workareas provide developers access to the local code being modified, while the remainder of the project code is secure in the shared baseline on the file system. These three patterns are offered as time-tested models for use by the creators of development tools suites.

Introduction

This paper was inspired by a desire to promote and discuss a set of practices in the authors' organization which has proven to be very valuable for the management of software, especially for large and complex code bases that are developed by teams who may be distributed both geographically and organizationally (i.e. by developers who contribute to the same code base but are assigned to different projects), and for software products which keep evolving over several years and in some cases even decades.

Although the practices described herein were developed internally in one organization, the number and diversity of software products in the authors' organization suggest that these practices have been validated and the ideas are of value to developers of software management tools, and perhaps to developers in general. The authors' organization uses proprietary tools for source control, software configuration management, build system, and issue tracking. These tools are built for the specific workflows of the organization, which leaves open the question of whether some of the practices can be implemented easily with commercial tools. Additionally, these proprietary tools do not provide the full set of capabilities currently supported by commercial tools. Nevertheless, this paper is put forth in order to share ideas, and to get in touch with others with similar interests who can help broaden and validate the contents.

Background and Overall Context

The patterns described in this paper have been in practice at a multinational technology company that develops complicated software systems using large, highly distributed teams. To help manage many of the inherent complexities, a proprietary configuration management and development tool suite is in place to help communicate and ensure consistency between all developers.

To fully understand many of descriptions found in the remainder of this document, it is important to understand a few key terms and concepts. The terms workarea, baseline and workpath have fairly specific meanings in this paper. A *workarea* is a directory hierarchy which may contain source code, build artifacts and a reference to the next workarea down the path. The files contained in each workarea overlays the corresponding file in the other workareas down the path. This makes the local source code supersede the corresponding source files down the path. Workarea found down the path are treated as if they are read-only. This is similar to the union mount concept support by some UNIX file systems (ref #3). Commonly, the next workarea is a baseline. A *baseline* is a workarea whose source has been fetched from the source code repository and is therefore reproducible at any time in the future. In the organization's vocabulary, a baseline also contains the results of a build attempt. A linked list of workareas is known as a *workpath*. The most common workpath consists of a developer workarea linked to a shared baseline, though a workpath may consist of any number of workareas. It is the capabilities and features made possible by "looking down the workpath" that make the patterns and concepts presented in this paper so powerful. See figure 1.

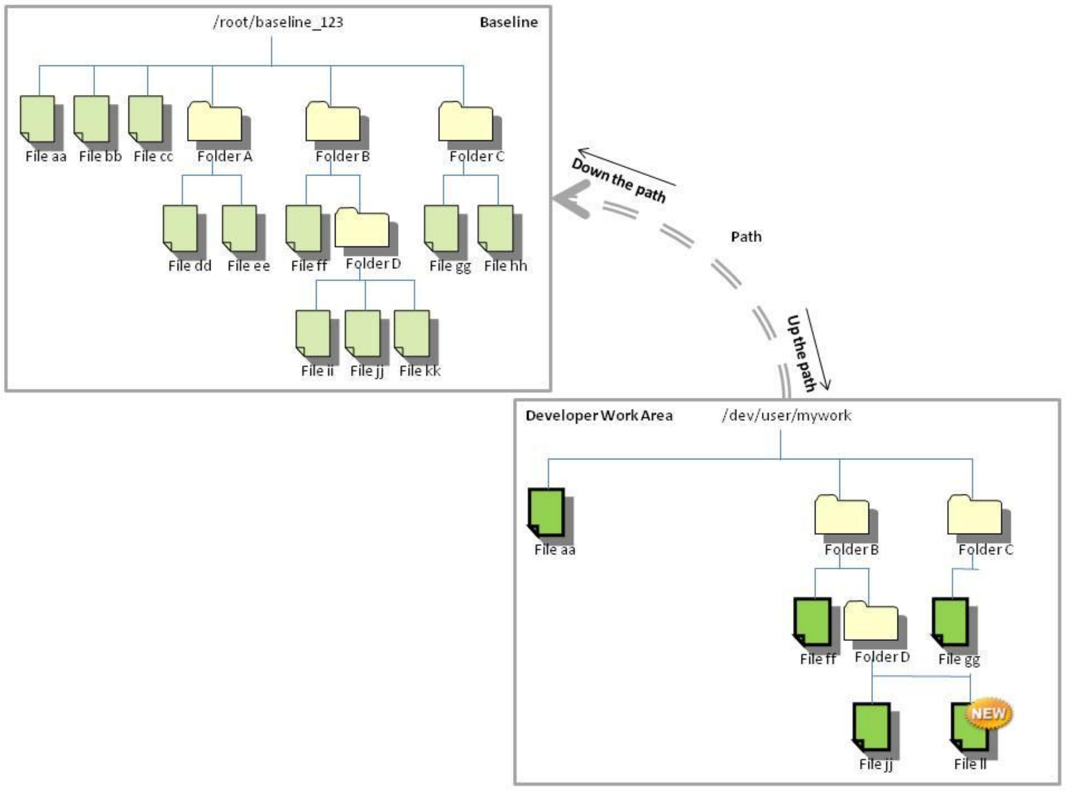

Figure 1: Workareas

Figure 2 shows the flattened projection of multiple areas as the workpath. The actual number of workareas that make up a workpath is not significant. The key point is the workpath view only shows the closest copy of a file down the path. Another common developer workpath which contains at least three workareas consists of a developer workarea pathed to a shared incremental-baseline which is in turn pathed to a full shared baseline. An incremental-baseline is a baseline build containing only the source modified since the next baseline down the path was built as well as all the dependent build artifacts.

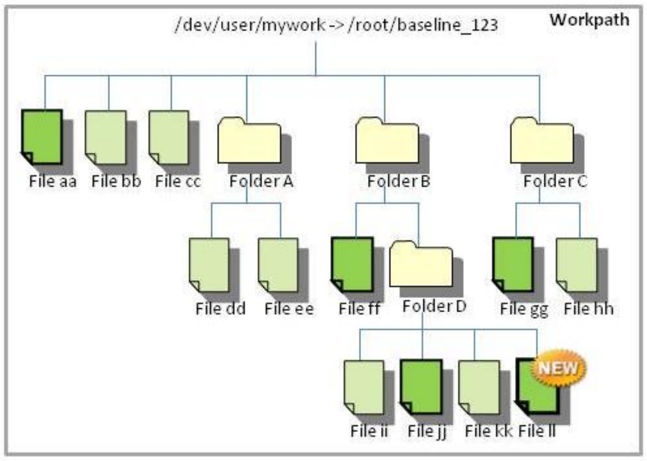

Figure 2: Workpath generated from the two workareas in Figure 1

A challenge for all project teams is to ensure that all members use a consistent development environment (i.e. the developer tools, third-party tools and versions, software development kits (SDK's), etc.). All developers on a team must use the same environment to ensure their software revisions are consistent to avoid team members from working with slightly different environments (e.g. different versions of an SDK). This can be made more difficult when teams must support both legacy releases in parallel with next generation releases.

The key principles of the software management solution include:

1. Developers work locally in their own branch built against a shared, baseline of known quality (based on published build and test results).
2. Developers continuously integrate and build their code against the shared baselines
3. There are no gratuitous duplicates of anything (source code, build rules, built artifacts)
4. Build dependencies are automatically determined by inspecting source code
5. The build and runtime environment is explicitly managed, including references to shared modules and third-party tools required
6. Dependencies from sharing of common software and frameworks are explicitly managed
7. Any previous software release, or baseline, can be exactly regenerated at any time

These principles are addressed by creating a flattened perspective of the local workspace contents with the contents found down the path. This flattened perspective is a high-fidelity replica of the developer's local changes applied directly to the current baseline source, thus approximating what the next baseline build would look like if the developer's changes were committed to the source code repository. The full description of the environment is out of the scope of this initial paper; the concepts discussed in this paper include:

1. Overlaying a developer's **Sparse Workareas** (i.e. containing only new or modified code) with shared, automated build results creating a workpath where local copies of files supersede their counterparts down the path. This uses the exact same lookup mechanism C/C++ developers are familiar with when they "#include <foo/bar.h" down the INCLUDE path.
2. Providing a consistent view of the complete code base by applying local workarea changes on top of a baseline down the workpath.
3. Merging the **Source Code-based Dependencies** (this pattern will be documented in a later paper) from the local workarea with the **Source Code-based Dependencies** found down the path.
4. Automatically applying the **Rule-based Environment** in all developer workspaces to ensure everyone uses the correct versions of SDKs, third-party tools and **Component Baselines.** These rules are typically archived along with the source code for a product.
5. Generating incremental, **Local Builds** triggered by local developer changes applied on top of the shared build results of an automated baseline build. All code affected by local changes are rebuilt in the local workarea.

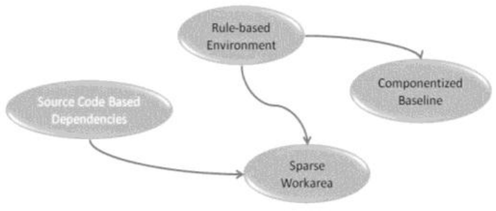

Figure 3: Patterns relationship

Name	Description
Rule-based Environment	Metadata used to establish a consistent, shared build and runtime environment to ensure code compatibility
Component Baseline	SDK-baseline distributed with source, build artifacts, redistributable manifests and rule-based metadata snippets to share code between projects and builds
Sparse Workarea	Workarea containing sub-set of code base with reference to next workarea on the workpath to leverage previous build results (e.g. a baseline)
Source code-based dependencies	Build dependencies automatically extracted from source files rather than maintained manually. Accurate dependencies allow incremental and parallel builds.

Rule-Based Environment

A Rule-based Environment is used to ensure a consistently controlled developer environment

Context

A software development team may have several developers working together on a shared code base to develop a software product or an update to an existing product. The developers have individual, independent development environments, and may be distributed throughout several geographical locations. The development environment depends on one or more third-party products being installed (compilers, application frameworks, testing frameworks) and possibly one or more **Component Baselines**.

Problem

How do I manage, document and apply a consistent environment for my whole development team?

Forces

Developers typically maintain their own development environments, yet it is critical that all team members' use a consistent set of development tools. If one member were to develop and test locally with one version of a third-party product while the remainder of the team used a different version, there may be testing and deployment issues introduced into the system.

Larger software products may be developed as a related suite of products. These products may share data formats, interfaces, and schemas and need to be released against a consistent set of sub-components. Development team members must be sure they are working against the correct version of subcomponents to avoid integration issues during building and testing.

Different release versions of a software product may require different versions of development tools. It is important that the developer utilizes the correct version of each tool when working against a particular product release (e.g. patches to a current commercial release and the next development release). Keeping track of various versions can be a challenge, especially to a development group consisting of members with varying lengths of experience on the team.

To manage changes to the development environment, team members must be aware of the changes and apply them appropriately.

Some developers may have to test new versions of the development environment and must be able to manage these alternate environments in parallel with the main versions.

Example

The developers on the Ariadne team each have their own laptop where they maintain their development software installations. The project uses a number of technologies (msbuild, SharePoint, Ext J4, NUnit). Although they started from a common set of installed tools, over time they have ended up using different versions of these same tools and are now experiencing build and runtime compatibility problems when performing integration builds and tests.

Solution

Create a managed developer environment by gathering environment-definition metadata from the workpath and apply the computed environment definitions prior to running build tools or when creating a build process shell. This ensures the developer's PATH, CLASSPATH, LIB, INCLUDE and REFERENCEPATH environment variables are defined appropriately and all 3rd-party file references get resolved correctly.

Below is a sample snippet of our environment rule metadata file. This snippet shows how the PATH environment variable is manipulated to reference the Visual Studio 2010 installation directories (i.e. via the $VSTUDIO_2010_DIR value) when the feature is enabled by the USE_VSTUDIO_2010 directive:

```
<Variable Name="PATH">
  <Prerequisites ifdef="USE_VSTUDIO_2010">VSTUDIO_2010_DIR</Prerequisites>

  <Prepends ifdef="USE_VSTUDIO_2010">
    <Prepend ifdef="WIN64">$VSTUDIO_2010_DIR\VC\bin\x86_amd64</Prepend>
    <Prepend>$VSTUDIO_2010_DIR\VC\bin</Prepend>
    <Prepend>$VSTUDIO_2010_DIR\Common7\IDE</Prepend>
  </Prepends>
</Variable>

<Variable Name="VSTUDIO_2010_DIR" ifdef="USE_VSTUDIO_2010">
  <DefaultValue>$VSTUDIO_2010_DIR</DefaultValue>
  <DefaultValue>d:\Program Files (x86)\Microsoft Visual Studio 10.0</DefaultValue>
  <DefaultValue>c:\Program Files (x86)\Microsoft Visual Studio 10.0</DefaultValue>
  <DefaultValue>$SystemRoot</DefaultValue>
</Variable>
```

The rule assures all users on this workpath have the correct version of Visual Studio on their PATH no matter the definition of their VS installation directory (i.e. $VSTUDIO_2010_DIR).

This solution works best when the environment-definition metadata (archived with the source code) is found down the workpath and combined with other metadata discovered down the path, for example, references to specific **Component Baselines**.

The ideal solution allows the same mechanism to assert whether or not the current environment meets expectations by identifying any missing components or incompatibilities.

Example Resolved

Using a rule-based environment scheme ensured that the whole Ariadne team used the correct versions of all required tools. Even the part-time developer who never reads his email announcing development changes to the team is kept up-to-date on environment changes each time he works against a new baseline build.

Resulting Context

When correctly applied, the described solution provides:

1. A managed, consistent developer work environment,
2. A traceable environment metadata artifact in which differences can be tracked to document changes to the build environment over time
3. Explicit assurance that all developer environments are configured as expected
4. A mechanism to assert the validity of the current developer environment
5. The ability to apply and test individual environment changes to the current development environment.

There are other approaches that attempt to provide consistent environments between team members, but suffer from some significant deficiencies. These approaches include:

1. Using static scripts to define the environment. These are hard to maintain and hard to read. Developers must take care to run the correct version before starting development tasks.
2. Using virtual machine (VM) environments to distribute a common environment. Though this is clearly a powerful approach for archiving environments used to produce production builds or providing a seed environment to all team members, they are not always easy to maintain and keep in synch between team members. It is also difficult to 'compare' the key features of one VM to another and ensure all environment changes have been applied to all developer VM images.
3. Common drop sites for shared software components. This approach makes it difficult to reproduce a previous build when a single drop site is updated periodically.

Discussion

To achieve the greatest benefit from a rule-based environment like the one described above, the developer tools must support the concept. Without integrated tool support, it is left to the user to apply the environment settings prior to launching any development tool. Towards this goal, the Open Source package management system NuGet (ref #4) helps incorporate third-party libraries into .NET applications. In the author's implementation there is the ability to apply the changes to a command shell process so any subsequently launched tools inherit the right environment. We also have Visual Studio and Eclipse plugins that manipulate the environment on startup or when the workpath is modified.

Each development team is also required to configure its rule-based metadata to suit its own environment. That implies each team must be able to, or have support to, configure the rule-based metadata to fit its needs. In some cases, there may be a need to extend the rule-based support to include new tools sets (i.e. new tools or new versions of tools).

Component Baselines

Component baselines can be thought of as "SDKs with benefits" and can be shared between multiple projects and teams.

Context

Software development projects usually depend on technologies developed by others in the form of third-party tools, frameworks, etc. These products can be a combination of proprietary modules developed by other projects in the company and commercial software. (Note: Determining the contents of a component baseline is outside the scope of this paper. It is up to the development team to decide how to best decompose their source code into smaller architecturally sound units).

Problem

How do I manage the build and deployment of shared components between multiple projects and teams?

Forces

When working with software components provided by commercial vendors or by other internal development teams, the consuming projects should be aware of the availability of new builds and have the ability to update to newer versions of the components at their convenience. When the version of a component is changed, it is crucial that the whole development team make the appropriate changes to their environments at the appropriate time.

Before newer versions of a software component are introduced to a project, an individual developer should first do a test build with the new version of the component. The only variable of the test build should be the new component and the minimum changes required to integrate it. This is so any changes in build results, product performance, stability or quality can be attributed to the new version of the component and not to other experimental changes applied simultaneously.

It is common for development projects to have multiple versions in development at the same time, for example, one or two supported commercial versions and a new version in development. It is critical that each product version use the appropriate external component versions. If a single developer works with more than one version of a component, he must be sure he is always working with the appropriate version. In the case of an internally developed component, it may be necessary to recreate a previous component baseline build from its original sources.

Example

The Ariadne program is a suite of applications that has been under development for many years. These applications share a common framework, application interfaces (API), and data schema, and are historically built together in a single, monolithic baseline. Though they are a highly interrelated suite of applications and nominally released on the same schedule, each application is commercialized independently and deployed on its own. Even when the code changes do not impact them directly, being managed in a single baseline causes build-turmoil for all application teams.   In the past, there have been compatibility issues when the applications where deployed from incompatible builds due to slight shifts in release schedules (and thereby the underlying common infrastructure).

Solution

Structure the shared software as a **Component Baseline** – an independently developed product delivered as a full product baseline (i.e. full source code and build artifacts), but consumed by other product development teams as if it were a software development kit (SDK). The difference between a third-party SDK and a **Component Baseline** is the ability to leverage a shared source code management system to manage the relationship between the consumer baseline and the component baseline via a compute-trigger file (see below) and a **Rule-based Environment**.

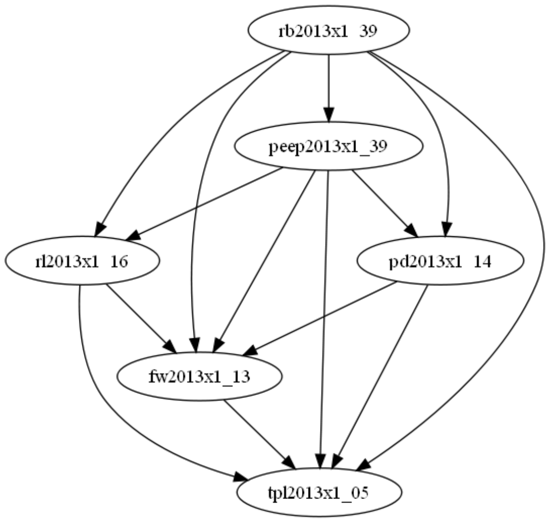

Figure 4: Component baseline dependencies example

A fully implemented component baseline provides:

1. A compute-trigger file is updated by the **Component Baseline** team to notify others of a new build. This file is also used by client baselines to declare interest in a specific version of a component baseline build. Updates to this file can be used to 'trigger' the need for a new client baseline build.
2. **Rule-based Environment** metadata to be consumed by client baselines when establishing their rule-based environments. Client baselines can ensure the appropriate build and runtime environments are applied as defined by the **Component Baseline.**
3. A redistributable manifest file declaring the publicly exported build artifacts client baselines can consume.
4. Optionally, an installation kit module (e.g. an msm or jar file) to be included by client baseline installation kits.

Component baselines share information about available builds via a compute-trigger file in the source code system. Each component baseline can/should have its own set of unit and regression tests to substantiate its quality. A component baseline is only built when there is a change. This isolates and reduces risk between client baseline builds.

The table below summarizes baseline producer and consumer actions and responsibilities with respect to component baselines:

Component baseline producers	Component baseline consumers (i.e. client baselines)
• Update a compute-trigger file with the each new build to notify others of the new build. • Maintain a redistributable manifest file declaring the publicly exported build artifacts. • Optionally generate an installation kit module consumed by the client installation kit (e.g. an msm or jar file). The contents typically follow the contents of the redistributable manifest file. • Optionally make the full baseline source and build artifacts available via the file system. This allows all consumers to share the binaries built by the original development team. • Generate a **rule-base environment** file snippet to help consumer baselines establish their build environments (i.e. update the PATH, LIB, REFERENCEPATH, CLASSPATH, etc.).	• Declare interest in a particular build of a component baseline by adding the appropriate version of the compute-trigger file to their source code configuration. • May allow updates of the compute-trigger file to trigger a build of their baseline. • May use the component baseline redistributable manifest file to make local copies of a component baseline's artifacts. This allows consumers of the client runtime to work without requiring access to the component baseline.

Example Resolved

The deployment problem for the Ariadne program was solved by the introduction of component baselines (see Figure 4). The original monolithic baseline used to produce all the applications was decomposed into multiple component baselines and multiple client application baselines. The component baselines were generally composed of stable code and the resulting application baselines became very small and easy to rebuild. The original deployment issue was resolved by having all application teams agree to release against the same version of the component baselines. Agreeing on the component baselines guaranteed the shared framework, API's and schemas were consistent for each release no matter when each baseline was released.

A serendipitous result of changing to a component baseline approach was that the team ended up with a much better architectural solution. The mere act of defining and implementing **Component Baselines** exposed several previously undetected architectural violations due to pathologic dependencies between the applications and the shared code. These violations were resolved by moving functionality from the application code to one of the shared component baselines.

Resulting Context

The **Component Baseline** approach offers a stable environment for all developers in all application teams. The code turmoil a team is exposed to is limited to their own component baseline and application changes. Changes to the other applications are removed from their field of vision since they are now managed in other baselines. Separating the baseline into multiple, independent pieces also allows individual application teams to experiment with new component baseline builds without exposing the other application team members to turmoil and risk until the **Component Baseline** changes are complete and tested.

These concepts seem quite simple, but are often not achieved in reality. We all know of instances where a product is built on a build artifact checked into the source code repository from an unknown build with a non-reproducible set of code.

Discussion

In the author's experience, **Component Baselines** have worked best when they contained automated regression and unit tests. This gives all consumers a good indication of the quality and stability of the component baseline itself.

Decomposing a large codebase into smaller architectural units makes sense for many of the reasons discussed above. Some of the other issues to consider include the tradeoff of creating too many components. For example, decomposing a project into one component baseline per .NET project would be non-productive. Clearly there is a small additional effort required to manage the contents of a component baseline and introduce it to the automated baseline-build queue.

It is important that each component baseline have clear ownership. An unfortunate consequence of separating the shared components from an application baseline is that the sense of ownership may not transfer to the component baseline. It is important that non-technical aspects of component baselines be taken into account when considering their implementation.

Sparse Workarea

A sparse workarea contains just the code being modified while the remainder of the project code is found in a shared baseline on the file system.

Context

Software developers who are charged with implementing a feature or fixing a defect establish a local workarea to integrate their changes with a particular branch of code. There may be a need to test or merge the same changes against multiple branches of the codebase.

Problem

How does an individual manage the integration of their work against the shared codebase developed by a team?

Forces

Development of a software project involves work done by multiple individuals contributing to a common codebase. Iterative development processes dictate that code should be released to the source code repository often and distributed to other team members as soon as possible. Though the goals of this approach are indisputable, they can lead to disruption for individual developers at inopportune times.

In general, software developers modify and improve existing source code and code bases more often than they create new files and components. Though these changes may be small relative to the whole code base, developers often end up fetching, merging and rebuilding everything locally in order to develop, test and release new changes. Dealing with the changes and instabilities introduced by others can be a frustrating situation when attempting to address an unrelated issue.

Example

Johnny tries to make changes to the user interface (UI) of the Ariadne-DevContr System, while Tammy has just made a release that introduced instability in the data access layer and the latest continuous integration (CI) build. The UI changes and the data access changes are completely independent, but pulling and attempting to use the latest codebase from the repository caused problems and delays for Johnny. Before resolving the issue, both developers ended up spending time troubleshooting the problem and lost work days.

Solution

Each user should use a **Sparse Workarea**, a work model where the developer workarea contains just the code being modified and overlays the remainder of the project code found down the path in a shared baseline on the file system. The link between the developer workarea and the baseline is applied and managed via a **Rule-based Environment**. Typically the build and test results of this shared build are posted and available for the whole team to see. If new CI builds are also created as sparse workarea builds on top of another baseline (i.e. an incremental baseline), then individuals may link their sparse workarea to their choice of baseline. That is, a new CI build does not obliterate the previous CI build results.

Individuals can manage their own workarea links and control when they update the link to another baseline. When linked to the latest code base, the resulting workpath becomes a high-fidelity facsimile of what the code base will become when the local workarea is submitted to the code repository. This results in the ability to perform CI builds in a user workarea before the code is permanently archived in the repository, avoiding possible turmoil for other developers.

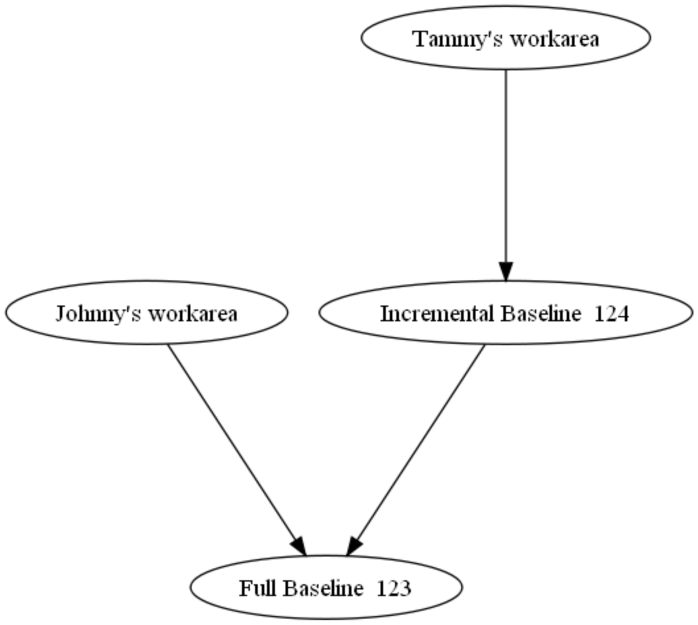

Figure 5: Sparse workareas and Incremental Baselines

Example Resolved

By creating a **Sparse Workarea** linked to the previous, stable build, Johnny is able to develop and test his changes against a known, good codebase while Tammy works on addressing the instability issues she introduced. If available, Johnny should re-path his **Sparse Workarea** to the next stable build for final integration testing after Tammy has completed her fixes. Johnny did not unnecessarily get exposed to any turmoil and risk while developing his changes. Figure 4 depicts Johnny and Tammy's individual **Sparse Workareas** as well as the incremental and full baselines.

Resulting Context

The **Sparse Workarea** solution allows individual developers to be confident that their changes are consistent with the latest snapshot of the code base by integration testing their code prior to submitting it to the repository. In fact, local changes can be verified against any other version of the code base by simply linking their workarea to another baseline. For example, if the individual file changes are consistent with the latest production build, the changes can be integration tested against that build by simply updating the workpath to reference the commercial baseline build.

Discussion

Classically, the common approach to this problem is to have the individual developer synchronize the project source code in his area, rebuild it all and test changes. Though developers using the **Sparse Workarea** approach still need to build and test their local changes, they are not required to rebuild changes made by others because they can leverage the results of the shared build found down the workpath.

Final Thoughts

As we pointed out in the introduction, this paper is our first attempt at sharing some internal practices on management of large and complex code bases. We are curious to see what feedback we can get, and we are hoping to find discussion partners to widen our understanding. Interactions with developers with other experiences will surely influence or way forward – hopefully leading to additional papers.

Acknowledgements

At time of submission, this paper was hardly more than an abstract. We are deeply grateful to our shepherd Philipp Bachmann who has taken both this paper and the shepherd assignment very seriously, and helpfully provided questions, comments and advice.

References

1. “Organizational Patterns of Agile Software Development” by James O. Coplien and Neil B. Harrison, ISBN 0-13-146740-9, Pearson Prentice Hall 2005
2. “Patterns and Advice for Managing Distributed Product Development Teams” by Lise B. Hvatum, Thierry Simien, Adrian Cretoiu, and Denis Heliot in “Proceedings of the 10th European Conference on Pattern Languages of Programs” p 279, ISBN 978-3-87940-805-4, UVK Universitätsverlag Konstanz GmbH, 2005
3. Sun’s Linker and Libraries Guide: http://docs.oracle.com/cd/E19253-01/817-1984/chapter5-90363/index.html
4. NuGet Package Management System: http://nuget.codeplex.com/