TRANSFoRm

Translational Research and Patient Safety in Europe

D3.4: First Draft Integration Plan

Custodix

Work Package: WP3, WT 3.6
Type of document: SQA
Version: V1.0
Date: 25 May 2012
Authors: Cédric Vansuyt (Custodix), Helmut Opsomer (Custodix), Brecht Claerhout (Custodix)

TRANSFoRm is partially funded by the European Commission - DG INFSO Under the 7th Framework Programme. (FP7 247787)
7th Framework Programme: http://cordis.europa.eu/fp7/ict/
European Commission: http://ec.europa.eu/information_society/index_en.htm

eHealth
Better Healthcare for Europe

European Commission

Table of Contents

1 Introduction 4
2 Overall Strategy 6
2.1 Software Release Cycle 6
2.2 Roles 9
2.3 Environments 11
2.3.1 Production Environment 11
2.3.2 Product Acceptance Environment 11
2.3.3 Software Integration Environment 12
2.3.4 Development Environment 12
2.4 Versioning 12
2.4.1 Versioning Scheme 13
2.4.2 Builds 14
2.5 Interfaces, Stubs and Project Planning 19
3 TRANSFoRm Project Management and Integration 21
3.1 Mapped Project Management Process 21
3.2 TRANSFoRm Roles 22
3.3 TRANSFoRm Versioning 24
3.4 Software Configuration Management 25
3.4.1 Establish Baselines 26
3.4.2 Track and Control Changes 27
3.4.3 Establish Integrity 28
3.5 Test Planning 28
3.6 Testing 29
3.6.1 Unit Testing 30
3.6.2 Integration Testing 30
3.6.3 System Testing & Acceptance Testing 31
3.6.4 Using Code Coverage Tools 32
3.6.5 Test Teams and Cross-Testing 32
3.6.6 Test Automation 32
3.7 Guidelines for Component Delivery 33
3.7.1 DEV-stage 33

3.7.2 SI-stage ... 34
3.7.3 PAC-stage ... 37
3.8 Integration Environments ... 39
4 Software Components ... 42
4.1 List of Software Components ... 42
4.2 Component Descriptions ... 44
4.2.1 WT 3.3: Security Framework ... 44
4.2.2 WT 4.3: Ontology Service for Diagnostic Evidence ... 44
4.2.3 WT 4.4: Web-based Evidence Service ... 44
4.2.4 WT 4.5: Data Mining Tools ... 45
4.2.5 WT 5.1: Data Quality Tool ... 45
4.2.6 WT 5.2b: DSS Plugin for EHR Tools (Triggering Tool) ... 46
4.2.7 WT 5.3 Query Tool and Data Extraction Workbench ... 46
4.2.8 WT 5.4: Provenance Service ... 47
4.2.9 WT 7.1: Model-based Semantic Mediation Service ... 47
4.2.10 WT 7.2: Terminology Service ... 48
4.2.11 WT 7.3: Metadata Repository ... 48
4.2.12 WT 7.4: Infrastructure for Deployment of eCRFs/Web Questionnaire with EHRs ... 48
4.2.13 WT 7.5: Infrastructure for Extraction/Linkage of Data ... 49
4.2.14 WT 7.6 Middleware for Mobile e-Health and Clinical Prediction Rule Derivation ... 49
4.3 Component Timeline ... 50
4.4 12-Month Integration Timeline: Jun 2012 - Jun 2013 ... 50
5 Summary ... 52
List of Abbreviations ... 55

1 Introduction

During the review of TRANSFoRm it became obvious that such a complex project requires excellent coordination of the software engineering approach and structures that guarantee the technical quality of the TRANSFoRm software implementation. For this purpose the TRANSFoRm consortium created "structures, rules and procedures to objectively ensure a seamlessly controlled and managed software integration, including quantifiable validation and verification measures and procedures"[1]. As central part of these efforts this software integration plan was created. The goal of this document is to provide a description and explanation of the process that should be followed within the TRANSFoRm-project regarding the integration of its software components and sub-systems, including component delivery, staging and verification and validation. The software integration plan will be coordinated with the implementation of the SQA framework as a tool to guarantee quality during development and implementation.

Software development is not done in a monolithic way, but by creating different components independently and then joining them to the final product. Thus integration of these software components becomes a critical step during development. Integration is in essence an iterative process of taking several components and successfully combining them into a larger, working entity until the final system is formed. Although this idea of having an iterative, step-wise process to create a larger system is easily formulated and understandable, the process itself can be quite complex and difficult to carry out. The process is not a fixed solution that is readily applicable to a project, it needs to be created and adapted to the needs of the project. However, more importantly it needs to be followed up and applied as close as possible. Therefore an integration plan is needed, closely tied to the design of the system and its decomposition into sub-systems and components, to improve and guarantee the overall quality of the software.

TRANSFoRm will deliver an interconnected system of sub-systems that together are able to serve the diverse goals of the project. The sub-systems themselves were then divided into different components, forming the subjects of the different work tasks, each the responsibility of one or more development teams. The TRANSFoRm system will be software-based; therefore software development principles apply to it[2].

This first section provides a summary of what will this document will provide. In section 2, the overall strategy of the integration plan is discussed. It is based on the iterative view of the software development process that can easily be adapted to the project management process specified in the Software Quality Assurance Plan (SQA plan) of the project. A detailed plan on how such a cyclic process can be achieved and which are the key roles to make it succeed is presented. Section 3 describes how

[1] See Technical Review Report , 08-09 June 2011, Point 1-c, p. 6

[2] See WT 3.5 Software Quality Assurance Plan

the suggested integration plan can be realised in the TRANSFoRm project. The software components identified in the TRANSFoRm project are described in section 4.

2 Overall Strategy

2.1 Software Release Cycle

Software development is generally accepted to be an incremental and cyclic process that is repeated over and over. Such a cycle is called a 'software development lifecycle' (SDLC) and consists of 5 phases, which can also be found in the SQA plan[3]:

1. Requirements analysis
2. Architecture and design
3. Coding
4. Verification
5. Validation

The development and maturity of a software component takes place in the last three steps: coding, verification and validation. Typically these steps form a smaller cyclic process. This cycle will be repeated over and over until a verified and validated system is produced. From this point on, we will refer to this cyclic process as the 'software release cycle' (or simply 'release cycle').

Several software development methodologies exist that each prescribe different ways to go through that cycle. At the same time, it has proven useful to divide this process into separate, consecutive stages within that cycle.

The number of stages within a release cycle is not fixed, so variations are possible. However, in practice, the following four stages can typically be observed:

1. Development (DEV): the stage during which the implementation of new features is done and/or identified bugs in the software are fixed. This stage includes testing, but is limited to pure unit testing.
2. Integration (SI, Software Integration): the stage during which components are coupled together to see if they can co-exist and/or co-operate as parts of a larger system. The goal is to have a verified[4] system.
3. Acceptance Testing (PAC, Project Acceptance): the stage during which the constructed system is tested on the validity[5] of its actions and its functioning.
4. Production (PROD): the stage in which the software is considered to be working as it was intended when development was started in stage 1 (DEV). A system in this state can be put to use as demo.

[3] See Figure 2 of the SQA plan

[4] Verification of a product/system/component is all about trying to answer the question: Are we building the product/system/component in the right manner?

[5] Validation of a product/system/component is all about trying to answer the question: Are we building the right product/system/component?

Every new software development lifecycle introduces new bits of functionality or improves existing functionality of the overall system. What new features or improvements will be implemented for a next release is determined by the project manager (in cooperation with the team leaders of the development teams that need to do the actual development on the different components). The final goal of a release cycle is to have a stable and tested version of the system that is composed of the different components, with some (or all) providing the intended additional functionality of the complete system. This stable and tested version is the actual release.

To obtain a tested and stable version, software testing is required. Software testing is too important to leave to the end of the project, and the V-Model of testing (see Figure 1) incorporates testing into the entire software development lifecycle.

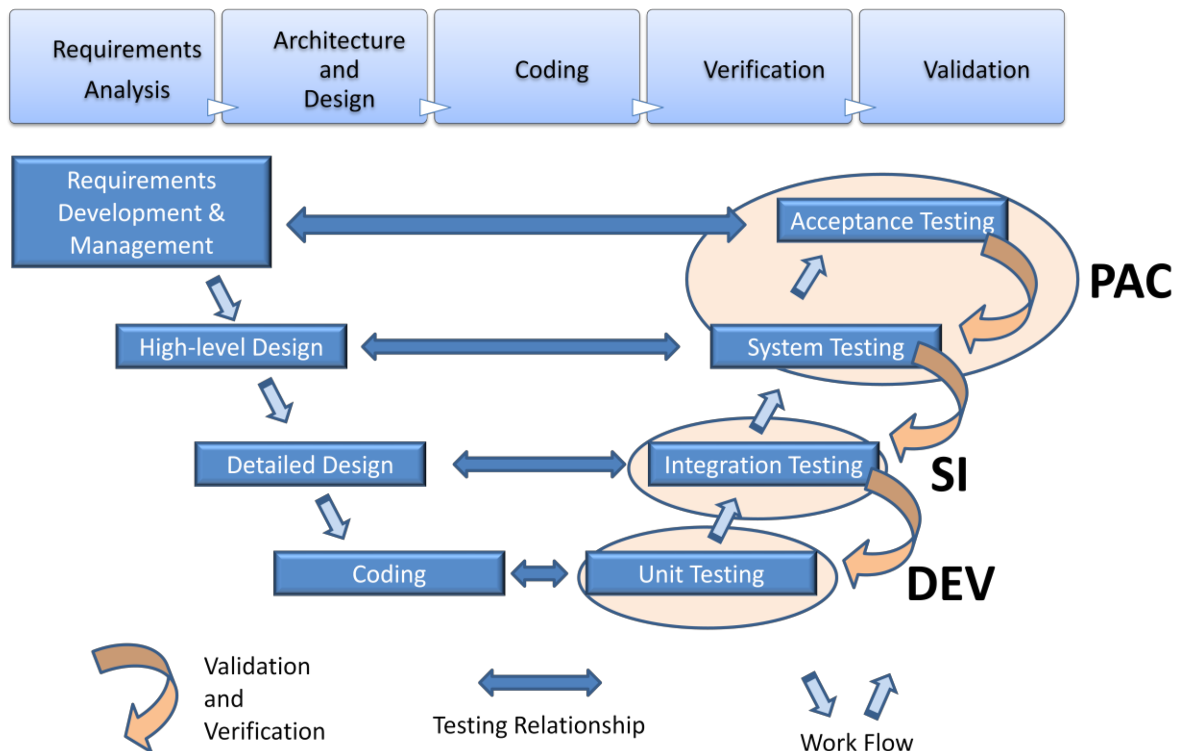

Figure 1: The V-Model of Software Testing

In a diagram of the V-Model, the V proceeds down and then up, from left to right depicting the basic sequence of development requirements and the corresponding testing activities. The model highlights the existence of different levels of testing and depicts the way each relates to a different development phase. Like any model, the V-Model has detractors and arguably has deficiencies and alternatives but it clearly illustrates that testing can and should start at the very beginning of the project. This is the first important step; the results of requirements gathering, like the derived requirements, product requirements, component requirements feed into the technical design. System and acceptance test plans - to be carried out in the PAC-stage - are written during these phases. When detailed specification

is ready, test cases for integrating testing are written, which will be used in the SI-stage. The developers write and run unit tests in the DEV-stage. The model illustrates how each subsequent phase should verify and validate work done in the previous phase, and how work done during development is used to guide the individual testing phases. This interconnectedness lets us identify important errors, omissions, and other problems before they can do serious harm.

One can make a distinction between executable and non-executable models. A model that can be generated into a working application without writing a line of code is called an executable model. Non-executable models typically create a set of classes and methods that can serve as a foundation for building an interface and logic layer. Non-executable models, like the Clinical Data Integration Model (CDIM)[6] and Clinical Research Information Model (CRIM)[7], are not integrated into the system as such, they are rather being used by the system. Therefore only executable models are considered in the release cycle.

The release cycle starts in the DEV-stage where developers start to work on their implementation work that addresses a single item (feature, improvement, bug fix) on the developer's task list. At a certain point they consider their work to be ready (i.e., successfully unit tested) to be integrated with the other components of the system. The developed component goes into the SI-stage. The SI-stage is considered to be finished only if the envisaged system is verified (the product/system build correctly), i.e., all of its components are up and running without any conflicts between them. If there are problems to get the components integrated, the component must go back into DEV-stage. Sometimes the added functionality of two (or more) components can be conflicting, therefore preventing the system to run. A conflict like this needs additional work from the developers, which warrants the return to the DEV-stage. Only when the changed components are integrated into the system and the system is proven to be running and working as expected by depending parties (i.e., it does not implement a different behaviour than expected), will it go into the PAC-stage. It’s not just about trying to get the system working; there should also be tests that can ensure the components that need to work together can actually “find” each other (integration tests).

The PAC-stage is all about testing. As DEV and SI also have a testing aspect, PAC is the stage where the system is validated (is the product/system doing what was intended?). The added and/or improved functionality is tested against the software requirements (system testing). Additionally, PAC is also the stage where user acceptance testing is executed. Issues found during PAC testing should be reason to send the system back into DEV-stage, with the intent of having the issues fixed. Of course this means that once the issues are fixed in DEV, it has to go through SI again, before it comes back into PAC where it will be tested again.

At each stage of the release cycle, regression tests are conducted. Regression testing tests that the rest of the application up to the point or repair was not adversely affected by the fix. Regression testing is conducted in parallel with other tests and can be viewed as a quality control tool to ensure that the newly modified code still complies with its specified software requirements and that unmodified code

[6] see Work Task 6.5 of the TRANSFoRm project
[7] see Work Task 6.4 of the TRANSFoRm project

has not been affected by the change. It is important to understand that regression testing does not test if a specific defect has been fixed.

Once no more issues are found and/or an agreed level of stability and functionality is reached (some issues found during testing might be too costly or just not significant enough to fix in the current release cycle, typically meaning that there is a workaround for the issue) the system can be considered ready for production (PROD). A system that reaches this stage is considered to be ready for use. All implemented functionality is working as expected and the system can be disseminated. Note that mostly it is not known upfront how many times a system will need to go through the SI-stage, however it is common practice to foresee at least a couple of visits to the PAC-stage within the planning, so the number of visits to the SI-stage is always at least as much as the foreseen number of rounds for PAC-testing.

2.2 Roles

In order to ensure adherence to the above process, it is important to have certain key roles in place. The green light to move a component up one stage should only be given when people with a certain responsibility agree to do so. If there is no agreement then the component should stay in the current stage (or fall back to the DEV-stage). There should be a sufficient number of roles to ensure effective supervision of the process, but not too many as that would make the process inefficient. The following roles should in general be sufficient to guard the process:

Necessary roles for release process	Description
Project manager	Person overseeing the entire project
Component owner	Person responsible for a software component within the project
Release/integration manager	Person responsible to guard the release/integration process
Developers	People responsible for implementing the requirements of the project
Test team leader	Person responsible to supervise the testing of the system
Testers	People responsible to execute the testing

Table 1: Roles in a release process

The **project manager** is closely involved with the overall planning of the project. The key is that this person should be able to adapt the planning if necessary. Components that were planned to be available (i.e., stable, in PROD) at a certain date should be, if not then an appropriate reaction is needed towards the planning, such as makes changes in the schedule. Another option as opposed to rescheduling is to keep the deadlines, but to limit the scope of what is delivered in an intermediate release (e.g., less features, workarounds instead of fixes...). This requires interaction between project manager and component owners.

The **component owner** is the person responsible for a specific component within the project. This role is important within the suggested strategy/process as he/she bears the responsibility of the state of a component. The component owner agrees with the project manager on the release planning of the component: what features, bug fixes go in which release, when releases are scheduled,... It’s the component owner who decides when a component is considered to be ready to transition from one

stage to another during one release cycle. This typically requires knowledge about many different development aspects. As such, the component owner is typically the team leader of the responsible development team. It is common that the work that needs to be done for one release of the component will be executed by several developers, each implementing certain functionalities or fixing a certain bug. One developer is bound to finish sooner than the other, but the team leader is the one that has the overview to see if all the development for the current release cycle has been finished. The component owner should be the one informing the project manager and the integration manager that the component can go into SI-stage.

The point of passing the integration stage is that a higher level of quality (improvements, extra features, etc.) is attributed to the components of the system. A good way of doing this is by having a certain level of independence between the component developers and those who do the component integration. Therefore it is advisable to have the actual integration done by combined component teams, i.e., part of the component team and part of another component team. Which components need to be integrated and when, is defined by the **integration manager**. Component owners report the integration status of their component to the integration manager, who keeps the overview of the overall status of the system during integration. If teams fail to deliver on time, the integration manager reports to the project management, who can adapt the planning or take corrective actions.

The **developers** do the actual implementation of the functionalities that were planned to be implemented for the upcoming release of a component. One important thing to note is that developers should also do the first round of testing their own code (called unit testing). Code is considered to be working if and only if it passes its own unit tests. As long as there is code failing a certain unit test, the component should not be deemed ready for SI. Actually if unit tests are not passing, code should not be checked in!

The **testers** are ideally composed of members of a separate team responsible for executing tests; they are not the same people who did the implementation. They are mainly involved during the SI-stage and the PAC-stage, executing different kinds of tests during each stage. However, because of the limited resources within a project it is not uncommon to do a kind of “cross-testing”, also referred to as “X-testing”, meaning that the different teams will test each other’s components, again trying to maximise the independency between the test team and the component's development team (as also discussed for the integration process). In the same way the development teams should communicate through the team leader (the component owner), the testers should communicate through their **test team leader**. As the test teams might be assembled from the actual development teams, the roles of test team leader and component owner leader can be combined in a cross-testing setup. The subject of testing will be discussed later on. However it is not the task of the integration plan to specify the testing strategy of the project, as the integration plan is only part of the overall testing strategy of the project.

2.3 Environments

So far we have postponed the subject of software environments as this topic is tied to the "stage" concept introduced in the previous chapter. With every stage, an environment corresponds. This implies that there should be four distinct environments:

1. PROD
2. PAC (also called “Staging”)
3. SI
4. DEV

How these environments are connected, is depicted in Figure 2.

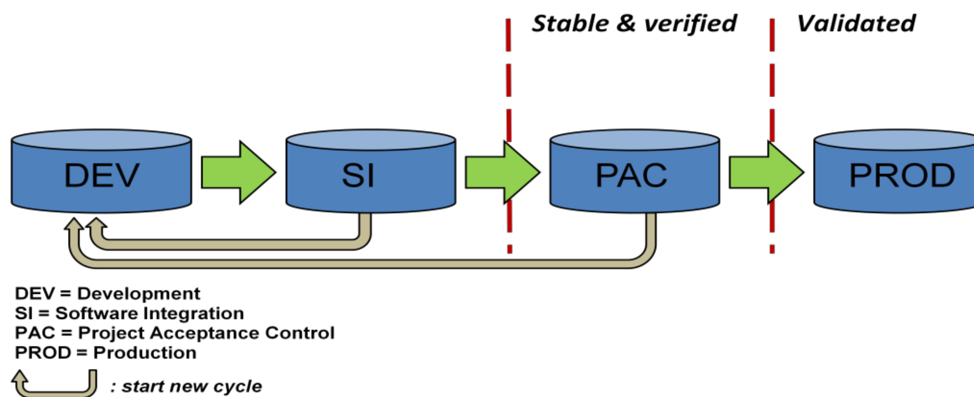

Figure 2: Software environments

Every environment has its prerequisites regarding the systems (hardware/software) it is composed of. Ideally environments share the same runtime and infrastructure components (e.g., operating system version, application server, virtual machine runtime, ...), but budget and timing restrictions may influence this distribution (e.g., PROD could have a replicated database setup while PAC might do without it for budgetary reasons).

2.3.1 Production Environment

Production should ONLY contain components that reached the PROD-stage (normally the deployment onto the PROD environment is the act that signifies the component being in PROD-stage). It is (should be) basically a promotion and literal copy of the acceptance environment once approved. PROD is the environment where the actual system will run on.

2.3.2 Product Acceptance Environment

This environment is also called the "staging environment". PAC should have as close a resemblance as possible to the PROD environment, this means that every component that is not part of a new release cycle should be stable, versions of installed software (operating systems, database systems, language runtime systems, ...) should be exactly the same as on PROD, as well as the used data (for example, data inside the databases). If multiple systems are involved (firewalls, proxies, ...) then this should be imitated as well.

PAC is used to do the testing involved during the PAC-stage. The result should be that a component is considered stable and ready to be deployed onto PROD. The best way of doing this is by mimicking PROD as much as possible[8]. The PAC environment should only be allowed to contain PAC versions of the components and/or PROD versions (more on versioning see 2.4).

2.3.3 Software Integration Environment

The SI environment's main focus is to provide an environment where the versions of components, that are ready to be integrated with other components, can be deployed. SI should mimic the PROD environment as well, but the main goal is to see if the components for a release can work together and/or coexist. Therefore uniformity in the versioning of the installed runtime and infrastructure software (operating systems, database systems, language runtime systems ...) is important.

It's important that only versions of components that are deemed to be ready for the SI-stage (or PAC or PROD) are deployed on the SI environment. No DEV versions should ever come on SI (more on versioning in 2.4)! This implies that release planning should avoid that new releases from different component teams have cyclic dependencies.

2.3.4 Development Environment

DEV is different from the other three environments, in the sense that DEV is under the control of the development teams. This environment is used by the developers to test their newly created code. In most cases there is not one DEV system, but each developer has an individual system set up to test if changes in the component he is working on are working.

On DEV every version of the components can and will be used (SI, PAC or PROD), as issues need to be resolved in DEV regarding these versions.

2.4 Versioning

The previous section about the testing environments already suggested that versioning of the developed components is very important. Together with the separation of environments it forms the basis to guarantee a coordinated and structured approach towards a software release cycle. Typically software code, written by the developers, will be kept in a so called version control system, also known as source-control system or revision-control system. Such systems help to maintain different versions of the code. Different developers are working on certain code intended for the same upcoming release of their component and check their changes into the source-control system. At a certain point the component owner will decide that the goals of the current development cycle of a component (implemented features addressing certain requirements, bug fixes) have been met. At that point it will be clear which versions of the committed source code files need to be included and the component owner can give the green light for actually building the component. A build should be seen as a "marking" of the code used for creating a certain version of the component. It is this "marking" that is the actual versioning that is

[8] Except maybe for system testing in the form of load and performance testing as the PROD-environment mostly has a bigger capacity and is probably a bit more powerful as well. Load and performance testing are mostly done to establish an upper boundary on the execution.

so important to coordinate the different release cycles of all the components involved in the project. This versioning should follow a certain scheme.

2.4.1 Versioning Scheme

The SQA plan provides a reference[9] to certain guidelines. These guidelines describe some possible versioning strategies/schemes (single-level and multi-level), but the SQA plan does not enforce a versioning with regards to software creation (i.e., the source code). Because agreeing on a common methodology for component versioning is highly advisable for correct and efficient interaction between multiple component development teams and increases development transparency, a special versioning scheme is suggested.

A common (multi-level) versioning scheme of software (components) is outlined in Table 2:

<component_name> X.Y.Z	
<component_name>	the name by which the component is known inside the project
X	the major release version number
Y	the minor release version number
Z	the “bug fix” release version number (not always used, as bug fixes could be seen, or planned to be fixed during a major or minor release)

Table 2: Software component versioning scheme

The major and minor releases are the ones that are usually planned upfront by the project management. What is considered a major release and what is considered a minor release of a component (or the complete system, which can be seen as an all encompassing component), is up to the project management to decide. In general, a major release is performed when drastic changes have been introduced or a major piece of functionality is added or removed. Also if a new release breaks backwards compatibility, it should be considered a major release. A minor release is typically performed when some minor functionality is added or improvements are made to a component without breaking backwards compatibility.

Bug fix release is an unplanned release in the sense that some component on the PROD-environment seems to be not working properly, the problem is assessed and the fix can be done quickly in the source code. Then, a special, short release cycle is scheduled to fix the bug, build a new release candidate (almost identical to the previous release candidate, only a small piece of code was adjusted that won't change deployment behaviour), test the fixed code (mostly directly into PAC) and once passed the test, released into PROD. This special release will warrant an increase of the bug fix release version number (Z). During the beginning of a project, working with bug fixes is often overkill. But once a full PROD-environment comes into play, a separated “bug fix release cycle” (DEV -> PAC -> PROD) could be

[9] http://ontolog.cim3.net/file/resource/reference/WorldBank-DeniseBedford-doc/Final_Versioning-Strategy.pdf

introduced as an additional feature. In this case, the choice can be made to not use the Z-index in the versioning scheme or always use 0 (zero) as the value of the Z-index, until it becomes necessary.

Although the major, minor release version numbering scheme seems sufficient, more sophistication is needed to differentiate the builds of the components on the different systems (SI, PAC or PROD). A component with version number x.y.z can be in the fully tested state PROD, but it can also be in the untested state DEV. It is however required that the state of a component is known at all times, because an unstable version of a component can cause an unstable system. The use of additional suffixes offers a solution to this problem. The following additional suffixes could be used:

- SNAPSHOT: used for DEV
- RCx: stands for Release Candidate and should be used in SI and PAC. “x” stands for the number of the release candidate[10]
- FINAL: indicates the stable version of the component

The suffix SNAPSHOT indicates to anybody who is confronted with such a version of a component that this version of the component should not be considered as stable and is still subject to regular changes. It’s the indication that this is something that is still in the DEV-stage.

The RCx-suffix (release candidate number x) in the version of a component indicates that this version is allowed outside the DEV environment. It states that this version is intended to be put on SI or PAC. It is a so-called release candidate. It is not yet a proper release as it still needs to be tested, but it has the potential to be the actual release if it passes the tests. The “x” indicates which build of the release candidate it is, as it is possible that several passes through SI and PAC are needed before the candidate is considered stable and earns the FINAL-suffix. The FINAL-suffix indicates that this version of a component is considered stable and has passed all tests. This suffix may be omitted, as the lack of a suffix in the version number of a component can be considered equal to being a FINAL version.

Everything that still has an RC-suffix or SNAPSHOT-suffix should be considered unstable. Therefore these suffixes are a useful remedy to prevent unstable building blocks to end up in the final “product”, weakening the complete structure.

2.4.2 Builds

The term software build (or simply 'build') refers to the result of converting source code files into standalone software artefact(s) that can be run on a computer. The stage transition process should only be triggered on the basis of the outcome of a particular build of the component. If the component owner claims the development of his/her component has finished, he/she should request an RC-build from the integration manager. Typical for such a RC-build is that it should not contain any code from that component that is not part of the requirements of the upcoming release and preferably only code that has been (unit) tested by the developer(s).

By checking the major/minor version numbers, implementations of requirements of an unwanted version can be prevented from ending up in the current version's build. When using a version-control system like Apache Subversion (SVN)[11], branches can be used to create separate development lines for

---

[10] Release candidates are sometimes also indicated with a suffix CR instead of RC, for example in Maven

[11] http://subversion.apache.org/

multiple versions of the same component. Builds are then more easily limited to include implementations of only one version. For example, by building the code on the 1.2 branch, no code that is in the 2.0 branch (and probably still experimental) can end up in the builds for release 1.2.

At a certain point there will be a built x.y-SNAPSHOT available during the DEV-stage of the version x.y of a component. The component owner knows this is the version containing all the necessary development efforts, meaning that everything in that SVN branch at that time is considered finished (by the developers). To capture the state of all the source code at that moment in time in the SVN branch of the x.y release a new tag should be created in SVN. The tag should be called:

x.y_RC1

Based on that tag, which ensures that all the correct versions of the source files are used, a new build of the component should be created and get the same version number and tag:

x.y_RC1

If the build has a dependency on another component, then the version of this dependent component used during the RC-build should be an RC-version or a FINAL-version of the other component. If no such version exists yet, and only SNAPSHOT-versions are available, then the build should fail. Such a failure of the RC-build prevents unstable versions of components to enter SI, PAC and PROD-stage. This is important as it ensures that certain “super components” get to be considered stable, while actually they are not and still contain unstable, experimental subcomponents. Normally the tests during the PAC-stage should help to uncover such unstable components, but the sooner it can be prevented, the better. The longer an issue remains undiscovered during the release cycle, the costlier it gets to fix it. Again this implies that component release planning requires careful consideration not to introduce cyclic dependencies between components.

If an RC-build fails, the responsible component owner(s) should be notified and they should fix it. For example, if a certain subcomponent does not have an RC-version yet, then this one should be build first, so that the component depending on it can be built as an RC as well.

Eventually an RC version will be available. The result of the build will be some kind of artefact (e.g., a jar file) that can be deployed. Deployment instructions need to be available such that the artefact (being an RC) can be deployed on the SI-environment without requiring the direct intervention of the involved development team. This deployment should be done by an independent party (another team that was not involved in the development of this component).

The deployment of the RC-build is an additional test of the component. It tests if the accompanying deployment instructions (could be a script or just a written procedure to follow) are correct, if no additional requirements of the target environment were needed but were forgotten (for example a missing library, database, web service),... A failure to deploy should be reported to the involved component owners, which will result in, probably, a new build for the component (increasing the RC version number).

Once the deployment has succeeded, the component owners should be notified. Once all new versions of the components intended to be released at the end of the current software development lifecycle are deployed, additional verification tests should be done. A successful build and a successful deployment are already very good verification tests. However, some additional tests are necessary before a component can be considered verified. These tests can be integration tests[12] and also some regression tests, especially for the components that are not involved in an upcoming release. If one or more of these tests fail, it means that the changes to the components involved in the upcoming release have "broken" the component(s) that were not changed for this upcoming release cycle.

Every failure should be reported to the project management and the component owners as proper actions can be taken (sometimes a certain failure might be expected and may have been already planned for in a future release, so not every failure must warrant a return to the DEV-stage).

Once everything is verified by the project management and the component owners, the transition to the PAC-stage in the release cycle can be made. This means that the latest RC-builds that were deployed on the SI-environments should now be deployed to the PAC-environment. Deployment should again be done by an independent team. The deployment should go without issues, as this is the whole point of SI, but to be absolutely certain the verification tests from SI could be repeated on PAC, especially if there are known environmental differences (e.g., load balancing in PAC, but not in SI).

A successful deployment on PAC should start the functional acceptance testing. This includes system testing[13] and regression testing[14]. Ideally, functional tests are generated automatically from test specifications (tools exist for that). PAC is also the stage where (end) user acceptance tests should be executed. Ideally user acceptance tests would be scriptable in order to make them repeatable.

If a return to DEV is necessary the whole process starts all over again. A new RC-build is made to enter SI, which should then go to PAC if nothing goes wrong.

If the project management and the component owners agree that the latest RC version in PAC is indeed sufficiently stable to go to PROD, a new build should be made to create the FINAL version. The build process for a FINAL release will be similar to the one for RC-builds. Labels should be set in SVN to indicate the stability of the source code and a FINAL version should be built for each component involved in this release. This build should only succeed when FINAL versions of the dependencies (libraries and/or other components) are available. If a dependency on an RC version still exists, then the build should fail. A dependency on a SNAPSHOT should fail the build as well, however because of a successful RC-build this should never happen at this stage.

---

[12] Integration testing tests the interaction with other parts of the complete system. The fact that a result is received will prove if the different components can actually cooperate. Whether the returned result is functionally correct according to the requirements, is not so important at this point.

[13] System Testing tests all components and modules that are new, changed, affected by a change, or needed to form the complete application.

Once successful FINAL versions are available they should be deployed on the SI and PAC environments first, replacing the RC version of the components. This will ensure that these systems are a good replication of the PROD environment. This "synchronising" of the systems is important. At this point the planned end of the release cycle should be in sight, which should coincide with the deployment of the FINAL version(s) of the components in the current release to the PROD environment.

Figure 3 shows a detailed overview of the software release process:

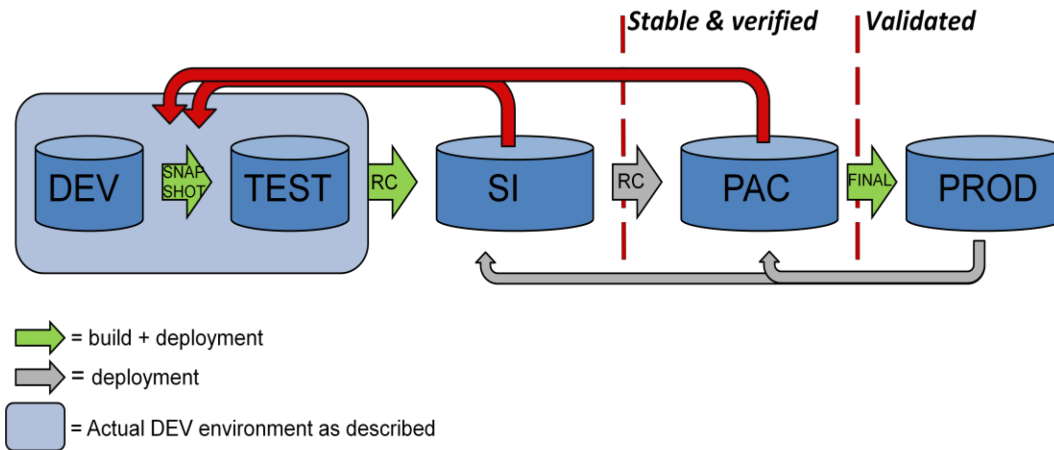

Figure 3: Software release process (Additional TEST-environment is just to clarify that DEV needs to test as well. DEV+TEST = DEV as described in section 2.3).

Test results should always be bundled in a report for the component owners and the project management and linked to requirement specifications for reasons of traceability. Certain failures might be expected, so an additional roundtrip to DEV-stage might be unnecessary. If certain results are not acceptable, the current release cycle has to go back to the DEV-stage. It should be noted that the further in the release cycle, the roundtrips to DEV will normally grow shorter.

From section 2.4.2, it should be clear that the versions are an indication of the release cycles that are running (major/minor version number) and the stages (DEV, SI, PAC and PROD) that have been visited in these release cycles are indicated by the suffixes in the version number[15] (remember that the FINAL-suffix is not necessary as lack of any suffix can indicate the finality of a version as well). As an addition, it should also be clear that there is a direct relation to the suffix in the version number and the environments. The following rules can therefore be deduced from this:

- SNAPSHOT-versions should never be used outside the DEV environment.
- SI and PAC environments can only have RC- or FINAL-versions of components installed/deployed, never SNAPSHOT-versions.
- PROD environment should only have FINAL-versions on them.

[15] Versioning scheme is described in 2.4.1

These rules should be followed to ensure a good progress of the release cycle of any component[16].

Continuous Integration

Continuous integration is a technique that is mostly used during the development stage in a release cycle. As different developers are working to implement new or improve existing functionalities of a component, a source control system is used. Every time a developer considers his/her code to be working (verified by using unit testing) he/she checks it into the source control system. To prove that is does not break the component he should create a new build for it (a SNAPSHOT-build). A successful completion of the build process[17] will prove to his/her colleagues and the component owner that the component can still be delivered as whole with the changes he/she made in it.

The process of starting a new build after code changes are checked in (i.e., made available to the other developers) is called continuous integration, as it tries to integrate the code changes into the component. There are several systems available that can automate this process (e.g., for Java: Hudson[18], Jenkins[19], Cruise Control[20], Apache Continuum[21]).

It is only during the DEV-stage that SNAPSHOT-versions are built, but a similar process can be used to create the RC- and FINAL-versions. The following steps are necessary:

- Check out the latest code, when a new check in was done
- Start the build
- The version that needs to be build should be indicated (done by a developer or the component owner) in some "build file"
- Compile the code
- Run the unit tests (additional checks can be done as well, for example check code conventions)
- Package the build (producing the artefact, documentation files can be included as well)
- Publish the build (to some repository)
- Publish a build report (containing any failures of one of the previous steps)

There are several tools available to build deployable artefacts from source code. Apache Maven[22] and Apache Ant[23] are examples of such build tools. Maven also provides dependency management, and a master build-file can be created to define project specific tasks. This way it is possible to automatically check for SNAPSHOTS when doing RC builds, applying correct building labels, etc.

[16] TRANSFoRm can be seen as the super component from a project management point of view.

[17] A build process should always contain a phase where it tests the code as well. Unit tests can easily be incorporated into such a build process. If all the unit tests succeed, then the build is allowed to be considered a success. If one test fails, then the build should fail and therefore not produce a new version of the component.

[18] http://hudson-ci.org/

[19] http://jenkins-ci.org/

[20] http://cruisecontrol.sourceforge.net/

[21] http://continuum.apache.org/

[22] http://maven.apache.org/

[23] http://ant.apache.org/

Ant does not have dependency management built in, but it is fairly easy to incorporate a dependency manager like Ivy[24] into the build.xml configuration file.

Documentation

The build process should not be restricted to producing software artefacts only, but should encompass things like documentation, automated testing reports, code quality reports, implemented features and bug fixes lists, code coverage reports, initialisation and migration scripts, ...

Documentation at the beginning of a new stage is important. However documentation at the end of a stage is equally important. Documentation at the end of a stage provides valuable information to developers, component owners and project management.

2.5 Interfaces, Stubs and Project Planning

Components implement certain functionalities. Some functionalities need to be made available to other components. This is done via interfaces. Interfaces are important as they provide a kind of contract regarding the way the functionality of a component can be used, and how the format of the result will look like.

The point of the integration stage (SI-stage) is to see if the components that need functionalities of other components can actually find the interfaces that encapsulate the desired behaviour. The end of SI should guarantee that the different promised interfaces for the upcoming release are available and can be reached. However, SI does not need to test the actual functionalities behind interfaces. This is done during the PAC stage. If no functionality is needed during SI, then how can all this pass SI stage? The key is to use stubs or drivers (mock tests).

With stubs, the actual functionality behind an interface of a component will be empty, in the sense that nothing will be done, but there will be a result returned, so that the component that calls this interface will be able to carry on. With drivers one can specify expected behaviour.

Stub testing can (and should) be taken into account within the planning of releases. For example, component A needs a certain functionality of component B, in order to guarantee certain functionality by itself. However, it was not planned to even start development of component B for at least one year, so what can be done? This is where stubs come into play. The interface for that wanted functionality should be determined and an initial version of component B should be created. This initial version will just be a stub of the interface. This initial version should obviously be a minor release version (e.g., 0.1). By applying this work logic, this initial version (being just a stub) can be taken into the project planning and the described release process can be used for it as well. As long as component B does not reach major version 1.0, it should be an indication to the project management that there is still some important functionality missing from component B, in this case it could indicate that there are still some stubs that need to be actually implemented before version 1.0 is reached.

[24] http://ant.apache.org/ivy/

Using stubs for a certain functionality of a component does not mean that there will be no RC builds necessary for that component. Even if a component only has one stub it should still follow the above described release process.

3 TRANSFoRm Project Management and Integration

3.1 Mapped Project Management Process

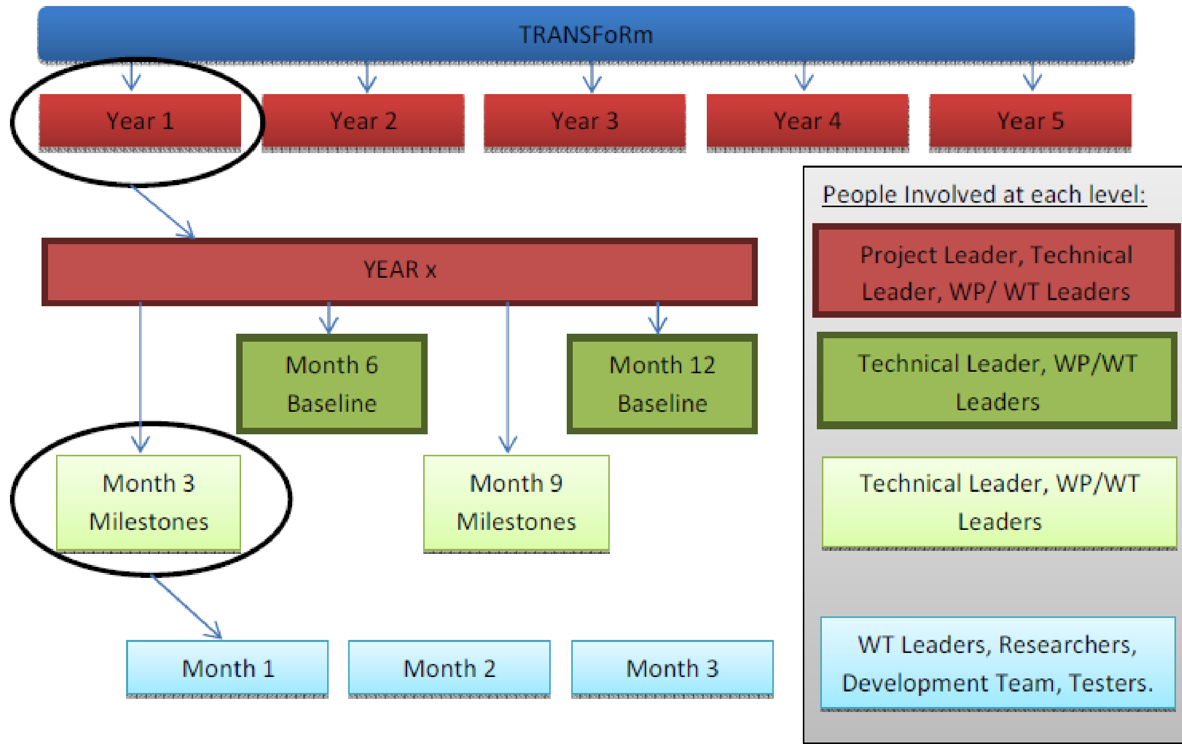

Figure 4: Project management process from SQA plan

The project management process from the SQA plan foresees 6-month, incremental, software development lifecycles of the overall TRANSFoRm system which deliver prototypes. During month 1 (most likely before the start of a cycle) project management and the component owners (team leaders) will decide upon what will be implemented for which components during the coming year. Some components may be excluded, as they might still be in a pre-development stage or because they first need the functionality of another component that still needs to be implemented. In case of the latter, using stubs could be a helpful mechanism (see 2.5).

There is a "baseline" set every 6 months (month 6 and 12) for each component involved for the upcoming release. The goal at the end of each 6-month period is to provide an incremental and functional prototype of the software being developed. To be sure the baselines are reached (or to be able to adjust the planning) a milestone is set every 3 months for each component. The idea is that every 6-month period the TRANSFoRm system will actually grow (functionally) as components have been delivered and should be stable (ready for PROD) at that point. At month 6 or 12 stable versions of the planned minor or major version of all the components involved should be ready. So the baseline corresponds to the final version of a major or minor release of a component (whether it is a major or a minor release depends on the implemented functionalities and is up to project management and component owners to decide). Final versions are markers for the source code as well, as they indicate

that a version of a specific source code file can be considered stable (as it corresponds to a final version of the component it is part of). Mostly when developers start the development of a new development lifecycle, they will want to start from a known stable version of their code, they want to start from a certain "baseline" (hence the term "*baseline*", which in TRANSFoRm denotes a stable version of a certain component).

Before the start of a lifecycle (the 6-month period) the requirements that should be met at the end are identified (done by project management and component owners). The work is partitioned into tasks for the team by component owner, which are then assigned and scheduled within each team of developers. How this work is divided and distributed internally is the responsibility of the component owner. For example, a component could be composed internally of several subcomponents all going through their own release cycle, but by the 6th month the component of the work task should have gone through DEV, SI and PAC at least once. At month 6 a PROD-ready version of his/her component should be available.

At the month 3 milestone meetings, it should be clear to the component owners and project management whether the planned requirements can be implemented and tested by month 6. Preferably, the involved components should already have entered the SI when the milestone meetings are held. As SI requires a lot of coordination - although probably to a lesser degree at the beginning of the project - it should be taken into account for the project planning. As explained in 2.4.2, multiple roundtrips to DEV might be necessary in order for SI to be able to complete. Therefore, development of a component does not end after the first exit of the DEV-stage. The same is applicable to SI.

3.2 TRANSFoRm Roles

The roles suggested in section 2.2 can be mapped to those suggested in the SQA plan and Description of Work (DoW) of the TRANSFoRm project. Some people can take on multiple roles, or one role can be split up over different people. However the goal should always be to stay as independent as possible towards the development process, meaning that for a certain component, except for the role of developer, people outside the team, responsible for the component, should take up a "release" role corresponding to his/her "project" role. The SQA plan mentions "individual project teams". This corresponds to a WT team, i.e., the component owner and his team of developers/testers. The following mapping of roles (see Table 3) is suggested:

TRANSFoRm project role	Release process role
Scientific Project Manager	Project manager
Technical Director	
WP Leader	Component owner
WT Leader	
Integration Manager	Release/Integration manager
Technical Director	
SQA Manager	
WT Teams	Developers
WP/WT Leader (other components)	Test team leader
SQA Manager	
Testers	Testers

Table 3: TRANSFoRm project roles

The role of **project manager** in the release process can be taken up by the Scientific Project Manager and the Technical Director of the project. They are both closely involved with the overall planning of the project.

Because of the partitioning of the TRANSFoRm project, the role of **component owner** corresponds to the role of the WT Leader or sometimes also the WP Leader. At the start of the project several components were identified that need to be delivered as part of a Work Task (WT), which by itself is part of a Work Package (WP). Therefore it can be that one WP Leader has the responsibility of delivering more than one component. Whether he/she takes on the role of component owner of all these components is not important, however he/she is responsible to appoint someone who will be the owner. As the SQA plan suggests, requirements will be developed by the WT Teams. The component owner and his team will develop requirements and tests in conjunction with those component owners who are depending on the particular software and with the project manager.

The role of **test team leader** can be executed by the component owner of a different component then the one being primarily tested by his/her development team, as the test teams are the actual development teams. The test team leader sends all test results he receives to the SQA Manager. In the PAC phase, testing will be done by under supervision of the SQA Manager.

As the SQA plan suggests[25], the project manager will propose a schedule for integration. Deployment of components will be done by an independent party (another team that was not involved in the development of this component). The task of integration testing is to be done by the WT Teams themselves under supervision of the component owners. The owners of the involved components should keep in close contact with each other to avoid any problems. The component owner reports the status of the component to the **integration manager**, who keeps an overview of the overall status of the system during integration. The technical overview is the responsibility of the project manager. The SQA Manager keeps an overview of the documents and test results of every component, verifying that the

[25] WT 3.5 SQA plan, p. 4

quality level is sufficient. If teams should fall behind, the integration manager reports this to the project manager who will follow up.

The **developer** role is part of the actual development teams who will implement the planned functionalities for the upcoming release of the component.

The **tester** role is mapped to the appointed testers of the TRANSFoRm project. All test results should be submitted to the test team leader, who in turn will send them to the SQA Manager. The validation testing in PAC-stage will be supervised by the SQA Manager.

3.3 TRANSFoRm Versioning

The SQA-plan requires that each document generated in TRANSFoRm "belongs to a Work Task, has an author and is reviewed by a TRANSFoRm team member. Reviewers should not be part of the team who is working on the Work Task that produced the document." All documents exhibit version control. Document version numbers have following format:

1. main version number, corresponding to year of development/set of features developed
2. secondary version number, corresponding to 6 month baselines
3. tertiary version number, corresponding to release numbers within the Work Task.

The versioning scheme for the software components in TRANSFoRm should be like the one presented in section 2.4.1. A "bug fix" release version number will not be necessary. Should it prove to be needed later on in the project, it can still be added without compromising the previously versioned components.

<component_name> X.Y-SUFFIX	
<component_name>	the name by which the component is known inside the project
X	the major release version number
Y	the minor release version number
SUFFIX	**SNAPSHOT:** used for DEV
	RCx: stands for Release Candidate and should be used in SI and PAC. "x" stands for the number of the release candidate.
	The absence of a SUFFIX means that the version is final.

Table 4: TRANSFoRm versioning scheme

Note that this versioning scheme should be used for all components identified in the TRANSFoRm-architecture. Individual teams are allowed to use a different scheme internally, but the component they deliver in the TRANSFoRm-environments (SI, PAC or PROD) should adhere to the versioning scheme described in this section. Advanced build systems[26] provide the necessary support for organising this.

[26] A build system is one or more applications or batch files that automates the creation of software builds (see 2.4.2).

3.4 Software Configuration Management

Software Configuration Management (SCM) reflects the current configuration and status of the software at any time, controls any changes to any object of the software, and maintains the traceability throughout the whole software lifecycle. SCM ensures the integrity and consistency of the software throughout the manufacturing process and ensures the availability of any software configuration at any time.

Software Configuration Management is the process used to understand, document and control the components of a software system. SCM answers the following questions:

1. *What constitutes "the system" at any point in time?*
2. *What changes have been made to get us there?*

The importance of Configuration Management CM) is often underestimated. However, is it crucial to have a clear view on the status of the system at any given time. Classic CM consists of four main parts[27]:

- Configuration Identification: What do we have, and how is it structured?
- Configuration Control: also known as "Change Control"
- Configuration Status Accounting: Documents on configuration, versions, change history.
- Configuration Auditing: Review of documents and systems

For Software Configuration Management, there are some additions to the classic CM. The use of a code repository, a versioning system, version description documents and release notes for each version are required.

The SQA-plan proposes the CMMI[28] Configuration Management Process Area to be applied in the TRANSFoRm project. This support process area at Maturity Level 2 has the purpose to establish and maintain the integrity of work products using configuration identification, configuration control, configuration status accounting, and configuration audits.

Identification of baselines for different work products is crucial, and allows team to control changes from baselines while maintaining the integrity of the baselines through the use of configuration management. In essence, establishing a baseline provides a defined stable basis from which product evolution can continue. From TRANSFoRm's perspective, there needs to be a certain degree of guidance and uniformity which needs to be provided to the teams to ensure cohesive configuration management across all teams.

To achieve the Specific Goals (SG) specified by the CMMI CM Process Area, some Specific Practices (SP) and possible implementations are suggested.

---

[27] R. Aiello, L. Sachs, *Configuration Management Best Practices: Practical Methods that Work in the Real World,* 2010

[28] Capability Maturity Model Integration: a process improvement approach whose goal is to help organizations improve their performance.

3.4.1 Establish Baselines

In this section, specific practices which can help the teams to identify and establish baselines are discussed. Establishing baselines is crucial to ensure useful configuration management. The practices of tracking and controlling changes and ensuring integrity can only take place correctly after baselines are established.

3.4.1.1 SP 1.1: Identify Configuration Items

CMMI defines a configuration item as "an entity designated for configuration management, which may consist of multiple related work products that form a baseline." We need to identify items and work products which will need to undergo configuration management. These may include products delivered to the customer, internal work products, acquired products and tools used in the work environment. From a TRANSFoRm perspective, key items also include deliverables to other teams within TRANSFoRm itself. Establishing configuration items includes the identification and selection of work products, assigning unique identifiers to each of these items, identifying the owners responsible for producing each of the items and documenting important characteristics of each item.

3.4.1.2 SP 1.2: Establish a Configuration Management System

A configuration management system is essentially the set of processes and tools used for accessing the configuration system. This generally involves a change management system which records and provides access to change requests to the configuration system or items within the system. This may also involve establishing a mechanism to provide different levels of control to the system.

Through the SVN-environment that has been set up by KCL, we can retrieve the configuration items through check-out command. To make changes to a file you must first check it out of the SVN database. When you check-out an item, SVN retrieves the latest copy of the file to your working folder and makes it writable. SVN supports files being checked out by multiple users. When checked in, the modified files will be merged into one file without losing any changes.

A working folder is the specified corresponding folders on a user's local computer used to store files when working with SVN projects. A user can make changes to files in the working folder and then check the modified files back into the SVN database for version tracking. Check-in command enables the user to store the configuration item in the SVN database. After you have finished the changes, you need to Check In the updated file back into SVN. When we say SVN keeps all your previous changes, we are not saying that SVN keeps every change you make, we are saying that SVN keeps all the versions you checked in. If a file is not checked in, there is no way that SVN knows that the file has changed and it should keep it.

A history is maintained whenever configuration items are checked out and checked in. In history we can determine who retrieved the configuration item and when or what modifications have been performed. This way configuration management records are kept about the configuration item.

3.4.1.3 SP 1.3: Create or Release Baselines

A baseline is a defined set of work products, components, specifications, etc that have been formally reviewed and agreed on, that thereafter serve as a common baseline from which all future development

and delivery takes place. In terms of software development, this could be the release of a version of the software, and a record of all the associated documents which go with it, from requirements specifications, architecture and design documents, source code, release notes, test plans, user manuals and so on.

Baselines for components should be established at the 6-month baseline meetings. The baseline corresponds to the final version of a major or minor release of a component, so all associated documents of this version of the component can be assigned that same version number. This version number then provides a common identifier for everyone involved to know and understand what is included in that baseline.

3.4.2 Track and Control Changes

Having established a baseline, the next thing which the configuration management system and the configuration items within it will undergo are changes. This goal looks at managing these change requests, keeping track of them and controlling the change requests which go through.

3.4.2.1 SP 2.1: Track Change Requests

Depending on the software development approach adopted, the source of the change requests may vary significantly. Change requests may come from quality assurance to fix defects in the products. All these change requests need to be recorded and analysed for their impact on the system.

A good way to automate the tracking of these requests, is using a tracking system like JIRA[29] or Bugzilla[30]. Change requests can be created in the tracking system and then be acknowledged or declined by the project leader. Both systems are not only able to track change requests, but can also be used to track bugs, issues and features. There are plugins available so commits made to the SVN repository are reflected in the related JIRA item just as long as the JIRA issue/change request number is specified in the commit log comment.

[29] http://www.atlassian.com/software/jira/overview
[30] http://www.bugzilla.org/

3.4.2.2 SP 2.2: Control Configuration Items
In this practice, we ensure that changes in the configuration items are controlled. Only approved changes are made to the configuration items, a new configuration is created if necessary and the baseline is updated when necessary as well. We ensure that all these changes are documented, and archives are created for the different baselines, allowing the team to revert back to a 'last known working version' in case of a major problem.

3.4.3 Establish Integrity
Processes here allow the teams to establish the integrity of the baselines after changes are made. Integrity is established based upon the initial criteria used to identify what a baseline is, and carried out after ensuring proper tracking and controlling of changes takes place.

3.4.3.1 SP 3.1: Establish Configuration Management Records
As explained by the name, these records keep track of the changes made to the configuration items, a revision history of the configuration items, status of configuration items and the main differences between the baselines. SVN records configuration management actions in sufficient detail so the content and status of each configuration item is known and previous versions can be recovered.

3.4.3.2 SP 3.2: Perform Configuration Audits
Configuration audits ensure that the new baselines and documents of the system confirm to standards and criteria established at the start. These results should be recorded. Periodical audits of the configuration management system, baselines and documents can also ensure that the system is being correctly utilised. In TRANSFoRm the SQA Manager will perform these audits.

Configuration audits consist of cross checking the CI description records with the current version of the CI as held in its development folder. These audits are normally carried out ahead of each project review, although they can be requested at any time. Configuration audits should establish that: the current physical representation of each CI matches its current specification, that there is consistency in design between each CI and its parent, if there is a parent. Finally it also establishes that the documentation is up to date and all relevant standards are being adhered to.

3.5 Test Planning
Although the release process described in section 2 is quite rigid together with the mapping described in section 3.1, the flexibility of it all lies within the planning. From the release process it is known that roundtrips to DEV-stage might be necessary in SI (and/or PAC), so this can be taken up into the planning, making the initial DEV-stage a bit more incremental and not just about fixing the SI-issues (or PAC-issues).

For example, in case that at the start of a new release cycle (for version 0.3) of component A the component owner plans to have a certain set of requirements implemented. The component owner knows that it is very likely that after the first RC-build (the first transition into SI), the component needs to be further improved to succeed in SI. Therefore a second RC-build will be necessary. The owner can

therefore split the set of requirements into 2 subsets and plan that version 0.3-RC1 will have implemented the first subset of requirements. The owner knows that RC1 will not pass SI-stage as not all requirements are met[31], but chances are that the deployment failed anyway. The failure of any tests or deployment can then become an additional requirement to the second subset of planned requirement implementations, which will be subject of RC2. After RC2, there still might be issues, and therefore the owner foresaw an RC3, which is solely intended to fix purely SI-related issues. Once RC3 is ready, it will probably pass SI. This is a useful way to already incorporate the SI stage into the component planning. The same can be done for PAC. To continue the example the component owner of component A might foresee additional builds, one RC and one FINAL. The purpose of the extra RC build (RC4) is purely to cope with the PAC-issues that came out of RC3 (which passed SI, and therefore went into PAC). RC4 can still have some (minor) PAC-issues. Component owner and project management should decide if the issues are important or not, so they can still be fixed when doing the FINAL build (which needs to go through SI and PAC, just to be sure nothing has broken, before it goes to PROD). If at some point a requirement is dropped, planning can take it up in a future release or drop it entirely in case the requirements have considerably changed.

There are three variables in project planning: time, scope and resources. Typical project planning involves pushing deadlines behind in order to include all foreseen functionality. The 3-monthly milestones are an additional aid with regard to the planning. The intent of these intermediate milestones is to make it clear to the component owners which adjustments need to be made towards the planned functionality of the upcoming release. The decision can be made to reduce the scope by mimicking certain functionality (see 2.5), because it is expected to take too much time to include this functionality for the upcoming release (timeboxing).

It has already been noted that it is very unlikely that every new release of TRANSFoRm will need a new version of each of its components. It is understandable that at the beginning of the project only a couple of components will get the main development focus as they deliver basic functionalities for the other components (components that can't work without that functionality being available). Or some components are included for now, but are still stubs. Some components might get regular releases in the first years, but will then reach a final state in the project as it has all the functionality that was foreseen. So, it is possible that component A already has version 2.3 during year 2, while component B only has a version 0.4.

3.6 Testing

Given its importance, testing has already been mentioned several times before. However, testing in highly complex projects like TRANSFoRm is a multi-level aspect that needs a proper strategy. In fact, the integration plan should be considered as part of a complete testing strategy, which by itself is a part of the SQA framework. Although a complete testing strategy is outside the scope of this document, a

[31] Tests should be included in the build that test the second (not yet implemented) set of requirements. These will be explicitly made to fail if the necessary constructs for testing the requirements are not yet available (have not yet implemented).

summary of testing activities in the context of this proposed integration plan that are relevant for TRANSFoRm to guarantee quality in development, is provided.

The described release process requires different kinds of testing during the different stages. The V-Model of testing identifies four software testing phases, each with a certain type of test associated with it.

Stage	Test Type
DEV	Unit Testing
SI	Integration Testing
PAC	System Testing
	Product Acceptance Testing
Regression Testing applies to all stages	

Table 5: Different test types

The composition of the different tests will be the responsibility of the testing team(s) in co-operation with the component owner of the component that needs to be tested. Tests can already be created at the beginning of the DEV-stage as the requirements are known at that point and the result of what will be built as well. Tests are specified in test scripts and test results are documented. The course of a test is described in a test plan.

Some components will need test data to perform solid testing. This is important for ensuring reproducibility of test results. Test data will be available in central repositories, specified in the test description. Also descriptions on the format of the data and how to access the data, need to be available.

3.6.1 Unit Testing

Unit tests can be created at the beginning of the DEV-stage as the requirements are known at that point and the result of what will be built as well. Unit testing is done during the DEV-stage, ideally by the developers themselves as part of a test-driven or test-evidence based development strategy. The purpose of unit tests is to test the implemented code on the smallest unit of software design (i.e., in isolation). It is very costly (time, effort) to fix bugs if they are discovered further from the point from which they are introduced. Therefore functionalities that are never accessed directly from outside a component still need to be tested for correct execution. This is typically done via unit tests. If a change is made to existing code, the unit tests should quickly indicate if something has been broken or not. This means that unit tests will need to be updated as well as the functionality of the unit has changed, and the test is not testing the correct functionality anymore. Unit tests are about verifying if the result of an implemented functionality meets the requirements. Unit testing is specific for one component and should not involve any dependencies (unless in the form of drivers or stubs). This is the reason why it is part of the DEV-stage as the developers are responsible for one component.

3.6.2 Integration Testing

Once out of DEV, the focus of testing shifts to the system level, no longer touching on the isolated behaviour of the individual component. SI delivers a stable system in the sense that all the components

can co-exist and co-operate. Once that has been verified, the SI stage is considered finished (unless there will be a roundtrip to DEV in PAC). It has been mentioned that the actual deployment of the components is the first goal of a test. A successful deployment is a verification of the system on itself. More is needed though. Components should be checked individually if they work (for example, can they access their database, can they access that other component they need to work ...). If a new component is introduced, do the others still work? Or let's say that a component C was excluded from the current release cycle (meaning that no new version will be built), but it still needs to be in the final release . At the same time a new version of components A and B is to be included in the current release. Then SI is the place to see if component C still works as expected, with the updated versions of A and B[32]. These tests of component C are called regression tests. They test previously implemented requirements of components. These requirements have not changed, so the implemented functionalities should not have changed as well. The set of regression tests varies from release to release. The emphasis of the tests in SI is on verification, not on the functionality itself. A functional test could be used for this, but the goal is to see if it works, not if the results are correct (semantically). There are several approaches to perform integration testing:

1. **Bottom-Up Testing** is an approach to integrated testing where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested. All the bottom or low-level modules, procedures or functions are integrated and then tested. After the integration testing of lower level integrated modules, the next level of modules will be formed and can be used for integration testing. This approach is helpful only when all or most of the modules of the same development level are ready. This method also helps to determine the levels of software developed and makes it easier to report testing progress in the form of a percentage.
2. **Top-Down Testing** is an approach to integrated testing where the top integrated modules are tested and the branch of the module is tested step by step until the end of the related module.
3. **Sandwich Testing** is an approach to combine top down testing with bottom up testing.
4. **Big Bang Testing** is an approach where all or most of the units are combined together and tested at one go. This approach is taken when the testing team receives the entire software in a bundle.

From above mentioned approaches only Top-Down & Bottom-Up are used mostly. So the main advantage of bottom-up approach is that bugs are more easily found as compared with top-down approach, it is easier to find the missing branch link and show the whole software process to user early.

3.6.3 System Testing & Acceptance Testing

The actual functional testing should be done in the PAC-stage. Here all the newly implemented requirements are tested, but also the old requirements that do not conflict with the newly implemented ones should be tested again. This may include regression testing, but here the emphasis should be on the functionality, meaning that if button X of component A is pressed, does it indeed return the expected result, which is constructed from results of certain functionalities of component D and E?

[32] Sometimes the failure of component C can be the expected result of the new versions of A and B, because C is planned to be working again in the following release cycle, when it gets a new version. It all depends on how it is planned.

3.6.4 Using Code Coverage Tools

Code coverage is an important type of test effectiveness measurement. Code coverage is a way of determining which code statements or paths have been exercised during testing. With respect to testing, coverage analysis helps in identifying areas of code not exercised by a set of test cases. Alternatively, coverage analysis can also help in identifying redundant test cases that do not increase coverage. After performing coverage analysis, if certain code paths or statements were found to be not covered by the tests, the questions to ask are whether the code path should be covered and why the tests missed those paths. A risk-based approach should be employed to decide whether additional tests are required. Covering all the code paths or statements does not guarantee that the software does not have faults; however, the missed code paths or statements should definitely be inspected.

Testing is mostly considered to be both never finished and dependent upon the acceptable likelihood of failure (it is usually not cost-effective to test everything).

3.6.5 Test Teams and Cross-Testing

Because testing is a resource hungry process, there are certain limits to be taken into account and proper planning is needed, as otherwise testing can become a nightmare (time, money and people). The larger a system gets, the more difficult it gets to fully test everything. Testing should be limited in time and to be sure it is done properly, it should be performed by a dedicated team (or several teams). A proper test planning should be implemented.

A good strategy is to have a rotation system set up with regards to the testing teams, so that different teams are responsible for the PAC stage testing in different release cycles. Or the tests for the PAC-stage should be distributed to all the teams, so that all the teams are testing something. When doing the latter, a certain level of impartiality needs to be taken into account as a developer that is testing his/her own component could be considered to be biased. Therefore "cross-testing" (or X-testing) will be performed, which means that teams are testing anything but their own work/component.

For SI, it is possible to employ a similar strategy as the one used in PAC, i.e., having one team (or several teams) dedicated to the SI work (deployment, testing) or get all the teams involved. For the latter, again being unbiased is key, as a team having to deploy or test their own component is not the best idea in the long run.

3.6.6 Test Automation

Automatic testing is a good way to help with limited project resources, but this mostly requires a certain expertise, sometimes not readily available. Automated regression tests are a great help, so sometimes it justifies the use of resources to come up with good automatic (functional) tests. Automated testing is not only about saving time, it also ensures the reproducibility of test results. Sometimes specific testing tools can be used to help achieve this automation.

A discussion of test automation is outside the scope of this document. However, unit testing is a good example of test that can be run automatically when doing a build (e.g., JUnit[33]). For integration and

[33] http://www.junit.org/

functional testing, tools like SOAPUI[34] (for testing web services) can be used to create test suites that can be automated (manual intervention may be required, although they can usually be fully automated as well). Even for acceptance testing, automation toolkits exist (e.g., Fitnesse[35], Concordion[36] for functional acceptance testing, Selenium[37], Canoo[38] for automated web testing).

3.7 Guidelines for Component Delivery

As described in section 3.1, the requirements that should be met at the end of a lifecycle are identified before the start of the lifecycle (the 6-month period) by the project manager and component owners. Also the required deliverables are identified. Further design is done by the component teams, creating the test plans to be carried out at the same time. The WT3.5 SQA plan contains an extensive description of the deliverables and checklists that are necessary to ensure quality. These deliverables and checklists can easily be mapped to the suggested software release cycle described in this document. At the end of every stage in the release cycle certain deliverables need to be provided, or when omitted an explanation for the omission should be included. Checklists are provided for each stage. When the component owner has answered all questions from these checklists for the corresponding stage, the answers will be checked by the project manager. The deliverables will be inspected by the SQA Manager. If the SQA Manager decides the deliverables meet the required quality level, and the project manager is satisfied with the checklist answers, that component can move to the next stage.

Every stage has its documents to be provided and its own checklist that must be answered. Below the required deliverables and associated checklist are listed for each stage.

3.7.1 DEV-stage

The release cycle starts in the development-stage where developers start to work on their implementation work. Each of the deliverables listed in Table 6 should be produced during the DEV-stage. If they are not included, the component owner must give a short explanation as to why this is the case.

Deliverable	Legislation
Product component implementation in code	
Documented Code	21CFR65
Unit test coverage and results	21CFR820.22
End-User Training Manuals	21CFR820.40,181
Installation Manuals	21CFR820.40,181
Operation Manuals	21CFR820.40,181
Maintenance Manuals	21CFR820.40,181

Table 6: Deliverables in DEV-stage

[34] http://www.soapui.org/
[35] http://fitnesse.org/
[36] http://www.concordion.org/
[37] http://seleniumhq.org/
[38] http://webtest.canoo.com/webtest/manual/WebTestHome.html

At a certain point the component owner deems a build ready to go to SI. He concludes the goals of the current development cycle (implemented features addressing certain requirements, bug fixes) have been met, and the code has been successfully unit tested by the developers. Then the component owner should answer each of these questions on the checklist below (Table 7). If the answer to any question is 'No', the component owner must give an explanation as to why.

Checklist
Are design patterns being utilised?
Is the code being documented?
Are unit tests written for all the individual units of code?
Is the code being reviewed by peers?
Is the implementation revised based on unit tests and code reviews?
Are there documentation standards identified and utilised?
Are the documents being reviewed by stakeholders?
Is the documentation being kept updated to reflect changes in the product?

Table 7: Checklist for DEV

The produced documents are submitted to the SQA Manager. When the SQA Manager decides that the required quality level is met and the project manager is satisfied with the checklist answers, the component is ready to be integrated with the other components of the system. Therefore an release candidate is built (see 2.1) for the SI-stage, and the integration manager is notified that the component will move on to the SI-stage.

3.7.2 SI-stage

In the SI-stage the various components are deployed and coupled together to form more complex components of the TRANSFoRm architecture. This deployment should be done by an independent party (another team that was not involved in the development of this component). To ensure the required quality level is obtained, the following deliverables should be presented at the end of the SI-stage:

Deliverable	Legislation
Product Integration Sequence	21CFR820.30
Rationale for Selecting or Rejecting Integration Sequences	21CFR820.30,80
Verified environment for product integration	21CFR820.70
Support Documentation for the Product Integration Environment	21CFR820.80,181
Product Integration Procedures & Criteria	21CFR820.70
Interface Description Document, with categories of interfaces, list of interfaces per category and the mapping interfaces to product components and integration environment.	21CFR820.65,181
Table of relationships between product components and external environment	21CFR820.65,181
Table of relationships between different product components	21CFR820.65,181
Agreed-to interfaces defined for each pair of product components	21CFR820.70,80,181
APIs	21CFR820.80
Acceptance documents for the received product components	21CFR820.80
Release Notes – Consisting of exception reports/known issues/waivers	21CFR820.80,181
Interface evaluation reports	21CFR820.80,100,191
Project integration summary reports	21CFR820.181

Table 8: Deliverables in SI-stage concerning product integration

Teams that have components which interact with other components will be responsible for defining and communicating the relevant interfaces to the components. The component owners will ensure that product integration is carried out in a structured and manner and report the integration status of their component to the integration manager. When technical problems occur, the project manager can help to find a solution. The component owner will use the checklist in Table 9 to ensure that activities have been covered.

Checklist
Are the components to be integrated identified?
Are the tests to be performed during integration identified?
Are there alternative product integration sequences identified?
Is the product integration sequence revised since the previous iteration?
Are the decisions being made and their rationale documented?
Are the requirements for product integration environment identified?
Are there verification criteria and procedures for the product integration environment?
Are product integration procedures established and maintained?
Are product component integration and evaluation criteria established and maintained?
Are validation criteria for delivery of the integrated product established and maintained?
Is the interface data complete and ensures coverage of all interfaces?
Are the interfaces updates to remain compatible with any changes, non-compliances or conflicts?
Is the evaluation of assembled product components documented?

Table 9: Checklist for SI-stage concerning product integration

The system still needs extra verification. Therefore the integration tests developed during the detailed and structural design phase (see Figure 1: The V-Model of Software Testing) are being carried out by the assigned independent team (the other component's team). During this process documents from Table 10 are being produced:

Deliverable	Legislation
List of work products selected for verification	21CFR820.20(d)
Verification methods for each selected work product	21CFR820.20(d)
Establish verification environment	21CFR820.20(d)
Verification Procedures	21CFR820.5, 20
Verification criteria	21CFR820.20
No Deliverable Legislation Peer review checklist & selected work products	21CFR820.22
Peer review criteria	21CFR820.22
Peer review training material	21CFR820.186
Peer review results, issues and data	21CFR820.200
Peer review action items	21CFR820.75,100
No. Deliverables Legislation Verification Results	21CFR820.22
Verification Reports	21CFR820.22
Analysis report	21CFR820.22
Change requests for verification methods, criteria & environment	21CFR820.70

Table 10: Deliverables in SI-stage concerning verification

Once the verification is finished, the component owner must answer all questions from the checklist presented in Table 11:

Checklist
Is there a traceability matrix to identify requirements for work products?
Are the verification methods utilised appropriate for the product?
Are the verification environment, equipment and tools adequate for testing and reflective of real usage of the products?
Are there verification criteria created to ensure that all requirements are tested in different ways?
Are there expected results recorded for all the different tests and criteria?
Are there requirements for data collection at the time of the peer review?
Are there entry and exit criteria for a peer review?
Are there criteria to determine is a subsequent peer review needs to be done?
Are there checklists for what needs to be reviewed during a peer review?
Are there defects being identified based on checklists and documents?
Is the peer review data being recorded?
Are there actions identified and communicated appropriately?
Is the review data being analysed?
Are there medium-term and long-term actions identified to provide continuous improvement?
Is verification of products taking place against the product requirements?
Are verification results being recorded?
Are actual results being compared to expected results to determine if criteria are being met?
Are action items being generated as required from verification procedures?
Is information being provided on the defects were detected, logged and possible solutions on how it could be resolved?

Table 11: Checklist for SI-stage concerning verification

If any issues occur, additional work from the developers is needed, which warrants the return to the DEV-stage. The component owner will create a change request and the component will return to the DEV-stage.

When no issues have occurred, all documents will be inspected by the SQA Manager. When he decides the delivered quality is sufficient and the project manager is satisfied with the checklist answers, the component can move on to the next stage, i.e., PAC-stage.

The integration manager is at all times notified of the new status of the component.

3.7.3 PAC-stage

PAC is the stage where the system is validated (is the product/system doing what was intended?). The added and/or improved functionality is tested against the requirements. Additionally, PAC is also the stage where acceptance testing is executed. These tests will be executed by an assigned test team under supervision of the SQA Manager.

To ensure the validation is performed in such a way that the required quality standards are met, the documents listed in Table 12 should be presented at the end of the stage.

Deliverable	Legislation
List of work products selected for validation	21CFR820.20, DMR
Validation methods for each product or component	21CFR820.20, DMR
Requirements for performing validation	21CFR820.20, DMR
Validation constraints on validation for each product	21CFR820.20, DMR
Validation environment for validating selected products and components	21CFR820.20, DMR
Validation procedures	21CFR820.20, DMR
Validation criteria	21CFR820.20, DMR
Validation Reports	21CFR820.80,90
Validation Results	21CFR820.80,90
Cross-reference Matrix	21CFR820.65
As-Run procedure logs	21CFR820.65
Operational demos	21CFR820 DHR
Validation deficiency reports	21CFR820.100
Validation issues	21CFR820.90
Procedure Change requests.	21CFR820.100

Table 12: Deliverables in PAC-stage

All questions in the checklist shown in Table 13 should be answered at the end of the stage. If the answer to any question is 'No', the component owner must give a short explanation as to why.

Checklist
Are key phases of validation for each product or component identified?
Are the validation requirements, constraints and methods reviewed by stakeholders?
Are third party software being validated?
Are validation environment requirements identified?
Are tools and test equipment identified?
Are validation resources which can be reused identified?
Is there a detailed availability plan for resources, if resources are not dedicated?
Are the validation environment, procedures, operational environments and criteria for validation documented?
Are subsequent changes in requirements and design being reviewed for
possible impact on validation processes?
Are products and components performing according to validation criteria?
Are the results being recorded?
Are the results analysis and issues identified being documented?

Table 13: Checklist for PAC-stage

When all tests have been executed successfully and the SQA Manager decides the delivered quality level is sufficient and the project manager is satisfied with the checklist answers, the component is considered to be ready for PROD.

If this is not the case, the component owner will create a change request containing the changes to be made to resolve the issues. The component will then be sent back to DEV-stage.

The integration manager will be notified of the new status of the component.

3.8 Integration Environments

TRANSFoRm will use several integration environments. These are listed below in Table 14:

System	Responsible organization
DEV	Local at developer
SI	KCL
PAC	KCL
PROD	UDUS
SVN	KCL

Table 14: Integration environments and responsible organizations

Runtime Environments

DEV, SI, PAC and PROD are the actual "runtime" systems where TRANSFoRm should run on during the different stages of the release cycle. The SI- and PAC-environments will be run by KCL. A validated version of PROD covering clinical trial components will be implemented by the University of Duesseldorf (UDUS) representing ECRIN[39] after the development cycle has been concluded (available at the beginning/mid of 3rd year), while every developer will have his/her own DEV-environment.

The production environment (PROD) is the final endpoint in the release cycle. This is also the point where the system validation of computerized systems used in clinical trials is conducted. Good Clinical Practice (GCP) requires a system validation before a system is used for a clinical trial. Assuring fitness for purpose is a key goal of any quality standards, but validation must always be a deliberate, thoughtful and proportionate process, based upon a risk assessment, rather than a blind and blanket application of rules. It is recognized that academic clinical trials units do not, in general, have the resources to conduct a full validation for every component – i.e., including detailed requirement specifications, validation and test plans, and then carrying out and documenting the test results (and then repeating the exercise after each major revision). Another difficulty with validation is that there is no simple way to know how much is required. Generally, one generates tests against requirements, but even if those requirements are fully comprehensive it is impossible to test all possible input scenarios prior to deployment. Therefore, GCP-validation at ECRIN will be done with complete clinical trials data (ghost study) to include as many "real trial" scenarios as possible.

In clinical trials no isolated application is used, but a computer-controlled system consisting of a set of hardware and software solutions and controlled processes. The computer-controlled area at the clinical trial centre consists of hardware, software and network components and the people who use them. It will include the TRANSFoRm tools (Figure 5). All components that may have an influence on patient security and data quality fall under GCP regulations. Thus, GCP system validation will evaluate

[39] European Clinical Research Infrastructure Network (http://www.ecrin.org/)

TRANSFoRm tools as part of the entire computer controlled environment, including the hardware/software but also instructions (e.g., SOPs[40]), instruments and people (e.g., competence).

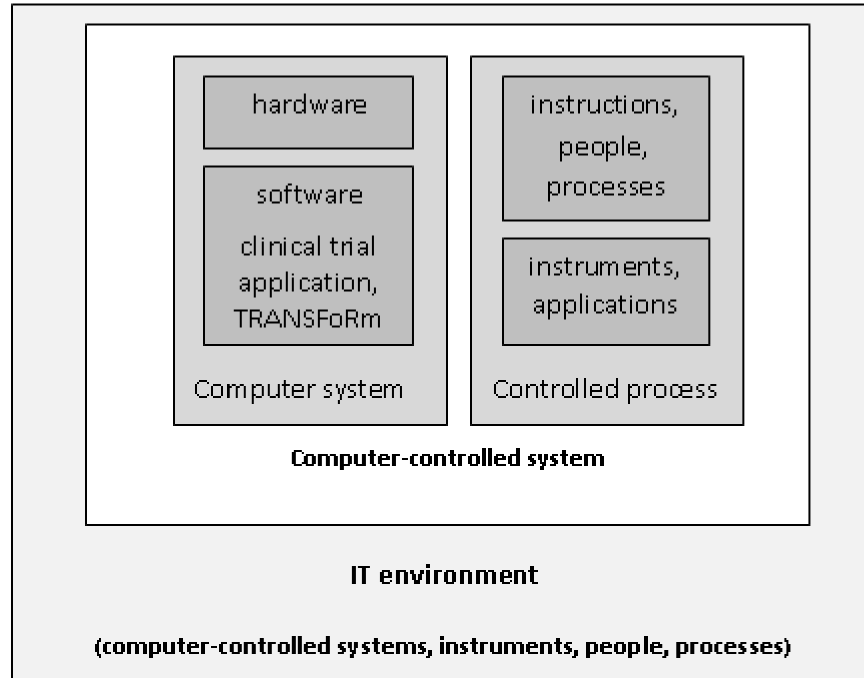

Figure 5: Structure of the components and parts of the IT environment at a clinical trials centre seen from the GCP system validation point of view

Support Systems

Next to the runtime systems, the TRANSFoRm-project will also need support systems to support management of the project, both from a developers perspective as from a project management perspective.

It has already been established that a **source code versioning system** is necessary to keep track of all the versions of the source code, produced by the developers in the project. This is done via an SVN system that is hosted by KCL.

From the software integration survey that preceded this software integration plan (see 4.2), it was clear that the teams involved use either the Maven[41] build system (with dependency management) or a purely Ant-based[42] build system (without any dependency management, except manual). Regardless of the tool used during the DEV-stage, the transition to SI-stage should be done by building an RC version of the component. Preferably, the build system/script used for this would prevent inclusion of any unstable development code in the RC. The use of the SNAPSHOT-suffix in the version numbers of builds in DEV can be beneficial. Maven knows how to leverage this as it will never include any snapshot

[40] Standard Operating Procedure
[41] http://maven.apache.org/
[42] http://ant.apache.org/

versions of any component (when resolving dependencies) if not explicitly indicated[43]. Maven and Ant are not mutually exclusive, so the teams not using Maven could still incorporate the process of not including SNAPSHOTS, via Maven[44].

The end result of a build should be an **artefact** of some sort (jar, war, exe, zip) that has a version number. These artefacts should be kept in **central repositories**. These repositories can be simple ftp-sites (File Transfer Protocol) or dedicated dependency management systems. Build systems like Maven use repositories not only to retrieve artefacts (and their dependencies), but also to upload the artefacts that are the result of a successful build. Ivy[45],another dependency management system that can be used in Ant-based build environments, also uses repositories. Therefore the combination of Ant as a build system and Ivy as a dependency manager can be an equivalent alternative to Maven, as long as the notion of SNAPSHOT, RC and FINAL releases can be leveraged as well.

As previously discussed, Continuous Integration (CI) can improve the execution of certain release cycle tasks when doing builds. It is recommended for TRANSFoRm. Suitable example systems include Hudson[46], Jenkins[47] or CruiseControl[48]. These systems allow for triggering automatic builds. For example, every time code is checked in into the SVN, the corresponding component gets rebuilt. In a continuous integration system several different builds can be defined for a component. A good idea is to have one for DEV (creating only SNAPSHOT –builds), one for SI and PAC (creating only RC-builds, or something similar) and one for PROD (creating only FINAL-builds or something similar)[49]. The DEV-builds could be triggered by a check-in of new or updated code into SVN, which creates regular SNAPSHOT-builds when all stages of the build have succeeded (for example, did the unit tests succeed, were there no major errors made regarding code convention, if code convention was checked ...). These SNAPSHOT-builds will create a tag in SVN (creating some kind of "snapshot" of the state of the code at the time of the build). At some point the component owner will deem a build ready to go to SI. He can then request an RC-build to the continuous integration system. The RC-build should differ to the SNAPSHOT-build in that it produces an RC-build, it actively checks if there are no dependencies anymore on SNAPSHOT-versions of subcomponents (mostly done by the build system not the continuous integration system, as it is actually the build system that does the build), and it creates an RC-tag in SVN. It increments the RC version number, but this could be done manually as well before the build is requested. Automatic checking of the version number is possible for example with Maven, but a custom plug-in could be necessary for this. Automation of the version number checking could be useful to prevent the same RC version to be created twice in a row. Additionally it could also execute a deployment script to automatically deploy the newly created RC-build.

---

[43] http://www.sonatype.com/books/mvnref-book/reference/pom-relationships-sect-pom-syntax.html. Although Maven can deal with the concept of a SNAPSHOT internally, to really help with the release process a plugin is necessary, like the release-plug-in http://maven.apache.org/plugins/maven-release-plugin/

[44] https://docs.sonatype.org/display/Repository/Deploy+Snapshots+and+Stage+Releases+with+Ant

[45] http://ant.apache.org/ivy/

[46] http://hudson-ci.org/

[47] http://jenkins-ci.org/

[48] http://cruisecontrol.sourceforge.net/

[49] The suffixes SNAPSHOT, RC and FINAL are not always used when it comes to build systems. Mostly there will be a corresponding concept, with a different name (like STAGE for RC or RELEASE for FINAL).

4 Software Components

4.1 List of Software Components

The software components identified by the architecture team within TRANSFoRm are listed in Table 16. The responsible team leaders and partners for each component are specified, as well as the due date for the component. Also intermediary deliverables are listed. The used acronyms are listed in Table 15.

Partner acronym	Full name
TCD	Trinity College Dublin
UB	University of Birmingham
RCSI	Royal College of Surgeons Ireland
IC	Imperial College
WRUT	Wroclaw University of Technology
NIVEL	NIVEL
GPRD	GPRD
UNIVDUN	University of Dundee
KCL	Kings College London
MIPC	Mediterranean Institute of Primary Care
KI	Karolinska Institutet
UA	University of Antwerp
INSERM/UR1	Institut National de la Sante et de la Recherche Médicale and University of Rennes
CSIOZ	Centrum Systemów Informacyjnych Ochrony Zdrowia
UDUS	University of Duesseldorf

Table 15: Partner acronyms

Work task	Component	Component owner	Responsible partner(s)	Intermediary deliverables	Due date
3.3	Security framework	Siobhán Clarke	TCD, UB	M18, M36	M48
4.3	Ontology service for diagnostic evidence	Derek Corrigan	RCSI, TCD		M30
4.4	Web-based evidence service	Derek Corrigan	RCSI, TCD	M36	M48
4.5	Data mining tools	Vasa Curcin	IC, RCSI, WRUT		M36
5.1	Data quality tool	Robert Verheij	NIVEL, GPRD, UNIVDUN	M24	M30
5.2b	Triggering tool	Theo Arvanitis	UB, KCL, MIPC		M42
5.3	Query and data extraction workbench with diabetes use case tools	Theo Arvanitis	UB, KI, UA	M24	M36
5.4	Provenance service	Vasa Curcin	IC		M24
7.1	Model-based semantic mediation service	Theo Arvanitis	UB, TCD		M30
7.2	Terminology service	Theo Arvanitis	UB,MIPC,INSERM/UR1		M18
7.3	Metadata repository	Theo Arvanitis	UB		M30
7.4	Infrastructure for deployment of eCRFs into EHRs	Theo Arvanitis	UB, KCL, KI, UA	M36	M42
7.5	Infrastructure for extraction/linkage aka Middleware	Siobhán Clarke	TCD, UB	M24, M36	M48
7.6	Middleware for mobile e-Health and CPR derivation	Przemysław Kazienko	CSIOZ, WRUT		M48

Table 16: TRANSFoRm software components

4.2 Component Descriptions

To have a good understanding of what technologies are already used by TRANSFoRm partners, Custodix sent a survey to all component owners with the intent to identify not only the different components within TRANSFoRm, but also to have a better understanding of the internal way of working of the different teams developing the components. The following sections provide an overview of the provided feedback.

4.2.1 WT 3.3: Security Framework

This component is an extensible technical security solution for data transfer, authentication and authorization for accessing, transferring, and querying clinical health records. It is a dynamic extensible usable security framework to provide interoperable integrated technical security solution covering TRANSFoRm distributed middleware, distributed services/agents and software and user services.

The security-related part will be based on Shibboleth[50], XML encryption and signing libraries[51]. It will also use Apache Tomcat[52].

Dependencies:

This component will be delivered as part of the WT 7.5: Infrastructure for Extraction/Linkage of Data.

4.2.2 WT 4.3: Ontology Service for Diagnostic Evidence

This component is responsible for providing a repository of clinical evidence in a computable format that supports defined clinical scenarios to be used to underpin the provision of a diagnostic decision support service based on evidence based clinical knowledge.

The main application programming language is Java, and Spring and Hibernate frameworks will be used. For the database MySQL[53] is being used. For the development of the ontology of clinical evidence, Protégé[54] will be used as tool.

The hosting of ontology is being done using Sesame triple store (using TomCat and MySQL).

Dependencies:

This component does not depend on any other software component.

4.2.3 WT 4.4: Web-based Evidence Service

This component is an agent-based or web-based evidence service to support clinical decision making in primary care, on the basis of the rule repository produced in WT 4.3: Ontology Service for Diagnostic Evidence. The system will be able to find in the back-end relevant Clinical Prediction Rules, trigger their application, and give decision and/or safety support in an integrated clinical pathway.

---

50 http://shibboleth.internet2.edu/
51 http://santuario.apache.org/
52 http://tomcat.apache.org/
53 http://www.mysql.com/
54 http://protege.stanford.edu/

Ontology is being represented using web ontology language and resource description framework (OWL/RDF), while web services will use either Java web services or REST based services.

The interfaces to this component will be provided by means of web services. There will be two main web service interfaces: a query interface to get clinical recommendations from the web service (used by the WP5 CDSS interface) and an update interface that will be used to update the underlying clinical knowledge from research data (through WT 4.5: Data Mining Tools). The interfaces will be implemented using standard web services technologies (such as Java Web services or REST[55] based services).

Dependencies:

This component depends on WT 4.3: Ontology Service for Diagnostic Evidence, which forms the rule repository used by this component, and will use WT 5.4: Provenance Service.

4.2.4 WT 4.5: Data Mining Tools

This component consists of collaborative analytical tools for validation and improvement of generated rules and managing their deployment into the rule repository. These will include visual model evaluation, together with drill-down methods for investigating the properties of the evidence sets used to construct rules. The component will expand the ability of the system to collect and analyse the data on symptoms and signs in association with the reason for encounter and diagnostic outcome data to validate existing, and develop new clinical prediction rules (or likelihood ratios) for diagnosis by means of advanced machine learning methods like ensemble, hybrid and collective classification as well as structure prediction models.

The current plan is to use an open-source workflow-based data mining environment such as Konstanz Information Miner (KNIME)[56], for portability, flexibility and good provenance support. SVN is used as source code versioning system, while issues are tracked with Bugzilla. Builds are managed with Maven.

Dependencies:

This component will depend on WT 7.6 Middleware for Mobile e-Health and Clinical Prediction Rule Derivation for the use of anonymisation or pseudonymisation tools to protect patient confidentiality, and will use WT 5.4: Provenance Service.

4.2.5 WT 5.1: Data Quality Tool

The Data Quality Tool is the component within TRANSFoRm that will measure data quality of different care data repositories and other databases. On the code level, this component does not depend on any other component in this project, nor do other components depend on this component, however the output data from the tool will be used by the Query Formulation Tool. It will be a web-tool, and its functionality is described as follows:

[55] REpresentational State Transfer
[56] http://www.knime.org/

Dependent on the type of research question, the researcher will be able to decide what databases are suitable to participate in a clinical trial or a cohort study. The data quality tool that runs on the database of the research network extracted from the practices can provide the information needed for the researcher to make this decision.

A SQL Server database is used for calculating the quality measures developed for each use-case, while the web tool will be using a separate MySQL database to store all the retrieved values for the quality indicators. These values can be updated by the database owners, most likely using Microsoft xls (Excel) or csv (Comma Separated Value) file formats. The format for saving user's queries will probably be xml (eXtensible Markup Language) or just plain csv files. The query results will be downloadable as csv and/or MS xls files.

Special care will need to be taken with the process of updating the database with the new data, since rolling back to a previous version might be very time-consuming. Updating active databases is an activity that might possibly need its own release process.

Dependencies:

The component uses WT 7.5: Infrastructure for Extraction/Linkage of Data for linking to data sources, and WT 5.4: Provenance Service to track the provenance of queries made.

4.2.6 WT 5.2b: DSS Plugin for EHR Tools (Triggering Tool)

The component is an adaptive interface in the EHR (Electronic Health Record) tools to support alerting and suggesting by the decision support system.

Project management, as well as issue-tracking is managed by Redmine[57]. All code is documented with JavaDoc-style comments for use with Doxygen[58] or equivalent. SVN is used as source control system, while the building is done with Apache Ant, while open for alternatives.

Dependencies:

The component links to WT 4.4: Web-based Evidence Service.

4.2.7 WT 5.3 Query Tool and Data Extraction Workbench

This component contains the interfaces necessary to create, manage, store and deploy queries of clinical data to identify subjects for clinical studies and to analyse the numbers of matching subjects in cohort studies. It consists of two parts:

a) Query Formulation Tool for the identification of research subjects based on existing care data and the analysis of numbers of matching subjects.
b) A workbench for managing data extraction and linkage for epidemiological studies.

[57] http://www.redmine.org
[58] http://www.doxygen.org

This is based on the use of a terminology service (WT 7.2: Terminology Service) with a research data model (WT6.4) and a workbench to allow the binding of terminology to the model. The challenge is to enable users to formulate analysis rules, procedures and queries of clinical data for research and to understand the results, whether as text, tables, graphs or complex visualizations.

Project management, as well as issue-tracking is managed by Redmine. All code is documented with JavaDoc-style comments for use with Doxygen or equivalent. SVN is used as source control system, while the building is done with Apache Ant, while open for alternatives.

Dependencies:

This component depends on WT 7.2: Terminology Service, which provides a terminology service, and uses the WT 5.4: Provenance Service.

4.2.8 WT 5.4: Provenance Service

This component manages the provenance of data obtained for research purposes. Besides tracking data provenance, it is used for data quality and auditing compliance with the privacy framework. This component should maintain trace of all relevant executed processes and data accessed and/or modified. It should also provide an interface for querying data provenance.

An API will be defined whose methods can be easily called from other components in order to store the desirable provenance data. – RMI API, WSDL API

SVN is used as source code versioning system, while issues are tracked with Bugzilla. Builds are managed with Maven.

Dependencies:

This component uses the authentication and authorization functions of WT3.3 and WT7.5, and is used by the WT4.4 Evidence Service, WT4.5 Data Quality Tool, WT5.3 query formulation tool and workbench, WT4.5 data mining tools, and WT7.6 mobile middleware.

4.2.9 WT 7.1: Model-based Semantic Mediation Service

Service that implements the ontology developed in WT6.6.

Project management, as well as issue-tracking is managed by Redmine. All code is documented with JavaDoc-style comments for use with Doxygen or equivalent. SVN is used as source control system, while the building is done with Apache Ant, while open for alternatives.

Dependencies:

This component does not depend on any other software component and is used only within other software tools developed by University of Birmingham.

4.2.10 WT 7.2: Terminology Service

This component is an integrated terminology service to provide a public web service that provides access to clinical terminology and vocabulary translations.

Project management, as well as issue-tracking is managed by Redmine. All code is documented with JavaDoc-style comments for use with Doxygen or equivalent. SVN is used as source control system, while the building is done with Apache Ant, while open for alternatives. WSDL is used for the web services.

Dependencies:

This component does not depend on any other software component and is used only within other software tools developed by University of Birmingham.

4.2.11 WT 7.3: Metadata Repository

Development of common data elements (CDE) and case report forms (CRF) metadata extensible model and repository Service. The repository will store and provide vocabulary controlled CDEs and CRFs contributed by the primary care community to enable clinical researchers to reuse or construct and design new case report forms for capturing and collecting studies' data for the purpose of the TRANSFoRm use-cases.

Project management, as well as issue-tracking is managed by Redmine. All code is documented with JavaDoc-style comments for use with Doxygen or equivalent. SVN is used as source control system, while the building is done with Apache Ant, while open for alternatives.

Dependencies:

This component does not depend on any other software component and is used only within other software tools developed by University of Birmingham.

4.2.12 WT 7.4: Infrastructure for Deployment of eCRFs/Web Questionnaire with EHRs

This component manages the collection of clinical study data with functional eCRFs embedded in an eSource compliant way with eHR systems. Quality of life surveys are done with a web questionnaire, connected with the eCRF. This component also manages the interaction of eCRF and web questionnaire with the EHR systems to trigger study-specific and event-triggered data collection activities. In addition, it acts as a hub for the import of data from different databases and the web questionnaires, and the merging of all study data in a study specific database.

Project management, as well as issue-tracking is managed by Redmine. All code is documented with JavaDoc-style comments for use with Doxygen or equivalent. SVN is used as source control system, while the building is done with Apache Ant, while open for alternatives.

Dependencies:

This component depends on WT 4.5: Data Mining Tools and WT 5.1: Data Quality Tool.

4.2.13 WT 7.5: Infrastructure for Extraction/Linkage of Data

This component will manage the provenance of data obtained for research purposes. It is used for validating data quality and auditing compliance with the privacy framework. It should maintain traces of all relevant executed processes and data accessed and/or modified. It should also provide an interface for querying data provenance within TRANSFoRm.

Technologies used by this middleware component include:

- Apache Camel (open source integration framework)
- XML
- Java
- Maven
- various Camel integration/routing patterns (file IO, JDBC access, emailing, message queue)
- Apache Tomcat
- Shibboleth
- libraries for XML encryption and signing (Apache Santuario)

SVN is used as source code versioning system, while issues are tracked with Bugzilla. Builds are managed with Maven.

Dependencies:

This component depends on WT 3.3: Security Framework, and is used by WT 4.5: Data Mining Tools, WT 5.3 Query Tool and Data Extraction Workbench and WT 5.4: Provenance Service.

4.2.14 WT 7.6 Middleware for Mobile e-Health and Clinical Prediction Rule Derivation

This component is an additional federated infrastructure required to integrate the enlarged WT 5.5 and WT 4.5: Data Mining Tools into the existing project.

Dependencies:

This component depends on WT 4.5: Data Mining Tools and WT 5.1: Data Quality Tool, and will make use of WT 5.4: Provenance Service.

4.3 Component Timeline

Below (Figure 6) the different software components are listed in an overall timeline.

Tool		Start	End	Year 1	Year 2	Year 3	Year 4
WT3.3	Security framework (v1)	1/03/2010	1/06/2013				
WT3.3	Security framework (v2)	1/06/2013	1/06/2014				
WT4.3	Ontology service for diagnostic evidence	1/09/2010	1/12/2012				
WT4.4	Web-based evidence service (v1)	1/07/2012	1/06/2013				
WT4.4	Web-based evidence service (v2)	1/06/2013	1/06/2014				
WT4.5	Data mining tools	1/06/2012	1/06/2013				
WT5.1	Data quality tool (v1)	1/12/2011	1/06/2012				
WT5.1	Data quality tool (v2)	1/06/2012	1/12/2012				
WT5.2b	Triggering tool	1/12/2012	1/12/2013				
WT5.3	Query and data extraction workbench (v1)	1/12/2011	1/06/2012				
WT5.3	Query and data extraction workbench (v2)	1/06/2012	1/06/2013				
WT5.4	Provenance service	1/03/2011	1/06/2012				
WT7.1	Model-based semantic mediation service	1/12/2011	1/12/2012				
WT7.2	Terminology service	1/03/2010	1/09/2011				
WT7.3	Metadata repository	1/12/2011	1/12/2012				
WT7.4	Infrastructure for deployment of eCRFs into EHRs (v1)	1/12/2012	1/06/2013				
WT7.4	Infrastructure for deployment of eCRFs into EHRs (v2)	1/06/2013	1/12/2013				
WT7.5	Infrastructure for extraction/linkage (distributed infrastructure) (v1)	1/03/2011	1/06/2012				
WT7.5	Infrastructure for extraction/linkage (distributed infrastructure) (v2)	1/06/2012	1/06/2013				
WT7.5	Infrastructure for extraction/linkage (distributed infrastructure) (v3)	1/06/2013	1/06/2014				
WT7.6	Middleware for mobile e-Health and CPR derivation	1/06/2012	1/06/2014				

Figure 6: Dependency graph

4.4 12-Month Integration Timeline: Jun 2012 - Jun 2013

Depicted below (Table 17) are the different software components that will be integrated in the next twelve months. Every three months this integration planning will be revised at the 3-month milestones/6-month baselines (see 3.1) and an integration plan for the next twelve months will be drawn.

Components to be integrated	Repsonsible teams	Start	End
WT5.4 + WT5.1 (v1)	IC & NIVEL	1/06/2012	1/12/2012
WT5.1 (v1) + WT5.3 (v1)	UB & NIVEL	1/06/2012	1/12/2012
WT5.4 + WT5.3 (v2)	IC & UB	1/12/2012	1/06/2013
WT5.3 (v2) + WT7.5 (v2)	TCD & UB	1/12/2012	1/06/2013
WT5.1 (v1) + WT7.5 (v2)	NIVEL & TCD	1/12/2012	1/06/2013
WT5.4 + WT7.5 (v2)	IC & TCD	1/12/2012	1/06/2013
WT4.4 (v1) + WT7.5 (v2)	RCSI & TCD	1/12/2012	1/06/2013
WT4.5 + WT4.4 (v1)	RCSI & IC & WRUT	1/12/2012	1/06/2013

Table 17: Integration timeline

WTs 7.1, 7.2, 7.3 and 7.4 are managed by University of Birmingham, and they are not integrating with any other tools, so it is assumed that they will handle the internal integration themselves, and no separate time is set for this.

5 Summary

This deliverable should be considered as the first step of a continuous process for developing a software integration plan that will accompany the development process in TRANSFoRm. It provides guidance for component delivery, test environment integration and the creation of a staging environment for the TRANSFoRm solution and it lists for each stage the responsibilities, testing methods and the necessary documentation. The emphasis is on the delivery of a practical approach which is feasible for the complex TRANSFoRm structure, and which incorporates best practices, sound tools and techniques, which can be applied in the project and other settings.

The TRANSFoRm software development lifecycle is a 6-month incremental cyclic process. The goal at the end of each 6-month period is to provide an incremental and functional prototype of the software being developed. Before the start of a software development lifecycle the requirements that should be met at the end of the 6-month period are identified (by project managener and component owners), partitioned into tasks for the team (by component owner), which are then assigned and scheduled within each team of developers (also done by component owner). At the month-3 milestone meetings, it should be clear to the component owners and project management whether the planned requirements can be implemented and tested by month 6. The coding, verification and validation of the component is also a cyclic process. This is repeated over and over, until a stable, verified and validated system is composed. This cyclic process is called a software release cycle (or release cycle). It has proven useful to divide this process into separate, consecutive stages within that cycle. In practice, four stages can be typically observed: DEV, SI, PAC and PROD. Each stage corresponds to an environment with an identical name.

A special versioning scheme for software components is suggested, because agreeing on a common methodology for component versioning is highly advisable for correct and efficient interaction between multiple component development teams and increases development transparency. The suggested versioning scheme is <component_name> X.Y-SUFFIX, with X the major release version number and Y the minor release version number. SUFFIX can be SNAPSHOT for builds in DEV-stage or RCx for release candidates in SI- or PAC-stage, x being the number of the release candidate. The abscence of a SUFFIX means that the version is final. The suffix SNAPSHOT indicates that this version of the component should not be considered as stable and is still subject to regular changes. The RCx-suffix (release candidate) indicates that this version is allowed outside the DEV environment. It states that this version is intended to be put on SI or PAC. Everything that still has an RC-suffix or SNAPSHOT-suffix should be considered unstable. Therefore these suffixes are a useful remedy to prevent unstable building blocks to end up in the final “product”, weakening the complete structure.

The release cycle starts in the DEV-stage where developers start to work on their implementation work (feature, improvement, bug fix). The environment developers work on (DEV) is different than the environments in other stages, in the sense that DEV is under the control of the development teams. This environment is used by the developers to test their newly created code through unit-tests. These unit tests are performed by the developers themselves or by their peers. When the component owner decides that the goals of the current development cycle (implemented features addressing certain

requirements, bug fixes) have been met, a request for a release candidate can be made. The SQA Manager inspects the provided documents, and decides together with the project manager if a release candidate can be built and move to the SI-stage.

This RC-build will be deployed on the SI environment by an independent party, i.e., another team that was not involved in the development of this component. If there are problems to get the components integrated, the component must go back into DEV-stage. Integration tests and also some regression tests are performed by combined component teams, i.e., part of the component team and part of another component team. All test results and necessary documents are sent to the SQA Manager, who will inspect them and decide together with the project manager whether the component is allowed to be promoted to the SI-stage.

Once all tests in SI-stage all successfully completed and all documents have been inspected by the SQA Manager, the component can move on to the next stage in the release cycle, the PAC-stage, which implies deployment on the PAC-environment. As in the SI-stage, this deployment is also done by independent component teams. System tests and acceptance tests are performed in the PAC-stage by a test team. Issues found during PAC testing should be reason to send the component back into DEV-stage, with the intent of having the issues fixed. Therefore a change request will be created. Of course this means that once the issues are fixed in DEV, it has to go through SI again, before it comes back into PAC where it will be tested again. After inspecting the test results and all provided documents, the SQA Manager decides together with the project manager if the component can move up to the PROD-stage.

At all times the component owners must share the status of their component with the integration manager. If teams should fall behind, the integration manager reports this to the project manager who will follow up.

The system environments setup by KCL are SVN, SI and PAC. PROD will be setup by UDUS at a later time, as no components will reach the PROD-stage in the first years.

Deliverables as required by legislation are listed for each stage. At each transition from one stage to another, the component owner must consider whether all deliverables are produced at the end of the stage. If a deliverable is not produced, a short explanation as to why must be provided. Also checklists are provided for each stage. These allow the component owner to establish whether the development has completed the quality requirements of that stage. At the end of each stage, the component owner should answer each of the questions listed. If the answer to any question is ‘No’, the component owner must give a short explanation as to why. These checklist responses will be validated by the project manager.

An overview of all identified software components is given, and an overall integration timeline is depicted. For the next 12 months all components to be integrated are listed in detail, along with the responsible teams. Every 3 months a new 12-month integration timeline will composed.

According to the project plan, the integration plan will be reviewed at each Steering Committee meeting. The Steering Committee will consist of each of the WP lead plus the Technical Director and

Scientific Project Manager. Subsequent progress and alterations to the plan will be reported annually in the project report. Especially the setup and maintenance of the necessary tools for assisting the software integration plan (e.g., continuous integration environment, central documentation and issue tracking, etc.) and experience during the management of the integration environment infrastructure will require some adaption of the integration plan.

List of Abbreviations

CDE	Common Data Elements
CDIM	Clinical Data Integration Model
CI	Continuous Integration
CMMI	Capability Maturity Model Integration
CRF	Case Report Forms
CRIM	Clinical Research Information Model
CSV	Comma Separated Value
DEV	Development
DoW	Description of Work
ECRIN	European Clinical Research Infrastructure Network
EHR	Electronic Health Record
FTP	File Transfer Protocol
GCP	Good Clinical Practice
KNIME	Konstanz Information Miner
PAC	Project Acceptance
PROD	Production
RC	Release Candidate
REST	REpresentational State Transfer
SCM	Software Configuration Management
SDLC	Software Development Lifecycle
SG	Specific Goals
SI	Software Integration
SOP	Standard Operating Procedure
SP	Specific Practices
SQA	Software Quality Assurance
SVN	Apache Subversion
WP	Work Package
WT	Work Task
XML	Extensible Markup Language
X-testing	Cross-testing