

Survey: Best Practice for Single Source and Release Management

Daniel Hafner, Ralph Schibli & René Stierli
Version 1.3 (final)

Index

1. INITIAL POSITION.....	2
2. SOME DEFINITIONS.....	2
3. MOTIVATION TO USE SINGLE SOURCE MANAGEMENT APPROACH ...	3
4. SELECTED DEVELOPMENT AND DEPLOYMENT APPROACHES.....	4
4.1 HOW TO DEVELOP WITH ONE SINGLE SOURCE BASE.....	4
4.2 HOW TO ADAPT SOFTWARE TO CLIENT SPECIFIC NEEDS.....	5
4.3 HOW TO DEPLOY IN A COST-EFFICIENT WAY	5
4.4 HOW TO RELEASE NEW VERSIONS.....	6
5. LESSONS LEARNED.....	7

1. Initial position

A large Swiss bank is faced with the situation of serving multiple international locations with the same bank application, each having its own country-specific flavor. Over the years every location works with its own release, and the number of releases to be supported is equal to the number of locations. This constellation leads to increased testing effort and maintenance, costs amongst other things. Consequentially, the bank is looking for a solution regarding the single source and release management.

The bank mandated itopia to perform a survey on the Swiss market with the goal of gathering best-practices in the single source and release management.

This document summarizes the results. The seven participants of the survey are two core banking providers, two business process outsourcing (BPO) providers, two classical software engineering corporations (providing e.g. CRM systems) and of course the bank itself. So, every participant is closely linked to the financial services industry. The names of any participants will not be disclosed.

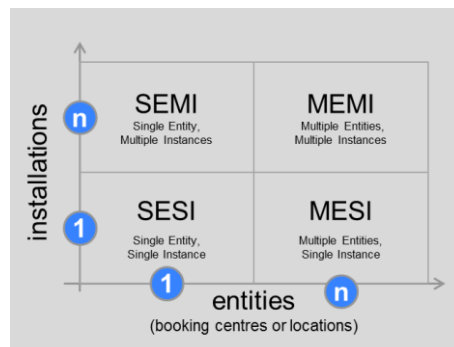
2. Some definitions

Before going into detail a few definitions are given.

During the interviews with the participants we noticed that especially for “*single source management*” no common shared definition exists. We define single source as a generic development pattern in which different functional behaviors are kept within the same piece of code across all instances and, enabled through external parameterization and configuration, avoiding the need to have different (duplicated) software modules and related maintenance costs.

To adapt the software to client specific needs (e.g. four-eye or six-eye-principle for payments) there must be mechanisms to do so. Amongst other things there are two concepts which need to be defined at this point: *parameterization* and *configuration*. Parameterization is done by scripting (e.g. an approval workflow). If you have a set of pre-defined workflows coded already into the software product and just want to activate a suitable one for your bank, you simply do a configuration by choosing one. Configuration and parameterization do not change source code.

Talking about the deployment side, we have to distinguish how the application is operated.



The most trivial *deployment mode* is the so called SESI, where one entity runs on one installation. So, if you operate a core banking system for 3 different locations, every location gets its own installation.

In case you operate all three locations on the same installation, the deployment mode is called MESI. In other words: The core banking system is multitenant (in German “mandantenfähig”).

If you combine the entities with multiple installations you get the deployment modes called SEMI and MEMI. MEMI is the most flexible/generic deployment mode.

Last but not least this study covers the *release management*. This process encompasses the planning, design, build, configuration and testing of hardware and software releases to create a defined set of release components. Release activities also include the planning, preparation, scheduling, training, documentation, distribution and installation of the release to many users and locations.

3. Motivation to use single source management approach

All survey participants were asked to name the main drivers/motivation factors for single source management (even not all participants have a pure single source approach). The answers in the order of number of mentions are:

- **Cost reductions:** By standardizing a software product (one common code base for every client) the costs for development and deployment can be reduced dramatically. Common functionalities and regulatory requirements can be made available for all clients without redundant implementation. Therefore a software engineering company can cope with lower resources to serve multiple clients (economies of scale). Finally, by choosing a standardized product the client can save money due to shorter release migration cycles.
- **Reduction of complexity and operational risks:** By developing out of one source for all clients instead of multiple sources, the complexity of the software engineering process can be reduced. For example, the testing process is simplified because the software developers have to test one instead of multiple builds. All this may lead to a higher software quality which helps to reduce operational risks in the bank.
- **Synergies for community:** In case a core software product is shared for a group of clients (not considering client individual adjustments) the setup of a community can make sense. The community can share knowledge, gather and prioritize new requirements (which might be considered for a future releases) and benefit from innovations proposed by others. Communities have a certain buying power which can lead to adjustments of pricing models.

Sometimes, and often on a management level, it is argued that with the abdication of single source management changes can be implemented much faster. This might be true due to fast decision processes and might be coupled to a release policy which foresees many releases per year. But in the long run the advantages mentioned above

cannot be achieved by simply optimizing one client's solution (instead of optimizing over all clients).

4. Selected development and deployment approaches

Out of the various development and deployment approaches we only present a subset which we consider to be best-practice.

4.1 How to develop with one single source base

A software product aimed to fit different customer needs has to be designed from scratch to allow future customization. In our interviews we identified several factors that play an important role in establishing and maintaining a successful single source base at development time.

Software design and architecture aspects:

1. *Stable database model:* Keeping and guarding a single and stable database model is a fundamental key factor for a long and prosper lifecycle of a company's software product. It is also enabling the MESI deployment concept which is quite common for running banking applications.
2. *Modularity:* Dividing a software product into multiple tiers with strictly separated modules and clear responsibilities helps to deal with growing complexity. A key element is the management and reduction of internal dependencies between modules on design and implementation levels. For example a software company may use a module-based design to provide client-specific functionality in specialized modules that are developed and deployed only for a certain customer.
3. *Enable client-side adaptations:* Provide mechanisms for client-side adaptations. One approach is to offer APIs or a plugin architecture that can be used to extend a company's software and integrate it into an existing environment. Another way is to provide a parameterization engine which allows scripting-like extensions to your software. Configuration profiles on different levels are also a common way to provide client-side customization.

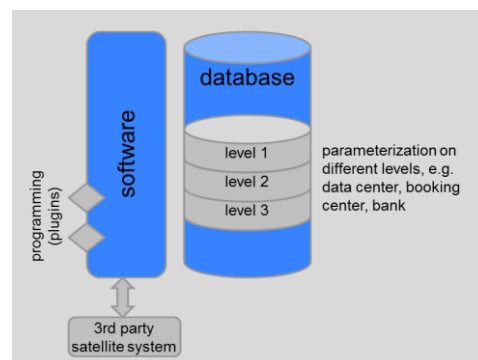
Organizational aspects:

1. *Separate code bases:* If a company is developing client specific modules or extensions, it is wise to keep this code in a separate repository, apart from the common kernel code.
2. *Developers' discipline:* If you have to support more than one version of your product, it is essential to guard a high level of discipline inside your development teams. Changes and bug fixes have to be carefully documented and labeled inside your code versioning system, especially if you are applying them to older versions. When the time has come to merge back to a trunk release for all clients, the identification of affected changes is crucial. The depicted approach helps to minimize efforts for merging back.
3. *Product owner:* Establish the role of a product owner. It should be his responsibility to oversee the further development of the software product and to develop a mid-term development roadmap. The product owner should also be empowered to decide which functionalities will be transferred from client-specific code into the product kernel.

4. *Flexible team organization:* A best practice approach is to provide one team of developers devoted to the product kernel and one or more teams who are client-oriented. Nevertheless, it's well-proven to staff new projects out of both teams and to share the responsibilities among all project team members. This helps to spread know-how and a common understanding of the product's concepts.

4.2 How to adapt software to client specific needs

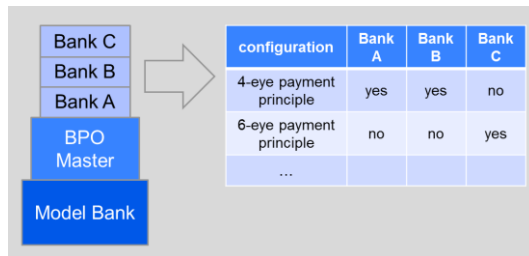
To adapt the software to client specific needs, there must be mechanisms in place to do so. One approach is to strictly separate the standard software code from the client-specific adaptation. There are four different ways to adjust software:



1. *Parameterization:* Parameterization (and/or configuration) can be done on different levels, e.g. on data center, booking center and bank level. Parameterization is inherited, but it is also possible to overwrite.
2. *Changes via future releases:* Clients gather requirements and submit change requests. If the change requests make sense for the whole community, they will be implemented.
3. *Plugin Programming:* If a change is not possible to be realized via parameterization, plugins with a special development kit can be developed. In order not to compete against the core product, plugins can be handled with certain runtime restrictions such as e.g. lower priority.
4. *Interfaces to 3rd party systems:* Because it does not make sense to program specialized functionalities already available within 3rd party systems (e.g. authentication via cross-off list, SecurID, OTP via SMS (one time passwords), etc.), the integration of such systems via standardized interfaces is possible.

4.3 How to deploy in a cost-efficient way

If you are a BPO provider in the banking industry and serving multiple similar client banks with the same platform, you should definitely consider the MESI approach. Usually a core banking provider delivers a so called model bank together with the software. The model bank is a basic parameterization/configuration of an ideal bank. The BPO provider then can take the model bank and add any additional adjustments (e.g. via parameterization) to the model bank which is relevant for his standardized banking processes, including interfaces to satellite systems. We call this the BPO master. The client banks of the BPO provider certainly have more or less additional requirements which have to be parameterized on top of the BPO master.



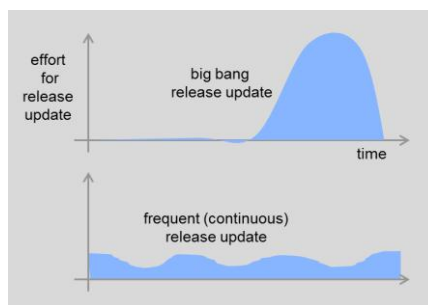
Basically, as the banks run in the same MESI environment, bank individual parameterizations (e.g. four-eye and six-eye payment principles) are available in every instance, but marked as non-relevant for certain banks and relevant for others. This concept is also known as *one source*.

To securely separate the client data from each other, the database can be set up as a virtual private database. A virtual private database masks data in a larger database so that security allows only the use of apparently private data.

This MESI approach does only work, if banks are willing to setup processes, products, etc. close to the ones defined in the BPO master. If a special requirement from one bank cannot be parameterized and special programming is needed, MESI reaches its limits.

4.4 How to release new versions

There are two fundamental release policies (with a few flavors) spread in the market:



- *Big bang release update*: Lots of changes are bundled into one big release (e.g. every 1-2 years). In addition, bug fixes are submitted via service pack releases during the year. Often a special project team has to be set-up to handle the big bang release updates.
- With the *frequent release update* there are many small releases per year (e.g. every month). The releases are handled by the existing IT ops teams.

Many software companies couple their release policy to a client-side obligation. With such an obligation, the clients have to migrate to actual releases within a certain period of time. Often, a release gap of 1-2 releases is allowed, meaning, if the actual release is 1.2, then 1.1 and 1.0 are still fine, but any older releases aren't supported any more.

We consider the frequent release policy to be best practice. By producing many small releases per year the quality of the code must be nearly faultless and the testing process well established and highly automated. Changes (functionalities, regulatory requirements and bug fixes) can be delivered very fast and don't have to be packed into special service packs.

As a precondition, the clients must be willing to update frequently (e.g. monthly). If some clients can't keep up with the monthly frequency, the software company could decide to offer a second release rhythm in parallel: e.g. quarterly releases. The quarterly releases are based on the monthly and nothing else than the aggregation of three monthly releases.

Actually, there's a trend to shift from big bang to frequent release policies. The reason for this change is the fact that the longer the period between two releases, the more functionalities and regulatory changes are packed into a service pack. Usually, as the service packs are tested differently than regular releases, the testing effort is increasing for regular releases (as you may correct bugs from service packs, too).

5. Lessons learned

It was interesting to get to know the lessons learned out of single source and release management from every participant. Some of the lessons learned are very much specific to one company. But never the less they are all worth to be mentioned, so we get a good picture of the maturity of the software engineering business.

1. *Enforcement by senior management:* The C-level has to enforce and support single source and MESI approaches. It is not just a nice idea from an IT architect. There should be a tight governance regarding change management especially regarding deviations from the standard (e.g. branch releases). In addition, large companies may require a kind of incentive program on project level to ensure that all projects are compliant with best-practices.
2. *Setup transversal organizations:* Approaches such as single source and MESI require transversal organizations with competence centers such as payments, foreign exchange, etc. In that way you can ensure that new requirements are handled promptly.
3. *Use an adequate software design/architecture:* Strict separation of standard product code and client specific configuration allows better maintenance and easy adaption to client needs. And always keep in mind that configuration files grow with every release and get more complex (inner dependencies). Consider the scalability of the development environment and the product itself!
4. *Don't break your concept and design, do refactoring instead:* If you feel like breaking your design and concept, it's time to revise your software architecture and design. But, be aware that refactoring normally only pays out in the long run: in the short view, a refactoring consumes a lot of time and costs and does not add any immediate benefit for the client.
5. *Listen to your clients, but decide by your own:* Establish external user/knowledge groups which gather requirements and share know-how. The lead of the groups must be in the hands of the software company, and the prioritization is done by the software company (e.g. by a product manager): No grassroots democracy!
6. *Optimize development quality and processes:* Reduce manual effort by automation (e.g. semantic and structural code checks, nightly builds, automated regression tests based on predefined test cases, etc.). Perform periodic code reviews. In addition, adapt the principles of agile software development inside your teams.
7. *Don't consider project-activities for the pricing only.* Always include a part for refactoring and technology refreshment. Plan enough time and budget for housekeeping during projects. And be aware that the more transparency the clients have about the price calculation the more they try to co-determine. At worst, this could end up in a discussion with clients not willing to pay for a technology refresh because it does not offer any functional enhancements.
8. *Convince the client of the benefits of a standard product:* real individualism can seldom be realized in a cost-effective way. In addition, if you buy a standard product an extensively adjust it to your needs, you also may lose synergies which come out of communities.
9. *Convince the client to upgrade in small release steps:* A release policy with frequent releases (and small changes) might be better than a big-bang release every two years. Frequent releases can be handled by the existing IT operations organization and don't

need a separate project to be set-up. And don't forget to inform the clients timely about future releases.

10. *Total flexibility from the technology side can turn out to be counterproductive:* Without rigid governance you may end up with several client-adapted solutions that have totally lost common base. Your clever software design may support it, but it's a nightmare to manage.