January 2011

A TOOL FOR OPTIMIZING SOFTWARE BUILD AND RELEASE PROCESS

Byron E. Rodriguez  
Worcester Polytechnic Institute

Han Li  
Worcester Polytechnic Institute

Jaymin Rajen Mehta  
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation  
Retrieved from https://digitalcommons.wpi.edu/mqp-all/3010
Build and Release Optimization of the Structured Credit Desktop

A Major Qualifying Project Report

Submitted to the Faculty of WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirement for the Degree of Bachelor of Science

Submitted on
Monday, December 20, 2010

Project Sponsor: Bank of America – Merrill Lynch

Submitted to:

On-Site Liaisons: Chris Rice (Principal; Capital Markets Credit)
Ron Toam (Vice President; Capital Markets Front Office)
Jin Lee (Assistant Vice President; Capital Markets Front Office)
Michael San Jose (Assistant Vice President; Capital Markets Front Office)

Project Advisors: Professor Daniel Dougherty, Department of Computer Science, WPI
Professor Arthur Gerstenfeld, School of Business, WPI

Submitted by:

Han Li, Computer Science
Jaymin Mehta, Computer Science
Byron Rodriguez, Industrial Engineering
Abstract

The Structured Credit Development team at Bank of America sought to optimize its software build and release process. Previously, this process required several extensive and manual steps, which was time consuming and greatly increased the possibility of human errors. After completing an in-depth analysis of this process, we proposed several future state designs. We then developed an application that automated the task of updating version numbers, and we also streamlined the management of different project configuration files.
Acknowledgements

We would like to thank Christopher Rice, Ron Toam, Jin Lee, and Michael San Jose for their guidance and support throughout the completion of this project. We also wish to express our gratitude towards other Bank of America employees: Sri Atluri, Scott Burton, Gaurav Ghatpande, Ekaterina Ratcheva, Ingemar Svensson, and the rest of the Structured Credit Development team. In addition, we would like to thank Professor Daniel Dougherty and Professor Arthur Gerstenfeld for their comments and suggestions, helping make our project a success.
Authorship

The development of the SCD Builder application and the written report were created with equal contributions from Han Li, Jaymin Mehta, and Byron Rodriguez.
Table of Contents

Abstract ............................................................................................................................... ii
Acknowledgements ........................................................................................................... iii
Authorship ....................................................................................................................... iv
Table of Figures ................................................................................................................ vii
List of Tables .................................................................................................................... viii
1. Introduction .................................................................................................................. 12
2. Background .................................................................................................................. 14
   2.1 Structured Finance .................................................................................................. 15
   2.2 The Structured Credit Desktop ............................................................................. 17
   2.3 Software Development Practices .......................................................................... 19
      2.3.1 Structured Credit Desktop: Development Framework ............................... 21
      2.3.2 Other Software Development Tools .............................................................. 22
3. Business Problem ........................................................................................................ 24
4. Implementation & Results ............................................................................................ 26
   4.1 Build and Release Process Analysis ...................................................................... 26
   4.3 Release Management .............................................................................................. 28
      4.3.1 Current Process .............................................................................................. 28
      4.3.2 Proposed Solution: Configuration Files Separation ..................................... 30
   4.4 Version Manager Application .................................................................................. 33
Table of Figures

Figure 1: Structured Finance ........................................................................................................... 16
Figure 2: Structured Credit Desktop .............................................................................................. 18
Figure 3: Structured Credit Development Group Build and Release Process ............................ 27
Figure 4: Build and Release Current Process .................................................................................. 29
Figure 5: Build and Release Process Impact .................................................................................. 33
Figure 6: Version Manager ............................................................................................................ 36
List of Tables

Table 1: Structured Credit Desktop Internal Applications ................................................................. 18
Table 2: Major.Minor.Patch Release Convention ................................................................................. 35
Executive Summary

The Structured Credit Development team is a global technology group that seeks to provide effective and efficient end-to-end technological service and support to the Global Banking and Markets division, specifically the Structured Credit Trading desk. As a technology team in the GMRT division, the Structured Credit Development team maintains and improves the stability, functionality, performance, and security of proprietary infrastructure and applications, such as the Structured Credit Desktop (SCD). The SCD application is a suite of applications that manages risk, market data, and profit and loss within the Structured Credit Trading business at Bank of America. To successfully maintain and improve the SCD application, the Structured Credit Development team tasked the WPI project team with optimizing its build and release process.

To optimize the build and release process, the Structured Credit Development team needed to modify the current development process to minimize wasted time, effort, and possibility of human error during the tedious task of updating version numbers, and also reduce risk when delivering new releases. Accordingly, we divided the optimization of the build and release process into three objectives: (1) analysis of the build and release process, (2) developing the SCD Builder application, (3) handover of SCD Builder to the Structured Credit Development Quality Assurance team. We divided the analysis of the Structured Credit Development team’s build and release process into two categories: (1) versioning management, and (2) release management.

“Versioning management” is the term we coined to refer to the process in which project configuration files of modified SCD applications are versioned. Before the SCD development team could build new iterations of the SCD, the team would appoint a release manager who,
among other things, manually compared changes in project files between the new and previous modules, and increment the version number if there were any differences. This was a long, manual and arduous process, potentially taking hours, that was prone to human error. In addition, there was no standard to the version numbering. For example, there may be several releases with the same version number, with only a label, such as “ER” (for an Emergency Release), differentiating them.

To address this issue, we developed a version management application. Not only does this application compare project files within the Perforce repository (or another specified path), but it also standardizes the versioning process by establishing the “Major.Minor.Patch” convention, in lieu of a Major.Minor and a text label such as “ER”, making it easier to keep chronological track of the releases.

“Release management” is the term we use to describe the actual build and release section of new SCD iterations. Once the release manager had updated versions, a developer executed an SCD build by using Maven, a Java software project management tool, to generate project Java Archive (JAR) files, packages of Java source files. During a Maven build, a release environment, which indicated the purpose of the release, needed to be specified. Next, the developer would execute another Maven command, building a single project Web Application Archive (WAR) file, a compressed package that contained the JAR files. Finally, the developer would deploy the WAR file on the appropriate server, and the SCD application would be accessible. However, there were procedural gaps that were highly inefficient, and left room for human error. The entire application content JAR file and WAR file were reproduced for each environment. Yet, the content JAR files were the same and the only differences were in the configuration files. Furthermore, when deploying to different release environments, developers needed to rebuild
project files. In between these different releases, source code could potentially be checked into the repository without proper testing, leading to the possibility of application corruption in the production release.

To address these gaps, we modified the build and release process to dynamically deploy different release environments by building one common WAR file, which contains the JAR content files and configuration files for all environments. In addition, we customized a JSP template, so that based on the input URL, a release environment will be accessed, and the appropriate project files will be downloaded.

We consolidated the version management and release management work streams under an application named “SCD Builder”. The SCD Builder contains the version management application, which can be executed through the command line. The application uses a customizable configuration properties file, which holds parameters and values. Through the command line arguments, the application executes the specified Maven build commands to integrate the release management process. Finally the product of the build and release process, a single WAR file, is available and can be configured for any desired environment from values in the configuration properties file.

The impact of the SCD Builder is summarized in four categories: process changes, time efficiency, risk mitigation, and human error. The build and release process was initially a manual and error prone process, which we standardized and documented. In addition, we reduced the version process from a multiple hour endeavor to take less than two minutes. Our modifications to the build and release process reduces code exposure, while enabling different release types of the Structured Credit Desktop application to be deployed dynamically without corruption to the quality assurance process.
1. Introduction

Bank of America’s Global Markets Research Technology (GMRT) division develops proprietary and non-proprietary software to support all sales, trading and research applications related to the strategic direction of Global Markets (Global Banking, Markets and Wealth Management Technology and Operations, 2010). Bank of America and other financial institutions rely on technology groups that internally develop applications to support their robust and complex trading businesses. The Structured Credit Development group, for example, develops and maintains the Structured Credit Desktop (SCD) – a robust suite of applications that support the Structured Credit Trading business at Bank of America.

To release new iterations of the SCD application, the Structured Credit Development group acts in accordance with an established build and release process. First, the designated release manager updated version numbers in pom.xml files – a lengthy and error prone process. Second, the team utilizes Maven, a tool used for building and managing Java-based projects, to create project files (JAR and WAR files) for the SCD’s core modules. It is important to note that JAR files are essential to the build process. JAR files contain pom.xml files that store release information and are interconnected with other project files, through parent and children dependencies. In the final step of the build and release process, the Structured Credit Development team decided to deploy the new release according to the region and environment (SCD Wiki, 2010). The team found it inefficient to rebuild JAR files for different release environments. Moreover, a production build would incorporate untested code, which was checked in after the Quality Assurance build was tested. Therefore, here, the rebuilding step invalidated and corrupted the Quality Assurance testing process.
As part of optimizing the build and release process, the Structured Credit Development group sought to enforce standard software development best practices, such as agile development and continuous integration, that the group previously employed. Agile software development is a philosophy of interactive and incremental software development methodologies (Agile Manifesto, 2001). Continuous integration instructs developers to integrate their work frequently rather than every time code is registered (Martin Fowler, 2006). Maven, utilized by the Structured Credit Development group, allows a project to build using its project object model and a set of plug-ins shared by all projects (Apache Organization 2010).

Optimization of the build and release process raised many unknowns, including how to efficiently extract the configuration files and dynamically deploy them. Additionally, the cumbersome versioning process was confusing to inexperienced release managers. Our solution needed to satisfy the following requirements: (1) embody agile development and continuous integration in its framework, and (2) empower release managers, in different geographic regions and different levels of development experience, to easily deploy new releases.

Our implementation will not only satisfy the outlined requirements, but also leverage software development best practices. In order to create this tool, we observed and evaluated the current build and release process. Additionally, we held several design discussions with members of the Structured Credit Development to gauge the direction and ultimate implementation of the application.
2. Background

Bank of America is one of the world's largest financial institutions, serving individual consumers, small and middle market businesses, and large corporations with a full range of banking, investing, asset management, and other financial and risk-management products and services. The company provides unmatched convenience in the United States, serving more than 59 million consumer and small business relationships with more than 6,100 retail banking offices, more than 18,000 ATMs and award-winning online banking with more than 25 million active users. Bank of America offers industry leading support to more than 4 million small business owners through a suite of innovative, easy-to-use online products and services. The company serves clients in more than 150 countries and has relationships with 99 percent of the U.S. Fortune 500 companies and 83 percent of the Fortune Global 500.

As a leader in wealth management, private banking, retail brokerage, and a leading provider of global corporate and investment banking services, Bank of America deals in structured finance. The Structured Credit Desktop (SCD) application primarily supports the Structured Credit Trading (SCT) business. Some of the financial products that the Structured Credit Desktop application supports are listed and defined below:

- Credit Default Swaps: A financial credit derivative that is helpful when one party who wants to hedge credit risk pays a fixed payment on a regular basis, in return for a contingent payment that is triggered by a credit event, such as the bankruptcy of a particular firm or the downgrading of the firms credit rating by a credit-rating agency (Mishkin, 2009, page. 353)

- Credit Default Swap Index: standardized credit security used to hedge credit risk or take a position on a basket of credit entities. (Investopedia, 2010)
• Interest Rate Swaps: Interest-rate swaps are derivatives, financial instruments derived from stocks, bonds, loans, currencies and commodities, or linked to specific events such as changes in interest rates or the weather. (Bloomberg, 2010)

• CDO²: A special purpose vehicle with securitization payments in the form of tranches, a CDO² is backed by a pool of collateralized debt obligations tranches. Similar to CDOs except that assets backed are not a pool of bonds, loans, or credit instruments. Instead CDO² are backed by CDO tranches, in other words their purpose is to allow banks to resell the credit risk taken in CDOs. (Investopedia, 2010)

Structured finance provides financial institutions, such as Bank of America, with innovative tools for mitigating risk. However, transactions incurred by financial institutions are often extremely complicated, and entail federally regulated pricing and risk management models.

2.1 Structured Finance

Structured finance is a practice of transferring a subset of a company’s assets into a bankruptcy-remote corporation, or other special purpose vehicle. The Securities Exchange Commissions defines structured products as “investment products that derive their value by reference to the price or value of an underlying asset (security index, currency, and commodity) or combination of assets” (New Straits Times, 2009). The recent growth in issuance volumes of these products highlights the importance of structure finance as a tool for credit risk transfer and a vital instrument that maintains the profitability of the structured credit trading desks.

Structured finance instruments, such as special purpose vehicles (SPV), serve as tools to mitigate credit risk and transfer it across financial institutions. Graph 1 depicts the key market participants. Note that SPVs pool assets, which derive their value from the underlying securities, generate tranches or classes of securities whose rating is higher than the average rating of the
underlying collateral asset pool. In other words, prioritized tranches (often classified as senior, mezzanine, and junior) create securities with different risk return profiles. Thus, the credit support resulting from the different prioritized tranches means that the most senior claims are expected to be somewhat protected (except in extreme circumstances) from the default risk by the losses of lower tranches (Fender and Mitchell, 2005, page 69). These SPVs and their different prioritized tranches are a quintessential example of a financial instrument that distributes risk among other participants in the financial market.

Figure 1: Structured Finance

Technological innovation in the finance industry has led to a reduction of transaction costs. More importantly, technology has enabled securitization, or the bundling together of smaller loans (such as mortgages) into standard debt securities (Mishkin, 2009, page 201). This has enhanced the ability to cheaply bundle and quantify the default risk of asset-backed securities. At Bank of America, the Structured Credit Development team supports the Structured
Credit Desktop application, an industry example of technology that drives modern financial transactions.

### 2.2 The Structured Credit Desktop

With approximately 150 internal clients (SCD Presentation, 2009), the Structured Credit Desktop (SCD) primarily supports Bank of America’s Structured Credit Trading (SCT) business. Some users of the SCD are Front Office (Sales, Trading, Structuring and Marketing teams), and Middle Office (Trade Support and Risk Verification teams). The SCD application occasionally has other businesses use some of the functionality as well, such as Prop/Distressed. Below, Figure 2 depicts the SCD application, which sits at the desktop of traders from the Bank of America’s Structured Credit Trading business.
Front office, middle office, Finance, and Risk Management use the Structured Credit Desktop toolset to manage, verify, and price structured credit trades (SCD Presentation, 2009). In addition, through SCD, Bank of America management can search for data at a high level and drill down to trade level, and even view both of these on a global medium. Table 1 below describes different sub-applications existing within the SCD that support the Structured Credit Trading business at Bank of America.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
</table>
| Base Correlation (BC) Marking Tool   | - Java application that sits within the SCD Desktop and excel spreadsheets which contain the raw data  
- The spreadsheets are updated daily by the Front office and the data is loaded into a DB and then computed to create Base Correlation Curve                                                                                                                                                                                                                                                   |
| Basket Manager (BM)                  | - Application allows SCT members to create Bespoke Baskets. In addition it serves as a repository for CDS curves, market data (obtained from Camden) and financial ratings (Bloomberg)                                                                                                                                                                                                                                                                                        |
| Random Factor Loading (RFL)          | - Application which serves two functions:  
1. Interfaces directly with Risk Engine and generates RFL calibration parameters for CDO\(^2\) trades, and these results can be uploaded into Camden (CDO\(^2\) trade valuation)  
2. Calculates component notional and exports results to excel spreadsheet                                                                                                                                                                                                                                                                                          |
| Report Mailer                        | - An XML driven Java based utility that serves two functions:  
1. Provides the ability to run reports in SQP and for the results (rendered in HTML, PDF, or Excel tables) to be sent to a distribution list via email  
2. Allows transfer of data/results of a SQL call from a target server and write the results to a destination server                                                                                                                                                                                                                     |
| Structured Data Kernel (SCK)         | - Application that reports: risk, profit and loss, reconciliation, market data, scenarios, and system status. Added features include filtering through portfolio, export reports in Excel or PDF formats, and aggregation of reports                                                                                                                                                                 |
| STC Data Tools                       | - An Excel plug-in library that contains a number of functions that retrieve entity and spread data from a variety of sources.                                                                                                                                                                                                                                                                                         |
| Trade Viewer (TV)                    | - GUI that allows SCT members to view trades (basket trades, CDOs, CDO\(^2\)). These trades can be organized by portfolio or entity name                                                                                                                                                                                                                                                                               |

Table 1: Structured Credit Desktop Internal Applications
Source: SCD Presentation, 2009
Periodically, the Structured Credit Development team conducts maintenance and improvements on the SCD. These modifications have a pre-defined build and release system, which was the focus of our analysis. However, these modifications also follow a standard set of software development best practices. The Structured Credit Development team implements different software methodologies to streamline and facilitate the release management.

2.3 Software Development Practices

Michael Perks (2003), a solution architect for IBM Corporation, notes that 80% of software project are unsuccessful because of budget, time, and missing function problems. Perks continues by stating, that more than 30% of software projects are poorly executed and even projects using modern technologies such as Java, J2EE, XML and Web Services are no exception to this rule. To combat the difficulties encountered when developing software projects, the Structured Credit Development utilizes several software best practices. The two primary software practices strictly practiced are agile software development and continuous integration.

1. **Agile software development:** The project is released based on iteration cycles involving multiple development teams in the US, Europe and India. Agile software development is a group of software development methodologies based on interactive and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. The Agile Manifesto introduced the term in 2001 (Manifesto for Agile Software Development, 2001). Brought up in the Agile Manifesto is the value of agile development, which values individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, responding to change over following a plan.
2. **Continuous Integration**: The members of a team integrate their work frequently; usually each person integrates at least daily – leading to multiple integrations per day, rather than integrating every time code source is checked in. Each integration is verified by an automated build (including test), to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows team to develop cohesive software more rapidly (MartinFowler.com, 2006).

3. **Hudson** or **Continuum** is used for continuous integration. Even though these tools are not in used for the current development environment, they are considered great potential help to the project’s iteration release management.

   a. **Hudson**: Hudson is an extensible continuous integration server. It monitors executions of repeated jobs, such as building a software project or jobs run by cron. The focuses of Hudson are building/testing software projects continuously, such as CruiseControl or DamageControl, and monitoring executions of externally-run jobs, such as cron jobs and procmail jobs, even those that are run on a remote machine. (Hudson, 2010)

   b. **Continuum** is an alternative to Hudson but with similar functionalities; Continuum also provides continuous integration and build server. It is an enterprise-ready continuous integration server with features such as automated builds, release management, role-based security, and integration with popular build tools and source control management systems. Continuum can help the build team put control of releases in the hands of developers, and therefore improve quality and maintain a consistent build environment. (Apache Continuum, 2010)
These tools are not implemented for the current development environment but are considered great aides that help the project’s iteration release management. Agile and continuous integration best practices guide software developers on a project management perspective. However, software development practices are often supplemented by software development framework methodologies. Perks (2003) notes that the strategy chosen for the development framework is critical to all other project activities as subsequent decisions are derived from the development framework.

2.3.1 Structured Credit Desktop: Development Framework

Maven, the Yiddish word for “accumulator of knowledge”, provides a standard way to build the projects. A development framework philosophy, Maven provides: (1) a clear definition of what the project encompasses, (2) an easy way to publish project information, and (3) a way to share JARs across several projects. Overall, Maven software development methodology is used for building and managing any Java-based project. (Apache Maven Project, 2010)

Maven allows a project to build using its project object model (POM) and a set of plugins that are shared by all projects using Maven, providing a uniform build system. POM is an XML representation of a Maven project held in a file named pom.xml, which is crucial to the project in the philosophical sense (Apache Maven Project, 2010). It is a one-stop-shop for all things concerning the project, including configuration files, the developing roles, the defect tracking system, the organization and licenses, the URL of where the project lives and the project’s dependencies. The “version” information needs to be updated every time a newer version of the program is released, together with the version of the files’ parent files and children files.
Another interesting feature about Maven is that it can be modified to best suit the needs of development team. In other words, the Structured Credit Development team can develop customized maven plug-ins with their preferred software development best practices. Because Maven “is at its heart a plug-in execution framework [in which] all work is done by plug-ins” (Apache Maven Project, 2010) the Structured Credit Development team can choose which plug-ins best embody their software development philosophies. For example, there are two Maven plug-ins, the Java Archive (JAR) and Web Application Archive (WAR) plug-ins, which are intricately included in the SCD build and release process. JAR plug-in creates a JAR file for developer’s project sources and are generally used to distribute java applications in the form of classes and associated metadata and resources (Maven JAR Plug-in, 2010). WAR plug-in is responsible for collecting all artifact dependencies, classes, and resources for the web application and packaging them into a WAR (web application archive) file (Maven WAR Plug-in, 2010).

2.3.2 Other Software Development Tools

In addition to implementing software development methodologies, the Structured Credit Development team’s build and release process is deeply influenced by other software building tools.

There are two repositories that the Structured Credit Development group uses, Perforce and Nexus. Perforce contains the SCD’s source code and thus serves as source code repository. As an application of software configuration management (SCM), this low-cost repository tool allows SCM to no longer be restricted to a particular type of development process, and assists the Agile software development workflow (Perforce Software, 2010). Nexus, developed by Sonatype, is used for SCD’s artifact repository for development, deployment, and provisioning.
(Nexus, 2010). The official Maven releases will be stored in the Nexus repository. Nexus can help the development team share those artifacts with other developers and end-users.

Other software tools are used to deploy the SCD application. The Structured Credit Development team developers execute the build process through a Jetty or Tomcat deployment goal specified in Maven (SCD Wiki, 2010). Jetty is used as a lightweight servlet container and would quickly deploy an application (Eclipse Foundation, 2010). Tomcat, created by the Apache Software Foundation (2010), on the other hand “powers numerous large-scale, mission-critical web applications”. Essentially Tomcat and Jetty are dynamic deployment solutions for java based-applications. In terms of the Structured Credit Development team, Jetty and Tomcat allow new iterations of SCD to be accessible to users on the World Wide Web.

The software development tools and the Maven software project management framework help the Structured Credit Development team construct a schema of how to maintain and improve the SCD. The Structured Credit Desktop is a large-scale and critical application to the Structured Credit Trading business. Christopher Rice, a senior technology project manager at Bank of America, mentioned that such a robust application required extensive regression testing for any minor modification(s) (personal communication, November 2010).
3. Business Problem

The Structured Credit Development team is a global technology group that seeks to provide effective and efficient end-to-end technological service and support to Global Banking and Markets division, specifically the Structured Credit Trading desk. As a technology team in the GMRT division, the Structured Credit Development team maintains and improves the stability, functionality, performance, and security of proprietary infrastructure and applications. The Structured Credit Development team needed to implement certain best practices and modify its current release management process. These modifications sought to minimize unnecessary risk while delivering new releases, and streamline a lengthy and manual process.

The Structured Credit Development team’s build and release process required lengthy versioning management of the Structured Credit Desktop. Release Managers needed to individually increment versions for new (or improved) pom.xml files. Because the Structured Credit Desktop is a suite of applications, and particular applications such as Basket Manager had their own version numbers, automatic versioning of all modules with a new release was not optimal.

In addition, the Structured Credit Development team needed to enhance the release process of the Structured Credit Desktop. Ideally, during a production release, module modification had undergone quality assurance and user acceptance testing. However, the files for these module changes that the build process produced, WAR files, were rebuilt to deploy in different environments (UAT, UAT2, PROD), and there was the possibility of human error. The validation process for the changes made was corrupted if the changes were redeployed to a different environment.
The following objectives were set in order to accomplish the goal of this project:

- Identify and develop the best practice to automate and enhance the versioning process required during the build phase
- Identify and develop the best practice to separate configuration files
- Combine both automated versioning and configuration file separation under an application which facilitates release management
4. Implementation & Results

As members of the Bank of America Structured Credit Development team, we analyzed the current build process, and implemented software development and release best practices to streamline this process. Our goal was to provide the Structured Credit Development team with a one-click automation process that facilitates global logistics, as the team exchanges release cycles every quarter. Upon our arrival to Bank of America, we planned to become familiar with Structured Credit Development team environment and begin the analysis of Structured Credit Desktop.

Appendix A contains the “Milestone Gantt Chart”, which outlined the different tasks and timeline we followed in order to accomplish our goals.

4.1 Build and Release Process Analysis

The Structured Credit Development team’s build and release process as outlined in Chapter 2, followed Maven software development practices. Maven projects are built by executing different “goals”, such as clean, install and release. In addition to “phases”, the Maven methodology includes pom.xml files, which “contain every important piece of information about [a] project” (Apache Maven, 2010), and are considered the basic unit of work in Maven. pom.xml files contain the different project files, versions indicators and artifact IDs (indicates the unique base names of the primary artifact (JAR file) that the project generates).

The SCD build process began with developers executing maven build commands. The phases executed and sorted through the local source code repository, analyzing resources and pom.xml configuration files, and packaging them into JAR files. The final product of the Maven build was a single WAR file. This WAR file contained JAR files, JSP specifications, and other
Figure 3, depicts in greater detail the steps traversed to build a new iteration of the SCD application.
As noted in Figure 3, after the WAR was built, developers in the Structured Credit Development team utilize Tomcat and Jetty, servlet containers that allow developers to launch web applications. The developer would deploy the WAR file to a website host server, which would in turn expand WAR file into several project JAR files. Then, if the developer wanted to release a new or modified SCD project, they would specify the JSP file in a web browser. The result would be the creation of a JNLP file that details different resources and other components necessary to build the SCD project.

4.3 Release Management

The Structured Credit Development team needed to enhance its build and release process for new modifications to the Structured Credit Desktop. Ideally, during a production release, module modification had undergone quality assurance and user acceptance testing. However, the files for these module changes, produced in the build process (WAR files), were built to deploy in different environments (UAT, UAT2, PROD). Thus, the possibility of human error existed. The validation process for the changes made was corrupted if the changes were redeployed to a different environment from the one these changes were built and tested. In addition, it was inefficient to deploy to different release environments and rebuild projects, when the “content” files were the same, and the only differences were in the configuration files.

4.3.1 Current Process

The Maven build parameters that were included to build the SCD application were environment-specific. In addition, for each build environment, build processes needed to execute in a specific order to have the environment available for use (with the environment ID specified during the build). Therefore, there were different JAR files for different environments. For different environments, if no further changes were made, the difference only existed in the
configuration files, which were overridden during the building process of maven, and generated based on the different environment profiles specified in profile.xml file. The configurations files were contained in the JAR file for each application module.

After the content was built and installed into the central repository, the developer used webstart to build a WAR file, which contained the built JAR files. Therefore, there were different WAR files corresponding to different build environments as well. Different host URLs were linked to different environment WAR files, and the default JSP access page, SCTDesktop.jsp, would match the URL with the appropriate WAR file.

The following diagram outlines the current process:

![Diagram of Build and Release Current Process]

**Figure 4: Build and Release Current Process**

Although the above process worked for most cases, there was a possibility of human error. The validation process for the changes made was corrupt if the changes were redeployed to a different environment. For example, if the internal release for the QA environment worked...
desirably and was fully tested, but developers made changes before the production release without proper testing, the application might be corrupt.

In addition, this was an inefficient process when deploying to different release environments to rebuild projects. Entire application content JARs and WARs were reproduced for each environment, although the content files were the same, and the only changes were in the configuration files.

4.3.2 Proposed Solution: Configuration Files Separation

Our goal was to separate the configuration files from the module JARs, and build a separate JAR file for each environment’s configuration files. This solution will allow the development team to move the content of the application from QA release directly to the production release. The new proposed build process will not require an environment to be specified. In addition, only one content JAR will be built. Identical for all environments, this JAR will also contain configuration JARs for each of the different release environments.

During the webstart phase of the build process, the new solution will build a single WAR file for the web application. The WAR file contains all the JAR files generated during the build process and is deployed on the server side. Note that the JAR files now contain different conflicting configuration files that would potentially generate failure during application run time.

In order to resolve this issue, we developed two approaches. Both approaches would have to dynamically filter configurations JARs for clients and download the correct configurations based on the request URL.

The first approach focused on developing a customized servlet to substitute the default JNLP download servlet. The proposed custom servlet will produce a JNLP file and recognize the host name for the build environment. As a result the servlet filters the redundant
configuration JARS. The second approach focused on modifying the existing SCTDesktop.jsp page. This page is used to connect download and launch WAR file. However, we modified the JSP page to sort through the different configuration JARs, and allow the download of the correct JARs. Ultimately, these methods achieve the same result. However, the second approach, with the JSP page, results in fewer modifications to the current system. Therefore, we chose the JSP method for our implementation.

In order to accomplish the goal, we separated the configuration folder from each module’s pom.xml file, so that the built JARs for the main project components do not contain the configuration folders. For different environments, we build separate configuration JARs that contain only the configuration folder specific to a certain release environment. The following steps are implemented for achieving the goal.

1. **Exclude the configuration folders from the main JAR**
   In each module’s pom.xml file, delete the included resources for this module configuration folder, which were specified under <resource> tag under <build>.

2. **Package into new configuration folders**
   For each specific environment, the configuration folder is different. We built a new module to separate configuration JAR for each environment, containing only the configuration folders for all modules for this environment.

3. **Include the configuration JARs**
   Finally, we included the newly created modules for each environment in the SCTDesktop’s pom.xml file. Therefore, the command “mvn clean package install” will build the content of the project as well as all the different configuration files for each sub-module.
The WAR file for web release will contain the main project component JAR and different configuration JARs. However, different configuration JARs contradict with each other, and therefore cannot be downloaded together. The run-time environment is specified in the access URL, and based on the URL request, the application chooses the correct configuration JAR for the download.

4. **Build the WAR**
In the webstart project’s pom.xml file, include the resources for the newly built configurations JARs. Therefore, the WAR file on the server contains the JAR file for the project content and the JARs for different environment. To deploy for different environments, change the WAR file name from “SCTDesktop-null.war” to “SCTDesktop-env.war” where “env” is the name of the environment, such as “uat”, “uat2”, “prod”, etc.

5. **Implement SCTDesktop.jsp to Build JNLP**
In the access page SCTDesktop.jsp file, instead of using the default string replacement during the build process to specify the build environments and host names, logic is added into the page so that it can analyze the request URL and fill in the host name and build environment dynamically. In addition, it includes all the content JAR files for the application as well as filters out the configuration JARs that do not match with the specified build environment.

6. **Access the JSP for Download**
The access method stays the same as before the implementation. Clients would request the URL in the browser in the following format: {hostname}:{port-num}/SCTDesktop-
\{env\}/ SCTDesktop.jsp? region= \{region\}, where “env” is the build environment and “region” is the parameter specified to choose the build region. Download will start after the URL is sent to the server and the application is to be built with the correct configuration.

The new implementation will result as the following diagram:

![Diagram of build and release process impact]

Figure 5: Build and Release Process Impact

4.4 Version Manager Application

The Structured Credit Development team’s build and release process required lengthy versioning management of the Structured Credit Desktop. Release Managers individually incremented versions in each pom.xml files, for any module that had any changes. Because the Structured Credit Desktop is a suite of applications, and particular applications such as Basket Manager have their own pom.xml files, automatically incrementing version numbers of all modules with new or emergency releases was not optimal.
4.4.1 Current Process

While section 4.1 described in detail the build and release process, current standards for versioning had not yet been detailed. First, the release managers needed to compare every file of every module between the new project and one in Perforce, and note any changes. If changes existed, the release manager needed to manually update the module’s version number. Once completed for a single module, this process had to be repeated for all nine of the Structured Credit Desktop’s modules. In addition, if a module’s parent module or dependency modules’ version numbers were updated, the changes would be reflected in this module’s pom.xml file as well. Finally, after all version numbers in parent/child and dependent modules were updated, the release manager could then begin the SCD build process through Maven command line arguments. Yet, the SCD build process could be corrupt if there was a single mistake, and the release manager would need to roll back and commence this version process again. We realized that this process could occupy release managers for many hours, and was prone to human error.

Below, Table 2 outlines the new convention for versioning we created. This convention is “Major.Minor.Patch Release”. Previously, there might be a text label such as “ER” (for emergency release) in addition to, or in lieu of, a patch number. There may be several variations of the same version number, with only differences in the text label. This made keeping track of version numbers difficult, and therefore, we standardized the numbering process.

<table>
<thead>
<tr>
<th>Nexus / Control Version Number</th>
<th>Major Releases</th>
<th>Minor Releases</th>
<th>Results</th>
<th>Default Release</th>
<th>Final Version Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>desktop module: 1.0.5</td>
<td>(blank)</td>
<td>(blank)</td>
<td>patch release</td>
<td>Yes</td>
<td>desktop module: 1.0.6</td>
</tr>
<tr>
<td>desktop module: 1.0.5</td>
<td>desk or</td>
<td>(blank)</td>
<td>major release</td>
<td>No</td>
<td>desktop module: 2.0.0</td>
</tr>
<tr>
<td>desktop module: 1.0.5</td>
<td>(blank)</td>
<td>desktop</td>
<td>minor release</td>
<td>No</td>
<td>desktop module: 1.1.0</td>
</tr>
<tr>
<td>desktop module: 1.0.5</td>
<td>(blank)</td>
<td>(blank)</td>
<td>patch release</td>
<td>Yes</td>
<td>desktop module: 1.0.6</td>
</tr>
<tr>
<td>BC module: 2.3.5</td>
<td>(blank)</td>
<td>(blank)</td>
<td></td>
<td></td>
<td>BC module: 2.3.6</td>
</tr>
</tbody>
</table>
4.4.2 Proposed Solution: Automated Versioning

We decided that a new Java application would be the best method to automate the versioning process, as per the Structured Credit Development team’s requirements. This application will provide for a method to compare the older release with the new release, in order to ascertain the correct versions of all the modules. The end user will simply have to provide the location for the current project they wish to have versioned correctly, and the location of a previous release to compare it with, ideally in their local and updated Perforce directory. The application will pull the current module, and locate the previous release, comparing them. If there are any updates, the appropriate pom.xml files are updated. Furthermore, the application loops and updates parent and dependent module version numbers as necessary. The diagram below depicts the features the new implementation.

<table>
<thead>
<tr>
<th>desktop module: 1.0.5 BC module: 2.3.5</th>
<th>desktop; BC</th>
<th>(blank)</th>
<th>major release</th>
<th>No</th>
<th>desktop module: 2.0.0 BC module: 3.0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>desktop module: 1.0.5 BC module: 2.3.5</td>
<td>(blank)</td>
<td>desktop; BC</td>
<td>minor release</td>
<td>No</td>
<td>desktop module: 1.1.0 BC module: 2.4.0</td>
</tr>
<tr>
<td>desktop module: 1.0.5 BC module: 2.3.5</td>
<td>desktop</td>
<td>BC</td>
<td>major release for desktop; minor release for BC</td>
<td>No</td>
<td>desktop module: 2.0.0 BC module: 2.4.0</td>
</tr>
<tr>
<td>desktop module: 1.0.5 BC module: 2.3.5</td>
<td>BC</td>
<td>desktop</td>
<td>major release for BC; minor release for desktop</td>
<td>No</td>
<td>desktop module: 1.1.0 BC module: 3.0.0</td>
</tr>
</tbody>
</table>

Table 2: Major.Minor.Patch Release Convention

Structured Credit Development
Figure 6: Version Manager

However, the version management is a complex application with features for different versioning scenarios. For a more detailed flow diagram of the Version Manager’s logic, refer to Appendix B.

Under default conditions, the application uses the default values specified in the config.properties file, and directly updates the version numbers. Otherwise, the application will prompt for these values. After the user enters the location of the current and control project, the application will compare the version numbers by parsing them from the control project, and parsing maven metadata files to get the latest version numbers in Nexus. If the control project is not the latest version, the application will prompt the user to confirm that he or she would like to update the version based on a previous release rather than the latest release. Additionally the user has the “dry-run” option that will generate a pomTest.xml instead of overwriting the original pom.xml, so that the user can compare the changes prior to making them.
The application parses through the module files, comparing the current and control project modules, and determining if there are any differences. If there are differences, then the application will increment the module’s version number in its pom.xml file. Next, the application will loop through and increment the parent version number and dependent module version numbers in every module’s pom file as well. In addition, the application will update the pom.xml file for the webstart module, which needs the version numbers for every module, and set its own version number to that of the parent SCTDesktop. The version numbers increment based on values from the config.properties file. If any module is specified as a major or minor release, those numbers are incremented, otherwise by default, the patch number is incremented. Outlined below is the specific logic of the application (depicted in Figure 7).

1. config.properties file is read into the application. Application checks the “default” key in this file.
   a. If “default” has the value “true”, then skip steps 2-5 and build using the values specified in the config.properties file.
   b. If “default” has the value “false”, then go to step 2.
2. Application will prompt user for the project path. Next the application will ask user whether to “build project” (bypass versioning) or “version project”.
   a. If “build project” then the application will bypass versioning logic and will commence maven build commands
   b. If “version project” then application asks user for the control project path
3. Application asks whether this is a “Dry Run”.
   a. If so, then project pom files will not be updated, but new pomTest files will be created; therefore, the user can examine the changes prior to committing
   b. If not, then the current pom files will be edited

4. Application will extract version numbers from the control project and Nexus repository.
   a. For control project, the application will loop through and parse each module’s pom.xml file (which contains the module’s version number). The application will then store these version numbers in a HashMap
   b. For the Nexus repository, the application loops through each Maven metadata file, and these version numbers are stored in a HashMap

5. Application will compare different sets of version numbers
   a. If a difference is found, then application prompts user to specify which version numbers to use: Nexus repository or control project
   b. If no difference is found, then the application updates version numbers from the control project.

6. For each module, the application compares the files between the control project and current project.
   a. If difference is found then the module name will be added to an “update map”

7. For each value in the update map, the application then increments version numbers in the pom.xml files, taking into consideration which ones are major and minor releases. In addition, the module name and new version number will be added to an “update map”.

8. Finally, the application loops through all the pom files, updating the parent SCTDesktop version number, and any dependent module version numbers. For the webstart module,
the SCTDesktop version number will become its own, and the all the dependent module version numbers will also be updated.

Additionally, there was much discussion about which files, whether Nexus, Perforce or the local repository would contain the latest or most appropriate version numbers. Some thoughts are below:

1. Nexus has the latest release version, so the new project can increment version numbers from Nexus
2. The development team may wish to branch from an earlier release, and increment version numbers from that previous release

Therefore, the application is flexible to incorporate both of these thoughts.

4.6 Integration

In order to make the build and release process run smoothly, we combined the versioning management process and release management. In the Java application that accomplishes the versioning update process, maven commands are introduced into the program as command line arguments, and environment information is located in the config.properties file.

Ideally, there are two command line arguments, which are two maven commands. They are included in the application after the pom.xml files are updated. The first command executes on the SCTDesktop build process, and the second command executes on the webstart build process.

This combination eases the entire process. “SNAPSHOT” or “ER” labels is not supported in the versioning update process, because the QA release and production release will be using the same WAR file (only renamed for different deployment).
Moreover, it reserves the feature to update version numbers and build the application separately. If the user does not specify any command line argument, the program will only update the pom versions, but not build the SCD application. If there is only one command line argument, only the SCTDesktop is built, and the webstart application will not be built. Developers can also execute the maven build command on the command line, separately.

4.7 Quality Assurance

The final step in our development process was to prepare the integrated release management and version management solutions into one consolidated application. At Bank of America, applications need to undergo multiple types of testing, such as unit tests, user acceptance tests, regression tests, etc. In order to guarantee the build and release management application’s performance, we worked with the Structured Credit Desktop QA team to insure the project goals are properly met.

Quality assurance is one of the most important steps during software development cycle, and therefore serves as a crucial section in our implementation. It is one of the project deliverables that we completed.

During the collaboration with QA team, we generated detailed use case documents. Since the test engineers were not as familiar with the SCTDesktop application’s implementation as the development team, we included enough details in the documents so that the QA team could have a clear understanding of our implementation and system performance. We listed all the different possible result for the program execution, and illustrated each detailed application feature with concrete examples. For example, the following screenshot demonstrates an example of the use cases that we generated for the pre-production build release process. We clearly detailed the execution steps with sample file content and command line input.
The completion of the quality assurance process will be out of the project duration scope; however, with the interaction with the QA team and the test documentation, the testing phase will be successfully completed.
5. Conclusions

The previous SCD release management required extensive and cumbersome manual procedures, including building specific environments for different types of application releases, comparing changes between release cycles, and overwriting correlated pom.xml files if there were any changes. This was time inefficient, as well as heavily subject to human error. Often if there were mistakes during the manual update process, the application may successfully compile, but would be corrupt during run time.

The SCD build and release optimization has a great impact on the SCD development and QA teams’ daily work. The SCD release process is now automated, which significantly minimizes the release manager’s work, and provides higher accuracy for the release management. Because the application implementation involves dynamic data fetching, it is reliable and portable for future development of the SCD application.

The first step of the project allows the application to be built with content JARs and different configuration JARs together at the same time. The SCD host server provides clients with the configuration JAR that matches the requested build environment to be downloaded. The second step of the implementation loops through the entire application, and compares each module with the previous release. If any difference is detected, the corresponding pom.xml for this module is updated with a newer version number, and pom.xml files of its related modules are updated as well. Finally, we integrated these two steps as the optimization to manage both the build and the release process; therefore, whenever there is a new release, the release manager only needs to run a command line command, and the application can be built for all types of environments, with the version number correctly updated.
## Appendix A

### Milestones Gantt Chart

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Duration</th>
<th>Start</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acclimation</td>
<td>10 days</td>
<td>Mon 10/25/10</td>
<td>Fri 11/5/10</td>
</tr>
<tr>
<td>In Depth Analysis of SCO</td>
<td>10 days</td>
<td>Mon 11/9/10</td>
<td>Fri 11/16/10</td>
</tr>
<tr>
<td>Configuration Files Application</td>
<td>20 days</td>
<td>Mon 11/1/10</td>
<td>Thu 12/3/10</td>
</tr>
<tr>
<td>Separate Config Files</td>
<td>20 days</td>
<td>Mon 11/1/10</td>
<td>Thu 12/3/10</td>
</tr>
<tr>
<td>Evaluation &amp; Quality Assurance</td>
<td>14 days</td>
<td>Mon 11/1/10</td>
<td>Thu 12/3/10</td>
</tr>
<tr>
<td>Test Config Separation Script</td>
<td>14 days</td>
<td>Mon 11/2/10</td>
<td>Thu 12/9/10</td>
</tr>
<tr>
<td>Unit Testing</td>
<td>14 days</td>
<td>Mon 11/2/10</td>
<td>Thu 12/9/10</td>
</tr>
<tr>
<td>Versioning Application</td>
<td>20 days</td>
<td>Mon 11/1/10</td>
<td>Thu 12/3/10</td>
</tr>
<tr>
<td>Automate Version</td>
<td>20 days</td>
<td>Mon 11/1/10</td>
<td>Thu 12/3/10</td>
</tr>
<tr>
<td>Evaluation &amp; Quality Assurance</td>
<td>13 days</td>
<td>Mon 11/15/10</td>
<td>Thu 12/5/10</td>
</tr>
<tr>
<td>Test Versioning Script</td>
<td>19 days</td>
<td>Mon 11/15/10</td>
<td>Thu 12/9/10</td>
</tr>
<tr>
<td>Unit Testing</td>
<td>19 days</td>
<td>Mon 11/15/10</td>
<td>Thu 12/9/10</td>
</tr>
<tr>
<td>Final Documentation</td>
<td>20 days</td>
<td>Mon 11/22/10</td>
<td>Fri 12/17/10</td>
</tr>
<tr>
<td>User Guide</td>
<td>20 days</td>
<td>Mon 11/22/10</td>
<td>Fri 12/17/10</td>
</tr>
<tr>
<td>Training Wiki</td>
<td>20 days</td>
<td>Mon 11/22/10</td>
<td>Fri 12/17/10</td>
</tr>
<tr>
<td>Design of &quot;One-Click&quot; Build &amp; Release</td>
<td>10 days</td>
<td>Mon 12/6/10</td>
<td>Fri 12/17/10</td>
</tr>
<tr>
<td>Prototype Application</td>
<td>10 days</td>
<td>Mon 12/6/10</td>
<td>Fri 12/17/10</td>
</tr>
<tr>
<td>Evaluation &amp; Quality Assurance</td>
<td>10 days</td>
<td>Mon 12/6/10</td>
<td>Fri 12/17/10</td>
</tr>
<tr>
<td>Handover (ensure sustainability)</td>
<td>11 days</td>
<td>Fri 12/3/10</td>
<td>Fri 12/17/10</td>
</tr>
</tbody>
</table>

### Legend

- Task
- Critical Task
- Progress
- Milestone
- Summary
- Rolled Up Task
- Rolled Up Critical Task
- Rolled Up Progress
- Rolled Up Milestone
- Split
Appendix B

Application checks configuration properties file

default field in configuration properties file

Application prompts for path to new SCD and sub-modules

Build Project or Version Project

Application prompts for path to SCD control workspace

The control project and its modules are the same SCD and sub-module versions as the newest release in Nexus?

No

Use conditions specified in properties file

Yes

Application prompts user to update the version numbers based on the latest version #s in Nexus or to the version #s in the control project

Yes

Application will update version numbers from newest Nexus release

Determine Version from configuration properties file

Application will update version numbers from control project

Dry Run

Version Numbers are incremented & Override original pom.xml files

Version Numbers are incremented & Create new pom.xml.test files

Application compares source code between branches/local repository

If difference

Yes

No

Application compare dependencies and children modules

Build project
References


Personal Communication. Christopher Rice. November 15th, 2010

Personal Communication. Ron Toam. October 8th, 2010

Structured Credit Desktop Presentation. Siavash Talebipour, Arjun Sen. April 21st, 2009