unibz

Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Fakultät für Informatik
Facoltà di Scienze e Tecnologie informatiche
Faculty of Computer Sciences

**Bachelor in Computer Science and Engineering**

**Bachelor Thesis**

# Automated Testing in a Microsoft Dynamics 365 Environment

**Candidate**: Samuele Ceol

**Supervisor**: Barbara Russo

March 2019

# Acknowledgement

Un grazie speciale alla mia famiglia, a mia madre e i miei fratelli per essermi sempre stati vicni ed avermi supportato (e sopportato) in tutte le mie scelte. Un grazie a Betta e Kleo, ai miei amici dentro e fuori Bolzano e a tutti i compagni con i quali ho condiviso anche solo una piccola parte di questi tre anni. Infine un ultimo ringraziamento va alla mia relatrice, Barbara Russo, a Thomas Lorenzi e a tutto il team Dynamics in Würth Phoenix per avermi guidato durante tutto il periodo di tirocinio e avermi permesso di avere una prima vera esperienza lavorativa come sviluppatore.

# Abstract

The size of modern software applications often requires particular focus to be placed on various testing techniques in order to ensure that high standards of quality are respected and to prevent the release of code containing possible defects or presenting unwanted behaviors. In the case of Enterprise Resource Planning (ERP) the employment of said practices is of utmost importance given the dramatic impact the failure of such a system can have on a company. ERP applications represent the heart of many modern businesses that use this kind of tools to complete a wide array of core activities necessary for the company success and survival. Examples of such activities are: Finance and accounting, production, distribution, sales, business intelligence, customer services, human resources management. It is easy to understand how an issue, even in a single one of those modules, could result in a huge loss of a company time and internal resources. For a team of developers successfully implementing testing related techniques often means finding a balance between having a fair amount of test coverage and respecting the time constraint imposed by the client needs for a specific project. We work on a vertical solution (called Trade+), developed by Würth Phoenix and based on Microsoft Dynamics 365 for Finance and Operations. The intent of the project is to cover a sample of defined core processes with a series of test cases aimed at ensuring the expected operativity of some aspects of the application in range of different scenarios. We start by using tools provided within the application to translate high-level interactions with the system into reusable task recordings. This interactions represent the analyzed business process and form the basis for the developed tests. We then convert these recordings into coded test cases that are expanded with tailored code specifically created to verify the behavior of the system when exposed to an array of different condition and edge cases. Finally, we integrate our test cases with the nightly builds in order to apply regression testing techniques on the subset of functionalities that we have covered.

# Zusammenfassung

Die Größe moderner Softwareanwendungen erfordert oft ein besonderes Augenmerk auf verschiedene Testtechniken, um sicherzustellen, dass hohe Qualitätsstandards eingehalten werden und um die Freigabe eines Codes mit möglichen Fehlern oder unerwünschtem Verhalten zu verhindern. Im Falle von Enterprise Resource Planning (ERP) ist der Einsatz dieser Praktiken von größter Bedeutung, da der Ausfall eines solchen Systems dramatische Auswirkungen auf ein Unternehmen haben kann. ERP-Anwendungen stellen das Herzstück vieler moderner Unternehmen dar, die diese Art von Tools einsetzen, um eine breite Palette von Kernaktivitäten abzuschließen, die für den Erfolg und das Überleben des Unternehmens notwendig sind. Beispiele für solche Aktivitäten sind: Finanz- und Rechnungswesen, Produktion, Distribution, Vertrieb, Business Intelligence, Kundenservice, Personalmanagement. Es ist leicht zu verstehen, wie ein Problem, selbst in einem einzelnen dieser Module, zu einem enormen Verlust von Zeit und internen Ressourcen eines Unternehmens führen kann. Für ein Team von Entwicklern gilt es, für die Implementierung von testbezogenen Techniken in einem bestimmten Projekt, ein Gleichgewicht zwischen einer guten Testabdeckung und der Einhaltung der vorgegebenen Zeitvorgaben des Kunden zu finden. Wir arbeiten an einer vertikalen Lösung (Trade+), die von Würth Phoenix entwickelt wurde und auf "Microsoft Dynamics 365 for Finance and Operations"basiert. Ziel des Projekts ist es, eine Stichprobe definierter Kernprozesse mit einer Reihe von Testfällen abzudecken, um die erwartete Funktionsfähigkeit einiger Aspekte der Anwendung in verschiedenen Szenarien sicherzustellen. Zunächst verwenden wir bereitgestellten Tools, um hochrangige Interaktionen mit dem System in wiederverwendbare Task Recordings zu übersetzen. Diese Interaktionen stellen den analysierten Geschäftsprozess dar und bilden die Grundlage der entwickelten Tests. Diese Aufzeichnungen werden in codierte Testfälle umgewandelt und mit einem dafür spezifisch entwickeltem Code erweitert werden, um das Verhalten des Systems zu überprüfen wenn es einer Reihe von verschiedenen Zustands- und Kantenfällen ausgesetzt ist. Schließlich integrieren wir unsere Testfälle mit den nächtlichen "Builds"(nightlies), um Regressionstests auf die Teilmenge der Funktionalitäten anzuwenden, die abgedeckt wurden.

# Sommario

Le dimensioni delle moderne applicazioni software portano spesso a dover porre particolare enfasi riguardo le varie tecniche di testing al fine di garantire il rispetto di elevati standard di qualità e di evitare il rilascio di codice contenente possibili difetti o comportamenti indesiderati. Nel caso dell'Enterprise Resource Planning (ERP) l'impiego di tali pratiche è della massima importanza dato il drammatico impatto che il fallimento di un'tale sistema può avere su un'azienda. Le applicazioni ERP rappresentano il cuore di molti business moderni che utilizzano questo tipo di programmi per completare una vasta gamma di attività fondamentali per il loro successo e la loro sopravvivenza. Esempi di tali attività sono: Finanza e contabilità, produzione, distribuzione, vendite, business intelligence, servizio clienti, gestione delle risorse umane. Risulta quindi semplice comprendere come un problema, anche in un singolo di questi moduli, possa comportare un'enorme perdita di tempo e di risorse per l'azienda interessata. Per un team di sviluppatori implementare con successo le varie tecniche di testing significa spesso trovare un equilibrio tra l'ottenimento di una discreta copertura di test e il rispetto dei vincoli temporali imposti dalle esigenze del cliente per uno specifico progetto. Lavoriamo su una soluzione verticale (denominata Trade+), sviluppata da Würth Phoenix e basata su Microsoft Dynamics 365 for Finance and Operations. L'intento del progetto è quello di coprire un campione di processi "core" con una serie di test case finalizzati a garantire l'operatività di alcuni aspetti dell'applicazione in diversi scenari. Iniziamo utilizzando gli strumenti forniti all'interno dell'applicazione per tradurre interazioni di alto livello con il sistema in task recordings riutilizzabili. Queste interazioni rappresentano i processi di business analizzati e costituiscono la base dei test case sviluppati. Le registrazioni vengono quindi convertite in test codificati che sono poi espansi con codice personalizzato creato appositamente per verificare il comportamento del sistema quando viene esposto a una serie di condizioni e casi limite specifici. Infine, integriamo i nostri test case con le build notturne (nightlies) al fine di applicare tecniche di regression testing sul sottoinsieme di funzionalità che abbiamo coperto.

# Contents

# List of Figures



# List of Tables

# Chapter 1

# Introduction

## 1.1. Background and Motivation

### 1.1.1. Company (Background)

Würth Phoenix is a software company part of the Würth-Group specialized in the creation of ERP, CRM and IT System Management tools. The company is a certified Dynamics Gold Partner and works in close contact with Microsoft to develop and deploy state of the art solutions capable of meeting the clients requirements. The Dynamics AX department employs more than 70 certified consultants, system engineers and project managers specialized in the creation of wholesale distribution software. One of the flagship products developed in this department is Trade+. Trade+ is a vertical solution based on Dynamics 365 for Finance and Operations that offers additional functionalities built on top of the pre-existing Microsoft software that are specifically tailored to the needs of modern wholesale distributors. The application is the result of many years of experience in the field and comes from the long collaboration between Würth Phoenix and local and international companies that operate in the industry.

### 1.1.2. Internship Project (Motivation)

The latest iteration of the Microsoft Dynamics tool (D365) introduced a wide array of new features and represents one of the biggest evolution of this software in recent years. This shift in the industry standard makes it often difficult for developers to create reliable code that complies with the client needs while still maintaining an appropriate level of quality and reliability. Given the relative novelty of this update and the fact that the framework relies entirely on closed source code, finding good sources of documentation proves itself to be a particularly complex task. Most of the times (in the current stage of the tool lifecycle) said documentation does not exist or is often particularly lackluster and fails to deliver any form of in-depth information. Because of this reason, the motivation behind this project (and the work related to it) originates from the need of the company of having some form of practical know-how (i.e. documentation) regarding the creation and implementation of test cases in the latest version of Dynamics 365 for Finance and Operations. Furthermore, testing some core features of the application ensures the operativity of said business processes during the whole development process.

## 1.2. Objective

The goal of this project is to create a set of test cases covering some of the core functionalities of the interested application (Trade+) and integrating them with the nightly builds. A complementary objective is to document the process in order for the company to have some sort of guidance regarding schedules, complexity and resources needed to obtain useful testing results. This includes

not only the actual development process (i.e. coding) and all the activities related to it, but also procedures such as (but not limited to): identification of core processes, recording of high level tasks within the application, build integration, training, etc. Furthermore, this work tries to present the difficulties encountered during the various stages of the process together with the approaches that had to be taken in order to overcome them. Particular attention is also placed in the motivation behind the various decisions taken during the whole duration of the project.

## 1.3. Initial approach

We decided to organize the first approach to our work into two main activities that had to be carried out before establishing further details regarding the structure of the project. Said activities were:

- *Problem definition.* A series of initial (Kickoff) meetings were carried out in order to find a balance between the needs of the company and the overall skills required to complete the assigned work. Particularly important during this first phase was the definition of a "core" requirement that had to be initially given absolute priority and of some "secondary" tasks that could eventually be resolved once the main portion of the project was completed. During this stage a timetable containing both the various project activities and results was also created (section 1.4.2). We also decided to organize periodical steering committees at the end of each major phase of our work. This committees were reunions organized to monitor the progress, review the content of the work, exchange feedback and adjust the scope of the project depending on the current requirements.
- *Planning of Training activities.* Together with the previously described tasks an initial training period was organized in order to form some basic knowledge regarding the covered topics, used technologies and company policies. We decided to divide training activities into face-to-face meetings and online lectures. The meetings covered an array of company-related subjects and were focused on the various functionalities of Trade+ (Wholesale and distribution, Retail and Production). The online lectures were provided by the Microsoft Dynamics Learning Portal and offered an in-depth analysis of the Dynamics 365 application, X++ programming language, Visual Studio development environment and Azure DevOps automation tool.

Only when the company considered these first activities completed the attention gradually shifted to the next phases of the project.

## 1.4. Method of work

We decided to structure our work in the same way in which larger project are handled inside the company on a day to day basis. This meant basing the organization of the various stages of the project following the guidelines imposed by Microsoft SureStep.

### 1.4.1. Microsoft SureStep

Microsoft SureStep [1] is a full life cycle methodology that helps partners develop, configure and implement Dynamics related solutions. The goal is to improve productivity and success rate while reducing costs and risk factors. This methodology also guarantees an higher degree of predictability during the implementation process and an increased involvement of the stakeholders. Furthermore, guidelines for both waterfall and iterative approaches are provided in order to successfully develop and deliver different types of solutions. SureStep recognizes five distinct phases that can be interlinked in various ways in order to support different kinds of projects. Said phases are:

- *Diagnostic.* Activities related to the initialization of the project. An high level description and analysis of the customer processes is formulated together with a first draft of the project timeline where scope and approaches are initially defined.

- *Analysis.* High level definition and documentation of the business processes are formulated together with the functional requirements. The training activities are also carried out during this phase.

- *Design.* Identification and planning of the implementation strategies and technical specifications together with data migration activities.

- *Development.* Implementation of new functionalities and adaptation of existing ones to satisfy the customer needs. Testing activities.

- *Deployment.* Setup of the functioning environment to the customer and system level testing. The deployment stage ends with the Go-Live.

- *Operation.* All activities related to a post Go-Live system (product support, project review, bug fixing, account management).

Another important activity of this methodology is the definition of team roles. Said role are divided between *consulting* (project manager, business analyst, solution architect, application/development/technology consultant) and *customer* (executive sponsor, purchase manager, IT manager, end/key user). SureStep provides a general definition of the tasks and skill required to cover each role.

### 1.4.2. Structure

Given the relatively small scope of the project described in this work we decided to apply the principles and phases of SureStep but to ignore (or only partially pursue) certain specific activities that were typical of larger project and not applicable in this context. The *Diagnostic* phase (already partially defined in section 1.3) started with the previously defined Kickoff meetings that allowed us to define a first version of the project timeline which contained the expected activities and related results of the various phases that had to be part of the project [Figure 1.1]. During this first period the work environment was also set up and the training activities were planned. In the *Analysis* phase we identified the specific test requirements of the application by analyzing its core features. The result was the definition of the required test cases that (we decided) had to cover the following core business processes: Purchase order, transfer order, sales order, production order and return order. The training activities were also carried out during this period. After this initial definition we started working (in the *Design* phase) directly on the application by utilizing an integrated tool, called task recorder, that allowed us to record high level (step-by-step) user interaction with the system. Said interactions were saved as XML files that could be then imported on any machine running Trade+ in order to reproduce the recorded processes. At the end of this activity we ended up with a set of recordings covering the previously described business processes. A built-in feature of the Visual Studio development environment allowed us to convert the task recordings into X++ coded test cases. This represented the starting point for the *Development* phase. Once created, the test cases were expanded with tailored code specifically written to verify the expected behavior of the application under a wide array of different edge cases and scenarios that could exist within the covered processes. The creation of test cases "ex novo" (not starting from a pre-existing recording) was also taken into consideration as a secondary requirement but later ignored given the level of expertise and knowledge of the system that such a task would have required. This last period of was also dedicated to the integration of the developed test in the CI workflow (i.e. nightly builds). In the context of this work, the *Deployment* phase was reduced to a series of final meeting organized in order to communicate the project results and to hand-in the documentation containing the acquired know-how. All the further testing related work that will be performed on the application in the future will be part of the *Operation* phase.

| **Phase (with calendar weeks)** | **Activity** | **Result** |
|---|---|---|
| **Diagnostic (40)** | Deadlines and Objectives definition<br>Environment Setup<br>Planning of training activities | Project timetable<br>Working Dev. Environment<br>List of training activities |
| **Analysis (41-42)** | Test requirements identification<br>Training | List of covered business processes<br>Application knowledge |
| **Design (43-45)** | Business processes recording | Set of recording files |
| **Development (46-52)** | Test cases creation<br>Test cases expansion<br>Build integration | Set of generated test cases<br>Set of expanded test cases<br>Test cases in nightly builds |
| **Deployment (1-3)** | Know-how sharing | Documentation |
| **Operation** | Future work | |

Table 1.1: Project Timeline

# Chapter 2

# Project Requirements

The result of the Diagnostic phase was the definition (with the previously mentioned Project Timeline) of the various results that needed to be achieved before the end of the internship period. This definition was further refined in the Analysis phase where a list of the covered core processes of Trade+ was created, together with their testing requirements. The test cases not only needed to verify that the tested features were able to function under a range of different scenarios, but also that the expected error or warning message was returned in the case of an unforeseen interaction with the system. The (initial) minimum requirement consisted in checking that the application could successfully complete all the tested processes under "normal" conditions and in isolation (*component/feature testing*). This meant: creating the XML recordings, converting them into X++ code, adding code to verify that the appropriate changes were performed on the database at the end of each process and obtaining a successful outcome after the run each test. A second requirement consisted in testing the application for edge cases and unexpected behaviors and verifying the correct response of the system during this particular scenarios. The purpose of this activities was not only to check if Trade+ was able to handle "non-convectional" situations, but also to verify that appropriate and informative messages were prompted to the end user in order to successfully complete a given activity. To achieve this, each process had to be broken down into its individual sub-activities, each of which needed to be studied in order to understand which kind of interactions could lead to potential issues. The text cases were then expanded accordingly, following the need of this second requirement. Finally, as a last testing related requirement, we verified to which degree the different processes were able to correctly interact with each other by linking them with one another and creating a single flow (*integration testing*) of actions that covered the purchase and production of goods for a specific location, the transfer of said items to another warehouse, the creation of a sales order to a specific customer and (at the end) the return of the goods to the selling warehouse. As mentioned before, a secondary objective (part of the Development phase) was the integration of the developed test cases into the nightly builds (*regression testing*). Said integration had to be started after the completion of the test cases development and was divided into three stages: Acquisition of the technical knowledge needed to start the integration process, application of the acquired knowledge on a sample test to verify the correct functioning of all the component needed to complete this process and, as a final step, integration of all the previously developed test cases. Finally, as part of the Deployment phase, we created the documentation needed to pass the knowledge of this experience to the company through a series of final meetings during which this acquired know-how was presented.

# Chapter 3

# Related Work

In this section we explore some tools and techniques related to the activities described in our work. We have decided to divide this chapter into three sections: *Product*, *Process* and *Automation*. Product describes *Jenkins*, an alternative solution to the utilized automation tool (Team Foundation Server/Azure DevOps). Jenkins is one of the most popular applications for the implementation of continuous integration,delivery and deployment techniques and it constitutes an interesting object of study given its open source nature as opposed to the closed source approach taken by Microsoft with Azure DevOps. The completely customizable extensibility of Jenkins with the use of plugins is another characteristic that further motivated this initial comparison. Process elaborates on the concept of *Behavior-Driven Development*, and on a possible tool (Cucumber) to employ said technique. We have decided to introduce the notion of BDD in order to describe a viable approach for the company to develop future applications with a major focus not only on the concept of testing, but also on the communication between developers, consultants and key users. This technique, while not directly utilized during the course of this project, may represents a key step for the improvement of the quality and quantity of testing activities applied during a project development. Because of this reason, we believed it to be worthy of description. Finally, Automation gives some background information regarding the notion of *Continuous Integration*. We decided to provide a brief introduction in this regard because of the fact that CI (and the related concepts of Continuous Deployment and Delivery) constitutes the basis for the employment of regression testing techniques which represent one of the final objectives of this work.

## 3.1. Product

Jenkins [2] is a java-based, open-source software originally created to automate the building and testing stages of the development process. The extreme flexibility of the tool made it adaptable to a very wide range of requirements such as version control, code quality monitoring, testing, build automation, etc. The tool can be extended with the thousands of available open source plugins that that allow the implementation (and automation) of various DevOps tools and techniques. In this context Jenkins is seen as an *orchestrator* tool. Given its flexibility, Jenkins is primarily used to support Continuous Integration and Continuous Delivery approaches. Continuous Deployment is also achievable but often times more conservative procedures (such as a manual one-click deployment) are preferred by companies in order to have direct control over the process. A suite of related plugins that enables the use of CI/CD techniques is called a Jenkins Pipeline [3]. Said pipelines [Figure 3.1] are fundamentally a series of repeatable interlinked steps that may allow changes in the code to be automatically deployed. This chain of operations provides various benefits such as: Grouping of multiple activities into clearly defined stages (if one of such stages fails, the process is interrupted), clear visibility of all aspects of the process and immediate feedback when problems arise. Jenkins pipelines are composed by two kind of elements: *stages* and *steps*. A step is a minimal operation that tells the software to perform an action (like executing a script). A stage is a group of logically related

sub-tasks (like building, testing, etc.). A machine executing a pipeline is called a *Node*. Pipelines can be written directly in Jenkins or can be created in text files called *JenkinsFile* where the single steps are defined. Benefits of using said files include prevention of data loss (pipelines definitions are stored in the repository), code quality improvement (through code reviews) and security (access restrictions are the same as the ones used for the source code).

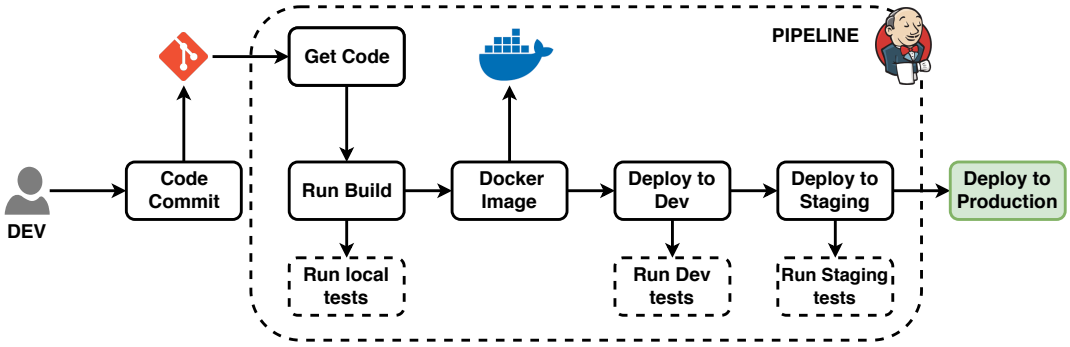


Figure 3.1: Example of a Jenkins Pipeline

We will later see how a similar approach to pipelines (in the context of an Azure DevOps environment) will be used in order to enforce regression testing techniques on the Trade+ application.

## 3.2. Process

Behavior-Driven Development [4] is a set of practices aimed at delivering high quality software while maintaining a stable stream of communication between developers, stakeholders and users. BDD tries to resolve the issues of Test-Driven Development and Acceptance Test-Driven Development by providing a better way for developers to understand scope and starting point of the testing activities that have to be performed on the application. In the context of BDD, system specification are defined in a way that provides an high degree of understandability and automation capabilities. In order to make this possible a common (*ubiquitous*) language is used (and understood) by all the parties involved in the development process. BDD is started with the identification of the expected (or desired) behaviors of the application, which are then translated into a set of features. From this features, user stories (that describe the interaction between user and system) are created together with their related scenarios (specific instances of a user story). The language used to present this scenarios allows to turn their (high level) description into executable specifications that will act as *acceptance tests* that verify the behavior of the system (behavioral testing). This process is made possible by mapping each step of a scenario to specific portion of code that can then be executed as part of the acceptance test. BDD also focuses on the readability of code which is seen as part of the project documentation. Because of this reason classes and methods names are usually self explanatory and allow the reader to understand what a specific portion of the code does without the need of additional information (this characteristic also affects the name and structure of unit tests). Being an highly collaborative procedure (in which information regarding the system specifications are shared in plain English, or any other language of choice), BDD requires a way to define feature and scenarios while still maintaining a sufficient level of automation. A tool commonly used to implement such procedures is *Cucumber* [5]. Cucumber is a command-line tool that takes plain language text files (called *feature files*), analyses them and runs them against the system. These files represent the previously mentioned executable specifications that tell the system *what* to do. The features are described by concrete examples that use a common vocabulary understood by all parties. Said files are written using a syntax called *Gherkin*. Gherkin enables software to derive meaning from plain language files by using a series of specific keywords. A Gherkin file [Algorithm 1] is organized into:

1. *Feature*. To give some initial documentation regarding the purpose of the document.

2. *Scenario*. To describe the interaction with the system. A single feature comes with multiple scenarios that describe the expected reaction of the system in various situations. A Scenario is composed by a list of steps that have to be executed sequentially. It can either pass or fail and should be able to run on its own without the need of pre-existing data. Scenarios are divided into:

   - Context. Defined by the keyword *Given*. It declares the preconditions and prepares the environment.
   - Event(s). Defined by the keyword *When*, It represents the tested action(s) performed on the system.
   - Outcome(s). Defined by the keyword *Then*. It describes the expected outcome.

   Each line of the scenario is known as a *step*. Multiple steps can be concatenated using the keywords *and* and *but*.

```
1 Feature: Transfer Order;
2     Scenario: Attempt a transfer without items;
3         Given I have 0 Items in Warehouse;
4         When I create a transfer order;
5         Then the Transfer Error message is prompted;
```

**Algorithm 1:** A Feature with Scenario

*Step definitions* are portion of code that provide meaning to each feature step and that allow feature files to be executed. They are written in Ruby and tell the system *how* to do something. All this definitions are usually bundled together in a single file. Cucumber scans the feature files looking for familiar patterns in order to map each step to the corresponding function declared in the step definition file. When a familiar pattern is found the associated code is executed.

## 3.3. Automation

Continuous Integration [6] is a development practice that revolves around the frequent (daily, o even multiple times per day) merging and integration of code. Said technique is employed to increase the rate of release cycles and to improve quality, reliability and productivity. The starting point of CI techniques is keeping all the developed code for a given project in a single shared repository. This is usually achieved using some form of source control tool (like GIT) that the developers can use to access the source code, monitor changes and save modifications. Modern source control systems also allow the use of branches in order to work on various iterations of the product at the same time. Changes to the shared code are handled by a CI server [Figure 3.2] that performs an automated build of the software when changes are committed by the developers in order to detect potential error brought upon by the newly developed code (performing at least a daily or nightly build is usually considered a minimum requirement). An example of such tool is the previously seen Jenkins. Said server can also be used to complete an array of additional tasks (like testing, best practice checking, etc). A CI server is also capable of notifying the developers in case of an unsuccessful build or when problems occurs during the testing stages.

Although a successful build may already indicate some level of quality in the interested code, additional steps can be taken to further improve code reliability. Despite not being strictly a part of CI, continuous testing [7] is usually at least considered within the scope of such techniques and allows (thanks to the funtionalities offered by the CI server) the employment of various testing procedures, including (but not limited to): *Smoke testing* (of the core product functionalities), *unit testing* (of individual units like functions, subroutines or methods), *integration testing* of multiple software units

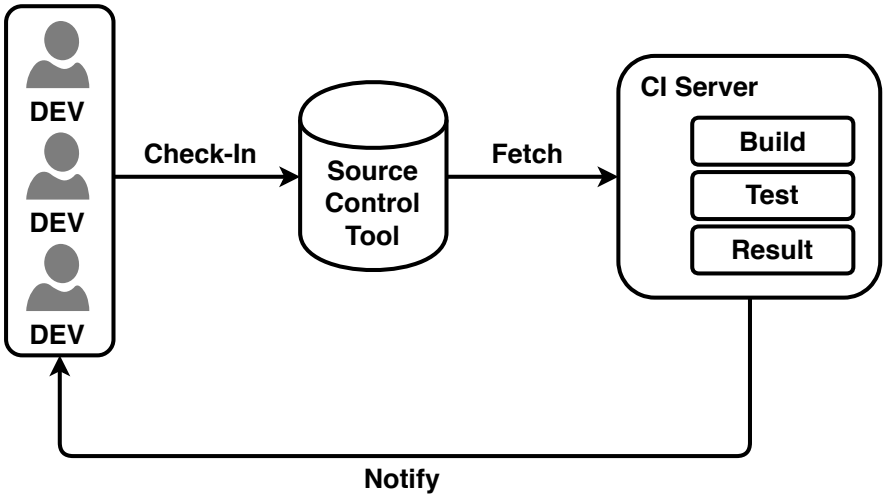


Figure 3.2: Continuous Integration

components interacting with each other that are tested to work as group and acceptance testing (of the required usage scenarios to the determine to which degree the application meets the specified requirements).

Naturally, the main advantage brought by the employment of CI techniques is the automation of a series of processes that, if executed manually, would introduce the possibility of human error and be very time (and resource) consuming or even impossible to implement effectively during the whole development process.

# Chapter 4

# Product

In this chapter we describe the most important tools used during the course of our work. For each section we start with a general description of the software (or framework) purpose and functionalities. After this initial description we provide a more in-depth look regarding some technical aspects of the application. Finally, we describe its relevance in the context of our work.

## 4.1. Dynamics 365 for Finance and Operations / Trade+

Dynamics 365 is a software suite developed by Microsoft Corporation that offers both Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM) solutions. Dynamics 365 for Finance and Operations [8] is an application, part of the Dynamics 365 suite, that offers core ERP functionalities with a particular focus on mid-sized businesses. Said functionalities are: retail, production, supply chain management, procurement (i.e. purchasing), human capital management and finances. The platform can be deployed on-premise, using the company infrastructure, or in the cloud utilizing the Microsoft Azure services. A hybrid deployment is also possible. The platform is accessible via a web-based user interface that replaced the Windows client of the older versions of the software. Typically customers are able to obtain the software through certified partners (like Würth Phoenix) that provide the application licenses and handle the various implementation related activities such as: development of new features and extension of existing ones, consulting, integration with previously existing infrastructures (in the customer's company) and system setup. The Dynamics 365 for Finance and Operations source code is composed by multiple *elements* (like classes, tables, forms, menu items, etc.) which are all organized into eight different *layers*. Elements in the lower layers (SYS, FPK, GLS) are handled directly by Microsoft and are related to core functionalities, patches and localization. Middle layers (SLN, ISV) contain elements used by independent software vendors to implement the added functionalities in vertical solutions (like Trade+). The higher ones (VAR, CUS, USR) are instead related to customer specific implementations. Modifications performed in lower layers have an effect on the elements in the higher ones. A group of related elements (all belonging to the same layer) is called a *model* and constitutes a solution (e.g. a warehouse management model). A model is always part of a *package*, which is a compilation/distribution unit formed by one or more models that also contains metadata describing their properties and behaviors. Packages can reference each other and can be combined to form a *deployable package*, which constitutes a deployable unit that can be integrated to an instance of the application. In Dynamics 365 for Finance and Operations we identify three data types: Setup data (specifies the underlying structure of the business and is almost never changed), master data (describes entities such as products and customers and is rarely changed) and transaction data (describes all the activities performed by the company and is constantly updated). The Architecture [9] of Dynamics 365 for Finance and Operations can be seen as a list of seven conceptual components [Figure 4.1]:

- **Identity**. The authentication management process and access control are managed by the Azure Active Directory, a cloud-based application capable of synchronizing with both on-premise directories and online services. In a cloud deployment of D365, AD uses the Security Assertion Markup Language 2.0 to securely exchange information between parties. Authentication in on-premise iterations of D365 is handled by the Active Directory Federation Services.

- **Storage**. A primary database handles the bulk of the read and write workload. This includes most of the primary transactions, the import/export of files, the financial reporting and the handling of stored documents. Some of the read-only interactions are redirected to a secondary database. A third read-only database handles the business analytics related activities. SQL Azure is used on cloud deployments while Microsoft SQL Server 2016 is utilized in on-premise ones (and always for the business analytics DB).

- **Platform**. Dynamics 365 for Finance and Operations provides its functionalities through different applications (Application Object Servers, Management Reporter and SQL Server Reporting Services) that are all hosted and run on Azure Compute Virtual Machines. The Application Object Servers is particularly important because it handles the communication between user/-client and database.

- **Application**. The application source code and metadata is the same for all deployment types. The application stack is divided into three models: Application Platform (which is the lower model that contains the code that handles the core application features such as runtime, data access, SQL Server Reporting Services, task recording, data import/export, batch and mobile frameworks), Application Foundation (containing functionalities shared by all applications such as Global Address Book and Number Sequence Framework) and Application Suite (that represents the top level code that provides the functionalities that are expanded or modified by Microsoft Partners).

- **Client**. The Dynamics 365 GUI is provided by the HTML 5 browsers-based client and the mobile applications. Both alternatives also integrate features from the Office 365 suite.

- **Development tools**. Visual Studio is the development environment used to extend or modify the application. The first step in customizing Dynamics 365 for Finance and Operations (by adding new code and functionalities) is the creation of a new model that can be deployed in its own package (Extension) or as a part of an existing one (Overlayering). This same procedure is also true during the creation of test cases.

  *Overlayering* means copying the elements of an existing model into a newly created one that can be then modified by the developer. This new model must reside on a higher layer and belong to the same package as the original one. When the application is executed only the code contained in the copy (residing higher up in the layer hierarchy) is considered. The peculiarity of this technique resides in the fact that effectively all the model code is duplicated (not only the one that is modified or added by the developer), hence potentially creating a big overhead. Modifying via *Extension* means generating a new model in order to add functionalities to an existing one (i.e. extend) without creating an entire copy of it. The new model (that, unlike overlayering, resides in his own package) is created together with one or more references to the existing models that need to be expanded. This references allow the new model to expand the functionalities and interact with the elements of the referenced ones (e.g. accessing a table, calling a method, etc.) without directly modifying their source code. Untouched source code of previously exiting models means shorter compilation times and decreased likelihood of issues when the application is updated. For both methods a developer works inside a *project*, which is a logical construct that helps a to manage the interested elements. A project may only contain elements of a single model. Developers work on the application using DEV VMs, which are machines pre-configured with Visual Studio, access to Dynamics and SQL servers. An additional machine is needed to take the developed code, perform the build operations, execute tests and create the deployable packages.

- **Lifecycle Services**. The implementation activities can be facilitated with the use of Lifecycle Services, a collaborative workspace used to define and share standardized processes across the company in order to improve the predictability of the implementation process.

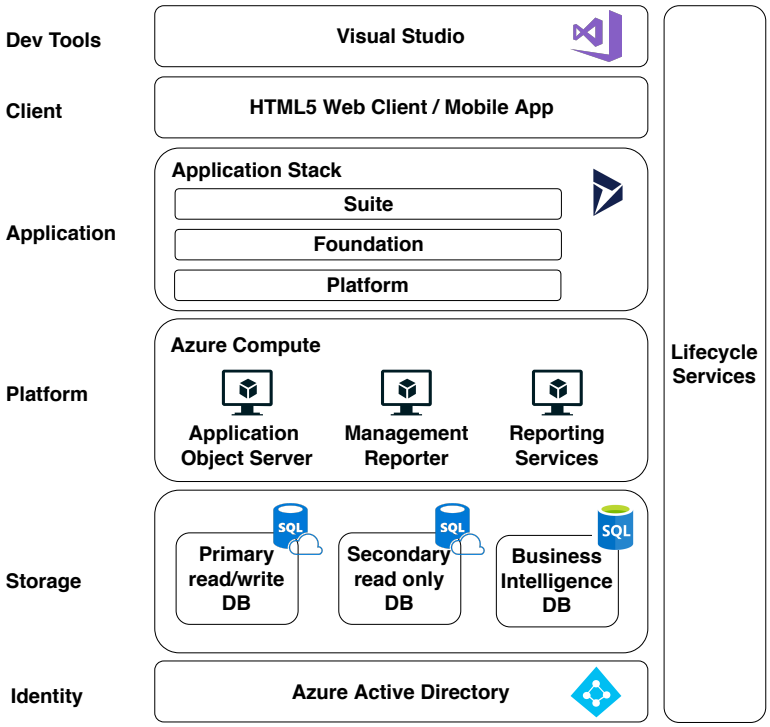


Figure 4.1: Dynamics 365 for Finance and Operations Conceptual Architecture

As stated before Dynamics 365 for Finance and Operations represents a general approach to ERP and it's not generally meant to be sold directly to customers without undergoing at least some degree of personalization. A personalized version of the software is called a *vertical*. Trade+ is the vertical solution offered by Würth Phoenix that focuses on enhancing the wholesale and distribution capabilities of the base offering. The main features offered by Trade+ are related to: Sales & Marketing (administration of associations and groups, back order processing, extended pricing, pricing administration, extended information in the order location, automatic e-mail dispatch), Purchasing & disposition (automatic forecast calculation, extended ABC classification, determination of average monthly requirements, dunning for suppliers, supplier evaluation, article discontinuation control), Warehouse management & logistics (Extended storage and retrieval strategies, preventive and urgent stock transfers, mobile solution for warehouse management, route management and route planning, label printing) and Business Intelligence & Reporting (Data Warehouse, Data Cubes and Reports for Trading and Production Companies). On top of this features the application also provides various customizations on existing functionalities that present added or modified aspects and user interface elements in comparison to the base version of the software provided by Microsoft. This fact later brought us to choose five business processes that, despite being already present in the base application, could be affected in some ways by the interaction with the mentioned changes and extensions provided by the company.

## 4.2. Team Foundation Server / Azure DevOps

Azure DevOps [10] (previously known as Visual Studio Team Services) is a lifecycle management tool developed by Microsoft. Just like Dynamics 365 this tool can exist on the cloud, as Azure De-

vOps, or on-premise, as Team Foundation Server (from 2019 onward this version will be renamed Azure DevOps Server). The architectural difference between the two iteration of the software goes beyond the scope of this chapter given the fact that both versions offer the same set of functionalities to the end user. For the sake of simplicity from now on we will refer to the application as Azure DevOps. Azure DevOps is composed by five elements, each covering a different aspect of the lifecycle management process:

- **Boards** offers a way to plan and track all the work activities related to development (epics, features and user stories), testing (test plans, test suites and test cases) and bug fixing. Each item is assigned to one or more team members and comes with various information such as: title, description, effort estimate, relation to other elements, etc. The visualization, tracking and modification of the status (new, active, resolved, closed, removed) of each item can be performed using a backlog or a Kanban board. Sprint management is also supported.
- **Repos** is a set of version control features that developers can use to track code changes, branch and create pull requests. Azure Repos supports both Git (as a distributed VC) and Team Foundation Version Control (as a centralized VC) and can be connected with various development environments. By using Git every developer saves a full copy of the repository on the machine and is able to work on it even when disconnected from the network. With TFVC only one iteration of the project is saved locally, all other data is maintained on the server.
- **Pipelines** takes the code from Repos (or any other supported providers like GitHub) and allows to enforce Continuous Integration and Continuous Delivery techniques by performing automated sequence of steps (called pipelines) to build, test and deploy the newly written code. The execution of each pipeline can be scheduled following specific trigger events (e.g. code changes are checked in) and their execution can be monitored directly from the application environment.
- **Test Plans** allows developers to perform automated, manual and load testing techniques and to monitor the status of each test case and test plan for a specific project. Issues during test execution are notified to all team member and can be further documented by adding additional information. Naturally the elements of Azure Test Plans can be used in the previously mentioned pipelines.
- **Artifacts** allows the management of (NuGet, npm and Maven) packages.

The application is normally accessed via the web-based client but its functionalities can also be integrated directly into the development environment with the Visual Studio Team Explorer plug-in. Once connected to a project the developer can see his assigned work items, manage his pending changes (i.e. modifications to the code that have yet to be checked in), explore and interact with the version control tool and manage builds related to his code changes. Of particular importance for our work is how Azure DevOps helps developers to plan, develop, monitor and integrate test cases (and other application extensions) for a given project in order to easily enforce CI/CD techniques [Figure 4.2].

## 4.3. SysTest Framework

The SysTest framework provides developers a set of classes and methods that can be utilized to author test cases. In general (for ERP application) we identify five kinds of applicable testing techniques: unit testing (SysTest), feature/component testing (SysTest + Task recorder), integration testing (SysTest + Task recorder), performance testing (PerfSDK), user acceptance testing (Task recorder) and regression testing (SysTest + DevOps). As already mentioned, the content of this work revolves around: component testing, integration testing and regression testing. The first step in creating a test case is extending the interested test class with the *SysTestCase* one. This class automatically handles the setup and teardown process of the test case (needed to prepare the environment and

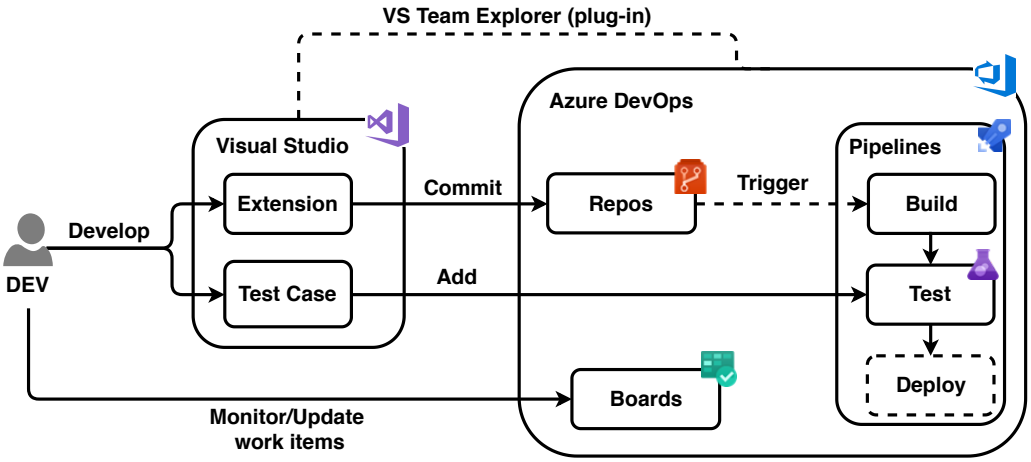


Figure 4.2: Testing and Development with Azure DevOps

rollback the performed changes at the end of the execution), allows the creation of test suites and provides the assert statement functionalities. Every test class should be named after the component/feature/process that it tests and may contain one or more test methods. Test methods are *void* and are recognized by the system via the decorator reference to the *SysTestMethodAttribute* class. As a best practice the level of granularity of the test method should also be stated via the *SysTestGranularityAttribute* class. As a general rule a test method should be organized into three sections:

- **Arrange**. The data that is needed during the test execution is created. This phase should be kept minimal (i.e. declare only data that is strictly needed in order to make the test case run) and can be skipped altogether in certain scenarios.
- **Act**. The testing operation are executed.
- **Assert**. The results of the act block are observed and validated. The *assert* statement is an essential part of a test method because it determines the nature of the outcome (successful or unsuccessful) once the test is executed. In order to perform an assert statement the *this* keyword is used (since the *SysTestCase* class extends the *SysTestAssert* one) followed by the needed assert method (assertEquals, assertTrue, assertNotNull, etc.).

As mentioned before it is also possible to generate coded test cases directly from XML recordings obtained through the Task recorder. Although a more in-depth look at this tool will be provided in the next chapter, it is important to describe how the tests generated from this process differ from the ones described above. A newly generated test (which is represented by a single test class) is composed by: a variable declaration area, a *void SetUpTestCase* method that prepares the environment, a *void setupData* method that assigns values to the previously defined variables and a single *void testMethod*. The peculiarity of this *testMethod* resides in the fact that it uses *Form Adaptors* in order to simulate interactions with a Dynamics 365 for Finance and Operations GUI. We move in the application using this wrapper classes (over forms) that provide an API that can be used to mimic user interaction with the system elements. This allows the creation of test cases in which the actions taken during the recording process by a user can be translated into X++ code. Because of this reason, test cases generated from task recordings are considered *headless*, this means that they are able to mimic GUI interactions without actually having to display it. This allows for a very fast test execution. Another peculiarity of this method is the fact that it does not contain an assert section. Because of this reason a generated test case is considered successful when all the steps can be completed without raising any errors.

Another relevant topic related to the the SysTest framework is how it handles the creation, deletion or modification of data during test execution. For Dynamics 365 the objective of Microsoft was

to allow developer to execute any test case without the risk of compromising the state of the underlying database. This meant returning to the original DB state at the end of every test execution. In order to implement this feature a rollback mechanism [Figure 4.3] was baked into the framework. This was achieved by treating each framework execution as a single DB transaction and by utilizing SQL SavePoints to implement nested/sub transactions. Because of this reason a savepoint is created (by a savepoint manager) every time the framework, a test suite or a test class are started. When one of this elements (or a test method) completes its execution, the data is restored to the last created savepoint, effectively nullifying the changes that have been performed up to that point. It is important to note that the rollback mechanism always follows the hierarchy of the transaction (e.g. when a test class is completed, the framework goes back to the last savepoint created before the start of the class). The implementation of this technique ensures that DB is not left in an unreliable state after the execution process.

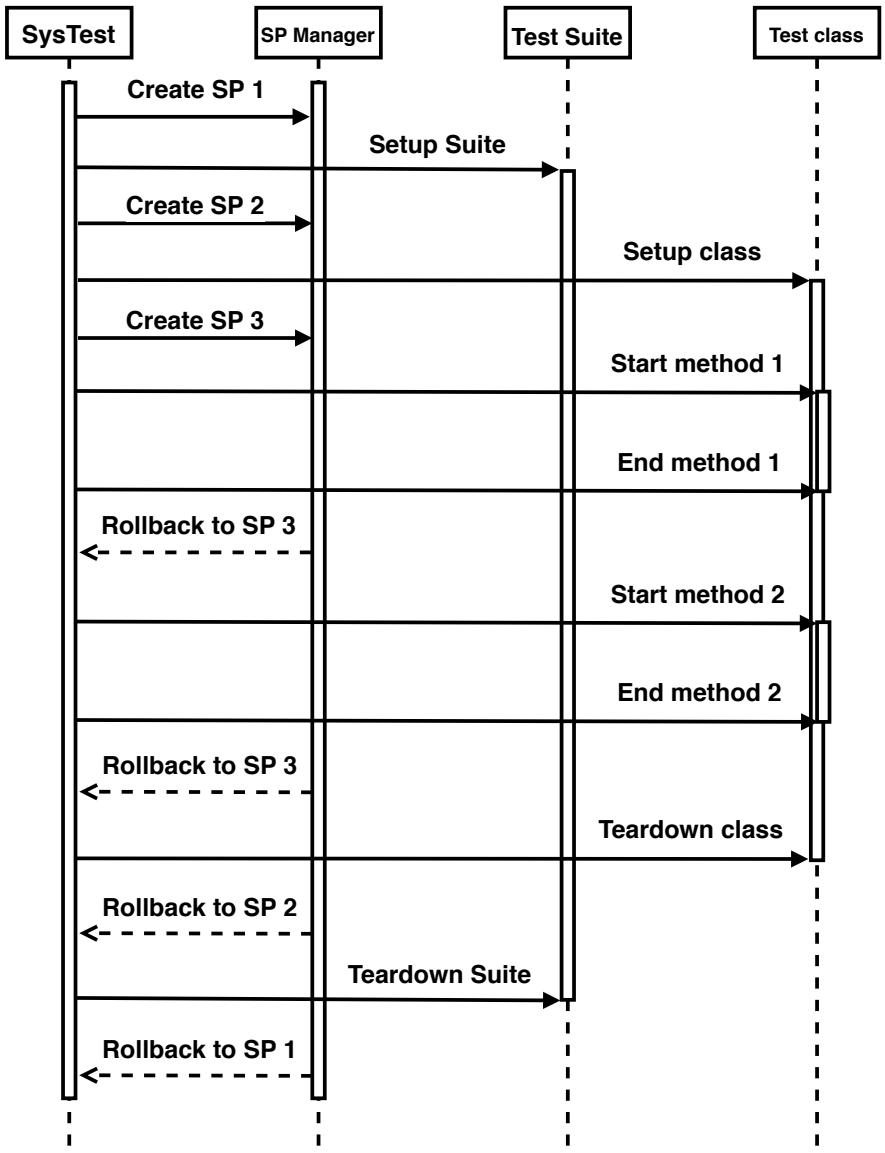


Figure 4.3: Data rollback

In the context of our work, the SysTest Framework allowed us to successfully cover the initially defined business processes with a set of executable test cases (Chapter 5).

# Chapter 5

# Process

## 5.1. Diagnostic

The idea of exploring the topic of testing in a Dynamics 365 for Finance and Operation environment was firstly introduced as a possible solution to compensate the lack of a proper internal methodology to follow in order maintain high standards of quality and reliability during the development of a project. At the beginning of this work we explained how the purpose of the Diagnostic phase was the creation of a project timeline that would allow us to form an initial determination of the project activities, together with their expected results and the amount of time to dedicate to each individual phase. This allowed us to better understand how many resources needed to be allocated for the completion of the project and gave us a tool to monitor to which degree we were able to respect the predefined timeline. In this section we will not provide further information regarding the structure of the plan (section 1.4.2) and the defined requirements (Chapter 2).

In this phase, a secondary activity was setting up the environment that would later be used to access the development tools. The company account was created and given development access to the Trade+ project on Azure DevOps. A dev virtual machine was also set up and the Visual Studio Environment was connected to the project using Visual Studio Team Explorer. Relevant links were provided in order to access the Dynamics Learning Portal, Lifecycle Services, Microsoft Developers Network, the company Intranet and the Trade+ project portal.

In this period the planning of training activities was also performed. This meant finding useful and complete resources that were related to the topics that would later be covered during the course of the project and organizing the face-to-face lectures necessary to obtain all the basic knowledge required to complete our work.

## 5.2. Analysis

The identification of the activities that had to be covered by the developed test cases revolved around finding a good balance between the skills that could be acquired during the training activities and the usefulness of the chosen processes. As mentioned before (Section 1.4.2) we settled on five business processes that had to be analyzed in order to have a satisfactory output (both in terms of coverage and documentation) at the end of the project. Said processes, together with the steps that compose them, read as follows:

- **Purchase Order**. Creation of new license plates to identify the purchased goods, creation of a new PO (with vendor and arrival site and warehouse), specification of items number and quantity, order confirmation, specification of order lines, baydoor registration, creation of product receipt (with id), posting of product receipt, invoice creation, invoice posting.
- **Production Order**. Creation of PO (with item number, delivery date and quantity), validation of PO, update of BOM (bill of materials) and production route, estimation and cost manage-

ment, scheduling/release/start of production, reporting of production as finished (with publishing of journal entry).

- **Transfer Order**. Creation of TO (with origin and target warehouse), specification of items number and quantity, inventory reservation, items picking from origin warehouse, items shipping to target warehouse, receival confirmation.

- **Sales Order**. Moving items from receival area to storage (transfer journal), creation of a new SO (with customer account and origin site and warehouse), specification of items number and quantity, inventory reservation, release to warehouse (creation of work item), validation and completion of work, creation of shipping load, shipment.

- **Return Order**. Creation of RO (with customer account, return site and warehouse and return reason code), specification of items number and quantity, specification of order lines, baydoor registration.

After this initial definition we also decided that it would be appropriate (as a secondary step) not only to test the business processes in isolation (*feature testing*), but also as single coherent workflow (*integration testing*) in which the various tasks were interconnected with one another [Figure 5.1] and where a given set of purchased (and produced) goods moved through all the stages of the covered processes.

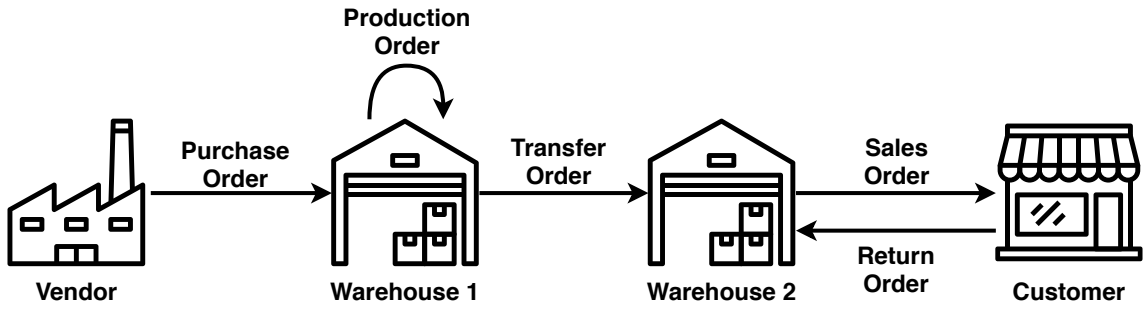


Figure 5.1: Processes workflow

The Analysis phase was also characterized by a series of training activities (mentioned in section 1.3) that were organized in order to give a satisfactory introduction to the world of ERP applications and that naturally had a particular focus on the Dynamics suite. The "Introductory lectures for new hires" were carried out with regularity by the company and aimed to form new team members to the internal practices, policies and tools. During the first weeks the following lectures were organized: Introduction to Dynamics, look and feel of Dynamics 365, sales and logistics, production, development process overview and introduction to Trade+. On the Dynamics Learning Portal we followed the set of lectures called "Development, Extensions and Deployment for Microsoft Dynamics 365 for Finance and Operations" (Exam MB6-894) that covered: The understanding of Dynamics 365 for Finance and Operations architecture (application stack, cloud components, server architecture, layers), development environment (Visual Studio and Lifecycle Services), development of new elements (creation and management of data types, tables and labels), the X++ programming language, user interface, security and component extension. Additional (independent) lectures were also followed on the topics of: System architecture and technologies (80765BE), cloud deployment (81144AE), advanced security (80929BE) and DevOps testing/practices/principles (Steven Borg, Sam Guckenheimer).

## 5.3. Design

Dynamics 365 for Finance and Operations provides a easy-to-use tool (the *Task Recorder*) to record business processes and download them as .xml or .axtr files. The tool is integrated in the browser client and, when activated, records every user interaction with the system creating a step-by-step list of the performed actions. Each step appears to the user as a human readable, high level description of a single action (e.g. In the Warehouse field, type '6') and, once the recording file is downloaded, corresponds to a portion of the XML code that provides the system with information on the operation that has to be performed on the GUI. The recording process can be interrupted and resumed at any point in time and every created step can be commented (to provide further information to the users that will reproduce the interested recording), modified or deleted. A downloaded recording can be saved in the Lifecycle Service portal as a future reference guide (.axtr), attached to a DevOps work item (.axtr or .xml), played back and edited by another user in his Dynamics environment (.axtr or .xml) and converted into a test case (.xml). The playback functionality is particularly useful to guide developers when a bug (that triggers after a series of specific actions) needs to be fixed. Because of this reason the recordings are usually created by consultants and attached to Azure Boards work items. Said items (as mentioned in section 4.2) are assigned to developers that can later download and use the attached recording file to emulate the interested behavior on their personal environment in order to better understand the problem without having to only rely on a written description.

After an initial period, necessary in order to get accustomed with the tool, we started to plan the recording activities. A major focus during this phase was the reduction of the steps needed to cover the interested business processes. This activity was of particular importance in order to reduce the amount of code that a given recording would later generate when converted into a coded test case. A shorter test case is generally easier to maintain, expand and modify. Furthermore, superfluous steps would have increased the likelihood of unexpected errors and behaviors. Other than reducing the total number of steps, we also made sure that each recording could be played back on any machine running the same version of Dynamics 365 for Finance and Operations, independently from the underlying status of the database at the start of the execution. This meant encompassing all the activities related with the creation of the needed data at the beginning (or during) the interested business process (e.g. If a license plate needs to be used, create it before using it). Some information (like the predefined selection in a drop-down menu) is saved locally by Dynamics 365 for Finance and Operations and depends on how the user has previously interacted with the system. This peculiarity had to be taken into consideration during the recording activities in order to create recordings that could be played back (and later executed) on any machine (e.g. make sure that each item in a form is explicitly defined during the recording).

Regarding the business processes covered in our work we decided to create a single recording for each activity, ending the Design phase with five .xml files that could later be utilized to create the foundation for our test cases.

## 5.4. Development

This section covers all the Development tasks that were carried out during the course of our work. We decided to organize it into three subsections corresponding to the different macro-activities of this phase (mentioned in section 1.4.2). The first subsection describes the *creation* of the project that hosted our code and the *translation* of the task recordings into coded test cases. The second one provides information regarding how said test have been *expanded* in order to fulfill the testing requirements of the application. Finally, the last subsection describes the process of *integrating* the test cases with the nightly builds.

### 5.4.1. Creation

As mentioned before (section 4.1), the handling of Dynamics 365 for Finance and Operations elements is done via a project that needs to be tied to a single model. Because of this reason, at the beginning of this phase, we decided to create a new model to contain all of our test classes. This new model (called *TradePlusTest*) was generated in his own package, hence following the *Extension* technique (discussed in section 4.1) and referenced the ones belonging to the application stack (Application Platform, Application Foundation and Application Suite). This was required in order for the test classes to interact with the application elements (e.g forms, tables, menu items, etc.). Another reference to the *TestEssentials* model was created in order to have access to the various methods and classes part of the SysTest Framework (section 4.3) needed to write functioning test classes.

Once the project was created, the five .xml recordings containing the instructions needed to cover our business processes were imported in the Visual Studio environment via the dedicated software utility that converted each recording into an executable X++ test case. During this initial stage the coded test cases were left mostly unmodified from their original state and a successful test run merely indicated that the system was able to correctly perform all the actions needed to complete the interested business processes. As mentioned before (section 4.3), generated test cases are not created with any assert statement. Despite this fact it is possible to add them later given the fact that test cases created from task recordings still extend the *SysTestAssert* class that provides the assert statement functionalities. As a first development activity, we decided to verify if the actions performed in the various business processes during a test run produced the expected results on the application database. This task included running SQL queries at the beginning and end of each test in order to check (using assert statements) if the data corresponding to the interested items was correctly created or modified during the business process execution. This activity, while still technically not expanding the scope of each test case, allowed us to obtain an initial outlook on how to interact with the generated X++ code.

### 5.4.2. Expansion

For expanding our tests we focused on verifying the behavior of the system in a variety of edge cases. That is, interactions that did not strictly comply to the ones expected during a normal execution of the business processes. This approach had the goal of proving the ability of the system of correctly reacting to an array of unexpected scenarios, not only by being able to resume the temporarily compromised workflow, but also by providing exhaustive information messages to the end user. Furthermore, having a set of "expanded" tests (and documenting the tasks related to this expansion) could provide a point of reference for the future testing activities. During the course of our work, we realized that implementing assert statements to expand the scope of our test cases meant analyzing all the actions and sub-tasks that were part of our business processes and recognizing and asserting all (or at least most of) the *successful* and *unsuccessful* scenarios that could derive from said interactions with the system. The analysis of unsuccessful scenarios was of particular importance for us given the fact we not only needed to verify that the client was able to handle these situations correctly (i.e. still being usable after an unexpected event occurred), but also that the appropriate error or warning message was prompted during the process. This last task revealed itself to be a particularly challenging activity given the fact that it required finding a balance between specificity and future proofness (e.g. what happens if an error message is changed after an application update?). Because of this reason we decided to handle the verification of prompted messages by identifying the presence of specific *keywords* that indicated that the correct issue was addressed by the application without having to rely on a (much less flexible) hard-coded identification of the whole message. This approach allowed us to verify the usability of Trade+ in the context of the tested processes without having to constantly refactor our test code as changes were brought to the application by new updates.

The identification and analysis of sub-activities in our business processes was of central importance in order to plan the expansion of the generated test cases. This task allowed us to better understand potential points of weakness in the application and to adapt our development activities

accordingly. The Analysis section of this chapter provided a good overlook on the various steps composing the investigated business processes. In this subsection, we decided to follow the same type of structural description in order to introduce the analyzed edge scenarios used to expand our test cases:

- **Purchase Order**. Creation of LP using already existing and invalid ids. New PO with non existing vendor/site/warehouse and with invalid combinations of said values. Addition of non existing or not available items. Insertion of out-of-range or invalid quantities. Missing or invalid values (site/location/license plate) during order registration. Creation of receipt without or with already existing id. Creation of invoice without or with already existing id, posting of invoice without matching process.
- **Production Order**. Creation of PO using non existing item number and invalid quantity. Past or invalid delivery date during validation. Modification of bill of materials lines (components/ingredients) and versions. Invalid quantity in production route. Past or invalid scheduling date. Invalid good/error quantity and error cause values.
- **Transfer Order**. Creation of TO with non existing/invalid/same origin and target warehouse. Addition of non existing or not available item ids. Insertion of out-of-range or invalid quantities. Reservation of unavailable items. Missing or invalid values (location/license plate) during item picking.
- **Sales Order**. Transfer journal with invalid or non existing origin/target site and warehouse and item quantity. Transfer Journal posting without validation. New SO with non existing customer account/site/warehouse and with invalid combinations of said values. Insertion of out-of-range or invalid quantities. Non existing user id for warehouse work validation, completion and shipment confirmation.
- **Return Order**. New RO with non existing customer/site/warehouse and with invalid combinations of said values. Invalid insertion of return reason code. Insertion of out-of-range or invalid quantities. Missing or invalid license plate.

Given the fact that some business processes share the same (or very similar) subroutines, it stands to reason that certain portion of different test cases will be expanded in the same way. At the same time, there are actions that are repeated multiple times inside the same process (e.g. if we create a purchase order for two items, we will need to repeat the activities related to each item multiple times). Both of this situations bring redundancy to the developed code that, subsequently, creates longer (and less manageable) test cases. Because of this reason, we decided to define a new shared class, called *SharedTests*, that would allow multiple business processes to handle this repetitions with simple method calls. In this newly created class each method represented either a *subroutine* (a series of actions performed on the application interface) or a test *expansion* (setting up an edge case, getting the error/warning message, asserting). As mentioned before (section 4.3), each generated test has a *setupData* method that assigns values to the used variables. We decided to create an additional method, *setupSharedTestsData*, called at the end of *setupData* in order to pass the needed values for a given test case to the *sharedTests* class.

Up until this point we have dealt with our business processes in isolation (component testing). This required us to generate starting data at the beginning of some test cases in order to have the necessary prerequisites for a successful execution. The previously mentioned rollback mechanism (section 4.3) made it somewhat difficult to merge the various processes in a single workflow (integration testing), given the fact that the changes made by a given business process were lost before the execution of the following one. Because of this reason, we decided to further investigate the structure of the SysTest framework in order to find a solution that would better suit our needs. Further examination into the (at the time poorly documented) framework revealed that the creation of a *Test Suite* was a valid alternative to overcome the data issue. Because of this reason, we implemented the functionalities provided by the *SysTestSuite* class by adding all the business processes (together with the *SharedTests* class) in a suite [Figure 5.2] that was recognized as a single entity by the framework and therefore allowed us to create an homogeneous workflow between the developed test cases.

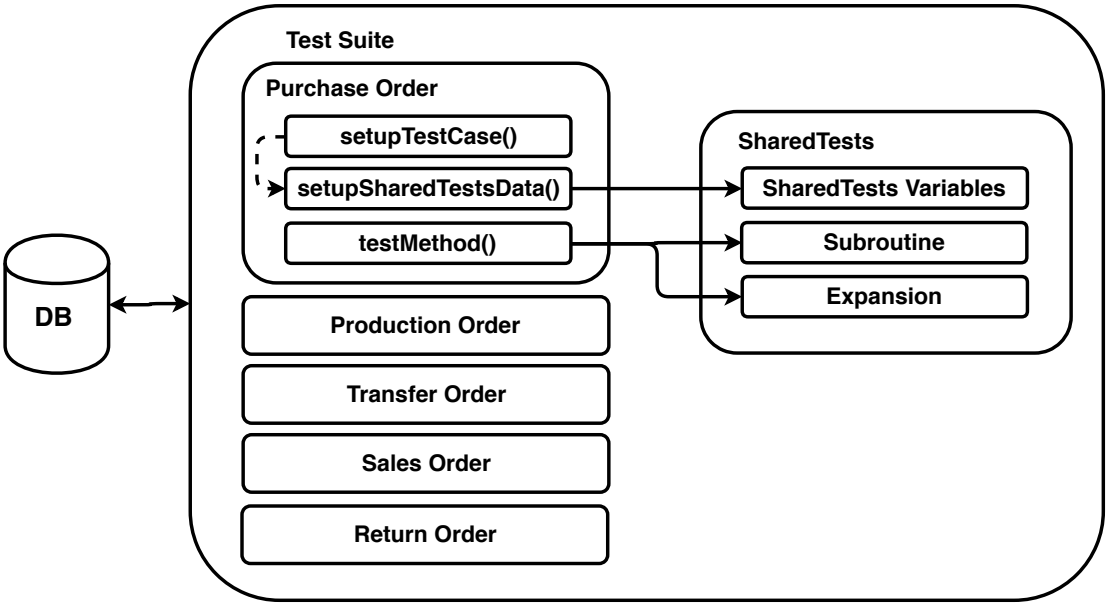


Figure 5.2: Structure of the test suite

### 5.4.3. Build Integration

The next step in the development process revolved around the integration of the developed test cases in the nightly builds. In this context, the build process was modified, extended and monitored in the Azure Pipelines section of the Azure DevOps environment. In Dynamics 365 for Finance and Operation the activities related to this process are handled by a build VM that contains: build agent(s) (that run the computational operations of the build process and that are connected directly to the interested Azure DevOps project), controller (that distributes the build workload to all the available agents), process template (where the build steps are defined) and definition (that specifies what to build, when a build should start, where the build output should be sent). The tests that need to be added to the build process are specified in the build definition and are executed after the build process has proven to be successful. Test execution is organized into three activities: test setup, test execution and test end (teardown). The Trade+ code is subjected to nightly builds that are executed every 24 hours and that can naturally support the integration of test cases via the Azure DevOps application. We decided to begin the integration process by adding a newly generated test case (without any added code or assert statement) in order to verify that the addition of the testing steps in the build pipeline of our project would function without any issues. Once this initial verification was completed, we decided to purposefully insert a failing assert statement to monitor the reaction of the system in this particular scenario. At this point we were able to verify, by looking at the recap feature provided by Azure Pipelines, that the build was registered as "partially succeeded". Further investigation allowed us to establish the pipeline step causing the problem (test execution) , explore the details regarding the specifics of the failure (name of the test case, error line, error message) and verify that everything was correctly reported. Finally, after having verified that each aspect of the integration process worked as intended, we decided to add the developed test suite to the build pipeline, effectively enabling the regression testing aspect of the project and ending the development phase.

## 5.5. Deployment

The integration of the developed test cases in the build process marked the end of the development phase. The focus of the next activities (part of the deployment phase) was related with the process of sharing to the company the knowledge acquired during the various tasks carried out to complete our project. This technical and procedural know-how was presented through a series of final meetings that focused on analyzing the individual phases executed to create and integrate (in the nightly builds) the set of developed test cases. Given the fact that said tasks have already been described in detail during the course of this chapter, we will not provide further information on this matter inside this subsection. Other topics that were discussed during this "end project" meetings were: a general evaluation regarding the resources required (in terms of time) to implement the described techniques (section 6.2) and some information regarding various best practices that could be followed in order to reduce the total amount of time needed in order to complete a similar set of testing activities. We considered important to communicate that during the course of our work the quality of a an XML recording could greatly influence the time spent refactoring the code of a generated X++ test case. We highlighted how the more good practices (like reducing the total number of steps to complete the business process, planning carefully the whole process before starting the recording activities, etc.) were respected during an XML creation, the least the developer had to work on the initial test code. Refactoring generated code was usually a lengthy and cumbersome process, it was always faster to create a good recording in order to save time during the development activities. We also stressed the importance of reducing as much as possible the amount of code for a given test case. Working with shorter tests was always easier and ideally a fine level of granularity should always be seeked in order to facilitate the diagnostic activities in the case of an unsuccessful test execution. We also communicated that splitting a long process into multiple sub-tasks and recognizing subroutines that have already been seen in other processes (in order to execute them in a shared class) was also an important technique that allowed us to reduce code redundancy and create more manageable test cases. The conclusion of these meetings indicated that the deployment phase was over and that all the relevant information was correctly communicated to the company.

## 5.6. Operation (Future Work)

The various activities related to this project were considered concluded after the know-how was passed through the previously mentioned meetings. Every task performed by the company to support and further expand the applied testing techniques falls under the operation phase and can be considered part of the future work. In this subsection we will briefly explore what could require attention in the future in order to maintain the already implemented testing techniques and further expand them. It is important to mention how Dynamics 365 for Finance and Operations falls under the Microsoft Modern Lifecycle Policy, which mandates users to update their application within 30 days after the official release of a new software version. Microsoft refuses to investigate issues or troubleshoot implementations that do not comply with the imposed deadline and that, because of this reason, are considered outdated. Knowing this fact, it is easy to understand the importance of having test cases that work with a certain degree of flexibility regarding slight changes to the underlying software infrastructure. We already mentioned that techniques such as the identification of process errors and warning messages through the use of specific keywords will certainly help in this regard. Despite this fact, it is fairly unreasonable to aim at the creation of a test case that will not be eventually affected by underlying software changes. Because of this reason, the refactoring of already developed test code will certainly be one of the main activities performed by the company if it will decide to further implement the testing techniques described in this work. Fortunately, the nightly build integration and, in a more general sense, the application of regression testing techniques together with the monitoring tools provided by Microsoft (i.e. Azure DevOps) allows the developer to identify a failure with a high degree of accuracy as soon as the problem arises during the test execution. Other than the maintenance of test cases related to the already covered business processes, the company will need to expand the developed suite in order to enlarge the scope of the performed

testing activities. In a general sense, the testing tasks described in this work can be applied and extended to varying degrees of intensity depending on the requirements of a specific project. We can identify three main approaches that could be *immediately* applied in order to support the company's various products during their development cycle:

- Applying testing techniques to all the added, extended or modified features. Very time consuming but also particularly effective.
- Identifying a list of core (added, extended or modified) functionalities that have to be covered (similar to what has been done in this work).
- Testing only after a specific feature of the application has failed (i.e. adding a test case after the fix). Not very effective but also less time consuming in comparison to the other approaches.

A *longer term* improvement would include the partial redesign of the development process with the introduction of test-first approaches (like the previously mentioned BDD/TDD/ATDD) that would greatly benefit the overall project results. Given the often fast paced and complex nature of the development cycles in this particular industry, together with the fact that the development start from a previously existing application (rather than from scratch), it is easy to understand how integrating this kind of techniques is a very complex task that would require careful planning and a relevant investment of time and resourced by the company.

Other activities that could also be explored in the future in order to improve the techniques described in this work and that were excluded (due to time or experience constraint) are: the use of the PerfSDK to apply performance (load, stress, endurance, spike) testing techniques and the creation of unit test cases from scratch (i.e. without starting from a pre-existing recording).

# Chapter 6

# Evaluation

In this chapter we provide a final evaluation regarding the results achieved during the course of this work. Initially, we estimate an approximate test coverage achieved on Trade+ with the analyzed business processes. Then, we elaborate on the total amount of resources needed (in terms of time) in order to acquire the required knowledge necessary to work and develop on the application and to apply said knowledge to create the required test cases. Finally, we give an insight regarding the usefulness of the project as a whole with a final focus on the applicability that it could have in a context with tighter schedules.

## 6.1. Coverage

Providing an accurate approximation on the total coverage of our test suite is a particularly complex task given the large and ever-expanding nature of the application that we worked on. In order to discuss this task as precisely as possible we will evaluate this topic in relation to three different metrics: total coverage of the chosen business processes (under normal conditions), amount of covered edge cases and degree of coverage of the application as a whole. The five business processes that we have chosen as a starting point for our work were covered completely in relation to the initially defined requirements. Knowing this fact, it is important to specify that in order to respect the project time constraint we intentionally left out (during the requirements definition) some additional activities that could have been involved in the covered processes. Such activities were mainly related to: Advanced warehouse management (warehouse configuration, partial shipment, inventory counting and replenishment), advanced production activities (material exceptions, BOM configuration/-customization) and testing related to the interaction between the application and the mobile WMS software. In regard to the considered edge cases, we focused mainly on user inputs that could have caused possible issues with the application. This meant focusing the expansion of test cases on invalid (wrong type), out of range or unacceptable input values in forms and unexpected interaction with UI elements (e.g. a missing step in a business process). Limitations in terms of time and experience brought us to the decision of excluding more complex kinds of edge cases and advanced techniques such as: the verification of performance under heavy load and role/duties/privileges - based testing. If we look at the application as a whole (considering all the available functionalities that it offers), we will notice that our test cases only covered a relatively small portion of all the provided features. Despite this fact we have to consider that not every business process will have the same relevance to the final customer and that identifying the core functionalities to prioritize during the testing activities was particularly important considered the limited time at our disposal. Furthermore, the future availability of the provided documentation will surely decrease the time required to apply the described techniques to previously untested portion of the application.

## 6.2. Resources

When we analyze the (temporal) resources required to complete this project it is important to take into consideration some concepts that influenced the final outcome of the work. In this regard we want to differentiate two aspects that played a role on this subject: The first relates to the resources necessary for a new team member to acquire the knowledge necessary in order to apply the testing techniques on the Dynamics 365 for Finance and Operation application and to create the documentation of the development process. Insight regarding the acquisition of this knowledge is particularly useful for the reader to have a more precise outlook regarding the scope of the work and for a company to decide if it wants to start the implementation of this kind of techniques on their applications. Regarding this first concept, it is important to notice that a big portion of the project time was spent familiarizing with the application, completing the necessary training activities and creating the documentation needed to share the final know-how (which was elaborated during all the phases of the project). This set of tasks was naturally of central importance for us in order to work effectively and, at the end, to provide some useful information to the company. However, it is easy to understand how a team member already part of the company would not need all the training activities that were previously described. Furthermore, the process of creating the documentation that describes the steps for the creation of a test case and the particulars of the used framework has to be performed only once and can later be uses indefinitely by the other members (hence drastically reducing the required time needed to write functioning tests).

The second aspect that we want to describe is related to the time required for a developer (already in possession of the previously mentioned know-how) to implement the described testing activities for a new business process (which is very important for the company in order to have a general idea of the expected amount of work needed when a new testing activity is started). We have estimated that a testing activity (from the creation of the first recording to the end of the expansion process) can be concluded from a single developer in a matter of hours if there is a constant availability of the relevant documentation and a good knowledge of the underlying business process and on what has to be tested and expanded. Giving a more precise estimation is fairly difficult given the fact that the size (and scope) of the tested feature(s) strongly influences the total time required to complete the interested testing activities. Naturally, our development required much more time in comparison to the one expected in a real-world scenario because of the learning curve that was involved in the process (it was the first-time approach to these techniques and no pre-existing know-how was provided).

## 6.3. Usefulness and Applicability

The project has certainly provided useful information to the company regarding the testing activities that could potentially be implemented in their future and currently ongoing projects. The fact that we have provided a well documented overlook on the whole process is naturally a positive aspect, particularly helpful in relation to the scarce availability of official documentation regarding certain technical aspects of the application. In regard to the general level of applicability of this work, we should take into consideration the fact that we worked on a project developed internally by the company and not strictly influenced by deadlines imposed on us by a specific client (as mentioned before, Trade+ is developed as a starting point for customers that can request their own personalization to be built on top of it). Applying testing techniques in projects that have tighter schedules that need to be followed in order to deliver specific functionalities in time would certainly be a more difficult task and would require a careful planning activity to be performed by the company. Despite this fact, we believe that introducing some form of testing in most (or all) projects can greatly improve the reliability of the developed software and, because of this reason, may represents a worthwhile investment (for both the company and the costumer) in order to produce an higher quality product.

# Chapter 7

# Conclusion

In this chapter we wrap up the previously described content by providing a short summary to recap the various activities that were carried out during the course of this project. In this work we described the approach that was taken in order to implement a set testing techniques in a Dynamics 365 for Finance and Operations environment (particularly in the context of the Trade+ solution). We started by providing an in-depth technical description of the used framework, application and life-cycle management tool. Then, we described the various training activities and initial meetings with the company that allowed us to define the structure and requirements of the project and to obtain the required knowledge in order to start the work on the application. After this initial description, we illustrated the process of creating the first task recordings in the browser client, saving them as XML files and converting them into coded test cases using the tool provided by the Visual Studio environment. This allowed us to perform and apply the first component/feature testing techniques by generating a set of executable test cases that worked in isolation (from the underlying database and from one another). Subsequently, we introduced the concept of the expansion of test cases to verify the reaction of the system under various edge scenarios. At the end of this activity we worked to allow the various test cases to interact with one another and function as a group that represented a single and coherent workflow covering all the interested business processes (integration testing). Finally, we integrated the set of developed test cases into the nightly builds to allow for the application of regression testing techniques. During the whole project we documented the various activities that were performed on a day-to-day basis and then we passed the relevant information regarding the acquired know-how to the company. The final results of this work showed us that the implementation of the initially defined requirements was not only possible in practice, but also a viable and useful approach to follow during the whole development process in order to increase the quality and reliability of the final product. Furthermore, it is important to notice that the application of the previously described techniques is not necessarily a binary approach, meaning that the company has the ability (particularly with the provided documentation) to implement this activities with various degree of intensity depending on the requirement of a given project and the particulars of the external deadlines imposed by the customer. Because of this reason we believe that this topic may be worth exploring for most companies producing ERP software that are interested in improving the overall quality of their final products.

# Bibliography


[1] Chandru Shankar Nilesh Thakkar, Vincent Bellefroid. *Customer Success with Microsoft Dynamics Sure Step*. Packt Publishing, 2014.

[2] V. Armenise. Continuous delivery with jenkins: Jenkins solutions to implement continuous delivery. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, pages 24–27, May 2015.

[3] Jenkins user handbook. `https://jenkins.io/user-handbook.pdf`, 2018.

[4] C. Solis and X. Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387, Aug 2011.

[5] Aslak Hellesøy Matt Wynne. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.

[6] Liming Zhu Mojtaba Shahin, Muhammad Ali Babar. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[7] Maximiliano A. Mascheroni and Emanuel Irrazábal. Continuous testing and solutions for testing problems in continuous delivery: A systematic literature review. *Computación y Sistemas*, 22(3), 2018.

[8] Andreas Luszczak. *Using Microsoft Dynamics 365 for Finance and Operations - Learn and understand the functionality of Microsoft's enterprise solution*. Springer Vieweg, 2019.

[9] Rahul Mohta JJ Yadav, Yogesh Kasat. *Implementing Microsoft Dynamics 365 for Finance and Operations*. Packt Publishing, 2017.

[10] Jamie Cool. Introducing azure devops. `https://azure.microsoft.com/en-us/blog/introducing-azure-devops/`, 2018.