DEVELOP MAINFRAME SOFTWARE WITH OPENSOURCE SOURCE CODE MANAGERS AND IBM DEPENDENCY BASED BUILD
CONTENTS

Develop Mainframe Software with OpenSource Source code managers and IBM Dependency Based Build

Introduction ........................................................................................................................................5
What does SCM stand for? ..................................................................................................................6
Mainframe Development practices and key concepts of modern SCMs and development ....................7
  Traditional Mainframe Library Managers VS. SCM Configurations .............................................7
  Working in full Configuration ........................................................................................................9
  Foundational concepts of modern SCMs applied to git .................................................................10
  Working in Releases .......................................................................................................................13
  Differences in building and deploying .........................................................................................13
Mainframe development and scopes ...............................................................................................16
  Layout of Dependencies of a Mainframe application ....................................................................16
  Applications and programs ...........................................................................................................19
  Applications and application groups ............................................................................................20
  Cross-cutting interfaces ................................................................................................................22
SCM scenarios ..................................................................................................................................24
  Develop at application Scope, then integrate and adopt new Interfaces .......................................24
  Adoption process ..........................................................................................................................27
  SCM layout scenarios ....................................................................................................................28
  Integration scenarios of shared Interfaces .....................................................................................38
  Conclusion ...................................................................................................................................57
Build scenarios ..................................................................................................................................58
  User build .....................................................................................................................................58
  Pipeline build ................................................................................................................................59
  Build strategies in relation to discussed integration scenarios .....................................................64
  Conclusion ...................................................................................................................................66
What’s coming in part two ...............................................................................................................67
Appendix ..........................................................................................................................................68
Tables of figures ...............................................................................................................................68
About the authors:

Dennis Behm, IBM Germany, dennis.behm@de.ibm.com
Nicolas Dangeville, IBM France, dangeville.n@fr.ibm.com
Rosalind Radcliffe, IBM U.S., rradclif@us.ibm.com

A very special thanks to the BNP Paribas team: Alexandre Antonescu, Sylvaine Caramelle, Pascal Rotilio and Frédéric Soigneux.

Thank you to the following reviewers:

Bruce Green, IBM Development Manager RTC Enterprise Extensions, Dependency Based Build
Daniel Bruce, IBM Dependency Based Build Developer
Francois Dumont, IBM Offering Manager for IBM RTC Enterprise Extensions, Dependency Based Build
Jenny Nicholson, IBM Continuous Integration Program Manager
Jorge Diaz, IBM Consulting IT Specialist
Liam Doherty, IBM Senior Software Engineer
Suman Gopinath, IBM Solution Architect for DevOps for IBM Z
Introduction

This document is intended as a reference for organizations who are interested in moving from traditional mainframe library managers to a modern Software Configuration Management (SCM) system supporting parallel development including branching and merging technologies.

This document outlines key differences between modern SCMs and traditional mainframe approaches. The purpose is to review relevant areas required when moving to a modern SCM, comparing the capabilities and providing explanations for how to adopt the new capabilities. There is an emphasis on how an existing mainframe application can be componentized and how the application interfaces can be described and managed, as well as then built. This paper is the first part of the story for a migration. Additional papers will be published describing areas such as packaging and deployment.

In this paper we will use Git\(^1\) and Jenkins as the example modern capabilities. Git is the de-facto standard for the open source community and is growing within major organizations. The industry estimates 48% of the software configuration management is now taken by Git. Currently Jenkins\(^2\) is the most widely adopted continuous integration coordinator for pipelines.

We will use IBM Dependency Based Build which enables you to implement mainframe build automation with Git and Jenkins for traditional mainframe artifacts.

This document is intended for build administrators from both the distributed and traditional z/OS sides as well as anyone interested in learning how traditional mainframe applications can take advantage of the capabilities of modern development tools.

Please visit the IBM Dependency Based Build Community at [http://ibm.biz/dbb_community](http://ibm.biz/dbb_community) to address questions and comments regarding this paper and get in contact with the authors. Look for [https://github.com/IBM/dbb](https://github.com/IBM/dbb) for additional help for developing your own IBM Dependency Based Build scripts.

The landing page as well as the documentation links for IBM Dependency Based Build are:


[https://www.ibm.com/support/knowledgecenter/SS6T76_1.0.1/welcome.html](https://www.ibm.com/support/knowledgecenter/SS6T76_1.0.1/welcome.html)

---

\(^1\) [https://git-scm.com/](https://git-scm.com/) Several distributions available, like GitLab, bitbucket, GitHub

\(^2\) [https://jenkins.io/](https://jenkins.io/)
What does SCM stand for?

SCM is the abbreviation for Software Configuration Management; it is also used for Source Control Management. Software Configuration Management is the task of tracking and controlling changes in the software development process. SCM provides practices such as the tracking of the version of configuration items. A group of files with a dedicated version is called a baseline of configuration items.

Software Configuration Management also includes the actual process of modifying the configuration items and implements a way to control the change process, also known as change management.

A subfield of Software Configuration Management is Source Control Management which covers the different areas of versioning configuration items, such as the isolation of different development activities into branches as well as practices like tagging, merging, and check-in and check-out operations.

Software Configuration Management differs from the term of Software Change Management which is used in the mainframe domain as the practice of version control and journaling the different changes between versions in a library structure.
Mainframe Development practices and key concepts of modern SCMs and development

Mainframe Software development started before the foundational concepts of configuration management were adopted in software development practices. The next section is intended as a level set on how companies are currently working with mainframe Library Managers and gives an introduction to the key concepts of modern SCMs.

TRADITIONAL MAINFRAME LIBRARY MANAGERS VS. SCM CONFIGURATIONS

Most of the traditional mainframe solutions control libraries on the mainframe and therefore are called library managers. The action to check out code from a repository is usually known as the reservation process because files get locked by the developer for the time of their change request. This leads to a serialization of processing change requests. Since only the files which the developer intends to edit are reserved and copied to the development stage, there is only a small subset of the overall controlled files in the development level. (This may be a single development level that all developers’ changes are copied to or may be a more isolated area with just the contents of a set of related changes known as a package.) To be able to compile a program, the library manager relies on the concept of concatenation of the build path through the SYSLIB, which means that the different stages are concatenated like a class path in Java development.

Let’s use a scenario to clearly illustrate this concept. We assume four staging levels: development, functional tests, acceptance test and production. In this environment three parallel paths have also been defined. The developer, let’s call him Dirk, is required to promote his change through these staging environments. Depending on the library manager, when Dirk checks out the file it might be in a development level or it might be in an isolated package. The first thing Dirk must do is identify which of the available paths he will use based on the change he makes. If it’s an emergency change, it’s easier, because he must use the emergency change path. If it’s not an emergency change, then he might be assigned a path based on the planned time for the release of the function, or he might have to look to see which environment is not currently changing the same parts he will work on. Once he has the path he can check out the file and begin to work.

If Dirk works in his own package, then he continues to add files to the package and build and test his change. Once satisfied with the changes and ready to promote, he needs to run an audit check to see if anyone else has changed anything related to his change, such as a copybook that has been included. If other changes have already been promoted, he must then get those changes into his package, rebuild, and test before moving forward.

If Dirk works in the development (dev) library level, then all other changes in that dev level are automatically included in his change. When ready to move forward, he must check to make sure any changes he may have picked up are also ready to move forward. If not, then he must manually remove the other change, rebuild, and then retest to promote.
As you can see from the scenario above, there are many manual tasks and reviews that must be performed to work on the function. Many times, rework is required due to late decisions about what function will move forward and what will not.

Have a look at the pros and cons of this approach:

### Figure 1 Workflow example, working with a library manager

- Dirk identifies the programs he plans to edit and the path he will follow
- He checks out the members to the development stage or his own package
- Dirk starts modifying the members.
- Dirk also checks if other changes will move forward or not. He may need to rebuild, retest before promoting.
- Based on the list of dependencies, he checks if there are any other changes with a dependency to his work in the staging hierarchy.
- Dirk needs to identify and anticipate dependencies to other members.

### Figure 2 Pros and cons of working with library managers

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>mainframe development teams are used to this approach</td>
<td>highly manual and repetitive tasks to check for dependencies</td>
</tr>
<tr>
<td>does not need much space</td>
<td>serialization of work when working with subsets of the configuration, dependencies to other activities in the staging hierarchy remain.</td>
</tr>
</tbody>
</table>
WORKING IN FULL CONFIGURATION

Compared to the approach described above, modern SCMs use the concept of configurations. A definition of a full configuration can be: “A configuration is made up of a given list of files, each of them at a given version. A configuration can have various scopes: for example, an application component, an application, a domain, or more.”

![Figure 3 Working in configurations](image)

With a modern SCM the configuration can be checked out together, supporting full isolation. At the same time, many other developers can also have their own version of the full configuration. This isolation allows full parallel development on the same artifacts.

Imagine the following scenario:

- Alice works on a maintenance request of the current release (release 11), while at the same time Bob implements new features in the same application component for the next release (release 12).

![Figure 4 Parallel release schedules](image)

- Alice implements the fix in an isolated configuration, while Bob works on the same files for the next release.
- Bob is a member of team working on features for release 12. The team can work on these individual features in isolation using feature branches.

Figure 5 Working in isolated configurations

- Once Alice completes her changes, she must merge those changes into the release 12 branch to avoid a regression. This merge capability is supported by the modern SCM. Bob also needs to merge those changes into his feature branch again using the built-in SCM functions.

Figure 6 Merging the different configurations

Conclusion: Branching enables teams to separate and isolate different development requests like hot fixing, maintaining the current release, or developing for a future release. It also provides greater flexibility to select features for a certain release and moving features between releases.

FOUNDATIONAL CONCEPTS OF MODERN SCMS APPLIED TO GIT

Git is currently considered to be the de-facto standard SCM in distributed software development, serving all necessary practices for the developer. Other modern SCMs continue to be used for specific use cases such as security requirements or highly regulated environments.

Git implements the functionality to work in teams, to branch and fork configurations, and to merge changes to the common codebase. Git is called a Software Configuration Management solution.
Unlike most other SCMs on the market, Git is a fully distributed SCM. The Git model is built on the assumption that there is a full copy of the repository anywhere it is used, including the entire history. With basic Git there is no concept of a master or a server; any copy could be considered the master. However, with most Git implementations in large companies today, there is a centralized instance of Git that is considered the server. This instance is used for audits and pipeline processing. Today there are many different implementations of a Git server that expand on the basic Git capability to improve support for automation and backup.

![Simplified Git workflow](image)

**Figure 7 Simplified Git workflow**

The picture above shows the distinct steps a Git user performs to edit code and share it with the team. The dark blue rectangle represents a central Git repository serving two branches (master + stable).

Since Git is a distributed SCM, each developer works in his own Git repository. A central Git repository is the single source of truth, which will be called `origin`.

- To edit a file, the user first **clones** the full repository to his workstation, including all defined branches and its history.
- Then he points to the branch where he plans to work. He **checks out** this branch to his sandbox, which is his environment for editing.
- He makes changes and then **commits** them to his local copy of the repository.
- To share the commit with the team and to the root repository, he needs to **push** the commit from his copy to the shared repository.
- The developer can also create personal branches which will also be pushed to `origin`.

http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102772
Please refer to the official Git documentation at https://git-scm.com/doc
WORKING IN RELEASES

Branches help development teams to work on several software releases in parallel.

Mainframe development is done incrementally working a few parts at a time, which in some environments allows deployment to production when required. Under the covers, change requests are handled in a sequential manner if they affect the same code. When an emergency change or a higher priority change is required, it can create a situation where the current work must be backed out so the priority work is completed first. The lack of true parallel development limits what can be done without doing lots of rework to back out changes or manually bring changes together. This limitation can at times cause a developer to lose changes when emergency changes must be made.

These strategies do not provide a formal plan on how to work on a larger scale or how to handle conflicting and depending change requests, creating a large demand for collaboration between individual developers and development teams. This impact magnifies as more changes depend on each other and testing needs to be coordinated. Timelines must be negotiated between the impacted development teams, which causes inefficiencies in the development process. To reduce this need for coordination, development teams agree with the line of business on a fixed release cycle.

For practices like Continuous Integration and Continuous Delivery, the fundamental goal is to have very short release cycles.³ To achieve this goal, development teams invest in automation and the ability to isolate and integrate quickly.

The SCM needs to support the methodology used for releases, meaning that several configurations can exist at the same time. During a release, a team will reach multiple different milestones that should be reproducible at any given time. For example, in Scrum the end of a sprint is a milestone.

Modern SCMs provide the function of identifying a set configuration for a milestone, many times called a baseline. For Git, this functionality is called a tag.⁴ This function provides the ability to mark important points in history within the configuration, such as a release candidate, the end of a sprint, a certain configuration for a build, or the actual released software.

DIFFERENCES IN BUILDING AND DEPLOYING

In today’s mainframe development process, building an application focuses on the compilation and linkage of the modified source files. Compiling and linking on a file basis or on a small-defined list of files is the common approach.

³ Article on Continuous Integration for Rapid development
https://martinfowler.com/articles/continuousIntegration.html

In distributed systems the build is managed by a build management system. For example, in C/C++, make manages the build. In Java development, Gradle, Maven or Ant usually drive the build with scripts defining the order of generating the binary files. The build management system uses automation to identify what was changed as well as the relevant dependencies. Very often, this leads to compiling all parts to generate the new complete application. In each of these cases, there is a build administrator defining the build process.

In traditional mainframe development, there is a hierarchy mapping to the testing environments. Developers promote their changes to move to the next stage of testing. This leads to a complete serialization of development activities.

![Mainframe Staging Hierarchy](http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102772)

**Figure 8 Mainframe Staging Hierarchy**

One of the key differences between traditional mainframe builds and distributed builds is working in a shared environment and having a static path for promotion versus the ability to work in an isolated environment and not requiring a static promotion path.

Distributed development teams work and test on isolated branches of the SCM and the test environment (see Figure 9 Isolated Development Branches), like branches of a tree.
Figure 9 Isolated development branches
Mainframe development and scopes

An IT system is developed by many teams and composed of different applications driven by the line of businesses and consumers. Applications need to interact to provide the overall system and interact through defined interfaces. Using well-defined interfaces allows the parts of the application to be worked on independently without necessarily requiring a change in other parts of the system. This application separation is visible and clear in a modern SCM, allowing clear identification of each of the distributed applications. However, in most library managers, the applications all share a set of common libraries, so it is much more difficult to create the isolation.

In this section, we discuss ways of componentizing mainframe applications so they can be separated and the boundaries made more easily visible.

LAYOUT OF DEPENDENCIES OF A MAINFRAME APPLICATION

From a runtime perspective in z/OS, programs run either independently (batch programs) or online in a middleware (CICS, IMS) runtime environment. Programs can use messaging resources like MQ queues or data persistence in the form of database tables, or files. Programs can also call other programs. In z/OS called programs can either be statically bound or use dynamic linking. If a COBOL program is the first program in a run unit, that COBOL program is the main program. Otherwise, the COBOL program and all other COBOL programs in the run unit are subprograms. The runtime environment involves various layers, including dependencies expressed between programs and resources or programs and subprograms.

There are multiple types of relationships to consider. The source files in the SCM produce the binaries that run on z/OS. To create the binaries, a set of source level dependencies must be understood. There is also a set of dependencies used during run time. These multiple levels of dependencies are defined in different ways, and in some cases not clearly defined at all. Understanding and finding the dependencies in source files is the first challenge.

Building a program involves different steps:

1. Compilation including any pre-compilation steps, defined as explicit steps or as option of the compiler, creates a non-executable binary (object deck) file.
2. Link-edit which assembles the object deck of the program with other objects and runtime libraries as appropriate. Link-edit can be driven by instructions (a link card) from the SCM or as dynamically defined in the build process.

© IBM Copyright, 2018

Version 1.0.1, Oct 15th, 2018

Develop Mainframe Software with Opensource Source Code Managers and IBM Dependency Based Build
http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102772
As part of the current build process some additional steps, such as binds to databases, are sometimes included. The function of these steps is to prepare the runtime for a given execution environment. These should not be included in the build process itself, but should instead be included in the deployment process.

**Source dependencies during the build differ from runtime dependencies**

Most of the time when people think about an application, it’s from a runtime point of view. Several components are required for an application to run. Some of these are required as dependencies such as the database or middleware and its configuration, others are required as related such as other applications that might be called.

Everything running in a runtime environment starts as source from an SCM. Or at least it all should when you consider infrastructure as code. Some source files represent definitions or are scripts that are not required to be built. Those that do require being built generally require other source files such as copybooks, but don’t require the CICS definition, for example. Some of the source files are also included in many different programs, for example, a copybook can be used by many programs to represent the shared data structure. It is important to understand the relationships and dependencies between the source files, and when those relationships or dependencies have importance. The copybook is required to build the program, so it is required at compile time, but it is not used during run time. The configuration for a program such as the CICS transaction definition or the database schema is related to the application, but is required only for the runtime environment.

A concrete dependency is the interface description when calling a program. A copybook defines the data structure to pass parameters to a program. So, the copybook is important to be shared while the program is part of the implementation.
Programs call each other either dynamically or statically

On z/OS there are two ways programs are generally called: dynamically and statically. Statically called programs are linked together at build time. These dependencies must be tracked as part of the build process to ensure they are correctly assembled. For dynamic calls, the two programs are totally separate. The programs are built and link-edited separately. At run time the subprogram is called based on the library concatenation.

Many organizations have been moving to increased usage of dynamic calls as that approach reduces the complexity at build time. However, this approach means that the runtime dependencies need to be tracked and understood if any changes are made that require updates in both program and subprogram.

These programs and subprograms are inter-dependent even when using dynamic calls. When a program calls another program, generally they share data. A transfer of control occurs between the program and the subprogram with the main program passing a set of data to the subprogram and generally expecting some data in response.

Different mechanisms exist to share pieces of data based on the language or the runtime. However, there is a need for the caller and the called program to define the data structure to be shared.

The call of a subprogram is based on a list of transfer parameters, represented in the interface description like an API, but it is more tightly coupled than today's REST-based APIs.

You commonly define your shared data structure in an included source file, for example COBOL uses copybooks.
It is very common to define multiple copybooks for your programs to isolate data structures and reuse them in other areas of your application component. Using copybooks allows more modularity at source level and facilitates dealing with private and shared data structures, or even private or shared functions.

APPLICATIONS AND PROGRAMS

In a web application, it is relatively easy to define an application because the physical artifact that is deployed is the complete representation of such an application: the EAR or WAR file. In the Windows world, it is more complicated since an application can be made of several executables and DLLs, but these are generally packaged together in an installable application or defined by a package manager.

An application is generally defined by the function or functions it provides. Sometimes there is a strong mapping between the physical parts that are shipped and sometimes it is a set of parts that run the application.

In the mainframe, we fall closer to the second case where applications are defined by functions. However, based on the way the applications have grown over the years, there may be no clear boundary as to where one application ends and another one begins. An application can be defined physically by a set of resources (load modules, DBRMs, definitions) that belong together as they contribute to the same purpose: the calculation of health insurance policies, customer account management, etc.
At the source file level, the relevant files contributing to an application are derived from the runtime of an application. These files can usually be identified by different means: a set of naming conventions, the ownership, information stored in the SCM, etc. It may not seem obvious at first glance, but most of the time it is possible to define which source files contribute to a given application.

Scoping your source files to an application has many benefits. It formalizes the boundaries of the application, and therefore its interfaces; it allows to define clear ownership; and it helps with the inventory of the portfolio of an organization. Planning of future features to implement should be more accurate based on this scoping.

APPLICATIONS AND APPLICATION GROUPS

Within an organization, multiple applications generally make up the business function. An insurance company may have applications dedicated to health insurance, car insurance, personal health, or group health policies. These applications may be managed by different teams, but they must interact. Teams must define the interfaces or contracts between the applications. Today many of these interactions are tightly coupled with only a shared interface defining the relationship.

As we've seen so far, for traditional z/OS applications the interface is not separate but defined in source via a shared interface definition, generally a copybook or include. This source must be included in each program build for them to be able to interact. With this information, an application can be defined by two main components: shared interfaces that are used to communicate with other programs and the actual implementation of the programs.

It is important to note that shared copybooks could be shared not only within an application but across programs, or across applications. The only way other programs or applications can interact with the program is by including the shared interface definition. A z/OS load module does not work like a jar file, because it is does not expose interface definitions.
As applications communicate, their implementation consumes the public interface of the applications with which they interact. This concept of a public interface is common in Java programs and the way the communication between applications is defined. This principle can also be applied to existing COBOL and PL/I programs to help explain the structure required for a modern SCM.
CROSS-CUTTING INTERFACES

There are additional capabilities that might need to be shared in addition to sets of data structures for application communication. These capabilities might include standard security or logging functions and can be considered cross-cutting interfaces. These capabilities may be developed once and then included in many different programs. It would be very helpful if these additional included capabilities could also be handled as shared components with their own application lifecycle. The challenge comes when these components change in a non-compatible way. These types of changes are generally infrequent but might be needed at times.
In the preceding sections, we have laid out some of the key factors when considering the source code of traditional mainframe applications. The environment generally consists of many different applications that can provide shared interfaces and could consume shared components, or cross-cutting interfaces.

The knowledge of these factors and their respective lifecycles can guide the desired structure of source files in the SCM. Several patterns are possible to provide appropriate isolation, but to also provide appropriate sharing based on different requirements.
SCM scenarios

DEVELOP AT APPLICATION SCOPE, THEN INTEGRATE AND ADOPT NEW INTERFACES

First we consider the applications that provide interfaces and rely on other applications interfaces. Each application has its own lifecycle of changes. In this process changes are made, the changes are stabilized, and then go to production. In the end, there is a single production execution environment where all the applications will run. The applications must integrate but the most common breaking point is their interfaces. If an application’s interface changes in a non-backward compatible manner and the other applications do not react to this change (to at least recompile the modules affected by the interface change), then the production application will break. Therefore, most changes are implemented in such a way so that the structure of the data that is shared is not changed, but filler fields are used, or data is added to the end where only applications who need to react to the change are required to respond.

There are several types of changes that can be made to an interface. The major types are:

- **A backward compatible change.** The interface changes, but the change is done such that all applications do not need to react. The process of reacting to changes is called adoption. The applications consuming the interface are referred to as consumers. A typical example is when a field is added to a data structure, but this field does not have to be handled by most applications except a selected few, which requested this new field.

  Usually, fillers are used in a data structure and when people declare a new field, the overall structure of the data structure stays the same. Only the consumers interested in the new field need to adopt by using the new element in the interface. The others should use the new interface to stay current but don’t need to adopt it at this point. A standard practice is to adopt the change at least for those programs which are changing and recompiling for other reasons. Many organizations don’t do this type of adoption until the change has made it to production, to be sure they won’t be dependent on a change that gets pulled from the release late in the cycle.

- **A breaking change.** A data structure has changed in such a way that the overall memory layout of the data is impacted. It could be because of an array in a data structure. There are other cases as well (such as changes in arrays or condition name changes also known as level 88). In these cases, all of the consumers need to rebuild, whether they have functional changes to their code or not.

Especially with breaking changes, a process needs to be defined for planning and delivering application interface’s changes. As previously described, this process is called the adoption process.

Today, the adoption process is often handled by an audit report generally run late in the cycle. However, having early notification and understanding of these changes helps improve and speed delivery of business function.
Let’s consider an example: the IT system of an organization is made up of several groups of applications. One of these groups of applications deals with customer information and registration while another one deals with accounts and inventory.

The application groups are managed by different departments within the company. Usually an application owner exists for each application or application group. This example also applies to companies who do not perform the development themselves and have outsourced the development to one or several IT companies. The company’s staff then receives, validates, and deploys the new versions.

For the next major release, there are several new features that need to be implemented in various parts of the IT system. At least two applications will be involved, each working on different features, but depending on each other.

Let’s imagine two application owners manage the set of applications in the figure below. Each application team has access to the source files (and repository) of a selected number of applications as well as any interface definitions they need.

![Diagram](image)

*Figure 15 Application teams develop new features by making changes to some of the applications of the IT system*

The applications are not independent of each other. In fact, both Application 16 and Application 21 consume the interfaces of Application 1. Also, Application 32 consumes the interface of Application 2.
Figure 16 Applications 2 and 16 consume interfaces of Application 1

The shared interfaces of Application 1 and Application 2 need to evolve to fulfill the business requirement. In this scenario we assume the modified interfaces are backward compatible. However, Application 16 would improve if it takes advantage of the new fields exposed by Application 1. Application 32 would also improve by using the updated interfaces of Application 2.

All changes are expected to be deployed to production at the same. For this example, we assume team Green is ready on time for shipping the change, but team Orange is not.

If team Green has adopted the latest version of the interface of application 1 in application 16 and team Orange cannot deliver its changes of application 1, there is a problem. The enhanced application 16 will need to be changed to fit with the old interface of application 1. This causes a delay and additional development efforts for team Green even though they were originally complete on time.

Such a situation is unfortunately common. To avoid such situations, development teams use the following process:

Application 21 consumes the new interface of application 1, because both are part of the delivery of team Orange. However, Application 16 does not use the new interface and postpones its adoption. It continues to use the previous one, which is currently running in production.

Application 32 only takes advantage of the new interface of application 2, as both applications are in the area of responsibility of team Green.
What we have described here is a multi-layered adoption process. This is a simplified example. It gets more complex when Application 1 interface changes are not backward compatible. In such a case, the adoption done by the multiple parties needs to be managed and coordinated (Application 1, 21, and 16 need to be deployed together), because only a single version of Application 1 will run in production.

ADOPTION PROCESS

Changes to public interfaces should be coordinated between the different applications. Various levels of coordination can be defined:

No process (aka immediate adoption)

As the change is made, everyone else is impacted immediately. This is the unfortunate reality of many organizations today. Developers can be hit by unexpected changes that break their code. To minimize the risk, developers start to develop and test in isolation before sharing the updated interfaces. With Git and DBB, they can develop in isolation, build within their isolation, and fully test before they integrate with the rest of the team.

Such a simple scheme is conceivable for interface changes in Application 2 of our example. But not across application teams like the changes in Application 1. Usually this scheme does not scale (for example it works for small teams, small set of applications only) and slows down development teams as they grow. It can be applied for application internal interfaces without impacts to other applications.
Everyone adopts at a planned date

Very similar to the first adoption pattern is the adoption at a planned date. A change to the interface is announced via a meeting, an email, or a planning tool. Probably an outline of the future version is made available with the announcement.

Everyone adopts at their own pace

The providing team publishes the new interface change. The teams using the interface can determine themselves the point of accepting the change to their application.

Whatever adoption method is selected, we need to ensure there is only one production runtime environment. On source code level, applications need to consolidate and reconcile development activities to ensure changes are not picked up accidentally before being ready to move forward. The most common method is to consolidate in the integration environment.

SCM LAYOUT SCENARIOS

Base scenario

We start with how an application repository can be organized before exploring a concrete scenario of how a shared interface is published to consuming applications.

For a better overview, we will first introduce the concepts used throughout the following pictures to explain the different scenarios.
The branch, being represented by a simple line, is highlighted if involved in a scenario.

The color coding of the Git repositories will be grey if it belongs to a dedicated team or blue when used for a commonly shared repository.

![Color Coding Diagram](#)

**Figure 19 Color coding**

We don’t explicitly distinguish between personal or central Git repositories for the color coding.

**Branch strategy**

Within the Git repository, the different configurations of the application source files are represented by branches. Remember when we described a configuration of source files that it includes all the source files at a given version of an application.

Git being a distributed SCM, each developer works in their own Git repository. These repositories can be synchronized between each other. A central Git repository is the central point of truth in an organization of repositories. This is the one that is the door to the delivery pipeline, it is generally backed up, and used for audit purposes. We will begin with the branch strategies of this central Git repository.

The simplest organization would be to have a single branch.
Figure 20 Introducing the Master branch

Tags can be set on any branch to indicate the status of that branch such as a stable version, a version currently in production, or the last version in production.

In this example the work proceeds sequentially. This simple configuration is used when there is a small team maintaining an application with a single always active state. (Developers work in their own branches, which are pushed to this central Git repository, and a pull request is issued to merge the capabilities in for deployment. These developer branches are deleted as soon as the pull request is completed, which is why this generally has one stable Master branch.)

A second branch is often introduced to allow the development activity to be isolated from the branch which represents the current production environment, i.e. Master.

Figure 21 Introducing the release branch

Over time, when the release branch is stable and its content validated, a merge is done to the Master branch. This process allows to slow down deliveries toward a release captured in the Master branch. When using Git for mainframe development, we recommend the Master reflects at any given time the contents declared for production.

For larger teams who are not continuously deploying to production, a more typical scenario involves three branches:
Through the Integration branch developers integrate and exchange their code. Unit test and early stage testing is performed from that branch content. The code is then merged into the release branch to prepare the release, perform more later stage tests (late state integration testing, final performance, and scalability testing for example). A tag on release is used to identify a release candidate.

The higher you are in the branch hierarchy, branches are expected to be more stable and changing less frequently.

In this schema you provide a level of isolation for development until a release is declared. However, development is still sequential: You cannot work on the next release in integration when the current one has been declared. Sometimes you want a part of your development organization to start work on the next release earlier, because stabilization does not require the whole team. In this case, an additional integration branch can be introduced to let development start earlier.
Some teams are set up so development is divided into maintenance and feature teams. Developing several features of the same application can be challenging within a single configuration, as each requirement can change the same parts for different purposes.

Through isolation, an individual developer can work more effectively and changes gain a level of stability before sharing them with the team. Git introduces the concept of short-lived topic branches. A topic branch is intended to provide isolation for the developer for working on a hotfix, a new feature, or a maintenance task.

Consolidation of the different topic branches happens most commonly on the integration branch, for example through a merge workflow like the GitHub pull request.

---

Additionally, development teams might need to demand several development branches to separate integration scenarios of different features under development. In this case a new branch level is introduced called development, which merges into the integration branch.

![Additional development branches](image_url)

*Figure 26 Additional development branches*
Team strategy

Git being a fully distributed source control management system and the requirements from financial authorities for auditability can sound contradictory. But there are many financial organizations that are successfully accomplishing the audit and security requirements with Git. The specific requirements need to be considered in your design of the infrastructure setup and SCM layout. Different implementations of Git are provided as cloud solutions like GitHub, GitLab or Bitbucket, or many can be run on premise as well. Depending on the selected implementation the team strategy can look different, but in all scenarios there will be a central Git server known as origin and personal repositories like the ones on the developer’s workstation.

Working in topic branches provides great isolation and flexibility to development teams, but also raises the need for integration and exchanging changes with the team, which is known as the merge process in Git. Git servers provide additional workflow support known under the term merge or pull request.

Master, release, integration, and development branches are shared branches, remaining active for a longer period. Topic branches have a short lifecycle. Developers create topic branches for each new work item and delete them after integrating the changes with the team.

To leverage the workflow capabilities provided by Git servers, personal branches are also stored on the central Git server.

Let’s imagine Bob receiving a new task. He first initializes a new topic branch within his local repository based on the configuration as well as the origin repository.
Bob can also backup his work on the central server in origin. When he is done with development and would like to integrate with his teammates, he initiates the pull request workflow process. Given that his topic branch is stored on the server, a teammate can easily review and approve his changes before merging them to the development branch. When integration is done, the topic branch is deleted. This avoids the continuous need to synchronize personal branches in personal and central repositories.

A second, highly unusual and not recommended approach - unless the branch is going to live for less than a day - is to keep topic branches just within the personal repository. A scenario could be a prototyping scenario, which should not be shared with the team yet.

For this scenario, Bob does not push his feature branch to the shared Git repository, he merges to the development branch in his personal Git repository and synchronizes the development branch with development in origin.

Moving towards the Master branch within the branch hierarchy, the merge workflow can require merging the branches totally or just selectively.
Figure 29 Merge commits from development to integration through pull requests

In case of merge conflicts, Bob needs to resolve them in his personal repository and then push them back to origin.

Content of branches

The content in all branches can look identical and is generally split into a folder structure separating the application source files in different folders, for example, for an application called App 1:
The above simplified structure is not the only possible organization of a branch. The key is to separate out the files into folders which hold related files and to provide an organization that makes sense to the development team. This simplified structure is showing a high-level split to be able to identify a shared interface separate from the internal implementations.

There are multiple possible strategies for identifying shared resources, but it is important to come up with a way these are identified such that the developer has an easy understanding of the impact of his change.

Clearly identifying the different file types will also be helpful when defining the build. Within the DBB sample provided at GitHub, you will find an example using file.properties which uses the file extension to map the files in the repository to the correct build scripts. At this point we will not discuss the actual build scripts in greater detail but will discuss that later.

Finally, we end up having several Git repositories, one per application. Our scenario is based on three branches: Master, Integration and Dev.
The proposed Git repository structure with a repository per application also addresses authorization concerns: that everyone has full access to the full codebase of an enterprise. Permissions are granted on repository level of the Git server.

However, applications depend on each other. Interfaces of applications are consumed by other applications and several versions of interfaces can be involved in the development lifecycle.

The next section describes several options to manage the workflow of public interfaces of applications. Choosing the right option for your application depends on many factors. We will outline the different strategies of how to manage the adoption process of a shared interface with regards to the already outlined adoption strategies. A breaking or disruptive change of the interfaces needs to be handled differently than a compatible change. In general, people avoid disruptive changes to the interfaces because it means also a potential rework for the consuming applications. However, support needs to be in place for these scenarios.

Option 1 describes one possible workflow of how a breaking change of a shared interface can be managed. We will consider the perspectives of the interface provider as well as the consumer. Option 2 outlines a simple workflow for interface changes which make sense if backward compatibility can be guaranteed.

INTEGRATION SCENARIOS OF SHARED INTERFACES

Option 1 – Manage the adoption process of a breaking change through a dedicated repository for shared interfaces

Scenario outline

Let's consider a scenario with multiple applications that are interconnected: they expose and consume interfaces from each other.

There is one Git repository per application. Each repository contains three branches: Master, Integration, and Development. Additional topic branches may exist but are not represented in the next schemas.

Figure 31 One Git repository per application

The proposed Git repository structure with a repository per application also addresses authorization concerns: that everyone has full access to the full codebase of an enterprise. Permissions are granted on repository level of the Git server.

However, applications depend on each other. Interfaces of applications are consumed by other applications and several versions of interfaces can be involved in the development lifecycle.

The next section describes several options to manage the workflow of public interfaces of applications. Choosing the right option for your application depends on many factors. We will outline the different strategies of how to manage the adoption process of a shared interface with regards to the already outlined adoption strategies. A breaking or disruptive change of the interfaces needs to be handled differently than a compatible change. In general, people avoid disruptive changes to the interfaces because it means also a potential rework for the consuming applications. However, support needs to be in place for these scenarios.

Option 1 describes one possible workflow of how a breaking change of a shared interface can be managed. We will consider the perspectives of the interface provider as well as the consumer. Option 2 outlines a simple workflow for interface changes which make sense if backward compatibility can be guaranteed.

INTEGRATION SCENARIOS OF SHARED INTERFACES

Option 1 – Manage the adoption process of a breaking change through a dedicated repository for shared interfaces

Scenario outline

Let's consider a scenario with multiple applications that are interconnected: they expose and consume interfaces from each other.

There is one Git repository per application. Each repository contains three branches: Master, Integration, and Development. Additional topic branches may exist but are not represented in the next schemas.
For the sake of simplicity, we have not introduced a release branch between Master and Integration. A release branch makes sense in several scenarios, but it also adds complexity to the process. We will have a specific discussion about adding a release branch and how to deal with it.

We introduce a dedicated Git repository to manage the published interfaces of the applications. Each application will publish in this repository its public interfaces. The repository will give access to published interfaces for the consuming applications.

The interfaces themselves are developed within the repository of the application by the owning application team. Keep in mind that in z/OS software development, interface specifications are exposed through Copybooks or Includes files and are required to build your application programs.

![Figure 32 A Git repository dedicated to published interfaces](image)

A single Git repository on the Git server is dedicated to all shared interfaces. The repository contains a copy of the Copybooks and Includes, but no implementation logic. The organization of the branch content is significant here: we manage the official version of the interfaces in one branch, and new versions to be adopted in another one. Each branch contains all the published interfaces of each of the applications, each in a dedicated folder.

The folder structure in a branch could look like this:

![Figure 33 Layout of the contents of the published interface repository](image)
Regarding the number of branches in the published interface repository, you might have noticed that it does not necessarily match the number in the application repositories. The Master branch contains the official versions of the interfaces. Integration is used as an intermediate branch, providing the ability for the teams to adopt the change of the interfaces within their application.

How does an application publish changes of the interface to the Published Interface repository?

*Publish from integration*

In our scenario, the integration branch is the one where the next release of the application becomes stable. The final build of a given program may happen at that stage. The team creates the packages to be tested and deployed in runtime environments from these binaries.

Given that the integration branch receives commits from development branches, the source code evolves quickly. But there are phases when new versions of the interfaces become mature and ready to be published. This could be, for example, in the last days of an iteration.

When the team considers the new public interfaces are ready to be exposed to other applications, then it triggers a specific Jenkins build. The purpose of this build is to extract the public interfaces of the Application and commit these files to the Published Interfaces repository in its Integration branch. The build identifies the files to process (the public interfaces) based on the setup of the branch content.

This publication build can be an option of larger Integration build. Triggering it may even be automated (for example, when the application has been validated in QA), but it may not be appropriate that every integration build publishes the interfaces.
Publish from Master

When the release is declared or will be deployed to production, the Master branch within the application repository will be updated with the commits of integration. It is now time to publish the public interfaces as the new official version.

An automatic build can be triggered when commits arrive in the Master branch of the application repository. Similarly, it will extract the source files (public interfaces) of the Master branch, and commit these files, but this time to the Master branch of the Published Interface repository.

There is now a question about the content of the integration branch of the Published Interface repository. In the following picture, we propose to update both the Master branch and the integration branch with the public interfaces of Application 1. In more advanced cases (for example, parallel development, commit of a subset of integration), a more complex management of the integration branch is needed. We won’t discuss it here.

![Diagram](image-url)
How to consume other applications’ interfaces

So back to our scenario: there’s a need for each application to consume the published interfaces provided by other applications or cross-cutting components.

We have seen there are various ways to deal with adoption or consider changes from other applications’ interfaces that my application is consuming. One adoption strategy is to adopt at my own pace. It means the team decides to obtain updates of interfaces at a given time and remain with this version for a given period, for example, an iteration. The team therefore works in isolation during that time.

To achieve isolation, the team needs to stay stable with a given configuration of the published interfaces. This can happen either through the use of a Git submodule within its own configuration or by using a clone of the Published Interfaces repository: the team controls how often the submodules or clone get synched with the origin. Please keep in mind at one point within the lifecycle the application team will need to integrate and build with the officially published interfaces.

Managing its own version of the published interfaces should only be used for the development and application internal integration phases. This provides full independency from other development activities.

Let’s now consider the usage of the consumed interfaces through the build: we have seen there are two published versions of the interfaces in two different branches; let’s discuss which one should be used, and when. But before doing that, let’s clarify the builds in our system.

Figure 35 Update master branch in common through merging
Overview of the builds
There is one pipeline build per branch for each application repository, except the Master branch, which reflects the contents of the production environment.

Figure 36 Builds related to branches
For the sake of simplicity, we’re not considering Topic branches here.

Let’s see how each build behaves with the public interfaces it consumes. There are a set of choices and scenarios, primarily depending on the phase you are in with your development.

At one point in development you are going to produce the official production binaries. Most of the changes will be independent of other applications, but as described in the intro of this section, there are requirements with dependencies across application boundaries.

We also have seen that in development phases, isolation might be needed. To provide the highest level of isolation, the application team manages its own version of the published interfaces. In the next phases, it will also need to integrate and rebuild using the official version of the published interfaces, when it proceeds towards production to produce the official production binaries.

The following schema gives an overview of the possibilities we are considering for each build:
Two versions of the published interfaces are available: the version from Master and the version from Integration. There is a mechanism for full isolation available in development.
Development build
Several scenarios are possible for the development build, based on the requirements of the application and the phase in the development cycle.

Within the early development phase, the team may use the published interfaces of the integration branch within the development pipeline build, either from the official repository, or from their clone if they pursue full isolation.

![Diagram showing development build consumption of interfaces from integration branch](image)

**Figure 38 Development build consumes interfaces from integration branch**

In later phases, or if the application does not require early versions of new interfaces, the team may use the official production version of the published interfaces of the Master branch. Using a clone that you synch explicitly with the Published Interfaces repository is still a possibility, but it is not a requirement anymore as the Master is meant to be more stable in time.
Integration build
Let's now consider the integration build. In our scenario, the integration build is where the final build of the binaries happens. It is therefore important that the version of the public Interfaces that is used is compatible between consumer and producer. The integration build either consumes the version of the interfaces from Master or from integration. The latter case might be caused by the adoption of a breaking change that needs to be managed: several applications need to go to production together.
Figure 40: Integration build consumes published interfaces from Master or Integration branch

You’ll notice that the Interfaces provided by the Application 1 Integration branch are grayed in our picture. We want to highlight the build uses two different sets of files for the interfaces of Application 1. The ones from Application 1 repository and the ones from the Published Interfaces repository.

When there are no breaking changes to the interfaces, it is safer to build with the published version of the interfaces. It ensures provider and consumer are in sync. When there are breaking changes that will be published to Master when the release is declared, the build needs to use the version from the Application repository. Other consumers need to adopt the new interfaces.

More advanced case: Release Build

So far, for the sake of simplicity, we have not introduced a Release branch. But having such a branch is typical in Git. Let’s discuss its purpose and the implications in term of build.

Our workflow so far is focused on creating as early as possible binaries that are candidates to be deployed. Binaries from development branch are not candidates to be deployed. It’s not guaranteed that they use the correct version of the interfaces, they have not been validated, and at best unit tested. The build options in development offer maximal debug capabilities.
The integration build however can be setup to produce such binaries. The build options are optimized, and the binaries have passed initial tests in development phases. A package can be created to be deployed in different environments. If the package meets the criteria, it could be declared as being ready for production.

Some organizations may introduce a Release branch to dissociate the work of integration, still considered as an extension of development, from the work of creating candidates for production. People would typically do that when the development work has been isolated for a long period of time: one or several new features may have required several iterations to be ready for integration. In this case, the integration involves more work. Maybe unit testing has to be performed again.

It’s the release branch that will therefore contain valid and stable content. The release branch would be the one to publish new interfaces. It would be automatic.

Figure 41 Release branch publishes interfaces to Integration
If a release branch is introduced, then the release build is the one that uses the interfaces from Master. The integration build either uses the interfaces from Master or from Integration. Development targets either the interfaces from Master or Integration.

![Figure 42 Interfaces consumed for each build](image)

**Option 2 – Manage the adoption process through publishing shared interfaces to the file system**

Scenario outline

In this scenario, interfaces are still published and consumed as we described in the previous scenario. But instead of using a Git repository to store and version the published interfaces, we use the file system, either on USS (with a dedicated set of folders) or MVS (with a dedicated set of PDSEs). The principles of publication and consumption remain the same.

Publishing interfaces just to the file system has many disadvantages, specifically when it comes to traceability and auditability. You can no longer prove which version of an interface you used in a specific build. This option is most commonly used for interfaces totally out of your control and are updated infrequently, such as system includes. The final build of your programs must be auditable.

The most common use for this is in a migration scenario. The file system is used in the transitioning phase to Git to enable building applications which reside either in the old SCM or Git. The files still managed by the old system are used in Git via the file system until everything is moved over.
How does an application publish changes of the interface to the shared file system?

**Publish from integration**
Like the previous option, a publication build is defined for Integration. It’s the team that decides when to trigger it, for example when the interfaces are ready to be published and exposed to others. The interfaces are extracted by the build and written to the z/OS file system. It can be either on USS or on MVS in a PDS library. The build generates a manifest that stores the configuration hash of Git and allows identification of the file versions being used.

**Figure 43 Publish application interfaces in file system (Integration location) explicitly**

**Publish from master**
When changes are deployed to production, the changes are merged into the Master branch of the application repository. A build triggers automatically when a commit occurs in the Master branch; it retrieves the public interfaces and puts them on the file system. The location containing the productive version of the shared interface is updated along with the manifests.
How to consume other applications’ interfaces

For this scenario, consuming applications cannot decide when they are going to adopt the changes. In fact, they are impacted as soon as a team publishes the changes. What the build can decide is whether it consumes the official version of the interfaces (Master) or the new ones (Integration).

It’s the responsibility of each application to test and verify modifications of the shared interfaces before they are published to the file system.

In this scenario, application 2 consumes interfaces from application 1. The build consumes the published interfaces from the file system: either PDSE or from HFS folders.

Development build

For the development build, the build can be configured to either reference the Integration or the Master folders of the published interfaces.
Integration build

Application 2 can decide if it would like to use the production version or the version from integration of the interfaces for the integration build. The team needs to ensure the final build is either be compatible with the production version of the consumed common interfaces or make sure it is going to production with or after the providing applications.
Due to the limitations of this option, we will highlight another alternative. It is close to publishing to the file system but has more auditability and flexibility with regards to the adoption process.

Option 3 - Managing the adoption process through publishing shared interfaces to an artifact repository (e.g. Artifactory or Nexus)

Scenario outline

In this scenario, interfaces are still published and consumed as we described in the previous scenarios. The difference with the previous ones is that we publish the interfaces as a package in an artifact repository. Publishing to an artifact repository has a set of advantages compared to just publishing to the file system. Additionally, it can serve all described adoption processes.

How does an application publish changes of the interface to an artifact repository?

Like the previously described scenario, there is a publication build which stores the shared interfaces of the application’s Master and Integration branch within an artifact repository. The artifact repository handles two types of interfaces:

- the interfaces which are stable, but not used in production, stored in the release branch
- the version of the interfaces in production, represented in the Master branch.

What is really happening here is the application interfaces are treated as deliverables. This is a deliverable not deployable to production, but nevertheless a deliverable. This deliverable is used to provide interfaces to consuming applications.

Publish from integration

In this scenario a publication build produces a package containing the interfaces of the application, triggered explicitly by the team.
**Publish from Master**

As soon as the application updates the Master branch (for example, when the release is declared), a publication build shares the updated version within the artifact repository.
How to consume other applications’ interfaces

As mentioned in the introduction of this section, there are several scenarios that can be implemented. There is no golden rule. The diagram below provides an overview of how the build can be configured. It’s important to remember that the final build needs to run against the production version of the interface if it’s a breaking change, or a reasonable workflow must be created depending on how changes are promoted to production in a consolidated manner.

Overview of the builds

Instead of using the equivalent names of the branches, we distinguish into the official production version and a stable version of the shared interfaces. These can be tags or different folders within your artifact repository.

For highest isolation of development activities, for example a long-term project, the application team can build its own configuration of the published interfaces to their dedicated file system. The team cherry-picks from the artifact repository and retrieves the versions to a library with which they work.

When moving towards production, the build needs to happen against the official version of the shared interfaces. The picture below shows the different possible configurations:

![Diagram of builds consuming interfaces published through an artifact repository](image)

**Figure 49 Builds consumes interfaces published through an artifact repository**

*Development build*

The development build can point to different versions for the shared interfaces. A customized set of them makes sense for a high degree of isolation, but it must be recompiled before a version that might go to production to avoid introducing a risk that the interfaces don’t match.
Integration build

The integration build can either use the latest published interfaces stored in the artifact repository or another version it is adopting. The build retrieves the shared interfaces from the artifact repository as part of the build job.

Figure 50 Integration build consumes interfaces published through an artifact repository
CONCLUSION

In this chapter, we explained how you can organize development within a Git repository, but also across repositories. We discussed adoptions requiring a process and some management to be handled efficiently. Then we looked more in detail at different schemes to deal with these adoptions.

We have seen many combinations and different flavors in the proposed workflows. Our purpose is to trigger a discussion about the option best for you. You might start with a given option and evolve later towards a different one. A process is designed to evolve over time based on experience and feedback.
Build scenarios

Build contains all of the steps to compile and link the source files into executable code. This paper does not get into all the fine details of the Groovy scripts used as part of Dependency Base build, instead it focuses on the different build strategies for the build scenarios and points to the relevant sections within the shared sample scripts via GitHub.

There are several different types of possible builds. A user build, a build done by a user on a single program, and a pipeline build, which will build the single or multiple changes pushed to the Git server.

Due to the nature of mainframe languages, source files need to be associated to a build type. For the build process it is important to know what type of processing the file needs, so the right build process can be associated with it. The correlation can be tracked through a fixed mapping list, through a folder organization, or a file extension.

Within the IBM Dependency Based Build samples on GitHub, a mapping file file.properties provides the mapping of the file to a certain build script.

![Figure 52 Mapping files to build script](image)

**USER BUILD**

In an early development phase, developers need the ability to build the single program they are working on, as well as the unit test programs being created.

IBM Developer for z Systems provides an option to enable the user to compile a selected program in an easy and fast way, without the need to commit or push the change to the repository, while still using the

---

7 https://github.com/ibmdbbbdev/Samples

© IBM Copyright, 2018 58 / 69 Version 1.0.1, Oct 15th, 2018

Develop Mainframe Software with Opensource Source Code Managers and IBM Dependency Based Build

http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102772
same build options defined for the pipeline build. The purpose is to build fast for the personal testing scenario.

![Diagram of application 1 and pipeline build]

**Figure 53 User build**

Additional material can be found in IBM's Knowledge Center for IBM Dependency Based Build and IBM Developer for z Systems.8

**PIPELINE BUILD**

A pipeline build is a build generally of a set of changes, but could also be a single change, that have been pushed to the Git server. It produces the official binaries, outputs that can be packaged and deployed to different environments, including production. By having the clear definition of what went into each build and the official build outputs, this ensures there are audit records.

**Scope of pipeline builds**

The pipeline can kick off builds for many different scenarios, providing different overrides for each. The build scripts need to handle all these scenarios.

A pipeline build is also possible for the short-lived topic branches. However, the produced binaries cannot go to production because they typically lack a review and approval process.

---

Using a pipeline there are multiple possible build scenarios including a full build, a dependency build, and a scoped build.

**Full build**

A full build compiles the full defined configuration. The full list of source files is provided as input from the pipeline to the build scripts.

One option is to allow the specification of the build type in the Jenkins pipeline as shown in Figure 54. The build script would then need to handle this input as the build type.

**Dependency builds**

Instead of a full build, most builds are dependency-based builds, only building the changed files and the files that depend on those files. However, there are different qualities of defining the scope of a dependency-based build, which is reflected in the impact calculation phase of a build.

**Impact calculation for dependency-based builds**

The easiest strategy is to only build the modified files. How to calculate this depends on the scenario. If it is a topic branch build, this can simply be the modified files identified in the Git commit. If this is a release build, you can use a diff of the current baseline to the previous successful build baseline to calculate the modifications.

The next consideration is if you want to include the directly impacted files in the list of buildable files. This calculation for direct dependencies occurs on the source file level. If a copybook is changed, all files using this copybook are directly impacted. This level of impact calculation is managed by the IBM Dependency Based Build server. The server contains a collection representing all direct dependencies in a configuration. Your build script needs to query the dependency-based build server to retrieve the list of impacted files and pass this along to the build script. The use cases can obviously vary. If you want to automatically rebuild...
programs including a modified copybook, you should pursue the strategy above. Instead of rebuilding all dependent programs, you might want to provide a function for the user to select which, if any, of these impacted files should be rebuilt.

The impacts.groovy sample script for IBM Dependency Based Build in the public GitHub repository provides an example of this impact calculation.

```groovy
println("Searching for programs impacted by changed file \$changedFile")
def resolver = tools.getDefaultImpactResolver(changedFile)
def impacts = resolver.resolve()
impacts.each { impact ->
def impactfile = impact.getFile()
  // only add impacted files that have a build script mapped to it
  if (ScriptMappings.getScriptName(impactFile)) {
    println("impactfile is impacted by changed file \$changedFile. Adding to build list.")
    buildList.add(impactFile)
  }
}
```

**Figure 55 Calculating direct dependencies**

An even more sophisticated impact calculation needs to be implemented for indirect dependencies. On the source level, when a build output of a file is used as an input to build another file, we consider an indirect dependency for the two source files. A mainframe scenario would be a CICS map (BMS), which produces an include file during build as an output, which is again an input to build the relevant program covering the business logic. For calculating indirect dependencies, the build administrator needs to consider the defined outputs and their traceability.

For the example of the indirect dependency between the BMS map and the program for the business logic, the calculation is almost trivial, because the assembly of the map produces a copybook with the exact same name. The build script adds a rule for the business logic programs to look for dependencies within the structure of BMS maps. Instead of tracking the intermediate relationship between the business logic program to the Copybook and from there to the BMS map, a direct dependency is set up between the business logic program and the BMS map. This is shown in the tools.groovy script shipped in the samples.
In addition to the dependencies at the source file level between programs and copybooks, we can have static dependencies between a main and subprograms when control is passed between the two programs through a static call. This requires that both programs are assembled into a single load module. This technique is known as static linkage, when several compiled objects are assembled to a single executable. Alternatively, modules can be dynamically called and are fetched from the available concatenation during execution.

When changing the subprogram, the build process needs to understand the dependency to the main program, compile the subprogram, and then bind the main and the subprogram to a new load module. With running in autocall mode, the linkage editor identifies the object files automatically and assembles the load module, but nevertheless the build process needs to be aware of the dependency and trigger the rebinding of the load module.

Dependencies on an object level can be retrieved from scanning the load module itself through the LinkEditScanner API within the IBM Dependency Based build toolkit. The LinkEditScanner provides information about the object decks which are bound to the load module and returns a list according to the dependencies on the source level. It is executed as an additional step after the compile + link steps of the main program to update the dependency data. To separate and avoid overriding dependencies in the collections, a new collection should be established for dependencies on the output level.
Figure 57 Collect and store link dependencies

Within the impact calculation phase, now the link dependencies can be included with the dependency resolver.

Figure 58 Add output dependencies to dependency resolver
User-defined build scope

While dependency-based builds cover strategies with a good level of automation for the build process, the user-defined build scope provides full flexibility to the user. It is up to the user to provide the list of files being considered in the build. The list could also reference a work item and its related changes. As a variation, the list can include the files from where the impact calculation should be performed.

A file containing the build list is stored as part of the repository.

Like the build scripts, we recommend storing the user-defined build list as part of the Git repository. Additionally, you need to consider a parameterized Jenkins job including the relevant logic, which can serve all three build types: full build, dependency-based build, and user-defined build.

Strategies for orchestrated builds

As we now have shown, there are different possible build scenarios. It is important to select the right build types for your setup. Using a mixture, depending on the level, should serve your needs. At each given level the choices will be limited, with the most flexibility at the development level. If you intend to maintain the traditional library population strategy, you can implement user builds to build development activities within the development stage and set up a dependency-based build in the integration stage to ensure consistency across the application.

It is important to recognize by moving to Git and using an artifact repository for outputs, you are no longer tied to a library structure. You can now use the same build configuration strategy while pulling any already-built parts out of the artifact repository.

BUILD STRATEGIES IN RELATION TO DISCUSSED INTEGRATION SCENARIOS

The following section maps out the implementation of the different build scenarios.
Build with a dedicated repository for common components

The following build setup - a Git repository for the scope of one application combined with a cloned repository of the common components - has the following characteristics:

- Ability to capture all dependencies and impacts
- Changes in the common repository for shared interfaces can trigger a rebuild of applications
- Impacts are easily determined through a collection in IBM Dependency Based Build server
- Handling two Git repositories in the Jenkins and build script setup
- Source code changes can be easily retrieved from the Git commit
- Through the SCM layout all defined build scenarios are supported

Integration build

Different strategies exist to manage your version of the common repository. Two of them include: managing your own through a clone of the repository or using Git submodules.

Build with interfaces shared to file system

Keep in mind that the pattern of a build using shared interfaces made available through the file system is addressing non-breaking interfaces changes. This affects the characteristics of this build scenario: a changed interface does not necessarily lead to a rebuild of consuming programs. Therefore, the characteristics of this build strategy are in this case:

- Changing a shared interface does not trigger a rebuild
- Impact calculation is performed primarily at the scope of the applications branch within their Git repository
CONCLUSION

In this chapter, we explained different build strategies and options. Like in previous chapters, our goal is to trigger a discussion of which fit is best for you. With the transformation there is a high potential to improve workflows and align them to your DevOps journey.

The build setup needs to be crisp, easy, and understandable by the development team. Therefore, we advise you limit the number of options available to your team and implement one setup which fits all.
What’s coming in part two

We’re reaching the end of this document but there are many topics we have not yet addressed. Our purpose here was to introduce SCMs and explain how they differ from Library Managers. In addition, how we can take advantage of their characteristics to evolve our development cycles into a DevOps cycle using iterations, parallel development when needed, controlled adoptions, and more frequent releases. We discussed how Git and DBB in combination with Jenkins and artifact repositories can be used in this transformation. We plan to publish a second part which will cover additional scenarios such as emergency workflow, the creation of packages to be deployed, as well as deployment itself. We will also address strategies of capturing and managing metadata needed to drive sophisticated pipeline builds and deployments.

We hope this document has helped you begin to understand how to prepare for your own transformation.
## Appendix

### TABLES OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Workflow example, working with a library manager</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Pros and cons of working with library managers</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Working in configurations</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>Parallel release schedules</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Working in isolated configurations</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>Merging the different configurations</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>Simplified Git workflow</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>Mainframe staging hierarchy</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>Isolated development branches</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>Producing object code and executables</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>Transfer of control sharing data</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>An application exposes a public interface</td>
<td>21</td>
</tr>
<tr>
<td>13</td>
<td>Applications consume public interfaces of other applications</td>
<td>22</td>
</tr>
<tr>
<td>14</td>
<td>Interfaces used by applications</td>
<td>23</td>
</tr>
<tr>
<td>15</td>
<td>Application teams develop new features by making changes to some of the applications of the IT</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>system</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Applications 2 and 16 consume interfaces of Application 1</td>
<td>26</td>
</tr>
<tr>
<td>17</td>
<td>Applications 2 and 16 consume interfaces of Application 1</td>
<td>27</td>
</tr>
<tr>
<td>18</td>
<td>Legend of used shapes</td>
<td>28</td>
</tr>
<tr>
<td>19</td>
<td>Color coding</td>
<td>29</td>
</tr>
<tr>
<td>20</td>
<td>Introducing the Master branch</td>
<td>30</td>
</tr>
<tr>
<td>21</td>
<td>Introducing the release branch</td>
<td>30</td>
</tr>
<tr>
<td>22</td>
<td>Introducing the integration branch</td>
<td>31</td>
</tr>
<tr>
<td>23</td>
<td>Update cycles of branches</td>
<td>31</td>
</tr>
<tr>
<td>24</td>
<td>Temporary integration branch to start new development earlier</td>
<td>32</td>
</tr>
<tr>
<td>25</td>
<td>Topic branches</td>
<td>32</td>
</tr>
<tr>
<td>26</td>
<td>Additional development branches</td>
<td>33</td>
</tr>
<tr>
<td>27</td>
<td>Develop locally and integrate on the central server</td>
<td>35</td>
</tr>
<tr>
<td>28</td>
<td>Use local Git repository for feature development</td>
<td>35</td>
</tr>
<tr>
<td>29</td>
<td>Merge commits from development to integration through pull requests</td>
<td>36</td>
</tr>
<tr>
<td>30</td>
<td>Folder structure in Git for an App1</td>
<td>37</td>
</tr>
<tr>
<td>31</td>
<td>One Git repository per application</td>
<td>38</td>
</tr>
<tr>
<td>32</td>
<td>A Git repository dedicated to published interfaces</td>
<td>39</td>
</tr>
<tr>
<td>33</td>
<td>Layout of the contents of the published interface repository</td>
<td>39</td>
</tr>
<tr>
<td>34</td>
<td>Update integration branch in published interfaces repository</td>
<td>41</td>
</tr>
<tr>
<td>35</td>
<td>Update master branch in common through merging</td>
<td>42</td>
</tr>
</tbody>
</table>
Figure 36 Builds related to branches.................................................................43
Figure 37 Builds relating to official or clone repository containing the published interfaces .....................44
Figure 38 Development build consumes interfaces from integration branch ....................................45
Figure 39 Development build consumes published interfaces from Master branch .............................46
Figure 40 Integration build consumes published interfaces from Master or Integration branch ............47
Figure 41 Release branch publishes interfaces to Integration ..................................................48
Figure 42 Interfaces consumed for each build ..................................................................................49
Figure 43 Publish application interfaces in file system (Integration location) explicitly ..................50
Figure 44 Publish application interfaces ("Integration" and "Master" location) when commit to Master .. 51
Figure 45 Development build consumes published interfaces from the filesystem ..........................52
Figure 46 Integration build consumes published interfaces ..................................................................53
Figure 47 A publication build stores a new version within an artifact repository .............................54
Figure 48 A publication build stores a new version within an artifact repository .............................54
Figure 49 Builds consumes interfaces published through an artifact repository ..............................55
Figure 50 Integration build consumes interfaces published through an artifact repository ...............56
Figure 51 Integration build consumes interfaces published through an artifact repository ...............57
Figure 52 Mapping files to build script ..........................................................................................58
Figure 53 User build ....................................................................................................................59
Figure 54 Jenkins build parameterization .....................................................................................60
Figure 55 Calculating direct dependencies ....................................................................................61
Figure 56 Calculating indirect dependencies for BMS map to program ..........................................62
Figure 57 Collect and store link dependencies ..................................................................................63
Figure 58 Add output dependencies to dependency resolver .........................................................63
Figure 59 User-defined file list ........................................................................................................64
Figure 60 Build workflow with dedicated repository for common components ..............................65
Figure 61 Build workflow with shared interfaces provided through file system ...............................66