Faculdade de Engenharia da Universidade do Porto

Manufacturing Equipment Data Collection Framework

Daniel José Barbudo Aguilar

Report of Project
Master in Informatics and Computing Engineering

Supervisor: Maria Teresa Galvão Dias (PhD)

July 2008

Manufacturing Equipment Data Collection Framework

Daniel José Barbudo Aguilar

Report of Project

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: António Augusto de Sousa (PhD)

External Examiner: César Analide (PhD)

Internal Examiner: Maria Teresa Galvão Dias (PhD)

31^{st} July, 2008

Abstract

The Bee Framework project is essentially a data collection framework used to collect the data generated by equipments which intends to help improving the equipment integration process. Even considering different equipment types, there are several common data collection functionalities that can be easily reused just by doing a small amount of light changes or even no changes at all. Some of these functionalities are, for example, database access, folder monitoring actions, communication and data collection, among others.

The main goal of the project described in this report is the elaboration of a detailed specification for a data collection framework used to collect the data generated by several equipments in an assembly line. Additionally, the core requirements of the framework should also be implemented and an equipment integration should also be done as a proof of concept. This framework intends to control and monitor these equipments so that the data they generate can be collected. This data will be fed into data analysis solutions that provide the tools required to follow-up on a continuous process evolution and optimization and process control.

However, the wide diversity of existing equipments used in such assembly lines is a well-known problem that makes it hard to adopt a common integration solution. Moreover, a large number of these equipments do not follow some of the international standards for data collection and data control used in the semiconductor industry. These strong difficulties to adopt a common solution lead to the software development of a specific data collection solution for each new equipment type.

The Bee Framework has been designed with a modular architecture in order to provide easier and general methods capable of integrating equipments and their related systems. By using such a framework, it will not be necessary to develop a specific data collection solution for each new equipment type's integration, since the core functionalities will already be available and ready for use. It will just be necessary to configure the framework according the equipment being integrated and adapt it to face the specific equipment requirements.

Beyond the aspects related to the framework architecture and specification described in this document, this project has also considered the development of a proof of concept. The main goal of the proof of concept is to show the resulting advantages of using the framework in the equipment integration process. The equipment type considered had already a specific solution to collect the data generated and is integrated in the Qimonda assembly line using a different approach. So, this proof of concept intends to establish a comparison regarding both the effort and time needed to integrate the same equipment type using the framework approach instead of the previous one.

Resumo

O projecto descrito neste relatório e designado por Bee Framework é, essencialmente, uma *framework* de recolha de dados usada para a integração de equipamentos, que pretende auxiliar e melhorar o processo de integração. Mesmo tratando-se de equipamentos de tipos diferentes, existe um grande conjunto de funcionalidades em comum no que à recolha de dados diz respeito, podendo assim proceder-se à reutilização das funcionalidades comuns efectuando apenas ligeiras ou até mesmo nenhumas alterações. Algumas destas funcionalidades são, por exemplo, o acesso a bases de dados, monitorização de directórios, comunicação e recolha de dados.

O principal objectivo do projecto descrito é a elaboração de uma especificação detalhada para uma *framework* de recolha de dados gerados por equipamentos utilizados em linhas de produção. Adicionalmente, os requisitos nucleares da *framework* devem também ser implementados, devendo ainda realizar-se a integração de um equipamento como prova de conceito. Pretende-se assim um controlo e monitorização destes equipamentos de modo a recolher os dados por estes produzidos e armazená-los em suportes que permitam posterior análise e interpretação dos mesmos, possibilitando uma contínua evolução e optimização dos processos.

No entanto, devido à grande diversidade de equipamentos existentes numa linha de produção e devido ao facto de alguns destes equipamentos não seguirem as normas existentes utilizadas na indústria de semicondutores no que à recolha e controlo dos dados diz respeito, torna-se necessário desenvolver soluções específicas de recolha de dados para cada novo equipamento que se pretende integrar, de modo a proceder à recolha dos dados que são gerados.

A Bee Framework possui uma arquitectura modular, de modo a fornecer mecanismos fáceis e gerais capazes de integrar equipamentos e os sistemas com eles relacionados. Utilizando uma *framework* deste género, não será necessário desenvolver uma solução específica para cada nova integração de um equipamento, uma vez que as funcionalidades nucleares estarão já disponíveis e prontas a utilizar. A *framework* deverá apenas ser configurada de acordo com os requisitos específicos do equipamento a integrar.

Para além dos aspectos relacionados com a arquitectura e especificação da *framework* descritos neste relatório, este projecto contou ainda com o desenvolvimento de uma prova de conceito. O seu principal objectivo é demonstrar as vantagens decorrentes da utilização da *framework* na integração de um equipamento, isto é, proceder à recolha dos dados produzidos pelo equipamento quando este se encontra em funcionamento numa linha de produção da Qimonda. Poderá assim ser efectuada uma comparação relativa à quantidade de esforço e tempo requerido, já que o equipamento considerado se encontrava já previamente integrado com outra abordagem.

Acknowledgments

I would like to thank Qimonda Portugal for giving me the required conditions I needed to complete my project. I would like to thank all my work colleagues, especially my project supervisor Nuno Soares, but also Rui Alves and all other colleagues belonging to the Equipment Control team for being so supportive and helping me better understand my project domain.

A special thank too for my project supervisor at Faculdade de Engenharia da Universidade do Porto, professor Teresa Galvão, for giving me all the encouragement and support I needed, not only in this project but also during the course of my studies, always with honest interest and friendship.

I also would like to thank all the teachers that in some way contributed not only to my education but also throughout my personal life.

Additionally, I extend my thanks to the English professor Sónia Nogueira for the help she provided me in the reviewing of this document.

Finally, my biggest gratitude to my family, especially my parents and sister, and to all my friends for all the support they give me, both in the good and bad times, always helping me facing my problems and giving me the encouragement and boldness I needed to keep going and achieve my goals. My sincere and special word of thanks to all of you.

Daniel Aguilar

To my family and friends

Contents

List of Figures

List of Tables

Glossary

AOI	Automatic Optical Inspection
API	Application Programming Interface
Application block	Software component designed to be as agnostic as possible to the application architecture, so that it can be easily reused by different software applications.
Assembly line	Manufacturing process in which parts are added to a product in a sequential manner using optimized logistic and operations plans in order to create a finished product faster.
CFGmgr	Configuration Manager
CLR	Common Language Runtime — Virtual machine component of Microsoft's .NET initiative.
CPU	Central Processing Unit
DAB	Data Access Block — Application block provided by Microsoft Enterprise Library related to database access architecture.
Daemon	Computer program that runs as a background process.
Data collection	Process of preparing, collecting and saving the data generated by some type of source.
Data mining	Process of sorting through large amounts of data and picking out relevant information.
Data warehouse	Electronic repository of an organization's stored data
DDL	Data Definition Language — Computer language for defining data structures.
Deadlock	Situation that occurs when two or more competing actions are waiting for the other to finish, and thus neither ever does.
Design pattern	General reusable solution to a commonly occurring problem in software design.

DLL	Dynamic Link Library
DMS	Decision Making System — Computer-based information system including knowledge-based systems that support decision-making activities. Also known as Decision Support System (DSS).
ECMA	European Computer Manufacturers Association — International and private (membership-based) standards organization for information and communication systems.
EDA	Equipment Data Acquisition — Collection of SEMI standards for the semiconductor industry to improve and facilitate communication between data collection software applications and factory equipments.
EDC	Engineering Data Collection — Engineering processes and tools used in data collection.
Event-based programming	Programming paradigm in which the flow of the program is determined by sensor outputs, user actions, or messages from other programs or threads.
Folder monitoring	Process of observing the contents of a folder and detect occurred changes in its files or folders.
Framework	Reusable design of a software system described by a set of abstract classes and by the way instances of these classes collaborate, allowing both the reutilization of code and design architecture, which considerably reduces the development effort needed.
GNU	Computer operating system composed entirely of free software.
GoF	Gang of Four — The group of authors formed by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
GUI	Graphical User Interface

HTML	HyperText Markup Language — Markup language that provides a means to describe the structure of text-based information in a document.
IDE	Integrated Development Environment — Software application that provides comprehensive facilities to computer programmers for software development.
Integration tests	Tests in which individual software modules are combined and tested as a group.
Interface A	Alternative name used to EDA (Equipment Data Acquisition).
Inversion of Control	Abstract principle describing an aspect of some software architecture designs in which the flow of a system is inverted in comparison to the traditional architecture of software libraries.
Locking (database)	Mechanism used to prevent data from being corrupted or invalidated when multiple users need to access a database concurrently.
Lot (semiconductor)	Set of semiconductor modules acting as a sole unit.
LotEquipParamsHistSrv	Lot Equipment Parameters History Server
Manufacturing equipment	Semiconductor manufacturing equipment consists of manufacturing equipment used in a clean room for the fabrication of semiconductor chips, test equipment used in the manufacturing and research and development environment and to test semiconductor manufacturing equipment, and fixtures in place to support a semiconductor fabrication facility.
Message center	Global access point of a software application used to monitor flow of messages inside the application and that guarantees the attending and dispatching of received messages.
Message handling	Comprises the concepts of sending, receiving and correctly routing messages in a software application.

MSMQ	Microsoft Message Queueing — Messaging protocol that allows applications running on disparate servers to communicate in a failsafe manner.
OCL	Object Constraint Language — Declarative language for describing rules that apply to UML models.
OEDV	Online Equipment Data Visualization — Tools used to online visualization of the data generated by equipments.
OMG	Object Management Group — Consortium focused on modeling and model-based standards.
OOP	Object-oriented programming — Programming paradigm based on the use and interaction of different software units known as "objects".
ORM	Object-Relational Mapping — Programming technique for converting data between incompatible type systems in relational databases and object-oriented programming languages.
PCB	Printed Circuit Board — Board used to mechanically support and electrically connect electronic components using conductive pathways etched from copper sheets laminated onto a non-conductive substrate.
PL/SQL	Procedural Language / Structured Query Language — Procedural extension to the SQL database language.
Raw data	Term used to refer unprocessed data (also known as primary data).
RDBMS	Relational Database Management System
Refactoring	Rewriting of some pieces of software code to improve its performance or to increase its understanding, without changing its initial meaning and behavior
Reflection (programming)	Reflection is the mechanism of discovering class information solely at runtime.

Rolling (log files) Rolling is a combination of rotation and translation operations used when adding a new log entry. Oldest entries are translated (or deleted if necessary) in favor of new entries, keeping a log file always updated with the recent entries.

RS-232 Recommended Standard 232 — Standard for serial binary data signals commonly used in computer serial ports.

SAM Statistical Appearance Model — Technique that detaches users from the parameter adjustment of complex algorithms and that makes a more systematic use of training images during teaching steps.

SECS/GEM SEMI Equipment Communications Standard / Generic Equipment Module — Standard interface used in semiconductor industry for equipment communications.

SEMI Semiconductor Equipment and Materials International — International organization which main focus is the promotion of the semiconductor industry and associate manufacturers of equipment and materials used in the fabrication of semiconductor devices.

Semiconductor industry Collection of business firms engaged in the design and fabrication of semiconductor devices.

SGML Standard Generalized Markup Language — ISO Standard metalanguage used to defined markup languages for documents.

SMT Surface Mounting Technology — Method for constructing electronic circuits in which the components are mounted directly onto the surface of printed circuit boards.

SMTP Simple Mail Transfer Protocol — Standard for email transmissions across the Internet.

SQL Structured Query Language — Database computer language designed for the retrieval and management of data in RDBMS.

SVN Subversion — Version control system used to maintain current and historical versions of files such as source code, web pages, and documentation.

TCP/IP Transmission Control Protocol / Internet Protocol — Set of communications protocols that implement the protocol stack on which the Internet and most commercial networks run. Also known as Internet Protocol Suite.

TDD Test Driven Development — Software development technique consisting of short iterations where new test cases covering the desired improvement or new functionality are written first, then the production code necessary to pass the tests is implemented, and finally the software is refactored to accommodate changes.

Test automation Test automation is the use of software tools to control the execution of tests automatically.

TNS Transparent Network Substrate — Transparent layer that enables a heterogeneous network consisting of different protocols to function as a homogeneous network.

UML Unified Modeling Language — Object modeling and specification language used in software engineering.

Unit test Automated test that validates if individual units of source code are working properly.

W3C World Wide Web Consortium — Main international standards organization for the World Wide Web.

Wrapper (software) Refers to a type of packaging that hides implementation code from end users and just provides the required interfaces that allow the execution of a *wrapped* functionality.

XML Extensible Markup Language — General-purpose specification for creating custom markup languages.

YODA	Your Own Data Adapter — Middleware cross-platform application that allows different applications to exchange messages in the same network.

Chapter 1

Introduction

This first chapter introduces the relevant themes needed for a full understanding of both the current document and the project context. The chapter includes five different sections to describe the project summary, the project main motivations and expected goals, the methodology used and finally a last section to describe the report structure.

1.1 Project Introduction

The project Manufacturing Equipment Data Collection Framework - Bee Framework - is essentially a framework specification for collecting the data generated by a wide variety of equipments in an assembly line.

Its subtitle, Bee Framework, is a pure analogy with what happens in the animal world of bee. Bees focus either on gathering nectar or on gathering pollen depending on demand from a wide variety of sources. Bees also play an important role in pollinating flowering plants, and are the major pollinator in ecosystems that contain flowering plants. It is estimated that one third of the human food supply depends on insect pollination, most of which is accomplished by bees [1]. Moreover, pollen and nectar can then be used to produce a large number of products, such as honey, candles and beeswax, among others, and its usage comprises areas like medicine, food, cosmetics, beebread, pollination and even pollution monitoring [2, 3].

Similarly, the Bee Framework project is intended to provide the necessary tools required to perform data collection from a wide source of manufacturing equipments. The collected data needs then to be prepared and finally fed into the data analysis solutions that support and help improving the manufacturing process. In summary,

Figure 1.1: Bee Framework logo

data may be collected from many sources and may as well have many target destinations, just like the pollen and nectar collected by bees that can be used in many areas.

Most of the generated data is available through equipment local databases or file systems. This means that this data needs to be collected by querying a database or by reading, parsing and interpreting files, respectively. Since there is a large number of different equipment types, integrating equipments and collecting the data they generate becomes an unpleasant and difficult task. The Bee Framework must then provide, most of all, the common methods to collect the generated data and the crucial services generally used while integrating equipment, such as database access, folder monitoring, communication, or logging, among others.

The impact of such a framework would be considerable in the equipment integration process, mostly because common requirements are general enough to be easily used, decreasing the amount of effort and time needed to integrate equipments in terms of collecting the data they generate.

1.2 Project Motivation

With the continuous development and improvement of existing and new data analysis and data mining software solutions, there is an increasing need to collect data. However, these software tools only provide valid and reliable results if large amounts of data are considered, otherwise the correctness of achieved results can be questionable and controversial. Regarding this need, it becomes obvious that data collection plays a critical role in the early phase of the process.

Nowadays, a growing variety of problem domains require the usage of this kind of work and decision tools in order to follow-up a continuous process optimization and control. In order to meet these demands from data analysis solutions, effective and powerful data collection tools are also needed.

A data collection framework enclosing common functionalities generally used to collect data from a wide scope of sources and save the collected data into numerous possible targets can then be particularly useful and critical. A framework meeting these characteristics could then be used in a large set of problem domains with a small amount of effort and in a very short time.

Moreover, since the problem domain considered is data collection in manufacturing equipments, there are still some several interface constraints and limitations in following international standards in the way equipments generate data and how these data can be collected. With such restrictions in the current scenario, it generally implies the development of new specific data collection solutions, without promoting reusability and decreasing developers' productivity by keeping them doing similar approaches.

1.3 Project Goals

The main goal of this project is to come up with a detailed system specification of a framework to be used for data collection in manufacturing equipments in an assembly line. Additionally, the core requirements of the framework should also be implemented and an equipment integration should also be done as a proof of concept. The proposed solution must take in consideration some of the existing data collection solutions and a wide range of possible equipments, so that it can achieve a high-level of abstraction. This required level of abstraction will then make it possible to collect data from different equipment types without much integration effort.

The data collection framework must consider the existence of common data collection functionalities and different approaches for integrating different equipment types. These common functionalities are considered the central part of the framework. They provide the necessary tools to promote code and components reusability while integrating equipments and must be general enough so that no changes need to be done when their usage is required.

Finally, a prototype should be built as a proof of concept of the proposed solution. This prototype should consider the core functionalities of data collection for manufacturing equipments. Since the data collection problem is mostly related to the equipment integration topic, the prototype should consider the integration of an equipment type used in the Qimonda[1] assembly line in terms of collecting and saving the data it generates.

[1] http://www.qimonda.com

1.4 Approach Methodology and Constraints

This section will firstly describe the approach methodology used during the project. Afterwards, some constraints that affected the normal progress of the project or that imposed some restrictions in terms of project decisions are also referred.

The collection and analysis of equipment data collection requirements has been the first task. Usual data collection flows and main requirements related to data collection have been analyzed and some previous equipment integration solutions regarding data collection have been studied, so that the common functionalities could be identified.

At the same time, a research on the state of the art technologies that could possibly be used in the prototype development has also been started, so that the decisions about the choice of appropriate technologies and applications could be made.

After collecting the major requirements needed for a data collection framework and after studying both the previous work done about this theme and the technologies used, the design of the solution started almost in parallel with the development of the framework core functionalities.

When the development of these core functionalities has been successfully completed, the prototype related to the effective data collection by integrating an equipment and collect the data it generates has then started.

Throughout the prototype development, whenever it was possible a Test Driven Development (TDD) methodology has been followed and the SVN (Subversion) version control system has also been adopted. In addition, some notes about architecture design and a high-level design document about the data collection regarding the AOI — Automatic Optical Inspection — equipments have been written. This equipment type and its data collection process will be detailed in Chapter 5.

The constraints that affected the normal progress of the project are mostly related to technological characteristics and decisions about the applications and technologies to be used in the project. For example, both C# programming language and UML modeling can be considered as being project normative.

These constraints are essentially correlated to software applications that could possibly be used and the lack of the required licenses for using them. This situation has lead to the usage of other software alternatives or, in most cases, to the usage of older releases of the desired software. These technological constraints will be detailed in the Technology Review section (section 3.1) of the State of the Art chapter (Chapter 3).

1.5 Report Structure

This report has been written and structured in order to help readers understanding the document and the project by performing a top-down analysis. This writing approach starts by giving readers a high-level overview of the project contents so that they can immediately understand the global concepts of the project and what it is intended to do. After this high-level overview, the document will then gradually disclose all the details concerning the problem analysis and design, as well as the current implementation of the proposed proof of concept solution.

According to this approach, this initial chapter elucidates the readers about the project overview context, its main goals and motivation and the used approach methodology that lead to the project conclusion.

Chapter 2 will present a more detailed analysis of the data collection problem. It also will refer the main motivations that lead to the origin of the problem and the expected results that may have a considerable impact improving the data collection. The global problem that concerns data collection will then be divided into smaller problems, detailing each one and presenting a review of the previous approaches.

Chapter 3 will focus on the State of the Art. The first section of this chapter will present a technology review, focusing in the technologies and applications considered. An individual description of each one of these technologies and applications will also be done. In addition, a comparative analysis relating these technologies and the main reasons that lead to their adoption or abandon will also be provided. The following section of this chapter will concern the previous work done regarding data collection in manufacturing equipments used in the assembly line.

Chapter 4 will refer to the proposed Framework Architecture and Specification. Initially, this chapter will focus on the main requirements identified for the presented solution, especially regarding the data collection problem related to manufacturing equipments. The following section of this chapter will then detail the proposed architecture for fulfilling the requirements identified and explain the usage of design patterns to solve some of those requirements and needs.

Chapter 5 will describe the project development and some of the technical decisions taken at implementation level. This chapter will mainly focus in the related characteristics of each requirement previously specified in the fourth chapter. Moreover, the chapter will explain the overall framework development, the required configuration settings needed and finally how the application works. Since the project is about data collection, specifically data collection from manufacturing equipments, the development of the proof of concept regarding the equipment integration and the consequent data collection for one type of equipment as an example will also be considered in this chapter.

In Chapter 6, the main findings will be discussed. Additionally, a comparison between the proposed solution and the previous approaches will also be done. This comparison will mostly comprise the integration of the same type of equipment using both approaches in terms of effort and time needed as well as performance evaluation.

Finally, the main conclusions achieved after the project conclusion will be related in Chapter 7. Furthermore, the conclusions will also include some final recommendations and perspectives of future work to improve and expand the proposed solution.

Chapter 2

Data Collection Problem Analysis

This chapter presents a detailed description regarding the data collection problem, relating the problem with manufacturing equipments. The needs that lead to the origin of the problem and the expected results that may have a considerable impact in tasks related to data collection will also be considered. The chapter starts by giving an overview about the data collection problem in general, then refers to the data collection problem considering the semiconductor industry and finally details the problem taking in consideration the specific case data collection at Qimonda.

2.1 Data Collection Overview

People live in an information society in which the creation, distribution, diffusion, use, and manipulation of information is significant for almost every activity. More now than ever before, governments, industry and society need reliable information to make better decisions in tackling these problems [4]. Consequently, the importance of data collection mechanisms has been and still is growing considerably, so that the collected data can be used to provide the adequate information required by society.

A data collection system collects data from the outside world and which main goal is to feed other systems with this data, such as decision making systems (DMS), usually for the purpose of controlling a system [5]. The concept beyond data collection means collecting data from a wide range of possible data sources and turn it into useful information that can be further used. This information can then be critical not only to control a process or a system, but also to provide a better understanding about the domain considered in the data collection process [6, 7].

2.2 Data Collection in Semiconductor Industry

The existing competitiveness in the semiconductor industry and the demands for high quality and very reliable memory products lead companies in this sector to adopt manufacturing processes based on a set of international standards, rules and conventions in order to guarantee final products that meet customer quality demands. The adoption of common standards is a sign of a mature industry. Worldwide semiconductor companies, through fierce competitors in the marketplace, have shown remarkable willingness to cooperate in creating and adopting common standards for factory automation [8].

Manufacturing processes and equipments used in the semiconductor industry have been continually improved. Along with the technical advances of the semiconductor manufacturing processes themselves, factory productivity and efficient manufacturing control are key to a fab's success. In fact, that success increasingly relies on the collection and analysis of growing amounts of detailed process, measurement and operational data from the equipment to improve yield, efficiency, productivity and more. As processes become more complex, it becomes more important to use the data to reduce process variation, minimize the impact of excursions, and improve overall equipment effectiveness [9].

During all the phases of the manufacturing processes, products are exhaustively tested to ensure a high quality level. The equipments used either when the memory products are being mounted either when they are being tested generate large amounts of data. This data is related to measurements, physical or electrical failures detected, temperatures or humidity values, tensions and voltages, or optical inspections, for example.

The data generated by these different equipment types is extremely important and plays a critical role in the improvement of manufacturing processes. Collection of real data is a vital step in managing a modern manufacturing organization. This data can be very useful since it provides Process engineers the required values that can help them better understand all the manufacturing process stages and help detect which steps can be improved and how these improvements should be achieved. Furthermore, information is available immediately so problems can be identified and corrected when a problem exists, not when it is noticed after a full day of incorrect production [10].

In order to face the issues related to the semiconductor industry, an organization have been founded during the decade of 1970. This organization is the Semiconductor Equipment and Materials International — shortly SEMI — and its initial main focus was to divulge and promote the semiconductor industry and associate manufacturers of equipment and materials used in the fabrication of semiconductor

devices such as integrated circuits, transistors, diodes, and thyristors [11]. Among other activities, SEMI acts as a "clearinghouse" for the generation of standards specific to the industry and the generation of long-range plans for the industry.

The most well known standard developed by this organization is the SECS/GEM (SEMI Equipment Communications Standard / Generic Equipment Model). The SECS/GEM interface is the semiconductor industry's standard for equipment-to-host communications. In an automated fab, the interface can start and stop equipment processing, collect measurement data, change variables and select recipes for products. The SECS/GEM standards do all this in a defined way, defining a common set of equipment behavior and capabilities [12].

With the purpose of always trying to improve performance and productivity for semiconductor fabs and equipment used, a new standard named Equipment Data Acquisition (EDA) Interface, also known as Interface A, is available and ready to be deployed in manufacturing organizations. Whereas the SECS/GEM standards were created to improve tool control and to facilitate and support high levels of factory automation, the EDA standards focus on improving process monitoring and control, given the advancing technology and increasing complexity of semiconductor manufacturing processes [13].

Although Interface A offers improved data ports over SECS/GEM, it does not replace the SECS/GEM standards, which pertain to equipment control and configuration. It is also distinct from Interface B, which facilitates data sharing between applications, and Interface C, which provides remote access to equipment data. Industry adoption of Interface A has been gaining momentum, but more needs to be done to fully implement the standard across the industry [14].

2.3 Data Collection at Qimonda

However, even considering the adoption of international standards by the industry, some of the equipments used in Qimonda assembly lines are generically used not only by the semiconductor industry. Consequently, some of these equipments do not follow the SECS/GEM standards defined by the semiconductor industry, making equipment integration tasks harder to perform and also introduces many difficulties in the data collection process.

The integration of such equipments lead to the development of specific data collection solutions for each equipment type considered. Since all these equipments follow different conventions and rules not only in the way they are used but also in how they generate data and how this data can be collected, it becomes very hard to promote consistency.

This situation led Qimonda equipment integration team to the development of multiple and different approaches related to data collection from these manufacturing equipments. These approaches are designed and planned almost exclusively taking in consideration a specific type of equipment, which causes a single and unique architecture design. Consequently, the amount of time and effort required from equipment integration team members increases, not only because they have to think and implement a new architecture solution each time a new equipment needs to be integrated but also because they have to document them. Obviously, the productivity of the team members is then negatively affected.

The existing lack of consistency related to different integration architectures approaches not only led developers to focus on specific equipments instead of a global architecture but also led them to adapt some software components used in previous integrations approaches achieved. The reutilization can be considered a positive aspect but also has some negative consequences and turn downs. Since integration solutions are developed focusing a single specific equipment, similar components can not usually be directly reused without having to change and adapt it to the new equipment requirements.

Consequently, these components have to be continuously adapted and developers recurrently face the same problems. Additionally, the reutilization of such components also introduces some constraints related to technological issues. Old versions of these software components are commonly used and this fact limits some choices related to the technologies and implementation approaches used due to compatibility requirements. This way, even when considering the development of new integration solutions, these solutions are limited since their beginning by out of date technologies, which can considerably affect both the performance of solutions and the maintenance effort needed.

A project to evaluate Qimonda EDA has looked at the problem of factory-wide deployment of Interface A from a number of perspectives and has tried to incorporate the goals of the EDC — Engineering Data Collection — refactoring into a comprehensive vision [15]. It is difficult to conclude with certainty how rapid and pervasive the adoption of the nascent EDA standards will be.

Moreover, even considering this new standard, the problem related to data collection is still not solved because there are still some equipments not following the standards of the semiconductor industry and for which a data collection approach is needed.

Chapter 3

State of the Art

This chapter introduces both the technologies used in the project and the previous work done to help solving the data collection problem described in the second chapter. Both introductions will be focused on the semiconductor industry, since some of the technologies used to create the framework are related to this sector.

Additionally, some alternative technologies that could possibly be used to act upon data collection will also be considered in this chapter. However, once again, some decisions about different technological choices have been made considering the semiconductor industry, so taking a decision about replacing a technology by other one should always take this factor in consideration.

3.1 Technology Review

This section presents the main applications and technologies studied during the initial phase of the project and later used in the project development. Additionally, some alternative technologies and applications that were considered but not used in the project will also be described. For these technologies, the main focus are the pros and cons of using them in the data collection problem related to manufacturing equipments and which were the main reasons that lead to their abandon.

3.1.1 Programming Languages and Tools

3.1.1.1 C# — C Sharp

C Sharp is a successfully adopted object-oriented programming language developed by Microsoft as part of the .NET initiative and later approved as a standard by ECMA [16]. C# 3.0 is the current version of the language and was released on 19 November 2007 as part of .NET Framework 3.5, but due to some technical and

licensing aspects, 2.0 version of the language and the 2.0 .NET Framework have been used instead in this project. This programming language has a procedural, object-oriented syntax initially based in the C family of languages but also including very strong influences from several other aspects of programming languages (C++, Python and most notably Java) with a particular emphasis on code simplification.

C# is intended to be a simple, modern, type-safe, general-purpose programming language which allows the development of robust and durable applications. The language includes strong type checking, array bounds checking, detection of attempts to use uninitialized variables, source code portability, exception handling and automatic garbage collection. This way, not only software robustness and durability are considered by the language, but it also helps programmers using it, increasing their productivity.

C# is an object-oriented language, but it further includes support for component-oriented programming. Currently, software design relies more and more on software components [17]. Key to such components is that they present a programming model with properties, methods, and events; they also have attributes that provide declarative information about the component and incorporate their own documentation. One of the biggest advantages about using C# is that this language directly supports all these concepts, making it a very natural language to create and use software components.

C# can be considered a very high-level programming language when considering other languages such as C or Assembly. Although C# applications are intended to be economical concerning memory and processing power requirements, the language cannot compete on performance with those low-level languages. C# applications, like all programs written for the .NET tend to require more system resources than functionally similar applications that access machine resources more directly.

Microsoft Visual Studio[1] has been the development environment chosen to develop the framework and the proof of concept. However, two different versions of this product from Microsoft have been considered: Microsoft Visual Studio 2005 and Microsoft Visual Studio 2008.

Figure 3.1: Visual Studio logo

[1] http://www.microsoft.com

Microsoft Visual Studio 2008

Visual Studio 2008 is the most recent version of this IDE — Integrated Development Environment — from Microsoft and has been recently released in November 2007. This version has been the one initially chosen and considered to develop the C# components of the framework. However due to licensing difficulties, this choice has been abandoned, which also has determined the choice of the .NET Framework and C# language versions [18].

Microsof Visual Studio 2005

Visual Studio 2005 is the ancestor version of the Microsoft IDE referred in the previous section and it was the one adopted as a development environment [19]. This product supports the features introduced by the .NET Framework 2.0 version.

Resharper

Furthermore, a trial version of Resharper 2.0 has been used. Resharper is a product from JetBrains[2] and is a refactoring add-in for Visual Studio which helps programmers increase their productivity while developing. Resharper allows code completion, easy refactoring, code analysis and assistance, code formatting, code generation and templates, and also easier code navigation [20].

This work tool has been proven very useful, allowing an easier and quicker development of the framework components.

Figure 3.2: Resharper logo

NUnit

NUnit[3] is an open source unit-testing framework for all .Net languages. Initially ported from JUnit (used with the same purpose for Java), it is written entirely in C# and has been completely redesigned to take advantage of many .NET language features, for example custom attributes and other reflection related capabilities [21].

This testing framework discovers test methods using code reflection and provides test automation to control the execution of unit tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions [22].

[2] http://www.jetbrains.com
[3] http://www.nunit.org

Since a Test Driven Development [23] agile methodology has been adopted, this unit-testing framework has been not only very useful but also fundamental in the development process.

Figure 3.3: NUnit logo

3.1.2 Modeling Languages and Tools

3.1.2.1 UML — Unified Modeling Language

UML[4] is a standardized visual specification language for object modeling used in the software engineering field. UML is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system, referred to as a UML model. UML is the OMG's — Object Management Group — most-used specification, and the way the world models not only application structure, behavior, and architecture, but also business process and data structure [24].

Figure 3.4: UML logo

Modeling is the designing of software applications before coding. Achieved models are very helpful since they let us work at a higher level of abstraction, helping specification, visualization and documentation modeling tasks of software systems, including their structure and design [25].

Its origin dates from 1994, when the large abundance of modeling languages was slowing down the adoption of object technology. A unified method was needed and a consortium with several organizations, named UML Partners, has been established in 1996 with the purpose of coming up with a specification of a unified modeling language. As a result of this collaboration, a strong UML 1.0 definition has been

[4] http://www.uml.org

achieved. This modeling language was already well defined, expressive, powerful, and generally applicable. It was submitted to the OMG in January 1997 as an initial Request for Purposal response [26].

UML has matured significantly since its early versions. Several minor revisions fixed shortcomings and bugs and the UML 2.0 major revision was adopted by the OMG in 2003. There are actually four parts to the UML 2.x specification: the Superstructure (defines the notation and semantics for diagrams and their model elements), the Infrastructure (defines the core metamodel on which the Superstructure is based), the OCL — Object Constraint Language — (defines rules for model elements) and finally the UML Diagram Interchange (defines how UML 2 diagram layouts are exchanged). The current versions of these standards follow: UML Superstructure version 2.1.2, UML Infrastructure version 2.1.2, OCL version 2.0, and UML Diagram Interchange version 1.0.

Visio

Visio[5] is a diagramming software originally developed by Vision Corporation, a company that has been bought by Microsoft in 2000. It uses vector graphics to create diagrams and follows the recent standards of UML modeling language.

Visio provides a wide range of templates - business process flowcharts, network diagrams, workflow diagrams, database models, and software diagrams - that can be used to visualize and streamline business processes, track projects and resources, chart organizations, map networks, diagram building sites, and optimize systems [27].

Visio 2007 is the most recent version of this software. The current version of Visio has been used to model all UML diagrams referring to framework specification and architecture.

3.1.3 RDBMS and Database Tools

3.1.3.1 Oracle

Oracle Database[6] is a relational database management system (RDBMS) commonly referred as simply Oracle which has become a major presence in database computing. Oracle Corporation, which is the company that produces and markets this database software, has been founded in late 70's (1977) and since then many widespread computing platforms have come to use the Oracle database extensively, making the company to actually be the market leader [28].

The last version of Oracle Database has been recently released and this software is currently in the 11g version. Once again, due to licensing and technical matters,

[5] http://office.microsoft.com
[6] http://www.oracle.com

Figure 3.5: Oracle Corporation logo

the version considered while developing this proof of concept is the 9i. Moreover, data collection generally involves large volumes of collected data, increasing the risks of performing migrations of current database migrations. However, database operations described in this report and used in the proof of concept should behave the same expected way in recent releases of this software, even considering the grid-computing technology that came with 10g or later versions.

Unlike the C# programming language referred before, a powerful development database environment was not needed. Some database applications, such as SQL Navigator or PL/SQL Developer have been considered but SQL Developer, which already comes up with the Oracle Database Client, meet the necessary requirements.

SQL Developer
Oracle SQL Developer[7] is a free graphical tool for database development. SQL Developer can be used to browse, create and modify database objects, run SQL statements and SQL scripts, and edit, run and debug PL/SQL statements. It can also run any number of provided reports, as well as create and save new types of reports. Additionally, SQL Developer allows to export / import data and DDL and supports version control. SQL Developer is a tool that enhances productivity and simplifies database development tasks [29].

SQL*Loader
SQL*Loader[8] is a bulk loader utility used for moving data from external files into the Oracle database. It comes with some configurable loading options and supports various load formats, selective loading, and multi-table loads [30]. Its usage is particularly recommended to load large volumes of data into Oracle Database because it consumes less resources, specially those related to time, memory and CPU processing.

3.1.3.2 SQL Server, Access and PostgreSQL

Microsoft SQL Server[9], Microsoft Access[10] and PostgreSQL[11] have been considered as possible alternative database management systems. However, since data collec-

[7] http://www.oracle.com/technology/products/database/sql_developer/
[8] http://www.orafaq.com/wiki/SQL*Loader_FAQ
[9] http://www.microsoft.com/SQL
[10] http://office.microsoft.com/access
[11] http://www.postgresql.org

tion can be very resource consuming, usually involving large volumes of data and strongly related to mechanisms of data warehousing and data mining, Oracle has been the natural database choice. The main reason for this choice is the well known robustness and efficiency of Oracle databases in such situations. Nevertheless, the data collection framework presented not only considers Oracle but also these alternative databases. This topic will be detailed and explained in Chapter 5.

3.1.4 Data Persistence

3.1.4.1 Enterprise Library

Microsoft Enterprise Library[12] or simply Enterprise Library is a collection of reusable software components for the Microsoft .NET Framework. These components are application blocks designed to assist software developers with common enterprise development challenges and problems that commonly are faced from one project to the next ones [31].

Application blocks are designed to encapsulate the Microsoft recommended best practices for .NET applications. In addition, they can be added to .NET applications quickly and easily [32]. Application blocks are a type of guidance, provided not only as source code but also as documentation that can be directly used, extended, or modified by developers to use on complex, enterprise-level line-of-business development projects.

This guidance is based on real-world experience and goes far beyond typical white-papers and sample applications. They provide proven architectures, production quality code, and recommended engineering best practices. The technical guidance is created, reviewed, and approved by a wide diversity of experienced people including Microsoft architects, partners and customers, engineering teams, consultants, and product support engineers. The result is a thoroughly engineered and tested set of recommendations that can be followed with confidence when building applications based on this guidance [33].

Enterprise Library and its applications blocks provide an API to facilitate best practices in core areas of programming such as logging, validation, data access, exception handling, and many others. However, application blocks are designed to be as "agnostic" as possible to the application architecture, so that they can be easily reused in different contexts.

Figure 3.6 shows the applications blocks available in the Enterprise Library 3.1 release and illustrates their interdependencies. [32] Both Data Access and Logging Application Blocks have been used in the Bee Framework. Their usage will be

[12] http://msdn.microsoft.com/entlib

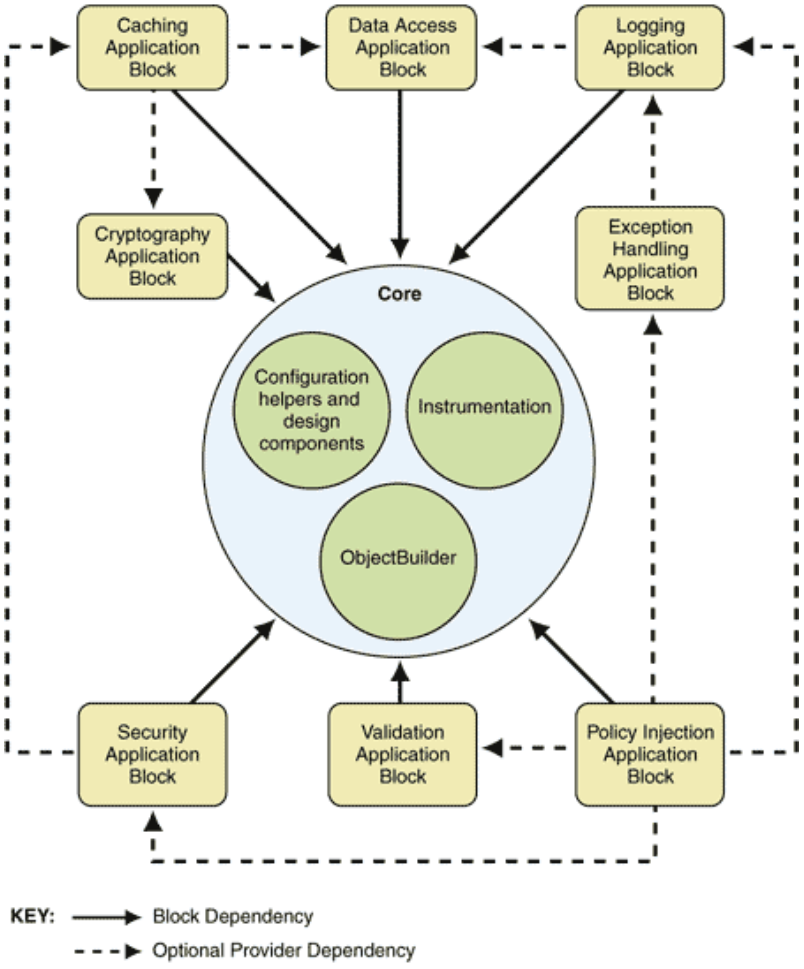


Figure 3.6: Interdependencies of the Enterprise Library application blocks

explained later in the Framework Specification and Architecture chapter (Chapter 4) because they have significant impact in the proposed architecture.

Amongst the main benefits of using Enterprise Library we can identify the productivity and testability enhancements: each of the application blocks provides several interfaces meant to satisfy common concerns and a level of isolation that allows individual testing of each block. Additionally, extensibility (developers can customize the application blocks and extend their functionality to suit own needs), consistency (design patterns are applied in similar fashion in all the blocks), ease of use and integration (application blocks can be used as pluggable components) are also strong advantages of using Enterprise Library [34].

Enterprise Library 4.0 is the most recent version and has just been released recently in May 2008. This last release includes some bug corrections, new applications blocks, some performance improvements, and already supports Microsoft

Visual Studio 2008 and the .NET Framework 3.5 [35]. However, since this last version has been released after the beginning of the project, the previous release of Enterprise Library, version 3.1 - May 2007, has been used instead. Moreover, as described before in the C# Technology Review section, Visual Studio 2005 and .NET Framework 2.0 have been used, so using the last release of Enterprise Library would be impossible due to software requirements.

3.1.4.2 NHibernate

NHibernate[13] is a port of the famous Hibernate Core for Java to the .NET Framework [36]. It is an Object-Relational Mapping (ORM) solution that provides an easy to use framework for mapping an object-oriented domain module to a traditional relational database, handling persisting plain .NET objects to and from the underlying database.

With this support to transparent persistence, object classes do not have to follow a restrictive programming module. These persistent classes do not need to implement any interface or inherit from a special base class. Just by giving a XML description of the entities and relationships, NHibernate automatically generates the necessary SQL for loading and storing the objects. This characteristic makes it possible to design the business logic using plain .NET (CLR — Common Language Runtime) objects and object-oriented idiom. This object-oriented approach relieves the developer from a significant amount of relational data persistence-related programming tasks.

NHibernate is free as open source software that is distributed under the GNU Lesser General Public License and its most recent version number is 1.2.1. NHibernate 2.0 is currently under development [37].

However, despite these apparent advantages and interesting features, NHibernate has been abandoned in favor of Microsoft Enterprise Library, described in the previous section. The reason for this choice is inspired by expressions as “no solution is final solution” and “all have pros and cons” because it depends on the the data access layer architecture and how this layer is implemented. Each case must be individually considered in order to determine whether a technology / application is better or worst than other one.

The main reason for this decision is the type of application considered: a data collection framework. When referring to a data collection framework, it is expected that all database accesses can be abstract, not depending on which type of database is used, which tables exists, which columns each table contains, and so on. Such a

[13] http://www.hibernate.org

framework is expected to have a high level of abstraction, providing a wrapper that can execute any kind of query, stored procedure, or transaction different databases. Of course, each query, stored procedure or transaction must be defined when using the framework services but the way of using database services remain exactly the same whatever the database is.

By choosing NHibernate it would not be possible to attend the required universal way because it would be necessary to generate the mapping XML files, then generate the SQL used for database access and finally generate all object-oriented class files. Even considering the usage of automatic code generation tools for NHibernate, such as MyGeneration[14] or CodeSmith[15], some adjustments in XML files and object classes must usually be done specially when dealing with complex entity relationships. This situation increases the complexity and difficulty of maintainability tasks because it is easy to introduce an error and very hard to detect its origin.

Unlike NHibernate, the Data Access Block (DAB) from Enterprise Library makes calling stored procedures very easy and uniform. The DAB manages the state of existing database connections and also provides the required uniform way for data access operations, making all the code look and behave similarly [38]. Moreover, Enterprise Library supports multiple database types (Oracle, SQL Server, DB2, or Access for example): due to its abstraction level regarding databases types and due to the usage of multiple abstract factories design patterns, there is no need to change a single line of code if the database type changes.

Additionally, Enterprise Library supports applications using multiple databases and provides a simple way of choosing and alternating between all the configured databases and connection strings.

3.1.5 Communication Technologies

3.1.5.1 TIBCO RendezVous

TIBCO Rendezvous[16] is a software product from TIBCO Company that allows messaging interchanging between different applications. It is a very efficient, robust, reliable, scalable product and is the leading low latency-messaging product for real time throughput data distribution applications. It is a widely deployed, supported, and proven low latency messaging solution on the market today [39].

TIBCO Rendezvous can be integrated with external components and provides different Application Programming Interfaces (API) to support the development of applications in different programming languages.

[14] http://www.mygenerationsoftware.com
[15] http://www.codesmithtools.com
[16] http://www.tibco.com

The basic message passing is conceptually simple. A message has a single subject composed of elements separated by periods and has some message parameters, each one following the name-value-type paradigm [40]. The message is then sent to a single Rendezvous Daemon and a listener announces its subjects of interest to a Daemon (with a wildcard facility). The messages with matching subjects are then delivered to that Daemon [41].

The main components for an application using TIBCO Rendezvous are the following:

- the messages and their content parameters;
- the events related to the subscription, sending and receiving of messages;
- finally, the transport and the logical connection between different applications, which includes the connection settings.

3.1.5.2 YODA

YODA is a middleware software solution developed by Infineon Technologies[17]. YODA stands for Your Own Data Adapter and is a set of components and libraries which are application, platform and technology independent. These components and libraries have defined a set of well-defined rules and conventions [42].

YODA main goal is to achieve an efficient and complete integration between distributed applications, using a reliable and quick method to exchange messages between them. YODA is likely an internal network based protocol that allows different applications running on different platforms to intercommunicate and exchange information via YODA messages.

It is a high-level layer based on TIBCO Rendezvous software that allows applications to communicate through a network. YODA provides a uniform and easy way to send and receive messages. The main advantage of using YODA is that the communication between different applications is done by sending and receiving messages via a network and both the sender and the receiver applications do not need to know their locations on the network.

An application subscribes the messages by creating a transport, *IfxTransport*, and specifying the subjects it wants to receive. When a message is available in the network, it will be delegated accordingly to the applications that have subscribed that message subject. These applications only have to install an event handler, so that they can receive the delegated messages and process them.

[17] http://www.infineon.com

3.1.5.3 Microsoft Message Queuing

Microsoft Message Queuing (MSMQ) is a technology provided by Microsoft that enables applications running at different times to communicate across different networks and systems. The main advantages of this technology are that it guarantees message delivery, efficient routing, security and priority-based messaging. Additionally, it also can be used for implementing solutions for both synchronous and asynchronous messaging scenarios, which means it also supports systems that may be temporarily offline [43].

MSMQ is a middleware tool, not responsible for passing the messages themselves bit by bit; this middleware leaves that low level work to already existing standards and only provides a friendly interface API to help developers. Each computer participating in the distributed application needs a message queue, which allows the application to send asynchronous messages to a disconnected computer [44].

However, there is no need to use an advanced technology like MSMQ to support communication, since most of its features are not required by the framework. Additionally, all the communications between different applications are already supported by YODA, which is widely used in Qimonda universe.

3.1.6 Markup Languages

3.1.6.1 XML

XML stands for Extensible Markup Language and it is a W3C[18] recommendation. XML was developed by an XML Working Group (originally known as the SGML[19] Editorial Review Board) formed under the auspices of the W3C in 1996 [45].

XML is a simple and very flexible text format originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere [46]. XML is a markup language much like HTML, but XML is not a replacement for HTML since they were designed with different goals. XML was designed to transport and store data, with focus on what data is; HTML was designed to display data, with focus on how data looks [47].

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure. Unparsed data is made by contents that may or not be text, and if text, may be other than XML.

[18] World Wide Web Consortium
[19] Standard Generalized Markup Language

XML language can then be used to describe any kind of data type because tags are not predefined; used tags must be defined as needed, which makes XML to be classified as an extensible language. This characteristic makes XML documents self-descriptive because these documents are easier and intuitive to understand since they are relatively human-legible and reasonably clear [48].

Amongst its main purposes are the facility of sharing and transport structured data across different information systems (interoperability), the encoding of documents and the serialization of data. This data is stored in plain text format, which provides a software and hardware independent way of storing data. This makes XML straightforwardly usable because it is much easier to create data that different applications can share. Moreover, XML documents should not only be easy to create, but the design of XML documents should also be formal, concise and quickly prepared. These characteristics also help reducing the complexity of exchanging data between incompatible systems, since the data can be read by different incompatible applications [49].

3.1.6.2 XPath

XML Path or shortly XPath[20] is a language for finding information in an XML document. XPath is used to navigate through elements and attributes in an XML document [50]. In addition, XPath may be used to compute values (strings, numbers, or boolean values) from the content of an XML document. XPath became a W3C Recommendation in November 1999 and the current version of the language is XPath 2.0 [51].

XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. The XPath language is based on a tree representation of the XML document, and provides the ability to navigate around the tree, selecting nodes by a variety of criteria. XPath has a natural subset that can be used for matching (testing whether or not a node matches a pattern) [52].

XPath gets its name from its use of a path notation for navigating through the hierarchical structure of an XML document. XPath uses path expressions to select nodes or node-sets in an XML document. These path expressions look very much like the expressions you see when you work with a traditional computer file system.

3.2 Previous Work

This section concerns some of the previous work done regarding the data collection in manufacturing equipments used in an assembly line. It presents the state of the

[20] http://www.w3.org/TR/xpath

art in terms of how data is collected from files generated by equipments, how the questions related to database concurrency accesses are resolved and how communication is handled between both equipments and applications and also between different components of the same application.

3.2.1 Collecting Data from Files

Some manufacturing equipments generate data and save it into files. These files are usually saved in a folder configured in the data collection application settings. The folder containing these generated files is usually accessed via a network mapping using the TCP/IP protocol, which allows to access these remote files just like if they were in the same computer of the data collection application.

This way, the problem related to accessing these files is solved. However, data collection approaches used at Qimonda for collecting files have some limitations, especially because applications do not know neither the exact moment a new file is created neither when the file becomes available for use and unlocked by the equipment software. Consequently, a periodic approach must be used. Depending on the frequency equipment generates data, the time value used between two consecutive folder inspections is adjusted. Because of these periodic inspections, it is impossible to know *a priori* which files have been generated between two consecutive inspections, so each inspection needs to check *all* the existing files and folders existing in the mapped network folder. This way, a list containing the files and folders existing inside a directory must be kept in memory, so that comparisons between two consecutive inspections can be made. Additionally, the list should always be updated at the end of each inspection, so that it can be used in the next inspection.

Another important point related to collecting data from files is related to file contents. File contents need to be parsed and specific parsers must be configured to match the requirements of each specific equipment. These parsing approaches commonly used implement a sequential parsing of files, which leads to less tolerant parsers if errors occur and also makes harder to find the desired information inside the file contents.

3.2.2 Database Concurrency

Databases play a critical role in the data collection process: some of the data may be collected from equipment local databases and mostly because the main target for the data collected is usually a database.

However, database accesses should be handled carefully because there is the possibility of having many reads and writes operations using the same data rows at the same type. This may be potentially dangerous due to concurrency problems

related to concurrent accesses. These concurrent accesses could result in undefined and generally wrong data being stored.

The only way to solve these concurrency problems is controlling concurrent accesses by performing locks in the database records. However, even if locking seems to be the ideal solution, there is another problem rising: there are many approaches for implementing locking, all of them with different strengths and weaknesses, so it is not an easy at all to choose which one is more adequate to a specific concurrency problem [53]. Typically, the common locking approaches to solve this problem are pessimistic and optimistic concurrency control.

Pessimistic control performs locking at the database records level, either by locking when the row is selected or on demand, preventing other processes from holding the same lock. This pessimistic approach is both simple to implement and easy to use, since the users are immediately notified if they cannot access a database row. Another advantage is its security level, because locks are performed at database level very reliably and no one will be able to ignore the lock and change data [54].

The main disadvantages of pessimistic locking are its prone to deadlocks, excessive and long locking, and the use of extra database resources and finally the incompatibility of this mechanism in different types of databases. This concurrency control method is very powerful and it is suited for use in environments where there is heavy contention for data, where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.

Optimistic locking does not really occur at the database level and the data is never really locked like in the pessimistic approach: conflicts are detected and solved when writing to the database. While reading a row from the database, a "snapshot" is done and then, when a row needs to be updated, the application must determine if that row has changed or not since it was read by doing a comparison between the snapshot and the current values before writing [55]. This kind of approach does not use extra database resources, it is supported by all databases, it is easy to use and implement and has lower risks of deadlocks occurrence than the pessimistic one.

Although, this approach is not very secure, it is single-row oriented and slows down updates. Also, users are not notified if an update goes wrong and all applications updating a database must agree on a locking protocol for the columns involved, making maintenance harder to perform. This method to deal with concurrency is suited for use in environments with a low contention for data, with low updates actions and where the cost of rolling back a transaction is not critical.

Other approaches are locking on a column or application-level locking. Locking on a column approach is a combination of the two previous approaches previously described, since it combines some of their advantages by trying to reduce the negative

impact of their disadvantages. This kind of locking may be understood as a soft pessimistic or as a hard optimistic locking. Instead of directly locking database rows, this approach uses a field (a column) to indicate whether a row is being used (locked) or not [56].

This method lets the application decide when a lock should be done and also makes possible to always know who did the lock and when it has been done. However, this approach does not enforce a true lock and others may ignore it and change the data anyway. Also, locked rows may remain in such state forever if the locks are not removed manually and it requires extra database space and transactions to support the new column and to set the lock, respectively.

Application-level locking is another locking mechanism but it is the hardest one to implement and use. An updated list of the shared objects by all applications that update the database must be kept. Unlike the previous locking method, no extra space and processing is required at the database level. It also allows notifications while updating and it is surely the most flexible method described in this section. However, it slows down performance, especially with large amounts of data being updated and while synchronizing the list of objects with multiple applications running.

Since data collection in semiconductor industry deals with large volumes of data being updated and high contention, the optimistic method has been discarded due to performance requirements. On the other hand, choosing locking on a column and changing database schema to add a new column on each table would be completely unfeasible, not only because the actual systems high dimension, but also because it would require lots of extra database space. Application-level locking, with all the disadvantages already mentioned before is also out of question.

Pessimistic locking is the safest approach and the easiest to use and implement. It is suited for huge volumes of data and it is also the one that ensures better performance, so it is the adequate locking mechanism. Additionally, this locking approach to solve concurrency problems is natively supported by Oracle databases, which also already has some means to deal with the mentioned disadvantages. Oracle databases already have implemented mechanisms to solve deadlocks (by freeing a locked resource) and they also support timeouts associated with operations to face long locking problems.

3.2.3 Handling Messages and Communication

Handling messages and support effective communication between manufacturing equipments and data collection systems is another crucial point related to the data collection process. Data collection is a complex process that commonly involves at

least the equipment itself and its software to generate data, the application used to collect data from the equipment, a final target to save the data collected. Consequently, a mechanism to allow communication between all these different components involved in the process is also required.

Message handling at Qimonda is actually supported by a middleware application that uses a network protocol to allow communication between different applications and software components, even if these components run in different platforms.

Since specific solutions are developed for each equipment type considered, internal components of a data collection application are typically highly coupled. This way, there is no need to adopt advanced mechanisms supporting communication between these components.

To support message handling, events are another option to handle messages between internal components of a data collection application. Event-based programming is a simple and effective way to send and receive messages between components. Moreover, the .NET framework already supports natively events and delegating actions.

However, even if events are natively supported, there also some kind of messages that can not be handled by the .NET framework. In order to face these limitations and to guarantee that all kind of possible messages can be supported, Windows messages are also a possible solution. Windows messages are related to everything that is happening in Windows operating system, just like a simple mouse click or move, a key pressed, a window closed or even a shutdown action.

Windows messages provide a powerful mean to guarantee that every kind of message can be supported. However, working directly with Windows messages is clearly working at a very low level and it obviously increases development and implementation effort and it is also more time consuming than just working with simple events.

3.3 Summary

Figure 3.7 shown bellow summarizes the areas related to technology review used in the Bee Framework project. Some other technologies have been studied but only the ones presented in the figure have been used. Each rounded rectangle contained in the main figure represents an area as described in the Technology Review section (section 3.1).

For each of the areas represented in the top of the figure, there are as well two smaller rectangles: the first one represents the technologies themselves and the second one the IDEs or applications used to work with them. In the bottom of the

figure, the remaining areas are represented as well as the respective technologies for each area.

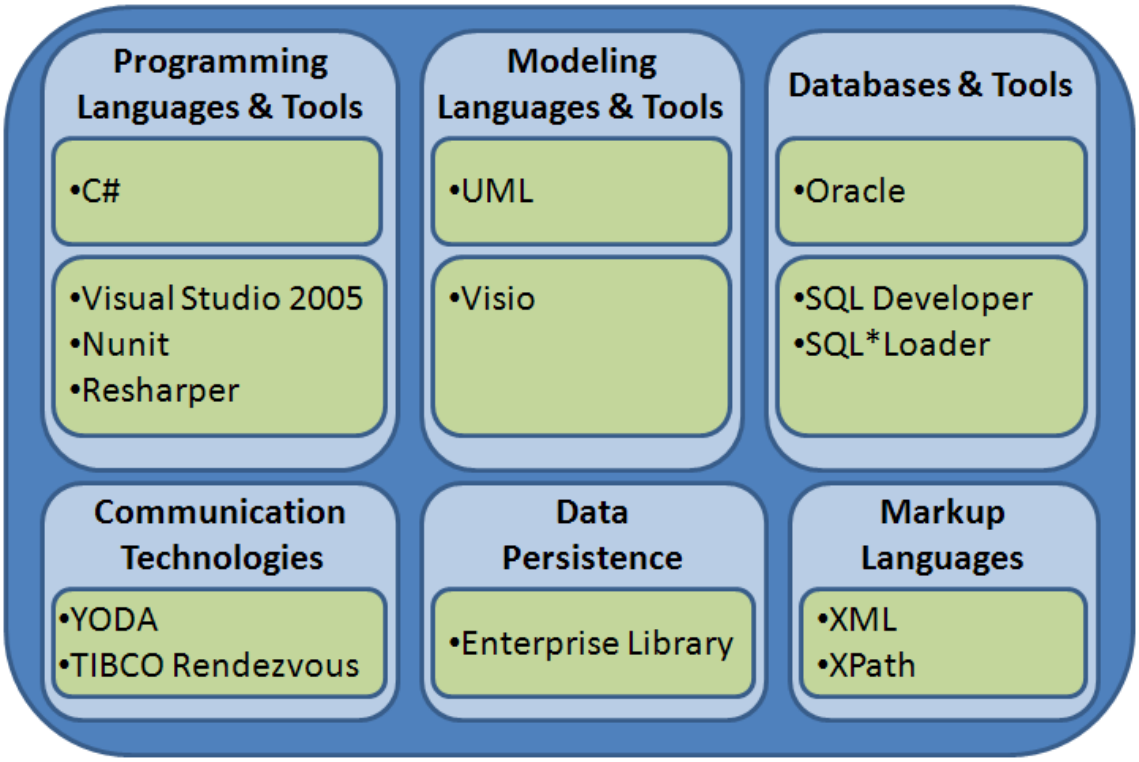


Figure 3.7: Technology review

The previous work related to data collection considered three different crucial points:

- Collecting data from files.

  Collecting files usually involves a periodic checking for new files in a listening directory. Parsing the contents of these files is typically done by using sequential parsing approaches.

- Database concurrency accesses.

  Managing concurrent accesses uses the native methods available in Oracle databases, which are directly related to the pessimist approach for locking.

- Handling messages and communication.

  Handling messages uses cross-platform middleware to ensure an effective communication between applications, even when considering applications and software components running in different platforms.

Chapter 4

Framework Specification and Architecture

The problem analysis explained in Chapter 2 has already exposed some of the existing problems related to equipment integration. The study of the previous work done while integrating equipments was also of great importance, since it has helped identifying some common problems occurring in integration tasks.

This chapter presents a detailed description regarding the proposed framework specification and architecture. Initially, the main focus of the chapter is related to the main requirements identified, especially regarding the data collection problem related to manufacturing equipments. The next section of this chapter refers to the proposed architecture to satisfy the requirements found. This architecture section also makes a comparison between the theoretical usage of design patterns and their real usage in the proposed framework architecture.

Sections 4.1 and 4.2 present a black-box and a white-box overview of the framework architecture. Black-box is a technical term for a system when it is viewed primarily in terms of its input and output characteristics. The opposite of a black-box is a system where the inner components or logic are available for inspection which is known as a white-box.

4.1 Framework Black-Box Overview

Since the Bee Framework refers to a manufacturing equipment data collection framework, the core requirements are obviously related to collecting data from different equipment types and saving it in different locations. Figure 4.1 shows a global overview of the Bee Framework collecting and saving data as a black-box.

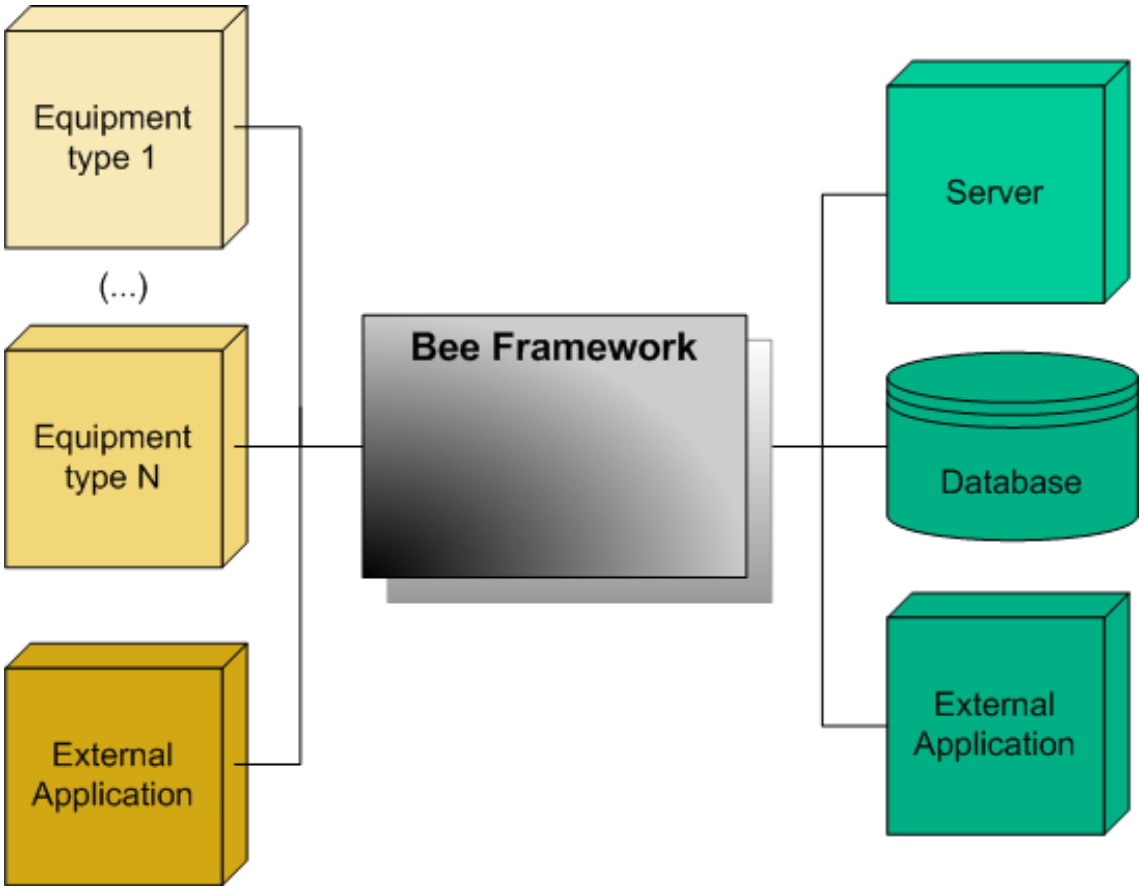


Figure 4.1: Black-box overview

On the left of the figure 4.1, different equipment types or other external applications generate data, which can be available through different data sources. The Bee Framework collects this generated data and then processes it. Finally, as shown in the right part of the figure, the collected data can be saved in different targets.

4.1.1 Collecting Data

The equipments may vary from manufacturer to manufacturer which consequently may result in different approaches to generate the data that will be collected. Such differences about the way equipments generate data also implies that data needs to be collected from different data sources and consequently different data collection approaches must be taken in consideration. These generated data may become available through:

1. Equipment local databases.

   Equipments generate data and store it in its own local database. In such situation, data information is initially stored in the equipment itself and data

collection will inevitably be done by querying the local equipment database in order to retrieve the available records that contain the required information.

2. Files in different formats (such as XML, plain text, proprietary files).

   Manufacturing equipments generate data and store it in the equipment local file system. These files need to be retrieved from the equipment file system and data collection will inevitably be done by reading, parsing and interpreting those files.

3. YODA messages.

   This possibility assumes that equipment generates data and immediately sends it using the YODA network, which as been described previously in the Technology Review section (see section 3.1.5.2). In such case, data is encapsulated into YODA messages and sent through the network.

4. TCP/IP and RS-232 (serial port).

   Finally, TCP / IP and serial port data collection approaches will not be considered because data collection using such data sources is currently being abandoned in manufacturing equipments.

4.1.2 Saving Data

After collecting the data using one of the approaches referred in the previous section, it needs to be saved so that it can be further used by other applications and feed them with the required information to follow-up on process optimization and control.

Saving data is the second problem that needs to be taken in consideration. Independently of how data is collected and what is the data information source, there are different ways to save the collected data. These different ways are listed bellow:

1. Databases.

   This is the common method used to save collected data. Databases are the natural approach because they are a reliable and efficient method to save data also allowing easy and quick access to information by performing queries.

2. File systems.

   File systems are used as a complementary method of databases. Commonly, when the data is available through files, these files are parsed to retrieve the required information. This information is then stored into databases and the files are moved to a server file system, keeping them in backup repositories.

3. YODA messages.

   The last approach considered encapsulates the collected data into YODA messages and sends these messages through the YODA network. These messages will then be collected by external applications subscribing these message subjects and the encapsulated data will then be used as defined by the external applications. The way these applications deal with the received YODA messages and the way they use the data received is out of the scope of this report.

4.2 Framework White-Box Overview

Collecting data from different equipment types and saving it into different target locations are the visible and final results of the usage of the Bee Framework . Consequently, when referring to data collection in an assembly line, the important questions to make by end users are:

- "Is data being generated correctly by the equipments?"
- "Is generated data being collected?"
- "Is collected data being saved?"

Obviously, for these end users data collection must be a black-box since they do not need to know the "how questions" about:

- "How is the data generated by equipments?"
- "How is the data collected?"
- "How is the data saved?"

These last "how questions" are the critical path of data collection and should be answered only by the equipment control team. This is where the framework plays an important role.

Figure 4.2 represents the white-box Bee Framework logical overview. This figure provides a better understanding about the global concepts and flow followed by manufacturing equipment data collection using the Bee Framework .

In order to face the different approaches for collecting data from equipments, it is needed a general interface to receive data from equipments (*Data Receiver*) and send it to the *Data Acquisition* view. When receiving data information, *Data Acquisition* checks configurations settings and validation rules for the specified equipment.

At this point, the framework has already started the data collection operations but it is still needed to store the data in some target location. However, there is

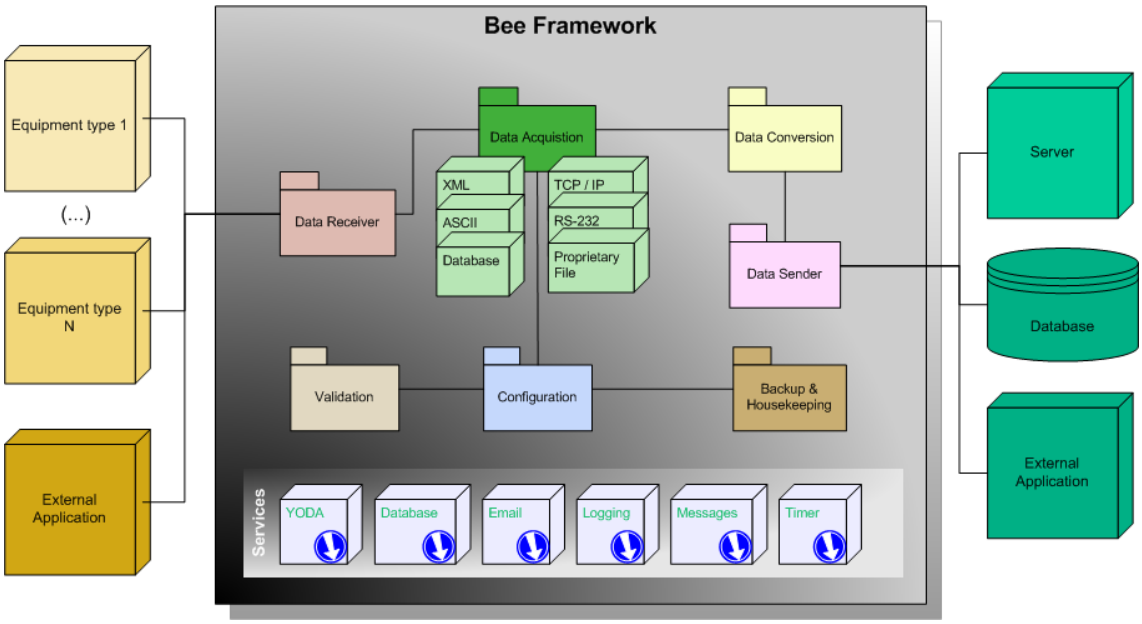


Figure 4.2: White-box overview

still another problem to solve before storing the data: usually, there is no need to save all the collected data and this data should be preprocessed and converted by the *Data Conversion* interface before being saved by the *Data Sender*.

The *Data Sender* is similar to the *Data Receiver*, with the inverse functionalities since it provides a general interface to save data. It analyzes the data format received from *Data Conversion* and according to the configuration settings applicable, it sends the data to the final target location. This target location has been previously determined by the *Configuration* interface while checking configuration settings.

The framework should also provide a way to do some backup and housekeeping operations, especially those related to server file systems and its log files. Performing backups of log files is important, but there is obviously no need to keep these files indefinitely. This way, *Backup & Housekeeping* interface should control how log files are backed up and how long they must remain available.

Another important point related to the framework architecture is the framework services provider. Those services are also shown on the bottom position of figure 4.2. They have not been considered framework modules since they are not related to the data collection logic flow itself; they provide the framework some very helpful functionalities that can be used by all modules, just by calling the desired service.

These services allow the following operations, for example:

- send or receive YODA messages;
- execute queries, stored procedures or transactions in a database;

- send emails;
- perform logging tasks;
- handle internal framework messages;
- install timers and generate alarm notifications.

These listed operations are just an overview to better understand the concepts related to each service and which are their meanings and purposes. A detailed description for each one of these services will be done in section 4.4.

4.3 Framework Architecture

The current section presents a global architecture overview of the Bee Framework. Figure 4.3 contains a class diagram model that represents the conceptual entities of the framework. Only the main entities are considered in the figure because the remaining entities are part of the framework services and are not considered critical to the overall architecture. These remaining entities will be referred in the Framework Services section (section 4.4).

The *BeeFramework* class is the framework main class and can be considered as being the kernel of the framework. This class is responsible for launching the framework itself and also all the selected modules specified in the configuration file. Moreover, this class is closely related to the message center provider, receiving not only messages from modules and YODA network but also sending messages to modules or to the YODA network accordingly.

Since this class is considered the kernel of the framework there is absolutely no need to have more than one instance of it. It must be ensured that this class has only one instance with a global point of access to it. Such description and characteristics correspond to a typical GoF — Gang of Four — creational design pattern, usually known as Singleton pattern [57].

Using a global variable would make the required object accessible, but by doing so it is impossible to guarantee that there is just a single object (singleton) instantiated. The solution is making the class itself responsible for its sole instance and ensuring that no other instance can be created, including by any threads that are running [58].

The Singleton pattern is implemented by creating a class with a method that creates a new instance of the *BeeFramework* class if one does not exist yet; if an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the constructor is made either private or protected [59].

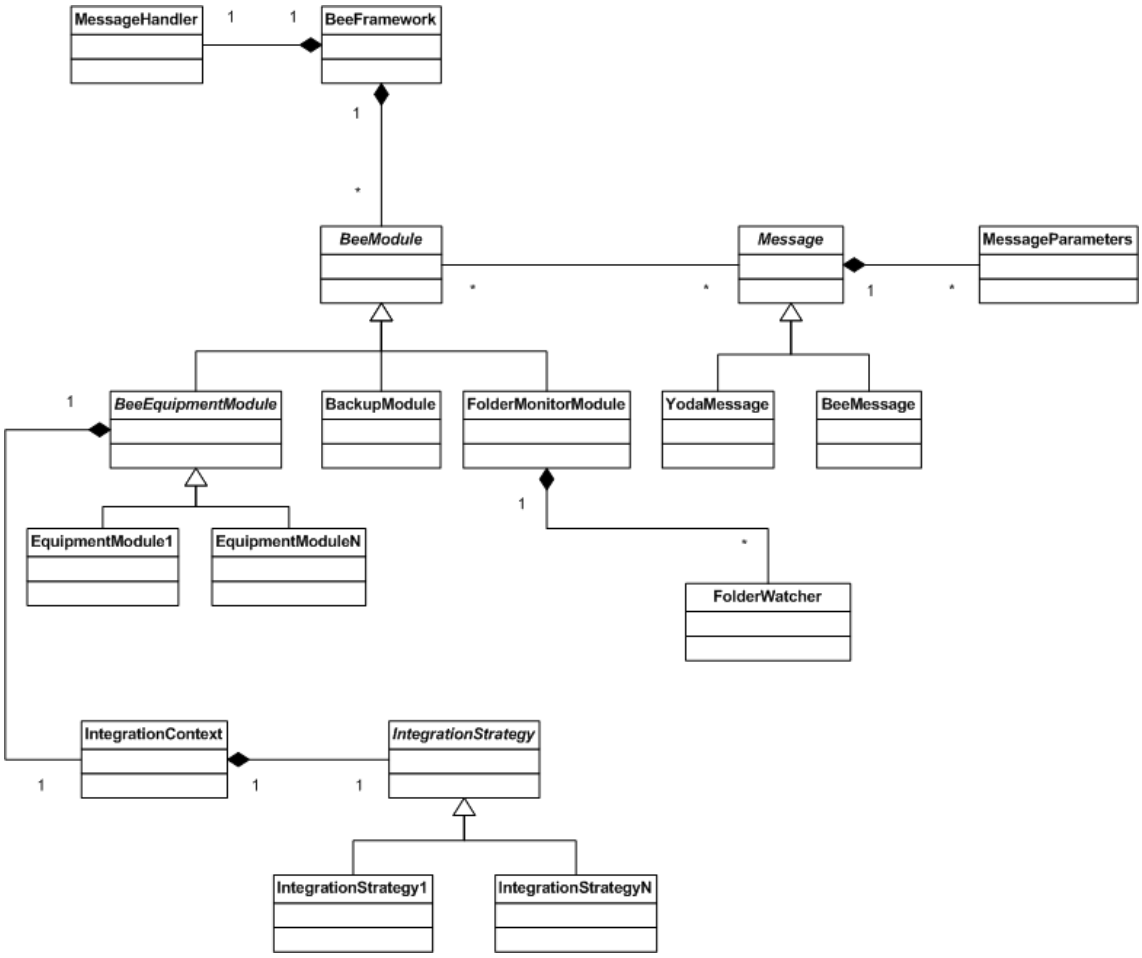


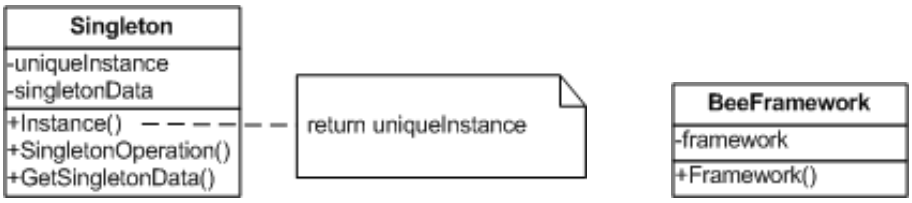
Figure 4.3: Framework architecture overview

This pattern avoids using global variables and allows refinement of architecture design. The power of the Singleton pattern goes beyond just controlling the instance count. As mentioned above, since access is controlled through a method, additional logic can be added behind them [60].

The singleton pattern structure and its architecture implementation in the framework are shown in figure 4.4: the GoF singleton design pattern is represented in 4.4a and 4.4b shows the design proposed.

The framework contains a hierarchy of distinct modules (see figure 4.5). However, even if each module has its responsibilities, behaviors and functionalities, there are many aspects that are common amongst the different modules, such as module instantiation, module configurations and module loading, for example.

Amongst the classes of the hierarchy, two main module classes can be considered as being the most important ones because both are parent classes: *BeeModule* and *BeeEquipmentModule*. *BeeModule* class is the parent class of all framework modules



(a) Gang of Four design, adapted from [57]

(b) BeeFramework design

Figure 4.4: Singleton pattern

and contains the common module methods; *BeeEquipmentModule* is the parent class of equipment modules classes.

The first level of child modules contains the following modules:

- *BeeEquipmentModule* — contains not only the methods and operations that are common to other framework modules but also the commons methods that are specifically related to equipment modules;
- *BackupModule* — contains the methods and operations to backup files and check for updates;
- *FolderMonitorModule* — contains the methods and operations to monitor directories and their content files.

The second level of the hierarchy comprises the equipment modules. Each module having the *BeeEquipmentModule* as a parent class allows data collection and equipment integration.

It would be nice if all the common characteristics of these modules could be handled the same way for all of them. If common characteristics are encapsulated into a parent abstract module, then all modules inheriting from the parent would have not only its own specific methods and functionalities but also those from the parent. Using such an architecture design, the *BeeFramework* class does not need to know all the specific module types shown in the hierarchy represented in figure 4.5 and can then deal with all of them using the same common methods.

This approach promotes an abstraction and makes the architecture related to the framework modules much easier to understand. Moreover, such an abstraction also makes possible to expand the framework easily. If a new module is required, the new module needs to inherit the parent module (the *BeeModule* class), which develops the necessary code to support the new module and integrate it in the framework. Since the main characteristics and behaviors of the new module remain the same, the new module works together with the other modules available. This allows for new derived modules to be introduced with no change to the code that uses the

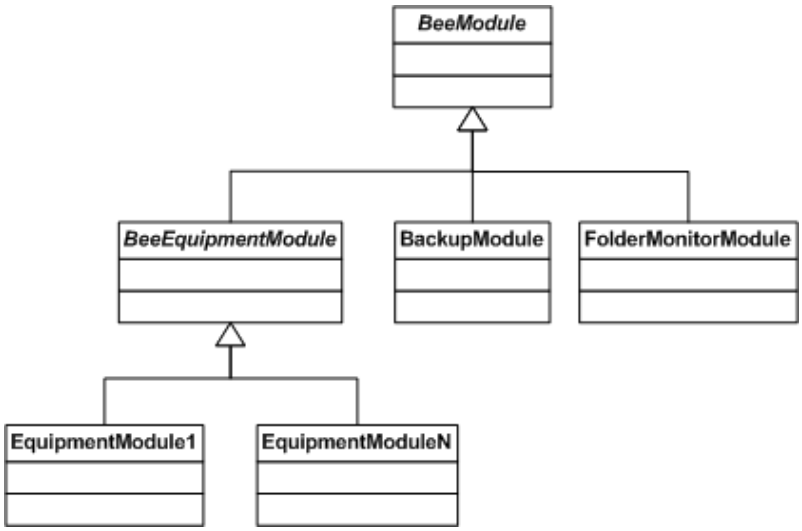


Figure 4.5: Framework modules hierarchy

parent class [61]. It wouldn't be necessary to change a single line of code in the framework kernel, since the common operations will be the same. Additionally, if a child module needs to define its own behavior, it can simply override the common method with its own implementation without changing the method call.

Such description and characteristics are common in software architecture and are widely used. Even if a large number of developers has already used this kind of abstraction, most of them probably do not know that they are facing another GoF creational design pattern, known as Factory Method. This pattern provides an interface for creating an object but let subclasses decide which class to instantiate. The pattern lets a class defer instantiation to subclasses, avoiding the need of specifying the concrete classes.

The Factory Method pattern is strongly recommended for this situation because the framework should be independent of how its modules are created, composed and represented. The framework only knows *when* a module should be instantiated, not *what kind* of module should be instantiated. The framework cannot anticipate the class of modules it must create and wants the subclasses to specify the modules it creates. The responsibility of instantiating a module is then delegated to one of modules subclasses.

Amongst the several benefits usually provided by using the Factory Method pattern, the most important ones concerning the framework are the isolation of concrete classes and the promotion of consistency among modules [62]. Since the factory encapsulates the responsibility and the process of creating modules, it isolates the framework from implementation classes because the framework will only manipulate abstract interfaces both when creating modules and when referring to common

methods [63].

The Factory Method pattern structure and its implementation in the framework are shown in the figures 4.6 and 4.7.

Participants of the Factory Method pattern shown in figure 4.6 are: [57]

- *Product* — defines the interface of objects the factory method creates;
- *ConcreteProduct* — implements the *Product* interface;
- *Creator* — declares the factory method, which returns an object of type *Product*;
- *ConcreteCreator* — returns an instance of a *ConcreteProduct*.

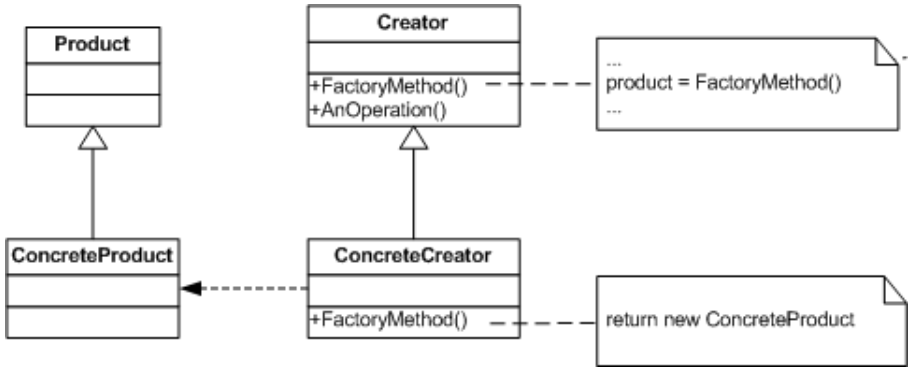


Figure 4.6: Gang of Four — Factory Method pattern, adapted from [57]

Figure 4.7 shows the analogy between the theoretical definition of the Factory Method pattern and its actual implementation in the framework.

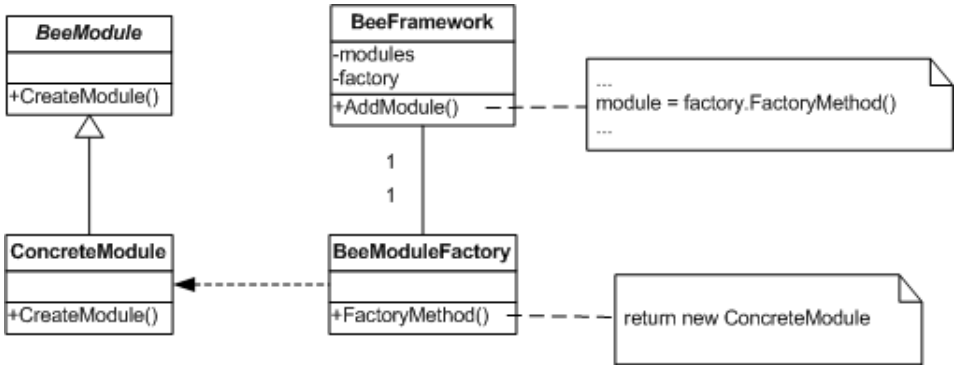


Figure 4.7: Bee Framework — Factory Method pattern

Similarly, the participants of the implemented Factory Method pattern are:

- *BeeModule* — defines the common interface of modules the factory method creates;
- *ConcreteModule*[1] — implements the *BeeModule* interface;
- *BeeFramework* — the application that contains the *BeeModuleFactory*, which returns a concrete object of type *BeeModule*;
- *BeeModuleFactory* — returns an instance of a *ConcreteModule* type.

In the following subsections, framework modules will be presented and described.

4.3.1 Folder Monitor Module

The *FolderMonitorModule*, as its name reveals, is the module used to monitor folder directories. Some equipments generate data and save it into different format files, such as XML, ASCII or proprietary formats. These files contain the data that needs to be collected and are saved into specific directories defined in the equipment configurations. This way, it is of high interest to have a module with the capability of monitoring one or more file system directories, being able to notify the framework whenever a file is created, renamed, deleted or changed.

If some of these actions occur inside a directory being monitored, this module will detect it and notify the *BeeFramework* class by delegating events containing the information related to the event detected. The *BeeFramework* class has already installed the handlers to catch these events sent by the *FolderMonitorModule* and broadcast them each time it receives a notification. These topics related to handlers, events, delegates and broadcasts are terms closely related to the framework message center and the way the framework deals with messages. This message center and this topics will be covered in the Message Handling section (see section 4.3.4).

A simple method to understand this module is to consider a database example: an event is fired and some operations are executed when an insert, delete or update action is performed in some table with a trigger associated.

The requirements identified for this module are listed bellow:

1. start and stop monitoring a directory;
2. detect changes related to created, renamed, deleted and changed actions;
3. receive messages and notify the framework about IO[2] changes;
4. list all files existing inside a directory;

[1] *ConcreteModule* is used just to simplify the diagram. It can be any of the existing leaf modules of the hierarchy presented in figure 4.5

[2] Input / Output

5. copy or move files;
6. allow regular expressions to filter files;
7. include subdirectories in folder monitoring.

These requirements and the characteristics of this module are related to another design pattern: the Observer pattern [64]. Although numerous variations of the Observer pattern exist, the basic premise of the pattern involves two actors: the observer and the subject. The logical association between these two actors is visible in figure 4.8. Whenever a change occurs in the subject, the observer *observes* this change and updates accordingly.

Figure 4.8: Relationship between Observer pattern actors, adapted from [64]

The observer pattern has been implemented in the framework as shown in the figure 4.9. *FolderMonitorModule* and *FolderWatcher* classes play the roles of observer and subject, respectively.

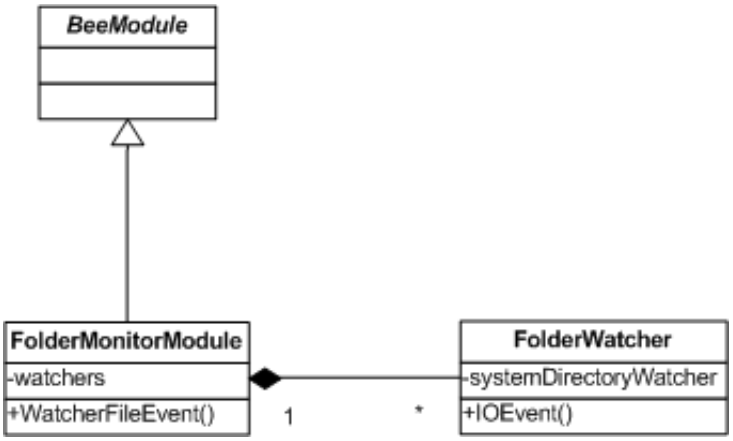


Figure 4.9: Implemented architecture of the Observer pattern

However, the proposed architecture of the Observer pattern is a slight variation on the base pattern, which is known as Event pattern. The Event pattern is an evolution of the Observer pattern. It is based on the usage of conventions related to event-based applications and event-based programming events, such as delegates, events and other related methods involved in event notification process.

In the current architecture design, when the *FolderWatcher* detects a change in the state of the directory being monitored it notifies the *FolderMonitorModule*

by delegating a notification message containing the change detected. By receiving the notification, the *FolderMonitorModule* will then know what kind of change has been detected and update its own state accordingly. Moreover, when the *FolderMonitorModule* receives the notification from one of its *FolderWatcher* objects, the module also notifies the *BeeFramework* and its message center. These notifications and messages will be explained in the Message Handling section (see section 4.3.4). This sequence actions and messages is visible in figure 4.10.

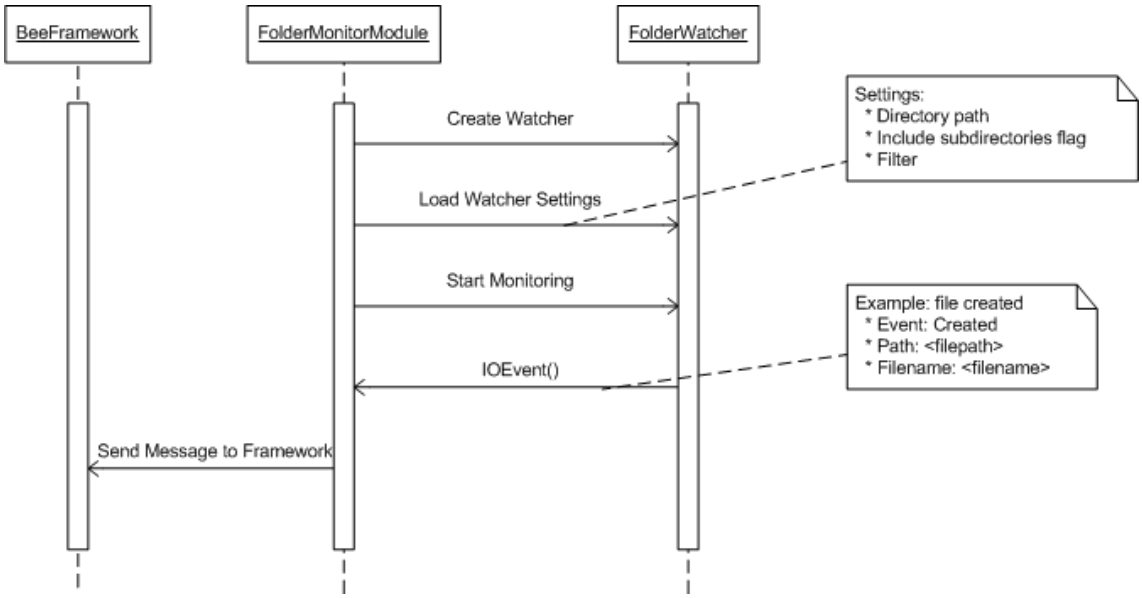


Figure 4.10: *FolderMonitor* sequence diagram

The main benefit of using this pattern is the abstract coupling achieved between subject and observer, the *FolderWatcher* and the *FolderMonitorModule*, respectively. The subject does not even need to know the concrete class of the observer, making the coupling between them minimal. Due to this low coupling between subject and observer, they can belong to different layers of the framework and the lower-level subject still can communicate and inform the higher-level observer [65].

Moreover, this kind of abstraction also implies a support for broadcast communication. Unlike a normal request, the notification that the subject sends does not need to specify the observer because the notification is automatically broadcast to all observers that subscribed it [66]. In this situation, all changes detected by running *FolderWatcher* instances are broadcasted to the single observer interested, the *FolderMonitorModule*.

4.3.2 Backup Module

As its name reveals, the *BackupModule* is used to backup files. This module is not critical and it is not directly related to data collection. It is used mostly with the two following purposes: backup files or check for updates.

The first purpose is quite simple: the module receives a message with the required details to backup a file. These details include the name and path of the file to backup, as well as the target destination. So, when such a message is received, this module only creates a message that will be sent to the *FolderMonitorModule*. The *FolderMonitorModule* receives the message and moves the file from its original location to the backup directory.

The second purpose is perhaps the most important feature of this module. The framework and its modules and services may use external assemblies to execute some operations. It allows other teams to develop some new unit features. Since these teams will only develop the code required for external assemblies, the framework keeps as a black-box and developers of these teams do not need to know how the framework works. Additionally, since they do not implement these features directly in the framework, the introduction of possible errors in the framework kernel is avoided.

The usage of these external assemblies should be used very carefully because it not only takes out the control from the framework scope (in extreme cases, a lot of primitive operations required may be defined in these assemblies) but also because the required code for invoking the methods contained in these assemblies becomes harder to understand. In fact, in order to use these assemblies, reflection[3] must be largely used, which introduces a high level of abstraction in the code: using reflection implies that classes need to be loaded at runtime and methods must also be discovered at runtime by examining the classes.

Since these assemblies are kept outside the framework, automatic updates of these assemblies without restarting the framework have been considered an interesting feature. The flow followed by these automatic updates is shown in figure 4.11.

The *BackupModule* uses the *FolderMonitorModule* to monitor a deployment directory. This directory is used to deploy new versions that will cause an automatic update. When the *FolderWatcher* that is monitoring the deployment directory detects a new file, a message is delegated to notify the framework and then this message is then forwarded to the framework modules, so that the *BackupModule* can receive and handle it. However, three possibilities have been considered about the files deployed in this directory:

[3]Reflection is the mechanism of discovering class information solely at runtime

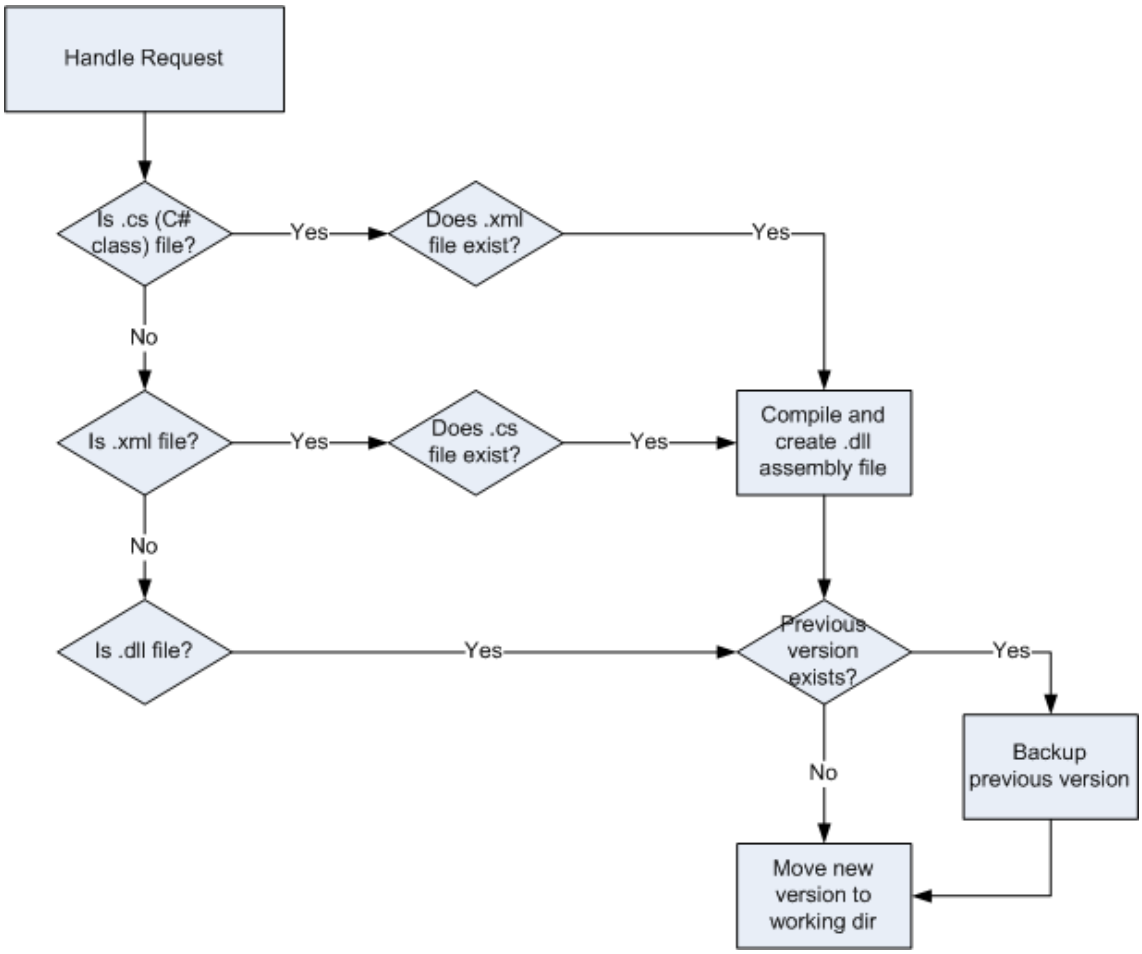


Figure 4.11: Automatic updates of external assemblies

- the file may be the assembly itself, already compiled and ready to be loaded and used;
- the file may be a class file;
- the file may be an XML file.

If the file is an assembly class (DLL), it can be directly used, so a backup of the existing version of the assembly is done and the new detected file is moved to the working directory. If the detected file is not a DLL file, then an assembly must be firstly created so that it can be used.

To generate the assembly dynamically, both a class file and a XML file are required: the class file contains the required code and the XML file contains the references used by the class. So, whenever one of these two types of files are detected, the *BackupModule* looks for both files in the deployment directory. If both files exist, the *BackupModule* loads the references from the XML file and compiles the class

file, creating an assembly. However, before moving the new version files detected from the deployment directory to the working directory, a backup of the existing files is made, just like in the assembly case.

4.3.3 Equipment Modules

An equipment module is a module used to define a strategy of collecting the data generated by a manufacturing equipment. Instead of having the *BeeModule* class as their parent, another abstract class has been considered: the *BeeEquipmentModule* class. The main purpose of using another abstract class as intermediate between concrete equipment modules and the basis class *BeeModule* is to provide an abstraction about the equipment and module types.

Since equipment module types are directly related to a set of aspects regarding the collection of data from equipments, inheriting from this intermediate class makes it possible to handle equipment modules and their common characteristics in the same way. Additionally, since the *BeeEquipmentModule* class also inherits from the *BeeModule* class, all the concrete equipment module types can have access to the common behaviors and characteristics of other modules. Figure 4.12 illustrates this hierarchy.

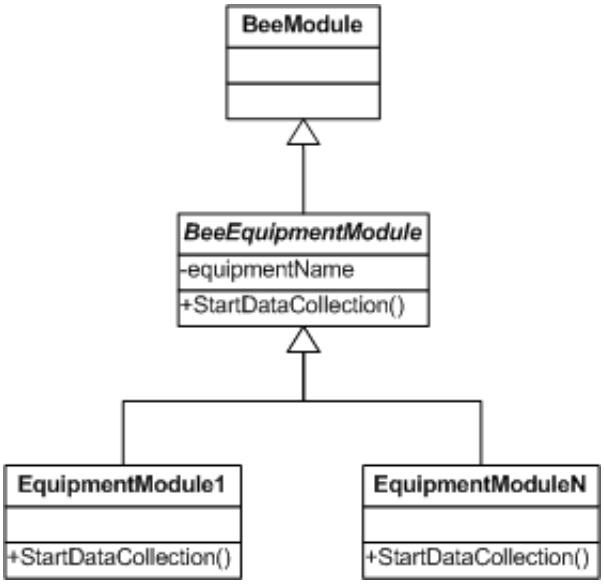


Figure 4.12: Equipment modules hierarchy

To start collecting data from equipments, the framework must analyze the type of all its instantiated modules. For each of its modules, a comparison is done to check if the module type matches the *BeeEquipmentModule* class: if so, the common *StartDataCollection* method is used to start data collection in the equipment

module; otherwise, the framework proceeds to its next module. Figure 4.13 shows the corresponding flow for this sequence of steps.

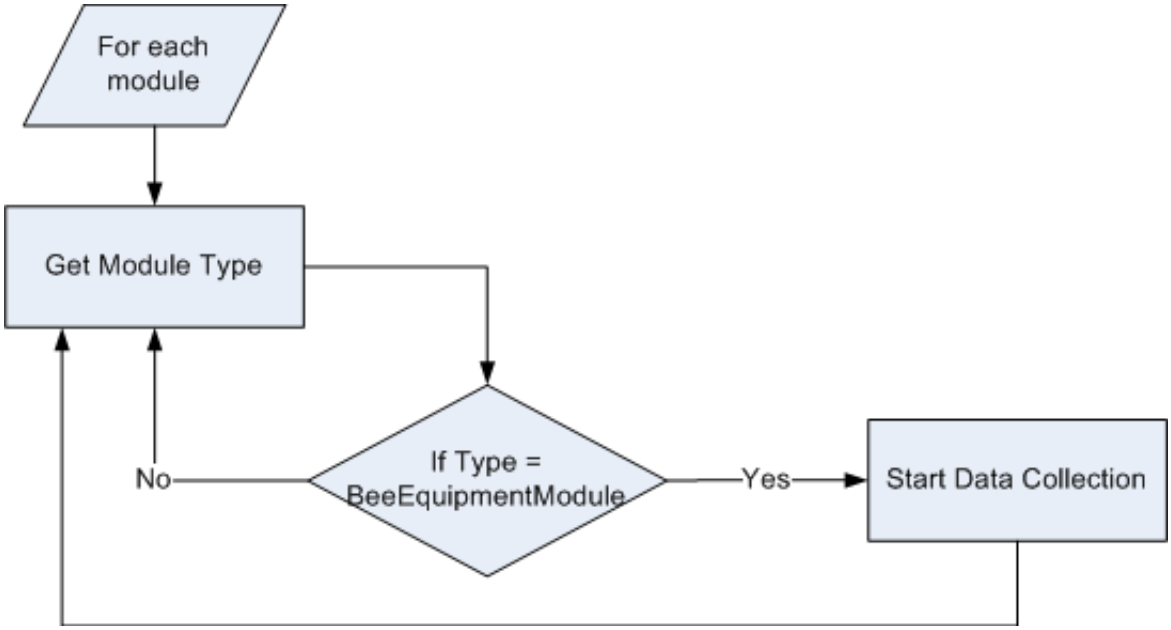


Figure 4.13: Start data collection flow

However, when referring to data collection, there are still two important points related to equipment modules that have not been considered yet:

- different strategies to collect data;
- small changes inside the skeleton of an algorithm.

4.3.3.1 Different Strategies to Collect Data

Suppose that different versions of the same equipment are available. These different versions may introduce some variations in the way data is generated and consequently in the way how this data should be collected, which would imply that completely different strategies could be needed to face these variations.

For example, what if an equipment type generates data and saves it into plain text files, while recent versions of the equipment saves the same data in XML files? In such situation, multiple strategies exist to perform data collection and both are required and appropriated at different times. This means that if the data collection algorithm is directly used into the equipment module class that requires it, not only the module becomes harder to understand but also becomes harder to change the desired algorithm. Moreover, it becomes difficult to add new algorithms and vary existing ones if data collection is an integral part of the equipment module.

These problems can be avoided by defining classes that encapsulate different data collection algorithms. An algorithm that is encapsulated in this way is called a *strategy*. These different classes are obviously related and differ only in their behavior. By encapsulating its behavior inside a strategy, an equipment module does not need to know the implemented algorithm and this algorithm is not exposed.

Such architecture corresponds to a behavior design pattern: the Strategy pattern. The classical structure of this pattern is visible in figure 4.14.

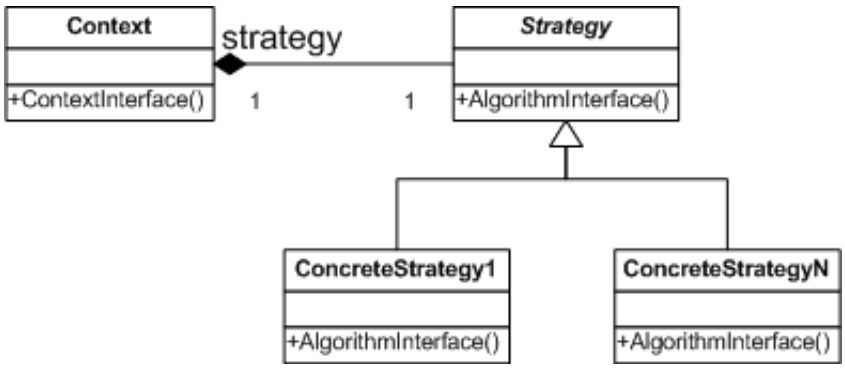


Figure 4.14: Strategy pattern, adapted from [57]

The *Context* class is configured with a *ConcreteStrategy* and maintains a reference to a *Strategy*. The *ConcreteStrategy* class implements the algorithm using the interface declared by the *Strategy* class and supported by all algorithms. This way, *Context* can call the desired algorithm using the common *AlgorithmInterface* method, without caring about the concrete strategy encapsulating this algorithm. Both *Strategy* and *Context* classes interact to implement the chosen algorithm and the context passes all data required by the algorithm to the strategy when the algorithm is called.

For the Bee Framework , this pattern has been implemented as shown in figure 4.15.

Each *BeeEquipmentModule* (and consequently each concrete equipment module inheriting from this class) contains an *IntegrationContext*. This context defines an *IntegrationStrategy*, decoupling the algorithm used to start data collection from the equipment module. Since the *IntegrationContext* needs to interact with the *IntegrationStrategy* so that the data required to execute the algorithm can become available to the strategy, two additional methods have been defined to configure a bidirectional communication between both classes. This way, each time a *BeeEquipmentModule* requests to start data collection, it exclusively interacts with the context and this last one forwards the request from the equipment module to the strategy.

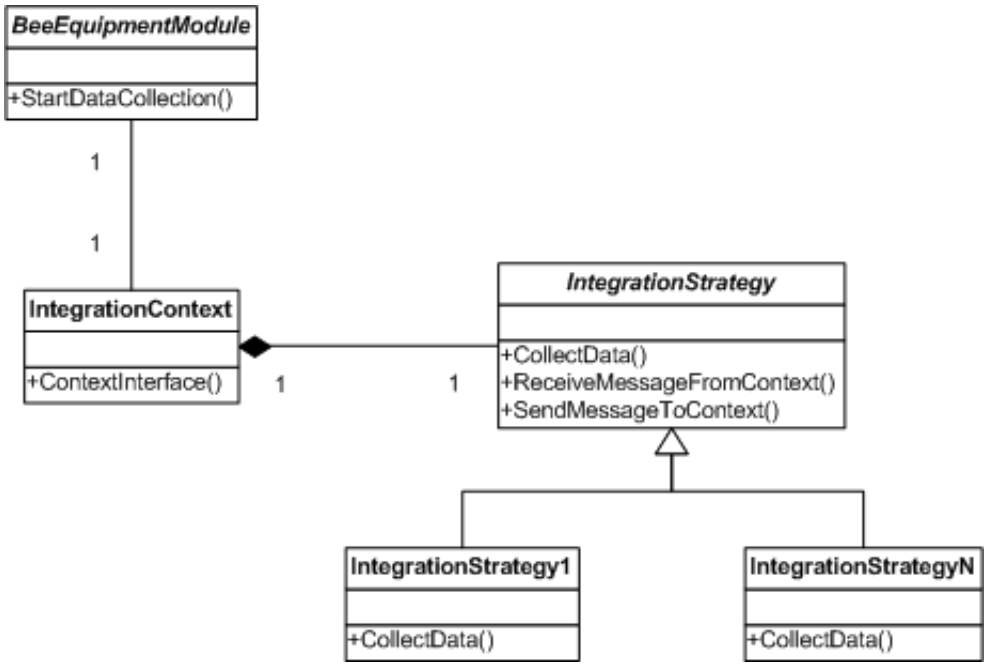


Figure 4.15: Implemented architecture of the Strategy pattern

4.3.3.2 The Skeleton of an Algorithm

The usage of the Strategy design pattern brings some advantages, especially because it promotes an abstraction by decoupling the data collection strategies from the equipment modules and by allowing the definition of the algorithm used for data collection in each strategy. However, what if this algorithm needs some few adjustments or improvements? Does this mean that a new integral strategy is required?

To answer the previous questions, another behavioral design pattern has been used: the Template Method pattern. This pattern can be very useful because it allows the definition of a skeleton of an algorithm in a strategy, deferring some steps to subclasses. The interesting point about using this pattern is that the algorithm remains the same: only some steps are redefined in subclasses. Figure 4.16 illustrates this pattern architecture.

The *CollectData* method is called the template method. It defines the algorithm in terms of the operations it must do and lets a subclass override these primitive operations. This way, small changes of an algorithm are allowed but the skeleton of the algorithm itself remains exactly the same. The *IntegrationStrategy1* class implements the skeleton of the algorithm and defines the primitive operations used by the algorithm. Next, if some of the steps of the algorithm need some variations, a new class *IntegrationStrategy1'* is defined and overrides just the steps that should

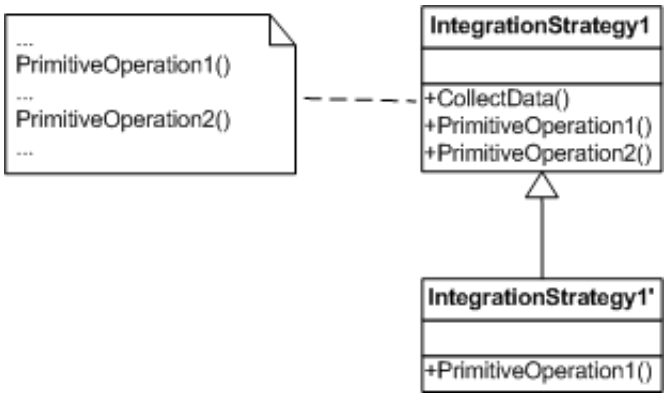


Figure 4.16: Template Method pattern

vary.

The implementation of such design pattern in the framework architecture brings some great advantages. It avoids the definition of a complete new strategy if only some small parts vary inside the algorithm behavior. Consequently, the invariant parts of an algorithm are implemented once and the behavior that can vary is left to the subclasses.

Such architecture design is a fundamental technique to promote code reuse. This technique is based on the Inversion of Control[4] and follows the *Hollywood Principle* — "Don't call us, we'll call you" — a software design methodology that takes its name from the *cliché* response given to amateurs auditioning in Hollywood [69]. The basic idea of this principle is that a class says what it does by implementing an interface and what it needs by requesting interfaces. Then, the framework decides when to create it and what concrete instances to give it [70].

4.3.4 Message Handling

Message handling is a crucial point in the framework architecture. In order to guarantee a robust architecture, framework modules and services should be divided into different components, promoting a framework with a modular architecture. However, separating these components introduces a important difficulty: the components cannot be tightly coupled, but they still need to have some kind of interaction between them.

Messages cannot simply be sent by direct invocation because this approach leads to a tight coupling. Moreover, such approach will cause the architecture to become harder to maintain if the number of interactions between framework components

[4]Inversion of Control is an abstract principle describing an aspect of some software architecture designs in which the flow of control of system is inverted in comparison to the traditional architecture of software libraries [67, 68]

increases significantly: changing a single component will probably cause changes in all other components directly interacting with it. Obviously, this approach cannot be considered scalable and expansible.

In order to promote an architecture with low coupling between framework components, an uniform way to send and receive messages is required to allow efficient communication both inside the framework domain and even with external applications. As described in section 4.3 the proposed architecture already considers an uniform way to deal with all modules using the same common methods, so it can be expanded to include the support of message handling. Figure 4.17 shows the architecture that makes message handling possible.

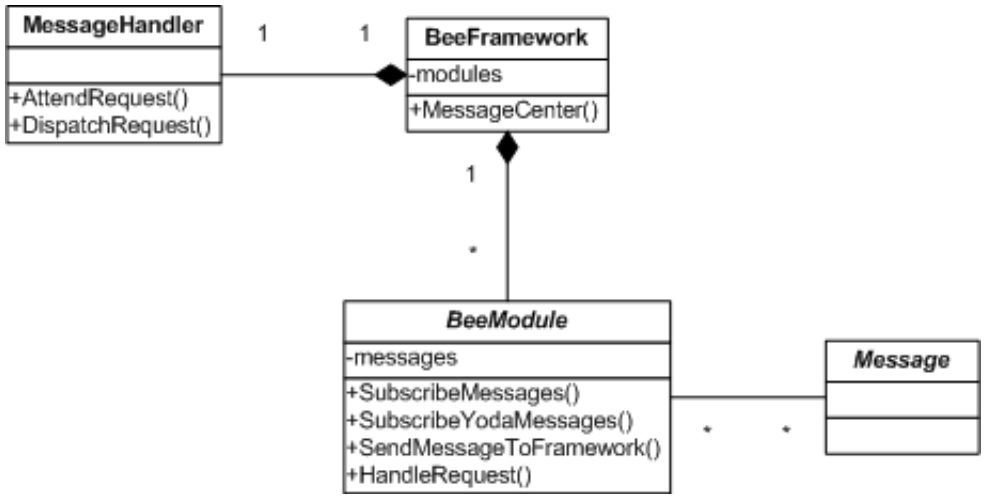


Figure 4.17: Architecture that supports message handling

Since all modules have the *BeeModule* as a parent class, then all modules inheriting from this class will also have its parent module methods available. These common methods can then be used globally by every single module to subscribe all the messages it wants to receive, i.e. the messages that it should receive when they become available inside the framework domain or via the YODA network. Additionally, these methods should not only allow each module to receive its desired messages but also to send its own messages to the framework message center. The *BeeFramework* class contains the message center, which is jointly used with *MessageHandler* class to control the flow of messages.

Figure 4.18 shows the sequence of executed actions when modules are instantiated.

Each module has its own configuration file and each file contains the message subjects that the module must subscribe so that it can receive the messages with those subjects. By doing so, the responsibility of handling messages is translated

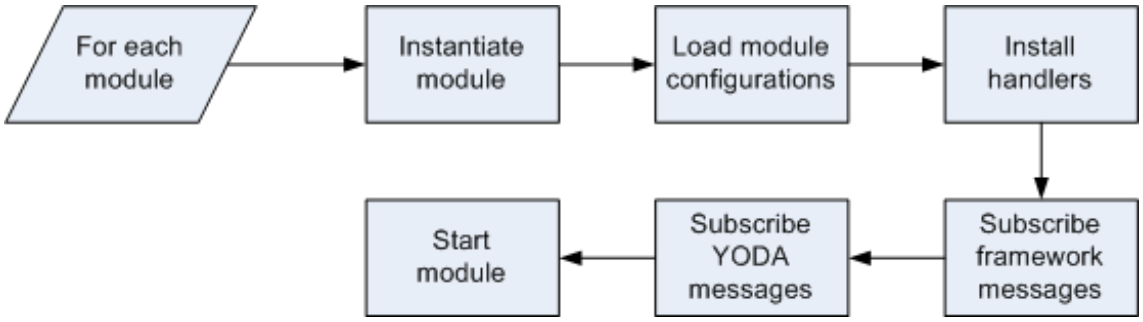


Figure 4.18: Flow of actions performed when instantiating modules

from the framework kernel and its message center to the modules. Hence, if a module needs to subscribe a new message there is no need to change the basis and nuclear classes of the framework itself; only the required code to handle the new message is needed in the module that wants to receive that new message and perform the actions related to the received message. Consequently, with such architecture the coupling between all the modules themselves and between the modules and the framework is considerably reduced.

BeeModule class uses a common event and the common *SendMessageToFramework* method to delegate a message to the framework message center existing in the *BeeFramework* class. Since the message is always delegated using the common event of *BeeModule*, no particular and concrete modules are considered and the framework just needs to deal with this single event instead of dealing with multiple events (one for each module). The *BeeFramework* class installs this common event to catch the delegated messages sent through events from its modules and has the *MessageCenter* method to firstly receive the messages. Next, this method uses the *MessageHandler* class to attend and dispatch every received message, broadcasting it to all modules interested in receving it.

Once again, this approach helps promoting scalability and reduces framework maintenance effort required. It not only helps in the goal of achieving a good design architecture but also keeps code cleaner, making it easy for developers to implement and debug the code: messages are sent and received uniformly, not caring at all neither about who is the sender module neither about who are the possible receiver modules.

Figure 4.19 shows the hierarchy used for messages and their composition in terms of parameters.

Framework messages and YODA messages, *BeeMessage* and *YodaMessage* classes respectively, are the two support types of messages. These two different types of messages are in fact very similar and both have the *Message* class as their parent

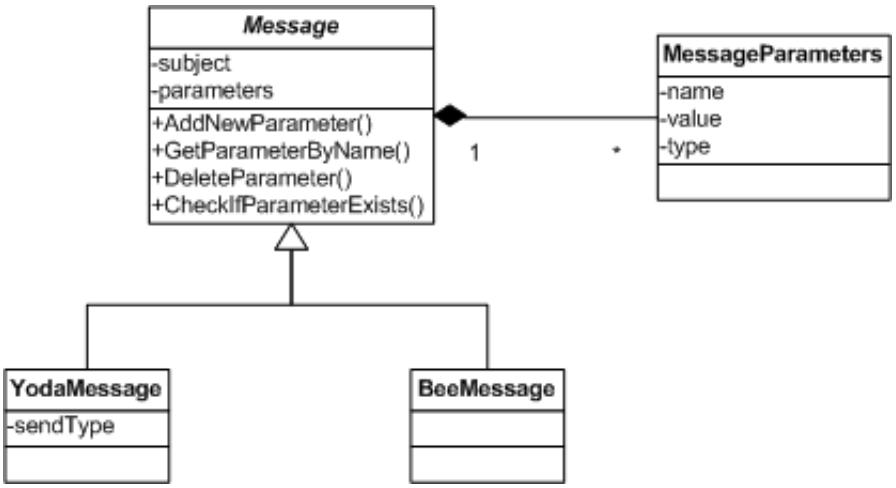


Figure 4.19: Messages hierarchy and parameters

class. The biggest difference between these messages is the fact that YODA messages can be used to exchange information with external framework applications. Moreover, YODA messages need to specify the sending type, because they can just be a publish, a request or a reply message. Framework messages do not need to specify this attribute value because they are only used internally and consequently controlled by the message center of the *BeeFramework* class. Moreover, since there are just a few differences between both types of messages, a *BeeMessage* can be easily converted into a *YodaMessage* and vice-versa.

Each *Message* contains a set of parameters that follow the name-value-type paradigm, allowing an abstraction regarding the parameters supported by messages. Instead of guessing how many parameters a message may have and which parameter types may be possibly used, this paradigm abstraction only requires a *Message* to have a set of parameters that can be uniquely identified which can be achieved if using different names. With this abstraction, a mapping between the parameter name and both parameter type and value is then established and a *MessageParameter* can encapsulate any object type. The main advantage of this abstraction is that a message can handle all parameters exactly the same way, without taking in consideration if the parameter represents a string, a boolean, an integer, or even a complex value. To get the value of a message parameter, the parameter must be retrieved by its unique name and a cast considering the parameter type should be done.

However, there is still a set of questions related to message handling not yet considered in the approach described in the previous paragraphs, namely:

- How to broadcast a message to the framework modules interested in receiving it?
- What if a module wants to send a message to the YODA network? How to send it?
- How does a module receive a message from the YODA network?

4.3.4.1 Broadcast a Message to the Framework Modules

As described previously in this section, when modules are being instantiated, each module subscribes the list of messages it wants to receive. However, as also has been said before, a low coupling between modules is required, so this low coupling also should be taken in consideration when referring to messages. Senders should not be coupled to receiver modules and the relationships between senders and receivers should also not be of the responsibility of the framework. The low coupling required can be achieved by giving each module the chance of handling a request. It would be better to have a chain of modules and pass a message along the chain, so that all modules can have the chance to handle it.

Creating a chain of modules may seem a contradiction because it seems modules are being coupled together. However, to build this chain of modules there is no need to specify concrete modules classes: the only class required is the abstract *BeeModule* class, which is the parent class of all modules, keeping the abstraction intact and avoiding the direct coupling between modules.

Such architecture of building a chain of modules and passing requests (messages) along the chain corresponds to a behavioral design pattern: the Chain of Responsibility pattern. The main idea of this pattern is to decouple senders and receivers by giving multiple objects an opportunity to handle a request. The request gets passed along a chain of objects until one of them handles it. This pattern promotes the idea of loose coupling, which is considered a programming best practice [71].

This design pattern is recommended to broadcast a message to the framework modules interested in receiving it because more than one module may handle a request, the handler is not known *a priory* and the receiver should not be directly specified. Figure 4.20 shows the architecture of the Chain of Responsibility pattern.

The *Client* initiates the request to a *ConcreteHandler* object on the chain. The request is then passed along the *ConcreteHandler* objects of the chain. Each of these objects forward the request to its *ConcreteHandler* successor. When receiving a request, each *ConcreteHandler* can then decide if it is or not responsible for handling the request. The typical sequence followed by a request in the chain is illustrated in figure 4.21.

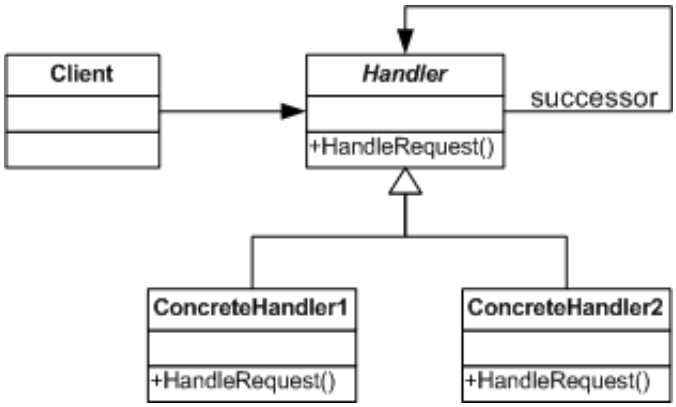


Figure 4.20: Chain of Responsibility pattern, adapted from [57]

However, this pattern has been implemented with some differences. Usually, a request passes along the chain only until an object handles the request. In the framework architecture, this is not true and a request passes along *all* the chain because it can be eventually be handled by multiple modules. The implementation of this pattern without following its classic restrictions is visible in figure 4.22.

Each module inheriting from *BeeModule* has its own method — *HandleRequest()* — to handle a request, implementing the abstract method of their base class, which defines the common interface for handling requests (just like *Handler* class in figure 4.20). The *BeeFramework* class is the *Client* that initiates the message in the chain and passes it to the first concrete module. This module checks if it can handle the message and also passes it through the chain so that other concrete modules can receive the same request.

When a module wants to send a message, in order to guarantee that it starts at the begin of the chain, it has to delegate it to the message center of *BeeFramework* class. This message center will then initiate the request in the chain. An example of this situation is illustrated in figure 4.23. Suppose that *FolderMonitorModule* wants to send a message that only *EquipmentModuleN* can handle. The message is delegated to the *BeeFramework* using the *SendMessageToFramework* method. Then, the message is initiated in the chain and passes through the *BackupModule*, the *FolderMonitorModule* and finally reaches its destination because *EquipmentModuleN* can handle it.

Even if this pattern primary benefit is that it provides loose coupling between the sender and the receiver, this pattern has other important benefit: using this pattern implies an increase of flexibility in assigning responsibilities to objects. It makes it easy to add a new potential receiver for a message or new ways a message can be received. This way, adding or changing responsibilities for handling a request

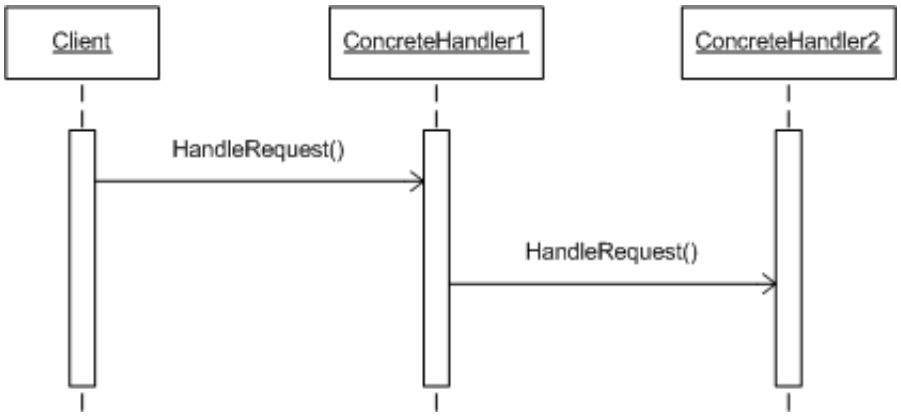


Figure 4.21: Sequence followed by a request in the chain

becomes easier because responsibilities are distributed among objects. However, the chain of responsibility must be handled carefully because there are no guarantees that a request will be handled by a receiver if the chain is broken or not specified properly [72].

4.3.4.2 Send a Message to the YODA Network

This section only intends to give an overview of how the proposed architecture is used to send a message to the YODA network. The details related to the different available options of sending a message to the YODA network will be presented in the YODA Service section (see section 4.4.1).

The main reason for the existence of the *MessageHandler* class is to provide an interface with the YODA network. Of course, all modules can send a message to the YODA network, but since these messages will be sent outside the framework, it is convenient to have a single point of access to send the messages instead of having multiple ones, one for each module.

Whenever a module wants to send a message to the YODA network, the *Message* type used is the *YodaMessage*. So, similarly as previously described for internal framework messages, the module uses the same *SendMessageToFramework* method to delegate the message to the framework message center. However, instead of initiating the received message in the chain of modules, the *BeeFramework* detects the message as being a *YodaMessage* and uses the *MessageHandler* and its YODA configuration settings to send the message to the YODA network.

4.3.4.3 Receive a Message from the YODA Network

Just like has been described in the Send a Message to the YODA Network section, receiving a message from the YODA network also uses the *MessageHandler* class as

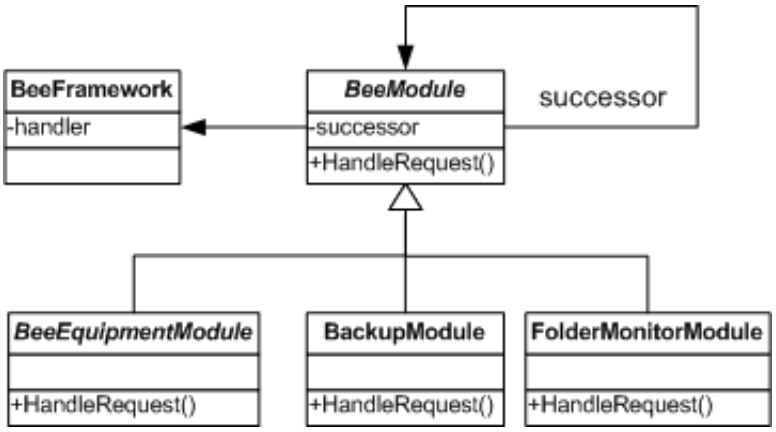


Figure 4.22: Implemented architecture of the Chain of Responsibility pattern

an interface. The purpose is also exactly the same, to avoid each module to have its own direct interaction with the network. When the modules are instantiated, each module subscribes the YODA subjects it wants to receive. These subjects correspond, obviously, to the messages that can be handled by the module which has subscribed them. However, there are not the modules who perform these subscriptions; The *MessageHandler* is used to subscribe them. This means that only the *MessageHandler* has subscribed YODA messages, which makes it the only access point to receive messages.

When the *MessageCenter* receives a message from the YODA network, this message should be broadcasted to the modules. However, after reading the previous paragraph, it becomes obvious that the module(s) which subscribed the message with the subject received aren't unknown at this point. The approach used to receive messages from the YODA network takes advantage of the architecture for handling messages previously described. The use of the Chain of Responsibility pattern helps solving this problem: the low coupling achieved by the usage of the pattern implies that there is no need to know *a priori* who the receivers are.

So, after converting the received message into a *BeeMessage*, this converted message is broadcasted to the modules just like a framework message. The only difference in these approaches regarding internal framework messages and YODA message is the source of the message: when referring to internal messages, the source is a framework module and the message is sent from its source using the *SendMessageToFramework* method; when referring to YODA messages, the source is the YODA network and the *MessageHandler* catches an event when a YODA message with a subscribed subject becomes available in the network.

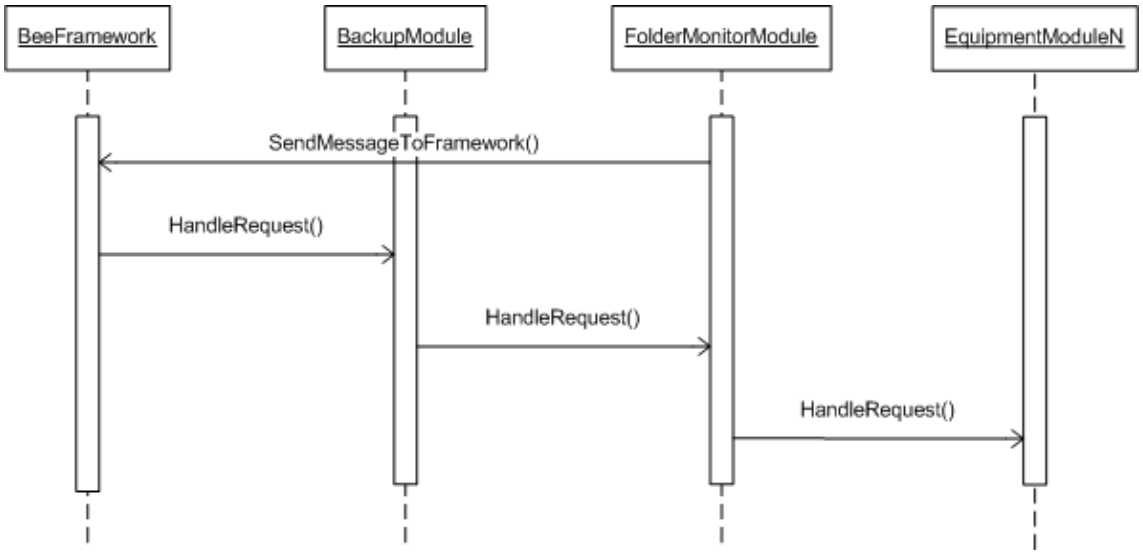


Figure 4.23: Example of broadcasting a message in the chain

4.3.4.4 Framework *vs.* YODA Messages

After reading the previous section, you will probably conclude that framework and YODA messages are very similar. Both need to subscribe message subjects to receive the corresponding messages, both need to publish and delegate messages so that they can be caught and processed by others, both use the same base and typical message structure composed my parameters, both use the standard name-value-type paradigm to describe each parameter.

YODA is a software that is surely in a very mature state of development, use and stability, especially if compared with the framework internal messages. Also, being a middleware widely used by several running applications at Qimonda and having full time teams maintaining and supporting it, YODA would be a natural choice to support all message handling, easily replacing internal messages.

A question can raise then: why does the framework need both types of messages support? It seems a pointless attempt to "reinvent the wheel", adding no value and wasting time, increasing framework complexity and turning it difficult to maintain and support. The usage of both approaches is related to important issues that need to be considered, especially those related to performance.

Being so identical, the abandon of internal framework messages has even been considered because YODA seemed to be a good and effective choice to support message handling. However, some charge tests using both approaches under the same conditions have been done and the results were quite impressive and concluding. These charge tests are described next.

Framework module 1 send a message to module 2, using both approaches. This message only contains two parameters:

- a string attribute type with the same name and value fields;
- a date/time attribute type which value corresponds to the sending time.

In both approaches, as soon as module 2 received the message sent by module 1, it checked the current time again and perform a subtract operation. In order to have a wide range of results, these tests have been done for a thousand messages and the times have been measured. The result times achieved while using YODA messages were about three times higher than while using framework internal messages. The average result time for sending a message using internal messages was approximately 30 milliseconds and using YODA around 90 milliseconds.

Internal framework messages just use events and delegates to handle messages, so it is a direct way of sending and receiving messages within the framework and its modules. Messages using YODA need to use a network, so the introduced overhead by sending and receiving messages via a network is naturally higher than the existing one related to internal messages. Additionally, in order to receive a delegated message, even using YODA services, there is still needed to install an event handler, just like it is done using internal messages.

There is also another disadvantage about using YODA. Since the messages are sent or published in the network, the control of handling messages wouldn't be done just by the framework itself. If not used carefully, some other applications could also be listening the same message subjects and perform some undesired actions.

These two possibilities should then be supported, giving the framework a high level of flexibility and a wide range of possible future features to add. Moreover, since both framework and YODA messages are similar, the complexity of understanding and supporting both does not increase much because the messages structure is identical.

4.4 Framework Services

This section presents a detailed specification of the requirements identified. It details the requirements of each service previously referred in the White-Box Overview section (section 4.2). The current section only focus on requirements. It does not take in consideration issues related to architecture or implementation details of the services. For further details related to framework services configurations, please consult Appendix B.

4.4.1 YODA Service

YODA is a very important middleware solution that is widely used at Qimonda assembly line and that allows a high level of integration between manufacturing equipments, servers and a large number of different applications running in different operating systems. So, referring to a data collection framework to be used in this assembly line without considering an effective support to YODA would cause the framework to have a lesser impact.

The main requirements identified for this framework service are listed bellow:

1. create transport configurations for messages;
2. subscribe / unsubscribe message subjects;
3. send messages to the network using different sending types:
   - publish;
   - blocking reply-request;
   - unblocking reply-request;
   - reply.
4. receive messages from the network;
5. convert YODA messages into framework messages;
6. retrieve configurations setting using the Configuration Manager.

4.4.2 Database Service

Database Service is perhaps the most important Bee Framework service. This service assumes a great importance because it is both related to the sources and targets of collect data. It can be used either to collect data from equipments (sources) or to save collected data into target databases.

This service should be able to:

1. create database instances;
2. create database connections;
3. open and close database connections;
4. execute queries;
5. execute stored procedures;
6. prepare stored procedures;

7. use binding variables;
8. open, commit and rollback transactions;
9. execute queries inside open transactions;
10. load large volumes of data from text files using SQL*Loader.

4.4.3 Email Service

Email Service main purpose is sending emails using a defined SMTP valid configuration. Email Service is not a core service of the framework, but can be very useful for example to send a notification if data collection in some equipment is not working as expected.

The two identified requirements for the Email Service are:

1. load configurations and define SMTP host settings;
2. send emails using SMTP configurations.

4.4.4 Logging Service

Logging Service is a complementary and very useful service of the framework. This service allows the framework to write messages about the application running and the data collection itself in log files, so that this information can be available further to manual consultation. Register events using the logging service can then be used to track erroneous situations and plays an important role by providing an audit trail that can be used to diagnose problems.

The requirements identified for the logging service are:

1. support different message severities / priorities;
2. support multiple logging categories;
3. allow different types of log file rolling[5];
4. control the number of files by specifying a maximum number of log files;
5. support different age units and control maximum age of log files;
6. support different size units and control maximum size of log files;
7. allow backups of oldest log files.

[5]Rolling is a combination of rotation and translation operations used when adding a new log entry. Oldest entries are translated (or deleted if necessary) in favor of new entries, keeping a log file always updated with the recent entries

4.4.5 Framework Messages Service

Framework Messages service is similar to YODA Service but used internally only by the Bee Framework , ensuring that messages are received and sent inside the framework domain. It is a YODA complementary service which main benefit is avoiding a network overload with unnecessary YODA messages.

The requirements for this service are:

1. subscribe / unsubscribe messages;
2. send and receive messages;
3. broadcast messages inside the framework domain;
4. convert framework messages into YODA messages.

4.4.6 Timer Service

A timer is like an alarm clock used to measure time in the application, which can be useful to perform some actions based on elapsed time, such as regular and periodic operations or simple single time operations.

Such a service can be very useful to generate alarm events to notify the main application. These alarms work as internal "reminders" which allow the application to be notified at a desired date and time in order to perform the processing as needed. If largely used by an application, timers can also be used as an agenda or even schedule, since a set of different timers can be configured and installed to trigger events that occur at different moments.

The requirements identified for the timer service are:

1. trigger an event just only once;
2. trigger periodic events given a time interval value;
3. start triggering events immediately or just after a specified given start date time value;
4. trigger events indefinitely or stop triggering events at a given date time value;
5. allow a timer to be restarted at any time, keeping the same time interval but changing the date time of the following events triggered by reseting the start time;
6. stop a timer at any moment.

4.5 Summary

This chapter described the proposed architecture of the framework. It considered a black-box overview of the framework as well as a white-box description in order to provide readers a better understanding of the framework architecture. It described the framework in terms of its main components, namely the modules, the message handling and the services.

A global overview in terms of classes used has been presented using a class diagram. Then, this class diagram has been consecutively decomposed in order to better explain and detail the classes related to specific problems and how their associations helped solving those problems with a concrete design architecture.

The proposed architecture to solve those problems most of times used design patterns in order to achieve a good architecture design. This way, when presenting each problem, the design pattern used to solve it has also been described. Additionally, the existing parallelism between the classical architecture of each design pattern considered and the way it has been implemented in the proposed architecture is also illustrated. The main advantages and consequences of using each design pattern to solve a specific problem have also been referred in terms of how the usage of such design patterns promoted a better design architecture for the Bee Framework.

The design patterns used in the framework architecture are:

- Singleton — ensures a single instance of the framework;
- Factory Method — provides an abstraction to consider all framework modules the same way;
- Observer — observes and listens directories to detect changes;
- Chain of Responsibility — ensures the message handling between modules;
- Strategy — allows different strategies to perform data collection;
- Template Method — defines the skeleton of the algorithm used by a strategy to collect data.

The framework modules described are:

- *FolderMonitorModule* — contains the methods and operations to monitor directories and their content files;
- *BackupModule* — contains the methods and operations to backup files and check for updates;

- *BeeEquipmentModule* — contains not only the methods and operations that are common to other framework modules but also the commons methods that are specifically related to equipment modules.

The services considered are presented in the following list. Each service has been described and detailed considering their requirements.

- YODA Service — send or receive YODA messages;
- Database Service — execute queries, stored procedures or transactions in a database;
- Email Service — send emails;
- Logging Service — perform logging tasks;
- Message Service — handle internal framework messages;
- Timer Service — install timers and generate alarm notifications.

Chapter 5

Prototype Development

This chapter presents the prototype development and some of the technical decisions taken at implementation level. It complements Chapter 4 by exposing the framework architecture and its services in practice. The chapter focus on the overall framework development and its configuration settings (consult Appendix A for details related to framework configurations). Additionally, this chapter also focus on the data collection process related to a real manufacturing equipment recently introduced in the assembly line.

5.1 Prototype Goals

The architecture of the framework and its services referred in Chapter 4 intend to help the process of collecting the generated data by manufacturing equipments. However, in order to prove the real utility and adequacy of the proposed architecture and services solution, a prototype has been developed.

The development of such a prototype works as proof of concept and intends to validate the architecture in a level that goes beyond than just the theoric demonstration. In order to present the framework utility regarding data collection in manufacturing equipments context, a real equipment used in the assembly line has been chosen. This equipment name is AOI, standing for Automatic Optical Inspection. Consequently, the main goals of the prototype are not only the validation of the theoric concepts previously referred related to the architecture and services but also to provide a new integration solution for this type of equipment.

The equipment should not be changed due to this new solution because the new strategy intends to act like a black-box. Both the format of the generated data that needs to be collected and the final target used to save this data remains exactly

the same; the only thing that needs to change is the internal process related to the data collection strategy used. This way, the final results of the data collection itself should be the same as the ones achieved with the previous approach, but the new strategy should also include some performance improvements.

5.2 AOI — Automatic Optical Inspection — Equipment Overview

AOI is one of the equipments of the SMT line. It is the equipment that inspects the modules[1] after they have been completely assembled. Figure 5.1 shows this equipment.

Figure 5.1: AOI equipment

The AOI equipment provides an automated optical inspection system which combines an advanced detection with faster speed and higher resolution. It features an unique technology that yields robust defect detection, high measurement accuracy, and inherently low false-call rates. This equipment can meet all of the PCB assembly inspection needs from defect screening to process monitoring. No matter how the system is employed, this equipment will help increase yields, reduce scrap, and decrease field returns [73].

[1] in the semiconductor industry, a module is an electronic package containing multiple integrated circuits but used as a single one

The AOI equipment employs CyberOptics'[2] proprietary Statistical Appearance Modeling (SAM) vision technology to provide the most comprehensive defect detection method in the industry. By simply showing SAM examples of acceptable components, solder joints, or other PCB feature, the system works out for itself how to distinguish the good from the bad. In addition, SAM learns process and feature variations by simply adding images to the model producing the lowest false-call rate of any AOI machine [74].

Programming the inspection tasks performed by the equipment is quite simple. Using the equipment software, the inspection locations are defined by drawing a box around the component, joint, or other feature. There are no parameters to adjust and no algorithms to select. This simple programming combined with a simple mechanical design yields the simplest inspection machine. In addition, the equipment software also contains defect review software which helps categorize real defects for rework for scrap.

This equipment is used in Qimonda assembly line to automatically inspect the modules. Operators define which lot of modules will be inspected by the equipment and configure it to inspect the modules of the defined lot. The equipment will then inspect each PCB module one by one and, for each one, it generates a XML containing the results of the measurements retrieved by the optical inspection. Each of these modules contain several board components and each of this boards contain a very large number of locations. These locations correspond to the inspected places and the inspection results are related to the measurements retrieved with the optical inspection.

These measurements correspond to the data that needs to be collected. By collecting this data it will be possible to know how many locations have defects, how many boards of the module have passed and how many have failed. With these measurements, it is also possible to decide which defects can be repaired if a rework is done or if the board is definitely lost and could not be recovered.

There is currently one equipment integration solution available for these equipments. This solution has been developed by an equipment control team by adapting some previous similar approaches. However, a new integration solution using the Bee Framework has been developed and is now available, so it will be possible to establish a comparison between both approaches. The integration details of the AOI integration using the Bee Framework will be explained along this chapter.

5.3 Collecting Data From AOI Equipments

The three main actions to achieve in order to integrate this equipment are:

[2] http://www.cyberoptics.com

1. collect the XML result files using the *FolderMonitorModule*;

2. parse the result files to extract the required data information;

3. consolidate the complete results (raw data plus summaries) into an AOI specific database schema.

The AOI equipments are responsible for XML files creation for each one of the lot[3] runs, containing the necessary data to integrate in target system. The collection of XML files is done by the *FolderMonitorModule*. When a file is created this module triggers an event and sends a message to the framework message center. This message contains some information about the event triggered, such as the name and the full path of the file. The message center broadcasts the received message to other available modules and the *AOIEquipmentModule* is the module that receives and handles the message. Then, the *AOIEquipmentModule* collects the XML files, parses and processes them and insert all raw data into the database. Summary information will be created also in the same database, based on imported raw data.

AOIEquipmentModule interfaces with *Configuration Manager* (CFGmgr), through YODA middleware, in order to get specific application configuration parameters. Finally, the Remote GUI is available to operators to interact with *AOIEquipment Module* and to get notification messages from it. This module also interacts with the *Lot Equipment Parameters History Server*, so that equipment break downs can be effectively logged.

Desirably, only one instance of the Bee Framework should be used for each one of the AOI equipments. However, the same instance of the Bee Framework can be used to multiple equipment modules, by creating an *AOIEquipmentModule* for each equipment. Additionally, all AOI equipment modules running in the same Bee Framework instance must use Folder Watchers identified by unique names. In this context, a *FolderWatcher* works like a folder "observer" or "spy" that sends a notification each time a change is detected in the folder that is being "observed". Using unique names ensures that the each XML file detected is only processed by the correct *AOIEquipmentModule* and that this module also refers to the correct AOI equipment.

Figure 5.2 explains how the AOI integration should be done.

[3]In this context, lot is a set of semiconductor modules acting as a sole unit. The modules contained in each lot are inspected by the AOI equipment

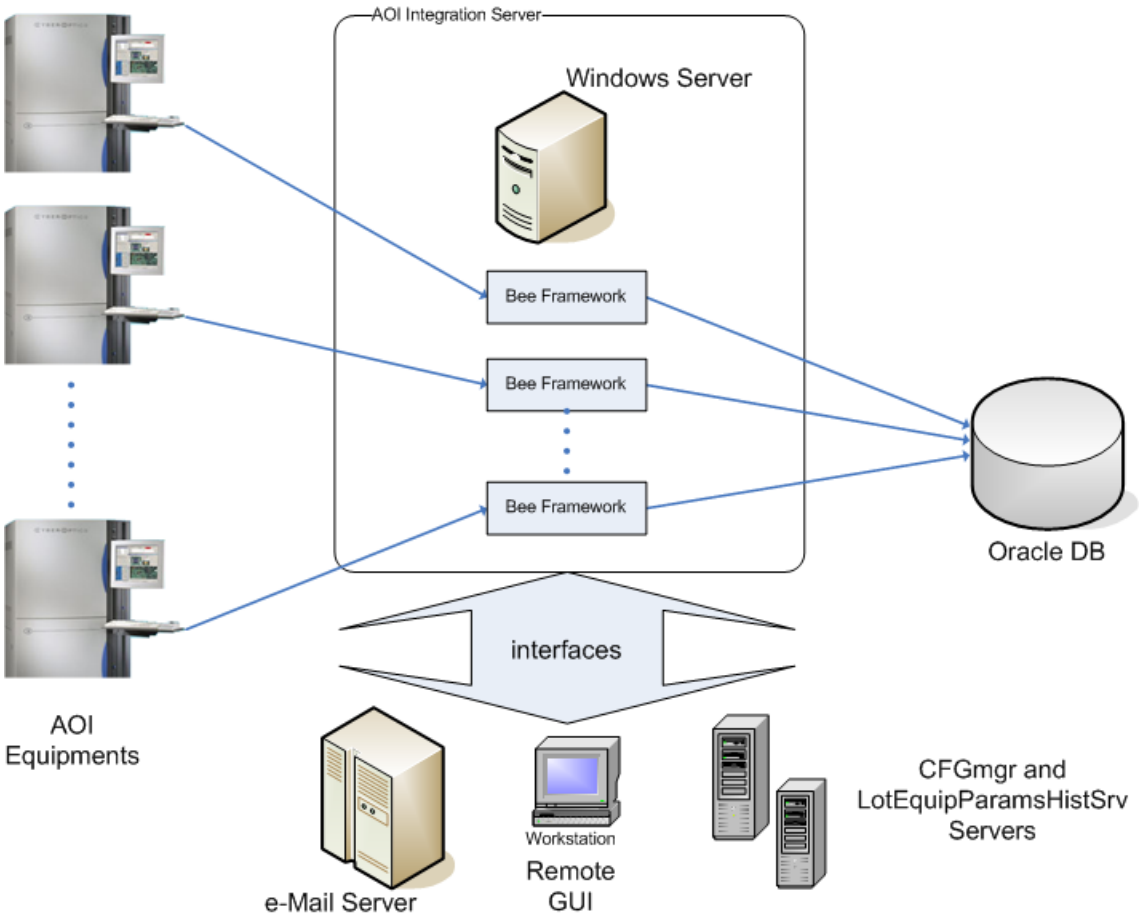


Figure 5.2: AOI integration overview

5.4 AOI Integration Use Cases

This section presents the use cases that specify the required functionalities of a system, showing the collaboration among the actors. The first use case diagram is mainly focused in the *FolderMonitorModule* and the second one in the *AOIEquipmentModule*.

FolderMonitorModule only contains *FolderWatchers* that are responsible for monitoring folders and send notifications each time a change is detected. *AOIEquipmentModule* is the core module of this integration, since it defines an integration strategy mostly related to the collection, parsing and processing of XML files. Figures 5.3 and 5.4 show the use case diagrams for both modules, respectively.

The use cases diagram presented in figure 5.4 illustrates the use cases directly related to the AOI integration itself, since they are specific for this equipment type. Collecting the data this equipment type generates cannot just take in consideration the collection of the data existing in the XML files. There are some complementary

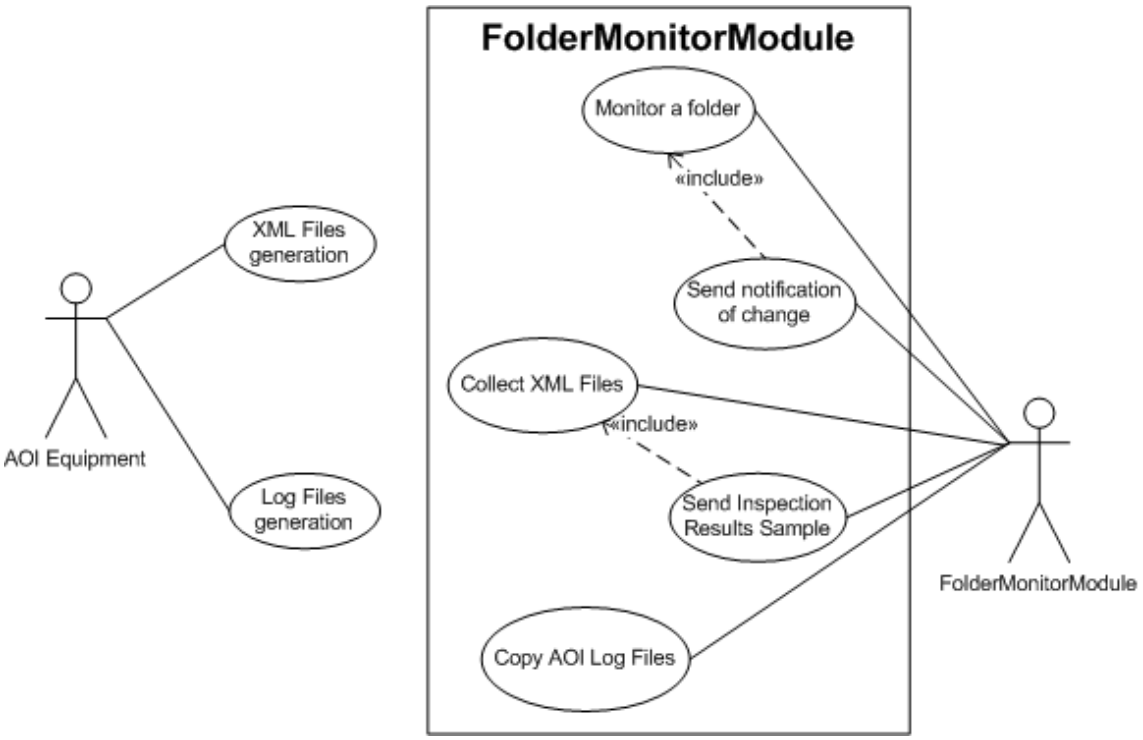


Figure 5.3: *FolderMonitorModule* use cases

functionalities not directly related to the data collection tasks themselves but that are also very important in the whole process of collecting data, namely:

1. starting or ending a lot inspection;
2. sending notifications if the equipment is not generating XML files;
3. backing up the equipment log files;
4. logging equipment breakdown reason.

5.4.1 Complementary Functions

The following subsections describe the complementary functions enumerated in the previous list.

5.4.1.1 Lot Start and Lot End Commands

Lot start and lot end commands are basically two instructions that equipment operators manually execute to tell the AOI equipment that a new lot will start to be inspected or that the defined lot inspection has finished, respectively. The instructions to execute these two commands are done via the equipment Remote GUI.

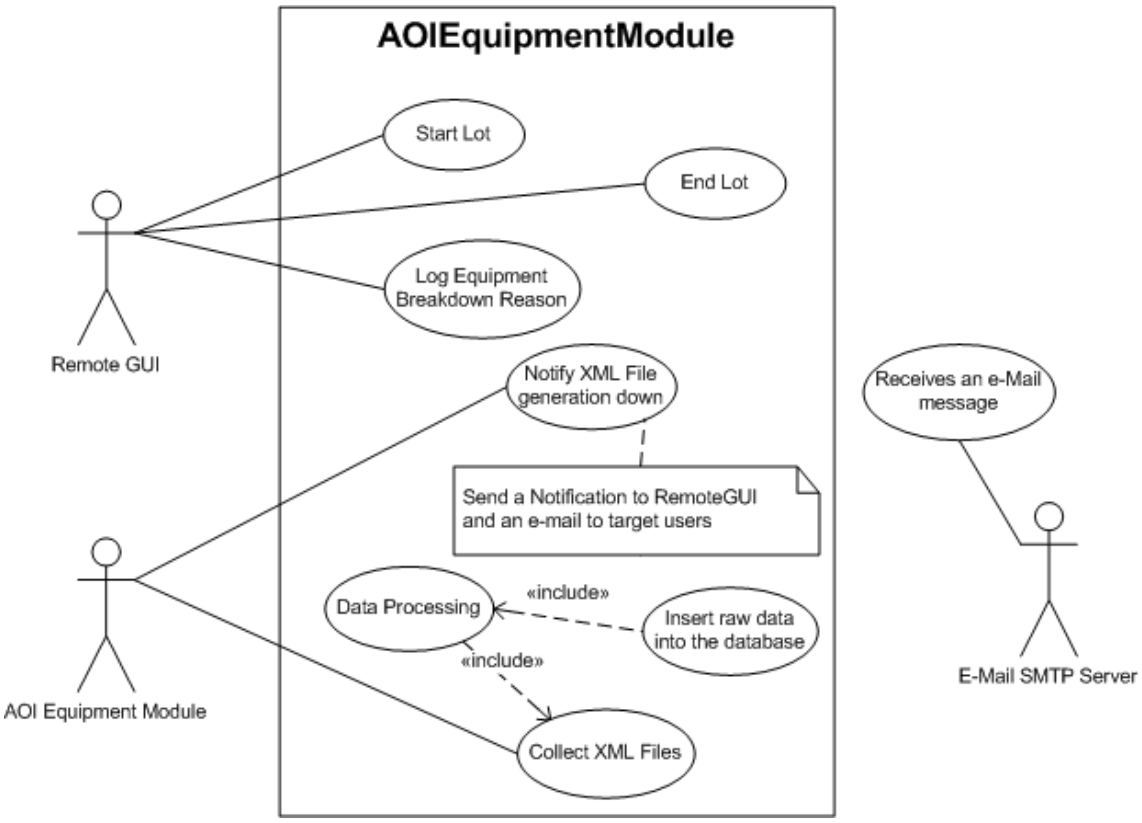


Figure 5.4: *AOIEquipmentModule* use cases

Operators use this remote interface to interact with the equipment and, each time an operator executes one of these command instructions, a message is sent using the YODA network in order to communicate the operation action to the application performing the data collection.

Since the Bee Framework has a service to support the handling of YODA messages, the framework is able to catch these messages and consequently become aware of the instructions executed by the operator in the Remote GUI interface. This way, the framework can then trigger both the lot start and lot end events and update the settings about the current lot accordingly.

The YODA message used to send both instructions has the same message subject which stands for *CommandLot*. This YODA message encapsulates the command lot type by using its message parameters and its message subject is subscribed by the *AOIEquipmentModule*, which means the module would be able to get the message as expected. When receiving a message with such subject, the module checks the parameters in order to retrieve the associated command type encapsulated in the message parameters. Depending on the command lot type received, the *AOIEquipmentModule* either configures and updates the settings and rules for a

new lot (start lot command) or ends the lot processing and resets the settings and rules for the current lot previously defined (end lot command).

Even if this complementary functionality is directly related to data collection itself, its importance is high. The commands referred are the ones that allow the *AOIEquipmentModule* to define the required lot information that would be used to validate each XML file and decide whether a file should or not be ignored.

5.4.1.2 Notification of XML File Generation Down

Since data collection from this equipment depends exclusively on the generated XML files that contain the results of optical inspections, informing operators when the equipment is not generating files is critical. If such a situation occurs, it means that the equipment is not working as expected and, consequently, there is no available data to collect from XML files. This way, knowing when a generation down occurs is of high importance so that operators can check why the equipment is not working correctly.

The Bee Framework, and more concretely the *AOIEquipmentModule*, is able to detect when XML result files are not being generated by the equipments and notify operators of this situation so that they can check what is cause of the generation down and correct the problem. The notification when such situation occurs is sent both via email to the Process engineers and also via the YODA network so that operators can receive the notification in a message box on the equipment Remote GUI.

The method used to determine when then generation of files is not happening is based on the amount of time elapsed between the generation of two consecutive result XML files. If this amount of time is higher than the existing value in equipment configuration settings, the notification should be sent.

The detection of XML file generation down is made by using the framework Timer Service. Once installed, the timer used to notify XML file generation down will send a notification if the elapsed time event is triggered. This event is triggered if the time elapsed since last XML file detection is larger than the configured time of that framework Timer Service.

Obviously, these notifications only happen when a lot is defined in the *AOIEquipmentModule* (defined using the start lot command). Otherwise, if no lot is currently defined, it means that the equipment is in stand by and that it is not performing any optical inspection. In such situation, the equipment is not generating XML files and the timer must be paused because notifications do not make sense. However, if a lot is defined, the timer counter must be restarted each time a file is parsed.

5.4.1.3 Backup of Equipment Log Files

During the data collection process, the *AOIEquipmentModule* controlling an AOI equipment logs the errors detected, as well as warnings and application messages. These log files should be collected from the equipment file system and placed on the server to keep them available on-line.

However, these files should not be available indefinitely because this would lead to a large number of files on servers, most of them never used and just kept for backup purposes. This way, it is important to have backups of these files but since only the most recent information is required, they should be deleted when their last modification date reaches a configurable age or must be reused if the file size grows and reaches a maximum file size.

5.4.1.4 Log Equipment Breakdown Reason

When an equipment breakdown is detected, it is important to know the cause beyond the malfunction of the equipment. Whenever a breakdown occurs, operators should register the reason of the breakdown. Some of these reasons are related to maintenance operations, optical inspection defects or equipment fails, for example.

Whenever an equipment breakdown occurs operators should log the reason of the breakdown by interacting with the Remote GUI. Using this interface, operators can select the equipment in which the breakdown is related to, select the reason from a pre-determined list of reasons and also insert an optional comment.

A message containing the breakdown information is then sent using the YODA network and it will be caught by a service listening the subject of that message. This service name stands for *LotEquipParamsHistSrv* and the subject it needs to subscribe in order to receive the messages is gathered from the INI Table configuration settings through the CFGmgr. This way, when an equipment breakdown message becomes available in the network, the *LotEquipParamsHistSrv* service will get the breakdown information and make it available for query in OEDV.

5.5 AOI Use Cases Implementation

This section describes how a XML file is processed in terms collecting its data and also how the complementary functions referred in the Complementary Functions section (see section 5.4.1) have been implemented. The implementation of the use cases regarding the AOI integration is described in the following subsections. The description of each use case implementation will also be complemented by sequence diagrams.

5.5.1 Process XML Files

Whenever the *XMLFolderWatcher* detects that a file has been created in the folder being monitored, it signals the *FolderMonitorModule* by delegating an *IOEvent*. When this modules receives the delegated event, it sends an internal framework message (*BeeMessage*) to the framework *MessageCenter*, which will then broadcast this message to all available modules. Since the message content was about a new eligible XML file in the listening folder, this message request will then be handled by the *AOIEquipmentModule*.

Next, when the *AOIEquipmentModule* receives the message, it loads the XML file by reading its contents, parses the required file to retrieve the information related to the lot. This information is critical because they allow the module to perform some comparisons between the existing information for the current lot defined in the module and the lot information contained in the XML file. This comparison is helpful to validate the file and decide whether the rest of the file should be parsed or immediately rejected.

If the validation of the file is successful, the *AOIEquipmentModule* uses its configured strategy to parse the file, retrieve the data related to the panel and boards information. The relevant data related to each measurement is called raw data and it is finally inserted into the target database. However, the data related to each single measurement do not provide much information itself. This way, all the measurements are considered so that this data can be summarized in total values, such as how many failures have been detected or how many boards passed, for example.

Depending on the result of the file parsing operation and after inserting the data into the target database, each XML file will then be moved either to the processed, ignore or error folder. To move the file, an internal framework message containing both the source and target path is created and sent to the framework *MessageCenter*. This message is then broadcasted to all available modules and is handled by the *FolderMonitorModule*, which will move the file to the desired target destination.

Figure 5.5 illustrates these sequence of actions related to the processing of a XML file.

5.5.2 Notify XML Generation Down

The *AOIEquipmentModule* has a Timer Service configured when the module starts. The timer uses a duration that is a predefined value loaded from configurations. Whenever the timer reaches the configured amount of time since the counting has started, it means that no new XML file has been processed during a batch lot

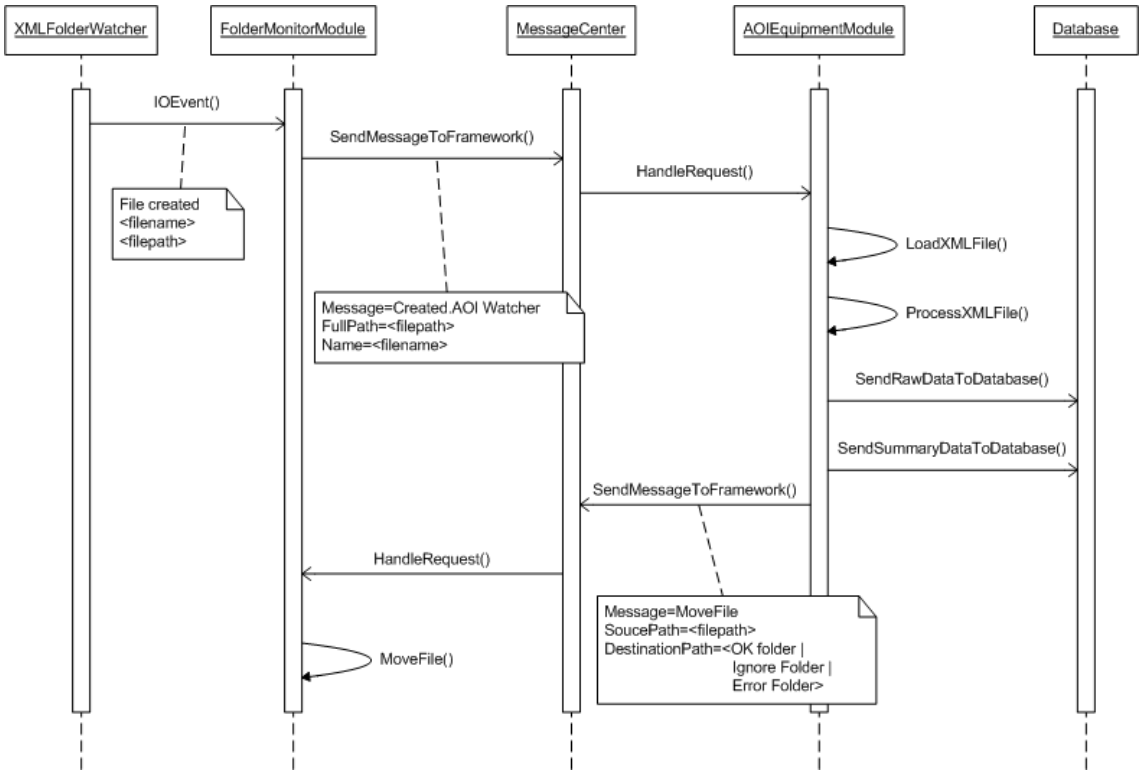


Figure 5.5: Use case: Process XML files

processing. If such a situation occurs, the elapsed time event is triggered, so that a notification can be sent.

This event will be caught by *AOIEquipmentModule* and a notification of the file generation down will then be sent to the Remote GUI. Additionally, if the defined settings also include the required email parameters for sending email messages, then an email is also sent to the configured target users.

The Timer Service plays a critical role to monitor the time elapsed since the last XML file generation. The timer is initiated when the *AOIEquipmentModule* receives the lot start command and is stopped when the module receives the lot end command. Additionally, when a XML file becomes eligible in the listening directory it means that a new file has been generated and the timer is reset, so that the counting of the elapsed time can start again.

Figure 5.6 illustrates these sequence of actions related to the sending of notifications in case of XML generation down.

5.5.3 Backup AOI Log Files

There are two available options to backup or archive the log files each equipment generates. If both XML and log files are created in the same folder, only one *Fold-*

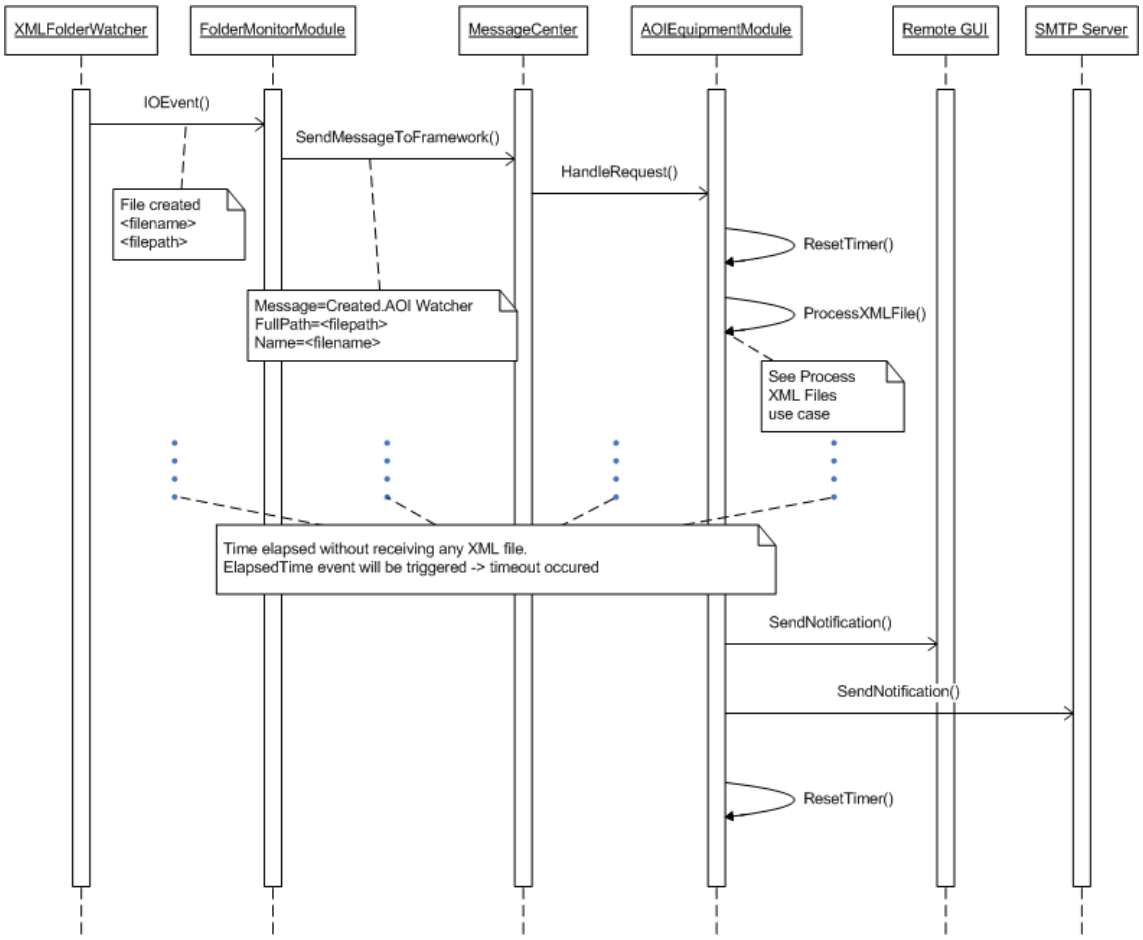


Figure 5.6: Use case: Notify XML generation down

erWatcher is needed but the *AOIEquipmentModule* must firstly check the filename to determine which kind of file has been created and then process it accordingly. If XML and log files aren't created in the same folder then two *FolderWatchers* are needed. In such case, when a file is detected the *AOIEquipmentModule* immediately knows which kind of file has been detected. Both options are easy to implement, but the second one implies a second thread running to monitor different folders, reducing performance.

The best approach is a third option which considers the best of both previous options. XML and log files are created in different folders, but both folders have the same parent folder. Using this approach, only one *FolderWatcher* instance is needed, which improves performance and provides a better understanding of the configurations used. If using this alternative option, the *FolderWatcher* instance must be configured in order to consider the monitoring in subdirectories.

Whenever the *BackupFolderWatcher* detects that a file has been created in the folder being monitored, it signals the *FolderMonitorModule* by delegating an *IO-*

Event. This module will then send an internal framework message to the framework *MessageCenter*, which will broadcast this message to all available modules. This request will then be handled by the *AOIEquipmentModule*. This situation is similar to the one previously related in the Process XML file section (see section 5.5.1).

The *AOIEquipmentModule* creates a new message containing both the source path of the log file detected and the target destination path where the file should be backed up. This new message is then sent to the *MessageCenter*. Once again, this message will be broadcasted to all modules and, in this specific situation, it will be handled by the *FolderMonitorModule*, which will move (backup) the file to the desired target destination.

Figure 5.7 illustrates these sequence of actions related to the backup of log files.

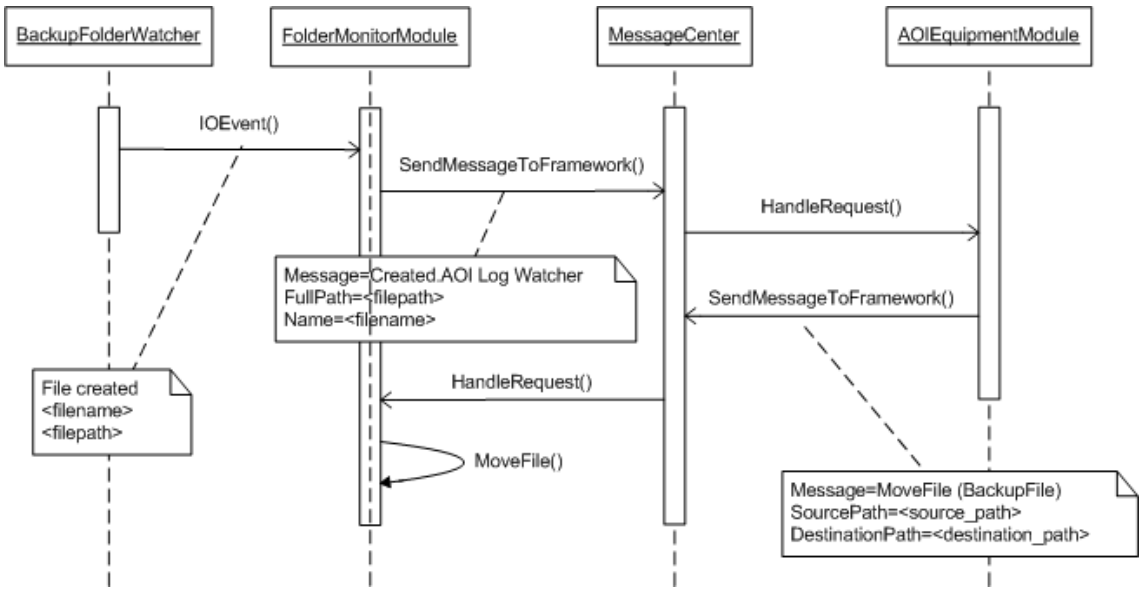


Figure 5.7: Use case: Backup AOI log files

However, this use case and the corresponding UML sequence diagram shown in figure 5.7 are applicable only if doing an extra backup of the log files is needed. The Logging Service already provides the necessary tools to control the size and age of the log files, the maximum number of log files allowed and even backups the files in a specified folder directory. This way, the use case presented can be avoided if the Logging Service configuration is done properly, specifying the backup directory and the age / size conditions desired for log files.

5.5.4 Log Equipment Breakdown Reason

Whenever an AOI equipment breaks down, the user currently operating the equipment can log a message using the Remote GUI interface and send the breakdown

message via the YODA network. The subject of this message is already subscribed by the framework *MessageCenter*, so it will be able to catch the message and broadcast it to the available modules.

The *AOIEquipmentModule* handles the message and uses the information contained in its message parameters. This breakdown information is then retrieved from the parameters and will then be sent to the *LotEquipParamsHistSrv* server. Once again, in order to send this message to the server, a YODA message is used and when the server receives such message it can effectively log the equipment breakdown reason. The event type sent is "LogBreakdown" service.

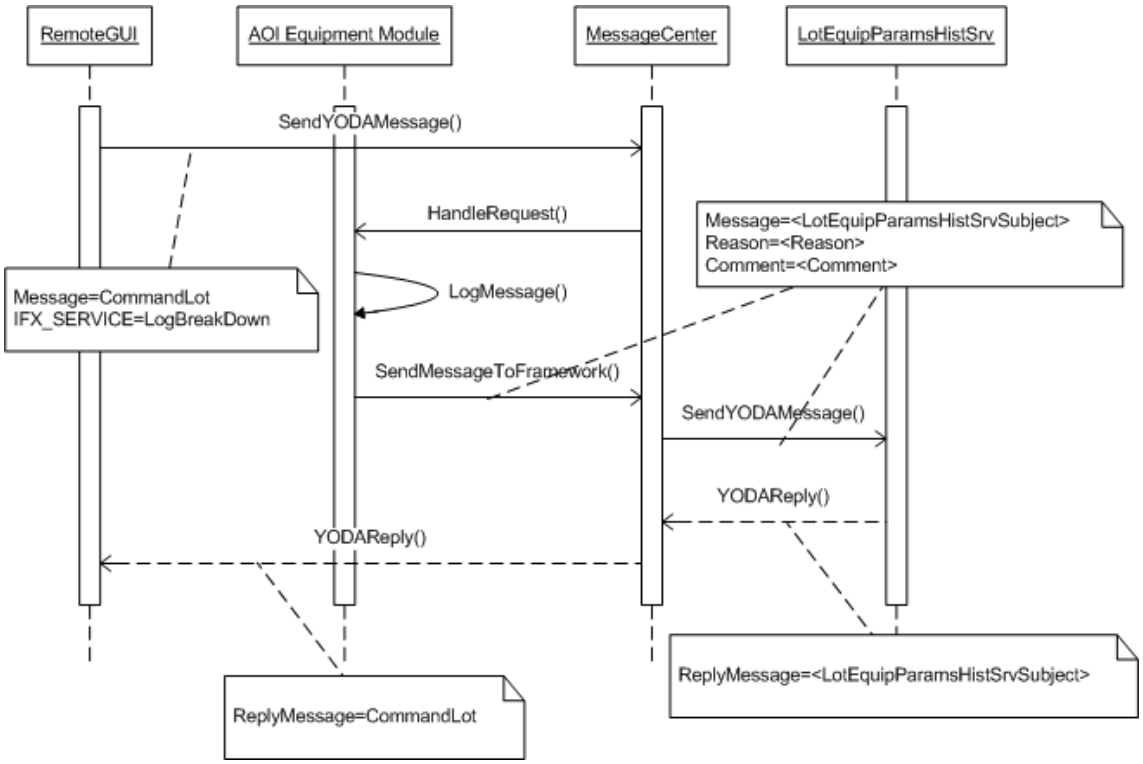


Figure 5.8: Use case: Log equipment breakdown reason

5.6 AOI Integration Architecture

This section describes the integration architecture used to integrate the AOI equipment. It starts by describing the main flow required to collect and save the data the AOI equipment generates. This flow is described in the following paragraphs of this section and, in order to better understand this flow, figure 5.9 should be analyzed because it illustrates the sequence of actions textually described. Additionally, this section also details both the global and Bee Framework logical views used to integrate the AOI equipment.

After loading the required framework and equipment configuration settings, the data collection process is ready to start. A Timer Service must then be started so that the use case related to notifications about XML file generation down can be possible. This timer will trigger the event that occurs when the time elapsed without any XML file being generated is reached. A notification is sent whenever this situation occurs, signaling the XML file generation down.

About the data collection itself and the gathering of the generated XML files, the *FolderWatcher* used to monitor the listening directory should also be started by the *FolderMonitorModule*, so that new XML files can be detected and their contents retrieved.

However, there is an important characteristic that should be referred and explained at this point. Both the file creation detection and the time elapsed are triggered using events. This means that after installing and starting the Timer Service and the launching the *FolderWatchers* of the *FolderMonitorModule*, both will stay in an idle state. They will only be enabled again when the action they are listening occurs. If an event related to these actions is triggered, it means that either the time was elapsed or a file was created, respectively. This avoids the periodic checking ("ping") approach to decide whether an action should or not be performed, reducing the overhead of possible unnecessary checking.

Whenever a XML file is detected in the folder being monitored, some initial validations regarding the lot information contained in the file data and what the expected results were are performed. If this validations fails, the file will not be considered and will just be moved to an ignore folder, previously defined in configuration settings. On success, the Timer Service must be restarted so that the counting time without any file generation can also be restarted.

The file is then parsed to extract the panel, boards and locations measurements data. This information represents the raw data that needs to be saved into the target database. There are two possible approaches available to load this data into the database: using normal SQL queries and stored procedures or using SQL*Loader tool (consult Appendix D and Appendix E for further details related to AOI database schema and SQL*Loader usage, respectively).

In the first approach, a transaction must be opened and, as soon as the required fields of a database record are complete when parsing the XML file, the record is immediately inserted in the correct database table. These approach considers that each database insert should be done using the open transaction and immediately when the data becomes available during the parsing. The transaction is committed at the end of the file parsing if no errors are detected during this phase; otherwise, the transaction must be rolled back, the file is moved to the error folder defined in configuration settings and a notification is sent both to the Remote GUI and also

via email.

The second approach uses SQL*Loader tool to send collected data from XML file to the database. Instead of inserting each record one by one in the database, the records are stored in a temporarily file. At the end of the parsing, that file will be fully loaded to the database at once. This approach is specially suitable for huge volumes of data because its performance increases by decreasing the amount of time required to parse and save the data of the XML file.

After saving the file contents into the database (either using queries or the SQL*Loader tool), the measurements related to the XML file just parsed are processed in order to calculate the total values related to the measurement results for the lot and panel defined in the file. Finally, if all steps are successful until this point, the file should be moved to the folder containing the XML files processed without any errors.

Figure 5.9 represents the main flow used on AOI equipment integration and the process used to collect the data this equipment type generates.

5.6.1 Global Logical View

This section considers the high-level logical view structure of the process related to data collection from the AOI equipment. It also refers the positioning and the importance of each component in the overall system, the existing relationships existing with external services, applications and databases.

At the beginning of the chain process, there are the AOI equipments that generate the inspection results and save them into XML files. Additionally, each equipment is also responsible for generating its own log files containing the errors, warnings and application messages logged during the process. These generated log files and their content information should also be collected. On the server's side, one instance of the developed prototype using the Bee Framework is running for each different AOI equipment. Each instance of the prototype is the "brain" of all the data collection process and fully controls how the main flow steps are achieved.

The running prototype instance retrieves some of its configurations from the IniTable Admin GUI and also from its own local configurations available through XML files. Local configurations are available through a single file that contains the general framework settings and also through multiple XML files, one for each of the modules required. Configuration settings are then retrieved from local XML configuration files in order to load both the framework and its modules settings. These configuration files are loaded and parsed, so that configuration settings can be retrieved. The additional required settings must be retrieved from the INI Table and are gathered using a YODA interface. INI Table configurations are accessed via

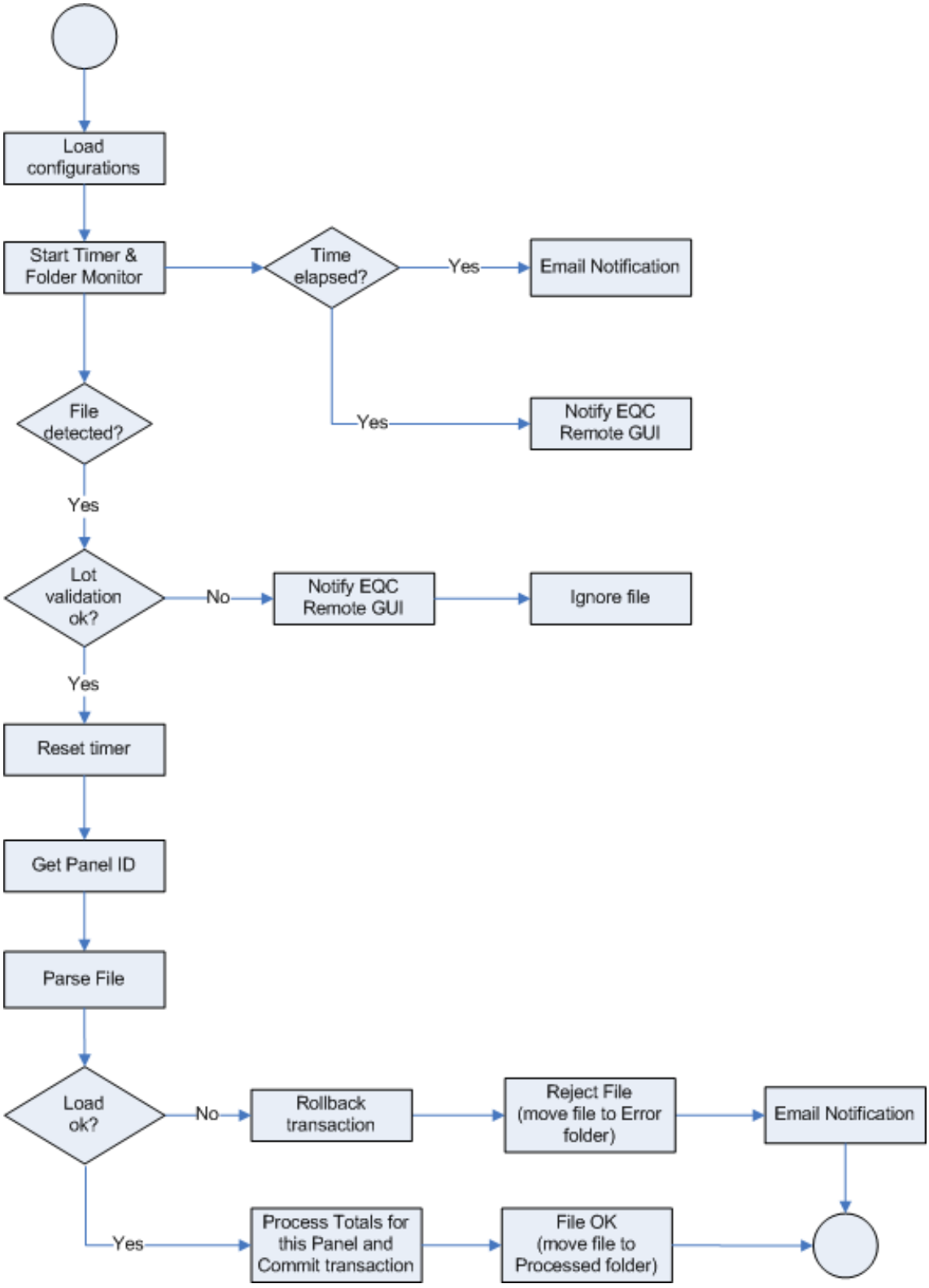


Figure 5.9: AOI main flow

YODA connection and are retrieved using the Configuration Manager (CFGmgr), which will receive these requests and reply with the necessary configuration settings.

Generated files are detected by the Bee Framework and their data contents are collected. This retrieved raw data is then saved , by inserting it into the target Oracle database defined through a TNS connection.

The Bee Framework also serves as a relay to interface with Remote GUI and *LotEquipParamsHistSrv* on equipment breakdown reason notifications. It also notifies the Remote GUI about XML file generation down timeouts and sends notification messages to desired target e-mails previously configured in the settings.

Figure 5.10 represents the overall architecture used in the AOI integration and regarding the collection of the data generated by this equipment type.

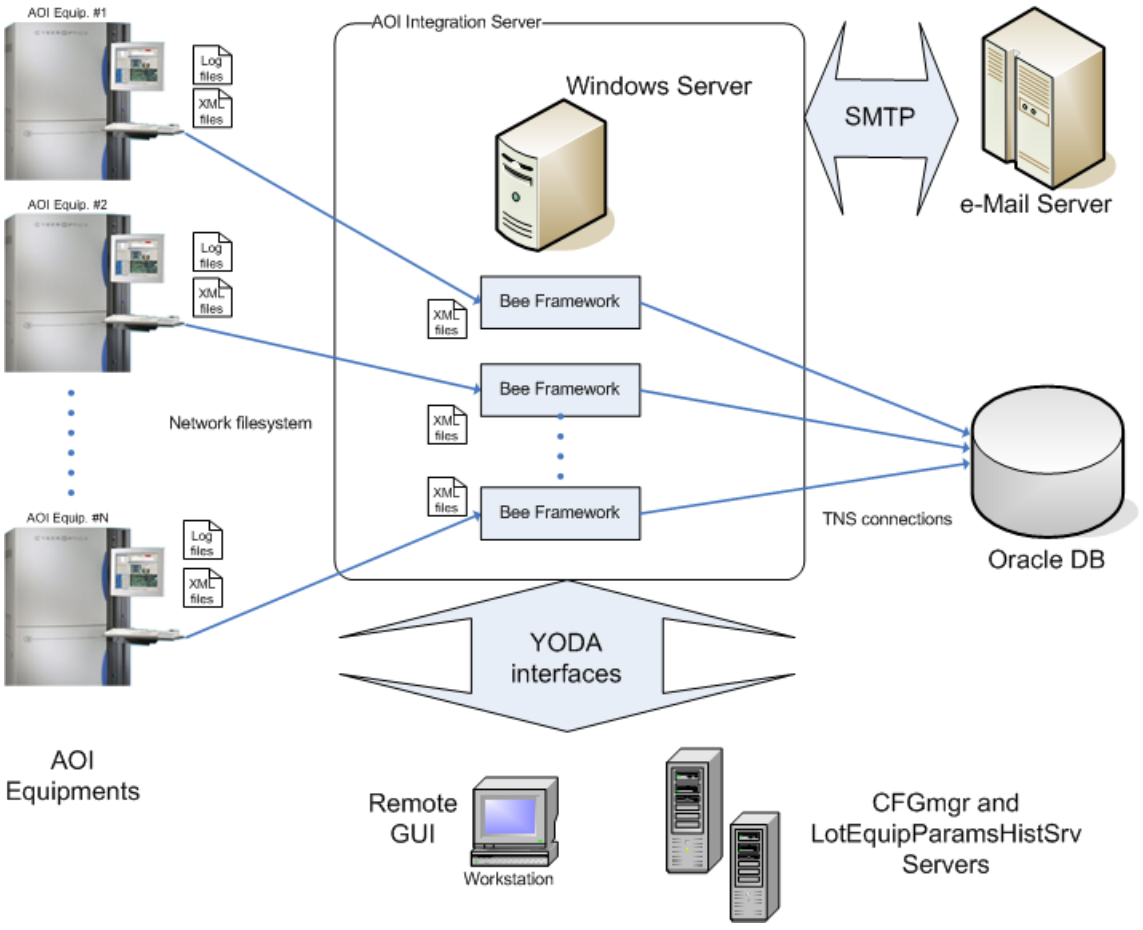


Figure 5.10: AOI integration entities

5.6.2 Bee Framework Logical View

This subsection describes the Bee Framework logical view used in the AOI equipment integration and complements the text description of the previous section (section 5.6.1). Appendix F contains some useful information about AOI configurations used.

Two framework modules are required: the *FolderMonitorModule*, which interacts directly with the equipment file system, and the *AOIEquipmentModule*, which contains the implementation and definition of the main flow represented in the state model of figure 5.9.

The *FolderMonitorModule* is responsible for detecting the creation of new XML files generated by the AOI equipment in the listening folder and send a message notification to the main application class, the *BeeFramework*. Then, the *AOIEquipmentModule* is notified about these new eligible XML files and loads their contents so that they can be processed. This means that the *AOIEquipmentModule* already knows which files are available for parsing.

The file starts being parsed and some data content validations are performed. Finally, if these validations are successful, the retrieved raw data is inserted into the target database. Additionally, this module is also responsible to process the measurement results retrieved from the file during the parsing step in order to calculate the total values related to the measurements and send the final summary data to the same target database.

The *AOIEquipmentModule* also has its own folder structure to save the detected files depending on the results of data validation and file parsing. These files can be either successfully processed or rejected or ignored, so that the module can decide which target destination is adequate for each XML file detected. Furthermore, this module has its own logging system which backups files automatically by checking the configurations of the Logging Service in use. However, log files can also be backup by detecting the creation of the log files in a listening directory (just like XML files) or by periodically checking the listening folder to retrieve the log files existing there.

These modules do not interact directly with each other and all interactions between them must be done by using internal framework messages and by sending them to the *MessageCenter*, which will then attend the received requests and broadcast them so that other modules can receive these messages.

The *MessageCenter* is also responsible for handling the interactions related to YODA Service, comprising both the subscription and the diffusion of YODA messages. The YODA messages each module is interested in receiving are subscribed in the *MessageCenter* and whenever a message with a subscribed subject is available in the network the *MessageCenter* notifies the adequate framework modules. If a

module needs to send a YODA message, the message will be sent by using the same *MessageCenter*. The message will then be dispatched and, depending on the message type, the *MessageCenter* may have to wait for a possible reply or for a timeout.

The *MessageCenter* and the interaction between different modules using it is important due to scalability issues: if every module would interact directly with all available others, it would be very difficult to maintain this interaction between all modules, especially if the number of modules increases significantly. Using the *MessageCenter* as intermediary, the only thing that needs to be done while adding a new module is defining it in the global configuration file used by the Bee Framework, specifying its type and the path for its own XML configuration file.

Finally, the most important part of the AOI equipment integration is related to the *AOIEquipmentModule* and with the Bee Framework services it uses. The following paragraphs will describe those services in a high level view, just to refer how and why each service is used in the AOI equipment integration.

Logging is an important but no critical service because it is used to record messages about application execution, such as error messages, warning messages or merely information messages. The logs can then be analyzed to better understand how application is behaving and identify the causes of possible errors.

The critical path of integrating an equipment is not only collect the data it generates but also save that data. The data generated by AOI equipments is collected and saved into a database, so the database service can be considered the most important service used to integrate such type of equipments. There are two possibilities concerning the usage of this service: the usage of normal queries and stored procedures inside an open transaction or the usage of an external application, SQL*Loader, to improve performance when the volumes of data are huge.

Timer is a service used to control the time related to the generation of XML files. It's a service used to monitor the interval time of consecutive generated files, so that problems can be quickly detected and notifications sent if the equipment is not working properly.

Email is a service that is just used in case of equipment failures. It provides general methods to send emails if notifications must be sent. Global email settings and target destinations are defined in the XML configuration file of *AOIEquipmentModule*.

Finally, the last service considered is a wrapper to YODA. All messages considered have the same format of internal framework messages, but with a flag to find out if the message is a YODA message or not. If a flag has a true value in a message, it means that the message considered is a YODA message. In such cases, when a

message is sent and handled by the *MessageCenter*, the flag will be checked, the message will be converted into a YODA message format and it will then be sent to the YODA network.

This YODA wrapper service also interfaces with two particular YODA services: *CFGmgr* and *LotEquipParamsHistSrv*. *CFGmgr* is used to acquire some configuration values about an equipment. These values directly influence the way the data is collected, specially about what concerns lot validations and the loading of data into the database. *LotEquipParamsHistSrv* is used to record equipment breakdown reasons.

Figure 5.11 illustrates both the existing interactions between the AOI equipments and the framework and between the framework and external applications. Additionally, it represents the internal interactions existing between the framework components, namely its modules, message center and services.

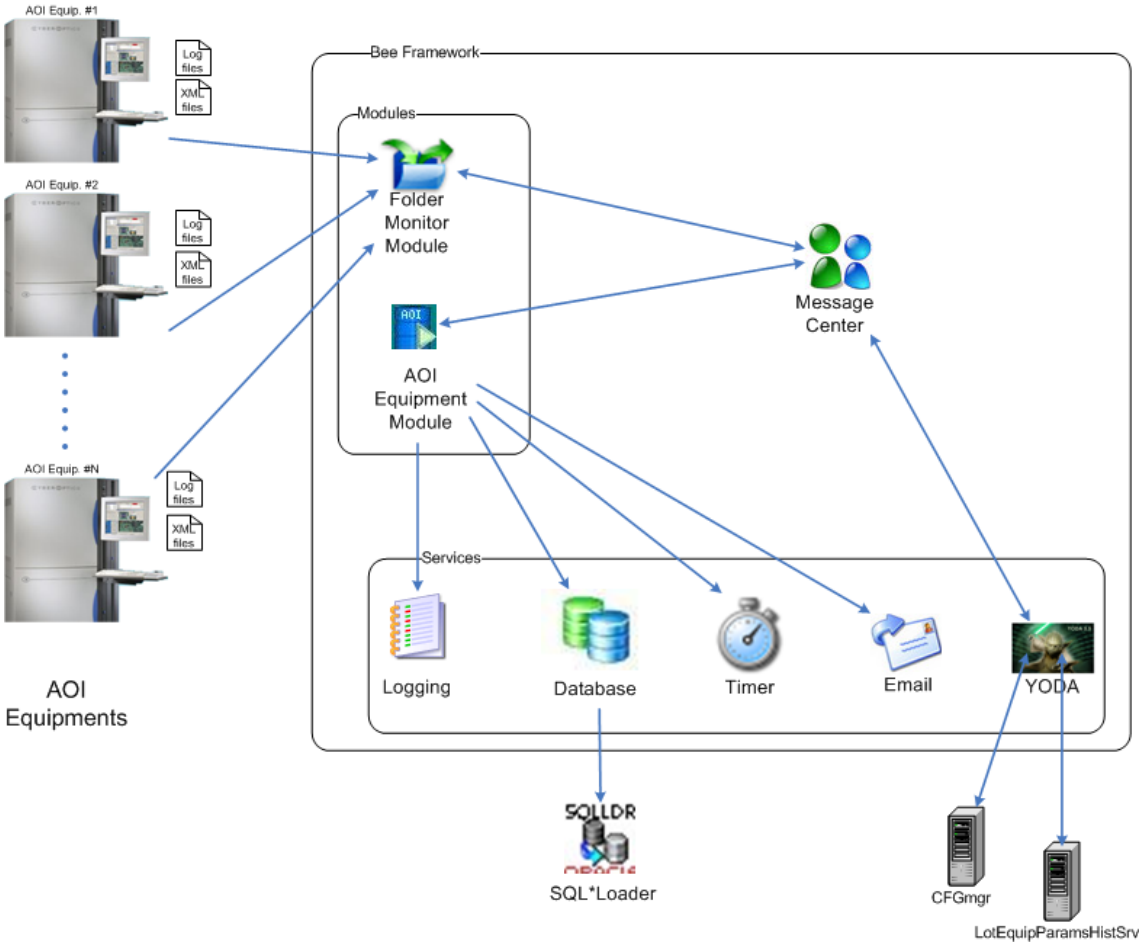


Figure 5.11: Bee Framework logical view

5.7 AOI Integration Test Cases

This section presents the integration test cases used to validate the implementation of the prototype. These test cases are related to the collection of data generated by AOI equipments and also with the complementary functions required to fully and effectively integrate this type of equipment.

For each integration test case, two tables are considered. The first table provides a description of the test in terms of identifier, test name, small description and creation date. Additionally, the first table also enumerates the pre-conditions required for running the test so that the expected results can be achieved. The second table of each test provides the required details in terms of the ordered sequence of steps required to run the test. Each step of the test case has a small description and has the expected results after the end of the step. Some additional notes are also presented for each step.

The list of integration test cases considered are:

- process XML files;
- notify XML file generation down;
- backup AOI log files;
- log equipment breakdown reason.

The following section shows an example of an integration test case related to the "Process XML files" use case. The remaining test cases are available in the appendix C.

5.7.1 Process XML Files

This test is the most important integration test case. It is directly related to data collection tasks: it considers the detection of new eligible XML files, performs the required validations, does the file parsing and inserts the collected data into the target database. Tables 5.1 and 5.2 illustrate the description of this test and the details related to the sequence of steps required to execute the test, respectively.

Table 5.1: Process XML files — Test case description

Test ID	Test T1
Test name	Process XML files
Test description	This test verifies if the *AOIEquipmentModule* inserts all necessary records into customized database (raw and summary data)

Continued on next page

Table 5.1 –continued from previous page

Test date	29^{th} May 2008
Pre-conditions	TNS connection properly configured and targeted to desired database. Prototype must be running and data collection from AOI equipment must be enabled. SQL*Loader definitions must exist in the INI Table. SQL*Loader header files must exist. All directories for processed files and SQL*Loader temp files must exist.
Expected results	Raw and Summary records exist in target database. XML files detected must be moved from the listening directory to the directory of processed files after the processing.

Table 5.2: Process XML files — Test case details

Sequence	**Step type**	**Step description**	**Expected results / state**	**Notes**
1	Action	Start the Bee Framework and begin data collection.	The application should have started.	See other preconditions on test case description.
2	Action	Trigger Start Lot event with Remote GUI.	Start Lot event triggered.	The *AOIEquipmentModule* must have received the Lot Start event.
3	Action	Ensure that AOI equipment XML files are created in the listening folder.	Equipment XML files are created in the input folder.	

Continued on next page

Table 5.2 – continued from previous page

Sequence	Step type	Step description	Expected results / state	Notes
4	Test	*AOIEquipmentModule* parses and processes XML files.	XML files are moved to processed directory.	A lot must be defined, otherwise the file will be ignored.
5	Test	Query the target database for processed lot number.	Raw and summary data exist in target database for that lot number.	
6	Test	Check if the XML files exist in the directory of processed files.	Files do not exist in the listening directory and exist in the directory used for already	

5.8 Summary

The present chapter gives an overview of the AOI equipment and relates it to the semiconductor industry. The chapter describes the prototype main goal, which is the integration of the AOI equipment in terms of data collection. By completing such integration, not only a new integration solution for these equipment becomes available but also the concepts related to the architecture proposed in the Chapter 4 are used in a real environment, going beyond the theoretic domain.

This chapter also explains the process of how data is generated by these equipments and how this data should be collected and saved: XML files are the result of optical inspections and are created in a listening directory; whenever a new file is

detected, the file is processed and if validations are successful the file is parsed and its data is saved into a target database.

The chapter also focused in the required use cases to integrate such equipment and perform data collection. At first instance, these use cases are explained and detailed conceptually. Then, the implementation of these use cases using the framework architecture and the services provided by it is described. These use cases are:

- process XML files;
- notify XML files generation down;
- backup of equipment log files;
- log equipment breakdown reason.

After the description of the implementation of the uses cases regarding the AOI equipment, the architecture used in the integration is described. Firstly, the main flow required to collect data from AOI equipments is detailed. Then, the global logical view related to the AOI integration and the external applications and components it uses is detailed. Furthermore, the Bee Framework logical view regarding how the framework modules interact and which services are used is also described.

Finally, this chapter presented the integration test cases related to the data collection from AOI equipments. These test cases are directly related to the use cases referred, provide a high-level description in terms of pre-conditions and expected results. Additionally, the sequence of steps to run each test is also referred.

Chapter 6

Findings and Discussion

This chapter presents the main findings related to the development of the framework and the integration of the Automatic Optical Inspection equipment. Additionally, a comparison between the proposed integration solution for the AOI equipment and the previous approach will also be done. This comparison is done along this chapter and mostly comprises the differences related to the AOI integration using both approaches, in terms of effort and time needed as well as performance evaluation.

6.1 Event-based Framework

The proposed architecture for the manufacturing equipment data collection framework is essentially based on events, delegates and handlers to catch these events. Combining object-oriented approach with the event-based paradigm brought several advantages in terms of performance questions.

Some of the advantages about this event-based framework are related to the detection of new files and with timer and message notifications.

6.1.1 Detecting Changes in Files

Without using events to monitor folder directories, the solution to control changes resides on periodic verifications. These verifications generally imply that a list of files and folders must be in memory, so that consecutive verifications can make a comparison between the existing list and the files and folders found. Moreover, these verifications not only need to compare the names of files and folders but also compare changes in modification times and permissions.

Additionally, these validations usually require a different thread to be running and, even worst, they require multiple threads to check changes in subdirectories.

This approach implies that developers must synchronize all the threads before returning the results containing the changes, which can be not only a painful task but also increases the probability of committing errors and leads to memory deadlocks.

Using an approach based on periodic verifications and comparisons is then more resource consuming in terms of memory and processor usage. The list of existing files inside a directory must always be kept updated and to achieve this performance degree it implies that the list must be updated after every single verification. Even if no changes have occurred between two consecutive validations, the list still need to be refreshed. In this case, the waste of resources is even higher because there was no need to perform a new verification, a new comparison and a new refresh of the list of files structure.

However, using an event-based framework and using events to monitor a directory and its subdirectories so that changes in files can be detected brings several performance improvements. Events provide a very clean method to detect changes that occur inside a directory without having to periodically perform verifications to check which files have been created, renamed, changed or deleted. Moreover, no list of files need to be maintained in memory, so there is also no need to do comparisons between two consecutive lists of files retrieved from verifications.

Using an event-based approach, the painful periodic verifications and comparisons are completely avoided. Using events the framework simply gets notified only and only when a change in the file is detected. Moreover, the type of change and the file where it has occurred are known *a priori* instead of having to match two possible long lists of files and match them to get the differences between them.

No additional verifications are required and there are also other positive aspects related to the detection of changes in files inside a monitored directory: the decision of including or not including subdirectories in the monitoring is easier because it only involves the configuration of a parameter flag. Being a configuration setting, it becomes quite simple to change this option without the need of developing new code and without changing any code at all. Filters are another advantage of using an event-based monitoring: when using a filter for monitoring a directory, only the filenames that match the regular expression of the filter cause an event to be fired. This is also a good improvement, since these validations are done without the explicit knowledge of developers.

6.1.2 Notifications

Events are also very useful when referring to timer and messages notification. Without using a timer based on events notifications it is very hard to monitor elapsed

time or to generate alarms at specific desired times. This implies the need of, just like the monitoring of directories, periodic verifications to compare the amount of time elapsed or to check if the time for firing an alarm has been reached. This approach is not only resource consuming but also can lead to time imprecisions. It would be very hard to exactly match the verification time with the alarm time, in order to get a result with just some milliseconds of delay or, even harder, with no delays.

Moreover, it would require developers to implement thread running loops to do these periodic verifications. Having an isolated thread to simply monitor and control time is not the ideal solution, specially if the number of alarms required start to considerably increase. Additionally, if some configurations like defining the start and end time were needed (as described in 4.4.6, for example), it would be even harder to guarantee the synchronism required when monitoring time.

Using an event-based timer approach, these verifications are avoided and there is no need to periodically check the value of current time and compare it with all existing alarms. This event approach follows exactly the inverse of the periodic checking: instead of performing periodic validations to check if a specific time has been reached, the event-based approach tells us at the right moment that time is reached, just like an alarm clock. This way, while developing the necessary code to integrate an equipment that needs time monitoring, developers only have to focus in defining alarms correctly so that they can fire notifications at the desired date time values. Also, when using a timer service based on event notifications, the framework only needs to wait for the notification sent when the desired times are reached.

Events also play an important role in what refers to messages notification. If no events were used to inform the framework when a module wants to send a new message, the framework message center would have to ask periodically each module about new messages. Using an event-based approach, the framework not only avoids this periodic questioning but also turns message handling more efficient and effective. The framework message center does not have to worry about when messages are available because it is notified about such situations instead. When a module needs to send a message, it has to delegate it to the framework and this one does not even need to know which module sent the message.

A similar situation happens not only about framework internal messages but also when referring to YODA messages. The message center obviously needs to be listening to the network so that it can be aware of the messages available in the network. However, it does not need to tell the framework each time a new YODA message becomes available; since the messages have been previously subscribed, the framework message center will only be notified through an event when a message with a subscribed subject becomes available in the network. Moreover, it not only

notifies the message center about a new subscribed message available but also gets the message from the network and immediately captures it. By doing so, the framework does not need neither to compare all messages that become available neither to check all messages to retrieve the one desired.

6.2 Parsing XML Files

The equipment considered in the proof of concept generates data and makes it available through XML files. Because of this specific equipment, collecting data from XML files has been highly focused during the prototype development. In previous approaches regarding the data collection from these files format type, the approach used to parse XML files was similar to the parsing of a plain ASCII text file: a sequential parsing was implemented and regular expressions to match XML tags were widely used. This kind of parsing not only is more difficult to implement but it is also harder to understand and less fault tolerant if some error occur during the parsing.

The approach used to parse XML files using XPath language has brought some advantages related to the parsing operations. Using XPath language, the parsing does not need to be done sequentially. Using XPath, there is no need to keep accessing tags sequentially, using the order they appear in the file. XPath allow XML tags and their values to be easily accessed and using random accesses without losing performance. This characteristic is a great advantage because allowing a random parsing of the files increases the level of flexibility.

The way an equipment generates a XML file and its internal structure usually depends on the manufacturer of the equipment. By using a random access parsing, this dependency level is reduced because the team integrating an equipment can implement the methods required for collecting data without taking in consideration the file format specifications provided by the manufacturer. This approach becomes particularly useful if some validations about the existing data in the file are required: instead of parsing all the file sequentially until the required tags are reached, these tags can be directly accessed, which is a performance improvement.

Moreover, the usage of XPath makes the parsing easy to understand. The path for a tag is identified by the sequence of nodes existing in the hierarchy, since the root of the XML document until the required tag. Since this approach is identical to the one used by operating systems to define the paths for directories and files, it also helps understanding the hierarchy of paths for accessing tags because users and developers are already familiarized with it. This way, by using XPath, it becomes easier to understand how a XML file is being parsed and which tags are being accessed.

Additionally, when a XML file is loaded using XPath, the document is scanned once and all its tags are indexed. Having the tags of a XML document indexed increases significantly the performance when accessing the path of a XML file because when searching for a tag only the tags are considered. This way, instead of parsing and considering all existing data as happens in the sequential parsing, only the indexed tags are used during a search and the parsing of contents and values of all tags is avoided.

6.3 Database Access and Saving Data

Database access and saving data into databases are another crucial aspect of the framework. Having a pool of database connections clearly reduces the time required for database operations because once a connection is open it remains open for further databases accesses and it is closed when the application exits. The main advantage of such a pool of connections occurs because it avoids the need of open and close a connection each time an access is required.

Another interesting finding is related to the use of cache and binding variables when doing multiple database accesses executing queries with the same structure but with different parameter values. By using binding variables the queries are cached and their execution plan remains the same. Since the execution plan remains unchanged, the database does not need to calculate a new execution plan for each query and uses the one it has in cache. This way, calculating an execution plan for each similar required is not necessary and database operations are executed quickly, reducing the time needed for each operation and increasing performance.

A similar situation occurs when referring to stored procedures. Since a pool of stored procedures is used, it is possible to cache and prepare a stored procedure when it runs for the first time. This way, just like using binding variables, stored procedures can be cached and reused whenever they are required in future attempts. By doing so, the time required to execute a stored procedure multiple times decreases, which also allows a significant enhancement in terms of performance.

Even if database accesses have already been considerably improved, the usage of SQL*Loader application introduced even more performance improvements, especially by reducing the time required to save the collected data into an Oracle database. This required time to save data using this approach is significantly lesser than using the optimized approaches described in the previous paragraphs. Considering the nonexistence of errors in the data that should be saved, this approach is strongly recommended to all cases using an Oracle database as final target. It not only avoids the use of SQL queries and stored procedures but also provides a very efficient method to save data into a database.

6.4 Time Required for Integration

Finally, the most interesting finding is related to the required time to integrate the AOI equipment, collect the data it generates and then save it into an Oracle database. The time required to implement and develop the previous integration approach already used in Qimonda was about a man month. After specifying and developing the core architecture of the framework and its services, a new integration and data collection effort using the Bee Framework has started.

The required framework configurations to integrate the AOI equipment have been defined. At this point, only the specific integration strategy to collect data generated by this equipment was required. Since the framework architecture and its core services were already specified and implemented, the time required to integrate an equipment started counting. The time spent since the start of the development of the integration strategy until its end has been approximately 1.5 weeks (with a person working on it full time).

The difference between the required time to integrate an equipment in terms of collecting its data and saving is considerable and quite significative. The global framework architecture was already implemented and ready to use; the same way, the framework services had already been previously developed and were ready to use. Consequently, the only components required to integrate the AOI equipment were a new module and a concrete integration strategy. This strategy only had to be implemented accordingly with the equipment and data collection needs, taking full advantage of the existing services and architecture.

This approach only had to consider the necessary adjustments required for an integration strategy regarding the AOI equipments. By making the architecture and framework services available and ready to use, the development of an integration solution for a new equipment only has to focus in the main characteristics of integration itself. No development regarding message handling, database accesses, email or logging is required; they are available and should be used by the integration strategy.

Even if these services are required to integrate an equipment, they do not play a critical role in the integration process and the development should be almost exclusively focused on data collection tasks. These are the expected results the usage of the Bee Framework intends to achieve: avoid or reduce considerably need of developing complementary services that are not directly related to integration and data collection and make them available and ready to use instead. This is basically the main reason that explains the existence of such a big difference between the previous integration solution and the integration achieved using the **Bee Framework**.

6.5 Summary

This chapter presents the findings achieved with the development of the Bee Framework , namely:

- the benefits of using an event-based approach, especially those related to detection of changes in files and notifications;
- the achieved improvements related to the parsing of XML files;
- the main conclusions and enhancements regarding the used database access approach;
- the time required to integrate the AOI equipment.

Using an event-based approach avoids the need of periodic verifications and comparisons because the framework is notified when an event is triggered. Since the framework is notified when such events occur, unnecessary verifications are avoided and performance increases because no additional processing and memory resources are required.

XML parsing using XPath essentially allows nodes and XML tags to be indexed, which makes random access possible and also makes parsing tasks easier to implement and understand. Additionally, accessing a specific node is direct and quick because the nodes are indexed, which decreases parsing time.

Having pools for database connections and for stored procedures makes database accesses more efficient in terms of performance and required time. Using binding variables and prepared stored procedures reduces significantly the time required to execute database access operations. SQL*Loader is strongly recommended to large volumes of data because loading and saving data into a database becomes considerably faster.

The difference between the amounts of time required to integrate the AOI equipment with and without using the Bee Framework is considerable. Without the framework the time was about a month; with the framework this time has been reduced to a week and a half. The main reason for this big difference is the focus *only* in the data collection integration strategy for a specific equipment. With the additional and complementary services as well as the core global architecture implemented, they are ready to use and the development does not need to focus on them again.

Chapter 7

Conclusions

This chapter describes the main conclusions achieved with the elaboration of the project. The chapter refers to the concrete applicability of the Bee Framework and the new approach to integrate the AOI equipment and collect the data it generates. Furthermore, this chapter also includes some final recommendations and perspectives of future work to improve and expand the proposed solution.

7.1 Project Applicability

This section describes the practical applicability of the framework and the AOI integration approach developed using the Bee Framework . Beyond the specification and development of a framework architecture following some good architecture practices based on design patterns, the framework design provides an easier method to integrate manufacturing equipments. The framework provides an architecture that can be used to easily plug other equipments and start collecting the data they generate.

Moreover, the core services commonly used in the data collection have been identified, specified and implemented. This way, not only the framework has been designed with a modular architecture to help improving the equipment integration and data collection processes, but also provides these services. These services are ready to be used and can be easily configured by changing the settings in XML configuration files. This means that no new code is required to immediately start using these services and use them in the integration strategies to perform data collection in manufacturing equipments.

Both the architecture and the services proposed and implemented allow an equipment integration team to focus only in the development of the integration strategy

itself needed to integrate a new equipment type. This way, developers do not have to worry about the integration architecture for each new equipment. Additionally, they also do not have to worry through the definition of the complementary services not directly related to the data collection process itself, but required to achieve a robust equipment integration. Since these services are already available, they do not need to focus their attention on the architecture and requirements for these services. Developers only need to know which services are available and how to configure them correctly so that they can be used as desired.

When integrating an equipment using the Bee Framework , instead of adapting some of the components required and previously developed in other equipment integrations, these components remain unchangeable. This way, instead of using an approach inspired by the expression "adapt and reuse", integrating an equipment using the framework and its services is based on the expression "configure and use". Consequently, using the Bee Framework significantly reduces the amount of time and effort required to integrate an equipment and perform data collection.

Beyond the functionalities and easy reutilization of framework components in other equipment integrations, the usage of the Bee Framework to collect data also has another great advantage: it promotes consistency. If collecting data from manufacturing equipments starts to progressively use this framework, the data collection process will have a tendency to become very uniform both on new equipment integrations and on maintenance tasks, even considering different equipment types. This desired consistency level will not only help the framework to become more mature and stable, but will also lead to new releases considering new improvement issues or even new requirements.

Since Qimonda assembly lines are working non stop 24h per day and because they follow very rigorous and strict security and safety rules, the migration of the data collection process regarding AOI equipments must be carefully planned. The impact of an emergency stop in Qimonda assembly line is extremely harmful, so this new integration approach using the Bee Framework must also be exhaustively tested to ensure that no problems are detected. This is essentially the main reason that justifies why this new approach is still not fully operational.

7.2 Final Recommendations and Perspectives of Future Work

This section details some final recommendations about the Bee Framework project and describes some perspectives of future work and improvements. Even if the framework architecture and its services have been tested, especially when referring to the AOI equipment integration, they should be exhaustively tested in the same conditions of the assembly line, executing stress tests to ensure that everything

works as expected and to guarantee that the data collection approach using the framework can be safely used in the assembly line.

The perspectives of future work regarding the Bee Framework at short and middle term are related to new requirements or improvements. Some of these perspectives are:

- Automatic update of the framework modules and services.

  This perspective of future work is perhaps the most important. It is similar to the one described in section 4.3.2 regarding the automatic updates of assembly files (DLLs). This feature has been implemented in the cases related to external assemblies, because they do not have dependencies with the framework. However, when an update is found it has to unload the DLLs currently used, load the new ones and ensure the new ones will be used afterward. These automatic updates are consequently harder to do than the updates related to external assemblies.

- Monitor the state of all framework modules and their properties.

  Monitoring the state of the framework modules is other interesting perspective. This perspective can be useful to check what the module is doing, which internal framework messages or YODA messages it has subscribed, or other internal properties related to each specific module.

- Monitor the state of all framework services.

  This item is similar to the previous one. Monitor a framework service could be useful to check if the service is working properly or which configurations are currently used. For example, when considering the database service, it would be good to see which databases are available or monitor the state of the different connections (open or closed).

- Monitor the messages handled by the *MessageCenter*.

  This perspective is related to the framework *MessageCenter*. It considers the development of a new interface to monitor all the messages sent or received. Furthermore, this interface would also consider the parameters of each framework message, allowing users to inspect which messages have been handled and which are their characteristics in terms of parameters.

- Graphical interface to configure module and services (instead of XML files).

  Currently, the framework loads its own configurations and the settings related to each module and service from XML files. This means that all configurations must be defined manually using a XML file editor, which may increase the

number of errors committed. Consequently, it would be very helpful to have a graphical user interface to configure all the settings required by the framework. This way, XML files could still be used, but users would not have to know the XML language syntax or the exact syntax configurations need.

- Include new methods for collecting data, such as the RS-232 (serial port).

  As described before in section 4.1.1, there are other possible sources for collecting data from equipments, such as the serial port. In order to provide a better coverage in terms of possible data sources, new additional methods for collecting data from these source should be considered and implemented in the framework. This way, the framework domain for collecting data would be expanded and the number of equipments covered by the framework in terms of data collection would also increase.

- Read configurations from system *Registry*.

  Some equipment types use the system *Registry* to read some of the required configurations used in the data collection process. This perspective of future work is not hard to implement and is not critical but would be nice to have this feature available and ready for use just like the other framework services.

- Generate statistics related to data collection.

  This final perspective of future work is useful to evaluate the framework performance itself. For example, it could be used to calculate the time required to send messages between modules or to send messages to the YODA network, to calculate the average time to load data into a database or the time required to parse a file. These calculated values could then be used to generate statistical reports about the data collection process related to the Bee Framework .

7.3 Final Conclusions

The results achieved with this project are positive and it is expected to integrate the AOI equipment in the productive environment of the Qimonda assembly lines using this approach shortly.

In a middle term it is also expected that other different equipment types can also be integrated using the Bee Framework , especially the new equipment types that can be acquired by Qimonda. In a long term, the previous integration approaches used in data collection for existing equipment types may as well be progressively migrated in order to use the framework solution proposed. This would clearly lead to a higher level of consistency regarding the data collection process from manufacturing equipments in Qimonda assembly lines.

References

[1] Danforth, B.N., Sipes, S., Fang, J., Brady, S.G. The History of Early Bee Diversification Based on Five Genes Plus Morphology. *Proceedings of the National Academy of Sciences 103*, 2006.

[2] LocalHarvest, Inc. Honey and Bee Products, 2008. Available from http://www.localharvest.org/store/bee-prods.jsp, last accessed at 3rd June 2008.

[3] Krell, R. *Value-Added Products From Beekeeping*. Food and Agriculture Organization, 1996. Available from http://www.fao.org/docrep/w0076e/w0076e00.htm, last accessed at 3rd June 2008.

[4] Information Society: The Next Steps Coming Soon, January 2006. Available from http://topics.developmentgateway.org/special/informationsociety/index.do, last accessed 30th June 2008.

[5] Ozkul, T. *Data Acquisition and Process Control Using Personal Computers*. CRC, April 1996.

[6] Park, J., Mackay, S. *Practical Data Acquisition for Instrumentation and Control Systems*. Newnes, First edition, June 2003.

[7] James, K. *PC Interfacing and Data Acquisition*. Newnes, August 2000.

[8] Iskow, J. SEMI Equipment Data Acquisition Standards. June 2005. Available from http://www.semiconductor.net/article/CA604510.html, last accessed at 30th June 2008.

[9] Rubow, B. The Standard Pieces of SEMI's Interface A. July 2005. Available from http://www.semiconductor.net/article/CA621798.html, last accessed at 30th June 2008.

[10] Data Collection — Process and Data Automation, 2008. Available from http://www.processanddata.com/data-collection/, last accessed at 28th June 2008.

[11] SEMI, 2008. Available from http://www.semi.org, last accessed at 29th June 2008.

[12] PANalytical. SECS/GEM, 2008. Available from http://www.panalytical.com/index.cfm?pid=204, last accessed at 29th June 2008.

[13] Crispieri, G. Improving Fab Productivity with New Standards for Equipment Data Acquisition. January 2007.

[14] Aaron, H. Fab Automation Gets Boost From Interface A, June 2005. Available from http://www.semiconductor.net/article/CA6343501.html, last accessed at 30th June 2008.

[15] Marsh, T., Eisfeld, M. Qimonda EDA Evaluation — Final Recommendations Report. Technical report, February 2008. Qimonda internal document with restricted access.

[16] Microsoft Corporation. Visual C# Developer Center, 2008. Available from http://msdn.microsoft.com/en-us/vcsharp/default.aspx, last accessed at 3rd June.

[17] Microsoft Corporation. C# *Language Specification Version 3.0*. 2007. Available from http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/CSharp%20Language%20Specification.doc, last accessed at 3rd June 2008.

[18] Microsoft Corporation. Visual Studio Developer Center, 2008. Available from http://msdn.microsoft.com/en-us/vstudio/default.aspx, last accessed at 3rd June 2008.

[19] Microsoft Corporation. Visual Studio 2005 Developer Center, 2005. Available from http://msdn.microsoft.com/en-us/vs2005/default.aspx, last accessed at 3rd June 2008.

[20] Resharper:: The Most Intelligent Add-In To Visual Studio, 2008. Available from http://www.jetbrains.com/resharper/, last accessed at 4th June 2008.

[21] Nunit, 2007. Available from http://www.nunit.org/, last accessed at 4th June 2008.

[22] Dustin, E. *Automated Software Testing*. Addison Wesley, 1999.

[23] Cockburn, A. *Agile Software Development*. Pearson Education, First edition, October 2001.

[24] Object Management Group — UML, 2008. Available from http://www.uml.org, last accessed at 4rd June 2008.

[25] Introduction to OMG UML, September 2008. Available from http://www.omg.org/gettingstarted/what_is_uml.htm, last accessed at 6th June 2008.

[26] History of UML. Available from http://atlas.kennesaw.edu/~dbraun/csis4650/A\&D/UML_tutorial/history%_of_uml.htm, last accessed at 6th June 2008.

[27] Microsoft Corporation. Microsoft Office Visio 2007 Product Overview — Visio — Microsoft Office Online, 2008. Available from http://office.microsoft.com/en-us/visio/HA101656401033.aspx, last accessed at 6th June 2008.

[28] Oracle Corporation. Oracle Database, 2008. Available from http://www.oracle.com/database/index.html, last accessed at 6th June 2008.

[29] Oracle Corporation. What is SQL Developer?, 2008. Available from http://www.oracle.com/technology/products/database/sql_developer/files/what_is_sqldev.html, last accessed at 6th June 2008.

[30] SQL*Loader FAQ — Oracle FAQ, 2008. Available from http://www.orafaq.com/wiki/SQL*Loader_FAQ, last accessed at 6th June 2008.

[31] Microsoft Corporation. Enterprise Library, 2008. Available from http://msdn.microsoft.com/en-us/library/cc467894.aspx, last accessed at 4th June 2008.

[32] Microsoft Corporation. Enterprise Library 3.1 — May 2007, 2007. Available from http://msdn.microsoft.com/en-us/library/aa480453.aspx, last accessed at 4th June 2008.

[33] Pereira, L. Microsoft Patterns & Practices: The Enterprise Library. 2008. Available from http://www.dotnetheaven.com/Uploadfile/leonpere/EnterpriseLibrary02012006061856AM/EnterpriseLibrary.aspx, last accessed at 4th June 2008.

[34] Microsoft Corporation. Enterprise Library Documentation, 2007.

[35] Microsoft Corporation. Enterprise Library 4.0 — May 2008, 2008. Available from http://msdn.microsoft.com/en-us/library/cc512464.aspx, last accessed at 4th June 2008.

[36] Red Hat Middleware. NHibernate for .NET, 2006. Available from http://www.hibernate.org/343.html, last accessed at 6th June 2008.

[37] Red Hat Middleware. NHibernate for .NET - Download Overview, 2006. Available from http://www.hibernate.org/6.html, last accessed at 6th June 2008.

[38] Enterprise Library, 2005. Available from http://www.theserverside.net/discussions/thread.tss?thread_id=32864, last accessed at 6th June 2008.

[39] TIBCO Software Inc. TIBCO Rendezvous, 2008. Available from http://www.tibco.com/software/messaging/rendezvous/default.jsp, last accessed at 6th June 2008.

[40] Donohoe, D. J., Neth, S. R., Kim, Y. B., Zak, B. D. Name Type Value Storage. Available from http://www.freepatentsonline.com/6401092.html, last accessed at 6th June 2008, YEAR=2002, MONTH=June.

[41] Microsoft Corporation. TIBCO Rendezvous Concepts, December 2007. Available from http://www.gorillatraining.com/en-us/library/aa559569.aspx, last accessed at 6th June 2008.

[42] YODA qShare Site, 2008. Available from http://qshare.qimonda.com/sites/it-mfg-yoda/quickplace/default.aspx, last accessed at 6th June 2008. Qimonda internal page with restricted access.

[43] Microsoft Corporation. Message Queueing (MSMQ), 2008. Available from http://msdn.microsoft.com/en-us/library/ms711472.aspx, last accessed at 16th March 2008.

[44] Mitchell, S. Microsoft Message Queue — An Overview. April 2000. Available from http://www.4guysfromrolla.com/webtech/041300-1.shtml, last accessed at 16th March 2008.

[45] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F. *Extensible Markup Language (XML) 1.0.* World Wide Web Consortium, Fourth edition, August 2006. Available from http://www.w3.org/TR/2006/REC-xml-20060816/, last accessed at 6th June 2008.

[46] Extensible Markup Language (XML), May 2008. Available from http://www.w3.org/XML/, last accessed at 6th June 2008.

[47] XML Introduction — What is XML?, 2008. Available from http://www.w3schools.com/xml/xml_whatis.asp.

[48] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., Cowan, J. Extensible Markup Language, 2004. Available from http://www.stylusstudio.com/w3c/xml11/sec-intro.htm, last accessed at 6th June 2008.

[49] XML Usage, 2008. Available from http://www.w3schools.com/xml/xml_usedfor.asp.

[50] Introduction to XPath, 2008. Available from http://www.w3schools.com/Xpath/xpath_intro.asp, last accessed at 7th June 2008.

[51] Clark, J., DeRose, S. XML Path Language, 2004. Available from http://www.stylusstudio.com/w3c/xpath/index.htm, last accessed at 7th June 2008.

[52] Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., Siméon, J. *XML Path Language (XPath) 2.0.* World Wide Web Consortium, January 2007. Available from http://www.w3.org/TR/xpath20, last accessed at 7th June 2008.

[53] Haddad, P., Donoghue, T. Choosing an Approach for Locking, 1998. Available from http://developer.apple.com/documentation/legacytechnologies/webobjects/webobjects_4.5/System/Documentation/Developer/WebObjects/Topics/ProgrammingTopics.2a.html#15618, last accessed at 4th March 2008.

[54] De Beijer, M. Database Concurrency Conflicts in the Real World. July/August 2006. Available from http://www.code-magazine.com/article.aspx?quickid=0607081\&page=1, last accessed at 4th March 2008.

[55] Microsoft Corporation. Optimistic Concurrency. Available from http://msdn2.microsoft.com/en-us/library/aa0416cz(VS.71).aspx, last accessed at 4th March 2008.

[56] Haddad, P., Donoghue, T. Locking on a Column, 1998. Available from http://developer.apple.com/documentation/legacytechnologies/webobjects/webobjects_4.5/System/Documentation/Developer/WebObjects/Topics/ProgrammingTopics.2b.html#10467, last accessed at 4th March 2008.

[57] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Sixth edition, April 1996.

[58] Reilly, D. Introducing the Singleton - "The Single Java Object", June 2006. Available from http://www.javacoffeebreak.com/articles/designpatterns/index.html, last accessed at 24th March 2008.

[59] Microsoft Corporation. Singleton, 2008. Available from http://msdn.microsoft.com/en-us/library/ms998426.aspx, last accessed at 24th March 2008.

[60] Singleton Pattern, October 2007. Available from http://c2.com/cgi/wiki?SingletonPattern, last accessed at 24th March 2008.

[61] Cohen, T., Gil, J. Better Construction with Factories. *Journal of Object Technology*, 2007. Available from http://tal.forum2.org/static/cv/Factories.pdf, last accessed at 24th March 2008.

[62] Goel, A. ONDotnet.com — The Factory Design Pattern, August 2003. Available from http://www.ondotnet.com/pub/a/dotnet/2003/08/11/factorypattern.html, last accessed at 19th June 2008.

[63] Data & Object Factory. Factory Method Design Pattern in C# and VB.NET. Available from http://ww.dofactory.com/Patterns/PatternFactory.aspx, last accessed at 19th June 2008.

[64] Purdly, D., Richter, J. Exploring the Observer Pattern, January 2002. Available from http://msdn.microsoft.com/en-us/library/,ms954621.aspx, last accessed at 20th June 2008.

[65] Vlissides, J. Design Patterns Project, August 2001. Available from http:/www.research.ibm.com/designpatterns/example.htm, last accessed at 20th June 2008.

[66] Minka, T. Introduction to Software Pattern — Observer Pattern, January 1997. Available from http://alumni.media.mit.edu/simtpminka/patterns/Observer.html, last accessed at 20th June 2008.

[67] Fowler, M. Inversion of Control Containers and the Dependency Injection Pattern, January 2004. Available from http://martinfowler.com/articles/injection.html#InversionOfControl, last accessed at 23rd June 2008.

[68] Johnson, R., Foote, B. Designing Reusable Classes. *Journal of Object-Oriented Programming*, June/July 1988. Available from http://www.laputan.org/drc/drc.html, last accessed at 23rd June 2008.

[69] Hollywood Principle, February 2008. Available from http://c2.com/cgi/wiki?HollywoodPrinciple, last accessed at 23rd June 2008.

[70] Hadlow, M. Code rant: The Hollywood Principle, October 2007. Available from http://mikehadlow.blogspot.com/2007/10/hollywood-principle.html, last accessed at 23rd June 2008.

[71] Grand, M. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*. Wiley, Second edition, September 2002.

[72] Lewis, M. C. Lecture 3 — Chain of Responsibility and Iterator, December 2007. Available from http://sol.cs.trinity.edu/~mlewis/CSCI3394-F07/Lectures/Lect3.pdf, last accessed at 21st June 2008.

[73] CyberOptics Corporation. Automatic Optical Inspection. Available from http://www.cyberoptics.com/products/aoi/aoi/, last accessed at 28th June 2008.

[74] CyberOptics Corporation. Flex Ultra — Automatic Optical Inspection, 2006. Available from http://www.cyberoptics.com/client_files/documents//8010020-Rev_B_-_Flex_Ultra.pdf, last accessed at 28th June 2008.

Index

Appendices

Appendix A

Bee Framework Configurations

This appendix illustrates how the XML file related to the global framework settings must be configured. This XML file is required when starting the application and considers two main XML nodes: `modules` and `services`.

As it name reveals, the main node `modules` contains the XML nodes related to the common definitions used by framework modules. Each of these sub nodes contain the follow information:

- `name` — the name of the module.

  The module can have any desired name. However, the value of this node should be chosen so that it can be easily distinguished from other modules. A meaningful name should also be used, so that the module purpose can also be immediately understood when the module is referred. The only limitation for this node value is that it has to be different from all other module names, so that it can be uniquely identified by the framework.

- `type` — the framework module type.

  This node refers to the module type internally used by the framework. This value *must* be one of the defined module types known by the framework. Because of the abstraction used to build the modules considering the Factory Method pattern explained in 4.3 section, these values are used to identify the module type that should be created. Additionally, multiple modules from the same type may co-exist, with the condition of respecting the unique names rules explained in the previous item.

- `ConfigurationPath` — the path for the XML configuration file of the module.

  This node value contains the path for the XML configuration file used by the framework module to load specific module configurations. The path of this XML file can be either relative to the path of the application or absolute.

Each module configuration file must also define, at least, the subjects of both internal framework and YODA messages that it wants to subscribe. To specify these message subjects, two main nodes are used: `messages` and `YodaMessages`, respectively. These main nodes can then contain a variable number of child nodes, each one specifying a message subject. The names of these child nodes are `message`

or `YodaMessage`, depending if they are related to internal framework messages or with YODA messages, respectively.

An example is illustrated in the following XML extract:

```
<messages>
  <message>List</message>
</messages>

<YodaMessages>
  <YodaMessage>CommandLot</YodaMessage>
</YodaMessages>
```

The `services` node contains the definitions for the services which configurations are global to the framework, namely the YODA Service and the Email Service.

Since multiple YODA configurations can be used, there is main node to define the YODA configurations required. The parent node or this service is the `yodas` node, which defines both the global settings related to YODA configurations and the list of possible YODA configurations.

The global settings related to YODA are:

- `CFGmgrSubject` — the subject of the Configuration Manager service.

  The Configuration Manager is a service listening the YODA network, so that it can receive the messages sent asking for the configuration settings of an application or equipment, for example. Consequently, if an equipment framework module needs to know which settings should be used to collect from an equipment, a YODA message with the required settings (usually the name of the parameters settings) should be sent to the YODA network using this Configuration Manager subject.

- `BaseSubject` — the base subject common to all YODA messages related to Bee Framework.

  In order to decrease the probability of mixing and wrongly receiving messages from the YODA network, a base subject is used to all YODA messages related to the framework. Obviously, each YODA message has its own subject, but it is mandatory that it starts by this base subject. This way, there is no need to specify the complete subject of YODA messages.

- `default` — the name of the default YODA configuration used when any configuration is specified.

  Since multiple YODA configurations can be defined and used, this node value defines the name of the YODA configuration that should be used by default when any YODA configuration is defined. This information is useful to avoid developers to always need to specify the YODA configuration required. Since most of times the default configuration will be used, they will not need to specify the desired YODA configuration each time the YODA Service is used.

Additionally, for each YODA configuration, a parent node standing for `yoda` should encapsulate all the nodes required for defining a YODA configuration settings. The `yoda` node should be located inside the `yodas` node. The required settings for each YODA configuration are detailed next:

- `name` — the name of the YODA configuration.
- `AppName` — the name of the application.
- `SubjectPrefix` — the IFX Subject Prefix.
- `RvNetwork` — the main transport configuration for the Network.
- `RvDaemon` — the main transport configuration for the Daemon.
- `RvService` — the main transport configuration for the Service.
- `LongRequests` — the node that defines the configurations of long requests messages.

  Long requests allows applications to send YODA messages following a request-reply mechanism. However, unlike simple normal request-reply messages, applications do not need to "block" waiting for the reply. When the reply for the request is available in the YODA network, the application simply gets notified and catches the reply. The settings used by long requests are presented next:

  - `LRFile` — the file used to store pending request-reply messages.
    In order to avoid the need of keeping the pending requests in memory, a file is used to store information.
  - `Interval` — the amount of time (in seconds) to clean the file used for long requests.
    The file used to store pending request-reply messages is periodically clean, so that the requests that have caused timeout or that have been attended between two consecutive inspections can be removed from the file. This helps maintaining the long requests file only with the necessary information and avoids that it keeps continually growing in size.
  - `Subject` — the subject an application requires to receive the replies.
    Since the application do not need to wait for a reply, it has to be notified when the reply becomes available in the network. Consequently, it has to subscribe a subject to catch the reply messages still present in the file used to store long requests.

Finally, the last configurations of these global settings file is related to the email definitions in terms of the SMTP server used to send emails with the Email Service. The global configurations required have the `email` node as a parent node. These configuration are:

- `smtpHost` — the host used by the SMTP server.
- `smtpPort` — the port number used by the SMTP server.

An example of a Bee Framework configuration file is presented next.

```
<?xml version="1.0" encoding="utf-8" ?>
<framework>

  <modules>
```

```
    <module>
      <name>Folder Monitor</name>
      <type>Folder Monitor Module</type>
      <ConfigurationPath>FolderMonitorConfiguration.xml</ConfigurationPath>
    </module>
    <module>
      <name>Automatic Optical Inspection</name>
      <type>AOI Equiment Module</type>
      <ConfigurationPath>AOIModuleConfiguration.xml</ConfigurationPath>
    </module>
    <module>
      <name>Backup</name>
      <type>Backup Module</type>
      <ConfigurationPath>BackupConfiguration.xml</ConfigurationPath>
    </module>
  </modules>

  <services>
    <yodas>
      <CFGmgrSubject>YOD_D.ITPAESEC.CFGmgr</CFGmgrSubject>
      <BaseSubject>BeeFw.</BaseSubject>
      <default>DefaultYodaConfiguration</default>
      <yoda>
        <name>DefaultYodaConfiguration</name>
        <AppName>Bee Framework</AppName>
        <SubjectPrefix>YOD_D</SubjectPrefix>
        <RvNetwork>;239.255.28.99</RvNetwork>
        <RvDaemon>7777</RvDaemon>
        <RvService>7777</RvService>
        <LongRequests>
          <LRFile>longrequests.lr</LRFile>
          <Interval>60</Interval>
          <Subject>POR.MEI.EC.BEEFW.LONGREQUESTS.REPLY</Subject>
        </LongRequests>
      </yoda>
    </yodas>

    <email>
      <smtpHost>smtp-por.intra.qimonda.com</smtpHost>
      <smtpPort>25</smtpPort>
    </email>

  </services>
</framework>
```

Appendix B

Services Configurations

This appendix describes the configurations related to the Bee Framework services referred in section 4.4.

B.1 YODA and Message Services

As described in the Message Handling section (see section 4.3.4), both YODA and Message services are part of the framework kernel. Consequently, these services are directly related to the framework configurations and module configurations that have been already described in Appendix A.

The Message Service is simpler than the YODA Service because it does not require additional configurations. Each framework module only has to contain in its XML configuration file a parent node named `messages`. This parent node encapsulates the subjects of internal framework messages. Each message subject must then be defined inside a child node tag named `message` (see Appendix A) .

These are the only definitions required to use the message service. The framework is responsible read this information from each module configuration file and then configure the *MessageCenter* so that it can deliver the received messages accordingly to the framework modules.

The YODA Service is also related to the *MessageCenter*, so its configurations are also related to the framework main configuration file (as described in Appendix A).

Each framework module contains in its XML configuration file a parent node named `YodaMessages`. This parent node encapsulates the subjects of YODA messages. Each YODA message subject must then be defined inside a child node tag named `YodaMessage` (see Appendix A). These subjects must be simple, without considering the base subject. To avoid the need of specifying the complete subject for all YODA messages, a base subject is defined and the framework is then responsible to append the base subject to all *YodaMessage* subjects.

B.2 Database Service

The configurations related to Database Service are mostly related to the XML configuration file required. In order to use the Data Application Block from Enterprise

Library, a XML file is required to define database configurations used by the corresponding service. A new XML file is necessary only because the values used to database configurations may be directly accessed without any additional parsing operations. This file must be named *App.config* and must be present in the same directory of the executable file so that their configurations can be accessed.

This XML file contains four sections:

- `configSections` — It is related to the Microsoft Enterprise Library settings in order to allow the inclusion of the configurations required to allow database abstractions to be used.
- `dataConfiguration` — It defines the default database that should be used when no database is specified.
- `oracleConnectionSettings` — This section defines additional configurations related to Oracle database connection strings, namely those related to Oracle packages.
- `connectionStrings` — This section is the most important because it defines the connection strings that can be used by the data collection application. Each connection string is added by using the `add` node, which contains the following attributes:
  - `name` — the name of the connection string. Any name may be used, but it has to be unique so that the connection string can be uniquely identified.
  - `connectionString` — the connection string itself.
  - `providerName` — the provider name that refers to the database type (Oracle or SQL Server, for example).

The settings existing in this XML file can be automatically generated if using the Enterprise Library Configuration GUI and exporting the configurations to the *App.config* file.

The XML code presented bellow shows an example of an *App.config* file, which sections are described in the following items:

- The first section, `configSections` defines the Microsoft Enterprise Library configurations required that are related to database and Oracle connection string settings.
- `dataConfiguration` section defines the default database used if no database is specified in a database operation. In this example, the default database is the database identified by the connection string named *AOI Connection String*.
- The third section, `oracleConnectionSettings`, defines the name of the Oracle packages used by each database. In the following example, a package named *Oracle Package* is related to the database identified by *AOI Connection String*.

- Last section of the example bellow contains three different connection strings, obviously related to three different databases. If a database operation related to a database different from the default one is required, the name of the connection string that refers to the required database must be specified.

```
<configuration>

  <configSections>
    <section name="dataConfiguration"
             type="Microsoft.Practices.EnterpriseLibrary.Data.
                   Configuration.DatabaseSettings,
                 Microsoft.Practices.EnterpriseLibrary.Data,
                 Version=3.1.0.0,
                 Culture=neutral,
                 PublicKeyToken=b03f5f7f11d50a3a" />
    <section name="oracleConnectionSettings"
             type="Microsoft.Practices.EnterpriseLibrary.Data.Oracle.
                   Configuration.OracleConnectionSettings,
                 Microsoft.Practices.EnterpriseLibrary.Data,
                 Version=3.1.0.0,
                 Culture=neutral,
                 PublicKeyToken=b03f5f7f11d50a3a" />
  </configSections>

  <dataConfiguration defaultDatabase="AOI Connection String" />

  <oracleConnectionSettings>
    <add name="AOI Connection String">
      <packages>
        <add prefix="AOI Oracle Package"
             name="Oracle Package" />
      </packages>
    </add>
  </oracleConnectionSettings>

  <connectionStrings>
    <add name="APT Connection String"
         connectionString="Data Source=LPCTEST;Persist Security Info=True;
                           User ID=APT;Password=APTTEST;Unicode=True;"
         providerName="System.Data.OracleClient" />
    <add name="Local Connection String"
         connectionString="Data Source=localhost;Persist Security Info=True;
                           User ID=DANIEL;Password=4Bee_Qpt"
         providerName="System.Data.OracleClient" />
    <add name="AOI Connection String"
         connectionString="Data Source=LPCTEST;User ID=EC_BEE;
                           Password=DCF#Bee_EC4"
         providerName="System.Data.OracleClient" />
  </connectionStrings>

</configuration>
```

B.3 Email Service

The configurations of the Email Service have already been presented in Appendix A. These configurations are related to the SMTP server settings, namely the SMTP

host and the port number. An example of these configurations is presented in the following XML code.

```
<email>
    <smtpHost>smtp.por.intra.qimonda.com</smtpHost>
    <smtpPort>25</smtpPort>
</email>
```

Then, to use this service, the method *SendEmailMessage* should be used. This method have multiple overloads, but the most complete has the following parameters:

- `from` — Email address of the sender;
- `displayName` — Display name of the sender;
- `to` — List of email addresses of the receivers (emails separated by semi-colons);
- `subject` — The subject of the email;
- `body` — The body of the email text;
- `cc` — List of email addresses of the CC receivers (emails separated by semi-colons);
- `bcc` — List of email addresses of the BCC receivers (emails separated by semi-colons);
- `attachments` — Array of attachments file paths.

B.4 Logging Service

The settings for a Logging Service are defined in the XML configuration file of the module that wants to use the service. The Logging Service is defined in the `loggings` node. This parent node have the following child nodes:

- the name of the logging identifier that should be used if no logging definitions are specified. This name refers to the name of the logging service and its value is encapsulate inside the `DefaultLogging` node.
- at least one `log` node. Each `log` node contains the settings of a Logging Service. Multiple Logging Services may be defined if ensuring that their names are unique, so that Logging Services can be uniquely identified by the framework.

Each `log` node contains the following settings:

- `name` — The name of the Logging Service.

  It is used to uniquely identify a Logging Service. Consequently, the value of this node must be unique and can not be used as the name for other Logging Services.

- `filename` — The full path of the main file used by the Logging Service.

  Each time the Logging Service is requested to log a message, the file specified by this node value is used. Depending on *rolling* settings, a rolling operation may be performed. However, when a message is logged, this message is the most recent log message so it will *always* be logged in this file.

- `type` — The type of the Logging Service.

  The type of a Logging Service may assume the following values:

  - `Flat` — If this type is used it means that no control about the size or age of the log file is done by the Logging Service. This is the simplest type of Logging Service, since all messages will be logged to the file specified in `filename` and the log file size will grow without any control as messages are logged.
  - `Rolling` — This type is more complex than the previous one because it allows better control of log files. Logging messages using a Logging Service with this type allows the control of log files size, age and the maximum number of log files in the same directory.

- `header` — The header of a log message.

  Each time a message is logged using a Logging Service, the header message defined in this node is used. If the node has an empty value, no header is written when the Logging Service is requested to log a message.

- `footer` — The footer of a log message. This node behavior is similar to the `header` node.

- `SizeUnit` — The unit used to control the size of log files. The size itself is specified by the value of the `size` node.

  This node can have the following values:

  - `Kilobytes`;
  - `Megabytes`;
  - `Gigabytes`;
  - `None` — If such value is used, a Logging Service using *Rolling* type does not perform rolling operations based on the size of log files.

- `size` — The maximum size allowed for log files.

  The value of this node is used in combination with the value of the previous node, `SizeUnit`. For example, if the value of this node is 2 and the `SizeUnit` is configured as `Kilobytes`, the maximum size allow for the log files of that Logging Service is 2KB.

  The value of this node is affected by the `SizeUnit` node value in two particular conditions:

  - If the size unit value configured in `SizeUnit` node is different from `None`, the value of the `size` node must be an integer value higher than zero.

  - If the size unit value configured in `SizeUnit` node is equal to `None`, the value used to control the size of log files of the Logging Service is immediately set to zero, without taking in consideration the value specified in the `size` node. Consequently, no control based on the size of files is done.

- `AgeUnit` — The unit used to control the age of log files. The age itself is specified by the value of the `age` node.

  This node can have the following values:

  - `Minutes`;
  - `Hours`;
  - `Days`;
  - `Weeks`;
  - `Months`;
  - `None` — If such value is used, a Logging Service using *Rolling* type does not perform rolling operations based on the age of log files.

- `age` — The maximum age allowed for log files.

  The value of this node is used in combination with the value of the previous node, `AgeUnit`. For example, if the value of this node is 3 and the `AgeUnit` is configured as `Days`, the maximum age allowed for the log files of that Logging Service is 3 days.

  The value of this node is affected by the `AgeUnit` node value in two particular conditions:

  - If the age unit value configured in `AgeUnit` node is different from `None`, the value of the `age` node must be an integer value higher than zero.
  - If the age unit value configured in `AgeUnit` node is equal to `None`, the value used to control the age of log files of the Logging Service is immediately set to zero, without taking in consideration the value specified in the `age` node. Consequently, no control based on the age of files is done.

- `MaxNumberFiles` — The maximum number of log files allowed in the same directory.

  When a Logging Service has its type specified as *Rolling*, a control based on the size and / or age of log files is typically done. This means that when the size or age limits are reached, a new log file may possibly be created depending on the number of log files in the logging directory. If the maximum number of log files allowed and the limits for both age and size values are reached, two special cases may occur when a message is logged:

  - the oldest log file is deleted so that a new log file can be created and the message logged;
  - the oldest log file is moved to a backup directory and a new log file may then be created.

If this node value is equal to 0 (zero), no control based on the maximum number of files allowed in the logging directory is done. Consequently, new log files are created in the logging directory if the age or size values of a log file are reached. In such situation, all files stay in the logging directory, the same of the main log file.

- `BackupDirectory` — The path of the directory used to backup log files.

  The value of this node is affected by the following conditions:

  - If the value of `MaxNumberFiles` node is zero, the value of this node is ignored because no rolling operations are based on the maximum number of files allowed and consequently files are not moved.
  - If the value of this node is empty then no backup directory is specified. Consequently, if the maximum number of log files in the logging directory is reached, the file is deleted and no backup of log files is done.

The following XML code shows an example of the logging structure that should be used. It contains two Logging Services named *Logging Service 1* and *Logging Service 2*. When logging a message, if no Logging Service is specified, *Logging Service 1* is used by default.

```
<loggings>
    <DefaultLogging>Logging Service 1</DefaultLogging>
    <log>
        <name>Logging Service 1</name>
        <filename>...</filename>
        <type>Flat | Rolling</type>
        <header>...</header>
        <footer>...</footer>
        <SizeUnit>Kilobytes | MegaBytes | Gigabytes | None</SizeUnit>
        <size>...</size>
        <AgeUnit>Minutes | Hours | Days| Weeks| Months| None</AgeUnit>
        <age>...</age>
        <MaxNumberFiles>...</MaxNumberFiles>
        <BackupDirectory>...</BackupDirectory>
    </log>
    <log>
        <name>Logging Service 2</name>
        <filename>...</filename>
        <type>Flat | Rolling</type>
        <header>...</header>
        <footer>...</footer>
        <SizeUnit>Kilobytes | MegaBytes | Gigabytes | None</SizeUnit>
        <size>...</size>
        <AgeUnit>Minutes | Hours | Days| Weeks| Months| None</AgeUnit>
        <age>...</age>
        <MaxNumberFiles>...</MaxNumberFiles>
        <BackupDirectory>...</BackupDirectory>
    </log>
</loggings>
```

After specifying the Logging Services settings in the XML files of the modules that want to use them, a message can be logged by using the following methods, which already consider the different categories and severity levels:

- *LogErrorMessage* — Logs an error message.

- *LogWarningMessage* — Logs a warning message.
- *LogApplicationMessage* — Logs an application message.

B.5 Timer Service

The settings for a Timer Service are defined in the XML configuration file of the module that wants to use the service. The Timer Service is defined in the `timers` node. This parent node must the contain at least one `timer` node. Each `timer` node contains the settings of a Timer Service. Multiple Timer Services may be defined if ensuring that their names are unique, so that Timer Services can be uniquely identified by the framework.

Each `timer` node may contain the following settings:

- `name` — The name of the Timer Service.

  It is used to uniquely identify a Timer Service. Consequently, the value of this node must be unique and can not be used as the name for other Timer Services.

- `repeat` — A boolean flag used to configure if notifications should or not be sent after the first elapsed time event.

  If configured as true, a notification will be sent periodically when the configured time is elapsed, depending on the interval specified.

  It is not a mandatory node and if this node is not specified the flag value is automatically set to *false*, which means that the elapsed time event will be triggered only once.

- `IntervalUnit` — The unit used to control the amount of time elapsed. The time itself is specified by the value of the `interval` node.

  This node can have the following values:

  - `Millisecond`;
  - `Second`;
  - `Minute`;
  - `Hour`;
  - `Day`;
  - `Week`;
  - `None` — If such value is used, the `interval` node is not considered. However, in such situation, the `ComplexInterval` node must exist and should be correctly configured. This node will be explained soon in this list of items.

  The value of this note only considers a single time unit at each time. In order to use multiple time units, `ComplexInterval` node must be used.

- `interval` — The amount of time used to trigger the time elapsed event.

  The value of this node is used in combination with the value of the previous node, `IntervalUnit`. For example, if the value of this node is 5 and the `IntervalUnit` is configured as `Minute`, the Timer Service will trigger the elapsed time event and send a notification 5 minutes after receiving the order to start counting elapsed time.

  The value of this node is affected by the `IntervalUnit` node value in two particular conditions:

  - If the interval unit value configured in `IntervalUnit` node is different from `None`, the value of the `interval` node must be an integer value higher than zero.
  - If the interval unit value configured in `IntervalUnit` node is equal to `None`, the value used to specify the interval is immediately set to zero, without taking in consideration the value specified in the `interval` node. In such situation, `interval` node is ignored and the `ComplexInterval` node is mandatory.

- `ComplexInterval` — The amount of time used to trigger the time elapsed event, but using multiple time units.

  Unlike `IntervalUnit` node, the `ComplexInterval` node allows a Timer Service to specify the amount of time elapsed till a notification is sent in multiple time units. If this node is used, the following child nodes must also be defined and configured:

  - `millisecond`;
  - `second`;
  - `minute`;
  - `hour`;
  - `day`;
  - `week`.

  All the previous child nodes must have a non-negative integer value and at least one of them must be higher than zero. Using this node allows the Timer Service to be defined with a very precise value without having to convert the desired time to one of the interval units values allowed by the `IntervalUnit` value.

  For example, if the values of child nodes `hour`, `minute` and `second` are 1, 30 and 15, respectively, the Timer Service will trigger the time elapsed event exactly 1h30m15s after it had been started.

- `DateStart` — The date time value used to start counting the time.

  This node value is useful to tell the Timer Service when it should start counting the time. This node is not mandatory and if not defined the Timer Service only starts counting time when it is explicitly required to do so. Otherwise, if this node exists, the Timer Service automatically starts counting time at the value specified by the node.

- `DateEnd` — The date time value used to stop counting the time.

  This node value is useful to tell the Timer Service when it should stop counting the time. This node is not mandatory and if not defined the Timer Service only stops counting time when it is explicitly required to do so or when the time specified in interval node is reached (considering `repeat` flag equal to *false*). Otherwise, if this node exists, the Timer Service automatically stops counting time at the value specified by the node.

  If specified, the value of this node must be both higher than the current time and higher than the value (if specified) of `DateStart` node.

The following XML code shows an example of the timer structure that should be used. It contains two Timer Services: *Timer 1* and *Timer 2*. The first timer uses the `IntervalUnit` node to define a simple interval unit and timer 2 uses the `ComplexInterval` node to define multiple interval units.

```
<timers>
    <timer>
        <name>Timer 1</name>
        <repeat>...</repeat>
        <interval>...</interval>
        <IntervalUnit>...</IntervalUnit>
        <DateStart>...</DateStart>
        <DateEnd>...</DateEnd>
    </timer>
    <timer>
        <name>Timer 2</name>
        <repeat>...</repeat>
        <ComplexInterval>
            <millisecond>...</millisecond>
            <second>...</second>
            <minute>...</minute>
            <hour>...</hour>
            <day>...</day>
            <week>...</week>
        </ComplexInterval>
        <DateStart>...</DateStart>
        <DateEnd>.../DateEnd>
    </timer>
</timers>
```

Appendix C

AOI Integration Test Cases

This appendix presents the integration test cases related to the AOI equipment integration use cases presented in Chapter 5, most specifically in section 5.7. For each integration test case presented in the following sections, both a description and the sequence of steps required to execute each test are described.

C.1 Process XML Files

Table C.1: Process XML files — Test case description

Test ID	Test T1
Test name	Process XML files
Test description	This test verifies if the *AOIEquipmentModule* inserts all necessary records into customized database (raw and summary data)
Test date	29^{th} May 2008
Pre-conditions	TNS connection properly configured and targeted to desired database. Prototype must be running and data collection from AOI equipment must be enabled. SQL*Loader definitions must exist in the INI Table. SQL*Loader header files must exist. All directories for processed files and SQL*Loader temp files must exist.
Expected results	Raw and Summary records exist in target database. XML files detected must be moved from the listening directory to the directory of processed files after the processing.

Table C.2: Process XML files — Test case details

Sequence	**Step type**	**Step description**	**Expected results / state**	**Notes**
1	Action	Start the Bee Framework and begin data collection.	The application should have started.	See other preconditions on test case description.
2	Action	Trigger Start Lot event with Remote GUI.	Start Lot event triggered.	The *AOIEquipmentModule* must have received the Lot Start event.
3	Action	Ensure that AOI equipment XML files are created in the listening folder.	Equipment XML files are created in the input folder.	
4	Test	*AOIEquipmentModule* parses and processes XML files.	XML files are moved to processed directory.	A lot must be defined, otherwise the file will be ignored.
5	Test	Query the target database for processed lot number.	Raw and summary data exist in target database for that lot number.	
6	Test	Check if the XML files exist in the directory of processed files.	Files do not exist in the listening directory and exist in the directory used for already	

C.2 Notify XML File Generation Down

Table C.3: Notify XML file generation down — Test case description

Test ID	Test T2
Test name	Notify XML file generation down
Test description	This test verifies if the notification of XML File generation down is done successfully.
Test date	29^{th} May 2008
Pre-conditions	The parameters for specifying the recipients of the message and the message itself must be configured in the XML configuration file. Bee Timer Service settings used for notifications must be present in the XML configuration file. Listening folder must be defined in the XML configuration file. A Folder Watcher must be related to that listening folder. The prototype must be running and data collection from AOI equipment must be enabled. A lot must also be defined (notifications will not be sent if no lot is defined). SMTP service must be up and running. Remote GUI must be running.
Expected results	Each recipient should receive an e-mail message as configured previously. Remote GUI should be notified.

Table C.4: Notify XML file generation down — Test case details

Sequence	**Step type**	**Step description**	**Expected results / state**	**Notes**
1	Action	Start the Bee Framework and begin data collection.	The application should have started.	See preconditions in test case description.
2	Action	Trigger Start Lot event with Remote GUI	Start Lot Event triggered.	The AOI Equipment Module must have received the Lot Start event.

Continued on next page

Table C.4 – continued from previous page

Sequence	Step type	Step description	Expected results / state	Notes
3	Test	Configured delay without any XML file being created in listening folder was reached.	Remote GUI and target users e-mail should be notified.	

C.3 Backup AOI Log Files

Table C.5: Backup AOI log files — Test case description

Test ID	Test T3
Test name	Backup AOI log files
Test description	This test verifies if the AOI equipment log files are backed up.
Test date	29^{th} May 2008
Pre-conditions	AOI log files are created in the specified logging directory and logging type defined is "Rolling". Max number of files in the logging directory must be specified and either age or size units must have a value different from "None". The backup directory exists and must be correctly specified in the Logging Service definitions.
Expected results	AOI equipment log files were automatically moved from logging directory to logging backup directory.

Table C.6: Backup AOI log files — Test case details

Sequence	Step type	Step description	Expected results / state	Notes
1	Action	Configure the Logging Service XML node in the equipment module XML File in order to force a low number of maximum log files and with either low size or low age units.	Parameters changed.	See preconditions in test case description.

Continued on next page

Table C.6 – continued from previous page

Sequence	Step type	Step description	Expected results / state	Notes
2	Action	Load Bee Framework configurations and start AOI data collection.	Application up and running.	Application must perform some logs until the maximum number of log files allowed in the logging directory is reached.
3	Test	The next log message that exceeds file size or age limits will cause a rolling operation and the oldest log will be moved to the backup directory.	Files backed up.	

C.4 Log Equipment Breakdown Reason

Table C.7: Log equipment breakdown reason — Test case description

Test ID	Test T4
Test name	Log equipment breakdown reason
Test description	This test verifies if the equipment breakdown reason notification is available in OEDV tool.
Test date	29^{th} May 2008
Pre-conditions	The subject for incoming and outgoing messages must be properly configured in the INI Table for the AOI Equipment Module. LotEquipParamHistorySrv service must be up and running. The prototype must be running and data collection from AOI equipment must be enabled. A lot must also be defined (notifications will not be sent if no lot is defined). Remote GUI must be running.
Expected results	AOI equipment log files were automatically moved from logging directory to logging backup directory.

Table C.8: Log equipment breakdown reason — Test case details

Sequence	**Step type**	**Step description**	**Expected results / state**	**Notes**
1	Action	Start the Bee Framework and begin data collection.	The application should have started.	See preconditions in test case description.
2	Action	Send equipment breakdown reason message from Remote GUI.	The message sent is collected by the Bee Framework, handled by the AOI Equipment and sent to the LotEquipParamHistorySrv service.	
3	Test	Query the target database with OEDV tool.	The Equipment Breakdown Reason must be viewed by OEDV.	Usually, the target database is the EQC database too.

Appendix D

AOI Database Schema

The current appendix refers to the database schema used to save the data generated by the AOI equipments and collected from XML files. This appendix explains the database design model that holds the data collected from XML files that is inserted by the *AOIEquipmentModule* into the target database, so that it can be later used by external reporting tools.

The appendix only considers an high-level design, presenting both the required tables and their relationships without considering a description for all attributes. The existing mapping between the node tags from XML files and the attributes from database tables is not considered in this report. For further details, equipment's documentation should be consulted.

As referred in section 5.2 in terms of which data is present in each XML file generated by the equipment, the database schema must take in consideration the following statements:

- AOI equipment inspects multiple lots of modules;
- each module contains multiple panels;
- each XML file only contains data referring to a single panel of a lot;
- each panel contains multiple boards;
- and each board contains multiple location measurements.

Moreover, the XML file generated by an AOI equipment may result from a normal inspection or from a rework inspection, after correcting some of the errors detected by the first optical inspection.

Figure D.1 illustrates the overall database schema used to save the data collected from AOI equipments. This schema will be detailed in the following sections of this appendix.

D.1 AOI Control Table

This table is the entry point to the XML file loading. It generates a new *LotID* for each unique combination of *LotNumber*, *RecipeName*, *EquipmentID* and a *LotCounter*. This way the same SMT line can process both sides of the panels for one

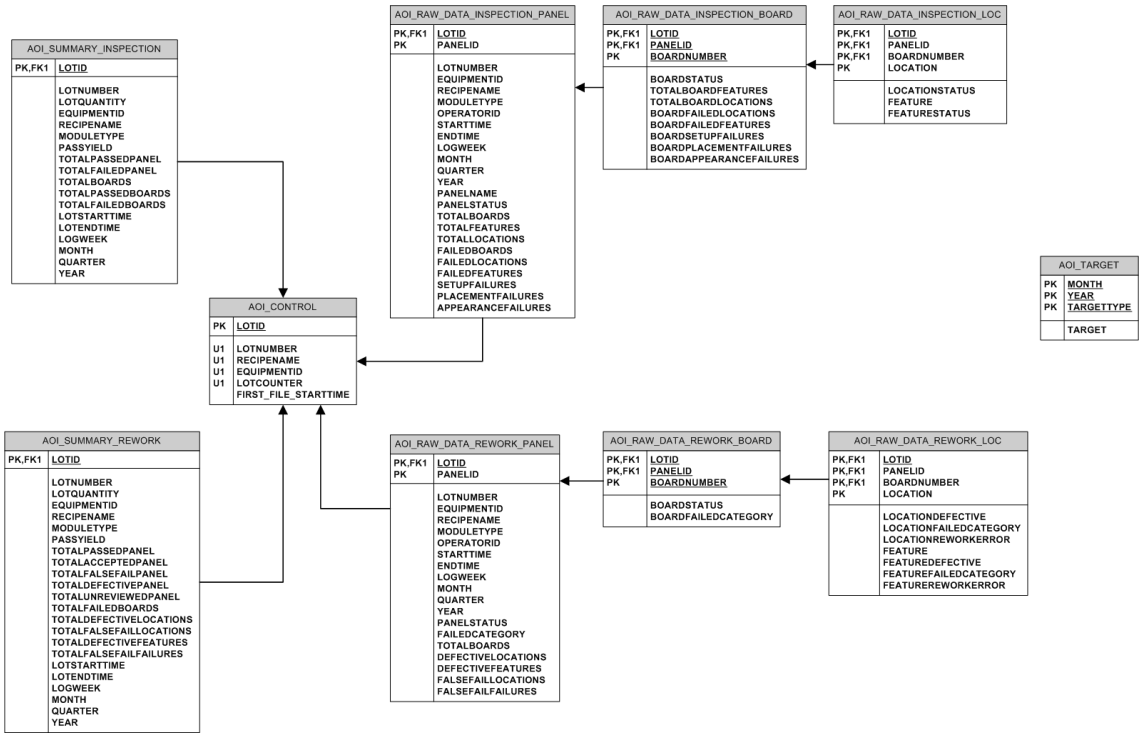


Figure D.1: AOI database schema

LotNumber without conflicting with primary key tables. Also, it would be possible to create foreign key constraints that will guarantee data integrity between all raw/summary tables with this one as the master table.

To avoid primary key violation when the same lot is inspected more than once with the same recipe and the same panel side (same recipe), the *LotCounter* column will guarantee the generation of a new *LotID* in order to allow multiple inspections for the same lot. This allows not only the data generated from the first inspection to be saved into the database but also to save the data resulting from further rework inspections if these are necessary.

Figure D.2 shows the *AOI_Control* table schema. This table serves as the master control table of the AOI database model.

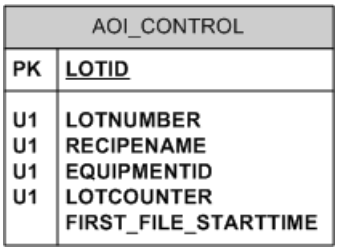


Figure D.2: AOI Control table

D.2 AOI Raw Data Tables

The raw data tables contain the records directly gathered from AOI XML equipment files. Each one of the generated XML files contains one panel information for one given *LotNumber*, identified by *PanelID*. A panel contains a group of boards, each one identified by the *BoardNumber*. Finally, each board contains a group of locations, each one identified by its own *Location* number.

The raw data tables also consider the two possible types of XML files generated by AOI equipment: inspection and rework types. Obviously, the main three level structure described in the previous paragraph remains the same. However, records information retrieved from XML files are different and consequently the collected data must also be stored using different raw data tables.

This three level combination results in the raw data tables schema for both inspection and rework types is illustrated in figure D.3, with foreign key integrity enabled constraints to ensure valid data.

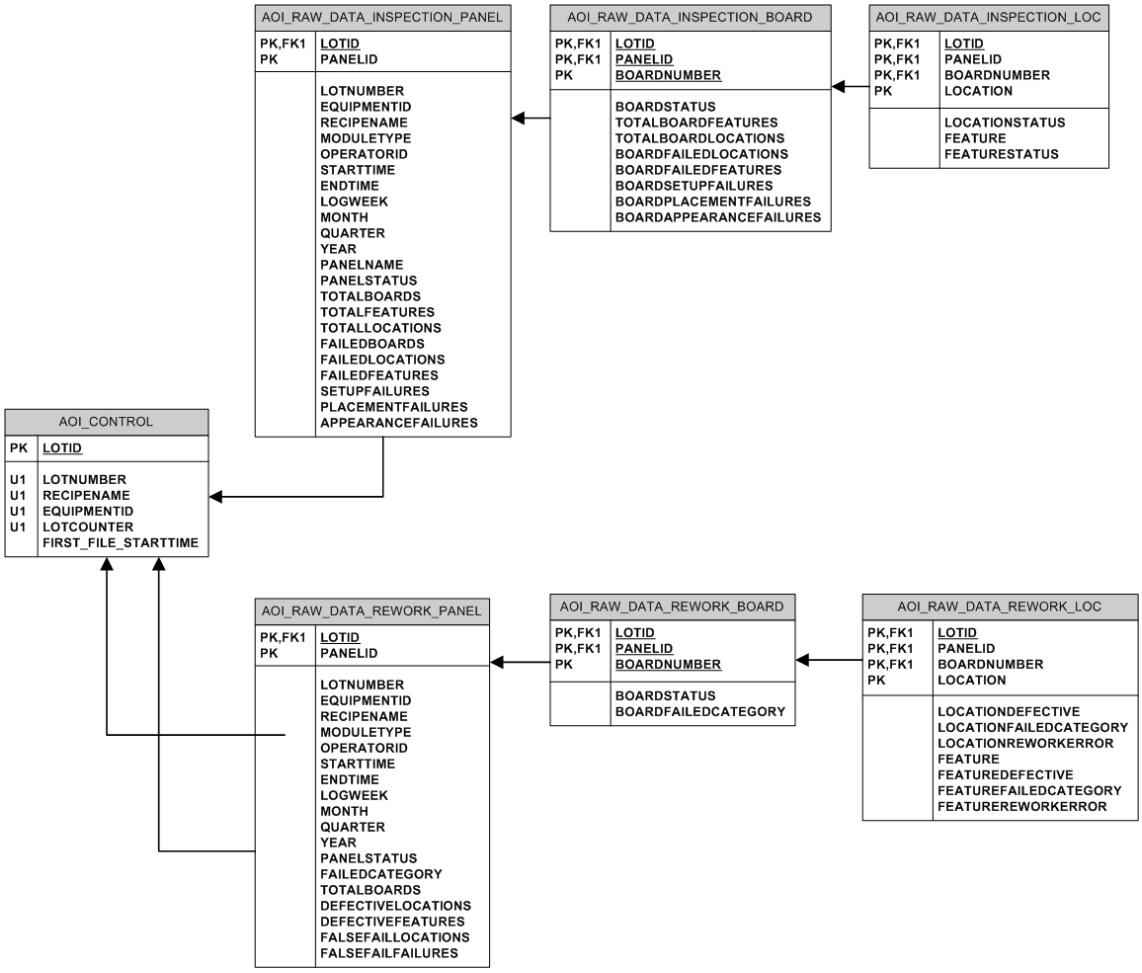


Figure D.3: AOI raw data tables

D.3 AOI Summary Tables

The summary tables represent an aggregation of the raw data tables. These summaries are *LotID* based and therefore can contain multiple inspections for the same *LotNumbers*. The schema illustrated in figure D.4 is also enabled with foreign key integrity constraints from the master table *AOI_Control*. Once again, just like for raw data tables, summaries must be differenced according the type of XML file (inspection or rework).

D.4 AOI Target Table

AOI_Target table is a table that was requested by end users. This table is not directly related to the data collection process itself and that is intended to be maintained only by end users.

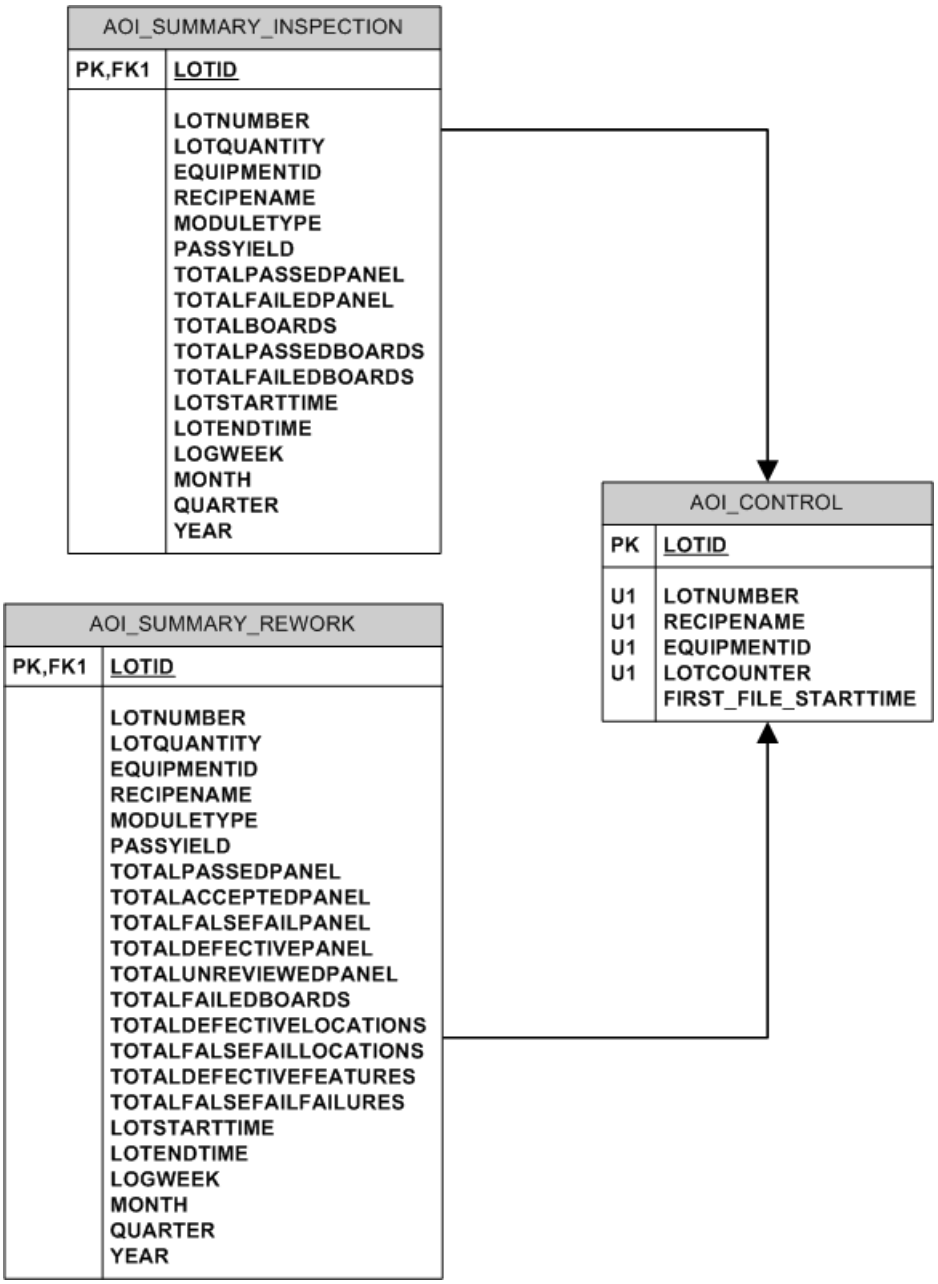


Figure D.4: AOI summary tables

Figure D.5: AOI Target table

Appendix E

AOI — SQL*Loader Usage

This appendix describes the usage of SQL*Loader tool used to load large volumes of data with higher efficiency and performance. It also refers to the required format of the header files used to load data.

Each utilization of SQL*Loader to load data into a database considers three different files, namely:

- control file.

  The control file is the main file used by the SQL*Loader tool. Such file is divided into two sections, the first one related to the settings used to load data and the second one related to the data itself.

- log file.

  When loading data, SQL*Loader tool generates a file used to log all information related to the loading process.

- "bad" file.

  Finally, the "bad" file is a file generated by SQL*Loader containing the data entries that could not be loaded into the database due to some error occurred during the loading process.

Whenever SQL*Loader is used to load data, it is important to check if the load was successful or if some errors have occurred during the whole process. Since large volumes of data are usually considered in such loading operations and it is infeasible to check the correctness of all database records, the errors during loading are detected by examining the files generated by SQL*Loader tool:

- log files are *always* generated, so if such file does not exist after a supposed loading operation is complete, then something went wrong and data has not been loaded into the target database;

- "bad" files only are generated if at least one error occurs during the loading process. If these files are not generated by SQL*Loader tool then it is a good sign: it means that no errors have occurred.

Consequently, the best way to ensure no errors have occurred during loading is to check the existence of the log file and the non existence of the "bad" file.

The following section explains the format of header files and presents the header files used to load the data collected from the XML files generated by AOI equipments.

E.1 SQL*Loader Header Files

This section presents the SQL*Loader header files used to save the data collected from the XML files generated by AOI equipments. These header files are used to define how data must be inserted into database tables.

As described previously in section 5.2 and also referred in appendix D, each XML file contains information related to a single panel from a specific lot. Consequently, it does not make sense to use a tool like SQL*Loader to insert a single row to associate the panel with the lot, either in *AOI_Raw_Data_Inpection_Panel* or in *AOI_Raw_Data_Rework_Panel*, depending if the XML file type is related to an inspection or a rework.

However, since each XML file contains a high number of measurements related to boards and locations that need to be saved, SQL*Loader can be very useful to reduce the amount of time required to save collected data into the target database. This data that needs to be loaded is appended at the end of the header specification and then the SQL*Loader tool uses the resulting file as an argument.

Each header specification contains the following information:

- the path of the file to load.

  The path of the file containing the header and the data is commonly specified by an asterisk in order to make the header settings global to all files. This asterisk allows the header to be used by all files and is replaced by each specific path at runtime by the SQL*Loader tool.

- the name of the database table.

  The name of the table is specified using the APPEND INTO TABLE code, followed by the name of the database table where data should be loaded.

- the character used as delimiter to delimit the data representing each table record.

  Since every row of data represents a record to insert into the database table specified in the header, a character must be used as a delimiter to delimit the data related to the multiple columns. In order to specify the delimiter, FIELDS TERMINATED BY code should be used.

- the order of data columns.

  Since there is no need to specify a SQL query for each insert statement, a match between the existing order of the data fields in the file used and the columns of the database table target must be specified. Consequently, when loading the data existing in the file to the target database table, SQL*Loader is able to match the fields and insert data into the database correctly.

The following subsections present the header files required to save data related to boards and locations for both inspection or rework file types.

E.1.1 Board Inspection Header

```
LOAD DATA
INFILE *
APPEND INTO TABLE AOI_RAW_DATA_INSPECTION_BOARD
FIELDS TERMINATED BY '|'
(LOTID, PANELID, BOARDNUMBER, BOARDSTATUS, TOTALBOARDFEATURES,
TOTALBOARDLOCATIONS, BOARDFAILEDLOCATIONS, BOARDFAILEDFEATURES,
BOARDSETUPFAILURES, BOARDPLACEMENTFAILURES, BOARDAPPEARANCEFAILURES)
BEGINDATA
```

E.1.2 Board Rework Header

```
LOAD DATA
INFILE *
APPEND INTO TABLE AOI_RAW_DATA_REWORK_BOARD
FIELDS TERMINATED BY '|'
(LOTID, PANELID, BOARDNUMBER, BOARDSTATUS, BOARDFAILEDCATEGORY)
BEGINDATA
```

E.1.3 Location Inspection Header

```
LOAD DATA
INFILE *
APPEND INTO TABLE AOI_RAW_DATA_INSPECTION_LOC
FIELDS TERMINATED BY '|'
(LOTID, PANELID, BOARDNUMBER, LOCATION, LOCATIONSTATUS, FEATURE, FEATURESTATUS)
BEGINDATA
```

E.1.4 Location Rework Header

```
LOAD DATA
INFILE *
APPEND INTO TABLE AOI_RAW_DATA_REWORK_LOC
FIELDS TERMINATED BY '|'
(LOTID, PANELID, BOARDNUMBER, LOCATION, LOCATIONDEFECTIVE,
LOCATIONFAILEDCATEGORY, LOCATIONREWORKERROR, FEATURE, FEATUREDEFECTIVE,
FEATUREFAILEDCATEGORY, FEATUREREWORKERROR)
BEGINDATA
```

Appendix F

AOI Configurations

This appendix presents the XML configuration files used in the AOI integration. Some of the XML nodes from files have already been explained in Appendix A and in Appendix B, so these nodes will not be explained again. The remaining XML nodes from these files will be detailed in the following sections.

F.1 Bee Framework Configuration File

The Bee Framework XML configuration file is the same given as an example in Appendix A.

F.2 Folder Monitor Module Configuration File

This section refers to the configurations related wit the *FolderMonitorModule* XML configuration file. This module is quite simple and just defines the messages the module should to subscribe. The following subsections describe the behavior of the module for both internal framework and YODA messages existing in the following XML code:

```
<?xml version="1.0" encoding="utf-8" ?>
<FolderMonitor>
  <messages>
    <message>CopyFile</message>
    <message>MoveFile</message>
    <message>LoadWatchers</message>
    <message>StartMonitoring</message>
    <message>ListFilesDirectory</message>
  </messages>

  <YodaMessages>
    <YodaMessage>CopyFile</YodaMessage>
    <YodaMessage>MoveFile</YodaMessage>
  </YodaMessages>

</FolderMonitor>
```

F.2.1 CopyFile Message

The *CopyFile* message has the purpose of copying a file from its source path to a destination path. The message behavior is identical for both internal framework or YODA messages. This message contains the following parameters:

- `SourcePath` — The full path of the file that should be copied.
- `DestinationPath` — The full path of the destination target.
- `Overwrite` — A boolean flag that specifies a file with the same name in the target directory should or not be overwritten.

F.2.2 MoveFile Message

The *MoveFile* message has the purpose of moving a file from its source to a destination path. Like the *CopyMessage*, the message behavior is identical for both internal framework or YODA messages. This message parameters are also the same as the previous message, but it does not consider the *Overwrite* parameter.

F.2.3 LoadWatchers Message

The *LoadWatchers* message has the purpose of configuring and preparing the list of *FolderWatchers* of the *FolderMonitorModule*, as described in section 4.3.1. This message only has a parameter: the `Watchers` parameter. These parameter is a XML node, which may contain multiple child nodes, each one referring to a watcher with the following configuration:

- `name` — The name of the *FolderWatcher*. It must allow the *FolderMonitorModule* to uniquely identify a *FolderWatcher*.
- `path` — The full path of the directory being monitored.
- `subdirectories` — A boolean flag specifying if the subdirectories of the directory being monitored should also be included in the monitoring.

The following XML code shows an example of the `watchers` configuration:

```
<watchers>
    <watcher>
        <name>AOI Watcher</name>
        <path>C:\UserData\classes\Working dir\</path>
        <subdirectories>False</subdirectories>
    </watcher>
    <watcher>
        <name>AOI Log Watcher</name>
        <path>C:\UserData\Bee\Framework\Logs\</path>
        <subdirectories>False</subdirectories>
    </watcher>
</watchers>
```

F.2.4 StartMonitoring Message

The *StartMonitoring* message is used to tell the *FolderMonitorModule* that it should start monitoring the directories specified in its list of *FolderWatchers* previously configured with the *LoadWatchers* message.

F.2.5 ListFilesDirectory Message

The *ListFilesDirectory* message is used get the list of existing files inside a directory. This message may contain the following parameters:

- `Directory` — The full path of the desired directory.
- `Filter` — The filter to use as a regular expression.

  Using a filter, only the files that match the regular expression specified are retrieved. This parameter is not mandatory and if not present, all file names will be retrieved, not taking in consideration the file name.
- `Locked` — A boolean flag used check if each file from the desired directory is or not locked.

  If this flag is set to *true*, each file is checked to see if it is being used by other process. If this flag is set to *true* and a file is being used by other process, this filename will not be listed as a result. Otherwise, if the flag is set to *false*, all the files matching the regular expression used as filter will be listed in the result, whether they are in use or not by other process.

F.3 AOI Equipment Module Configurations

This section describes the *AOIEquipmentModule* configurations, including the messages subscribed and other settings related to services and the equipment itself. The complete XML code is in the end of this section.

F.3.1 CommandLot Message

The *CommandLot* message is a message received from the YODA network and is related to a command executed by operators in the AOI equipment Remote GUI. Such message contains the following main parameters:

- `EquipmentName` — The name of the equipment on which the command has been executed through the Remote GUI.
- `IFX_SERVICE` — The service that identifies the command executed by operators. It may have the following values:
  - `DefineLot` — The message contains the required parameters to define the settings of the new lot that will inspected by the AOI equipment.
  - `EndLot` — The *CommandLot* message contains the required parameters to end the current lot being inspected by the AOI equipment.

- `GetCurrentLotInformation` — The message contains the parameters that uniquely identify the lot being processed, so that full lot information can be retrieved.
- `LogBreakDown` — The message contains the required parameters to log a message related to a breakdown.

F.3.2 *Created.AOI Watcher* and *Created.AOI Log Watcher* Messages

Both the *Created.AOI Watcher* and *Created.AOI Log Watcher* messages are related to notifications sent by the *FolderMonitorModule*. These messages are the result of the *FolderWatchers* monitoring and are created either when a new XML file is generated by the AOI equipment or when a log file is moved to the backup directory.

Both these messages contain the following parameters:

- `FullPath` — The full path of the file created.
- `Name` — The short name of the file created.

XML File

The following XML code shows the complete configuration file used by the *AOIEquipmentModule*.

```
<?xml version="1.0" encoding="utf-8" ?>
<AOIModule>

    <name>MDAAOI-0001</name>
    <ApplicationName>AOI_DBLoader</ApplicationName>

    <modules>
        <module>
            <name>Folder Monitor</name>
            <watchers>
                <watcher>
                    <name>AOI Watcher</name>
                    <path>C:\UserData\classes\Working dir\</path>
                    <subdirectories>False</subdirectories>
                </watcher>
                <watcher>
                    <name>AOI Log Watcher</name>
                    <path>C:\UserData\Bee\Framework\Logs\</path>
                    <subdirectories>False</subdirectories>
                </watcher>
            </watchers>
        </module>
    </modules>

    <services>
         <timers>
            <timer>
                <name>Generation Down Timer</name>
                <interval>360</interval>
                <IntervalUnit>Second</IntervalUnit>
                <repeat>True</repeat>
            </timer>
        </timers>
```

```
        <database>
            <DateFormat>MM/DD/YYYY HH24:MI:SS</DateFormat>
            <connection>AOI Connection String</connection>

            <SqlLoader>
                <headers>
                    <header name="BoardInspection">
                                SQLLoaderHeaderFile4BoardInspection</header>
                    <header name="BoardRework">
                                SQLLoaderHeaderFile4BoardRework</header>
                    <header name="LocationInspection">
                                SQLLoaderHeaderFile4LocationInspection</header>
                    <header name="LocationRework">
                                SQLLoaderHeaderFile4LocationRework</header>
                </headers>
            </SqlLoader>

        </database>

        <loggings>
            <DefaultLogging>Log1</DefaultLogging>
            <log>
                <name>Log1</name>
                <filename>C:\UserData\Bee\Framework\log.txt</filename>
                <type>Rolling</type>
                <header></header>
                <footer></footer>
                <SizeUnit>Kilobytes</SizeUnit>
                <size>1</size>
                <AgeUnit>None</AgeUnit>
                <age>0</age>
                <MaxNumberFiles>10</MaxNumberFiles>
                <BackupDirectory>C:\UserData\Bee\Framework\Logs\</BackupDirectory>
            </log>
        </loggings>

    </services>

    <folders>
        <OkFolder>C:\UserData\AOI\OK Folder</OkFolder>
        <IgnoreFolder>C:\UserData\AOI\Ignore Folder</IgnoreFolder>
        <ErrorFolder>C:\UserData\AOI\Error Folder</ErrorFolder>
    </folders>

    <messages>
        <message>Created.AOI Watcher</message>
        <message>Created.AOI Log Watcher</message>
    </messages>

    <YodaMessages>
        <YodaMessage>CommandLot</YodaMessage>
    </YodaMessages>

</AOIModule>
```

Beyond the nodes used to define which internal framework and YODA message subjects are subscribed, this configuration file also contains the following main nodes:

- `name` — The name of the AOI equipment.

- `ApplicationName` — The application name defined in the INI Table.
- `modules` — This node name is used just to help understanding the configurations of the XML file.

  It has a reference for the name of the framework module that contains the *FolderWatchers* required by the *AOIEquipmentModule*. These *FolderWatchers* are configured so that the *AOIEquipmentModule* can be notified when a new XML or a log file in the backup directory becomes eligible.
- `services` — This node contains the information related to the Timer and Logging Services required by the *AOIEquipmentModule*.

  Additionally, it also contains some definitions related to the header files used by SQL*Loader tool.

  - Timer Service — The configurations used by the Timer Service that allows the module to send notifications if the equipment is not generating XML files. For further details related to this service, please consult Appendix B.
  - Logging Service — The configurations used by the Logging Service that allows the *AOIEquipmentModule* to log messages related to the data collection process concerning the AOI equipment.
  - Database — This node contains the name of the connection string that corresponds to the database used by the *AOIEquipmentModule* to save the collected data. It also includes the format to be used by *datetime* fields. Finally, this node contains the name of the SQL*Loader wrappers, so that the corresponding loading operations using this tool can be configured by the database service.
- `folders` — The node that contains the full paths of the folders used to save the XML files depending on the success of the parsing applied to the XML files generated by the AOI equipment. It considers the following folders:

  - `OkFolder` — The path of the folder used to save the XML files if the parsing and the loading of the data into the database succeeds without any error.
  - `IgnoreFolder` — The path of the folder used to save the XML files if no lot information has been previously defined. The XML file is ignored in such situation.
  - `ErrorFolder` — The path of the folder used to save the XML files if the parsing or the loading of the data into the database is not successful.