Starting Fresh Every Morning: Building a Development Image Every Day

Yann Monclair
yann@monclair.fr

August 17, 2008

Abstract

Source code management, version management, and release management are issues which are taking a greater place in our development cycles than they used to. The needs have increased, as well as the demand. Not managing your code branch properly can lead you to lose a lot of time in the delivery stages of your work cycle, as the application can not be built easily, when it should.

Kapital[1] tries to address such issues by rebuilding images every day. Every morning, the developers get the latest version of the development image. Using a fresh image every day allows us to gain time on merges, and enables us to cope with work coming into the repository from over 70 developers, split between various sites and timezones. This is a good practice, and can save developers a lot of time by eliminating issues other ways of handling source code and version build could lead to.

Introduction

Whether you use stripped down images or full blown images, when you make a release for a Smalltalk application, you still need to build an image. Building an image is not necessarily a fully automated process. But, like every software process, the more you automate it, the less prone to errors it becomes.

For Kapital, we build images for production releases, for testing, and every morning we build one from the development branch.

This paper assumes a minimal knowledge of Smalltalk, and of the object terminology. It also assumes a minimal knowledge of merging, regardless of the Source Code Management *(SCM)* system used.

While the aim is to stay as general as possible, the fact we use ENVY/Developer *(ENVY)* as our SCM may make parts of this paper harder to apply to other SCM systems.

[1]Kapital is a financial risk management and pricing system developed internally by JPMorgan. Kapital is written in Smalltalk, using Cincom's VisualWorks and GemStone's GemStone/S.

1 Benefits of Continuous Integration

There are many good reasons for changing the development image often. It's the easiest way to ensure your application's code can still be loaded in a fresh image. It's also a good way of removing a lot of "junk" you could gather in your image over time, such as undeclared variables, unused global variables and dead code not versioned in your SCM system.

Agile development encourages practices such as continuous integration. But being able to build an image isn't all you need for this, you also need automated testing to validate the quality of your build. This is more or less easy to achieve depending on the number of changes done to the application, and of course depending on the size of the application. For some applications, unit testing can be enough. For others, you may need some more specialised testing, to offer end-to-end testing of the various code paths. Kapital uses a testing system, named the "Smoke Test", to validate code changes, as well as the development image, when it's made available every morning. The tests include checking the external interfacing still works (SOAP, MQ, FTP, Email, GemStone...) as well as the core Kapital functionality (i.e. the financial capabilities of the application).

To give an idea of the scale of Kapital, the development branch has over 22,000 classes, roughly 10 times the size of the base VisualWorks image used for building Kapital. In a development cycle, approximately 5000 classes will be changed. Every day, we change from 60 to 150 classes.

This fast moving code base drives the way we work on the application and the way we interact with other developers.

1.1 Resynchronizing the Code Base With the Other Developers

Working in a big team on a big system doesn't mean everyone works on a different part. Every system has some classes that are central to any aspect of it, and get modified all the time. To handle changes on such classes, you often need to remerge your code on top of the currently released version of that class.

To illustrate that, let's reconsider the figures mentioned in the introduction. Everyday, 60 to 150 class changes are committed in Kapital's development branch. Each changeset pushed by a developer contains an average of 5-8 classes. And everyday we get 25 changesets committed on average. We can see that the maths brings us to the conclusion it's likely one class will be modified multiple times in a week, if not in a day.

The sooner you remerge, the easier. Resynchronizing the code base everyday helps in that respect.

Another advantage of resynchronizing is to avoid multiple implementations of the same thing. It's very likely that if you're adding functionality to a class, and if someone else meets the same issue at the same time, (s)he will also implement similar functionality.

While the implementation can be entertaining, it's never good to have multiple implementations of the same functionality. Eventually, different parts of the system will use one or the other, and any change to one implementation will need to be checked against the other implementations. This adds a lot of work for changes, such as system upgrades or bug fixes.

1.2 Avoiding Important Splits From the Main Branch

A danger of not merging changes often is to end up with important splits from the main development branch. While this is understandable when working on a big code change, it's always adding effort to the development, as the remerge effort will be increased.

By synchronizing your code often, and committing to the main branch, you can benefit from all the sanity checks the daily build offers. It allows you to validate your changes, and helps you decompose big tasks into manageable smaller tasks.

1.3 Checking Prerequisites

Depending on the versioning system used, the emphasis on prerequisites varies. Some systems won't bother with prerequisites. Another approach is to offer the option of specifying prerequisites, but not strictly enforcing it. The last approach is to enforce the use of strict prerequisites. This is what ENVY/Developer does, which is the SCM system used for Kapital.

To be able to extend or subclass a class, the package where you do it needs to have as a prerequesite the package defining the original class, or a dependent of that package. This forces developers to adapt the architecture of the code to accomodate these restrictions.

Because of the automated release of code we use on Kapital, we do not check the prerequisites when we release the code. This is validated when we build an image the following morning.

1.4 Avoiding Unknown Dependencies

When you work in the fully loaded code base, you can call any method from any package. While this works in the fully loaded image, it may not work in an intermediate step, during the build process.

A classic example is calling a method on a class during the class initialize method. While this will work most times, you could be calling a method that is defined only later in the load sequence. In the same way, the code could be referring to a class only defined later, or to a global variable.

Defensive programming can be applied in some cases to solve issues. However, using such methods will make your code much heavier and reduce the benefits you can get from the various refactoring tools available.

```
MyDummyClass>>#someMethod

#{MyClassOrGlobalVariable}
    ifDefinedDo: [:thing | thing
    doStuff]
```

Figure 1: A Smalltalk example of defensive programming: using guard clauses in methods

The issue with a method like Figure 1 is you can't rely on it having been properly executed on invocation. This can lead to an incomplete initialisation of your system, but we'll come back to that later.

2 Continuous Integration in Kapital

My experience building images is based on one of the biggest Smalltalk applications out there. I work with more than 70 developers, and we have a lot of changes going into the development branch everyday. Our team is also split geographically, and across many timezones, so changes don't stop when we leave the office.

In a big team, I would recommend having a subteam that specializes on the build process and its maintenance. You don't want everyone to be an expert in the build process. Having everyone be an expert could lead to a lot of wasted time around fixing the issue. It will also make the onboarding of any new member even longer, especially if your application is as complex and big as Kapital. Ideally, you have a designated person working on the build each day. The frequency this role gets reassigned (if at all), is up to you.

Before going into more detail, let's introduce ENVY, since people may not be familiar with this source code management system.

2.1 A (very) Quick Introduction to ENVY

ENVY/Developer is a source code management for Smalltalk-80. It was developed by OTI, who got bought by IBM. Up to VisualWorks 5i.4, Cincom also provided support for ENVY. When Kapital upgraded from VisualWorks 5i.4 to VisualWorks 7.x, the choice was made to port ENVY to VisualWorks 7.x rather than migrating Kapital's code to StORE and rewriting our workflow framework.

ENVY has a fine granularity for changes. It keeps versions of methods, classes, applications and configuration maps.

Let's start from the bottom. In Smalltalk, code is held by methods. Methods are held by classes (or class extensions). Applications (ENVY's name for packages) can hold classes and other applications. Configuration maps hold applications and other maps. A configuration map lets you specify the specific version of the applications and the maps you want to use. This allows tremendous flexibility when preparing the software for release, as you can fine tune the configuration to include only the changes you desire.

2.2 Loading the Top Level Map

Before we continue, let's look at how we use ENVY with Kapital. The Development branch in Kapital is an open map. It will only be versioned off at the end of the development cycle. All development code changes are applied via the change approval workflow framework we implemented over ENVY.

The way Kapital builds an image is strongly based on the ENVY notion of loading a configuration map.

ENVY includes a notion of ordering in the prerequisites. So if my map's prereqs are a, b, c, d, loading of my map with all required maps, will load a, b, c, d, and then my map.

The major difference with code management in Smalltalk is that grabbing the code from the repository and building the application are one and same step.

This leads to a clever mind trick, which has cost the writer precious hours on some occasions, when trying to fix a build. The code you are loading can be evaluated on load, but more importantly, it could be called later in the build.

For example, if we were to modify the way a configuration map is loaded from ENVY. we would start with a vanilla image, and start loading the Kapital map. The first part of the load would be done with the original code for loading maps. But once our code is loaded, it will take over. If we made assumptions in the code on other functionality being there, which is only loaded later, we will break the build.

To build Kapital, we load the top level map in a base image. Our base image is not a vanilla Cincom image, since we need to load ENVY into it, but it's as close as we can get with our restrictions. This load triggers the load of all the required maps, in order of prerequisite. Once the top level map is successfully loaded, we have a Kapital image. We can then simply save that image down and deploy it as the new Kapital development image.

2.3 Validating a Build

Now we have built a new version of Kapital, we need to validate it. We quickly mentioned this in the first section, but let's look at it in more detail now.

For Kapital we distinguish two families of testing systems for the development branch. Figure 2 shows the validation process used for a Kapital image build. Firstly, we have an automated run of our unit tests. While unit tests are always a good thing to have, they aren't the best tool for performing end to end testing of an application. The results of the SUnits are published to all developers, who can then look for the cause of failures of the tests they have interest in (tests the developer owns, or works on, or simply has interest in). This is our code driven testing. We are testing specific code paths.

Secondly, we have data driven tests. They provide end to end testing to ensure that the functionality still works as expected. These test systems are very specific to the application and its purpose. For Kapital, we use two separate systems. One is designed to check basic "business as usual" functionality, over a large set of data. For example, we value each product held in the database with the latest version of Kapital. This ensures that this very basic functionality is maintained for all existing products.

Another is designed to check more elaborate "business as usual" functionality, over a selected set of data. For example, we reuse known suites of calculations used in production, and run them against the development image, using only a subset of the full population of financial products. This validates that the necessary functionality is maintained, but to be able to check more functionality, we reduce the data set.

The former is only run against the fresh image built every morning. The latter is used to validate the fresh image, but it is also available for developers to test their changesets.

The results of the various tests are made available via web tools, for ease of access for all developers.

2.4 A Fresh Image Every Time

For Kapital, we don't reuse an image. We always start a fresh image, and load our changes into it, before continuing our work. This helps us ensure that the code is always in a loadable state in a fully built image.

As a general rule, if your code can't load in a development image, your code is wrong. While there are cases where your code can't load but is correct, they are rare. You always want to check these things with the wise men of your team (i.e. those who know the build process inside out).

2.5 Dangers of Savedowns

Building an image is a great thing, but it takes time. Building a kapital development image takes around 1 hour. There are many factors that slow down the image load. The verbosity of ENVY[2] and the slowness of refreshing the Transcript don't help

[2]ENVY prints a warning on the Transcript every time it creates an undeclared reference. ENVY will also print warnings when you compile a method that references a class by name.

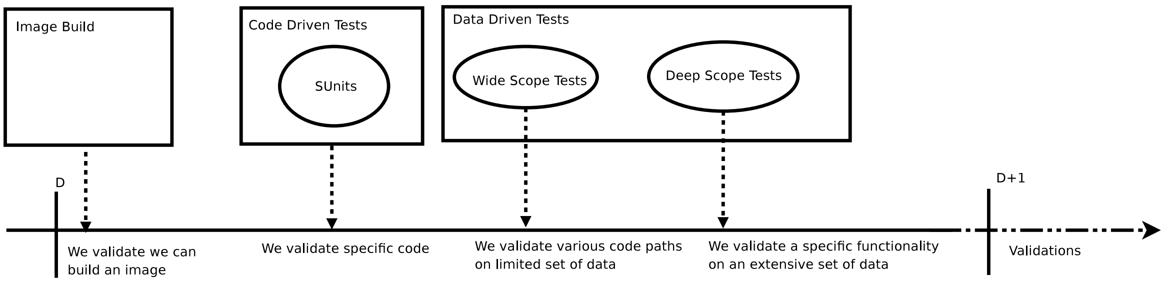
Image Build
Code Driven Tests
SUnits
Data Driven Tests
Wide Scope Tests
Deep Scope Tests
D
D+1
We validate we can build an image
We validate specific code
We validate various code paths on limited set of data
We validate a specific functionality on an extensive set of data
Validations

Figure 2: The validation process for an image build

to speed things up. But mostly, it's the titanic size of the Kapital code base. To give some scale, vanilla VW 7.3 (with ENVY) has 2200 classes (approx.). Kapital has 22,000 classes (approx.).

To speed up things, when you only want to try out a small change, you might just use a savedown.While this is quick, it is not a great practice for at least three reasons. Firstly, this usually means the code in the image hasn't been tested with a complete build. Secondly, savedowns also carry the risk of getting out of sync with the main branch and creating merge issues. Using savedowns goes against all the concerns raised earlier.

Lastly, and this is more an ENVY issue, there is no configuration map in the repository that matches the configuration of that image. This could lead to tricky issues[3] in your image if you tried to connect to a copy of the repository that doesn't have the class edition you use in your image.

While it is sometimes useful, this is not a practice that should be encouraged.

3 Breaking the Build and Fixing It

After overseeing the development image build for 18 months, I can tell you first hand you will always find things that keep you looking for much longer than you should.

You do not want the build process to become something that gets in the way of writing code. On the contrary, the build process is here to make it easier. After all, is not that the aim of any development practice, to make things easier ?

3.1 The Build Process

For Kapital, we have a team that handles the build (amongst other things). Like most software shops, we have our set of cronjobs running here and there. The development build is triggered by a cron entry. When we get in the office, the build and all the tests should be completed. If there are issues, we pick them up as we start our day, and start working on a resolution.

To help us fix possible issues, we have built a set of tools around our build process. These tools, like our implementation of the build process, are likely specific to Kapital, and can't be reused out of the box for another application. We have a developer wiki where we post useful information about the build run and the tests. For example, we publish the list of the change sets applied since the last build. This list is updated everytime we run the build script. This saves us the bother of going through the notification emails we get for each applied changeset, and it is also less prone to error. We also use a centralized wiki page where we summarize the status of the build and the tests. This offers us a quick check of the overall state of the system.

3.2 Broken Build

Because code is never perfect, and the build process isn't either, your build will eventually break. This is not a catastrophe, it's just one of the sanity

[3]The issues would be mainly having decompiled code in your image, as the equivalent source would not be available in the envy repository your image is connected to.

checks that failed. A change that went in since the previous build is causing this.

You always want to identify the cause of the failure before you attempt to fix it. The greater your experience and your knowledge of the application (and of the build process), the easier to identify issues and solve them.

In Kapital, we have identified three ways a build can fail. The first way is a image that successfully built, but failed the tests we put it through. This is the most common type of failure. The cause of failure will depend on your application and the types of tests. Code driven tests will fail because the code they exercise isn't behaving as expected anymore. Data driven tests can fail either because the code path is broken, or because the data is no longer appropriate.

The second way is that the build process itself will fail. The Kapital development build runs on a VNC display. This allows us to check the state of the build by checking on the image. Since we are using a Smalltalk image to build, we should be in a position where we have a debugger raised by the exception. Though the debugging capibilities of the debugger may be hindered by the unstable state of the image, you should be able to identify the method causing the issue, and find enough information about the sending context to start looking for possible offenders in the recent changes applied to the dev branch.

Most likely, it will be caused by an issue of prerequisites where the code calls a method only introduced later in the loading process. It can also be an uninitialized variable, in which case you would have an `UndefinedObject>>#doesNotUnderstand:` error. In any case, it's something you can chase down to a code change introduce after the last successful build.

The last failure is one that can be missed at first glance. For Kapital, it gets revealed by the automated testing. The build completed, but the load wasn't complete. In this case, you will find a development image that misses classes, methods... This is harder to investigate, since you don't have the exception visible in a debugger.

From my experience, such failures are generally related to the way errors are handled by your source code management system.

For example, vanilla VisualWorks allows overrides of methods in parcels, if you use StORE. But ENVY doesn't handle overrides. If you have an override (i.e. a second definition of an existing method in another application), ENVY will fail silently when attempting to load that application. This is the way ENVY handles failures while loading code.

The way you would observe such an error is by restarting a build, or manually loading the whole dev branch in a vanilla image. Like second type errors, these errors are also due to code changes.

If your source code management allows it, you can check what is loaded and what isn't. This will give you a good place to start investigating.

As a general rule, always identify the code change causing the error before attempting to fix it.

3.3 Build Errors

If the issue is with the development image not building, you need to solve that issue before being able to validate the build through your automated testing system.

I have seen three major types of errors introduced by code changes.

Calling a Method Not Yet Present

A simple illustration is an `#initialize` method on the class side of some class, let's call it MyClass. This assumes we added a class side method on OrderedCollection, or in its hierarchy, called `#with:with:with:with:with:with:`. So if MyClass wants to build an ordered collection with 6 elements while being initialized, we could implement `#initialize` as shown in Figure 3.

MyClass is defined in MyApplication. But we added the method on OrderedCollection in MyOtherApplication, which is loaded after MyApplication. When MyClass is initialized in MyApplication, the system does not contain the 6-way with: method. This will result in a Message Not Understood[4]

[4]This is also known as Does Not Understand. #doesNotUnderstand: is the name of the method invoked by Smalltalk on the object. If the object doesn't handle the message not understood in that method, it will raise an er-

```
MyClass class>>#initialize
"Initialize the class on load"

MySharedVariable :=
    OrderedCollection
        with: 1 with: 2 with: 3
        with: 4 with: 5 with: 6.
```

Figure 3: My initialize method using the 6-way #with: method.

There are two ways to fix this code. We could decide to move the new method on OrderedCollection to MyApplication, or even an application loaded before MyApplication. Or we could rewrite the `#initialize` method without using the 6-way with: method. This would give something as shown in Figure 4.

```
MyClass class>>#initialize
"Initialize the class on load"
MySharedVariable :=
    OrderedCollection new
                add: 1;
                add: 2;
                add: 3;
                add: 4;
                add: 5;
                add: 6;
                yourself
```

Figure 4: My initialize method, without the 6-way #with: method.

Whether you should choose one solution or the other is up to the coding conventions you use and the one you are more comfortable with.

Depending on Code Not Yet Released

When you work as a team, you often load code one of your fellow team members is working on, even if it is not released in the dev branch yet.

This is not in anyway a bad practice, it is part of software development. But it can lead to mistakes, which can result in a broken build.

ror MessageNotUnderstood.

If I apply a change requiring a method, or a class, introduced in my team member's change, this could result in a failure for the build. In this case, you have two possibilities to resolve the issue. You can either backout[5] the change which has an unsatisfied dependency, or you can apply the change it depends on. Again, this is a chicken and egg problem, and you have to make the call yourself, to know which is the best way forward. If you didn't introduce that code change, and don't feel confortable fixing it (or simply don't have the time), you should push it back to the original developers. My decisions are driven by one major concern: fixing the build as soon as possible.

Clashing Code

The last way of breaking the build is probably the most entertaining, when it comes to fixing the build. When two developers have code changes that clash, and that result in a build failing.

Let's look back at our types of failures. The first is a successful build that fails the subsequent tests. The second is when the build process falls over. Finally, the third type is an apparently complete build, that in fact misses a portion of the code. Clashing code is a way of introducing the latter type of failures.

In this case, both code changes are at fault. Depending on your approach to fixing the build, you might want to simply backout both changes or try and fix the build. In our team, our approach is to backout, and get developers to do the hard work of merging the changes. While this is a necessity in a team our size, you might want to choose a different approach, depending on your team size and the usual practices of your software shop.

As you can see, the reasons why the build breaks are usually trivial. Nonetheless, the issues can be difficult to identify and debug.

By catching build issues early in the process, you can save time when it comes to the release. If you let these errors slip unseen by not doing regular builds, you could waste a lot of time when you build the application for release.

[5]To backout refers to reverting back to the previous version of a class. We backout changesets, so we revert to the previous class version for each class in the changeset