Optimization of software installation procedures

Orlando Mendez

Partners Philips Healthcare Eindhoven University of Technology

Date November 23rd 2009

Optimization of software installation procedures.

Stan Ackermans Institute / Software Technology

Contact Address	Eindhoven University of Technology. Department of Mathematics and Computer Science. Department of Mathematics and Computer Science HG 6.57, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands +31402474334
Published by	Eindhoven University of Technology Stan Ackermans Institute
Printed by	Eindhoven University of Technology Universiteits Drukkerij
Status	Final version
ISBN	978-90-444-0913-0
Preamble	Partial software updates in Allura cardiovascular systems are required because the current update procedure is expensive, time consuming and therefore, represents a source of dissatisfaction both for Philips Healthcare and its clients. Partial software updates proposed in this report have reduced the current installation time between 25% and 61%. The reliability of this partial installation process allows the Cardiovascular Business Unit of Philips to consider releasing automatically their software to the systems in the field, which will contribute to satisfaction of the customers. Furthermore, because the size of a partial installation set is significantly smaller than that of a full installation set, the distribution (via computer network) of this smaller set and hence its remote installation become feasible.
Keywords	Software Installation, Partial Installations, Delta packages, MD5, validation
Preferred reference	Orlando Mendez, TITLE: Optimization of software installation procedures. Eindhoven University of Technology, SAI Technical Report, November, 2009. (ISBN 978-90-444-0913-0). Eindverslagen SAI; 2009/070. A catalogue record is available at the Eindhoven University of Technology Library.

Partnership	This project was supported by Eindhoven University of Technology and Philips Healthcare.
Disclaimer Endorse-ment	Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the Eindhoven University of Technology or CLIENT. The views and opinions of authors expressed herein do not necessarily state or reflect those of the Eindhoven University of Technology or CLIENT, and shall not be used for advertising or product endorsement purposes.
Disclaimer Liability	While every effort will be made to ensure that the information contained within this report is accurate and up to date, Eindhoven University of Technology makes no warranty, representation or undertaking whether expressed or implied, nor does it assume any legal liability, whether direct or indirect, or responsibility for the accuracy, completeness, or usefulness of any information
Trademarks	Product and company names mentioned herein may be trademarks and/or service marks of their respective owners. We use these names without any particular endorsement or with the intent to infringe the copyright of the respective owners.
Copyright	Copyright 2009. Eindhoven University of Technology. All rights reserved. No part of the material protected by this copyright notice may be reproduced, modified, or redistributed in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the Eindhoven University of Technology and CLIENT.

Foreword

Installation. Software installation. From the outside it looks so simple. One puts a DVD in a drive, hits enter and within (hopefully) a few minutes the new software is installed on the system and ready to use. In practice it is hard to achieve this simple functionality, certainly in a complex development environment as we know it within our business unit.

Our ultimate goal is to have a fully automated software distribution and installation process where our customers quickly benefit from the new developments we provide. Orlando's work has been a major step towards this goal.

When looking backwards, the project has been a rough journey. We encountered steep hills, many hurdles where there to take, perseverance has brought the project where we are now. The challenge of taking the results a step further has been taken up by our organization. A project leader has been appointed to take the results and bring it to the internal test- and integration process. After successful deployment it can be released to our customers.

Best, November 2009

Peter Dingemans. Software architect

Philips Healthcare, Business Unit cardio / vascular.

Preface

Have you ever wondered how soon your computer would be limited in functionality because it does not have its software updated? Or what would happen if the cardiovascular system which you were supposed to be intervened with is not available due to a failure in its software? These questions popped into the author's mind while working on the project described in this report. This report is the main deliverable of a project carried out at the Cardiovascular business unit of Philips Healthcare, in the context of the two-year Software Technology program of the Stan Ackermans Institute (3TU.School for Technological Design.)

Nowadays software is a critical factor for business success and more importantly for patient safety in many devices. Medical imaging devices rely on the safe and stable operation of the software that controls them, and their operation begins with the software installation process. Software installation is a supporting process to the production of Cardiovascular X-ray systems. In the case of this type of systems produced by Philips Healthcare, the magnitude of the installation process is extensive when compared to other types of embedded systems such as mobile devices. The main goal of this project is to make the software installation of cardiovascular systems faster, more managed, and remote.

Although the target audience of this report is people with a technical (computer science) background, the author has made an effort to make it accessible to broader audiences. To accomplish this aim, there is detailed information inside the report as well as a number of available on-line references.

Acknowledgments

I want to thank first my company supervisor, John Vugts for providing me with this challenge. My university supervisor Alexander Serebrenik has given me valuable feedback throughout the project. Special thanks are due to the OOTI program management, Ad Aerts and Maggy de Wert. Mevrouw Nelleke de Vries deserves also special acknowledgment for her valuable teaching of the Dutch language.

I am in debt to the following list of persons at Philips Healthcare in Best - no particular order: Radhakrishnan Kodakkal, Kees Rijnierse, Peter Dingemans, Tiemco de Jong, Len Jebbink, Ronald Dons, Alex Visser, Paul Langemeijer, Francois Louw, Bram Jansen, Roderick Polder, Paul van Geldrop, Nico Barendse, Pieter Beks, Ivo Canjels, Rob Luijten, Duncan Stiphout, Frank Spronck, Marc Loos, Peter Chao, Ron Dielhof, Jan Soetens, Frank Stevens, Pascal van Kempen, Stefan Smits, Piet Tax, Henrie Gielen, Ingmar Dal, Richard Philipsen and Guillaume Stollman.

Last but not least, there has been special support from different people: families Alonso, Korevaar and Mulderij for their financial aid. Friends like Heidi Romero, Pascha v.d. Breekel, Mara Saeli, Leon and Maartje Merkx, Martin Kavuma and Giovanni McDowell for the many received favors.

To my three treasures: Viviana, Amaia and Zitsli. Thank you for your support during those long waiting hours.

Thank you all.

Orlando Mendez

November 23rd, 2009.

Executive Summary

Partial software updates in Allura cardiovascular systems are required because the current update procedure is expensive, time consuming and therefore, represents a source of dissatisfaction both for Philips Healthcare and its clients.

In order to achieve the installation time reduction, a number of changes need to be implemented in the current installation process regarding:

1. THE GENERATION OF THE FULL INSTALLATION SET
    - (a) **All subsystems must be released in installer packages (.msi) with a three-component version number.** Currently they have a four-component version number.
    - (b) **Subsystems must not have nested installer packages**. Tests done with nested installer packages show that such packages do not perform updates correctly. Currently there are three subsystems with nested installer packages.
    - (c) **Limit the use of custom actions**. Post install tools and custom configuration actions are currently performed in a number of ways: 1) By the post install tools invoked by the system installation script after MSI installation; 2)Within COM registration.
      These actions modify the installed files at some stage during installation. Because these steps are unmanaged, it becomes impossible to track what these modifications are, and therefore the validation method here proposed cannot take this into account when calculating MD5 hashes. There are two solutions to this problem: 1) Make a policy that custom/actions and post install tools do not modify the installed files in anyway; and 2) Define a number of standard custom actions and allow only those to be used by developers.

2. THE INSTALLATION PROCEDURE ON THE ACTUAL SYSTEMS. The introduction of partial installation is a step towards the full automation of (remote) software installation of systems in the field. Currently software is installed via a manually invoked script . With partial installations a script /program should be triggered that automatically:
    - (a) shuts all running application processes down;
    - (b) updates the system's software by installing the partial installation set;
    - (c) checks the outcome of installation and takes appropriate actions to ensure the safe operation of the system.

Partial software updates proposed in this report have reduced the current installation time between 25% and 61%. The reliability of the partial installation process allows the cardiovascular business unit to consider releasing automatically their software to the systems in the field, which will contribute to satisfaction of the customers. Furthermore, because the size of a partial installation set is significantly smaller than that of a full installation set, the distribution of this smaller set (via computer network) and thus its remote installation become feasible.

The work done during this project identified all the changes presented in group one of the list above, and solves 2b and (partially) 2c. Therefore, it is recommended to the Cardiovascular business unit of Philips, to address specially group one and 2a mentioned above.

Contents

List of Figures

List of Tables

Glossary

Expression/ acronym	**Meaning**
AUA	Allura Update Application
BCD	Boot Configuration Data. BCD is the new Windows boot manager. It replaces the boot.ini file and is used in Windows Vista and above
BOM	Bill of Material
COTS	Commercial Off-The-Shelf components. IN this document, software applications ready-made and available for sale, lease, or license to the general public.
CRC	Cyclic Redundancy Check. is a non-secure hash function designed to detect accidental changes to raw computer data.
CV	Cardio/vascular (system). Cardio/vascular is also the name of the business unit where this project was carried out.
Delta package	A package containing the binary differences between two releases of a software product [2]. Given the constraints discussed in section “Operating system platform”, a delta package matches Microsoft’s definition of Patch Package introduced in [3].
FAMAR	Faster, Managed and Remote. Acronym used as a shortcut to the name of the project which is documented in this report.
FCO	Field Change Order, which can be not only a SW change, but also HW (e.g. a physical part of the system).
FSE	Field Service Engineer. An engineer who provides technical support and maintenance to Allura systems.
Field Service Delivery	Area responsible within Philips Healthcare of providing infrastructure and content to perform maintenance of Allura systems in the field
FSF	Field Service Framework. The FSF provides local and remote servicing for medical equipments.
HW	Hardware
ICE	Internal Consistency Evaluator. Custom actions written in a script or executable program. When ICES are executed, they scan the MSI database for entries in database records that are valid when examined individually but that may cause incorrect behavior in the context of the whole database.

Install Shield©	An authoring tool to create MSI packages and InstallScript installations for Windows platforms. More details in [4].
Installation set	A set of files containing a setup program that is used to install a software release. This set of (MSI) files can be delivered as one self-extracting executable
Major Upgrade	A comprehensive update of the product warranting a change in the ProductCode property. Can be shipped as a full installation package (MSI) or as patch package (MSP) [3]. A typical major upgrade removes a previous version of an application and installs a new version
Minor Upgrade	A small update making changes significant enough to warrant changing the ProductVersion property. Can be shipped as a full installation package (MSI) or as patch package (MSP) [3]
MID	Master Integration Diagram. In Software Configuration Management, it is a diagram used to determine in which sequence the features are delivered to a (pre-)integration stream
MSI	Microsoft installer. MSI is the file extension of installer packages used in the Microsoft Windows platform
MSP	Microsoft Patch file. Throughout the text of this report, we will use MSP or delta package as equivalent terms.
PACS	Picture Archiving and Communication Systems are computers or networks dedicated to the storage, retrieval, distribution and presentation of medical images
PBL	Product Base Line. A number given by SCM to a collection of MSI files. This number is used for identification of software releases in CV systems released to hospitals.
PC	Personal Computer
PDC	Product domain content. A version number representing an installation set generated on a biweekly basis by the development area. This number is used for development and administrative purposes. For each PBL releases delivered to hospitals, there are several PDC releases made during development
QSA	Questra Service Agent. An application used in Allura systems to remotely monitor Allura systems and distribute installation software packages
Rebasing of DLLs	Portable Executable (PE) files such as Dynamically Linked Libraries (DLLs) are compiled to a preferred base address, and all addresses emitted by the compiler/linker are fixed ahead of time. If a PE file cannot be loaded at its preferred address (because it is already taken by something else), the operating system will rebase it. In this project, rebasing of PEs is done at installation time.

RSN	Remote Service Network
SAP	System Analysis and Program Development
SCM	Software Configuration Management. Organizational area within the Cardiovascular business unit at Philips Healthcare
SDK	Software Development Kit
SRS	Systems requirement specification
SW	Software
SWAT	Software Architecture Team
TAF	Test Automation Framework
Update package	A package that adheres to the format specified in section 5 of [5]. Important attributes of this package are: a) it is one file, b) must specify its pre-requisite configuration, and c) must specify its resulting configuration
XML	Extensible Markup Language

Chapter 1

Introduction to FAMAR

Preamble

In this chapter, an introduction is given to explain the general context of this project, related notions (e.g. field, factory, development, Allura Xper), the challenges faced during the installation and the way these will be addressed.

1.1 Context of the project

The goal of the project presented in this report is:

> "*[To] Define, design and realize means to have partial managed updates of a complex software (SW) system that is still manageable with reference to testing, installation on different system configurations and proving the correctness of the system*".[6]

Philips Healthcare produces complex state-of-the-art Cardiovascular (CV) X-Ray systems for 2D and 3D imaging of the human cardiovascular system. These systems are commercially known as the Allura Xper series, called Allura for short in this report. These systems are from the medical point of view very useful as they can aid doctors in diagnosis and treatment of heart diseases, such as calcific aortic stenosis [7]. Allura systems also can interact with other medical systems (e.g. ultrasound devices) and infrastructure in hospitals to store and analyze patient information.

An Allura system consists of a large number of computing nodes connected using different network technologies. These computing nodes address a wide set of challenges such as geometry control, image acquisition, image processing, viewing, and communication with the hospital infrastructure. Figure 1.1 shows the parts of the system that a patient normally sees.

Figure 1.1: Allura system, model 20/10

The software applications for this CV system are developed with modern technologies (e.g., COM and .NET), and programming languages such as VC++ and C#. These software applications are packaged, distributed, and deployed for installation in three different areas known as development/testing, factory and field. An overview of these areas and their relationship with the entire installation process is depicted in 1.2. The explanation of this overview is as follows:

1. As step 1, the Allura software is developed by the development teams of Philips Healthcare, which deliver the subsystems composing the system.

2. Each subsystem has its configuration management department, which produces an installable package for that subsystem -step 2.

3. At system level -step 3, the configuration management department receives the subsystem installable packages and generates from these an installation set and the tooling required to install it on the hardware.

4. The testing teams of development use the installation set to validate its correctness (step 4). This validation is done through the installation (step 6) and configuration (step 7) of the software in a number of test systems. These are iterative steps, whose iteration stops after passing the required tests.

5. The installation set is published (step 5) for distribution to the field (step 8) and usage in the factory (step 9)

6. The software is installed at the factory on the produced systems via an installation image, i.e. an installation that has been actually made into one system, and then copied to other systems (step 10)

7. After installing the software image, each system is configured according to customer's specifications (step 11)

8. Once distributed, a new version of the installation set is installed on systems in the field (step 12) by the service department, and the system is configured (step 13) if required.

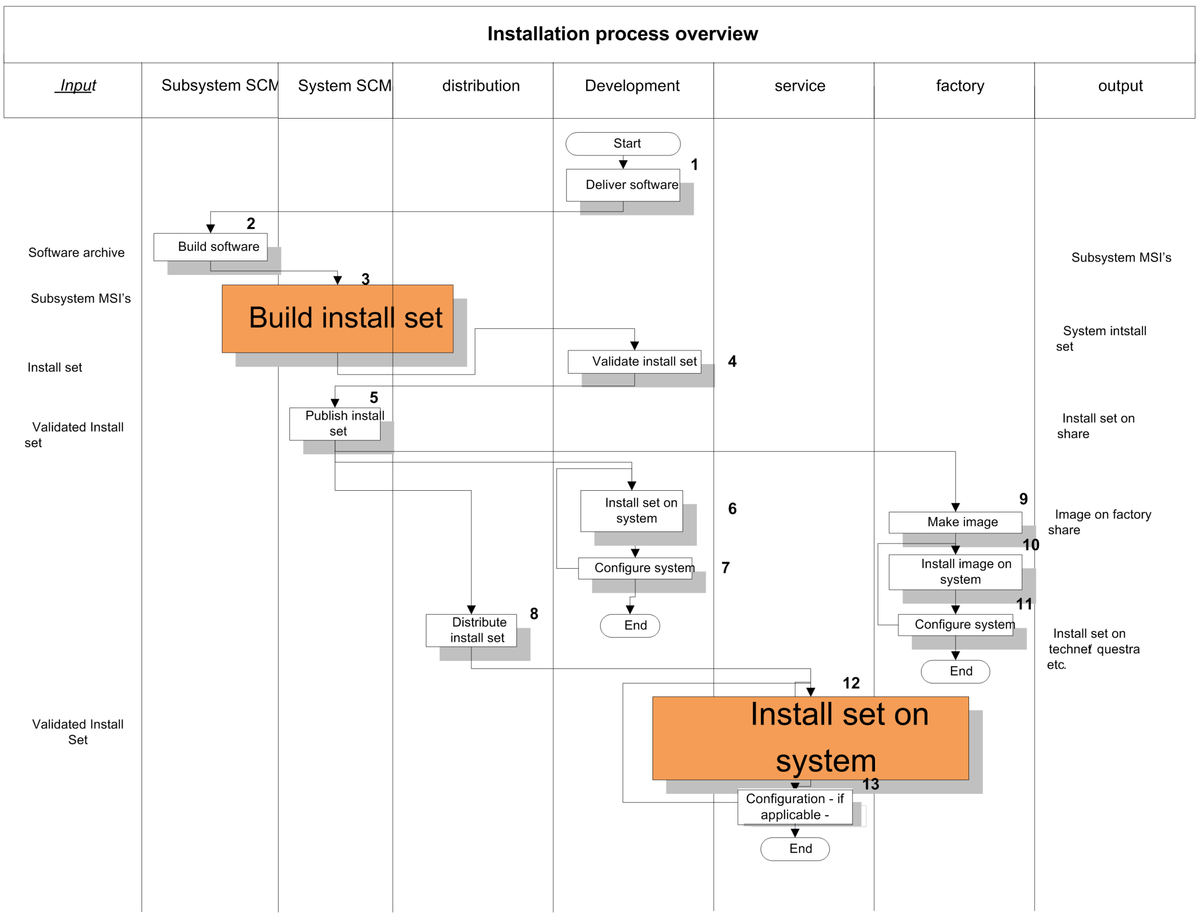


Figure 1.2: Installation process of Allura SW system wide

The main focus activities in this project pertain to improvement of the generation of the install set (step 3) and its installation, which is a recurrent activity (steps 6, 10 and 12). In the coming list, more detail is given about installation:

- **Development/testing.** Development and testing occur at Philips Healthcare development units at its premises in Best. Developing a CV system targets a wide variety of platforms used at the hospitals. Hence, different system (software/hardware) configurations are required for development and testing. These configurations are partial because it is not feasible to assemble complete CV systems in terms of costs, effort, space, and material resources.

- **Factory.** The term factory refers to the Philips factory premises in Best, the Netherlands. The factory is the place where the Host PC, an important computing node, is installed. The Host PC contains many of the software applications used in the system and serves as the entry point for installing the rest of software to other computing nodes (e.g. headless PCs).

- **Field**. By the field we mean locations of regular deployment and operation of the CV systems, i.e., hospitals.

In all three areas it is highly desirable for the installation to be Fast, Managed and Remote (FAMAR):

Fast. Currently the installation of a software update takes more than five hours -as reported in [8]. The financial costs associated with this update are recurrent since for every CV system, software is installed on at least three different occasions, namely during development, at the factory and periodically in the field. There is a need to reduce the installation time, thereby contributing to the installation cost reduction both for Philips and the hospitals.

Managed. In the medical domain there are strict legal regulations prescribing that the manufacturer has to build up evidence about the correctness of SW configuration of every system in the field. For instance, [9] states:

> “When computers or automated data processing systems are used as part of production or the quality system, the manufacturer shall validate computer software for its intended use according to an established protocol. [An] established protocol should be interpreted as the documentation plus the actual procedures that implement the protocol. “

Software (SW) installation is a process that supports the functioning of a CV system (the product), and any addition or change to this process needs to be documented like the software running on the product itself. In the coming chapters — specially Chapters 5, 7, and 10— we will see that the current installation process needs to change to achieve the goals of this project.

Remote. Two features that contribute to this aspect are remote distribution and installation of software. In this project we want to contribute to this goal by achieving remote software updates in Allura systems. Although there is a network infrastructure available to distribute software, this has not been used to distribute software updates of Allura software yet.

1.2 Product Roadmap

The author considers the product roadmap of Allura systems as part of the project's context. In [10], the product roadmap for the Allura family is described. The Allura product family began 10 years ago with the release of Allura Xper FD10. As of the time of writing, the release 7 of Allura Xper is taking place. From this release onwards, the functionality and network infrastructure are available to distribute partial software updates .

Beginning with the software release for Allura Xper release 7, all future releases can be partially and remotely updated in their software.

1.3 Outline

The rest of this technical report is organized as follows:

In Chapter 2 we introduce the main stakeholders and how are they related to the installation process of Allura systems and to FAMAR itself.

Chapter 3 states the general goal of the FAMAR project. Next, the challenges enunciated in Section 1.1 are decomposed and their associated root causes are discussed. Finally, a selection of focal points to tackle during this project is presented along with the methodology followed.

In Chapter 4, we give an overview of the domain in which FAMAR is executed. We introduce the activities associated with the software installation process of Allura systems in each of the three areas mentioned in Section 1.1. Next, we explain the phases of the installation process, and close with the generation of the installation SW set and the configuration management used to control versions of the Allura SW.

Chapter 5 presents the elicitation of requirements for this project, taking as reference use cases that the solution to the problems posed in Chapter 3 should cover.

Chapter 6 describes the feasibility of the technology and tools necessary to satisfy the requirements posed in Chapter 5. In Chapter 6 we introduce the reader an analysis of tools used to generate delta packages and validate their installation, along with some lessons learned from such exercise.

Chapter 7 discusses the design of the solution to achieve partial installations. The use cases discussed in Chapter 5 enable us to define the design in terms of generation of delta packages, validation of these before their distribution, installation of the packages at the target system, and a validation method to ensure that the system is left in a consistent state in terms of configuration. Should the partial installation fail, we also present a recovery method .

Chapter 8 describes the way the designed solution is implemented and deployed, whereas Chapter 9 addresses the validation and testing methodology followed in the project.

Chapter 10 summarizes the concluding findings of this project, along with a set of recommendations for the stakeholders who receive the project results. Finally, Chapter 11 explains how the project was managed and the lessons the author has learned from this activity.

In the coming chapter, we present an analysis of the stakeholders involved in this project.

Chapter 2

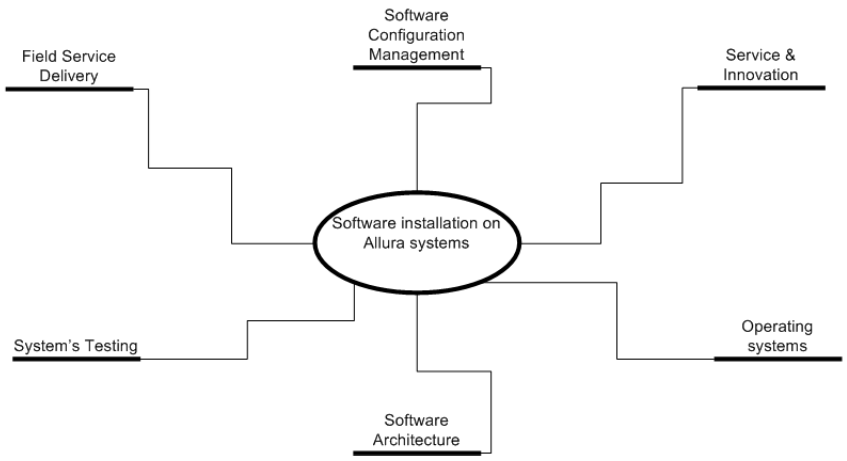
Stakeholder Analysis

“ Every stakeholder group within the industry has strong thoughts and ideas about what’s in the best interest of their group, as one would expect them to. Our concern is what’s in the best interest of the entire system.”

Danny Davis; cited from [11].

Preamble

In this chapter an overview is given of the most important stakeholders for the software installation process of cardiovascular systems at Philips Healthcare. By analyzing for whom this project is meant, we obtain an insight of the interests and expectations of those persons involved in the installation process.



2.1 Stakeholders overview

Figure 2.1 illustrates the relationships among the stakeholders described in the previous sections and the SW installation process of Allura systems.

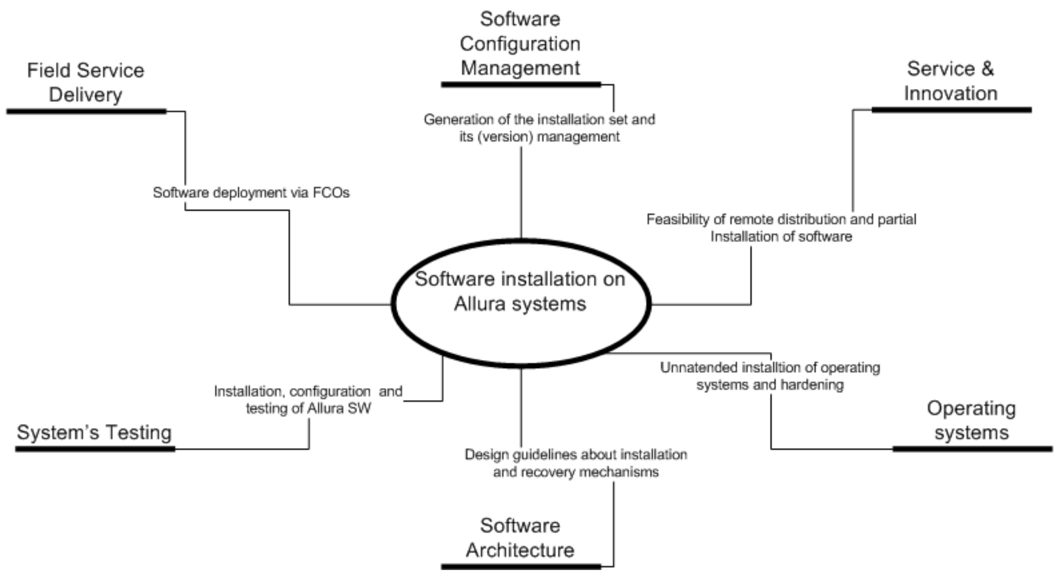

Software Configuration Management
Field Service Delivery
Service & Innovation
Generation of the installation set and its (version) management
Software deployment via FCOs
Feasibility of remote distribution and partial Installation of software
Software installation on Allura systems
Unnatended installtion of operating systems and hardening
Installation, configuration and testing of Allura SW
Operating systems
System's Testing
Design guidelines about installation and recovery mechanisms
Software Architecture

Figure 2.1: Stakeholders and their relation to the software installation process of Allura systems

Service and Innovation is a department studying feasibility of new service quality improvement methods such as (re)installation of software. This is the main stakeholder at Philips Healthcare.

Software Configuration Management is responsible for creating and defining the version control policy of the software installed in Allura systems. The installable software created is called installation set, which is discussed in more detail in Section 4.3.

Sosftware architecture dictates the design guidelines for the installation of software in Allura systems. This stakeholder also ensures that the solution produced in this project matches the ongoing (Allura) system-level efforts related to the installation process.

Field Service Delivery distributes and installs software updates to hospitals via Field Change Orders (FCOs). These orders are related to maintenance of Allura systems, and they can involve software as well as hardware.

System Testing is an important area that uses the software installation set. Given the frequency of (partial) installations performed during development, this stakeholder provides relevant insight in the overall installation process discussed with more detail in Sections 3.2.2 and 4.2.

Finally, the Operating Systems (OS) group provides support for automated installation of operating systems present in Allura systems. The OS group not only installs operating systems, but also ensures their further maintenance by controlling the deployment of OS-related updates to systems in the field. Furthermore, the OS group currently gives maintenance to the tooling used to generate the MSI installation set.

2.2 Summary

In Table 2.1 the expectations of each stakeholder are written as expressed by stakeholders themselves.

Table 2.1: Stakeholders' expectations about the FAMAR project

Stakeholder	Expectations
Service and Innovation	The scheduled maintenance time for a software update in the field should reduce from more than five hours to one hour. The software update process/mechanism is reliable
Software Configuration Management	The installation set should be generated and released as a consistently versioned set of software. It should always be possible to determine which versions of software are installed
Software Architecture	Low system downtime (less than 5 minutes) for simple software updates. Dummy proof installation process, at least as (simple as) Windows.
Field Service Delivery	Automated installation and remote updates contribute to solve software installation issues in the field -See issues in Subsection 3.2.4.
System Testing	An installation without incidents caused by incorrect configuration
Operating Systems	Automated installation / update of the operating systems needed by an Allura system

Chapter 3

Problem Analysis

"Okay, Houston, we've had a problem here"

John Swigert, Jr. during NASA's Apollo 13 mission.

Preamble

In this chapter we present an overview of the main problem, that is: how do we make the software installation of Allura systems faster, more managed, and remote?

3.1 Goal of the project

As reported in [8], the installation of software updates for one Allura system takes more than five hours. Installation is thus a very expensive process as it has to be carried out multiple times for every system during its development, in the factory, and in the field (see Sections 3.2.2, 3.2.3, and 3.2.4 respectively). Installation of software is a recurring cost and hence a problem for Philips Healhtcare.

Moreover, for field installations, a software update can be in the order of Gigabytes in size and thus the distribution of the update to the hospitals takes prohibitively long. The current solution is to distribute the software using a DVD.

The ultimate goal of the FAMAR project is

> "[To] define, design and realize means to have partial managed updates of a complex [Allura] software (SW) system that is still manageable with reference to testing, installation on different system configurations and proving the correctness of the system".[6]

Managed updates in the project's context can be further detailed in a number of aspects:

1. **From the software configuration management perspective**: The existing process generates the (full) installation set and delivers a consistently versioned set of software for Allura systems. This consistency in versioning facilitates manageability, which is the ability to gather information of the installation set releases and control these. Manageability is expected to be present with other types of installation too.

2. **From the software distribution perspective**:The size of the installation set of the latest release is 1.3 GB (excluding operating systems). For the coming release, the expected size of the installation set will increase to 2.6 GB, and for the subsequent release, it is estimated that the installation set size will grow up to 4GB. The size increase demands making the installation software manageable, so that it can be quickly and easily distributable.

.

In order to achieve remote distribution and partial installation of software, a number of challenges have to be addressed:

1. **Generation of delta packages**. A delta package is an installable binary that incorporates only the functional changes of the latest version. A delta installation set is composed of one or more delta packages. The delta installation is not currently generated for Allura systems. The reasons behind this problem are explained in Table 3.1 (in the penultimate row, second column).

2. **Distribution and installation of delta packages into systems in the field**. Once the generation of delta binaries is achieved and automated, the next step is to distribute them and install them into the Allura systems deployed.

3. **Full install (roll-up) v.s. delta upgrades strategy**. An updating policy needs to be in place to determine when it is preferable to install the complete installation set and when is better to install a delta installation set, introduced in the first item above.

4. **Version control**. Given the current versioning policy of the SCM department, it is not clear which version should be present in the system once a partial installation is done.

5. **Dependency validation**. There are dependencies between subsystems, which need to be verified in order to partially install. This installation must be verifiable, since legal regulations demand evidence of the correct behavior and safety of the system after a software change has occurred.

6. **Recovery mechanism for partial installations.** Installation of software needs to be transactional: either it is done or it is not. If it is not, then the system must be recoverable, which means that the software of the previous version should be put back in the system.

7. **Integration of the recovery mechanism with partial installations**. Given the fact that the current recovery mechanism is a backup of the complete hard disk partitions in the system, what other recovery alternatives exist when a partial software update fails?

3.2 Problem decomposition

The seven challenges described in the previous section are the result of a wider problem decomposition task performed during this project. The decomposition was made by means of Root Cause Analysis (RCA) techniques. We start by discussing the methodology followed.

3.2.1 Methodology

1. **Creation an initial list of problems**

   We arranged a number of interviews of several stakeholders involved in the installation process. From these interviews we extracted a list of issues perceived.

2. **Application of RCA techniques**

   From the initial list of problems/issues gathered in the previous step, we distilled a number of them by applying the Five Whys technique. The Five Whys technique was applied in three sessions with assistance of project mentors, and its results were validated by the interviewees. Basically, this technique is a question-asking method used to explore the cause/effect relationships underlying a particular problem. Ultimately, the goal of applying the Five Whys method is to determine a root cause of a defect or problem [12].

After the validation process of issues observed and analyzed, the following subsections show the installation issues related to the software installation process of Allura systems.

3.2.2 Issues in development

Table 3.1 displays the problems found during development of the Allura applications and their root causes. Please note that the issues typeset in bold also affect the installation process in the field. We stress the importance of the field because the main stakeholder in this project focuses on providing a better service to the users of Allura systems.

Table 3.1: Installation issues during development

Issue	Root cause
1. Automatic SW installation (in development/field) has not been fully achieved. This lack of automation makes installation inefficient and error prone.	Automatic installation has not been a priority for the Cardio Vascular (CV) business unit of Philips Healthcare
2. Patches should be part of automatic installation. Since developers do not have access to tooling for authoring MSI files, many ad-hoc tools are used for installation. These tools become obsolete/useless as the developed SW becomes more stable, since this tooling is not reusable from one release to another.	It is expensive to have licenses of tools for authoring MSI files installed on each development machine
3. Not all installation scripts can resume their execution after a system's restart. Since system restarts occur several times during an installation, system restart is a time-consuming activity.	An installation process capable of resuming after reboot has not been a priority for development department within the CV business unit.
4. The most complex and error prone part of the installation process is the input of system's settings, i.e. the configuration.	The complete set of settings is not validated at the system level due to independent subsystem life cycles, meaning that configuration changes at subsystem level are not sufficiently validated when the system is integrated
5. During an upgrade, it is very time consuming to have to install all versions of all subsystems again, instead of having to install only binary patches which are faster to download, copy, and/or to install	**It is technically difficult to manage versioning of non-binary files (.XML, .bat, .ini, ...) related to the system, since version information cannot be attached to non-binary files.** **Philips cannot generate binary patches of the install set.**
6. The SW controlling the CV system is not clever enough to detect the exact HW configuration present in the system	It is expensive to design and develop smart HW

3.2.3 Issues in the factory

Table 3.2 shows the only installation issue that occurs in the factory in relation to the installation process.

Table 3.2: Installation issues in the factory

Issue	Root Cause
Import of XML files generated by the Field Service Framework (FSF) is hard to maintain in the factory because their format and even content and names may change in each development release (see PDC in glossary).	Currently interfaces are complex in the Allura System. Furthermore, interfacing with FSF is not managed at system level

3.2.4 Issues in the field

Table 3.3 shows a summary of the problems that occur in the field related to software installation. Note that the nature of problems is not only technical, but also involves other limitations such as the Field Service Engineers' (FSEs) expertise.

Table 3.3: Installation issues in the field

Issue	Root cause
1. Currently the FSE needs to introduce some settings manually more than once.	There is not a system-level configuration repository, due to lack of resources during development and other priorities within the business unit
2. Unnecessary user's waiting time causes lagging behind in the installation/update process.	The design of installation software has not considered human interaction points to make installation efficient
3. One of the biggest dissatisfiers for customers is the inaccuracy in forecast maintenance time (Field Change Order FCO-time) Currently the SW used to upgrade is not sufficiently robust. If for any reason the installation software fails, this will add to the actual time required for maintenance.	The expertise level of FSEs is limited. Although this is not an issue directly related to software, the expertise of FSEs can influence the accuracy in forecasting maintenance time, especially in troubleshooting unexpected situations.
4. Remote installation can fail. After downloading a package, there is no guarantee that the installation is successful.	The SW installation set of Allura is not infallible. Non-remote installation needs to be optimized first
5. How can we guarantee the right version is installed and running (and not the previous or an unknown version)?	Validation of software installation has not been a priority issue for the CV business unit

Issue	Root cause
6. Currently it takes two hours to make a backup (In the latest release the backup time is reduced to 30 minutes)	In the beginning people at the CV business unit did not anticipate that there would be so many updates (FCOs)
7. Currently user data is not completely migrated during an update/upgrade.	Hospital personnel are reluctant to back up images in PACS because importing them back from PACS is slow (approximately 512 GB must be transferred back)

3.3 Focal points selection

From the issues discussed in Section 3.2, three focal points have been selected whose solution match the project goal introduced in Section 3.1. We needed to make a selection due to the fact that many problems require a coordinated effort between all stakeholders involved in the installation of software, falling thus beyond the scope of FAMAR.

3.3.1 Methodology

The following two-phased method was iteratively applied to select our focal issues:

1. **Pruning of topics**:
   (a) **Organizational.** Some of the issues have an organizational nature, e.g. the (low) priority given by the CV business unit to installation of software. These type of topics were discarded to solve in this project.
   (b) **Requiring changes at a system level**. There are issues currently being tackled by joint efforts including different actors of the installation process.
   (c) **Non-relevant for the field**. We have discarded those issues that do not contribute to solving the problems in the field.
2. **Calculation of potential benefits**

We define a formula applicable to all problems discussed in Subsections 3.3.2, 3.3.3, and 3.3.4. Thus, we estimated the potential time savings gained by solving an issue (see Equation 3.1). Next, the calculation expressed in time was converted to money (see Equation 3.2).

$$Potentialtime = Minutes\,per\,occurrence * number\,of\,CV\,systems\,affected/year \tag{3.1}$$

$$Potential\,money\,savings = (potentialtime/60) * salary/hour\,of\,one\,FTE \tag{3.2}$$

Note that the figures obtained by these calculations are estimates, as no previous indicators were available. For confidentiality reasons, the actual results of using Equation 3.1 and Equation 3.2 are omitted. These results however are recorded in [13]. In the coming subsections, three focal points are chosen based upon the discussions (the pruning part) held with members of the steering committee of FAMAR, and the calculation of potential benefits (the actual selection).

3.3.2 Focal point 1: Partial installations

One of the problems mentioned in Table 3.1 concerned the need to completely uninstall the old software version and install the new one. In order to be able to perform partial installations, we should address a number of challenges previously introduced in Section 3.1

1. Generation of delta binaries. This challenge is related to item 5 in Table 3.1.
2. Installation of these deltas on systems in the field. This challenge is related to item 4 of Table 3.3.
3. Updating policy: roll up (full install) strategies v.s. delta upgrades. This issue is further discussed in Section 5.9.6.

3.3.3 Focal point 2: Validation of partial installations

After updating the software in the system there has to be evidence that the system's configuration will operate according to specifications. Therefore, the following challenges need to be addressed:

1. After a partial installation, can we guarantee the right version is installed and running and not the previous or an unknown version? This challenge is related to item 5 of Table 3.3 and Table 3.1 .

2. How to validate correct functioning of a partial installation when there are dependencies to other (older version) components?

3.3.4 Focal point 3: Recovery mechanism

Like any other process, partial installation of software is not risk-free. Should an error occur, there must be mechanisms to recover the system's configuration to an earlier known and safe state. Therefore the following questions arise:

1. Which updates (e.g. hot-fix, security update) require a recovery?
2. How can we cope this recovery mechanism with partial delta installations?

These questions are related to the items 6 and 7 presented in Table 3.3.

3.4 Summary

In this chapter we have seen the general problems related to software installation of Allura systems, and the specific issues of this process in each of the three areas where software installation is done. Given the fact that the issues found are caused not only by technical, but also by organizational and external reasons, there is a need to focus only on the aspects amenable to solve from a software design perspective. Therefore, a selection of specific problems was made in order to reach the goals of this project.

However, the result of the analysis made has served as input for further actions in order to improve the installation process at the system level. An installation working group established by one of the Allura system architects is benefiting from the results obtained.

To conclude, in Table 3.5 we can see how the selected items addressed in Section 3.3 match the high level requirements of FAMAR (discussed in Section 1), and where these issues have been identified. We use the following abbreviations in the table cells:

1. Partial installations (PI)
2. Recovery mechanism (RM)
3. Validation of partial installations (VPI)

Table 3.5: Matching between selected issues and the high level requirements of FAMAR

	Development	Factory	Field
Faster	PI. No need to install all subsystems, but only those that have changed. In this way the development/ testing cycle can be more dynamic than it is	PI. Should an important fix need be installed to the software of the host PC before deployment), a partial installation can be done	PI. The software installation part of FCOs can be automatically performed.
Managed	VPI. By automatically installing and validating partial installations, issues in development like ad-hoc installation tooling (see Table 3.1) can disappear RM. Providing a recovery method during development can save time both in the installation and the configuration phases of the installation project, discussed in Section 4.2		VPI. A validation mechanism that verifies consistent version control of partial installations, contributes (by proving that the expected version is installed and running) to satisfy the requirements posed in Subsection 5.9.1 RM. As indicated in Table 3.3, a reliable mechanism can reduce the 2 hours currently needed to backup the Allura software
Remote			PI. Partial installation packages are feasible to be remotely distributed using the infrastructure introduced in Section 1.2. Furthermore, by implementing partial installations in such a way that their result is reported to Philips, we can contribute to this remote requirement.

Chapter 4

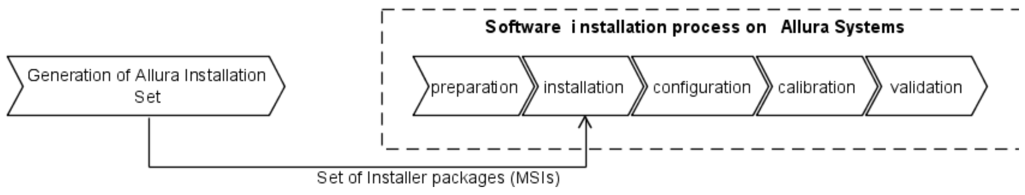
Domain Analysis

'. . . And a mighty king shall stand up, that shall rule with great dominion'

Dn 11:3

Preamble

In this Chapter, the activities related to the software installation of Allura systems are presented. Relationships between installation variants and their steps are shown.


Software i nstallation process on Allura Systems
Generation of Allura Installation Set
preparation
installation
configuration
calibration
validation
Set of Installer packages (MSIs)

In the remainder of this Chapter, we discuss the details involved in the existing installation process for Allura systems.

4.1 Functional areas where software installation is performed

As mentioned in Section 1.1 the installation of software for CV systems is carried out at three different places: development, factory and field.

In [8] there is an overview of how the installation of software for Allura systems is currently done. The functional areas within Philips Healthcare where the software installation process is performed are: development, testing, factory and field.

4.1.1 Development

There are 25 software subsystems present in an Allura system. With exception of the operating systems, the rest of application software is developed at Philips by the respective subsystem teams (e.g. image positioning, image detection). Each team is thus responsible for the development, partial testing and tooling to make its SW subsystem (un)installable.

4.1.2 Testing

Next to the development effort (which includes installation), there is another important activity where software installation is done: testing. There are two important software testing periods in the Allura applications life-cycle:

1. **System Acceptance test** (SAT), done at the end of each Allura project (i.e., a project whose outcome is a new version of the cardiovascular system). This test lasts 20 weeks, out of which 16 involve installation and configuration.

2. **Feature Acceptance test** (FAT), which lasts about 3 to 4 weeks. This type of test is done per subsystem feature. More details about integration of features in subsection 4.4.1.

4.1.3 Factory

Most of the software controlling Allura systems runs in a Host PC located in a Main cabinet (M-cabinet). Being integrated with the rest of (hardware) elements of the Allura, the software of the Host PC is installed in the QL-II factory facility in Best.

4.1.4 Field

Once deployed to the hospitals, the Allura systems are updated every six months on average. These minor software changes are delivered via FCOs, which are performed by FSEs.

Major changes in the software are deployed every three years on average, and are provided by FSEs too.

4.2 Installation phases

The installation process of software for Allura systems is rather large in terms of duration and number of steps. Therefore, this process can be divided into five important phases:

1. **Preparation**. The steps performed in the preparation phase correspond to activities prior to the installation of software in Allura. Examples of these steps are : General check of the hardware /mechanics , run Pre- SW Upgrade scripts (which check that all installation pre-requisites are met), and Delete unwanted items (i.e. delete items that are not useful to restore in case a recovery needs to be done)

2. **Installation**. This phase involves the steps required to extract the software application from Microsoft installer files (MSI files) and distribute their content into the hard drive(s) and boards of the system. Besides the application software for Allura, operating systems and stress testing applications are installed.

3. **Configuration**. Given the size and complexity of the Allura systems, there are several values that need to be configured after installing the software. Examples of steps that belong to this phase are: Configure the network settings, Configure System Interface Board, and Configure Image Detection.

4. **Calibration**: Although this phase is not as software intensive as the previous, it does involve software-hardware interaction. Examples of steps in this phase are: calibration of the stand and table, and calibration of the image detector front-end. The adjusted hardware values are stored into the system's repository for calibration data.

5. **Validation**. Once the system is installed and calibrated, governmental regulations prescribe that the system's safety should be made evident. The system must therefore pass compliance tests after every software update. For more details about compliance tests, we refer the reader to [9, 14].

4.3 Generation of the installation set

Since we are mainly interested in the installation phase of the entire installation process, it is worth describing what happens prior to this phase. In this section we discuss the preparation of the installer packages, also known as installation set (see definition in glossary).

At Philips Healthcare, Install Shield™ (IS) is used to create the installation set. The set is created as follows: first, the configuration files (i.e. files defining components and file groups) and the set of files to install each subsystem are given as input to an IS project, which is as set of scripts. Next, the install shield project is compiled. Once successfully compiled, the IS project is executed to generate the installation set, which consists of a set of *.MSI files, or a single self-extracting *.exe file if specified during the generation.

4.4 Software Configuration Management (SCM) at Cardiovascular

According to [15], software configuration management is

> "The task of tracking and controlling changes in the software. Configuration management practices include revision control and the establishment of baselines. "

In this section, we present an overview of the software configuration management for Allura systems. Given the extensiveness of SCM, we concentrate on three main aspects: base-lining strategy, versioning of files, and naming conventions.

4.4.1 Baselines

> A baseline is "a specification or product that has been formally reviewed and agreed upon, thereafter serves as basis for further development and can be changed only through formal change control procedures." [15]

At Philips Healthcare the base-lining strategy for software in Allura systems is divided into four levels (see Figure 4.1):

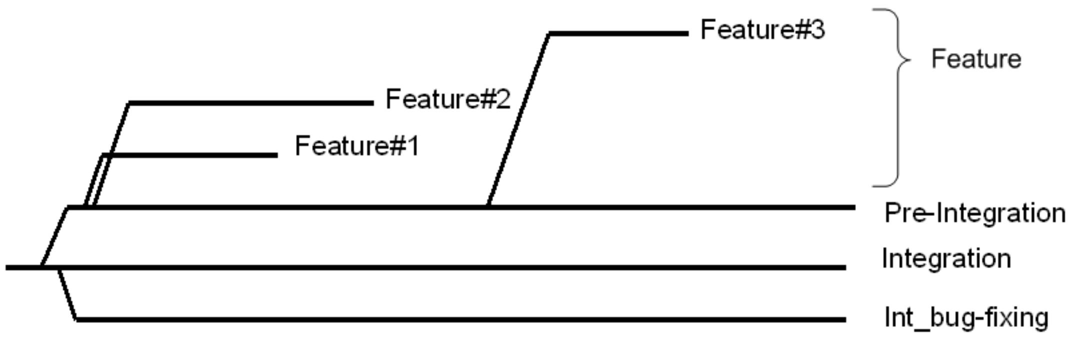


Figure 4.1: Baselining strategy of software in Allura systems

1. **Feature level**. This is the level where the development takes place. In a feature level, one or more features are developed by one or more developers.

2. **Pre-integration level**. Once a stable set of features has been established on the pre-integration stream, this set can be delivered towards the integration level for further testing. While testing this set of features (often called an increment), preparation for the next set of features on the pre-integration level can start.

3. **Integration level.** MSI files built from baselines created on this level are used for testing within a Component Development Group (CDG), and can be delivered towards the project.

4. **Integration bug-fixing level**.When problems are found during testing these deliveries, they will be solved in the Int_bug-fixing level. These fixes are delivered to the Integration level, resulting in a new delivery.

4.4.2 Versioning

The versioning of the software subsystems that integrate the installation set is the following:

The Product Domain Content (PDC) number consists of 3 numbers **x.y.z**.These numbers have to be within the following range:

X 0..255

Y 0..65536

Z 0..65536

For mainstream projects, the meanings of X, Y and Z are as follows:
X = [ab], i.e., a two-digit number where :

- [a] is a number between 1 and 24. This number refers to the Allura project.
- [b] is a number between 0 and 4 representing the increment of the project.

The number [ab] is specified in the Software Configuration Management Plan.

Y is the PDC sequence number, which starts at zero. For each major delivery, Y increases by one. These major deliveries are planned in the Master Integration Diagram (MID) and the PDC planning schedule. Y also can be increased in case parallel PDCs have to be supported (e.g. for a test at a customer site). ZZ starts at 0 when Y increases. For the minor deliveries (also known as service packs), Z will increase by 1.

Some examples:

First PDC of a new project: 70.0.0

Next major delivery: 70.1.0

Second service pack on this release 70.1.2

Second increment/major release: 71.0.0

A Product Base Line (PBL) number is similar to a PDC number, except in the following: PBL number is used for distribution of software to customers:

4.4.3 Naming

The naming convention for the installer files used in Allura systems is as follows:

PMS_subsystemIdentifier.msi

Example:

PMS_IP.msi (where IP = Image Positioning)

In the next chapter we address the requirements to meet in this project given the stakeholders, problem and domain information gathered in the last three chapters.

Chapter 5

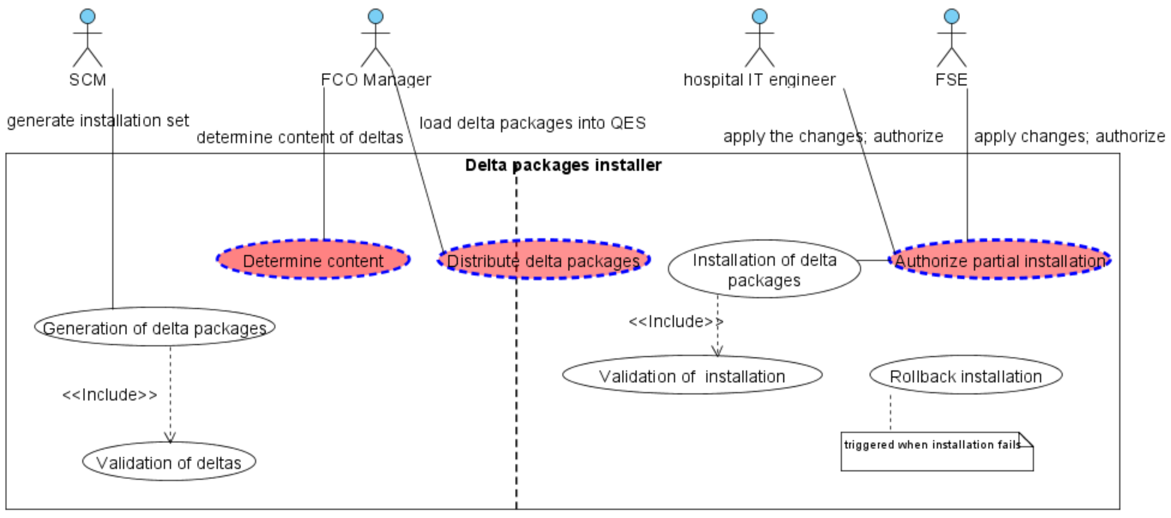
System Requirements

"Requirements are need."

Andrew Hunt & David Thomas, in "The Pragmatic Programmer: From Journeyman to Master".

Preamble

In this chapter, we present the requirements for the FAMAR project, and the method followed to obtain them.


SCM
FCO Manager
hospital IT engineer
FSE
generate installation set
determine content of deltas
load delta packages into QES
apply the changes; authorize
apply changes; authorize
Delta packages installer
Determine content
Distribute delta packages
Installation of delta packages
Authorize partial installation
Generation of delta packages
<<Include>>
Validation of deltas
<<Include>>
Validation of installation
Rollback installation
triggered when installation fails

After having discussed in Chapter 3 the problems to be solved in FAMAR, and introduced the details of the software installation process of Allura systems in Chapter 4, now we present requirements of this project.

The rest of this chapter is organized as follows: Section 5.1 delimits the scope of the requirements elicitation. Section 5.2 presents an overview of the use cases that we use as reference frame to this chapter. Sections 5.3 to 5.7 describe the realization of the use cases. Section 5.8 expresses common constraints within this project. Section 5.9 addresses non-functional requirements to be met, and finally Section 5.10 concludes the chapter with a summary.

5.1 Scope

When eliciting requirements for this project, the following considerations must be made:

1. This project focuses on updating software on systems in the field. Other systems (e.g. test systems) are not considered. From Subsection 3.3.1, we can see that this project is focused (and based) on the needs of the field service area.

2. The generation, installation and validation of partial installations will address packages to be installed on the host PC only. This decision is based on two reasons:

   (a) The host PC runs most of the Allura software (see Section 1.1), and

   (b) The updating mechanism of embedded software is a topic that deserves attention from other stakeholders involved in the overall installation process.

3. Automatic installation of delta packages is limited to software updates where

   (a) No configuration is required. Rationale: Configuration is a specialized activity that requires the presence of an FSE. (See Section 3.2.4)

   (b) No data migration is needed. Rationale: Data migration requires human validation which must be made by an FSE. Furthermore, automatic validation of data migration is technically difficult to guarantee in automatic partial installations.

4. The target software release upon which the requirements (and eventual solution) are based, is PBL 71. The reason to choose this PBL is that it is a release to which most systems in the field will be upgraded. This upgrading began in March 2009.

5. In order to distribute the delta packages to the field systems, it is assumed there is infrastructure for that purpose(see Section 3.2). Limitations of the infrastructure (e.g. limited bandwidth or broken connections) should be considered.

5.1.1 Methodology

In order to gather requirements for this project, we proceeded in the following way:

1. **Define use cases**. By defining use cases, we can set the intended use scenarios of a solution that addresses the challenges posed in Chapter 3 (Problem Analysis).

2. **Use use cases as reference to distillate requirements**. During sessions held with stakeholders of this project, use cases are used to *inspire* stakeholders in defining what is required to achieve partial installations.

5.2 Use case overview

Figure 5.1 shows the high level use cases for the FAMAR project. The project should deliver tools to perform the following

1. Generation of delta packages

2. Validation of deltas

3. Installation of delta packages on the target CV system

4. Validation of the installation of packages

5. Rollback the installation. Backup and restore the previous configuration when the installation fails.

In the coming sections the use cases above are described. The use cases with dashed lines and red background[1] are not discussed in this document but rather used to give context to the project. Use case "determine content" is mentioned in [16]; use cases "distribute delta packages" and "authorize partial installation" are discussed in [5]. The rest of use cases are further described in the coming sections

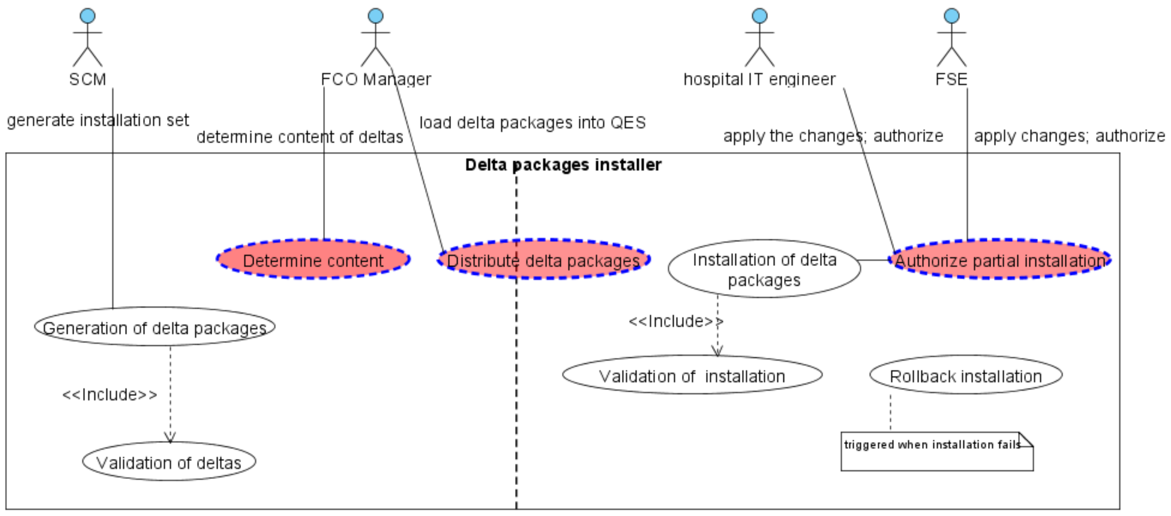

SCM
FCO Manager
hospital IT engineer
FSE
generate installation set
determine content of deltas
load delta packages into QES
apply the changes; authorize
apply changes; authorize
Delta packages installer
Determine content
Distribute delta packages
Installation of delta packages
Authorize partial installation
Generation of delta packages
<<Include>>
Validation of deltas
<<Include>>
Validation of installation
Rollback installation
triggered when installation fails

Figure 5.1: Reference use cases

[1]Or grey in grey-scale printouts

In the coming sections, we use the following format to explain the realization of the use cases above shown:

Assumptions
Assumptions made and conditions to be met before the use case is realized
Inputs
The inputs to the (alternative) flows of the use case
Outputs
The expected result produced after the use case has been realized

Furthermore, each use case is realized through a main flow, and a set of alternative flows when applicable. Next to the flows, a refinement in terms of functional requirements is described.

5.3 Generate delta packages

The SCM manager generates binary deltas of the Allura install set. This delta set is generated Allura system-wide, like the installation set discussed in [2]. The delta install set is meant to be distributed as an update package via the Remote Service Network (RSN) mentioned in [5].

5.3.1 Use case overview

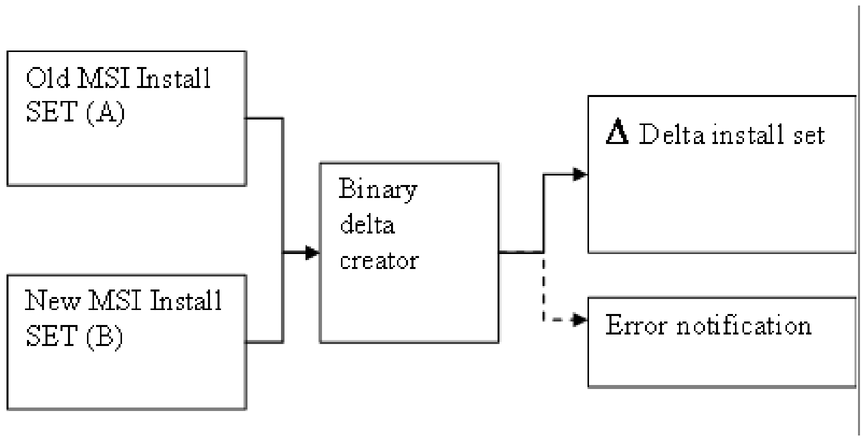


Figure 5.2: Binary delta generation

Assumptions
There are two installation sets available: A(version X) and B (version X + n)

Inputs

Installation sets A and B, as in Figure 5.2

Outputs

A set of files that contains the (code) changes of installation set B with regards to A.

5.3.2 Main Flow

1. SCM generates a set of files for each of the MSI file pairs that belong to the installation sets A and B (e.g. PDC 71.18.10 and PDC 71.19.0)
2. Upon successful delta generation, the SCM is shown a summary with the number of delta files generated and the time spent.

5.3.3 Alternative Flows

Alternative flow A

During the generation of deltas, an error occurred.
1. The user is reported where and when the error occurred.
2. Errors are corrected by the software configuration manager(s), and the delta package set is generated again

5.4 Validation of deltas

Validation of the content of delta packages before deployment to the field. This validation is done at system-level.

Assumptions

There are available two installation sets: A(version X) and B (version X + n) (see Figure 5.2) plus the delta install set generated in use case 1 - “Generation of delta packages”

Inputs

A delta Install set

Outputs

A binary value indicating the success or failure of the validation

5.4.1 Main flow

1. Use case “Generate delta packages ” triggers the validation of content of the delta packages
2. The result of the validation is communicated to SCM.

5.5 Install delta packages

Automatic installation of the update package into the Allura system. The installation is done via Questra software distribution mentioned in [5].

5.5.1 Use case overview

Assumptions

1. Installation is performed during non-clinical use of the CV systems.
2. Required expected version is installed in the target system
3. Before installing the downloaded packages are verified not to be corrupted

Inputs

The update package

Outputs

A newer version (of some components) of Allura SW is installed on the target CV system

5.5.2 Main Flow(s)

1. An installation procedure is invoked by the user to install the update package. More details in Section 4.5 of [5]
2. The installation procedure makes a system backup. This backup will be used in use case 5.7-“Rollback delta installation”
3. The content of the updated package is installed in the system
4. After installation the system validates the installation as specified in section 5.6 –“Validate installation of delta packages”.
5. The user is informed about the result of the installation

5.5.3 Alternative Flows

Alternative flow A

During the installation of delta packages, use case ”Rollback delta installation” will be executed:
1) When an error occurs, or
2) When a cancellation (in the case of non-silent installations) occurs. Cancellation should be triggered by the user (either FSE or hospital IT engineer) .

5.6 Validate installation of delta packages

Validate the installation of delta packages at system level

5.6.1 Use case overview

Assumptions

The validation is performed during non-clinical use of the CV system.

Inputs

All items that make up the Allura SW installation. For a detailed list of items, please refer to Chapter 7 -Design Specification

Output

A binary value indicating success or failure of the validation.

5.6.2 Main Flow

1. The installation procedure introduced in section 5.5verifies that the version of the installed update package is the expected version as specified in the BOM.ini file for delta packages.
2. Verify the Allura installable items discussed in [FDS_PI]
3. Return result of validation to the flow of use case 5.5 -"Install delta packages". The result is binary: Passed or Failed

5.6.3 Alternative Flows

Alternative flow A

An error during validation occurred.
1. An error code is generated, and
2. Use case "Rollback delta installation" will be executed

5.7 Rollback delta installation

Restore the system to the point set before installing the update package

5.7.1 Use case overview

Assumptions

1. A backup of the Allura system is available

Inputs

Installation error message/event

Outputs

The system is restored to the previous software configuration

5.7.2 Main Flow(s)

1. The restore procedure is invoked by use case 5.5 -"Install Delta packages".
2. The user is informed of the success of the restore procedure.

5.7.3 Alternative Flows

Alternative Flow A. Rollback failed

1. The rollback procedure returns a value different from 0 (SUCCESS).
2. The user is informed, and requested to contact the FSE. More details are in [5]

5.8 System Constraints

5.8.1 Operating system platform

The execution environment shall be the Windows XP operating system.

5.8.2 Bandwidth and connection capacity of the RSN

The Remote Service Network (RSN) is the infrastructure that allows remote software distribution. The RSN's bandwidth imposes a size constraint on the packages generated in use case 5.3 -"Generate delta packages ". Packages must be small in size, i.e., smaller than 50 MB.

5.9 Non Functional Requirements

The following list of non functional requirements is considered in the context of the FAMAR project. The list is in order of priority.

5.9.1 Reliability

The installation of upgrades has to be such that Philips can build up evidence that every system configuration in the field operates according to specifications. Validation must adhere to FDA regulations (see Appendix B -Applicable FDA regulations).

As part of distribution, the packages downloaded to the Allura system shall be validated.

5.9.2 Testability

The implemented solution shall be testable in Philips Healthcare testing environments. A test plan shall be designed and executed that fully verifies the tools created as part of the FAMAR project.

5.9.3 Serviceability

After a successful installation, provide the user with release notes in customer language. Satisfying this requirement is responsibility of the Questra software included in the Allura Service Framework.

The current service time required for software updates in the field shall decrease. A test report shall be provided that shows the reduction in time.

5.9.4 Maintainability

The developed solution shall be implemented in scripting language for use cases described in sections 5.3, 5.5, 5.6, and 5.7.

The developed solution shall be designed and implemented according to the Philips coding standards to enable further maintenance.

5.9.5 Performance in the target system

As ideal target the overall installation time of delta packages should not exceed 10 minutes, including pre/post validation and reboot. An acceptable time reduction shall be between 50% and 90% of the current MSI installation time. The actual installation time will depend upon the number of changes between the old and new releases, and pre/post installation steps required.

5.9.6 Update policy

The designed solution shall adhere to any update policy (e.g. Full roll-up vs. delta installation, Incremental vs. cumulative update).

5.10 Summary

This chapter discusses the high level user scenarios that the delta packages installer should perform. These use cases are used as a reference frame to which system requirements are attached.

Finally, Table 5.1 provides a summary of the requirements to be met in this project. The priorities[2] shown were assigned by project stakeholders (introduced in Chapter 2) during two requirements elicitation sessions discussed in Subsection 5.1.1.

[2]H= high, M=medium, L=low

Table 5.1: Priority of requirements

Section	Description	priority
5.3	Generation of delta packages at system level	H
5.4	Distributed (and thus downloaded) packages must be verified	L
5.5.1	In case of semi-automated installation, everybody should be able to install.	L
5.5.1	Installation should be schedulable	L
5.5.3	Cancellation of the installation should be possible in case of emergency	H
5.6	(Automatic) validation of the partial installation	H
5.8.2	Size of distributed delta packages needs to be small.	H
5.9.1	Validation must adhere to FDA legal rules: Changing SW is an “essential modification” to the system	H
5.9.5	Installation of deltas shall reduce time installation with regards to the existing process between 50% and 90%	H

Chapter 6

Feasibility Analysis

"The question of feasibility, the question of cost ..., the question of the effect of this project ... all these effects are in discussion."

Gerhard Schroder, cited from [17].

Preamble

In this chapter, we present a feasibility study related to the FAMAR project. Basic aspects to evaluate are

- The evaluation of component-off-the-shelf (COTS) tools that enable us to generate delta packages
- The evaluation of installing delta packages created from the current Allura MSI installation set.
- The evaluation of COTS tools for validation of installed delta packages

In Chapter 5 we identify the requirements for remote partial installations on Allura systems. In this chapter, we explain in Section 6.1 how we choose the tool that generates delta packages, and document the installation of these packages to an Allura system (Section 6.2). In Section 6.2 we present preliminary results of installing delta packages, and finally in Section 6.3 a discussion on tools to verify the partial installation process is presented.

6.1 Feasibility analysis of delta generators

An important element of the development process of this project was to select the tool to use in order to generate the delta packages (see Section 5.3).

6.1.1 Methodology

In order to select the most suitable tool to generate deltas, a feasibility analysis was carried out using the following methodology:

1. Select a number of Commercial Off-The-Shelf (COTS) applications that generate delta packages.
2. Create delta packages and measure
   (a) Time required to generate the package
   (b) Size of the package
3. Report on the results and make a selection.

6.1.2 Tool requirements

The first step of the method described above was done by looking into different tools that satisfy the following minimum requirements -in order of importance:

1. The tool should possible to automate -i.e., scriptable (see constraint in Subsection 5.9.4).
2. The size of the patches should be small (see restriction in Subsection 5.8.2)
3. The delta package should be generated taking as input two MSI files (we are required to generate delta packages taking as input two different install sets, as explained in 5.3.1).

6.1.3 Selection of tools

From [18] we obtained a list of COTS applications that generate delta packages. The following applications were selected as most promising to be used based on their advertised features:

1. Install Shield 2009 (enterprise edition)
2. Virtual Patch 3.0
3. Advanced Installer 7.2

4. Active Patch
5. Windows Installer SDK 4.5
6. Centurion Setup
7. SSE Setup

However, after filtering the above listed applications using the requirements enunciated in Section 6.1.2, the list was reduced to lack of desired functionality.

1. Visual Patch 3.0
2. Advanced Installer
3. Windows Installer SDK 4.5

The other tools were discarded due to, for instance, the fact that InstallShield does not accept as input MSIs to generate patches, but needs manual indication of the files to be updated. The same limitation holds for Active Patch and Centurion Setup. In the case of SSE Setup, it does not provide a command line interface to automate the delta packages generation.

6.1.4 Performance of tools

From the tools listed above, we generated delta packages using Allura's UI (MSI filesize : 201 MB) and Acquisition (MSI size 67 MB) subsystems.

In Tables 6.1 and 6.2 we can observe the results produced by the chosen tools in terms of file size and time required to generate it.

Table 6.1: Results of generating patches for the UI subsystem

File Name	time required (minutes)	file size (MB)
Virtual patch 3.0	5	174
Advanced installer 7.2	62	18.2
Windows Installer SDK 4.5	59	31.2

Virtual Patch was discarded for a second test given the enormous size difference in the output produced: An important requirement is that delta packages are small so that they can be distributed over a Remote Service Network (Subsection 5.8.2). Delta packages should not be bigger than 50 MB. Therefore, Virtual Patch's output of 174 MB is not acceptable.

Table 6.2: Results of generating patches for the Acquisition subsystem

File Name	time required (seconds)	file size (MB)
Advanced installer 7.2	20	12.4
Windows Installer SDK 4.5	15	12.7

From the results obtained, we can see that Advanced Installer has the best performance. However, Advanced Installer has a limited command line interface, i.e., it does not have yet the functionality to generate delta packages from the command line.

In conclusion, we choose to generate delta packages using the Windows Installer SDK itself, as the results are comparable to Advanced Installer's and thus we can produce deltas small enough to distribute them via the Remote Service Network.

6.2 Feasibility of installation of deltas

In the previous section we evaluated the tooling to generate delta packages. The next step is to perform tests on the installation of such packages. To this end, a number of installations were made.

We installed the delta packages produced with the tools above mentioned. From this installation experience there are a number of valuable lessons learned:

1. **Version number is just a number**. When updating software using the *msiexec* command provided by Windows Installer, **the REINSTALLMODE property [3] determines how the system should be updated**. We require a post-validation process to determine whether the files are indeed updated and the registry entries are modified accordingly.

2. **There are important properties in the MSI files that must be met in order to generate delta packages**. The following is an excerpt from [3] that enunciates these properties (see "Creating a Patch Package"):

   "*When you author a patch package you must use an uncompressed setup image to create a patch, for example, an administrative image or an uncompressed setup image from a CD-ROM. You must also adhere to the following restrictions:*

- *Do not move files from one folder to another.*
- *Do not move files from one cabinet to another.*
- *Do not change the order of files in a cabinet.*
- *Do not change the sequence number of existing files. The sequence number is the value specified in the Sequence column of the File Table.*
- *Any new files that are added by the patch must be placed at the end of the existing file sequence. The sequence number of any new file in the upgraded image must be greater than the largest sequence number of existing files in the target image.*
- *Do not add new files to the end of an existing cabinet file. All new files must be added after the last cabinet file in the sequence.*
- *Do not change the primary keys in the File Table between the original and new .MSI file versions.*

Note The file must have the same key in the File Table of both the target image and the updated image. The string values in the File column of both tables must be identical, including the case.

Do not author a package with File Table keys that differ only in case, for example, avoid the following table example

File	*Component*	*FileName*
readme.txt	*Comp1*	*readme.txt*
ReadMe.txt	*Comp2*	*readme.txt*

6.3 Feasibility analysis of software validation tools

Part of the analysis in this project is about the feasibility of tools to validate installation of delta packages. Given the different aspects of the installation that can be verified, we list below the main cases:

- **validation of files**. Allura application files.
- **validation of registry settings**. Allura configuration settings recorded into the Windows registry.
- **validation of migrated user/system data**. In this category are configuration files and databases.

Check-sum functions are chosen as means to verify the correctness in files updated by delta packages. In the coming subsections we address two of the most widely used check-sum functions.

6.3.1 Validation using CRC

Cyclic Redundancy Check (CRC) is a hash function "*designed to detect accidental changes to raw computer data*" [19]. CRC allows us then to use it as a validation method for files installed or updated.Figure 6.2 shows the performance result of a number of CRC tools selected in this feasibility analysis. The tools were used to generate the CRC values of files installed by the User Interface subsystem of Allura - which has an install base of 279 files in PBL release 7.0.0.

6.3.2 Validation using MD5

The Message-Digest algorithm 5 (MD5) operates on four words of 32 bits each one, where each word is processed in four different rounds. Each of these rounds consists of 16 similar operations based on a non-linear function F, modular addition and left rotation. Figure 6.1 shows one of these MD5 operations and Figure 6.3 shows the performance of the tools evaluated using the same input as in section 6.3.1.

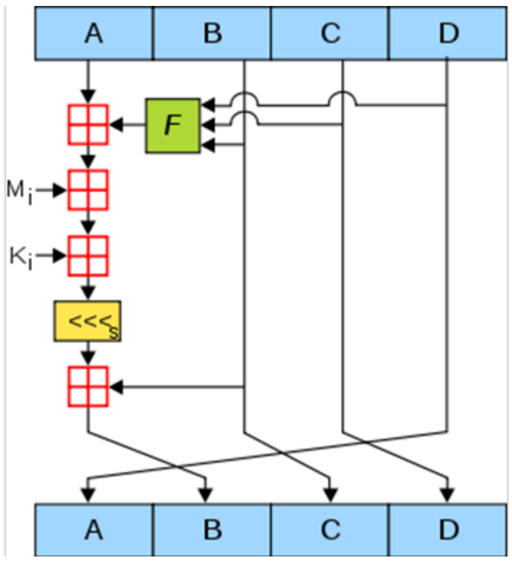


Figure 6.1: MD5 operation (from [1])

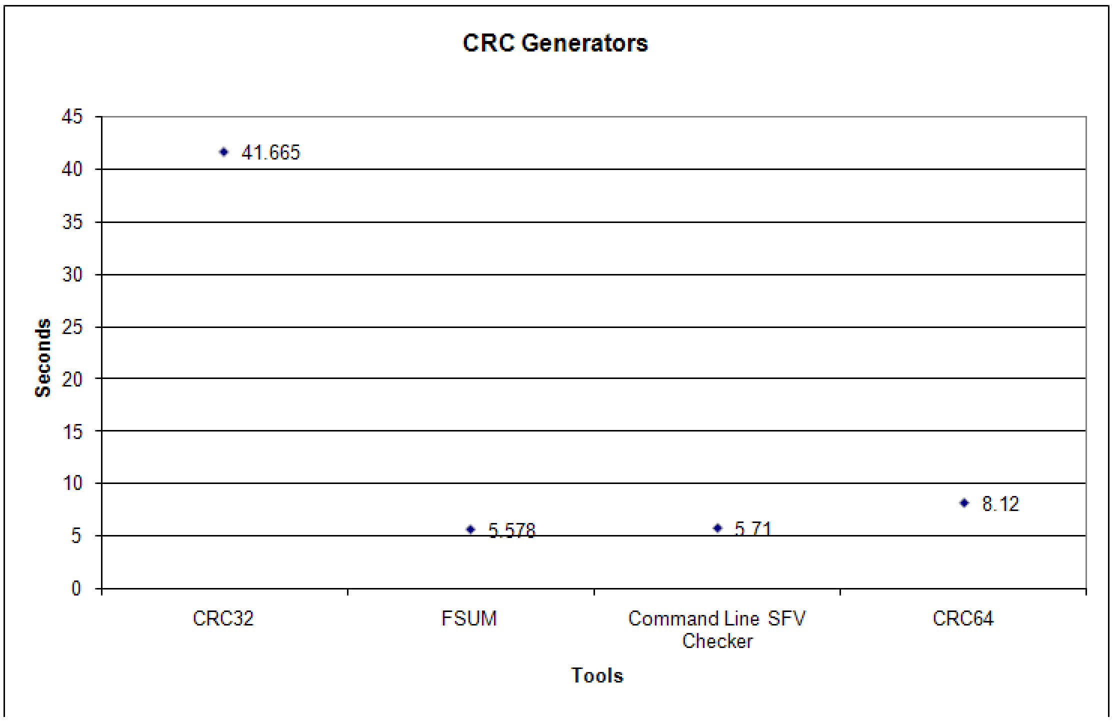


Figure 6.2: Performance of CRC tools

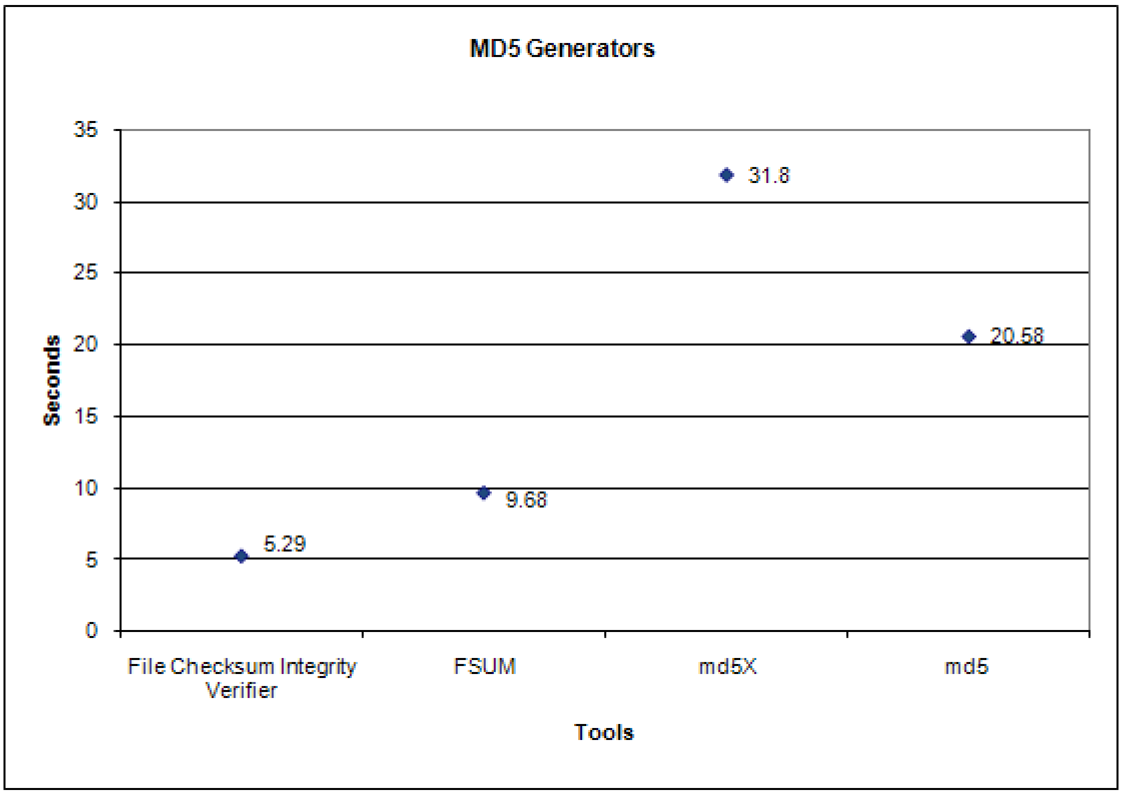


Figure 6.3: Performance of MD5 tools

In this section we analyze the performance of a number of check-sum computation utilities. These utilities allow us to determine whether the right files are in place before and after a partial installation. A MD5 check-sum utility was chosen over a CRC alternative due to two main reasons:

- CRC has a higher possibility of collisions than MD5. This is arguable since the bigger the polynomial used, the less likely collisions will happen. Moreover,

- MD5 is to be used as "*standard checksum function in monitoring tasks in the Allura [system]*", cite from Ben Kokx, Privacy and Product Security Officer -11/July/2009. N.B. This constraint was given **after** this feasibility analysis was made.

6.4 Summary

In this chapter three main elements are analyzed in order to proceed with the rest of the project. First we addressed in Section 6.1 a number of *components-off-the-shelf-tools (*COTS*)* that allow us to create Microsoft patches. Next, in Section 6.2 we have determined whether Microsoft patch files do update an application as expected. Lastly, in Section 6.3 a number of COTS are evaluated for computation and verification of CRC and MD5 checksum values.

In the next chapter we address the design based on the results produced in this chapter.

Chapter 7

Design Specification

"Every design begins with somebody's unhappiness"

K. v. Overveld, coach on creativity and modeling techniques at the OOTI program.

Preamble

In this chapter the blueprint that guides the implementation of partial installations for Allura systems is presented. The content of this chapter is: visual representation and design decisions of the processes to generate, install, validate and rollback delta packages.

7.1 Introduction

Figure 1.2 on page 7 gives an overview of the current installation process of Allura SW at Philips Healthcare. In this project, we introduce a new deployment method with the aim of satisfying the requirements discussed in Chapter 5. By introducing this new method, we extend steps 3 and 12 in the figure mentioned before. In Figure 7.1 we can observe the changes introduced:

1. Taking as input the MSI install sets produced by System Configuration Management (SCM), we generate a partial install set, i.e., a set of MSP files.

2. The installed content coming from the MSI install set is taken to compute MD5 hashes (introduced in Section 6.3) to use these hash values as validation means of the partial installation. These MD5 values are stored in an encoded XML format. XML files together with the MSP files form an update package to be distributed via the Remote Service Network introduced in Section 5.8.2.

3. Before any modification to the system, a restore point is created, i.e., a system backup. This backup will be needed in case something goes wrong during installation.

4. After reception and validation of the update package at the customer site, before installing the delta packages we need to ensure that the right system configuration is installed. Otherwise, there is no need to proceed with the installation. This validation must be done for the entire system at file level.

5. Provided the expected system configuration is present, we proceed to install all delta packages.

6. After successfully installing all the delta packages, we verify that indeed we have the desired system configuration. This validation is also at file level.

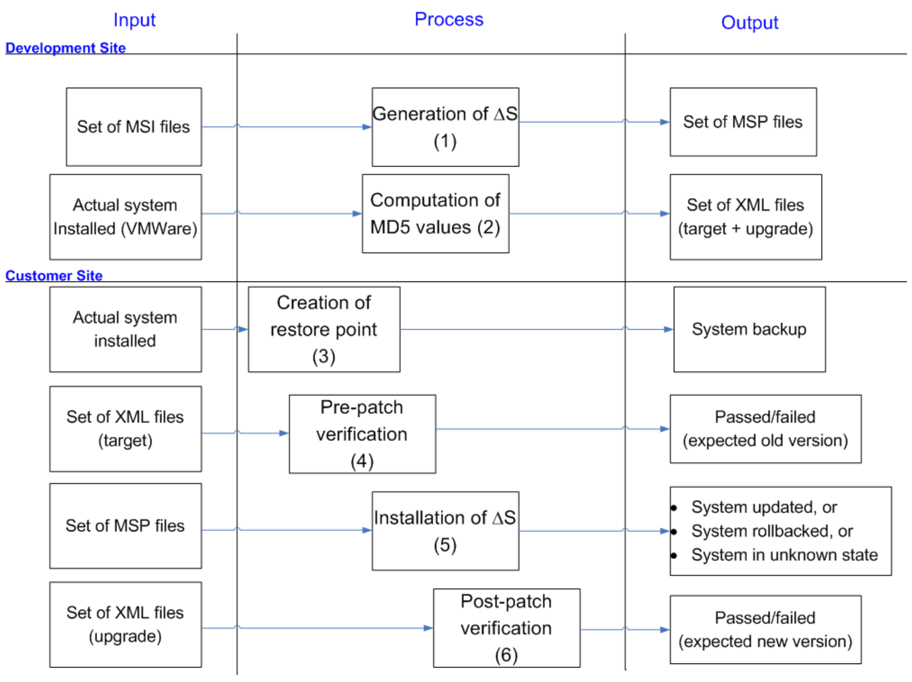

Input
Process
Output
Development Site
Set of MSI files
Generation of ΔS (1)
Set of MSP files
Actual system Installed (VMWare)
Computation of MD5 values (2)
Set of XML files (target + upgrade)
Customer Site
Actual system installed
Creation of restore point (3)
System backup
Set of XML files (target)
Pre-patch verification (4)
Passed/failed (expected old version)
Set of MSP files
Installation of ΔS (5)
System updated, or
System rollbacked, or
System in unknown state
Set of XML files (upgrade)
Post-patch verification (6)
Passed/failed (expected new version)

Figure 7.1: Process view of partial installations of Allura software

In the coming sections, each use case presented in Chapter 5 corresponds with a design realizing that use case. In this chapter the following format is applied to explain the design:

1. **Preconditions**. Assumptions made during this design, which need to be verified to proceed with the actual implementation.

2. **Description of the workflow**. A description of diagrams or any other graphical notation explaining the (workflow) design.

3. **Postconditions**. The result of the workflow.

4. **What-if scenarios**. Next to the alternative flows described in Chapter 5, this Section describes some of the contingencies that could appear (based on the assumptions made) while implementing the design, and gives some suggestions on how to proceed in such situations.

5. **Design alternatives.** In this subsection the advantages and disadvantages of other explored solutions are discussed.

For implementation prerequisites, please refer to Section 8.1 on page 74.

IMPORTANT. The rest of the chapter is a condensed version of [20].

7.2 Generation of delta packages

One of the first steps in the installation process occurs at SCM, where the different subsystems are assembled into the full installation set. With the introduction of partial installations, we need to use as input the MSI packages created in SCM.

7.2.1 Preconditions to implement the workflow

The following preconditions must be met in order to implement the design described in the next section:

1. The installation sets of the target and upgrade installation sets are in two different directories. This is the way the current Allura releases are published.

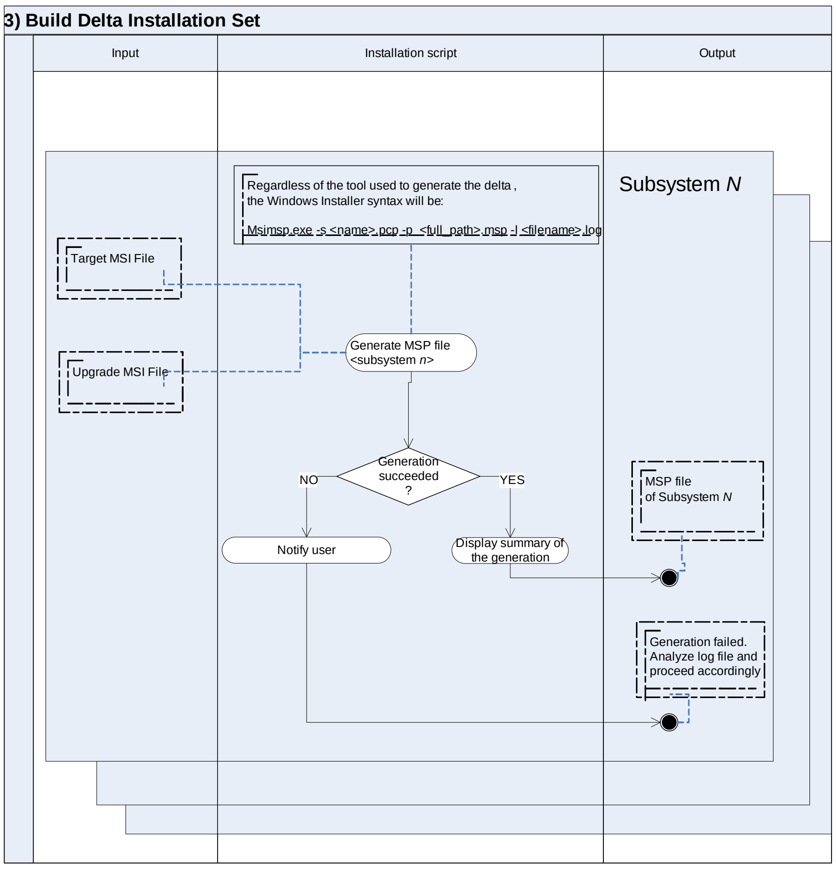
2. The names of the subsystem MSIs are the same from one release to another

3. It is possible to automatically fill-in values of the Patch Creation Properties File (PCP) file

4. There is sufficient storage space for two uncompressed install sets (Target and Upgrade) plus the space of the MSP files generated. An estimation of the space required for PBL 7.0.0 and its Service Pack 1 (i.e., PBL 7.0.1) is 10 GB. To this space must the added the payload contained into the delta packages, which for the before mentioned PBLs requires 100 MB

7.2.2 Description of the workflow

Figure 7.2 displays the workflow of the delta packages generation at system level. This figure shows the high-level design, and additional information related to this workflow can be found in Appendix A. "Patching Memo".


3) Build Delta Installation Set
Input
Installation script
Output
Subsystem *N*
Target MSI File
Upgrade MSI File
Generate MSP file <subsystem *n*>
Generation succeeded ?
NO
YES
Notify user
Display summary of the generation
MSP file of Subsystem *N*
Generation failed. Analyze log file and proceed accordingly

Regardless of the tool used to generate the delta , the Windows Installer syntax will be:

```
Msimsp.exe -s <name>.pcp -p  <full_path>.msp -l <filename>.log
```

Figure 7.2: Generation of delta packages for Allura software

Provided there are several computer resources available, the generation of delta packages for all subsystems can be parallelized. For every subsystem, an MSP file is created, which can be generated independently of packages from other subsystems. However, in order to succeed in all the test cases executed in Chapter 9, we need to create major and minor upgrade MSP files, depending upon the installation behavior needed (see Section 7.4.5) and the impact to the current installation process. Table 7.1 indicates the main advantages and disadvantages of both major and minor upgrade MSP files.

Table 7.1: Advantages and disadvantages of minor vs. major upgrades in the context of Allura software

Patch type	Plus	Minus	Interesting
Major upgrade MSPs	No need to change the product code. This change impacts the current MSI build process	Patches are not uninstallable, i.e. the complete product has to be removed.	* Changes are required to the current MSI build process: i.e., A). to have (installed) MSIs with only three-field version numbers, B). To include conditions to execute custom actions when updating via MSP files, and C). No nested MSIs * Higher versioned files are replaced by lower versioned ones, because the *amus* switch is the only one that guarantees file update regardless of file versioning. This guarantee is regardless of the patch type.
Minor upgrade patches	Adheres to Windows Installer's best practices on using patches: i.e., MSPs are meant for service packs and small updates.	Adds extra complexity in the MSI build process: i.e., RemoveFile and RemoveRegistry tables, in cases where the upgrade release contains less files than the target release	Same as with major upgrade patches

MSP files are created in the following process:

7.2.2.1 Create a PCP file

As shown in Figure 7.3, a PCP file is created and provided as input to the *msimsp.exe* utility included in Windows Installer's SDK. Inside the PCP file, tables shown in Figure 7.4 are filled in. In this subsection we highlight only some of the columns / properties inside these tables -the extensive description of each property is discussed in Subsection 3.1.2 of [20], and an example is given in "Creating a Patch Creation Properties File" of [3].

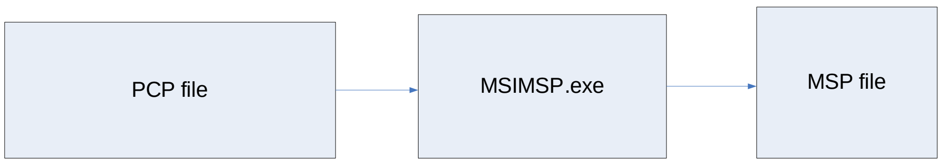


Figure 7.3: Creation of an MSP file

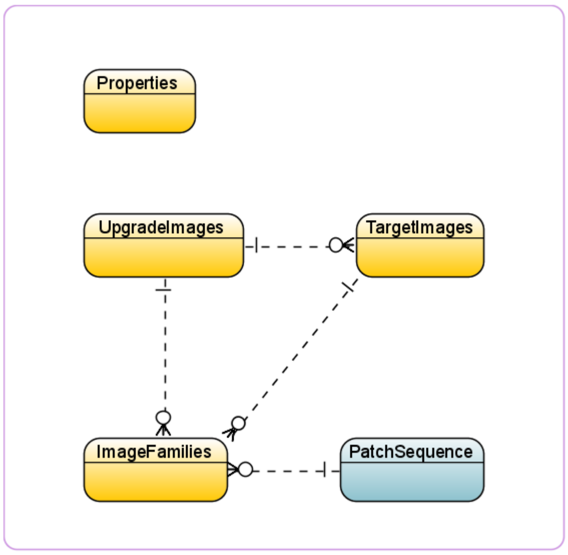


Figure 7.4: Tables of a PCP file

1. **Properties table**. This is a required table containing global settings for the patch package. Important properties used in this table are: a)A Global Unique Identifier (GUID) value for the MSP file, b)The output path of the patch, c) a binary flag to generate either major or minor upgrade patches, and d) another flag to obtain the smallest patch size possible. In order to meet the requirements posed in Section 5.8.2, this is a very important property to use.

2. **TargetImages table**. In this table we need to indicate the path of the target MSI, i.e., the version that will be patched. Important values filled in this table are a) The full path (including file name) to the location of the target MSI file, i.e. the version we expect to be installed before updating the system, and b) a flag to indicate the application order of patches. In this project we apply patches incrementally rather than cumulatively, since we want to obtain the smallest patch size possible.

3. **UpgradeImages table**. In this table we need to indicate the path (file name included) of the upgrade MSI file, i.e., the newer version to which we want to upgrade.

4. **ImageFamilies table**. Patches are grouped into families for administrative purposes when deploying and applying patches. Like a human family, the ImageFamilies table contains a family name, with which MSP files are grouped. This is a compulsory column to include, and serves to map records from the **TargetImages**, **UpgradeImages**, and **PatchSequence** tables.

5. **PatchSequence table.** In the case of minor upgrade MSP files, this table contains information about the application sequence of the patch. We have chosen to use the version number of the upgraded MSI to fill in this table, because with the upgrade version number we control the patch application sequence.

Once we have filled in the PCP file, we proceed to call the *msimsp.exe* utility. This utility can generate a log file, and hence it must be called as follows:

```
Msimsp.exe -s <name>.pcp -p <full_path>.MSP -l <filename>.log
```

For the .MSP file name, the following naming convention is defined:

```
Filename := <subsystemName> + '_' + <TargetVersionNumber> + '_' +
            <UpgradedVersionNumber> + '.msp'
```

Example:

```
PMS_UI_71.3.10_71.4.19.msp
```

Since we log the result of the MSP file generation, we can evaluate success of the process. After executed the workflow described above for all the subsystems, the log files is parsed.

1. In case of a successful generation of all the MSP files, a summary of the generated files is displayed.
2. In case of errors during the process, a notification is displayed. A format of the notification is shown in Table 7.2.

Table 7.2: Summary of failure in generation of delta packages.

The generation of delta packages was not completely successful:		
Subsystem	Cause of failure	Time
PMS_Acq_PDC_76.1.8	ERROR: UpgradedImages.Upgraded = 'fixed': ProductVersion is not valid.	13:08:25

7.2.3 Post-conditions of the workflow

At the end of the workflow, the successful end-result is a set of MSP files, along with a summary of the generation process. In case that the generation of any subsystem fails, this event is recorded in the corresponding log file. Furthermore, after successful generation of the MSP installation set, the next step is the validation of this installation set before distribution to the field. This validation is explained in the next section.

7.2.4 What-if scenarios

1. In case it is technically not possible to parallelize the generation, then make it sequential. This alternative is acceptable provided the time required for generation is in the order of minutes (and not hours nor days). However the actual duration is dependent upon the number of changes between the target and upgrade releases.
2. If is technically possible to parallelize, but generation exhausts memory , CPU and/or other resources, then make generation semi-parallelizable, e.g. taking 5 subsystems at a time to generate them in parallel.

7.2.5 Design alternatives

The options to generate delta packages are discussed earlier in Section 6.1 of this report. Basically we choose to use the *Msimsp.exe* utility over other alternatives because it satisfies the requirement that generation of deltas should be automated at system-wide level (see section 5.9.4 on page 39).

7.3 Validation of delta packages

The validation of delta packages before their deployment to the field consists of:

1. **The computation of MD5 hashes.** These hashes will be used during the installation at the customer site, as mentioned in section 7.4.

2. **The installation of the delta packages set before distribution**. The installation of delta packages in place of full install packages. This installation represents the current testing activities and hence is outside the scope of this project. Nonetheless, the outcome of the partial installation process is verified in Chapter 9. The validation is made assuming that the test systems reflect the conditions of the systems in the field.

7.3.1 Preconditions

1. Since we compute the MD5 values from an installed system, we assume the installation is sucessful.

2. All Allura services and applications running are stopped, otherwise the *fciv* utility will issue an error when computing the MD5 values.

7.3.2 Description of the workflow

Computation of hash values. Part of the delta generation process introduced in Section 5.3 involves the computation of delta hashes, i.e. the MD5 values of:

- Items installed in the target Install set
- Items installed in the upgrade Install set, and
- MSP files to be distributed to systems in the field

The computation of MD5 hashes is taken from an actual installed system, as depicted in Figure 7.1. MD5 computation is done using the *fciv* utility assessed in Section 6.3. After full installation in a test system and once that files are unrebased –see section 7.5.2, *fciv* takes as input the directories where the Allura software is installed, and computes the MD5 values of each file inside these directories –Figure 7.5. The outcome is a XML file. *Fciv* generates also a log file.

The main advantages and disadvantages of this approach are in Table 7.3.

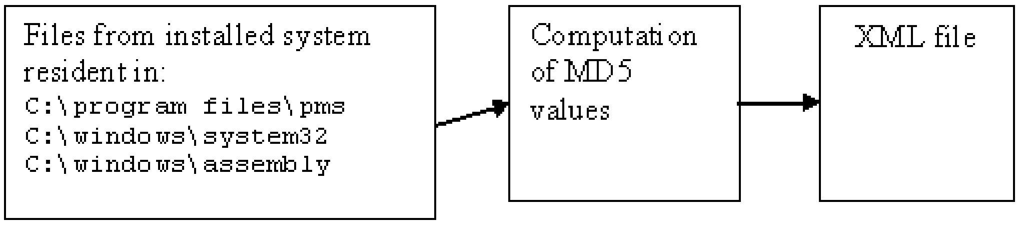


Figure 7.5: Computation of MD5 values from an actual system installation

Table 7.3: Pros and Cons of computing MD5 values from an actual installed system

Advantages	Disadvantages
• Removes the problem of installed items whose MD5 value does not match, e.g. due to effect of post-installation tools • To compute hashes from the installation directories is relatively fast: ~18 minutes.	• Pre-patch validation in the target system takes ~15 minutes. This is due to extra files that are selected during the computation of the hashes.

7.3.3 Post-conditions

Upon successful computation of the MD5 values, an XML file is obtained.

7.3.4 What-If scenarios

In case something goes wrong during the computation (e.g., a file locked), the log generated by *fciv* will indicate the reason why computation did not succeed.

7.3.5 Design Alternatives

There are two main options that were explored to realize validation of delta packages

- **ICEs.** Internal Consistency Evaluators (See Glossary on page 1) allow examination of records inside MSI tables, to determine whether these records contain information that may cause incorrect behavior during installation. We explored the possibility of using these ICEs to validate MSP files. However, ICEs must be run using a tool which only regards MSI files as databases. Therefore this option was discarded.

- **File attributes**. File attributes such as file size, time stamp and version are common for most installed files. Nonetheless, in Allura software there are a number of files that have not updated version although their content changed (e.g. DLLs and some executables). Besides, the time-stamp of files belongs to the building process rather than to the file creation/modification. These constraints led us to look for other means to verify the content of delta packages before distributing them.

7.4 Installation of delta packages

Figure 7.6 depicts the installation of (delta) packages in the Allura system. The installation process follows the transactional concept used in Windows Installer technology (see Multiple-Package Installations in [3]).

7.4.1 Preconditions to implement the workflow

1. The delta packages need to be included in a package file that adheres to the structure specified in Section 5.1 of [5]. Such package should be already downloaded from the Questra Enterprise Server (QES) into the target system.

2. The AUA is responsible for triggering the installation, either automatically or indicated by the clinical user or FSE (sections 4.5 and 4.6 of [5]).

3. Similar to the normal MSI installation process, there should be a file that indicates the installation sequence of delta packages. We call this file *BOM_ MSP.txt*

4. The Allura SW is not running, i.e., the system is in open profile.

7.4.2 Description of the workflow

Figure 7.6 displays the execution workflow of installation of delta packages. In the list below we describe each of the steps included in this figure.

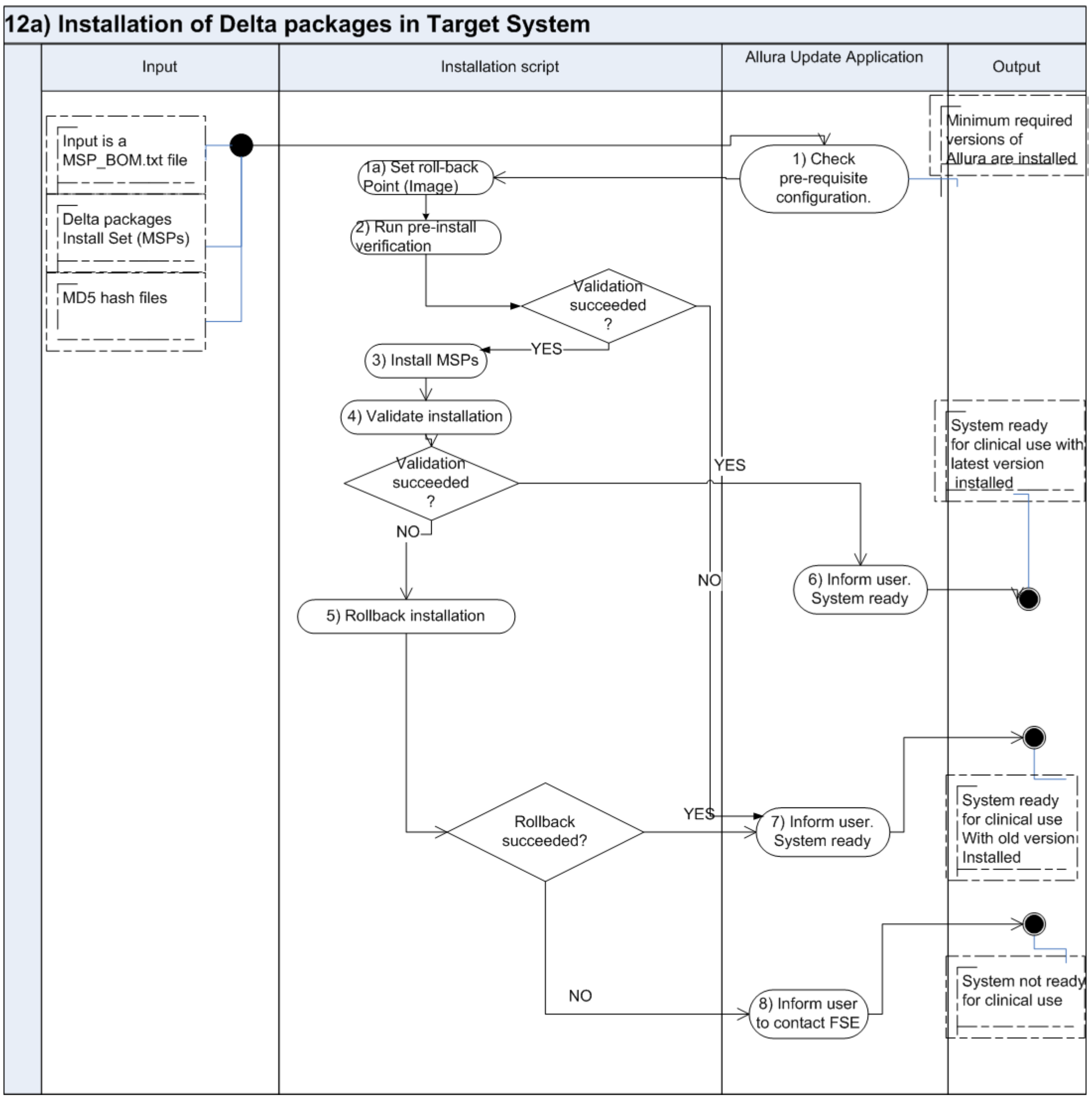

12a) Installation of Delta packages in Target System
Input
Installation script
Allura Update Application
Output
Input is a MSP_BOM.txt file
Delta packages Install Set (MSPs)
MD5 hash files
1a) Set roll-back Point (Image)
2) Run pre-install verification
Validation succeeded ?
YES
3) Install MSPs
4) Validate installation
Validation succeeded ?
NO
YES
NO
5) Rollback installation
1) Check pre-requisite configuration.
6) Inform user. System ready
Rollback succeeded?
YES
7) Inform user. System ready
NO
8) Inform user to contact FSE
Minimum required versions of Allura are installed
System ready for clinical use with latest version installed
System ready for clinical use With old version Installed
System not ready for clinical use

Figure 7.6: Installation flow of delta packages in the Allura system

1. **Verify pre-requisites** —step 1. Before installing, the following pre-requisites are verified by the Allura Update Application (AUA):

    (a) Minimal version installed. The package as distributed by the (QES), should contain in its meta-data file the PBL version in the target system. See sections 5.1 and 5.1.6 of [5]

    (b) Delta packages are not corrupted due to distribution or any other reason

2. **Mount rollback point** —step 1a. A restore point is set-up in case partial installation fails. The design of this restore point is further discussed in Section 7.6.

3. **Run Pre-patch validation**. In order to guarantee that the system configuration is the expected at file level, we need to run a pre-patch validation. The details of this validation are in Subsection 7.5.

4. **Install MSP files**. Once the installation prerequisites have been met, we proceed to install each of the delta packages. In order to install, Windows installer executable (*msiexec.exe*) is called. The installation order is prescribed

by *BOM_MSP.txt,* included in the update package downloaded from QSE –see Section 5.1 of [5]. During the installation Windows Installer verifies that:

(a) The MSP files are not corrupted, else it will signal via an error message, and

(b) The expected subsystem version is actually installed.

The return code of *msiexec* in case of success will be ERROR_SUCCESS (0). In other cases, refer to [3] – "Error Codes". All return codes are logged in the log of the currently patched subsystem.During the installation, the system notifies the (clinical) user through the system's screen(s) that an installation is ongoing. Note that during the installation process the user may want to cancel it, and thus by pressing a key, the installation is interrupted and rolled back.

5. **Validate partial installation**. This is an optional step (number 5 in Figure 7.6) given the performance overhead it imposes –see more details in Section 7.5.5.

7.4.3 What-if scenarios

When an unexpected error/event during the installation occurs, it must be possible to invoke the rollback procedure. Table 7.4 illustrates the possible scenarios where unexpected errors/events would lead to a cancellation or rollback of the installation. The thick symbol (✓) means that the error / event can occur during installation, validation or rollback.

Table 7.4: Unexpected Error/Events during installation of delta packages

Error/event	Installation	Validation	Rollback	Recovery procedure
-> Power failure. The host PC is unexpectedly shut-down -> Hibernation	✓	✓	✓	-> During step 1 of Figure 7.6, restart the installation procedure -> During step 2 –pre-patch validation, repeat the step and determine to proceed or not with the installation. -> During or after step 3 of the same figure, on restart of the system call the rollback procedure to go back to the previous known system state.

Error/event	Installation	Validation	Rollback	Recovery procedure
Ctrl-C user kills the process	✓	✓		->During Pre-patch validation, cancel the installation -> During installation, inform the user that the system state is inconsistent, and it should be restarted. Upon restart, initiate the rollback procedure
Disk failure	✓	✓	✓	Abort installation –if failure is detectable.
Disk full	✓		✓	Cancel current installation and invoke rollback procedure
Memory errors	✓	✓	✓	
Memory full	✓	✓	✓	FSE –if present- will have to switch off running services/applications and let the installation proceed. Otherwise clinical user will know what to do in this scenario, assuming documentation considers the scenario.
Script becomes unresponsive ("this program is not responding").	✓	✓	✓	Set a timeout limit after which the program is ended. Then call the rollback script to remove the latest set of MSPs installed.
Another program interrupts the execution of the installation	✓	✓	✓	Indicate the user that an error unexpected occurred during installation, and indicate to restart the system. The rollback should execute on restart of the system.
Blue screen	✓	✓	✓	The user has to restart the system, and upon restart initiate the rollback procedure

Error/event	Installation	Validation	Rollback	Recovery procedure
Virus attack/malware	✓	✓	✓	No installation should be triggered under this condition. How is this currently handled/detected by the anti-virus and -the Allura application? A suggestion is to provide notification to Contact FSE.
Expected system configuration not found before patching	✓			Cancel installation. Inform QES
Expected system configuration not found after patching	✓	✓		Invoke rollback procedure
Rollback point corrupted			✓	Indicate the user the system is not ready for clinical use, and indicate that needs to contact FSE.
Registry corrupted	✓	✓	✓	invoke rollback procedure
Binaries corrupted (due to disk fail)	✓	✓		invoke rollback procedure
Expected patch configuration not found	✓	✓		Cancel installation. Inform QES
Clinical emergency	✓	✓	✓	Indicate the user that rollback will be executed. Execute rollback immediately.

7.4.4 Post-conditions

After successful execution of the workflow, the Allura system is updated with the features of the upgrade version –State coming from step 6 of Figure 7.6.

In case that pre-patch validation does not succeed, then the installation is canceled and the system is left in the previous state –state coming from step 7 of Figure 7.6. The same system state holds after a successful rollback. Finally, in case rollback fails, the system is considered in an inconsistent state and the FSE must be contacted –state coming from step 8 of Figure 7.6.

7.4.5 Design Alternatives

Table 7.6 shows the advantages and disadvantages of the different switches evaluated to apply patches.

Table 7.6: Evaluation of switches used to apply patches

Switch	Plus	Minus
amus	Overwrites all files regardless of versioning.	Takes longer than *omus* and *emus.* Causes undesired file overwriting (i.e. lower versioned files replacing higher versioned ones, and user/config changes lost).
omus	Is faster than *amus* (by factor two or more)	Demands introduction of versioning -at file level- in the current MSI build process. Currently many files keep the same version number throughout releases
emus	-> Is the fastest of all three switches, when installing from minor upgrade patches -> All test cases of [21] pass, except when old version files have to replace newer versioned ones.	When installing major upgrade patches, there's no time difference compared to *amus.* When lower version files must be placed back –i.e. a downgrade, it will not work. For that we need major upgrade patches.

From the tests realized in [22] we choose to use the *amus* switch when installing major upgrade patches. The reason to use *amus* is that we ensure that all files are overwritten regardless their version. This is particularly desired when there are fewer files in the upgrade version, and/or we want to downgrade some files.

For minor upgrade patches we use the *emus* switch because Windows installer will patch files with equal or older version, and thus we prevent that lower version files in the upgrade release overwrite higher versioned files installed in the target release. Furthermore, installing minor upgrade patches does not impose a change in existing file versioning and it is the fastest to update the software.

7.5 Validation of delta packages installation

For validation of the installation, the selected tool is FCIV version 2.05, discussed in Subsection 6.3.2. The design is however open to other tools in the future.

7.5.1 Preconditions

1. Hash values for all installable items have been computed before-hand. The decision at the moment of writing is to use MD5 hashes. These values will be part of the update package distributed via Questra. More details in step 1 of next section.

2. All the delta packages are installed with no error return codes at the development site. See point two in Subsection 7.3 ("Validation of delta packages").

7.5.2 Description of the workflow

Validation of the installation of deltas is decomposed into two stages:

1. **Un-Rebasing of DLLs.** Files installed in the system with .dll, .ocx and .sdm extension are rebased —See "Rebasing" in the Glossary. Although rebasing is needed for a number of important reasons —as stated in Section 2.5.9 of [2], verifying this type of files will fail due to differences in the MD5 values. The chosen solution to this situation is un-rebasing: rebase files with a fixed time-stamp and load address, compute their MD5 values, and repeat this rebasing step in the actual system as depicted in Figure 7.7. Another alternative considered is summarized in Table 9.

2. **Validation as pre-installation step.** In this step we validate that the system matches the target system version (Old MSI installation of Figure 5.2 in Section 5.3). Step two of Figure 7.6 depicts that before installing the delta packages, the previously computed hash values are compared against the actual installed files' hashes.

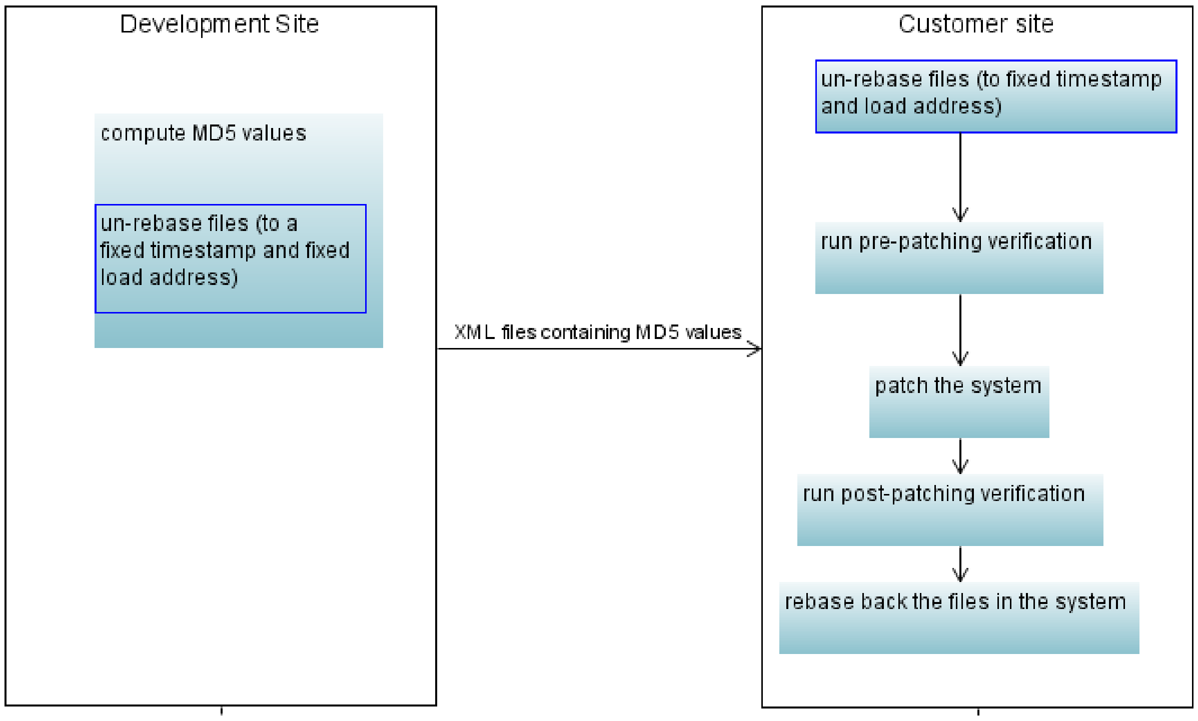

Development Site
compute MD5 values
un-rebase files (to a fixed timestamp and fixed load address)
XML files containing MD5 values
Customer site
un-rebase files (to fixed timestamp and load address)
run pre-patching verification
patch the system
run post-patching verification
rebase back the files in the system

Figure 7.7: Process to verify rebasable files

7.5.3 Postconditions

The output of the validation process is binary: either Passed (1) or Failed (0).

7.5.4 What-if scenarios

If the validation is interrupted for any error/event, it should be possible to repeat it once more. Table 7.4 indicates the scenarios applicable to the validation of installed deltas.

7.5.5 Design alternatives

The coming two subsections indicate the alternative design options analyzed regarding validation of installed delta packages.

7.5.5.1 Validation of Rebased files

Table 7.7 shows another option considered to the problem of verifying rebasable files mentioned in Subsection 7.5.2. This approach considers the possibility to rebase files before their inclusion into the MSI installation set, to avoid the rebasing during installation at the system.

Table 7.7: Rebase files and then ship into MSIs

Strengths	Weaknesses
Validation of installation will usually succeed, since there is no need to (un)rebase in the target system.	Involves measurement of how much memory each subsystem needs (budgets), therefore memory segments per subsystem must be assigned. Still there's a small chance that MD5 won't match due to any post-install tooling.

7.5.5.2 Decision of when to validate

Table 7.8 shows the main advantages and disadvantages of pre and post-patching validation.

Table 7.8: Pros and Cons regarding options for validation

	Before	After
Pros	Time is saved: if the validation fails, there is no installation of MSPs	The post-install validation confirms we have the latest version installed
Cons	Whatever alters files after MSI installation, will affect the success of validation (e.g., post-installation tools), and the only solution is to compute MD5s from an installed system	Post-patching validation adds up to the installation time. This is not desirable as stated in Subsection 5.9.5

Validation as pre-patching step. As seen in Subsection 7.3.2, depending upon the input files to compute MD5 hashes there is a trade-off between accuracy and speed. If we compute MD5 hashes from a completely installed system (i.e. installed via MSIs and including the effect of post-installation tools), the pre-patch validation time is long (15 to 26 minutes), but accurate since it verifies all installed items. This is the best approach when partial installations are introduced.

However, as the patching process becomes more stable and mature, the preferred solution should aim for speed. In this alternative approach is possible to compute MD5 hashes of only those items we know that will change due to patching.

Validation as post-patching step. In this step (number 4 in Figure 7.6) we verify that after patching the system we have the expected version –i.e. upgraded version. Post-patch validation is similar to pre-patch validation (step 2 of Figure 6), being the only difference that this validation is made after installing all delta packages. Post-patching validation is left as optional considering the time impact it has in the installation process (approximately five minutes), and the fact that the output of the delta generation and installation sub-processes is verified in [21, 22].

7.6 Rollback of the delta packages installation

To satisfy the requirement of reliability posed in Sections 5.7 and 5.9.1, we present in this section part of the work presented in [23].

7.6.1 Preconditions

The Master Boot Record (MBR) of the system boot disk needs to have the latest (Windows Vista and later) version of the Microsoft MBR. This supports booting using a method which allows booting into an instance of Windows pre-installation environment (PE).

7.6.2 Description of the workflow

The design makes use of the following two Microsoft tools/technologies:

- **Volume Shadow Copy Service.** This technology is available in Windows XP (and later) and it allows volume backups to occur while the system is still running. It essentially creates a snapshot in time of a set of volumes and then allows for that snapshot to be backed up.
- **ImageX.** This is Microsoft's file imaging technology. The *ImageX* tool is used to create, restore and manipulate Microsoft image files (.wim). It is a file-based imaging technology (as opposed to sector-based, such as Ghost) that supports compression and "single-instancing". The advantage of "single-instancing" is that the content of identical files, which are added to the same image more than once, is only stored once. This means that when incremental backups of a drive are made only the difference is stored in the image file, resulting in a much smaller image file.

The key features of the approach are:

- The backup runs on a live system. No reboot into a Ghost DVD are required.
- The backup and restore process are scriptable and run unattended.
- Backup and restore have similar performance in terms of speed when compared to current Ghost method.

7.6.2.1 Backup

The following steps describe the flow of the backup process. The backup process can be initiated as part of the software installation/update script that will be initiated by the FSE or by the remote software management infrastructure. Alternatively it can be initiated by the FSE via a GUI interface.

1. The *Vshadow* command line tool is initiated to create a snapshot of the desired volumes that are to be backed up.
2. The *Vshadow* tool interfaces to the VSS service and creates a shapshot set.
3. Once the snapshot set is created and ready for use, the *Vshadow* tool runs a callback script that will create the backup from this snapshot. In theory the installation of the system software could start at this point and run in parallel to the system backup, but more testing is required to assess the reliability of the parallel execution.
4. The callback script creates an image of the specified volumes using *ImageX*. The images are stored on the backup drive. After the images have been created the callback script returns.
5. The *Vshadow* tool releases the snapshot set and returns.

7.6.2.2 Restore

Whilst it is possible to create an image of a live system, it is not possible to restore an image onto a live system. For this reason it is necessary to reboot the system into Windows PE, which is a minimal version of Windows that runs from memory. The restore procedure is initiated automatically by the software installation/update script when an error is encountered. Alternatively it can be initiated by the FSE via a GUI interface. The steps of the backup process are as follows:

1. The backup process starts by copying the WinPE files, which are stored on the backup partition, to the root of C:
2. System is rebooted
3. The MBR of the system disk finds the presence of the Boot Configuration Data (BCD) and boots into WinPE.
4. The startup script of the customized WinPE automatically finds the latest backup image(s) on the backup drive.
5. The drives to be restored are reformatted
6. The *Imagex* tool is used to restore the image(s) to the selected drive(s)
7. System is rebooted
8. Because the C: drive was restored, the WinPE files are no longer there and therefore the system boots the Allura OS.

7.6.3 Postconditions

After restoring the system we obtain the system state coming from step 6 of Figure 7.6, i.e., the system is ready for clinical use with the old version installed.

7.6.4 Design Alternatives

Table 7.9 shows other options considered to create a restore point for Allura SW.

Table 7.9: Design Alternatives of Rollback

Option	Pros	Cons
Ghost 14	It is the current de facto standard in backup/restore. It is well supported and known to be reliable. It can create backups on a live running system.	Ghost 14 uses a new image file format that is not compatible with the current Ghost32 tool that is used at Philips. It installs a service on the system that was observed to use resources in the back ground and negatively affect the start-up time of the system. It was not able run under WinPE, but uses its own boot disk. This reduces flexibility in our restore process. Licensing costs.
Windows XP System Restore	It is a standard Microsoft technology that is built into Windows XP.	This does not provide a complete system backup. Only system files are backed up and software still needs to be uninstalled in order to rollback correctly. Based upon the experiments carried out by the author of [23], this method cannot guarantee that the system is 100% the same as the time the restore point was created.
Windows NTBackup	It is a standard Microsoft technology that is built into Windows XP.	Difficult to script. The file format (.bkf) can only be used by NTBackup self and therefore results in limited flexibility. It is problematic to get *ntbackup.exe* working under WinPE 2.1.

7.7 Summary

In the table below we summarize the main design decisions taken in FAMAR, and other alternatives considered during the project. Note that decisions made are in uppercase and alternatives are aligned to the right.

Table 7.10: Main design decisions taken in FAMAR

DECISION	ALTERNATIVES	PROS	CONS	MAPPING TO HIGH LEVEL REQUIREMENTS
GENERATION OF DELTA PACKAGES				
GENERATION USING WINDOWS INSTALLER'S SDK ITSELF				To create MSP files from MSIs makes the installation process manageable (Section 1.1) since the partial installation set is made in a deployment media that Windows Installer can handle. Moreover, the MSP files form an installation set that is manageable, i.e., feasible to remotely distribute compared to a full MSI installation set
	Advanced Installer	Small patch size; creates both major and minor upgrade MSPs	no command line interface to create MSPs	
	Install Shield	De facto authoring tool for installation at Philips Healthcare. It has a command line interface for most tasks	Does not create MSPs from MSIs but rather the user needs to explicitly indicate which installable items changed between two releases.	
	Virtual Patch	Generates delta packages fast	The size of the packages generated is almost as big as the input MSIs (See Table 6.1)	
INSTALLATION OF DELTA PACKAGES				

DECISION	ALTERNATIVES	PROS	CONS	MAPPING TO HIGH LEVEL REQUIREMENTS
USE LOG MESSAGES FROM WINDOWS INSTALLER ITSELF DURING PARTIAL INSTALLATIONS		Provides detailed information of the updating process, which is useful for problem shooting if installation fails	In the long term, logging files might exhaust disk space.	
	Implement logging from scratch		Limited. In case installation failed, it could not be possible to detect the exact reason why the installer engine failed	
ONE-PHASE VERIFICATION, I.E. BEFORE UPDATING THE SYSTEM		It guarantees that before the actual partial installation, the system configuration is in a known functional state	Relies on extensive in-house testing, to guarantee that post-install validation is not required in the field.	Satisfies the requirement of Reliability in the partial installation process (Sections 5.6 and 5.9.1)
	Two-phase validation, i.e. before and after system update	It can implemented to verify only files that will be changed. Guarantees to have the expected system configuration after a partial installation	It is time consuming, which is not desired in field.	
VALIDATION OF THE INSTALLATION				
USE MD5 AS CHECK-SUM FUNCTION		A widely used check-sum function, available in several tools. the implementation chosen has a fast performance (see figure 6.3). Solves the constraint that other file attributes (e.g. version, time-stamp) do not suffice to conclude the system is updated		It is an effective mean to guarantee file singularity, and thus to demonstrate that files are correctly update after a partial installation.

DECISION	ALTERNATIVES	PROS	CONS	MAPPING TO HIGH LEVEL REQUIREMENTS
	CRC	A widely used check-sum function, available in several implementations	If the polynomial used is small, there's an increasing probability of collisions- i.e. two different files with the same CRC value. The performance of tooling is slower compared to md5 tools (see figures 6.3 vs. 6.2)	
	Use file attributes	Attributes such as file size, time stamp and version are commons for most installed files	There are a number of files that have not updated version although their content changed. the time-stamp belongs to the building process rather than to the file creation/modification	

Chapter 8

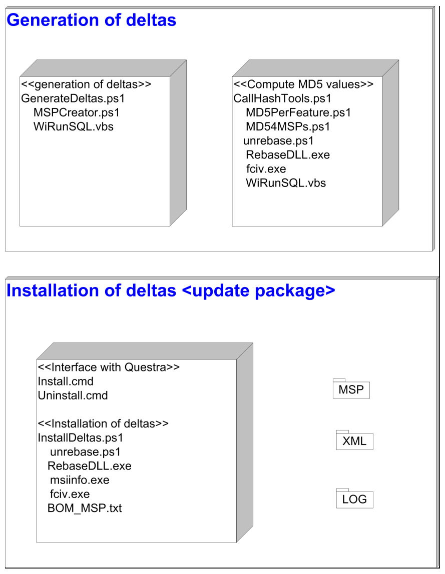
Design Implementation and Deployment

For the things we have to learn before we can do them, we learn by doing them.

Aristotle, cited on [24].

Preamble

This chapter reflects the details about how the design described in Chapter 7 was implemented. Included content is: description of components, functionality, details about the language/tooling used and code/output produced.



8.1 Implementation

The design addressed in Chapter 7 requires the following (software) resources to be realized:

- Windows installer software development kit (SDK) version 4.5 (or higher). The utilities contained in the SDK directory, need to be added to the PATH environment variable of the PC where the SDK is installed.
- Windows script 5.6 or higher. There are useful Visual Basic scripts in the \ *Tools* directory of Windows SDK (provided by Microsoft) that require Windows Script Host to be executed.
- It is required to have installed on the Allura system the latest version of Windows Installer. The version considered in this document is 4.5
- The prototyped scripts delivered in this project can be reused for production purposes. For these scripts, Windows Powershell Version 2 is required.

From the deployment point of view, the implementation of the design described in Chapter 7 is split into two main blocks :

1. The set of tools that generate the content for the update package, and
2. The set of tools that install and verify the installation of the update package

8.1.1 Implementation decision

Since we are required to build a solution that facilitates automation (as indicated in the Requirements chapter, Subsection 5.9.4), Windows Powershell was chosen as language to implement the design, because *"Powershell allows . . . to more easily control system administration and accelerate automation"*[25]. Useful Powershell features for this project are:

- Allows rapid prototyping, i.e. it provides an intuitive syntax and sufficient documentation/examples
- .NET types can be used, allowing reuse of existing code/assemblies
- Scripts are easy to debug within Powershell's integrated scripting environment

Table 8.1 summarizes other alternatives considered and their respective advantages and disadvantages.

Table 8.1: Selection of scripting language

ALTERNATIVES	PROS	CONS
Perl	It is the official development for SCM *only*	A steep learning/forgetting curve
Command line shell	It is the implementation mean of the current installation tooling	.bat/.cmd files are difficult to read and debug

8.2 Deployment

Besides the scripts implemented in Powershell, there are other (third party) tools that are brought together to form the tool suite produced in this project. These tools are mentioned in Table 8.2.

Table 8.2: Third-party tools used in the FAMAR project

Name	Origin	Functionality in the project
WiRunSQL.vbs	Sample scripts provided by the Windows Installer SDK 4.5	Executes SQL statements to Windows Installer packages
fciv.exe	Microsoft	File Checksum Integrity Verifier utility. Computes and verifies the integrity of files using MD5 hashes.
RebaseDLL.exe	Adaption of the program that does re-basing of DLL files in the current installation tooling	Implements the re-basing of files explained in Section 7.5.2

8.2.1 Generation of delta packages

The two main areas of focus for this project are the generation of the install set and the installation itself. Table 8.3 provides a list of scripts written to accomplish generation of delta packages. Scripts written in Powershell (*.ps1) are further listed in Appendix C on page 99

Table 8.3: Scripts for generation of delta packages

File name	Functionality
Generation of delta packages	
GeneratorDeltas.ps1	Create a set of delta packages given as input two installation sets. This script generates delta packages Allura system wide.
MSPCreator.ps1	Takes as input two Allura MSI files and generates a MSP as output
Computation of MD5 values	
CallHashTools.ps1	Takes the administrative installation directories (i.e. target and upgrade) and computes the MD5 hashes of their contents. This script computes the MD5 hashes Allura system wide.
MD5s4MSPs.ps1	Computes MD5 hashes for the delta install set. This script is invoked from CallHashTools.ps1
MD5PerFeature.ps1	Takes as input the administrative installation of a MSI file and computes the corresponding MD5 hashes. An important attribute of this script is that allows the computation of files which belong to a specific product feature, e.g. *test, production, stub*

File name	Functionality
ComputeMD5s.ps1	Takes the administrative installation directories (i.e. target and upgrade) for one product (i.e., Allura sub-system) and and computes the MD5 hashes of their contents. This script is alternative to MD5PerFeature.ps1, when files belonging to all features are installed.
Unrebase.ps1	Calls iteratively RebaseDLL.exe, to re-base files within a given directory.

8.2.2 Installation of delta packages

Installation of delta files is accomplished with the scripts mentioned in Table 8.5.

Table 8.5: Scripts to install delta packages in the target system

File name	Functionality
Install.cmd	This is a required interface script with the Questra Service Agent discussed in [5]. It calls the *InstallDeltas.ps1* script
Uninstall.cmd	This is a required interface script with the Questra Service Agent discussed in [5]. The purpose of this script is to uninstall an update package
InstallDeltas.ps1	Installs the set of delta packages, as discussed in sections 7.4

8.3 Summary

In this chapter we describe how the prototype requested in this project is implemented. There is a compromise between adopting one of the scripting languages used at CV, or introducing one that allows us to do rapid prototyping. We favor the introduction of Powershell in order to speed prototyping. Finally, in Figure 8.1 we observe the deployment's view of the tooling written / reused during this project.

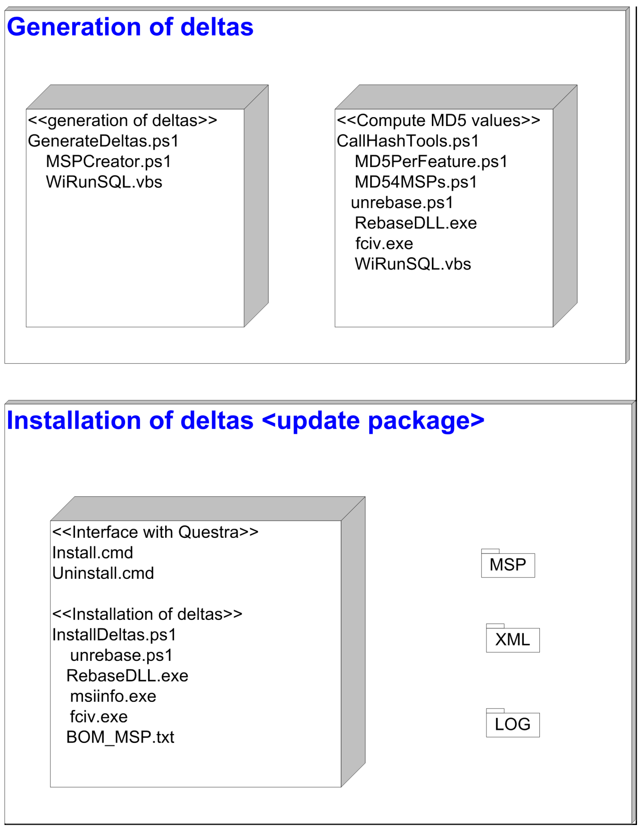


Figure 8.1: Deployment view of the tooling used in the FAMAR project

Chapter 9

Validation and Validation

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger Dijkstra, in "Notes On Structured Programming".

Preamble

In this chapter we describe the methodology followed to validate the design and its results. Contents addressed is a description of the test design, test cases and the results obtained by the actual execution of tests.

9.1 Introduction

In order to satisfy project requirements related to reliability and testability of partial installations (discussed in Subsections 5.9.1 and 5.9.2 respectively), a Test Design Specification and its corresponding test cases were executed.

IMPORTANT: The testing execution was done by an independent testing team, to which the author only provided delta packages. In this way we satisfy the requirements mentioned above. The discussion presented in this chapter is a summarized version of [21] and [22].

9.2 Test Design

A test design to validate the result of delta generation and installation is made taking into account the following:

- A set of test items that represents the contents of real Allura installation items
- A set of test cases. The testing technique is black-box: the output of the generation and installation processes is evaluated.

9.2.1 Test items

The following list displays the test items (files) that resemble the items installed in the Allura system:

String DLL Files	Win32 exe Files
ActiveX Controls	Font Files
Registry Files (irs)	ProxyStub DLLs
.NET assembly DLLs	.NET executables
Program Database Files	Language DLLs
NeverOverWrite Files	Non Self Registering DLLs and Application Shared
Non Self Registering DLLs and System Wide Shared	Non Self registering Exe
Permanent Data Files	Text Files Registry Files (reg)
Self Registering DLLs and System Wide Shared	Self Registering DLLs
Self registering Exe	Shared DLL
Static library Files	

These file types are described in [2], and resemble the real Allura files in that they are grouped in terms of the list above. Files are classified according to these file groups, by means of .ini files. These .ini files are given as input to the tooling introduced in Subsection 4.3, which generates Allura's full installation set.

9.2.2 Features to test

The installation was tested evaluating the following features:

1. Addition of new files

2. Deletion of existing files
3. Replacement of existing files with files having a different (higher) version
4. Replacement of existing files with files of the same version but different size
5. Replacement of existing files with the same version and size but different contents
6. Custom actions added. A custom action "*enables the author of an installation package to extend the capabilities of standard [*install*] actions by including executables, dynamic-link libraries, and script" [3]* .
7. Custom actions removed.

9.3 Test Cases

Three different test cases were defined to cover all the features mentioned in the previous section

1. Installation of Delta Packages
2. Validation of Delta Packages Installation
3. Cancellation of Delta Packages Installation

Test cases were executed in the following environment:

- **Windows XP SP3**. The Operating system version installed in the host PC
- **Windows Installer 4.5**. The installation engine used in Windows operating systems
- **.NET framework v 1.1 and 2**. These versions are required for a number of application files to run
- **InstallShield X Professional**. The build tool currently used to generate full install sets
- **A VMware image with a clean (operating) system**, i.e. without any applications installed. Tests cases were executed using this image.

9.3.1 Results

Table 9.1 displays the results obtained of the tests made with minor upgrade delta packages. The table indicates the result of testing each of the features introduced in Subsection 9.2.2. For the last two tested features we see that they did not pass. Moreover, when executing feature test six, it was observed that higher version files are replaced by lower version files (shipped in the upgraded version). After identifying the causes of this failure, a number of modifications are required in the existing build process to make these cases succeed. These modifications are discussed in section 10.2.

Table 9.1: Results of test cases executed

Test feature identifier	Result: Passed(✓) or Failed(✗)
1	✓
2	✓
3	✓
4	✓
5	✓
6	✗
7	✗

9.4 Partial installations of Allura software

Next to the test cases discussed in Section 9.3, the author realized a number of partial installations using actual Allura SW releases. These partial installations were made in a VMWare environment, and tested the installation of major upgrade patches using the *amus* switch. There were three different updates installed, which main results in terms of time saved are shown in Table 9.2. Here we present only percentages due to confidentiality reasons, but the actual values are recorded in [26].

Table 9.2: Results of partial installations using three different Allura releases

Release (PBL)		Percentage (%)	
Target	Upgrade	Subsystems changed	Time saved compared to full installation
6.0.0	6.0.1	17	61
6.0.1	6.0.2	61	25
7.0.0	7.0.1	81	39

From the values shown in the table above, we would expect a linear behavior in the second row compared to the other two releases installed. However, for this particular release, the update of the UI subsystem contained many changes (60% difference between the target and upgrade releases[1]) and had therefore a negative influence in the time saved (UI took 42% of the total installation time).

9.5 Summary

This chapter presents the testing approach followed to validate the prototype implementation of this project. The fact that the test design and test execution was carried out independently by a testing team of Philips, supports the effectivity of the tooling produced in this project. Analysis of the results obtained specially in the test cases that did not succeed allows to identify necessary changes to the current build process. These changes are addressed in the coming chapter.

[1]For the first row (6.0.0 to 6.0.1) there were no changes in the UI subsystem, and for the third row (7.0.0 to 7.0.1) there were 26% of changes.

Chapter 10

Concluding findings

Better is the end of a thing than its beginning...

Ec 7:8

Preamble

In this chapter we present the conclusions reached in FAMAR project.

10.1 Main results

The main challenges in this project were how to make the installation of software in the Allura system faster, more manageable and enabling at the same time to be remote.

Faster. As we have seen in Section 9.4, compared to the current process installing via MSIs, the savings in time are between 25% up to 61%, depending upon the number of changes between the target and upgrade releases at hand. The experiments recorded in [26] consider three different Allura releases (PBL), which have 17%, 61% and 81% of subsystems changed.

Managed. With the ability to reliably identify each of the files updated in the system, we are able to determine the success or failure of partial installations. The prototype delivered to Philips Healthcare consists of a two-phased validation mechanism that verifies the existence of the right system version before updating and confirms that after installing the update package, the system is left in a consistent state according to the upgrade version. This validation satisfies the requirements of Reliability and Testability stated in Subsections 5.9.1 and 5.9.2, respectively.

Remote. Because the size of a partial install set does reduce compared to a full install set —as recorded in [26], now Philips Healthcare is able to distribute update packages that satisfy the constraints mentioned in Subsection 5.8.2.

The small size of the update package not only makes possible its distribution, but it also enables to remotely update the system.

10.2 Future work

In the coming subsection we mention those things we believe will add the most value to Philips Healthcare in the context of this project.

10.2.1 Changes to the current build process

From the current MSI build process introduced in Figure 1.2 -steps 2 and 3, there are a number of required changes based upon the test results described in Chapter 9 and experiments carried out by the author.

1. **Provide only three-field version MSIs**. If we want to deploy updates via MSP files, we must provide three fields only in the ProductVersion specified inside each MSI file. This means a change of the versioning scheme defined in [27].

2. **Keep MSIs with ProductNames and UpgradeCode properties intact in order to update via MSPs**. For the tooling that creates MSP files it is required to keep these two properties unchanged throughout releases. This is not the case in a minority of subsystems of the Allura software.

3. **Do not use nested MSIs to install Allura software.** Nested MSIs subsystems do not patch correctly. Microsoft discourages to have nested MSIs and rather advices to use more than one MSI if needed and sequentially install these –See “Windows Installer Best Practices; Do not ship concurrent installations” in [3].

4. **Adapt installation tooling**. Currently Allura software is installed by a batch file that is (manually) run, now it should be possible to trigger a script /program that

   (a) Shuts all running application processes down

   (b) Installs the delta packages

   (c) Checks outcome of installation and take appropriate actions according to Table 7.4.

   The scope of this project only gives a solution for item b and the first part of item c – i.e., checks outcome. The rest should be done by system installation.

5. **Adhere to Microsoft's restrictions**. (sub)System Configuration Management needs to observe a number of important conditions —imposed by the tooling used- when creating MSP files for Allura software. The following is an excerpt from the documentation found in [3]– "Creating a Patch Package", which is relevant to Allura software:

   When you author a patch package you must also adhere to the following restrictions:
   ** Do not move files from one folder to another.*
   ** Do not change the sequence number of existing files. The sequence number is the value specified in the Sequence column of the File Table.*
   ** Do not change the primary keys in the File Table between the original and new .MSI file versions. Note: The file must have the same key in the File Table of both the target image and the updated image. The string values in the File column of both tables must be identical, including the case.*
   ** Do not author an (MSI) package with File Table keys that differ only in case, for example, avoid the following table:*

File	*Component_*	*FileName*
readme.txt	*Comp1*	*readme.txt*
ReadMe.txt	*Comp2*	*readme.txt*

10.2.2 Installation of deltas

The installation of delta packages can report via log files the result of Windows Installer when applying the patches. However, a complete integration with the logging process of the Allura Service Framework was not done due to time constraints. We had to favor other aspects like validation of the installation process.

10.2.3 Validation of installation

Although the design considers the validation of Windows registry settings (mentioned in section 7.5), time did not allowed us to verify updates of other installable items such as .NET assemblies and shortcuts. Nonetheless, in the test cases described in Section 9.3 some testing work was done in this direction.

10.2.4 Rollback of partial installations

Rollback of partial installations was analyzed but due to time constraints it was left outside the author's scope in this project. The design is specified in [23], and its implementation is part of the work that people at CV will have to undertake.

Chapter 11

Project Management

The Pertinent Question is NOT how to do things right — but how to find the right things to do, and to concentrate resources and efforts on them.

Peter Drucker, in "Managing for Results: Economic Tasks and Risk-Taking Decisions".

Preamble

This chapter reflects the relevant issues related to this project's management. The chapter contains the following subjects: Work-Breakdown Structure, Planning and Scheduling, and finalizes with learning points derived from the project's execution.

11.1 Work-Breakdown Structure

From the first meetings held with the steering committee, the author proposed an iterative approach to solve the challenges posed in FAMAR, which had an original duration of nine months. Three iterations were proposed as shown in Figure 11.1. The blocks with thick line and blue font are deliverables as agreed in the kick-off meeting (held on early January) of the project.

From January to March, the main deliverables were an overview of the software installation process of the Allura system, and the identification of specific problems of this process. The purpose of the overview is to clearly determine the project's scope, while the selection of specific points is to decide what can be solved given the author's profile.

Once identified these specific problems –see Subsection 3.3.1 on page 20, the deliverables for the second iteration from April to July were a requirements and a design specifications. Within this second iteration it was considered also to begin with the implementation of the designed solution.

Finally, the main deliverables of the third iteration were planned to be the complete implemented solution along with its validation results. For the university as stakeholder, the main deliverable is this report, which is filled in with the outcome of the previously mentioned deliverables.

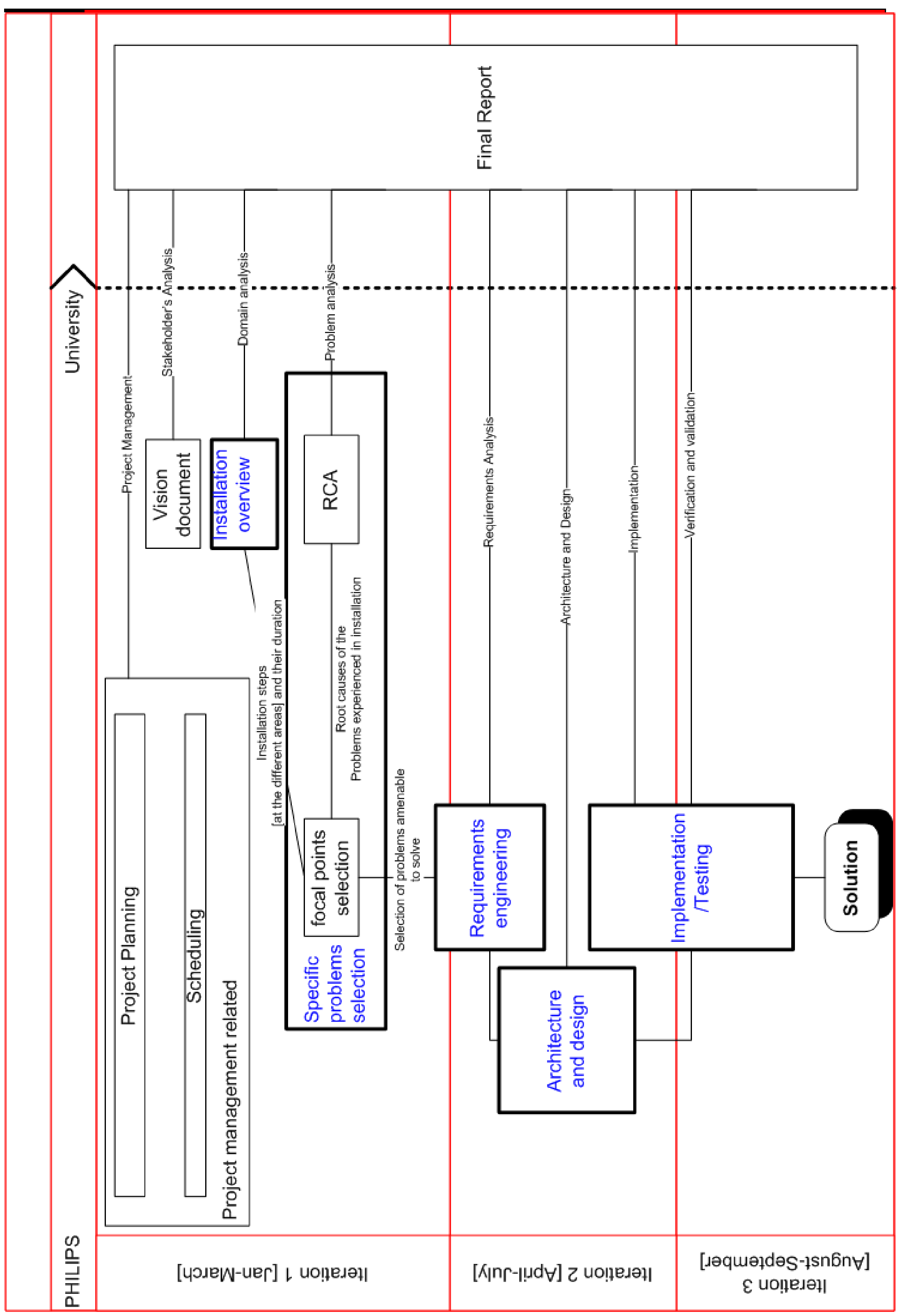


Figure 11.1: Workflow of the project phases and deliverables

11.1.1 Monthly meetings

Every month from January until August, the steering committee of the project gathered with the author to discuss the progress and re-prioritize efforts. The results of these meetings were captured in minutes that were distributed to the members of the committee.

11.2 Planning and scheduling

Based upon the deliverables and the iterative approach chosen, in Table 11.1 we can observe the scheduled and actual dates of each deliverables. Intermediate versions of relevant deliverables are included.

Table 11.1: Scheduling of the project

	Scheduled date		Actual date	
Deliverable/task	begin	end	begin	end
OVERVIEW OF THE CURRENT INSTALLATION PROCESS	06-01-2009	02-02-2009	06-01-2009	25-03-2009
IDENTIFICATION OF DETAILED PROBLEMS EXISTING IN THE INSTALLATION PROCESS AND OPPORTUNITIES TO SOLVE THEM	09-02-2009	06-03-2009	10-02-2009	18-03-2009
SELECTION OF PROBLEMS TO SOLVE (I.E., FOCAL POINTS) IN THE REST OF THE PROJECT	06-03-2009	09-03-2009	13-03-2009	31-3-2009
REQUIREMENTS SPECIFICATION	1-4-2009	29-4-2009	1-4-2009	30-4-2009
Revision in June			05-06-2009	05-06-2009
Revision in September			17-09-2009	17-09-2009
Final version delivered at Philips			01-10-2009	01-10-2009
DESIGN SPECIFICATION	30-4-2009	25-5-2009	15-5-2009	5-6-2009
Update in June	1-6-2009	10-6-2009	5-6-2009	29-6-2009
Update in July			21-7-2009	31-7-2009
Update in August			7-8-2009	21-8-2009
Update in September			11-9-2009	11-9-2009
Revisions in October			7-10-2006	26-10-2009
Final version delivered at Philips			4-11-2009	23-11-2009
DESIGN IMPLEMENTATION	27-05-2009	24-07-2009	30-05-2009	7-10-2009
DESIGN VALIDATION	15-07-2009	24-08-2009	22-06-2009	12-10-2009
TECHNICAL REPORT	10-04-2009	11-09-2009	23-03-2009	6-11-2009
FINAL PRESENTATION	18-09-2009	18-09-2009	17-11-2009	17-11-2009

Looking at the actual dates, it is evident that there is a delay of approximately 8 weeks in the project. There are a number of reasons that have contributed to the delay:

- **The scope of the project had to be determined**. The goal of the project always was clear (See Section 3.1). However, part of the challenge posed in this project was to produce the first two deliverables mentioned in Section 11.1, and hence determine the project scope.

- **Planning was overly optimistic.** The scheduling of the project was adjusted after the first iteration. However, estimates did not properly consider
  - The complexity of some analysis and feasibility tasks to be done
  - Interaction with some stakeholders (and hence, agenda conflicts)
- **Communication of project (re)scheduling missed clarity**. During the project execution, setbacks and their implications were not properly communicated to stakeholders. There were messages about the overall planning sent from January until beginning April. However, as of May, the author only reported monthly goals, missing thus the "big picture".

11.2.1 Reflection on the management process

Projects in essence are challenging. They are resource constrained (time, money, people,...) and often, only their desired outcome is clear. Projects can be metaphorically seen as a battle to win: there are enemies (challenges) to defeat and the victory is ahead. Nonetheless, in the middle of the battle is easy to forget what it is that we are fighting for, and it is not always evident what we should do. It is only after the battle finishes that (provided we survive) we can learn something about it.

From the experiences obtained in this project, the author wants to share with his readers the following lessons:

- **Keep always in mind the big picture.** With all the day-to-day small tasks and problems to overcome, it becomes easy to loose the notion of time, and thus the importance of things. In a project like in life, we ought to keep in mind not only our (final) goal(s), but also how and why do we want to reach the(se) goal(s). Loosing the big picture appears when we are solving the symptom rather than the problem, and we forget about the rest of work to be done.
- **Communicate constantly, even if these are bad news**. For many people, it is always difficult to acknowledge the need of help, or that we have made a mistake. At many points in time the author missed to effectively communicate, specially his concerns or mistakes. Communication as a bilateral process is complicated, let alone when it needs to be as accurate as possible, specially in an engineering environment. Efective communication does not only mean to broadcast a message assuming it will be understood as the sender intended. The next step for the sender is to make sure the receiver(s) indeed got the message and is (are) aware of the implications.
- **Manage stakeholders' expectations**. People expect much or little from us. Derived from the previous point, it is our responsibility to make sure that stakeholders' expectations are adapted to what we can provide them, keeping a sober-headed judgment of ourselves. In any case it is always better to begin with a modest profile and (at last) gently exceed people's expectations, than the opposite. And yet again, in practice this is easier said than done.

11.3 Summary

In this chapter we present how the FAMAR project was managed. The author defined with the project's steering committee a list of top deliverables, upon which the planning and scheduling was made. During the execution of the project, there where risks and setbacks not timely addressed, which caused a delay in the completion of the project. From this experience, the author has learned valuable lessons that hopefully he will put into practice in the future. These lessons in short are:

- **Keep always in mind the big picture,**
- **Communicate constantly, even if these are bad news, and**
- **Beware of stakeholders' expectations.**

Appendix A. Patching Memo

To : A. Visser, P. Beks, O. Mendez, P. Dingemans
Copy to : M. Loos
From : P. Langemeijer
Subject : Creating small installation sets

Introduction

Current Allura installation sets are very large. This is a problem when distributing a new version. This memo describes two mechanisms how to create a small installation set by using Microsoft patch tools.

Microsoft patch tools

Microsoft Platform SDK.

When Should I Distribute a Patch?

From InstallShield X help:

Also: search on: Creating a patch creation properties file

The benefit of using the patching mechanism to distribute your upgrades is that a patch can be significantly smaller than a full installation package. This enables you to deploy your upgrades using much less bandwidth than that required to deploy a full installation package. However, under certain conditions, you should use a full minor or major upgrade instead of a patch to update a previous version of your product. For example:

* If the target image was created with Windows Installer 1.2 or earlier, and the upgraded image is created with Windows Installer 2.0 or later, you should distribute your update as a full minor or major upgrade, but not as a patch. Creating patches for packages that span this schema inflection point can be problematic. For more information, see Val0011, which describes the associated validator for upgrading and patching validation.

* If you want your update to move one or more files on the target system from one location to another, you should distribute your update as a full minor or major upgrade, but not as a patch. If your end users install a patch for an update that moves files on the target system, problems may occur. For example, the patch may not work, a repair to the target system may not work, a subsequent patch may not work, or end users may not be able to uninstall the product. As a workaround, you can create your patch so that it deletes the files in the old location and adds the files to the new location.

* If you want to change your installation from compressed to uncompressed, or vice versa, you should distribute your update as a full minor or major upgrade, but not as a patch. If you use a patch in this scenario, a repair to the target system may not work, a subsequent patch may not work, or the end user may not be able to uninstall the product.

* Patches cannot be created for major upgrades of InstallScript MSI projects. Therefore, if you need to distribute a major upgrade for an InstallScript MSI project, you should create a full major upgrade.

In a patch, the package code is different than the one in the previous installation package. In addition, for all target images created with a version of Windows Installer earlier than 3.0, you should change the product version for a patch. This change is recommended because of the way that Windows Installer performs repairs: first it looks at the product version number, then at the file dates and file versions. If an unversioned file is updated in a patch, the product version should be updated; otherwise, if end users install patches out of sequence, a repair may reinstall the wrong unversioned file.

In most cases, you will begin patch creation in the Patch Design view. If you are looking for a simple patching solution, you can create a QuickPatch project. To determine which patching strategy you should use, see Which Patching Strategy Is Right for Me?

Using MSP (Microsoft Patch file)

Advantages:
* Small patch packages
* Fast install
* No reassemble needed
Disadvantages:
* Not well documented
* Unclear if it can be used for major upgrades
Results:
Original compressed size 67Mb (uncompressed 275Mb), patch file is 25 Mb.
The version number of an MSI should be 3 fields instead of 4 fields.
Before you can create a path, you must expand the MSIfile:
Msiexec /a <MSI> TARGETDIR=<path>
Example:
* Copy the MSI files from:
\\nlybstqvp4ms024\Publish\Allura\System\PDC_76.1.7\InstallationSet\SubSystems
* Mkdir D:\PMS_CV_AlluraXper_FSClient_76.1.7
* msiexec /q /a PMS_CV_AlluraXper_FSClient_76.1.7.msi
TARGETDIR=D:\PMS_CV_AlluraXper_FSClient_76.1.7
* Assert that the version contains only 3 fields. Otherwise:
o MsiUpdateVersion.exe PMS_CV_AlluraXper_FSClient_76.1.7.msi 76.1.7
* Create a .pcp file (how: http://support.microsoft.com/kb/314131)
* Change dir: "C:\Program Files\Microsoft SDK\Samples\sysmgmt\msi\Patching"
* Create the patch:
Msimsp.exe -s <name>.pcp -p <full path>.msp -l Allura.log
* Measure the size of the patch
* Precondition: the old MSI has been installed. (you may need to install PMS_Infra.msi first)
* Install patch: Msiexec.exe /p Sample.msp REINSTALL=ALL REINSTALLMODE=omus /L*v logfile.txt
* Measure installation time
* Validate
Relevant links:
http://support.microsoft.com/kb/314131
http://msdn.microsoft.com/en-us/library/aa370324(VS.85).aspx
http://www.codeguru.com/csharp/.net/net_general/visualstudionetadd-ins/article.php/c9663
Example command line applying the patch:

Msiexec.exe /p Sample.msp REINSTALL=ALL REINSTALLMODE=omus

Using binary patching tools

http://blogs.msdn.com/jmstall/archive/2006/11/07/binary-diff.aspx

Advantages:

* Small patch packages

* It probably works

* Works for major upgrades

Disadvantages:

* Takes a lot of time to reassemble

* Takes a lot of time to install

The patch tooling is only effective when the difference between the old and new files is small. For instance, when applying a patch between two MSI files, the patch file will be very large (about 90% of the original file) even if the difference between the two MSI versions is small. The reason is that the MSI contains the files in a compressed format.

Thus the solution is to use MSI files with uncompressed files. A single MSI file can be split into a small MSI file and uncompressed files using an administrative installation (msiexec /a).

The next sections describe how this tooling could be used.

Installation Set

An Allura installation set consists of:

- The MSI files to install
- A bill of material file containing:
  - The MSI files to install (and their version)
  - The version of the installation set
- Installation tooling (install, uninstall)

Use case Get latest installation set

* A system in the field requests the Software Distribution Center for the latest installation set while supplying the current version of the installed installation set. The system in the field has the installation set (MSI files) stored somewhere on a disk.

* The SDC creates a patched installation set (difference between installed installation set and the latest installation set). This will be a single file (zip format).

* The system in the field receives the patched installation set.

* The latest installation set is reassembled using the received patched installation set and the installed installation set.

Creating a patched installation set

* For both the latest as a previous installation set:

convert each single MSI file into a small MSI file (containing only the administration for the installation) and the uncompressed files:

msiexec /a <MSI file> /passive /q TARGETDIR=<temp location>

* Make the patch files between the previous and latest installation set.

* Compress all patch files into a single zip file.

Reassembling an installation set

TODO

Appendix B. Applicable FDA Regulations

The following excerpts from the subchapter of medical devices of the FDA are applicable for (automated) updates of software in Allura systems:

> *(a)General. Each manufacturer shall develop, conduct, control, and monitor production processes to ensure that a device conforms to its specifications. Where deviations from device specifications could occur as a result of the manufacturing process, the manufacturer shall establish and maintain process control procedures that describe any process controls necessary to ensure conformance to specifications. Where process controls are needed they shall include:*

1. *Documented instructions, standard operating procedures (SOP's), and methods that define and control the manner of production;*
2. *Monitoring and control of process parameters and component and device characteristics during production;*
3. *Compliance with specified reference standards or codes;*
4. *The approval of processes and process equipment; and*
5. *Criteria for workmanship which shall be expressed in documented standards or by means of identified and approved representative samples. ...*

   (i)Automated processes. When computers or automated data processing systems are used as part of production or the quality system, the manufacturer shall validate computer software for its intended use according to an established protocol. All software changes shall be validated before approval and issuance. These validation activities and results shall be documented.

The text above was ratified in an e-mail sent to the FDA personnel, which is shown next:

> *Mr. Mendez: I forwarded your inquiry to the CDRH software expert, and am including his response below. If you need further assistance regarding regulation of medical software, I suggest you contact Mr. Murray directly. If you require assistance with regulatory authorities relative to diagnostic x-ray systems, you may contact me as shown in the following signature block. Best regards, Tom Tom L. Mosely Consumer Safety Officer ...* ——-

From: Murray, John F. Sent: Wednesday, April 15, 2009 9:00 AM To: Mosely, Tommy L Cc: Boyd, Sean M; Gunzburg, Charles R.; Gullick, Barbara; Walker, J. Nick; Facey, Normica Subject: RE: Radiological Health Program Feedback Tom, The installation of software in a medical device is considered to be a step in the manufacturing process. For example, the installation of medical device software into a generic PC would result in a finished device, so that the installation of the software would be considered a step in the manufacturing process that leads to a finished device. The installation of this software can come in many forms: factory installed, field installed by the technician, field installed by the user, installed by the OEM at the point of purchase, installed via download from a website. The device manufacturer gets to decide what method he or she will use. The manufacturer is responsible for the successful outcome of this manufacturing step, regardless of who actually does it or where it is actually done. This is true because the manufacturer decides on the method. Since this is a manufacturing step 21 CFR 820.70 would apply; at a minimum, part (a) is most significant, also part (i) if any section of the installation is automated.

Appendix C. Listing of scripts developed by the author

This appendix shows the prototype scripts written by the author, which were introduced in Section 8.2 on page 75.

GeneratorDeltas.ps1

```
#this script takes as input compressed MSI files (from two directories)
# and generates an MSP file # the calling arguments are
# .\GeneratorDeltas.ps1 <target dir | file> <Upgrade dir | file> -[option]
# examples of use:
# .\GeneratorDeltas.ps1 'D:\Target\PMS_DBContent.msi' 'D:\Upgrade\PMS_DBContent.msi'
# or simply
# GeneratorDeltas 'D:\SandboxPatching\Target\' 'D:\SandboxPatching\Upgrade\' [-sync]
#alternatively, an additional argument can be given to the script:
#-sync: to synchronize install sequences in both MSIs
#########################################################
#declare function(s) and auxiliary environment
#########################################################
[System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
[string]$option= $args[2];
function CallCreator{

      for($i=0;$i -lt $listA.length -1; $i++) {
      #call sequence sync if specified as argument
           if ($option.Equals('-sync')){
           .\SyncSequences.ps1 $ListA[$i] $ListB[$i]
           }
      .\MSPCreator.ps1 $ListA[$i] $ListB[$i] #if ($i -eq 1 ) {break} ;
      }

}
clear;
echo "generation process begins at: " (Get-Date).ToString()
#test input parameters
if ($args[0] -eq $null -or $args[1] -eq $null){

      echo 'examples of use:'
      echo '.\GeneratorDeltas.ps1 <targetversion.msi> <ugradeVersion.msi> [-sync]';
      echo 'or: ';
      echo ' <pathTargetDirectory> <pathUpgradeDirectory [-sync]';
      break;

}
#we have two files as input
$FILE1= $args[0];
$FILE2= $args[1];
$file1 = $file1.split('\')
$file2 = $file2.split('\')
$file1= $file1[$file1.length -1].ToUpper();
$file2= $file2[$file2.length -1].ToUpper();
#FIND IF FILES ARE OF THE RIGHT TYPE
if ($file1.Contains(".MSI") -and $file2.Contains(".MSI")){

      #call sequence sync if specified as argument
      if ($option.Equals('-sync')){

           echo 'equalizing install Sequences...'
           .\SyncSequences.ps1 $args[0] $args[1]
           echo 'done'
      }
      #we pass the complete file paths
      .\MSPCreator.ps1 $args[0] $args[1]

}
#we (must)have (then) directories as input
else{

      #read the paths where the two installation sets reside
      $listA= Get-ChildItem $Args[0] -recurse -include *.msi | sort ;
      $listB= Get-ChildItem $Args[1] -recurse -include *.msi | sort ;
      if($listA.length -ne $listB.length) {

           $a = Read-Host 'the number of subsystems differs; do you want to proceed: (y), (n) `
           -not recommended-? '
           if ($a -eq 'y' ) #| 'Y' {
                # call the script that creates a MSP file
                CallCreator
           }
           else {
                echo "generation cancelled by user"
                break
                }
           }
           else {
                CallCreator
           }
      }
      ########################################################
      # create a Build of Material for MSP files
      ########################################################
      $MSPDir= '..\MSP\';
      $installDir = '..\installation\';
      gci -Path $MSPDir *.msp | fl Name > ($installDir + 'BOM_MSP.txt');
      echo "generation process ends at: " (Get-Date).ToString()
```

MSPCreator.ps1

```
[System.Reflection.Assembly]::LoadWithPartialName("System")
########################################################
#script to create PCP and MSP files
########################################################
$script:PatchName -as [string];
function getLargestSeq {

      $SQLCommand= 'Select `Media`.`LastSequence` from `Media`';
      $SeqTarget = cscript /nologo .\WiRunSQL.vbs  $TargetFile $SQLCommand;
      $SeqUpgrade = cscript /nologo .\WiRunSQL.vbs  $UpgradeFile $SQLCommand;
      [int]$t= $SeqTarget[0];
      [int]$u= $SeqUpgrade[0];
      if ($u -gt $t)
           {return $u}
      else {return $t}

}
function GetVersion {

      param ($FileName)
      $SQLCommand= "select `Value` from `Property` where `Property`.`Property` = `
                  'ProductVersion'";
      $productVersion -as [string]
      $productVersion=cscript /nologo .\WiRunSQL.vbs  $FileName $SQLCommand;
      return ([string]$productVersion).Trim();

}
function ModifyVersion {
param ($FileName)
$SQLCommand= "select `Value` from `Property` where `Property`.`Property` = 'ProductVersion'";
$productVersion -as [string]
$tmp -as [string]
$productVersion=cscript /nologo .\WiRunSQL.vbs  $FileName $SQLCommand;
$shortversion = $productVersion[0].split('.')
$SQLCommand= "update `Property` set `Value` = '" + $shortversion[0]+ "."+ $shortversion[1] +
             "."+ $shortversion[2] +            "' where `Property`.`Property` = `
             'ProductVersion'";
#echo "-->Running SQL on $FileName : $SQLCommand"
cscript /nologo .\WiRunSQL.vbs  $FileName $SQLCommand;
#assemble patch file name
if ($PatchName.Equals(""))
{

      if ($FileName.GetType().name -eq 'String') {
           $FileName=$FileName.split("\");
           $PatchName=$FileName[$fileName.length -1].TRimEnd(".msi") + "_"+ $shortversion[0]+
           }
           else {

                $PatchName=$FileName.name.TrimEnd(".msi") + "_"+ $shortversion[0]+  $shortversion[1] +
                $shortversion[2] ;
           }
                return $PatchName;
           }
           elseif ($PatchName.length -gt 0){
                #second part of the patch name
                $tmp = "_"+ $shortversion[0]+  $shortversion[1] + $shortversion[2] + ".msp";
                return $PatchName.Trim() + $tmp.Trim();
           }
           else {$PatchName='MyPatch.msp'}
      }
#input parameters: Target filename and Upgrade filename
$ScriptPath = split-path $MyInvocation.mycommand.path
echo "entering " $MyInvocation.ScriptName " with Args" $args[0] " and " $args[1] $TargetFile=$args[0];
$UpgradeFile= $args[1];
$SQLCommand='';
[string]$script:PatchName='';
#if($TargetFile.Name -ne $UpgradeFile.Name){
#   echo 'Target and Upgrade files have different names!' `n  'generation skipped'
#  continue
#}
#verify ProductVersion property; if both are = then skip rest of process
$SQLCommand= "select `Value` from `Property` where `Property`.`Property` = 'ProductVersion'";
[string]$productVersionTarget =  cscript /nologo .\WiRunSQL.vbs  $TargetFile $SQLCommand;
[string]$productVersionUpgrade =  cscript /nologo .\WiRunSQL.vbs  $UpgradeFile $SQLCommand;
if ($productVersionTarget.equals($productVersionUpgrade)){


      echo "both target and upgrade MSIs have the same version, there's nothing to update";
      continue;

}
# the script does the following:
#######################################################
#Prepare a temp working folder
#######################################################
if (test-path temp) {

      del -recurse -force temp

}
mkdir temp
mkdir temp\target
mkdir temp\upgrade
$TempDir=[string]$PWD + '\temp'
#######################################################
#Prepare input MSIs
#######################################################
#make productCode the same for target and upgrade files
$SQLCommand= "select `Value` from `Property` where `Property`.`Property` = 'ProductCode'";
$productCode -as [string]
$productCode =  cscript /nologo .\WiRunSQL.vbs
$TargetFile $SQLCommand;
#$SQLCommand= "update `Property` set `Value` = '" + $productCode[0].Trim() +
             "' where `Property`.`Property` = 'ProductCode'";
#echo "-->Running SQL on $UpgradeFile : $SQLCommand"
#cscript /nologo .\WiRunSQL.vbs  $UpgradeFile $SQLCommand;
#Assert that ProductVersions contain only 3 fields.
$PatchName=ModifyVersion $TargetFile $PatchName=ModifyVersion $UpgradeFile
```

```
#$PatchName = "output.msp"
$UpgradeVersion -as [string]
$UpgradeVersion=GetVersion $UpgradeFile
$UpgradeVersion=([string]$UpgradeVersion).Trim()
######################################################
#Unpack MSIs
######################################################
echo "Current dir ==== $PWD"
if ($TargetFile.gettype().Name.Equals("String")) {

      $tmp= $TargetFile.split("\");
      $tmp= $tmp[$tmp.length -1].TRimEnd(".msi");

}
else{

      $tmp= $TargetFile.Name;
      $tmp= $tmp.TRimEnd(".msi");

}
#echo $location2;
[string]$Installpath = $TempDir + "\Target\" + $tmp;
cmd /c msiexec /a $TargetFile  /qb TARGETDIR=$Installpath;
#now target file becomes unpacked msi
$TargetFile = Split-Path -leaf -path $TargetFile
$TargetFile = $TempDir + "\target\" + $tmp + "\" + $TargetFile
if ($UpgradeFile.gettype().Name.Equals("String"))
{

      $tmp= $UpgradeFile.split("\");
      $tmp= $tmp[$tmp.length -1].TRimEnd(".msi");

}
else{

      $tmp= $UpgradeFile.Name;
      $tmp= $tmp.TRimEnd(".msi");

}
$Installpath = $TempDir + "\upgrade\" + $tmp;
cmd /c msiexec /a $UpgradeFile /qb TARGETDIR=$Installpath;
#now upgrade file becomes unpacked msi
$UpgradeFile = split-path -leaf -path $UpgradeFile
$UpgradeFile = $TempDir + "\upgrade\" + $tmp + "\" + $UpgradeFile;
######################################################
#Fill  the PCP file
######################################################
#copies an empty pcp file from the sample from SDK
$pcpFile = $ScriptPath + '\template_base.pcp';
Copy-Item  -path $pcpFile -Destination ($TempDir + '\template.pcp')
$pcpFile = $TempDir + '\template.pcp'
#Modify Properties Table
$SQLCommand= "update Properties set Properties.Value ='"+ $PatchName.TRim() +"' " +
             " where Properties.Name= 'PatchOutputPath'";
#echo "-->Running SQL on $pcpFile : $SQLCommand"
cscript /nologo .\WiRunSQL.vbs  $pcpFile  $SQLCommand;
[string]$guid= "{" +  [system.Guid]::NewGuid().ToString() + "}";
$GUID= $guid.Trim();
$SQLCommand= "update Properties set Properties.Value ='"+ $guid.ToUpper() +"'" +
           " where Properties.Name= 'PatchGUID'";
cscript /nologo .\WiRunSQL.vbs  $pcpFile  $SQLCommand;
#Modify TargetImages
Table $SQLCommand= "insert into TargetImages (`TargetImages`.``Order`` , `
                  `TargetImages`.`Target`,`TargetImages`.`MsiPath`," +
           "`TargetImages`.`Upgraded`,`TargetImages`.`IgnoreMissingSrcFiles`,
`TargetImages`.`ProductValidateFlags`) " +
"values(1,'previous','" + $TargetFile + "','newer', 0, '0x00000922')";
echo "-->Running SQL on $pcpFile : $SQLCommand"
cscript /nologo .\WiRunSQL.vbs $pcpFile $SQLCommand;
$fam='family1'
if ($TargetFile.Name.Length -ge 1) {

      $fam= $TargetFile.Name
      $fam= $fam.TrimEnd(".msi");
      $fam= $fam.TrimsTART("pms_");
      if ($Fam.length -gt 8) {
           $fam= $fam.substring(0,8)
      }
}
#Modify ImageFamilies Table
[int]$SeqNum= getLargestSeq($TargetFile, $UpgradeFile); #$lastSequence[0];
$SeqNum= $SeqNum + 1;
$SQLCommand="insert into ‘ImageFamilies‘(‘Family‘,‘MediaSrcPropName‘,‘MediaDiskId‘,‘FileSequenceStart‘)
             VALUES (’" + $fam + "’, ’"+ $fam +"_MediaSrcPropName’, 2," + $SeqNum +")";
echo "-->Running SQL on $pcpFile : $SQLCommand"
cscript /nologo .\WiRunSQL.vbs $pcpFile  $SQLCommand
#Modify UpgradedImages Table
$SQLCommand= "insert into UpgradedImages (‘Upgraded‘,‘MsiPath‘,‘PatchMsiPath‘,‘SymbolPaths‘,‘Family‘) " ‘
            + " values (’newer’,’" + $UpgradeFile +"’,”,’ ’, ’" + $fam + "’)"; #" + (Get-Location) +"
echo "-->Running SQL on $pcpFile : $SQLCommand"
cscript /nologo .\WiRunSQL.vbs $pcpFile $SQLCommand
#######################################################
#create a MSP file
#######################################################
echo ’creating patch...’ $OutputDir= ’..\msp’;
if ((Test-Path $OutputDir) -eq $false) {

      mkdir $OutputDir;

}
$logFile= (Get-Date).Year.ToString() +            (Get-Date).Month.ToString() +
           (Get-Date).Day.ToString() +            ’_’ +(Get-Date).hour.ToString() +
           (Get-Date).minute.ToString() +’.log’;
$logFile = $TempDir + ’\MSP.log’ $retval -as [string]
Msimsp.exe -s $pcpFile -p ($OutputDir + ’\’ + $PatchName.Trim()) -l $logFile
if ( $? ) {

      echo ’done!’

}
else  {

      echo ’generation failed; check log’;
      type $logFile | find ’"ERROR"’

}
```

CallHashTools.ps1

```
#######################################################
#script to compute MD5 values for a set of files administratively installed
# using msiexec.exe the computation done is Allura-system wide.
# This script assumes there are two MAIN ’root’ directories:
#a target directory which contains all the subsystem directories
#that belong to the base version, and #an upgrade dir containing all the subsystem directories
#that belong to the upgrade version #examples of usage are:
#callHashtools c:\temp\target c:\temp\upgrade
#######################################################
[System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
clear;
echo (Get-Date).DateTime
#evaluate input arguments
if ($args[0] -eq $null -or $args[1] -eq $null){

      echo ’missing arguments; example:’;
      echo ’callHashTools.ps1 <Target_Root_Directory> <Upgrade_Root_Directory>’;
      break;

}
#compute MD5 values for the MSP files
$option= Read-Host ’compute MD5 values for MSP files? (y/n):’
if($option -eq ’y’ -or $option -eq ’Y’){

      .\MD5s4MSPs.ps1

}
$dirsTarget = dir $args[0] | Where {$_.psIsContainer -eq $true} | sort;
$dirsUpgrade = dir $args[1] | Where {$_.psIsContainer -eq $true} | sort;
if ($dirsTarget.length -ne $dirsUpgrade.Length){


      [System.Windows.Forms.MessageBox]::Show("Both Target and Upgrade directories ‘
      must have = number of directories")
      break;
}
$index=0;
foreach ($item in $dirsTarget){
#here $item represents the sub-directory where the admin install subsystem is
#.\ComputeMD5s.ps1 $item.FullName $dirsUpgrade[$index].FullName;
$msi= Get-ChildItem $item.FullName *.msi;
if ($msi -ne $null){

      .\MD5PerFeature.ps1 $msi.FullName $item.FullName -t
      $msi= Get-ChildItem $dirsUpgrade[$index].FullName *.msi;
      .\MD5PerFeature.ps1 $msi.FullName  $dirsUpgrade[$index].FullName -u
      $index++;

}
}
echo  'computation ended at ' (Get-Date).DateTime
```

MD5s4MSPs.ps1

```
[System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
#compute MD5 values for the MSP files #example of use:
# MD5s4MSPs.ps1 <valid_path_with_MSPs>
clear;
[string]$OutputDir= '..\XML\';
echo (Get-Date).DateTime
$MSPsPath= $args[0];
if ($args[0] -eq $null) {


      $MSPsPath= Read-Host 'input the directory containing the generated MSPs ';

}
if ($MSPsPath -ne $null -and (Test-Path -Path $MSPsPath)){

      $MSPs= Get-ChildItem -Path $MSPsPath -Include *.msp;
      echo 'creating xml database';
      if((Test-Path -path .\fciv.err) -eq $true) {
           del .\fciv.err;
      }
      cmd /c .\fciv.exe -wp $MSPsPath -wp -type *.msp -MD5 -xml ($OutputDir + 'MSPs.xml')
      echo 'checking if there are errors...'
      if((Test-Path -path .\fciv.err) -eq $true) {
           $errors =cmd /c findstr.exe /n "Error msg" fciv.err;
      }
      if ($errors -ne $null){
           echo $errors }
      else{
           echo 'computation of MD5 values for MSPs succeeded';
      }

}
else{


      [System.Windows.Forms.MessageBox]::Show("Path does not exist", "InvalidPath");

}
echo 'computation ended at ' (Get-Date).DateTime
```

MD5PerFeature.ps1

```
#script to compute MD5 values based on features.
#this script is called from callHashTools.ps1
# it receives the path of a given MSi administratively installed
# it also creates a file to store the files (to be installed) + their timestamp
##############################################################################
# known issues:
# if the filename is > 260 characters the gci cmdlet will fail; example:
# FSCGeoCVLArcBeamLongitudinalPotentiometerAdjustmentParamDic.xml
# Get-ChildItem : The specified path, file name, or both are too long. The fully
# qualified file name must be less than 260 characters, and the directory name must
#be less than 248 characters.
# see line 76 char:30
##############################################################################
###############################################################################
# validate args
###############################################################################
if($args[0] -eq $null -or $args[1] -eq $null -or $args[2] -eq $null) {
        echo 'missing arguments; example of use:'
        echo 'MD5PerFeature.ps1 <fileName.msi> <path_Admin_install> <suffix: -t or -u>'
        break;

}
function GetVersion($MSIName){


        $SQLCommand= "select `Value` from `Property` where `Property`.`Property` = 'ProductVersion'";
        $productVersion -as [string]
        return cscript /nologo .\WiRunSQL.vbs  $MSIName $SQLCommand;

}
function Replace-String($find, $replace, $path)       {


        echo "Replacing string `"$find`" with string `"$replace`" in file contents and file names `
              of path: $path"
        ls $path | select-string $find -list |% { echo "Processing contents of $($_.Path)";
        (get-content $_.Path) |% { $_ -replace $find, $replace } | set-content $_.Path -Force }
        #ls $path\*$find* |% { echo "Renaming $($_.FullName) to $($_.FullName.Replace($find, $replace))";
        mv $_.FullName $_.FullName.Replace($find, $replace) }

}
function ReassemblePath($path){

        $newPath='';
        if ($path.length -ge 1) {
             foreach ($item in $path)  {
                  $newPath= $newpath + $item +  "\\";
             }
                 return $newPath;
        }

}
function getPath($component){

        $SQLcommand= "Select Directory_Parent from Directory, Component "  +
                     " where Component.Directory_ = Directory.Directory "  +
                     " and Component.Component ='" + $component.Trim() + "'";
        return cscript.exe /nologo .\WiRunSQL.vbs $db $SQLCommand;

}
function getUnique($found, $filePath){

        foreach ($item in $found){
             $tmp= $item.directoryName.Tolower();
             if ($tmp.Contains($filePath[0].ToLower()) -eq $true){
                  return $item;
             }
        }

}
 function getGacPath ($found)  {

        $actualPath= $env:windir + '\assembly\GAC';
        #find version, name and public key token
        $info = [System.Reflection.Assembly]::LoadFrom($found.FullName);
        $Assemblyname= $info.GetName().Name;
        $version = $info.GetName().Version.ToString();
        $PKT = $info.GetName().GetPublicKeyToken();#public key token
        #iterate to convert PKT from hexa to decimal
        $decimalPKT= '';
        foreach ($byte in $PKT){


             $decimalPKT = $decimalPKT  + [Convert]::ToString($byte, 16)
        }
        #assemble path
        $actualPath = $actualPath + '\' `
                     + $Assemblyname + '\' `
                     + $version + '\' `
                     + $decimalPKT ; # + '\'
                     #+ $found.Name;
        return $actualPath;
```

```
}
function ComputeMD5($files) {

      [string]$values ='';
      $defaultAdd= 10000000;
      #find the file
      foreach ($file in $files){
      #separate file and component
      $file = $file.split(" ");
      $component= $file[$file.length -1];
      $file= $file[0];
      #remove the piping stuff
      if ($file.Contains('|')) {
           $tmp= $file.split('|');
           $file = $tmp[$tmp.length -1].Trim();
           #echo 'after removing |:' $file
      }
      echo $file;
      $found = $null;
      if ($file -eq $null -or $file.Length -le 0) {continue;}
      $found= get-childitem -Path $path $file -recurse
      if ($found.GetType().BaseType.Name.Equals('Array') -and ($found -ne $null)){
           # more than 1 file Found with the same name,
           # we just want the one that matches the component
           #echo $found.GetType();
           $component = getPath($component);
           $found = getUnique $found $component;
      }
      echo 'computing MD5 for' $found.fullname #`t $tmp[$tmp.length -1].Trim();
      # rebase files to default address and timestamp
      if ($found -ne $null -and $found.Length -gt 0){
           $tmp= $found.Extension.ToUpper();
           # only files located in 'program files\pms\fusion' directory are (un)rebased
           if (($tmp.Contains(".SDM") -or $tmp.Contains(".OCX") -or $tmp.Contains(".DLL")) `
                 -and ($found.DirectoryName.Contains('Windows') -eq $false)
                 -and ($found.DirectoryName.Contains('Fusion') -eq $true)) {

                #Add-Content  -Value $values -Path $timeStampsFile;
                .\RebaseDLL $found.fullName $defaultAdd;
                      }
           #compute its MD5 value
           .\fciv.exe -add $found.FullName -xml $XMLfile;
           if ($found.Directory.FullName.Contains('GlobalAssemblyCache') -eq $true){

                # replace globalassemblycache in the path, for the actual path
                $tmp= getGacPath $found;
                Replace-String "c:\\GlobalAssemblyCache" $tmp $XMLfile;
           }
      }

}
}
$tmp=$args[2]; $XML_suffix= '_Target';
switch ($tmp) {

      -t {
           $XML_suffix= '_Target';
      }
      -u {
           $XML_suffix= '_Upgrade';
      }

}
[string]$OutputDir= '..\XML\';
if ((Test-Path $OutputDir) -eq $false) {

      mkdir $OutputDir;

}
$DB = $args[0];
$path = $args[1];
$tmp= $db.split('\');
[string]$version= GetVersion $db;
```

```
$version= "_" + $version.Trim();
$XMLfile= $OutputDir + ($tmp[$tmp.length-1].trim('.msi') + $version + $XML_suffix + '.xml');
#create the .csv file that contains timestamps for rebasable files.
$timeStampsFile= $OutputDir +  ($tmp[$tmp.length-1].trim('.msi') + $version + '_ts.csv');
#use comma delimited files; later decide to use another format
$tmp = "file,second,minute,hour,day,month,year";
#set-Content  -Value $tmp -Path $timeStampsFile;
#select features from the MSI
$SQLCommand = 'select `Feature` FROM `Feature` order by `Feature`';
$Features= cscript.exe  /nologo .\WiRunSQL.vbs $db $SQLCommand
foreach($feat in $features){

      IF ($feat.Contains('_PROD')){#echo $feat

           $SQLCommand = "select `File`.`FileName`, `File`.`Component_` from `File`," + #`File`.`File`,
                             "`Component`,`FeatureComponents` where `File`.`Component_` " +
                             "= `Component`.`Component` and "  +
                             "`FeatureComponents`.`Component_` = `Component`.`Component` "  +
                            "and `FeatureComponents`.`Feature_`= '" + $feat +
                            "' order by `File`.`FileName` ";
           $Files= cscript.exe  /nologo .\WiRunSQL.vbs $db $SQLCommand;
           echo $files.Length;
           ComputeMD5 $files;
      }

} # remove the files base path; this path won't be used when verifying the files
# in the actual system
$chopped= $path.split('\');
$tmp= ReassemblePath $chopped;
Replace-String $tmp.toLower() "c:\" $XMLfile;
Replace-String "c:\\winroot" "c:" $XMLfile;
```

ComputeMD5s.ps1

```
#Script to create target and Upgrade xml containing MD5 values
#The computation is generated both for target and upgrade directories, which are
#provided as arguments to the script.
# call example:
# ComputeMD5s.ps1 D:\images\PMS_Target D:\images\PMS_Upgrade
function Replace-String($find, $replace, $path)      {


      echo "Replacing string `"$find`" with string `"$replace`" in file contents and file names `
            of path: $path"
      ls $path | select-string $find -list |% { echo "Processing contents of $($_.Path)";
      (get-content $_.Path) |% { $_ -replace $find, $replace } | set-content $_.Path -Force }
      ls $path\*$find* |% { echo "Renaming $($_.FullName) to $($_.FullName.Replace($find, $replace))";
      mv $_.FullName $_.FullName.Replace($find, $replace)
      }

}
function GetVersion($MSIName){


      $SQLCommand= "select `Value` from `Property` where `Property`.`Property` = 'ProductVersion'";
      $productVersion -as [string]
      return cscript /nologo .\WiRunSQL.vbs  $MSIName $SQLCommand;

}
function ReassemblePath($path){

      $newPath='';
      if ($path.length -ge 1)     {
           foreach ($item in $path)        {
                $newPath= $newpath + $item +  "\\";
           }
           return $newPath;
      }

}
########################################################
#get the subsystem names -from the MSI files
########################################################
#echo (Get-Date)
$TargetF = Get-ChildItem $Args[0] -recurse -include *.msi | sort;
```

```
#get the target MSI FileName
$UpgradeF= Get-ChildItem $Args[1] -recurse -include *.msi | sort;
#get the upgrade MSI FileName
if ($TargetF -eq $null -or $UpgradeF -eq $null){

      echo 'missing arguments; computation skipped:';
      exit;

}
if ($TargetF.getType().Name -eq "FileInfo") {

      $Target = $TargetF

}
else {

      $Target = $TargetF[0]

}

      if ($UpgradeF.getType().Name -eq "FileInfo") {
      $Upgrade = $UpgradeF

}
else {

      $Upgrade = $UpgradeF[0]

}
#[string]$TargetXML= $target.DirectoryName + "\" + $target.Name.TrimEnd("msi") + "xml"
#[string]$UpgradeXML= $Upgrade.DirectoryName +  "\" + $Upgrade.Name.TrimEnd("msi")  + "xml"
[string]$version= GetVersion $Target.fullName; $version= "_" + $version.Trim();
$TargetName = $Target.name.TrimEnd(".msi") + $version.Trim() + "_Target.xml";
$version=  GetVersion $Upgrade.fullName; $version= "_" +$version.Trim();
$UpgradeName = $Upgrade.Name.TrimEnd(".msi") + $version.Trim()  + "_Upgrade.xml";
[string]$OutputDir= '..\XML\';
if ((Test-Path $OutputDir) -eq $false) {

      mkdir $OutputDir;}
      [string]$TargetXML= (Get-Location).ToString() + "\" + $TargetName
      [string]$UpgradeXML= (Get-Location).ToString() +  "\" + $UpgradeName

########################################################
#call the MD5 computation
########################################################
echo 'generating MD5 values for' $args[0]
#cmd /c .\fciv.exe -r $args[0] -md5 -xml $TargetXML;
#select features from the MSI $SQLCommand = 'select `Feature` FROM `Feature` order by `Feature`';
$Features= cscript.exe  /nologo .\WiRunSQL.vbs $Target $SQLCommand
echo 'generating MD5 values for ' $args[1] cmd /c .\fciv.exe -r $args[1] -md5 -xml $UpgradeXML;
########################################################
#verify there were no errors
########################################################
echo 'displaying errors (if any)...' cmd /c findstr.exe /n "Error msg" fciv.err
$UnfilteredTarget= $TargetName; $UnfilteredUpgrade= $UpgradeName;
########################################################
#replace the base path
########################################################
echo 'removing basepath directory from xml files...';
#cmd /c .\replace.exe -find $args[0] -replace $tmp -fname $UnfilteredTarget;
#cmd /c .\replace.exe -find $args[1] -replace $tmp -fname $UnfilteredUpgrade;
[string]$tmp= $args[0];
$chopped= $tmp.split('\');
$tmp= ReassemblePath $chopped;
Replace-String $tmp "c:\" $UnfilteredTarget;
$tmp= $args[1];  $chopped= $tmp.split('\');
$tmp= ReassemblePath $chopped; Replace-String $tmp "c:\" $UnfilteredUpgrade;
########################################################
#replace winroot -if present in XML
########################################################
echo 'replacing "Winroot" from xml files...';
#cmd /c .\replace.exe -find Winroot -replace $tmp -fname $UnfilteredTarget;
#cmd /c .\replace.exe -find Winroot -replace $tmp -fname $UnfilteredUpgrade;
Replace-String "c:\\WinRoot" "c:\" $UnfilteredTarget;
Replace-String "c:\\WinRoot" "c:\" $UnfilteredUpgrade;
########################################################
#use the xslt filter to exclude test and stub  files
########################################################
echo 'excluding test and stub  files...';
echo cmd /c .\msxsl.exe $UnfilteredTarget `
.\infraFilter.xslt ($OutputDir + $TargetName.TrimEnd(".xml") +
       "Filtered.xml"); cmd /c .\msxsl.exe $UnfilteredTarget .\infraFilter.xslt -o ($OutputDir +
       $TargetName.TrimEnd(".xml") + "Filtered.xml");
echo cmd /c .\msxsl.exe $UnfilteredUpgrade `
.\infraFilter.xslt  ($OutputDir + $UpgradeName.TrimEnd(".xml") +
       "Filtered.xml"); cmd /c .\msxsl.exe $UnfilteredUpgrade .\infraFilter.xslt -o ($OutputDir +
       $UpgradeName.TrimEnd(".xml") + "Filtered.xml");
########################################################
#remove unfiltered .xml files
########################################################
del $UnfilteredTarget del $UnfilteredUpgrade
#echo (Get-Date)
```

unrebase.ps1

```
#rebase files
#####################################################
# verify argument
#####################################################
if ($args[0] -eq $null){

      write-host "no input directory provided"
      write-host "example of use:" ".\unrebase.ps1 <Allura_install_directory>" exit

}
$dlls = gci $args[0] -Recurse -Include *.dll;
$ocx = gci $args[0] -Recurse -Include *.ocx;
$sdm = gci $args[0] -Recurse -Include *.sdm;
$defaultAdd= 10000000;
write-host `t "unrebasing " $dlls.length " DLLs..." -nonewline
foreach($ITEM IN $dlls){ .\RebaseDLL.exe $item.FullName $defaultAdd  }
write-host "done."
write-host `t "unrebasing " $ocx.length " OCXs..." -nonewline
foreach($ITEM IN $ocx){ .\RebaseDLL.exe $item.FullName $defaultAdd #10 10 10 17 07 2009 }
write-host "done."
write-host `t "unrebasing " $sdm.length " SDMs..." -nonewline
foreach($ITEM IN $sdm){ .\RebaseDLL.exe $item.FullName $defaultAdd #10 10 10 17 07 2009 }
write-host "done."
```

InstallDeltas.ps1

```
#####################################################
#install the MSPs (upgrade version) #this script takes as inputs:
# A) the directory where the patches (MSP files) are
# B) the BOM_MSP file that prescribes the installation order of patches
# C) the execution mode: -i for patching, and -u for patching removal
# example of use:
# InstallDeltas.ps1 <MSP_Files_Directory> <BOM_MSP_file> <execution_mode>
#####################################################
#####################################################
# function to write to a pre/post patching log
#####################################################
function WriteVerifyLog ($message, $logFile) {

      [string]$m=$message;
           if ($m.contains('Unable') -or $m.contains('should be')) {


               Add-Content -Value $message -Path $LogFile; return $false
           } else{
           return $true;
           }

}
#####################################################
# logging variables
####################################################
$logDirectory= '..\Log\';
if ((Test-Path $logDirectory -PathType container) -eq $false){

      New-Item $logDirectory -type directory;
```

```
}
$logPreInstall= $logDirectory + 'logPreInstall.txt';
$LogInstall= $logDirectory + 'LogInstall.txt'; #store return codes from msiexec.
$logPostInstall= $logDirectory + 'logPostInstall.txt';
####################################################
# function to verify the system is in a known working
# state from the cofiguration point of view.
# this function verifies the following:
# a) the MSP files to install have not been tampered
# b) the system to be updated contains the right configuration
# not only subsystem level, but even file level. For this verication
# we assume the xml files are in the same directory where the MSP
# files are
####################################################
function verifyPreInstall{

      $targetSuffix= '*target*.xml';
      $success= $true;
      $tempSuccess= $success;
      write-host "performing pre-install validation...";
      #verify the MSP files
      cd $MSPsPath;
      write-host `t 'Validating MSP files...' -nonewline;
      $result = cmd /c $installPath\fciv.exe -v -xml $XMLPath\MSPs.xml
      $success= WriteVerifyLog $result $logPreInstall
      if ($success -eq $false){

           write-host -ForegroundColor Red "MSPs corrupted; validation failed. `
                         Exiting the installation script"; exit;
      }
      else {
           write-host -ForegroundColor Green "MSP files ok";
      }
      cd $installPath;
      #read from the XML directory all target XML files
      $targetList= Get-ChildItem $XMLPath\$targetSuffix
      write-host 'verifying subsystems: '
      foreach($item in $targetList){
           write-host `t $item.Name.TRimEnd(".xml") '...' -nonewline;
           Add-Content -Value ('verifying subsystem ' + $item.Name.TRimEnd(".xml")) `
                       -Path $logPreInstall;
           $result= cmd /c .\fciv.exe -v -xml $item
           $tempSuccess = WriteVerifyLog $result $logPreInstall;
           if($tempSuccess -eq $false) {

                $success = $tempSuccess; write-host -ForegroundColor Red 'failed';
           }
           else{ write-host -ForegroundColor Green 'OK'; }
      }
      return $success;

}
####################################################
# function to verify system after update
####################################################
function verifyPostInstall{

      $UpgradeSuffix= '*upgrade*.xml';
      $success= $true;
      $beginDate= (Get-Date);
      write-host 'validating installation of MSPs...' $beginDate.ToLongTimeString();
      #read from the XML directory all upgrade XML files
      $targetList= Get-ChildItem $XMLPath\$UpgradeSuffix
      foreach($item in $targetList){
           write-host `t $item.Name.TRimEnd(".xml") '...' -NoNewline;
           Add-Content -Value ('verifying subsystem ' + $item.Name.TRimEnd(".xml")) `
                       -Path $logPostInstall;
           $result= cmd /c .\fciv.exe -v -xml $item
           $tempSuccess = WriteVerifyLog $result $logPostInstall;
           if($tempSuccess -eq $false) {
                $success = $tempSuccess;
                write-host -ForegroundColor Red 'failed';
           }
           else{
                write-host -ForegroundColor Green 'ok';
            }
        }
        return $success;

}
####################################################
# function to retrieve the Data for MSP removal
####################################################
function getRemovalData($FileFound){

        [System.Array] $AllProps, $properties;
        $AllProps = MsiInfo.Exe $FileFound; $i=0;
        if ($AllProps -ne $null){
            foreach($p in $AllProps){
                if ($p.contains('/p') -or $p.contains('/v')){
                  write-host 'to be done' }
                }
            }
        }

####################################################
#function to rollback the installation
#this function is responsible for removing the patches previously installed.
#it uses as reference the BOM_SMP.txt file used for installation, but read in
#inverse order
####################################################
function rollback{

        write-host rollbacking...
        for($i=$bom_msp.length -1; $i -ge 0; $i--){
            if ($bom_msp[$i].toString().Equals("")){continue;}
            $fileName= $bom_msp[$i];
            $filename= $filename.split(':');
            $FileFound= Get-ChildItem -path $MSPsPath $filename[$filename.length-1].Trim();
            if ($FileFound.length -ne $null ){

                #$retCode= Msiexec /package <> /uninstall $FileFounf.FullName /passive
                write-host 'removing patch...'
                $MSIData = getRemovalData $FileFound;
                $retoCode = Msiexec /I $MSIData[0] MSIPATCHREMOVE=$MSIData[1] /qb `
                /l*v ($logDirectory + $result.Name.trimEnd(".msp") + "_removal.log");
            }
            if ($retCode -eq 0) { continue }
            #successful removal
            else {
                # something went wrong during removal; log
                set-content -path $LogInstall -value $retCode
                return 1; #return to Questra
            }
        }

}
####################################################
# this function installs the set of msp files in the
# order indicated by the BOM_MSP.txt file
####################################################
function install {

        [string]$fileName;
        [bool]$result;
        for($i=0;$i -le $bom_msp.length -1; $i++){
            if ($bom_msp[$i].toString().Equals("")){continue;}
            $fileName= $bom_msp[$i];
            $filename= $filename.split(':');
            $result= Get-ChildItem -path $MSPsPath $filename[$filename.length-1].Trim();
            if ($result.length -ge 0 ){

                $retCode= cmd /c msiexec /p $result.FullName REINSTALL=ALL REINSTALLMODE=amus /qb `
                        /l*v ($logDirectory + $result.Name.trimEnd(".msp") + "_install.log");
                #$filename[$filename.length-1]
            }
            if ($retCode -eq $null) { #patching succeeded.

                set-content -path $LogInstall -value ('patching of ' + $result.Name
                            + ' succeeded');
                continue;
            } else {
                write-host $retCode;
                set-content -path $LogInstall -value ('patching of ' + $result.Name + `
                    ' failed; error code: ' + $retCode); rollback
            }
        }#foreach

}
#######################################################
# starting point of execution
#######################################################
clear;
write-host 'installation of delta packages begins at' (Get-Date).ToLongTimeString();
#evaluate input arguments
if ($args[0] -eq $null -or $args[1] -eq $null -or $args[2] -eq $null){

    write-host 'missing arguments; example:';
    write-host 'InstallDeltas.ps1 <MSP_Files_Directory> <BOM_MSP_file> <execution_mode>';
    write-host 'execution mode is either "-i" for install, or "-r" for rollback' break;

}
$MSPsPath= $args[0]; #Read-Host 'input the directory containing the MSPs'
$installPath= (Get-Location); $XMLPath= '..\XML';
[string]$p=$args[0];
#######################################################
#testing parameters
#######################################################
if((Test-Path -path $p) -eq $false) {

    write-host "MSP directory not found: " $args[0]; break;

}
$p=$args[1];
if((Test-Path -path $p) -eq $false) {

    write-host "BOM_MSP file not found: " $args[1]; break;

}
$bom_MSP= get-content -path $args[1] #FileName to read the installation order
[string]$command= $args[2];
$list= Get-ChildItem $MSPsPath -recurse -include pms*.msp
switch ($command) {

    #install MPS
    -i {
        #verify pre-requirements #unrebase the rebased files, so that
        # verify pre-install passes MD5 checking
        write-host 'unrebasing files...' (Get-Date).ToLongTimeString();
        Set-Content -Value ("unrebasing files begins..." + `
                  (Get-Date).ToLongTimeString()) -Path $LogPreInstall;
        .\unrebase.ps1 'C:\Program Files\PMS\Fusion' >$logDirectory\Unrebase.log;
        Add-Content -Value ("unrebasing files ends..." + (Get-Date).ToLongTimeString())
                   -Path $LogPreInstall;
        # get-history -count 1|fl @{l="duration of unrebasing: ";
        e={$_.endexecutiontime - $_.startexecutiontime}};
        $beginDate= (get-date);
        write-host 'pre-patch validation begins at ' $beginDate.ToLongTimeString();
        $PreInstallVal= verifyPreInstall;
        $timediff= (Get-Date).Subtract($beginDate);
        write-host 'pre-patch validation ends at ' (Get-Date).ToLongTimeString();
        write-host `t 'Duration of pre-patch validation: ' $timediff.Minutes ' minutes, ' `
            $timediff.Seconds ' seconds';
        if ($PreInstallVal -eq $true){
            write-host 'pre install validation succeeded!';
            $beginDate= (get-date);
            write-host 'installation begins at ' $beginDate.ToLongTimeString();
            install
            $timediff= (Get-Date).Subtract($beginDate);
            write-host 'installation ends at ' (Get-Date).ToLongTimeString();
            write-host `t 'Duration of installation: ' $timediff.Minutes ' minutes, '  `
                $timediff.Seconds ' seconds' -ForegroundColor Cyan;
```

```
           }
           else { #prepatching failed
                write-host "Pre-install validation failed.";
                #exit;
                $beginDate= (get-date);
                write-host 'installation begins at ' $beginDate.ToLongTimeString();
                install
                $timediff= (Get-Date).Subtract($beginDate);
                write-host 'installation ends at ' (Get-Date).ToLongTimeString();
                write-host `t 'Duration of installation: ' $timediff.Minutes ' minutes, ' `
                           $timediff.Seconds ' seconds' -ForegroundColor Cyan;
           }

#verify after installation
$beginDate= (get-date);
write-host 'post-patch validation begins at ' $beginDate.ToLongTimeString();
$postPatchVerification= verifyPostInstall;
$timediff= (Get-Date).Subtract($beginDate);
write-host 'installation ends at ' (Get-Date).ToLongTimeString();
write-host `t 'Duration of post-patch validation: ' $timediff.Minutes ' minutes, ' `
    $timediff.Seconds ' seconds' -ForegroundColor Cyan;
if ($postPatchVerification -eq $true){

      write-host 'installation succeded';
      return 0; #return to Questra

}
else {

      rollback;
      return 2; #return to Questra
      };

} #-i
#remove MSPs
-u {

      rollback;
      }

}
write-host 'installation of delta packages ends at' (Get-Date).ToLongTimeString();
```

Bibliography

[1] Wikipedia. (2009, Aug) Md5. web site. Wikimedia Foundation. [Online]. Available: http://en.wikipedia.org/wiki/Md5 XI, 45

[2] P. Langemeijer, "Installation rocket software. software unit design specification," Philips internal document, Jun 2004, document XDB-064-04-0084. 1, 34, 64, 80

[3] Microsoft. (2008, May) Windows software development kit documentation. .chm file. Microsoft. [Online]. Available: http://www.microsoft.com/downloads/details.aspx?FamilyID=5a58b56f-60b6-4412-95b9-54d056d6f9f4&displaylang=en 1, 2, 44, 53, 58, 60, 81, 84, 85

[4] Acresso. (2009, Apr) Installshield. how the world installs software on microsoft windows. website. Acresso Software. [Online]. Available: http://www.acresso.com/products/is/installshield-overview.htm 2

[5] N. Rijnierse, Kees; Barendse, "Functional requirements specification. software distribution and installation." word document, Oct 2008, document of Philips. Document XDY040-072398. 3, 33, 34, 35, 36, 38, 58, 59, 60, 76

[6] J. Vugts, "Project proposal form," PDF file, Oct 2008. 6, 16

[7] A. Cribier, "Percutaneous transcatheter implantation of an aortic valve prosthesis for calcific aortic stenosis," American Heart Association, Service de Cardiologie, Hôpital Charles Nicolle, 1 rue de Germont, 76 000, Rouen, France, Special Report Report number 106:3006, Nov 2002, electronic version browsed on 28/10/2009. 6

[8] O. Mendez, "Installation process overview," Excel sheet, Mar 2009, internal document of Philips. 8, 16, 26

[9] FDA. (2009, Apr) Cfr - code of federal regulations title 21. website. U.S. Food and Drug Administration. Food and Drug Administration 10903 New Hampshire Ave Silver Spring, MD 20993-0002. [Online]. Available: http://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfcfr/CFRSearch.cfm?fr=820.70 8, 27

[10] A. Visser, "Project and software merge strategy," Powerpoint presentation, Oct 2009, internal document of Philips. XCX610-094337. 8

[11] D. Davis. (2009, Oct) Danny davis quotes. website. ThinkExist. ThinkExist.com. [Online]. Available: http://en.thinkexist.com/quotes/danny_davis/2.html 11

[12] Wikipedia. (2009, Feb) 5 whys. website. Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/5_Whys 17

[13] O. Mendez, “Selection of focal points,” Excel sheet, Mar 2009, internal document of Philips. 21

[14] FDA. (2009, Apr) Radiation-emitting products. website. U.S. Food and Drug Administration. 0903 New Hampshire Avenue WO66-4613 Silver Spring, MD 20993. [Online]. Available: http://www.fda.gov/Radiation-EmittingProducts/default.htm 27

[15] H. Thelosen, “Configuration management process training,” PowerPoint document, Oct 1999. 28

[16] H. M. T. Tinus, *Maintenance Process. Complaint Handling Procedure*, Philips Healthcare, internal document. Document identifier: XCV-0303201. 33

[17] G. Schroder. (2009, Oct) Feasibility quotations. website. BrainyMedia.com. cited in the URL. [Online]. Available: http://www.brainyquote.com/words/fe/feasibility164061.html 41

[18] QArchive. (2009, May) Windows installer patch. website. QArchive. References to COTS patching tools. [Online]. Available: http://windows-installer-patch.qarchive.org/ 42

[19] Wikipedia. (2009, Aug) Cyclic redundancy check. web site. Wikimedia Foundation. [Online]. Available: http://en.wikipedia.org/wiki/Cyclic_redundancy_check 45

[20] O. Mendez, “Allura partial installations. design specification,” Word document, Oct 2009, internal document of Philips. XCY610-094211. 51, 53

[21] K. V. Nagendra, “Delta package installation verification specification,” Word document, Aug 2009, internal Philips document. Document XCT-0308203 Ver 0 3. 63, 66, 80

[22] K. Nagendra, “Installation test report,” Philips Healthcare, Document XCT-0308204, Oct 2009, internal document of Philips. 63, 66, 80

[23] F. Louw, “System restore and backup,” Word document, Oct 2009, internal document of Philips. 66, 68, 86

[24] Aristotle. (2009, Oct) Aristotle quote. website. quotationsbook.com. Cited in the URL included. [Online]. Available: http://quotationsbook.com/quote/434/ 73

[25] Microsoft. (2009, Oct) Windows powershell. website. Microsoft Corporation. Visited on October 2009. [Online]. Available: http://www.microsoft.com/windowsserver2003/technologies/management/powershell/default.mspx 74

[26] O. Mendez, “Size comparison a partial install vs. full install sets,” excel file, Aug 2009. 82, 84

[27] P. van Kempen, “Ucm for rocket projects,” Philips internal document. Document XDN041-039028, Feb 2009. 84

About the Author

Orlando Mendez did his Bachelors in Informatics at the Technological Institute of Apizaco, Mexico (1995-1999). Afterwards he worked for 2.5 years at the Ministry of Finances in the Province of Tlaxcala, Mexico. From August 2002 until August 2004, he attained his Master's degree in Software Engineering at Delft University of Technology, the Netherlands. In 2005 Orlando did voluntary work in Chili teaching English at an elementary school. From February 2006 until September 2007 he worked developing software both as free-lancer and within some companies in Mexico City. In October 2007, Orlando joined the Software Technology program of Stan Ackermans Institute, in the Netherlands.