MASTER IN COMPUTER SCIENCE

Reverse Engineering Variability for Configurable Systems using Formal Concept Analysis
The Odoo case study

EL IDRISSI, Zakaria

Award date:
2022

Awarding institution:
University of Namur

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

Reverse Engineering Variability for Configurable Systems using Formal Concept Analysis: The Odoo case study

Zakaria EL IDRISSI

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2020–2021

Reverse Engineering Variability for Configurable Systems using Formal Concept Analysis: The Odoo case study

Zakaria EL IDRISSI

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Xavier DEVROEY

Co-promoteur : Gilles PERROUIN

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Abstract

Reverse Engineering a Feature Model (FM) of an existing system, allows its migration to a software product line approach in order to simplify the management of this system by applying a Software Product Line Engineering methodology that focuses mainly on the FM in order to determine the reusable artifacts and the variation points of the system. This thesis is a case study on the Odoo framework to define a reverse engineering approach that can drive an automatic synthesis of an FM to represent the variability architecture of the system. We executed a manual exploration of the Odoo framework source code to identify variability patterns, then exploited Formal Concept Analysis properties to derive the FM based on the Odoo module's dependencies. The heuristic that we executed for the process of reverse engineering is effective and results in FM, which describes the product configuration variability.

Acknowledgements

I want to thank all those who have been supporting me in pursuing my studies, in particular:

To my parents, who always supported me and gave me the strength to fight and win in life with honesty;

To Jessica Friart for her love, encouragement, and understanding in difficult times;

To my beautiful girl Naysam for all the happiness she brings me;

To all the staff of the University of Namur for caring to provide us with a quality education;

To the professors who contributed to this work. Dr. Xavier DEVROEY and Dr. Gilles PERROUIN for their advice, competence, availability, and knowledge transmission.

Contents

List of Figures

List of Tables

Code Listings

List of Abbreviations

SPL	**S**oftware **P**roduct **L**ine
SPLE	**S**oftware **P**roduct **L**ine **E**ngineering
FCA	**F**ormal **C**ontext **A**nalysis
RE	**R**everse **E**ngineering
ERP	**E**nterprise **R**esource **P**lanning
CRM	**C**ustomer **R**elationship **M**anagement
MVC	**M**odel **V**iew **C**ontroller
OOP	**O**bject **O**riented programming languages **P**aradigms
ROI	**R**eturn **O**n **i**nvestment

Chapter 1

Introduction

A software product line (SPL) is a collection of products created from a set of reusable parts. Instead of delivering a single, standardized product, manufacturers have turned their attention to diversification, meaning the ability to offer several products tailored to different market segments, including products for niche markets [1].

Software product line engineering (SPLE) is a methodology that allows the development of multiple software products or systems with significant savings in cost, time, and quality [2]. Furthermore, reduce code duplication, reuse, and maintenance effort [3].

However, adopting the SPLE methodology requires a significant investment before achieving benefits. That is why software companies do not rely on SPLE as a first methodology to guide their new product development. Instead, they opt for ad-hoc solutions such as clone and own to meet the multiple requirements of their first customers, assuming the entire risk attached to the use of such ad-hoc techniques. However, in the long term, as the systems grows in size and complexity, adding a new feature requires much modification, and the maintenance requirements become overwhelming. This situation forces system designers to start adopting the SPLE methodology to manage the scalability and variability of their systems. For this purpose, several Reverse Engineering (RE) techniques exist to enable migration to SPLs.

A SPL adoption consists of mining the features and variation points that the system offers and how they are coupled (e.g., require dependencies or the exclude/alternative relationship between features). Furthermore, manual migration is time-consuming and error-prone, so research has been done on the automatic synthesis (reverse engineering) of Feature Model (c.f, Section 2.1.3). Several methodologies and tools facilitate and lead the automation of this feature extraction from existing software variant descriptions.

1.1 Thesis goal

Initially, a manual investigation of the different artifacts of the Odoo framework was performed, which allowed getting familiar with the architecture. Next, research on the mechanisms of realization of the variability was done,

which allowed us to note the use of specific mechanisms to implement the variability within the framework. Finally, an approach based on Formal Concept Analysis (FCA) (c.f., Section 2.3.2) for reverse-engineering the system was selected and executed on four variants that the system proposes. This approach resulted in extracting a subset of the variability of the system as a Feature model.

The dissertation explores three research questions :

- **RQ1** : *How does Odoo platform realizes Variability?*
  Does this question verify whether the variability proposed by Odoo has been realized through variability implementation techniques such as the mechanisms cited in Section 2.2.2, or is there a variability not previously cited in the literature?

- **RQ2** : *How can Formal Concept Analysis support dependencies extraction within Odoo frameowrk modules?*
  The answer to this question will determine an approach of a repeatable process that will allow the execution of a reverse engineering process to generate a feature model of the Odoo framework automatically.

- **RQ3** : *What is the variability model of the Odoo framework?*
  RQ3 is the main question of this thesis. By Answering the research question RQ2, we will produce a well-defined heuristic for extracting the dependencies of product variants. Based on these dependencies, we will be able to identify relevant information about the features of modeling a feature model.

Chapter 2

Background

2.1 Developing a Software Product Line

A Software product line (SPL) is a set of closely related software systems, with systematic reuse of assets to build variants, which allows the customization of the software on a large scale. The SPL development methodology is fundamentally different from traditional single-system development. This difference is driven by the strategy of considering an entire specific domain rather than just an ad hoc view of an individual project. Moreover, Software product line engineering is based on a fundamental distinction between development for reuse and development with reuse [4].

To this end, software product line engineering (SPLE) has been proposed as a specific development model consisting of two parts [2]:

1. **Domain Engineering** (development for reuse), where the domain model is analyzed, and reusable artifacts are created.

2. **Application Engineering** (development with reuse), where, based on a configuration, the domain artifacts are composed and sometimes completed to derive a concrete variant.

Figure 2.1 provides an overview of domain engineering and application engineering, and we present them in detail in the following subsections.

2.1.1 Domain Engineering

Domain engineering analyzes the domain of a product line and develops reusable artifacts. The main goal of domain engineering is to prepare the artifacts for use in the product line rather than developing a specific use case software. In the domain engineering process of a software product line, engineers conduct a domain analysis. Then, reusable artifacts are developed based on the domain knowledge that can be configured to derive different products or applications of a software product line [1].

So, domain engineering aims at development for reuse, and the product line infrastructure is made of all components that are relevant throughout the software development life cycle [4]. It has three key objectives:

- The set of software products should be defined using all the knowledge from the domain analysis.

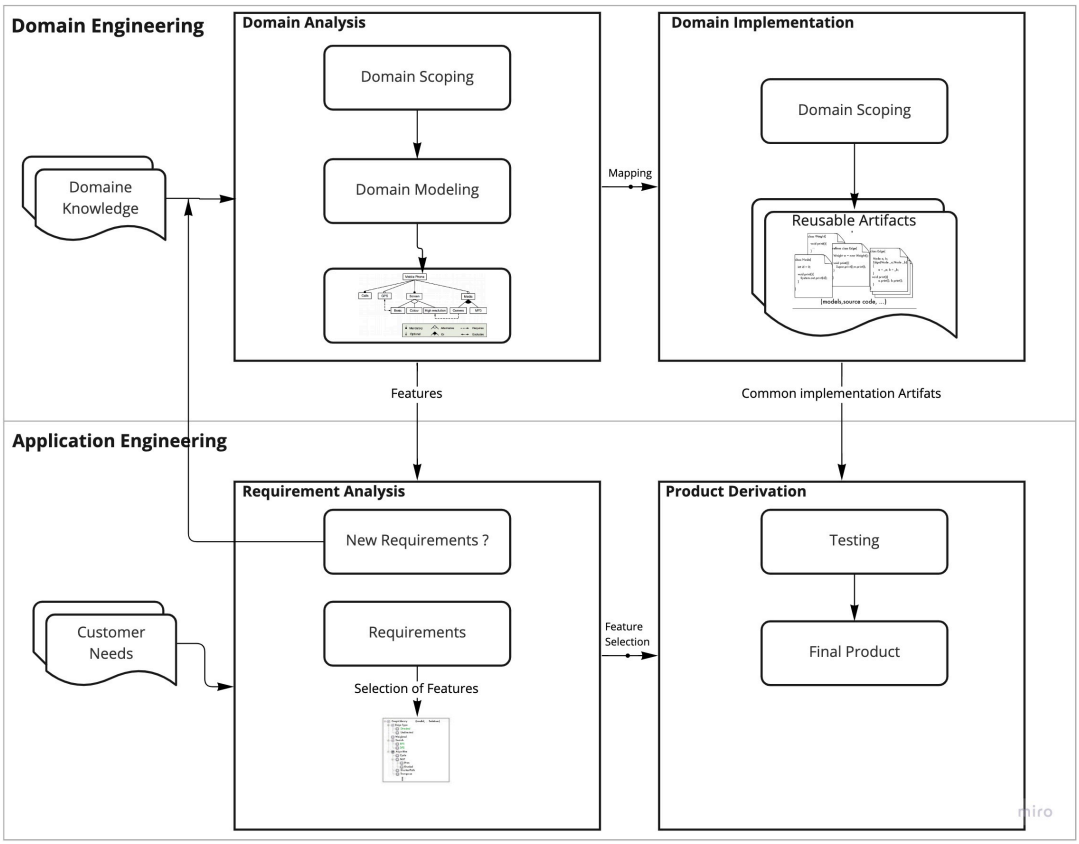


FIGURE 2.1: An engineering perspective on software product lines.

- The commonalities and variability of the software product line should be identified and modeled as a feature model.
- Reusable artifacts should be developed to realize the variability and develop a specific product for the needs of a particular customer during the application engineering phase.

In order to achieve the above objectives, domain engineering mainly consists of two parts: *domain analysis* and *domain implementation*.

Domain Analysis

Domain Analysis is a form of requirements engineering for an entire product line. First, this analysis, performed by domain experts, helps determine which product features correspond to a particular product line. Furthermore, which features are relevant and should be implemented as reusable artifacts. Second, the domain analysis results lead to a feature model to document and present information about commonalities and variabilities [1]. As we hold caught, domain analysis comprises two primary tasks: domain scoping and domain modeling.

Domain Scoping In the software product line, the domain scoping is driven by business goals, and this tracing is performed by domain experts who both decide what is included or excluded from this SPL cluster [1], [2].

Also, the domain scoping describes all common and variable features that are desired for future products. In particular, some features may also change for future applications given changing market demand and technology, highlighting the vital role of domain experts in preventing any potential evolution. Experts should explore what exists (e.g., a concurrent product, interviewing potential customers). In concrete terms, everything can provide sufficient knowledge of the domain [1].

Domain Modeling Domain modeling is a process comparable to requirements engineering; it is conducted for the entire software product line. During this process, the domain engineers separate the commonalities (which are common) and the variabilities (which is variable) across the desired products to present a feature model that captures all the relevant information in terms of relationships and constraints between these features. So the modeling domain includes two essential steps: commonality analysis and variability analysis. Commonality analysis analyzes the common elements that form the basis of the entire software product line. If there is more commonality than variability, the design process will require less effort. Variability analysis aims to extract and define the variation points by anticipating potential variants. Then all of this information about commonality and variability is systematically documented and used to build a feature model generally[1].

Domain Implementation

Domain implementation implements the various artifacts evaluated as reusable in the domain analysis process. We can then establish the traceability links between the features of the feature model and the implemented artifacts. The implementation of the domain is mainly based on selecting implementation strategies, such as the mechanisms for realizing variability. The choice of these mechanisms determines the implementation of the common parts and the points of variation. In this way, building different products stands for a simple configuration of reusable artifacts [1].

2.1.2 Application Engineering

Requirements Analysis

Requirements analysis in software product lines can be similar to requirements analysis in traditional software development. Requirements analysis in SPLs aims to study and analyze the requirements of a specific customer. However, the main difference is that we have already gathered the domain knowledge during the domain analysis, so potential requirements have already been identified and documented in a feature model during the domain analysis

process. The resulting FM allows the analysts to match the customer's requirements with existing functionality. In some cases, according to the customers' needs, the requirements analysis also highlights new requirements requested that miss in the previous domain analysis and its FM. In this situation, they can feed these new requirements back into the domain analysis and then adapt the feature model and the corresponding implementation artifacts [1].

Product Derivation

At this step, we have a selection of functionalities resulting from requirements analysis. We also have a set of readily reusable artifacts resulting from the domain implementation. Finally, to derive the desired product (Product derivation), we must choose an implementation approach (cf. Chapter **??**) to combine the different artifacts. The product derivation can be done in a manual or automated (push-button) fashion. However, the manual mode is usually time-consuming, or developers have to write some glue code to connect the artifacts and cover the missing gaps for which there are no reusable artifacts. On the other hand, automating the derivation has some exciting advantages:

- reduction of costs linked to the derivation;
- adapting the final product to several specific use-cases;
- When a given artifact evolves, or a bug fix occurs, the automation puts all the artifacts back together in their current state.

The resulting product must be validated before delivery in automatic or manual derivation cases. Validation is the last step of application engineering and is performed with automated unit tests derived from the artifacts provided during domain engineering [1].

2.1.3 Feature Model

As part of this Software product line engineering (SPLE) process, the feature model, as shown in the figure 2.2, is typically used to identify commonalities and differences among related systems, usually expressed through features and their relationships (i.e., constraints and dependencies) [1]. A feature is generally the software system's functionality, and it is visible to the end-user of a software system [5], [6]. Features play an indispensable role in the feature model because features can identify commonalities and differences between products throughout the software lifecycle. Each feature can fulfill a requirement and provide potential choices for configuration. When using a feature model in SPLs, variants can be derived by selecting features that meet specific requirements. Feature selections must satisfy the dependencies presented in the corresponding feature model.

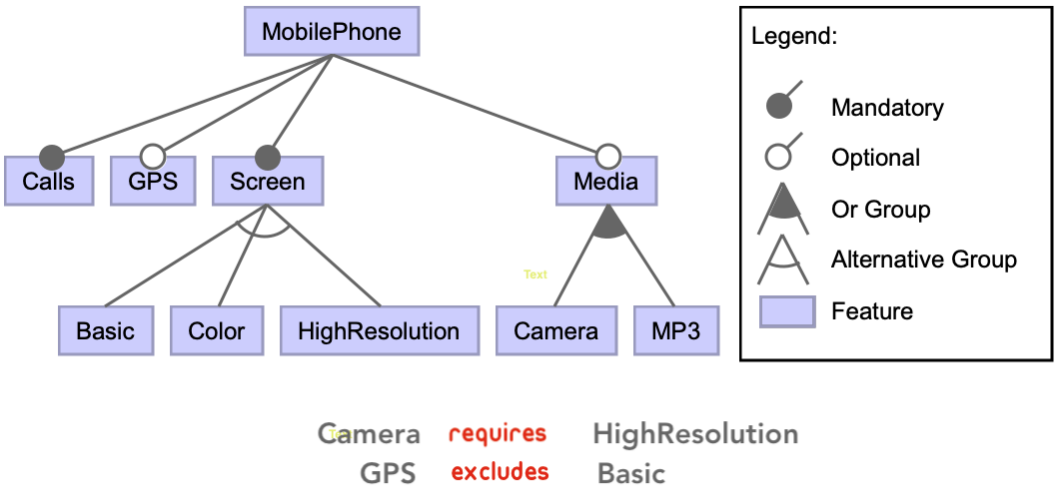


FIGURE 2.2: A sample feature model of a Cell Phone.

Feature modeling is, an essential step in SPLE as part of domain analysis, where domain engineers manually analyze requirements to identify commonalities and differences (i.e., differences) between products in the domain.

Mandatory: the sub-feature must be included in every possible configuration;

Optional: the sub-feature may be included in the configuration;

OR: one or more sub-features can be included in the configuration;

Alternative (XOR): only one sub-feature should be included in the configuration.

In addition to the above relationships between parent-child features, cross-tree constraints also express dependencies between features in the feature model. The most common ones are as follows:

A requires **B**: The occurrence of feature **A** implies the occurrence of feature **B**;

A excludes **B**: The occurrence of feature **A** implies the absence of feature **B**.

2.2 Variability

Software variability is the ability to change or customize a system [7]. The variability within an SPL will allow the representation of the differences and

the common elements between different products of this SPL. In contrast, the variability of a software product refers to its ability to be customized, modified, extended, or configured. This possibility can be offered by well-known software engineering mechanisms such as interfaces, class abstractions, conditional compilation, or dynamic class loading [8]. The variability of software product lines is modeled to facilitate the development of customized applications by reusing artifacts [2].

2.2.1 Taxonomy for classifing Variability

In Section 2.1.2, we discussed that product derivation is supported by variability mechanisms to derive different products from a set of features selected according to the customer's need. So it is necessary to choose the most appropriate mechanism for the specific situation correctly. With this in focus, Apel et al. (2013) describe three classification criteria to guide the choice of a given mechanism including : binding time, technology, and representation.

Binding Time is a variability classification criteria that decide the time when features should be included in the software of the final product. This process is determined either before or at compile time, load time is decided after compilation when the program is started, and runtime, where the latter variability decisions can be performed and changed during program execution [1].

Each binding time has its advantages and disadvantages. For example, early binding time resolves the variability configuration upstream of the process and can allow optimal efficiency by reducing runtime overhead and memory consumption. However, once the software is created and installed, it is no longer possible to be variable [1], [9]. On the other hand, load or late binding time offers flexibility and dynamic system adaptation[10]. Because instead of recompiling a new product after each change, users can change the settings and restart the same software. In the case of link-time execution, there is also no need to restart the software, and it can change while the program is running.

However, both mechanisms have a memory and performance overload because all the variations are compiled into a single binary code to avoid any inconsistencies occurring at runtime [1].

The technology criteria have both language-based and tool-based approaches. Approaches that use mechanisms provided by the programming language, then implement and manage the variability features are made in the software source code. On the other hand, the tool-based approach uses one or more external tools to implement features in the code and control the process of product derivation [1].

Finally, the third criteria are representation by ***annotation and composition***. In technology-based approaches, all features are inserted and tagged in code. During the software derivation process, code that belongs to untagged

	Binding Time		Technology		Representation	
	Compilation	*Loading*	*Language-Based*	*Tool-Based*	*Annotation*	*Composition*
Parameters		X	X		X	
Design Patterns	X	X	X			X
Frameworks	X	X	X			X
Components	X	X	X			X
Version Control	X			X		X
Build Systems	X			X	X	X
Preprocessors	X			X	X	
Feature-Oriented programming	X	X	X			X
Aspect-Oriented Programming	X	X	X			
Separation of Concerns	X			X	X	

TABLE 2.1: Classification of Variability Implementation Techniques in SPL [1]

functionality is either deleted at compile time or ignored at run time. Note that the annotation approach supports negative variability, so removing the code on demand is possible. Thus, the growth in adopting this approach is simple because it is easy to use; also, programming environments offer native support for this technique. Composition-based approaches, such as frameworks and components, identify the code associated with a feature or a combination of features, a container or a module, and implement it as composite units, ideally one unit per feature. In the product derivation process, all units of all selected features and valid combinations are composed to create the software. As a result, this approach to composition promotes positive variability and adds code on-demand [1].

Table 2.1 shows a classification of variability implementation techniques in SPLs against the classification criteria. Each technique is composed of more than one criteria.

2.2.2 Variability Mechanisms

Apel et al. [1] proposed a classification of variability implementation techniques under four main groups: Binding-Time, Technology (Language-Based Versus Tool-Based), and finally, Representation (Annotation Versus Composition). In addition, an interesting dimension of classification is noted by Tërnava and Collet. (2017), which is the Traditional or Emerging techniques[11]. The following Section 2.2.2 presents a definition and explanation of implementing these techniques for realizing variability. We grouped the techniques into two explicit categories, language-based and tool-based. At the same time, other dimensions are mentioned but not explicitly.

Language-Based Variability Mechanisms

Language-Based Variability mechanisms are techniques that share a common characteristic that can be implemented in the most common programming languages. Also, they are well-known and commonly used mechanisms in practice. Moreover, most SPLs have implementations of variability with this set of techniques and focus mainly on run-time binding. We will introduce

the following mechanisms: Parametrization, Cloning, Inheritance, Design patterns, Frameworks, and Components & Services. Besides, we will highlight the distinctive properties to justify the classification.

Parametrization : is a traditional and straightforward way to implement variability through conditional structures. These conditional structures (e.g., *if-else*, *switch*) modify the execution flow. For example, conditional statements determine a flow based on the parameters passed to methods or modules during the program's execution. The result is an immediate effect or, in some cases, requires a restart of the software [1]. In addition, developers often use parametrization to implement run-time variable features, supporting run-time binding [10]. After the compilation, the technique does not allow adding new variants [12]. Usually, the configuration parameters are globals variables of boolean type; the conditional structure will evaluate these parameters to determine the flow. However, this discourages modular solutions. In contrast, parameterization facilitates the traceability of features due to renaming conventions of parameters [1].

Cloning **:** Tërnava & Collet insisted on not excluding Cloning from their classification of techniques because they consider it one of the most applied techniques, and it is a traditional mechanism for the realization of variability [11]. Also, Zhang et al. points out that Cloning is one of the most used mechanisms in the industry. Specifically, it consists of copying an artifact (code or non-code) and evolving this copy without connecting with the original artifact [10]. Two techniques of Cloning exist:

- Moving the physical copy of the artifact to another location;
- The configuration of the branch management.

Cloning is chosen as a variability mechanism when existing constraint on the delivery time and maintainability is not a priority [13]. Second, it offers independence and does not represent a risk to existing variants. Finally, Cloning is used as a prototype to evaluate an experimental functionality [10].

Design patterns : Design patterns can be applied in SPLs because several design patterns provide solutions for managing the variable aspect. Observer, Template-method, Strategy, and Decorator are suitable for implementing variability [1], [14]. Design patterns rely on object-oriented languages and use mechanisms like inheritance and polymorphism to implement variability [12]. Instead, they capture the project's intent by identifying the distribution of objects, interactions, and responsibilities [1].

Frameworks : A framework technology consists of reusable and flexible base structures that can be extended and adapted to solve related problems by supporting granularity and providing explicit points for extensions, often named plug-ins, into which developers can expand it [1], [7].

Currently, frameworks with plug-ins are the most developed in SPL because each feature corresponds to a plug-in constructing the final program according to feature selection [1]. Frameworks as composition mechanisms have practical benefits like easily identifying variants and do not constrain efficiency because they are executed at run-time [10]. This technology has proven a better evolution in SPL than other mechanisms [15], and code reusability is maximized [14]. However, how the Framework provides extensibility makes the Framework be of type white box or black box.

The *White Box Framework* consists of concrete and abstract classes, where developers implement or override in a subclass extending the Framework. White box because the development team has to identify the methods and understand the internal components of the Framework [1]. So, there is a trade-off between flexibility when adding extensions and the effort of adapting to understand the system, as the White Box Framework allows for modification of existing behavior from additional implementations for future extensions. However, developers need a detailed understanding of the Framework. Each feature in SPL is considered an extension (a subclass of a given class). However, white-box frameworks are more suitable for implementing alternative features, and their main difference is that the frameworks can be developed and provided by third parties.

Black-box frameworks catch strategy pattern and observer pattern standards by separating the code between Framework and extensions via interfaces. Those frameworks are called plug-ins and are also known as black box because developers need to know only the interface, dropping the need to understand the internal implementation of the Framework. Developers can only add plug-ins at the access points provided in the framework structure, so there is a limitation of flexibility. On the other hand, this limitation allows for the decoupling of extensions, leading to an easier understanding and application of the Framework. Theoretically, in SPL development, each feature is implemented as a plug-in and then combined with the Framework, allowing automation in the final product generation. The Framework and plug-ins have an independent evolution as the plug-in interfaces remain unchanged [1].

Components and Services : The Component is a unit of compositions and provides functionality through an interface, and we can consider a component as a feature. Its internal implementation is encapsulated and forms a modular and reusable unit to be considered a feature. When variability is realized via components, we rely on a composition approach, and the Component may be composed of other components in different combinations [1]. To build a program, developers can implement and deploy their components independently and, if necessary, compose third-party components because a component is independent of a specific application or product line. We can compare components with plug-ins only if their interfaces are designed to the same standards. In SPL, determining when to create a reusable component is an important design decision made

in the domain analysis that helps decide how to divide the code into components. If a reusable component provides much functionality, maybe it is easy to integrate and use, but eventually, the Component may not fit into some applications. In contrast, we can create small reusable components that can be flexibly combined, but many connections need to be made between the components and the base code; this becomes discouraging. Consequently, developers need to balance deploying a component that provides functionality but is small enough to be reused in many contexts. The similarity between a service and a component is the same in encapsulating the feature's functionality behind an interface, but standardization, interoperability, and distribution are valuable factors of the technique. Another differentiating factor is that services written in different languages can interact because communication is standardized through protocols or conventions [1].

Tool-Based Variability Mechanisms

After presenting a set of variability implementation techniques based on the concepts of programming languages, we will now study mechanisms based on external tools to enable the implementation of variability in SPLs. We will describe the following mechanisms: Aspect-Oriented Programming, Feature-Oriented Software Development, Preprocessors, Build Systems, and Version-Control Systems. Tool-based mechanisms generally target compile-time binding, whereas programming language-based mechanisms focus mainly on run-time binding.

Aspect-oriented programming is a language and composition-based approach to implementing variability in SPL [1]. Aspect-oriented programming is a concurrent technique to develop feature-oriented product lines. However, it is used principally for code tracking, logging, and exception handling [16]. The approach is to develop the software to work generically and then superpose of the product-specific concerns. Different variability concerns are woven into the source code just before the code is compiled [17] cited in [12]. Different weaving technologies support different binding times, including compile-time binding and load-time binding [1]. In practice, programming languages do not offer this functionality by default, and external tools such as AspectJ[1] allow this technique [10]. The straightforward approach is to implement one aspect per feature. Based on the selection of a feature by the user, then the complementary aspects are included in the weaving process, possibly controlled by a construction system.

Feature-oriented Software Development is a composition-based approach to implementing variability in SPLs that relies directly on the notion of features [1].

[1] https://www.eclipse.org/aspectj/

Definition 2.2.1 *"A feature module encapsulates changes that are made to a program in order to add a new capability or functionality. Such modules (often interpreted as transformations) are composed sequentially. If* f *and* g *are feature modules, their composition* $f \cdot g$ *represents the combined set of changes made by* f *and* g*." [18]*

In Feature-Oriented Software Development, programs are generated by composing modules that implement features. The client selects precisely the features, and then a generator composes the modules implementing the corresponding features to build a concrete program. The feature modules can be composed statically at compile time or even dynamically at run time [19].

Conditional compilation is one of the most common mechanisms for implementing variability in software product lines. Moreover, it is a mechanism that has proven itself in the software industry [10]. Conditional compilation is performed with the help of a tool named a preprocessor that manipulates the source code before compilation [1]. The preprocessor allows wrapping fragments of code by using the macros, including or excluding optional or alternative code before the compilation according to the conditional compilation directives defined by the user, such as *#ifdef* and *#endif*. We say that such a code fragment is annotated, making conditional compilation an annotative technique. This technique allows implementing variability in software product lines, especially in embedded systems [20]. Also, large open-source systems (e.g., Linux kernel) have adopted this mechanism to implement variability [21], [22]. Most programming languages offer this mechanism, but developers often implement conditional compilation with C/C++ preprocessor by *#ifdef #ifndef* as shown in Figure 2.3(a) [10] or with java using *Munge*[2] preprocessor by similar syntax using feature directive annotation { *IF ... END* } representing the variation as shown in Figure 2.3(b) [1]. Another option is to use a principal file as a decision template as shown in Figure 2.3(c), which includes decisions from other files that correspond to the source files. In this way, a hierarchy of decision files can be constructed [14].

A build system performs all the tasks involved with planning and executing source code, including running generators, compiling source code, running tests, and creating the final deliverable. There are many different build systems with different levels of sophistication. In the simplest case, a build system is usually a shell script that executes the necessary tools according to predefined parameters, as shown in Figure 2.4(a) example without variability and with variability in the Figure 2.4(b). However, more sophisticated systems, such as *Make*[3], *Ant*[4], and *Maven*[5], that support

[2]https://software.opensuse.org/package/munge-maven-plugin
[3]https://gcc.gnu.org/wiki/HomePage
[4]https://ant.apache.org
[5]https://maven.apache.org

```c
static int __rep_queue_filedone(dbenv, rep, rfp)
  DB_ENV *dbenv;
  REP *rep;
  __rep_fileinfo_args *rfp; {
#ifndef HAVE_QUEUE
  COMPQUIET(rep, NULL);
  COMPQUIET(rfp, NULL);
  return __db_no_queue_am(dbenv);
#else
  db_pgno_t first, last;
  u_int32_t flags;
  int empty, ret, t_ret;
#ifdef DIAGNOSTIC
  DB_MSGBUF mb;
#endif
  // over 100 lines of additional code
#endif
}
```

(a) Conditional compilation with the C preprocessor [1].

```java
class Node {
  int id = 0;
  /*IF[FEAT_COLORED]*/
  Color color = new Color();
  /*END[FEAT_COLORED]*/
  Node (int _id) { id = _id; }
  void print() {
    /*IF[FEAT_COLORED]*/
    Color.setDisplayColor(color);
    /*END[FEAT_COLORED]*/
    System.out.print(id);
  }
}
```

(b) Conditional compilation with the Java preprocessor [1].

FileA

```
#define FileA
#define PartA
/* standard code for PartA */
#include DecisionModel
/* standard code continues */
#undef PartA
/* rest of the code follows */
```

DecisionModelA

```
#ifdef PartA
#ifdef Functionality2
/* code for Functionality2 */
#endif
#endif
```

DecisionModel

```
#ifdef FileA
#include DecisionModelA
#endif
#ifdef FileB
#include DecisionModelB
#endif
```

(c) Conditional compilation directives with Decision Model [14].

FIGURE 2.3: Conditional compilation implementation example.

multiple build targets manage dependencies, avoid unnecessary recompilation to optimize performance, automatically download and update dependencies, and create build reports. The Build System is an evident candidate for managing variability, particularly and naturally in the build-time phase [1].

Version control systems are a category of development tools that supports teams in managing source code changes over time. To facilitate collaborative development, version control systems track all performed changes in source code, especially other development artifacts. Popular examples are *Git*[6] and *Mercurial*[7]. An important feature of version control systems is creating different branches of the same file. These different branches can be modified independently, where changes applied over time

[6] https://git-scm.com/about/free-and-open-source
[7] https://www.mercurial-scm.org

```
#!/bin/bash -e

rm *.class
javac Graph.java Edge.java Node.java \
      Color.java
jar cvf graph.jar *.class
```

(a) Build script for the graph example without variability [1].

```
#!/bin/bash -e

if test "$1" = "--withColor"; then
  cp Edge_withColor.java Edge.java
  cp Node_withColor.java Node.java
else
  cp Edge_withoutColor.java Edge.java
  cp Node_withoutColor.java Node.java
fi

rm *.class
javac Graph.java Edge.java Node.java
if test "$1" = "--withColor"; then
  javac Color.java
fi

jar cvf graph.jar *.class
```

(b) Build script for the graph example with variability [1].

FIGURE 2.4: Build script example

in one branch do not affect the files in other branches. In this case, we speak of variants of the same file. Instead of revisions over time, developers use versioning differently as a patch to create variants of the same file, sometimes called variation in space. Variants are not ordered and do not replace each other; they exist in parallel. Each variant can have an independent revision history. As illustrated in the graph in Figure 2.5, developers create new branches from the main branch that hosts the shared code and develop customer-specific feature variants on the new branches when realizing variability. Note that the added features are not merged into the main branch. Instead of developing each product in a different branch, we can also implement each feature separately and create products by merging the corresponding feature branches. Figure 2.6 illustrates this model where developers create one branch per feature to implement [1].

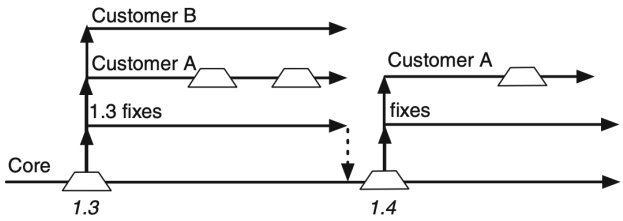


FIGURE 2.5: Branch per variant product [1]

2.3 Software Product Line Methodologie Implementation

Section 2.1 characterized the two main processes of Software Product Line Engineering, i.e., Domain Engineering and Application Engineering.

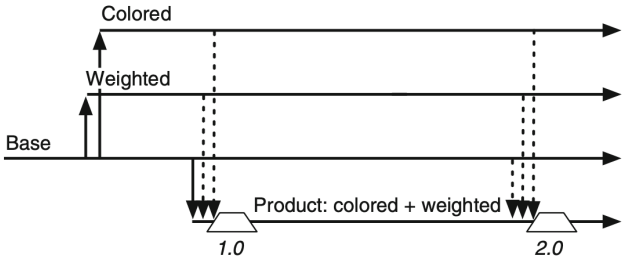


FIGURE 2.6: Branch per feature[1]

Moreover, we have specified that the Application Engineering process takes place after completing the Domain Engineering. This constraint imposes to have the first variants after the Domain engineering, which creates a limitation to forming an SPL and starting deriving software variants.

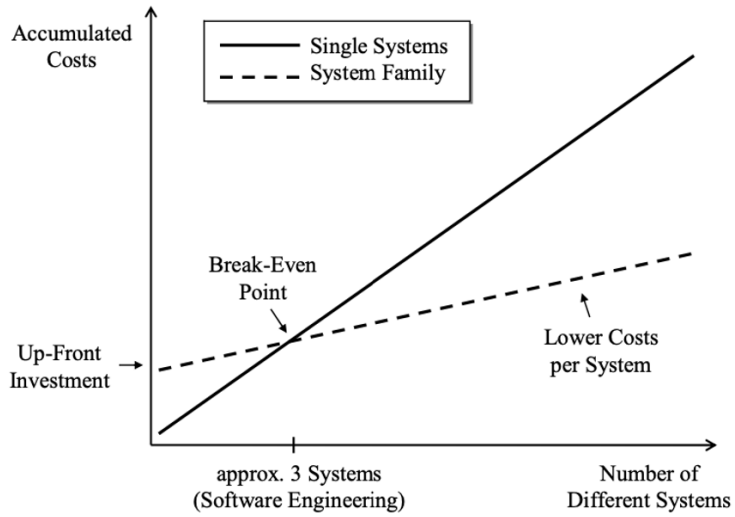


FIGURE 2.7: Development costs of n single system versus product line engineering development [2]

However, the advantages of reuse can be appreciated only after forming an SPL with some software variants. Indeed, adopting an SPL methodology requires a solid investment because the benefits take some time to be seen, maybe years before being exploited [23]. In the literature, it is considered that developing an SPL methodology becomes advantageous after completing the development of 3 variants [2]. This theory is illustrated in Figure 2.7, which distinguishes the classical single system development from the SPL development according to their costs to the number of variants developed

The complexity imposed by the SPL development process leaves professionals with no option than to develop individual variants, especially

for new projects, to deliver functional products more efficiently to different stakeholders. Then when the complexity of maintaining these products increases, engineers turn to adoption strategies of SPL in a second place. In terms of these strategies, we have the proactive adoption strategy, which follows the two processes defined in Section 2.1; also, the reactive adoption strategy is adopted when the requirements of the SPL to be built are not well defined in advance. In this case, the new requirements will be sent to the domain engineers to analyze and extract the corresponding features. Finally, the extractive adoption strategy consists of reverse engineering an SPL from existing variants.

The extractive adoption strategy seems to be the most popular: it assembles an SPL based on existing variants. These variants are usually documented in a variability model defined during the domain engineering, such as FM. Therefore building such a model for a software product family is the first step of migration [23].

However, manually building such a model is a time-consuming, fallible and challenging process, even with a few variants. Several works have been done on mostly open-source projects to migrate to an SPL, such as Halin et al. built a product chain that considers variability on top of an open-source industrial web-app configurator JHipster [24]; Al-Msie'DeenRaFat et al. presented an approach called **REVPLINE** (REVPLINE stands for RE Software Variants into SPL.) to identify and document features from the object-oriented source code of a family of product [25]; Shatnawi et al. performed an approach to reverse engineering a set of product variants to identify architecture-level variability and dependencies [26].

2.3.1 Reverse Engineering

As we discussed, the difficulty of maintaining these products when they become massive and complex leads to migration to SPL-type approaches [23]. Furthermore, to facilitate this migration, research has been done on the automatic synthesis (reverse engineering) of FMs. Several methodologies and tools facilitate and lead the automation of this feature extraction from existing software variant descriptions. Most of the approaches to performing reverse engineering of FMs are based on high-level models such as product descriptions and requirements; Other approaches deal directly with low-level artifacts such as source code. Some approaches offer an acceptable solution but cannot identify essential parts of the feature model, such as cross-tree constraints (require and exclude), the **AND** group, the **OR** group, and the **XOR** group.

Reverse Engineering approaches

The diversity of the approaches used to reverse engineering FM from a family of software product variants shows that we present the existing RE approaches in the literature presented in chronological order below.

- **Weighted graph clustering** is a semi-automatic approach based on requirements clustering to build FMs from the functional requirements of applications. This synthesizing technique requires using the Weighted graph clustering theory to describe the relationships between requirements [27].
- **The model-driven** approach is a transformation process that matches use cases to functionality [28].
- **Formal concept analysis** (FCA) it is mainly used to derive implicit relationships between objects described by a set of attributes and shows its importance for many applications in the industry, like information retrieval and classification. Also, AL-MSIE'DEEN et al., [29] proposes an approach (REVPLINE) based on FCA to mine features and feature models from the object-oriented source code. [30] [26] .
- **Text similarity** approaches combine two distinct sources of information: textual feature descriptions and feature dependencies in propositional formulas [31].
- **User input** approaches extract FMs from product/feature configurations that contain variability [32].
- **Meta-heuristics** exist to obtain models from all the possible configurations automatically. This method first proposes identifying the root of the characteristics available in all the variants, and then the rest of the model is generated recursively from top to bottom [33].
- **Intermediate representations of variability** is an approach that uses mutex graphs to extend the approach of Czarnecki and Wasowski in [34] with methods using the mutex graph of a propositional formula [35].
- **Dedicated algorithms** include evolutionary algorithms, hill-climbing, and random search [36].
- **Extraction and Evolution of Architectural Variability** is a tool-assisted approach to extracting and managing the evolution of software variability from an architectural perspective [37].
- **But4Reuse** (for bottom-up technologies for reuse)is a set of tools that bring different tasks for reverse engineering of SPL [38].

2.3.2 Formal Context Analysis

Out of all the RE approaches described in Section 2.3.1, we will dive deep into the FCA because, afterward, this concept will be adopted to implement the RE approach on the Odoo ERP.

Formal Concept Analysis is a field of mathematics that emerged in the 1980s [39], [40]. Its main feature is knowledge representation by providing specific

	Blue	Yellow	Red	white
Orange		X	X	X
Violet	X		X	X
Green	X	X		X
Brown	X	X	X	X

TABLE 2.2: The FCA table represents the color example

diagrams called lattice diagrams. The formal analysis of concepts brings extra value to the visualization because it allows focusing on the most interesting points during the variability analysis to the decision-making process [41], [42]. The following example given in Table 2.2 explains the FCA technique, and shows colors based on their primary colors. The objects represent the types of colors, such as $\mathbf{O} = \{Orange, Violet, Green, Brown\}$, and the attributes or properties represent the component colors, such as the set $\mathbf{A} = \{Blue, Yellow, Red, white\}$. The symbol X represents the existence of a relationship between an object and its attribute.

It is a principles-based approach to deriving a hierarchy of concepts from a collection of objects and their properties. A formal context is a triplet $(\mathbf{O}, \mathbf{A}, \mathbf{I})$, where **O** is a set with elements called objects,**A** is a set containing members called attributes, and $I \subseteq \mathbf{O} \times \mathbf{A}$ is a relation called the incidence relation. If $(\mathbf{O}, \mathbf{A}) \in \mathbf{I}$, we say that "***object o has attributed a***.".
This formal context $(\mathbf{O}, \mathbf{A}, \mathbf{I})$ will be transformed mathematically into a concept lattice structure with no information loss during this transformation. The concept lattice obtained after transformation can be visually represented to ease the communication about the formal context. Graph 2.8 represents the concept lattice belonging to the context of colors.

A two dimensions table usually represents a formal context. Objects are given in the row headers, attributes in the column headers, and a mark in row o and column a *if and only if* $(o, a) \in \mathbf{I}$.

Given a set of objects $\mathbf{G} \subseteq \mathbf{O}$ of a formal context $(\mathbf{O}, \mathbf{A}, \mathbf{I})$, the derivation operators defined as follows: $\mathbf{G}' \doteq \{a \in \mathbf{A} \mid \forall o \in \mathbf{A}(o, a) \in \mathbf{I}\}$ and $\mathbf{T}' \doteq \{o \in \mathbf{G} \mid \forall a \in \mathbf{G}(o, a) \in \mathbf{I}\}$. Allow to extract any attribute that is a member of **A** and is common to all objects of **G**; Similarly, for a set $\mathbf{T} \subseteq \mathbf{A}$, we can extract the objects with all attributes of **T**.

A formal concept is a pair $(\mathbf{E}, \mathbf{I}) \in \mathbf{O} \times \mathbf{A}$ such that **E' = I** and **I' = E**, where **E** is called the extension and **I** is the intention of the concept. The formal concept set has a partial order such that for any two formal concepts (**E1=I1**) and (**E2, I2**), we have **(E1, I1)** $\leq$ **(E2, I2)** if and only if **E1** $\subseteq$ **E2** (and in this case **I2** $\subseteq$ **I1**). The set of concepts ordered by $\leq$ constitutes a complete set of concepts [43] called the conceptual set.

The conceptual set obtained from a formal context **(O, A, I)** is labeled $\mathbf{B}(\mathbf{O}, \mathbf{A}, \mathbf{I})$ and the theorem on conceptual lattices states that a conceptual lattice $\mathbf{B}(\mathbf{O}, \mathbf{A}, \mathbf{I})$ is a complete lattice in which for every set $\mathbf{C} \subseteq \ \mathbf{B}(\mathbf{O}, \mathbf{A}, \mathbf{I})$

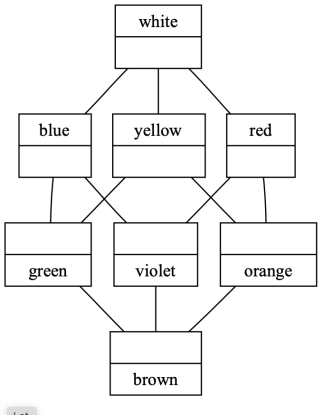


FIGURE 2.8: The concept lattice belonging to the formal context in table X

gives the *supremum* and *infimum* $\wedge\mathbf{C} = \left(\cap\mathbf{X}, (\cup\mathbf{Y})''\right)$ and $\vee\mathbf{C} = \left((\cup\mathbf{Y})'', \cap\mathbf{Y}\right)$, where $\mathbf{X} = \{\mathbf{E} \mid (\mathbf{E}, \mathbf{I}) \in \mathbf{C}\}$ and $\mathbf{X} = \{\mathbf{I} \mid (\mathbf{E}, \mathbf{I}) \in \mathbf{C}\}$ [39].

Chapter 3

Odoo in a Nutshell

Odoo previously **OpenERP** and Tiny **ERP**, founded in 2005 by Fabien Pinckaers, is an open-source Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM). Odoo includes many modules to meet a wide range of business management needs.

Odoo covers all common business needs, such as sales, purchasing, human resources, project management, logistics, inventory, manufacturing, and billing. In addition, its framework allows to adapt it to specific contexts, whether by customizing new business workflows, new specific domain information, or dashboards.

Odoo is driving dynamic innovation, with remarkable advances and new features in each major release, making it a prosperous solution functionally, which allows it to cover all needs and business applications in a specific domain.

Odoo Community is the core on which Odoo Enterprise is built, and the user can change the version at any moment. Odoo was distributed under the AGPL 3.0[1] license, but they have announced that they will change their license for Odoo v9 to * LGPL 3.0[2]. The main difference is that module developer now no longer have to provide the source code of their modules to their customers, which allows developers to have more control over their intellectual property [44].

Every year, Odoo S.A. ensures the release of a new version of its product. Currently, Odoo is in version 15 [45]. Their next version will be released in October 2022. The updates and new features will follow Odoo's existing strategy and vision.

Odoo is an object-oriented framework that relies on a three-tier architecture. It is written in *Python*[3] for all the backend logic, *Javascript*[4] and *CSS*[5] for the interface interaction with the user and website, and *PostgreSQL*[6] database for database maintenance and management. Odoo follows the Model View Controller (MVC) design model, splitting the design architecture into three

[1]https://opensource.org/licenses/AGPL-3.0
[2]https://opensource.org/licenses/LGPL-3.0
[3]https://www.python.org/
[4]https://www.javascript.com/
[5]https://www.w3.org/Style/CSS/
[6]https://www.postgresql.org/

complementary and dependent parts. We will go deeper into the architecture topic in the next section 3.1.

3.1 Odoo Framework Architecture Overview

After a brief introduction of the Odoo product and its commercial vision, we will dive deeper into the application's architecture in this section.

Firstly, The Github repository that hosts all the open-source applications of Odoo Community in its version 15 included 1.6 billion lines of code[7] . We can also observe many dependencies between modules and classes that make up the framework code source. As we can see in figure 3.1, there are many interdependent classes. Figure 3.2 shows the module that constitutes the core of the web layer and all the dependencies on this module. In addition to that, we have the framework's architecture designed in multi-layer Web, Back End, and database. The notion of reuse has guided the design of all these components described in Section 3.1.1.

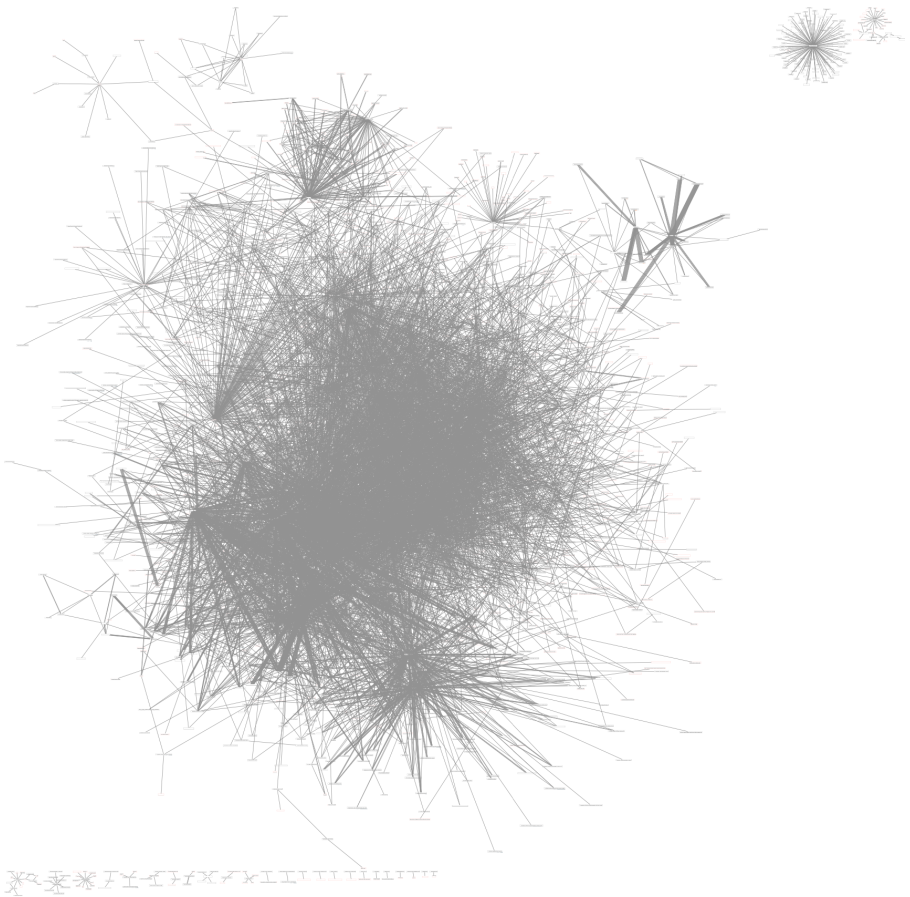


FIGURE 3.1: The PyCharm plug-in generates a static view of all the Odoo framework classes.

[7]https://github.com/odoo/odoo/

FIGURE 3.2: A zoom on the static view of the figure 3.1.

3.1.1 The logical architecture of the Odoo Framework

Odoo is software based on an MVC architecture, and this architecture is made out of three main components communicating together. As shown in figure 3.3, we have the three main components of the application, which are:

- **Model**: represents the PostgresSQL database and the functional logic of the application. The model is responsible for data management and storage; the model objects fetch and save the model's state in the PostgresSQL database system.
- **View**: represents the user interface of the application; through the view, the user accesses the data. However, all access to the database is via corresponding models. The view allows the user to retrieve information and apply changes to the database. There are several views: *form*, *tree*, *gant*, *kanban*, and *calendar*.
- **Controller**: the controller is coded in Phyton, and it manages both the views and the models; it receives instructions from the view and then sends the appropriate instructions to the model.

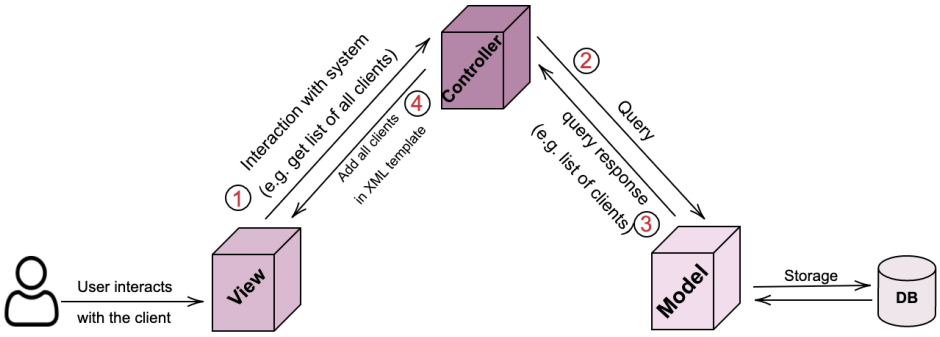


FIGURE 3.3: Odoo MVC architecture.

This architecture model offers different possibilities to interact with the controller and the business logic, as illustrated in figure 3.4. In this way, specific queries can be executed on models. This communication is mainly provided by *XML-RPC*[8] or *JSON-RPC*[9], which are methods to call remote procedures using *XML* or *JSON* via *HTTP*[10].

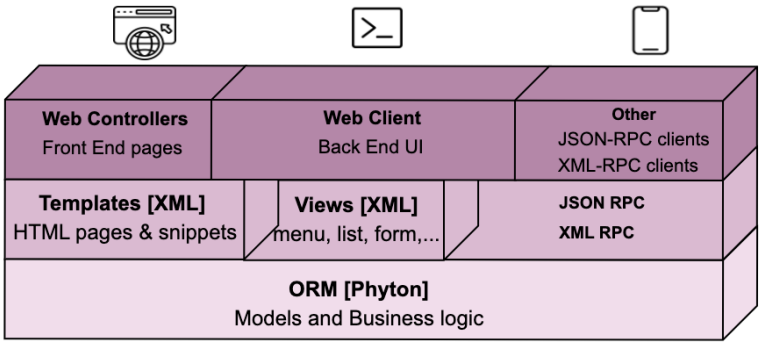


FIGURE 3.4: Communication logic of the Odoo framework

3.1.2 Odoo Modules Structure

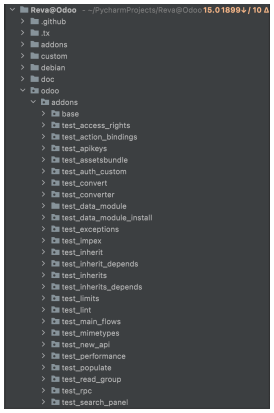
Odoo is based on the principle of using a modular and independent structure that allows both to regularly improve the existing modules and have the flexibility of modifying or deleting the useless modules without changing any part of the system.

[8] http://xmlrpc.com/
[9] https://www.jsonrpc.org/
[10] https://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html

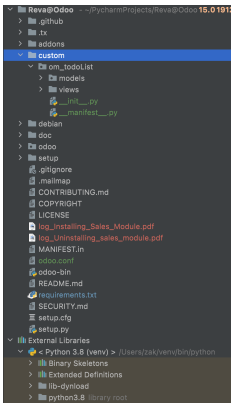
Odoo Modules

The framework by default contains several modules ready to be deployed or tailored in a customization approach. The modules are contained in a directory named */addons*[11] (see Figure 3.5(a)), and it is possible to install your modules in it, with the condition of following the standard layout of a module. However, it is preferable to use a custom folder (a separate addons folder), as shown in Figure 3.5(c). The fact that the addons are located in a single directory hides the dependencies and makes searching in the code more complex.

(a) The */addons* directory holds the modules available for deployment from the Odoo community.



(b) The */addons* directory is included in the Back End core of the framework.



(c) The custom directory that holds the tailor-made module.

FIGURE 3.5: The differents */addons* directory

Modules Structure

Odoo modules are organized as a directory containing different subdirectories and files according to a well-defined pattern. This convention is a blueprint to be followed by the developers in order to create new modules or customize an existing one. On the other hand, it increases the framework's maintainability, reliability, and extensibility.
So, to develop Odoo modules requires an understanding of the technical architecture of an Odoo module, as shown in Figure 3.6.

Three mandatory directories in this structure are required:

- The `__manifest__.py` file is a simple Python dictionary to specify the different metadata of the module such its name, dependencies,

[11] The framework contains another directory included in the Back End core, as shown in Figure 3.5(b).

description, composition, version, author, website, data files, demos, security, and more. Listing 3.1 shows an overview of what this file can hold;

- The `__init__.py` file is the initialization python file of the module containing all the other python files to import.
- The `models` file This is a directory that will contain all the python files that constitute the business logic of the module.

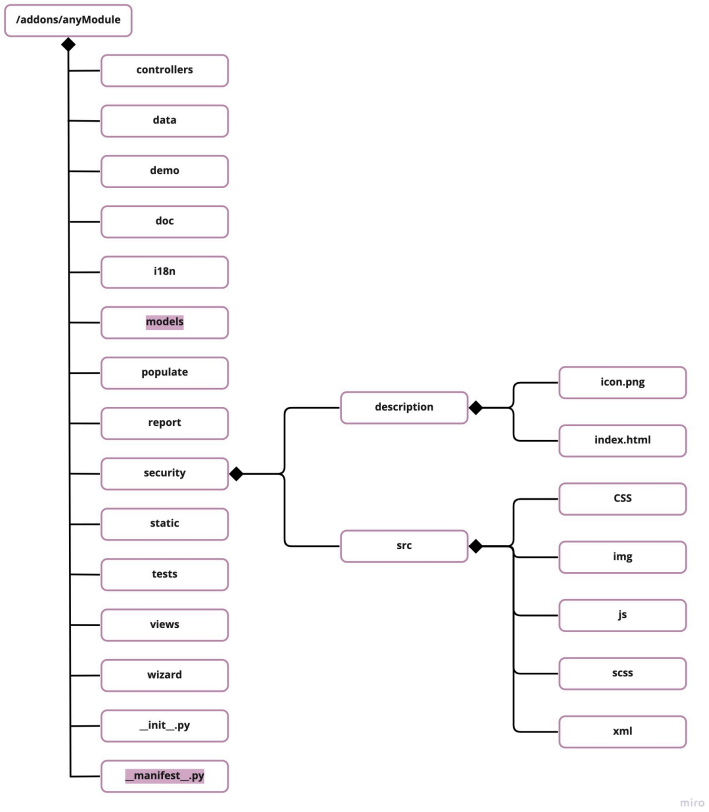


FIGURE 3.6: Odoo modules structure.

```
{
    'name': 'Sales',
    'version': '1.2',
    'category': 'Sales/Sales',
    'summary': 'Sales internal machinery',
    'description': """
This module contains all the common features of Sales Management
    and eCommerce.
    """,
    'depends': ['sales_team', 'payment', 'portal', 'utm'],
    'data': [
        'security/sale_security.xml',
        'security/ir.model.access.csv',
        'report/sale_report.xml',
```

```
        'report/report_all_channels_sales_views.xml',
        'data/ir_sequence_data.xml',
        'data/mail_templates.xml',
        'data/sale_data.xml',
        'wizard/sale_make_invoice_advance_views.xml',
        'views/sale_views.xml',
        'views/crm_team_views.xml',
        'views/payment_templates.xml',
        'views/product_views.xml',
        'views/utm_campaign_views.xml',
        'wizard/sale_order_cancel_views.xml',
        'wizard/sale_payment_link_views.xml',
    ],
    'demo': [
        'data/product_product_demo.xml',
        'data/sale_demo.xml',
    ],
    'installable': True,
    'auto_install': False,
    'assets': {
        'web.assets_backend': [
            'sale/static/src/scss/sale_onboarding.scss',
            'sale/static/src/scss/product_configurator.scss',
            'sale/static/src/js/product_discount_widget.js',
        ],
        'web.report_assets_common': [
            'sale/static/src/scss/sale_report.scss',
        ],
        'web.assets_frontend': [
            'sale/static/src/js/sale_portal_sidebar.js',
            'sale/static/src/js/payment_form.js',
        ],
    },
    'license': 'LGPL-3',
}
```

CODE LISTING 3.1: Sample manifest file

Reusability and Extension capability

As described in section 3.1.1, the Odoo framework is based on a multi-tier architecture. The design of the modules follows the same architectural vision. In addition, to facilitate the development of these modules, the framework offers each layer (Front-end, Back-End, and persistence layer) a core that will offer the necessary functionalities. The main classes of the Back-end layer core are implemented in the folder `/odoo/odoo`[12], and Figure 3.7 shows a static view of the classes implementing this core. All the models offered by the framework, or models that will potentially extend an existing one or a new module created from scratch, will necessarily inherit at least one of the core classes as described in the Listing 3.2. The technique used here to ensure usability is inheritance, as proposed by the object-oriented programming languages paradigms (OOP). In this way, the Odoo

[12] /odoo/odoo

framework maximizes reusability with the concept of inheritance as a powerful tool that allows the improvement and fast development of applications based on it.

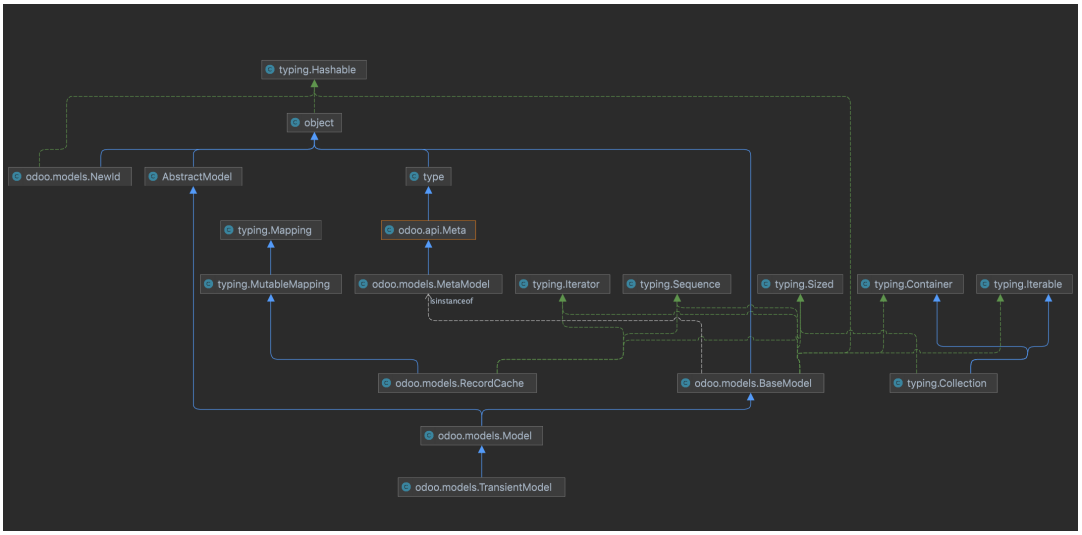


FIGURE 3.7: The static view of the classes implementing the back-end core of the Odoo framework

```
1
2  from odoo import api, fields, models
3
4  class CalendarEvent(models.Model):
5      _inherit = 'calendar.event'
6
7      @api.model
8      def default_get(self, fields):
9          if self.env.context.get('default_opportunity_id'):
10             self = self.with_context(
11                 default_res_model_id=self.env.ref('crm.
       model_crm_lead').id,
12                 default_res_id=self.env.context['
       default_opportunity_id']
13             )
14         defaults = super(CalendarEvent, self).default_get(fields
       )
15
16         # sync res_model / res_id to opportunity id (aka
       creating meeting from lead chatter)
17         if 'opportunity_id' not in defaults:
18             if self._is_crm_lead(defaults, self.env.context):
19                 defaults['opportunity_id'] = defaults.get('
       res_id', False) or self.env.context.get('default_res_id',
       False)
20
21         return defaults
22
23     opportunity_id = fields.Many2one(
24         'crm.lead', 'Opportunity', domain="[('type', '=', '
       opportunity')]",
```

```
25          index=True, ondelete='set null')
```

CODE LISTING 3.2: Example of model inheritance by the Calendar module

The Odoo framework is designed to be extensible to meet more specific user needs in parallel to maximizing reusability. For this, its architecture has been designed to be as extensible as possible. In order to allow this extensibility, the framework proposes mechanisms called Model Inheritance, and this designation may lead the reader to believe that it is object-oriented inheritance. However, these mechanisms are implemented with other techniques, such as composition and software design patterns based on the Python language, allowing certain flexibilities compared to other programming languages like Java. Here are three possibilities of Model Inheritance to extend the framework models modularly, as illustrated in Figure 3.8

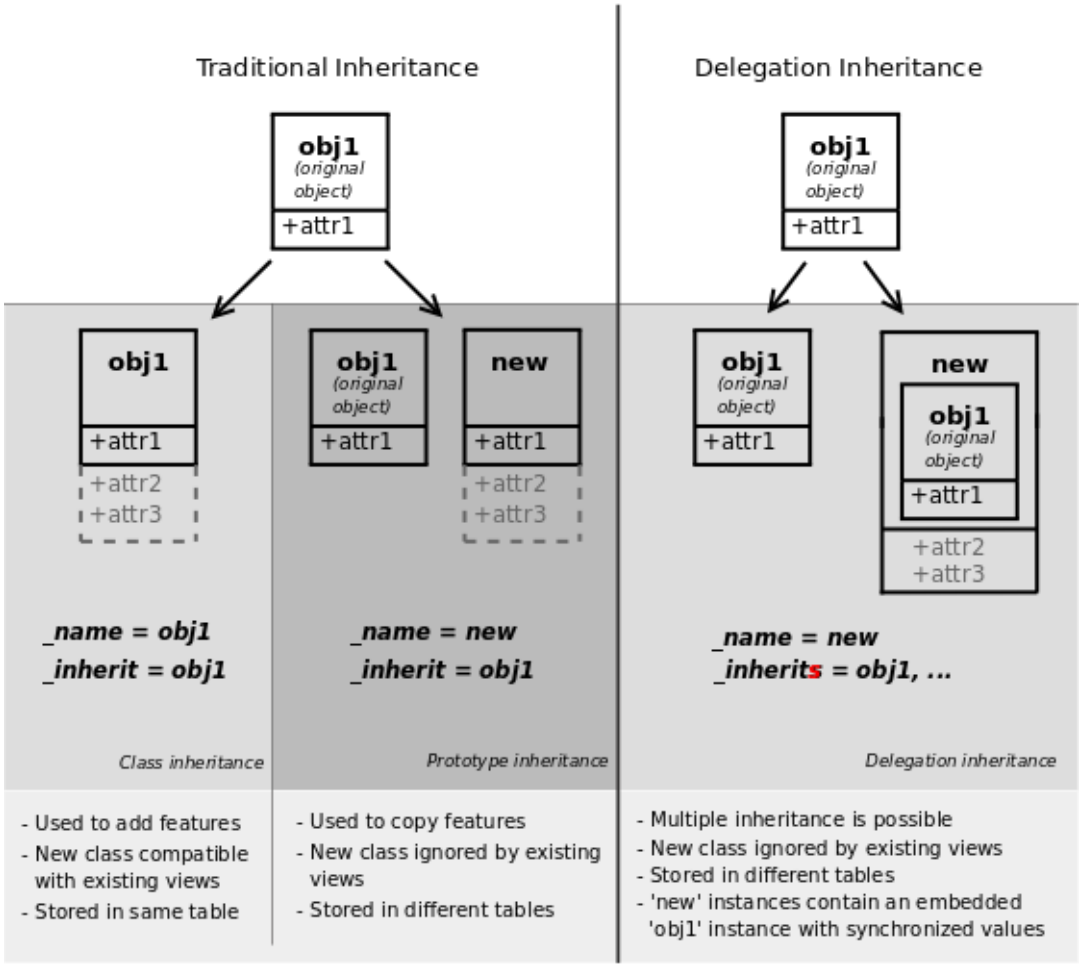


FIGURE 3.8: Model inheritance types

- **Classical Inheritance** is achieved by combining the attributes (`_inherit`, `_name`), so the new model gets all fields, methods, and

meta-information (e.g., default values) from its base. It is the most commonly used type for extending class, views, or other components. Listing 3.3 illustrates a simple example of applying this technique. The second model has inherited from the first model's check method and its name field but overridden the call method, as when using standard Python inheritance.

```
1
2
3       class Extension0(models.Model):
4       _name = 'extension.0'
5       _description = 'Extension zero'
6
7       name = fields.Char(default="A")
8
9       class Extension1(models.Model):
10      _inherit = 'extension.0'
11
12      description = fields.Char(default="Extended")
13
14  """ simple test
15  record = env['extension.0'].create({})
16  record.read()[0]
17
18  will yield: {'name': "A", 'description': "Extended"}
19  """
```

CODE LISTING 3.3: A simple Classical Inheritance example

- **Extention** is achieved when `_inherit` attribute is used alone without the `_name` attribute, the new existing template(extending it in place by copying all the data) replaces the old one. This technique helps add new fields or methods to existing templates or customize or reconfigure them (e.g., change default values). Listing 3.4 illustrates a simple example of applying this technique.

```
1
2       class Extension0(models.Model):
3       _name = 'extension.0'
4       _description = 'Extension zero'
5
6       name = fields.Char(default="A")
7
8       class Extension1(models.Model):
9       _inherit = 'extension.0'
10
11      description = fields.Char(default="Extended")
12
13  """ simple test
14  record = env['extension.0'].create({})
15  record.read()[0]
16
17  will yield: {'name': "A", 'description': "Extended"}
18  """
```

CODE LISTING 3.4: A simple Extention Inheritance example

- **Delegation** offers more flexibility (modification at runtime) and realized by the `_inherits` attribute. The instance of the created class will contain the instance of the original class. Each time an object is created on the new class, another object will be created without copying the data. Methods are not inherited, only fields. Many-to-One relationship will be created instead of duplicating the mother class. In a nutshell, When using delegation, the model has **one** instead of **is one**, turning the relationship into a composition instead of Inheritance. Listing 3.5 illustrates a simple example of applying this technique.

```
1
2  class Screen(models.Model):
3      _name = 'delegation.screen'
4      _description = 'Screen'
5
6      size = fields.Float(string='Screen Size in inches')
7
8  class Keyboard(models.Model):
9      _name = 'delegation.keyboard'
10     _description = 'Keyboard'
11
12     layout = fields.Char(string='Layout')
13
14 class Laptop(models.Model):
15     _name = 'delegation.laptop'
16     _description = 'Laptop'
17
18     _inherits = {
19         'delegation.screen': 'screen_id',
20         'delegation.keyboard': 'keyboard_id',
21     }
22     name = fields.Char(string='Name')
23     maker = fields.Char(string='Maker')
24     # a Laptop has a screen
25     screen_id = fields.Many2one('delegation.screen',
       required=True, ondelete="cascade")
26     # a Laptop has a keyboard
27     keyboard_id = fields.Many2one('delegation.keyboard',
       required=True, ondelete="cascade")
28
29 """ simple test
30 record = env['delegation.laptop'].create({
31         'screen_id': env['delegation.screen'].create({'size
       ': 13.0}).id,
32         'keyboard_id': env['delegation.keyboard'].create({'
       layout': 'QWERTY'}).id,})
33 record.size
34 record.layout
35 -------------
36 will result in:
37 13.0
38 'QWERTY'
39 """
```

CODE LISTING 3.5: A simple Delegation Inheritance example

3.1.3 Conclusion

In the Odoo framework, other components can be extended to maximize the reusability, like web controllers, security groups, python methods, qweb templates, and more. In short, it touches the three layers of the MVC model. In addition to the Python inheritance mechanisms, Odoo has a homemade mechanism that allows the improvement and rapid development of applications based on the Odoo framework. Also, this mechanism mimicks the object-oriented Inheritance and allow access to entities defined in other add-ons (existing ones or custom). In this way, the add-ons can extend all the system models. The downside with these techniques is that it is not an OOP, which means the IDE does not support these techniques, and the developer needs to have more discipline to work with it.

Chapter 4

Odoo Variability Mechanisms

This chapter aims to answer the research question **RQ1**. We will first, list some of the variability implementation mechanisms that we have identified through a manual exploration of the framework in its code source. Then, we will discuss these results.

4.1 Odoo variability mechanisms exploration

Template-method : As mentioned in section 3.1.2, inheritance is a mechanism used in all system layers systematically. The mechanism provides the main connection between the *core* and the *addons* to use the core functionality in a standardized fashion. The investigation has included multiple artifacts of the framework, but especially the `models.py` file. Figure 3.7 shows a static view of the core models indicating that the `Model` class in the `odoo/models.py` file inherits from `AbstractModel` and `BaseModel`, abstract classes in the OOP context. The `BaseModel` class implementation includes a signed method `view_init` specified in the parent class. The implementation of the `view_init` method is left free to the needs of the child classes. Listing 4.1 presents the `BaseModel` class, which is an abstract class, and the specification of the view-init method. Hence, traditional inheritance and inheritance combined with design-pattern template methods are two mechanisms used in the Odoo framework to maximize the reuse of the different functionalities.

```
1  class BaseModel(metaclass=MetaModel):
2      """Base class for Odoo models.
3
4      Odoo models are created by inheriting one of the following:
5
6      *   :class:`Model` for regular database-persisted models
7
8      *   :class:`TransientModel` for temporary data, stored in
       the database      but automatically vacuumed every so often
9
10     *   :class:`AbstractModel` for abstract super classes meant
       to be          shared by multiple inheriting models"""
11
12     """
13     .
14     .
```

```
15      .
16      """
17
18
19      @abstractmethod
20      def view_init(self, fields_list):
21          """ Override this method to do specific things when a
    form view is
22          opened. This method is invoked by :meth:`~default_get`.
23          """
24          pass
25
26  """ Some ligne below """
27
28  AbstractModel = BaseModel
29
30  class Model(AbstractModel):
31      """ Main super-class for regular database-persisted Odoo
    models.
32
33      Odoo models are created by inheriting from this class::
34
35          class user(Model):
36              ...
37
38      The system will later instantiate the class once per
    database (on
39      which the class' module is installed).
40      """
41      _auto = True                # automatically create database
    backend
42      _register = False           # not visible in ORM registry,
    meant to be python-inherited only
43      _abstract = False           # not abstract
44      _transient = False          # not transient
```

CODE LISTING 4.1: /odoo/models.py Excerpt

Parametrization : Conditional structures are widely used in implementing variability, as seen in Section 2.2.2. Using naming conventions will facilitate the traceability and mapping of features to their respective implementations in various artifacts. However, in the Odoo framework, after investigation and searching by keyword features name in the code, this mechanism is not widely used to implement variability. We found the only case after investigating the console log and then studying the code of the classes loaded when the application was started. The loaded classes include the file `/odoo/modules/loading.py`, which manages and load the modules (also called addons). In the implementation of this class, we have the method signed `_get_files_of_kind` illustrated in Listing 4.2, which, based on the received parameter, will filter what type of data to load for the startup, either demo data or accurate data that can be imported via XML or CSV file.

```
1  def load_data(cr, idref, mode, kind, package):
2      """
```

```
3
4       kind: data, demo, test, init_xml, update_xml, demo_xml.
5
6       noupdate is False, unless it is demo data or it is csv data
        in
7       init mode.
8
9       :returns: Whether a file was loaded
10      :rtype: bool
11      """
12
13      def _get_files_of_kind(kind):
14          if kind == 'demo':
15              kind = ['demo_xml', 'demo']
16          elif kind == 'data':
17              kind = ['init_xml', 'update_xml', 'data']
```

CODE LISTING 4.2: /odoo/modules.loading.py Excerpt

Cloning : Odoo is open source, which allows cloning the repository easily; also, Odoo offers the possibility of creating a generic mode of use, which allows adding or customizing the existing features. We have created a simple to-do list application based on the Odoo framework cloned from their official repository of the Community version. Figure 4.1 shows the structure of the to-do list application, and that follows the typical architecture of a module as described in Section 3.1.2, where we have the complete business logic in the file todotask.py. Cloning is a very used mechanism in the industry; Odoo lets the possibility of using their framework so that the end-user experiences the need for customization when the modules offered by the framework do not work with the needs of the end-users. However, the consequences of using such a technique must be carried out, and the architect or a developer has to balance the pros and cons while opting for this kind of approach.

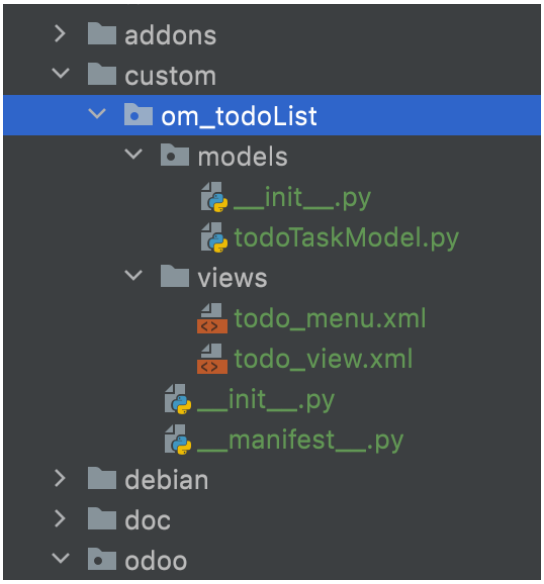


FIGURE 4.1: to-do list module structure

Decorator : The dynamical aspect of the Python language offers an escape from limitations[1]. It makes the use of design patterns such as the GoF family not very suitable for this category of programming language. However, design patterns are still applicable in many areas. Python programming language comes with a feature[2] to decorate any method to change its behavior with another method simply by adding a tag before the function, without changing the attributes of the decorated function or the class it belongs to. In simple terms, the decorator class is created and used to wrap up another class. Our example study on the Odoo framework noted the massive use of decorators, practically in all the framework artifacts. The file odoo/api.py[3] hosts the implementation of several decorator functions. However, the specifications of these decorator functions show that the use of decorators is not to add features to the run time variability but instead for the implementation of the framework itself.

Technology Framework : Frameworks are a proven technology for implementing variability and, at the same time, maximizing the reusability aspect to ensure the evolution of features. In Section 3.1.2, we have discussed that the Odoo framework offers the possibility to extend the functionality by adding new modules at explicit points in the framework. For this purpose, developers only need to consider the interface offered by the framework to add plug-ins, without caring about its internal

[1]Static languages are considered too rigid compared to the flexibility offered by dynamically typed languages. Because it requires upstream design and structure decisions, while in dynamic languages (e.g., Python), code can be imported dynamically, and classes can be created at runtime.

[2]https://wiki.python.org/moin/PythonDecoratorLibrary

[3]odoo/api.py

implementation, in a similar way as we have done with the to-do list application. So we can confirm that Odoo relies significantly on the Black-Box Framework technology (c.f. Section 2.2.2) to implement variability.

4.2 Odoo Binding Time

We already discussed in Section 4.1 that different variability implementation mechanisms are utilized, but not all support adding features that aim to extend the variability scope offered by the Odoo framework. The two mechanisms that support variability realization are inheritance, framework technology, and cloning. The inheritance and the framework technology mechanisms support the compilation and loading time as binding time (c.f. Section 2.2.1). The clone-and-own approach only support static binding time. Also, during the installation or uninstallation of an Odoo module, the console log lists all the loaded or uninstalled modules, the user interface at this particular time is in frozen mode. Moreover, when installing more than one module, interaction is only allowed with one module at a time via the same client. However, the modules can communicate by accessing the same database. Based on the information provided, we can discard the dynamic binding time proposition and determine that the Odoo framework achieves variability with a loading time-binding.

4.3 Discussion

This section aims to discuss our results and attempt to answer the research question **RQ1**, i.e., whether Odoo uses variability realization mechanisms in its development approach ?

Odoo mainly uses two mechanisms to implement variability: inheritance and the Black-Box framework technology. However, we determined that the inheritance has the purpose of connecting the plugin modules with the core to take advantage of its functionalities in an invariant method, and the framework technology is used to introduce the variability to match the business needs; in this way, they reduce the coupling while providing the possibility of extension without the necessity to make any modifications to the core. As a result, we can assert that the Odoo framework development uses variability realization mechanisms for the enhancement of user features, and this answers our research question **RQ1**.

However, it is not as trivial as to implement variability. Odoo uses variability implementation mechanisms that are widely used in every SPL. Odoo's goal is not necessarily to have an SPL according to the standards of the official methodology. Their goal is to have a kind of open variability that does not structure the space with an FM but allows to implementation of new modules by reusing the existing ones to have more freedom while realizing a maximum of use cases. For this purpose, their homemade

inheritance mechanism (an annotation mechanism! it is weird, but it is true) facilitates the extension of modules.

However, regarding the SPL development methodology, one of its main processes is Domain Engineering, ensuring that each SPL artifact is reusable. For this purpose, a precise execution flow is carried out, which includes, first, identifying the features, then modeling all these features to result in an FM (c.f., Section 2.1). To successfully derive a product from all possible configurations. Here, up to now, in our case study, we have not experienced that this methodology is applied in developing the Odoo framework. Also, according to our experience using the framework, we have observed that the features are used exclusively. If, for example, we intend to install the `Sale` module, we have no control over its dependencies. Everything is pre-configured in advance. So based on the elements we have learned yet, we cannot assert that the Odoo framework is an SPL. However, it shares a lot of common points with SPLs.

Chapter 5

Automatic Extraction of Feature Model

This chapter presents our RE approach and answers the research question **RQ2** and **RQ3**. For this, an overview of the RE process is illustrated and then executed step by step to obtain a variability mapping of the Odoo framework. The strategy we have followed for the RE is to retrieve the information extracted from the console log during the configuration (installation/uninstallation) of some variants proposed by the community version of the framework. Then using the FCA, we identified the topology of the links between all the different components of the variant products. Finally, we complete SPLA identification by recovering the existing architectural dependencies between the different variants. For this, our approach is inspired by the work of Shatnawi et al [26].

5.1 Reverse Engineering FM Step-by-Step

This section details the RE process of FM step by step. According to our approach, we identify the FM in seven steps as outlined below. First, using the FCA properties, we derive a concept lattice hierarchy based on objects and their properties; this hierarchy is presented as a concept lattice that adds value by visualizing the variability of the dependencies between all modules. Then, we complete SPLA identification by extracting the dependencies between modules to classify the Alternative, OR, AND, Require, and Exclude dependencies.

5.1.1 Modules dependencies identification

In section 4.1, we used the console log information from the installation and uninstallation of variant products in the Odoo Community Framework as a starting point for our investigation of the patterns and mechanisms for the variability implementation. We have opted to continue using the console log information for the RE process. We would have obtained the same results if we had exploited the `manifest.py` file, specifically the `'depends'` field, which points out all the dependencies concerning a given module. The Appendix A.1, illustrates the output of the console log when we install the E-commerce module on a clean database. So at the moment when the

installation is triggered, we get information about all the modules that the system loads one by one during 32.48 seconds (it can vary on another machine). As a reference, the user interface is in Frozen mode during this loading time.

5.1.2 Common and variable modules classification

To identify the modules that are common to multiple product variants and the modules that realize variability, we use FCA concepts (c.f., Section 2.3.2) to generate the lattice concept. Our study is limited to four variant products for readability reasons and illustrates the process with results to the reader in a simple way. So we have combined four product variants: Manufacturing, Sales, E-commerce, and Events. To drive the FCA, we organized our product variants and their dependencies so that the product variants represent the objects and the dependant modules represent the attributes. As described in section 2.3.2, Table 5.1.2 illustrates this presentation in Objects/Attributes [1].

[1]The Table is transposed for presentation reasons. However, the checkmark captures the dependency of a product variant(object), with component(attributes).

TABLE 5.1: Formal context of 4 components variants.

Attributes / Objects	**Sales**	**E-commerce**	**Events**	**Manufacturing**
account	✓	✓		
account_edi	✓	✓		
account_edi_facturx	✓	✓		
analytic	✓	✓		
auth_signup	✓	✓	✓	
auth_totp_mail	✓	✓	✓	
auth_totp_portal	✓	✓	✓	✓
barcodes				✓
base_setup	✓	✓	✓	✓
bus	✓	✓	✓	✓
digest	✓	✓	✓	✓
event			✓	
event_sms			✓	
fetchmail	✓	✓	✓	✓
google_recaptcha		✓	✓	
google_spreadsheet		✓	✓	
http_routing	✓	✓	✓	✓
iap	✓	✓	✓	
iap_mail	✓	✓	✓	✓
mail	✓	✓	✓	✓
mail_bot	✓	✓	✓	✓
mrp				✓
partner_autocomplete	✓	✓	✓	✓
payement	✓	✓		
payement_transfert	✓	✓		✓
phone_validation	✓	✓	✓	
portal	✓	✓	✓	✓
portal_rating		✓		
product	✓	✓		✓
resource	✓	✓	✓	✓
sale	✓	✓		
sale_management	✓			

sale_sms	✓	✓		
sale_team	✓	✓		
sms	✓	✓	✓	✓
snailmail	✓	✓		✓
snailmail_account	✓	✓		
social_media		✓	✓	
stock				✓
stock_sms				✓
uom	✓	✓		✓
utm	✓	✓	✓	
web_editor	✓	✓	✓	✓
web_unsplash	✓	✓	✓	✓
website		✓	✓	
website_event			✓	
website_mail		✓	✓	
website_partner			✓	
website_payement		✓		
website_sale		✓		
website_sms		✓		

Based on the dependencies presented in Table 2.2, we derived this concept lattice presented in figure 5.1 with the assistance of the Concept Explorer tool presented by Yevtushenko [46]. This concept lattice intends to present top-down sort, from the most general components in common for two or more variants to the most specific components belonging to a single variant. So based on the generated concept lattice, we have at the top of the structure all the components loaded by the four variants simultaneously (e.g., `mail`,`mail_bot`,`iap_mail`). Below in the structure, we have more specific components to one variant (e.g.,`barcode`,`mrp`,`stock`) that only belong to the Manufacturing variant.

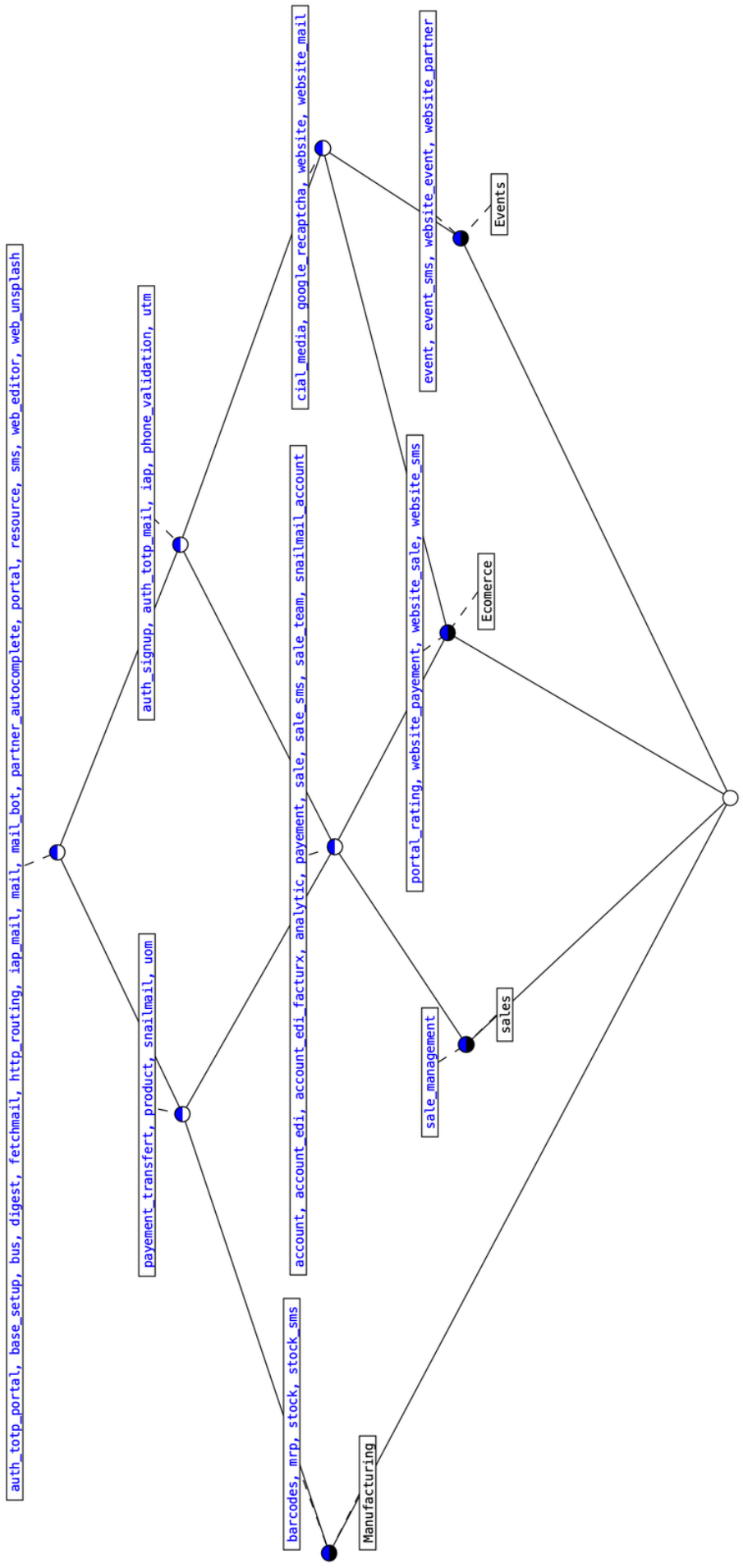


FIGURE 5.1: Extracted from FCA lattice showing the distribution of components composing the product's variants.

5.1.3 Extracting required Dependencies

When analyzing the concept lattice in Figure 5.2, we can identify the parent-to-child relationship by traversing the nodes from the top to the very bottom, so if node **A** *requires* node **B** only if node **B** is located higher than node **A** in the structure of the concept lattice. In the present example, the module `mail`shares a required relationship with all the nodes located under it in the hierarchy of the structure. So we can assert that the `website_mail` module requires the `mail` module and so forth for the entire structure.

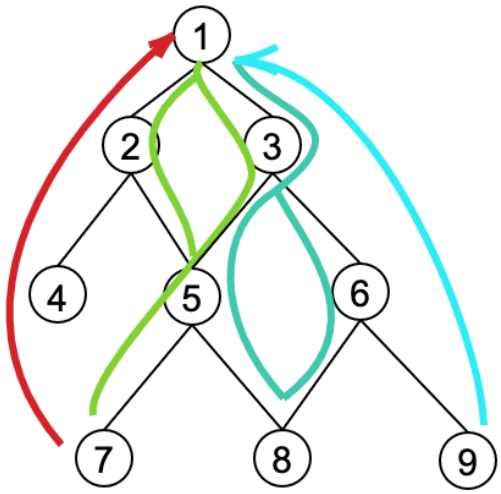


FIGURE 5.2: The extracted paths from FCA lattice

5.1.4 Extraction of Exclude and Alternative dependencies

An essential notion for continuing our dependency extraction heuristic is the notion of the path defined by Shatnawi et al.,

> *"The configurations are represented by paths starting from their concepts to the lattice concept root. The idea is that each object is generated starting from its node up and going to the top. This is based on sub-concept to super-concept relationships. This process generates a path for each object. A path contains an ordered list of nodes based on their hierarchical distribution; i.e., sub-concept to super-concept relationships)."* [26]

Moreover, to compute all possible paths in the concept lattice, Shatnawi et al., uses Breadth First Search (BFS) algorithm [47].

This concept of a path will be used to extract the exclusive dependency. So to extract all the exclusive dependencies, we will have to seek all the

components that can't be shared by the same variant if we refer to Figure 5.2, which represents the different paths that exist in the context of our lattice concept based on the Cormen et al., algorithm. We can assert that node 2 and node 9 represent a pair that share the Exclude relationship. Algorithm in Figure 5.3 automates the extraction of all nodes that share the Exclude dependency. The results produced from applying Algorithm are in Table 5.3.

Alternative dependencies are extracted by generalizing Exclusive dependencies calculated previously. When any node in the lattice concept shares an exclusive dependency with all other nodes in the lattice, these resulting nodes constitute an Alternative situation. In the lattice in Figure 5.2, node 4 is excluded concerning nodes 7, 8, and 9. Similarly, node 7 is excluded concerning nodes 8 and 9. Moreover, finally, node 8 is excluded concerning node 9. These mutual exclusions assert that nodes 4, 7, 8, and 9 represent an Alternative dependency.

```
Input: All Pairs of Lattice Nodes and Paths(Pairs, Paths)
Output: A Set of Pairs Having Exclude Dependency(ED)
ED = ∅;
$Search for pairs having an exclude$
for each pair ∈ Pairs do
    isFound = false;
    for each path ∈ Paths do
        if path.contains(pair) then
            isFound = true;
            break;
    end
    $If the pair is not found in the paths$
    if isFound == false then
        ED = ED ∪ pair ;
end
return ED
```

FIGURE 5.3: Identifying exclude pairs [26]

5.1.5 Extraction of AND dependencies

Components that share an AND dependency are the easiest to extract. Because AND dependencies are a two-way form of Required dependencies, meaning that if component A requires component B, then component A requires component A. So if a configuration must contain component A, it must also contain component B because both components A and B are linked with an AND dependency. This translates into all components located on the same node on the lattice, e.g., the components `payment_transfer`, `product`, `snailmail`, and `uom` are connected with an AND dependency.

TABLE 5.2: depencies extracted

	Node1	**Node2**	**Node3**	**Node4**	**Node5**	**Node6**	**Node7**	**Node8**	**Node9**
Node1				TR	TR	TR	TR	TR	TR
Node2	R		OR			Resolved			EX
Node3	R	OR		EX					
Node4	TR	R			EX	EX	EX, ALT	EX, ALT	EX, ALT
Node5	TR	R	R			OR			EX
Node6	TR	Resolved	R		OR		EX		
Node7	TR	TR	TR	ALT	R			EX, ALT	EX, ALT
Node8	TR	TR	TR	ALT	R	R	ALT		EX, ALT
Node9	TR		TR	ALT		R	ALT	ALT	

TABLE 5.3: All Depencies extracted (legend: R=required, TR=Transitive Required, OR=OR, EX=Exclude, ALT=Alternative)

5.1.6 Extracting OR dependencies

To calculate the OR dependencies, we must first extract all other dependencies such as Require, Exclude, Alternative, and AND. If any node must be concerned by an OR dependency, it cannot have any other dependencies. The algorithm in Figure 5.4 provides a procedure for extracting the OR dependencies automatically. E.g., the lattice concept in Figure 5.2 contains a dependency, OR for nodes 2 and 3.

At this point, we can say that we have answered our RQ2 and say that FCA allows extracting dependencies from product variants of the Odoo framework.

Algorithm 5: Identifying OR-groups.

```
Input: All Pairs (ap), Require Dependencies (rd), Exclude
       Dependencies (ed) and Alternative Dependencies (ad)
Output: Sets of Nodes Having OR Dependencies (orGroups)
$Remove pairs and groups having required, exclude and
alternative$
OrDep = ap.exclusionPairs(rd, ed, ad);
$Remove pairs having transitive required$
OrDep = orDep.removeTransitiveRequire(rd);
$Identify nodes that share some components$
ORPairsSharingNode = orDep.getPairsSharingNode();
$Process the dependencies among the unshared components$
for each p ∈ ORPairsSharingNode do
    if otherNodes.getDependency() == require then
        orDep.removePair(childNode);
    else if otherNodes.getDependency()= exclude || alternative
    then
        orDep.removeAllPairs(p);
end
orGroups = orDep.getPairsSharingOrDep();
return orGroups
```

FIGURE 5.4: Identifiying OR-groups [26]

5.1.7 Extracting the final hierarchical tree

Once the dependency relationships Required, Exclude, Alternative, AND, and OR have been extracted. We have the necessary elements to represent the dependency groups in the FM diagram. Algorithm in Figure 5.5 defines a procedure to identify a hierarchical representation. Then with the help of the FeatureIDE[2] visualization tool, we have realized the resulting FM on Figure[3] 5.6 following all the previous steps. This answers the RQ3.

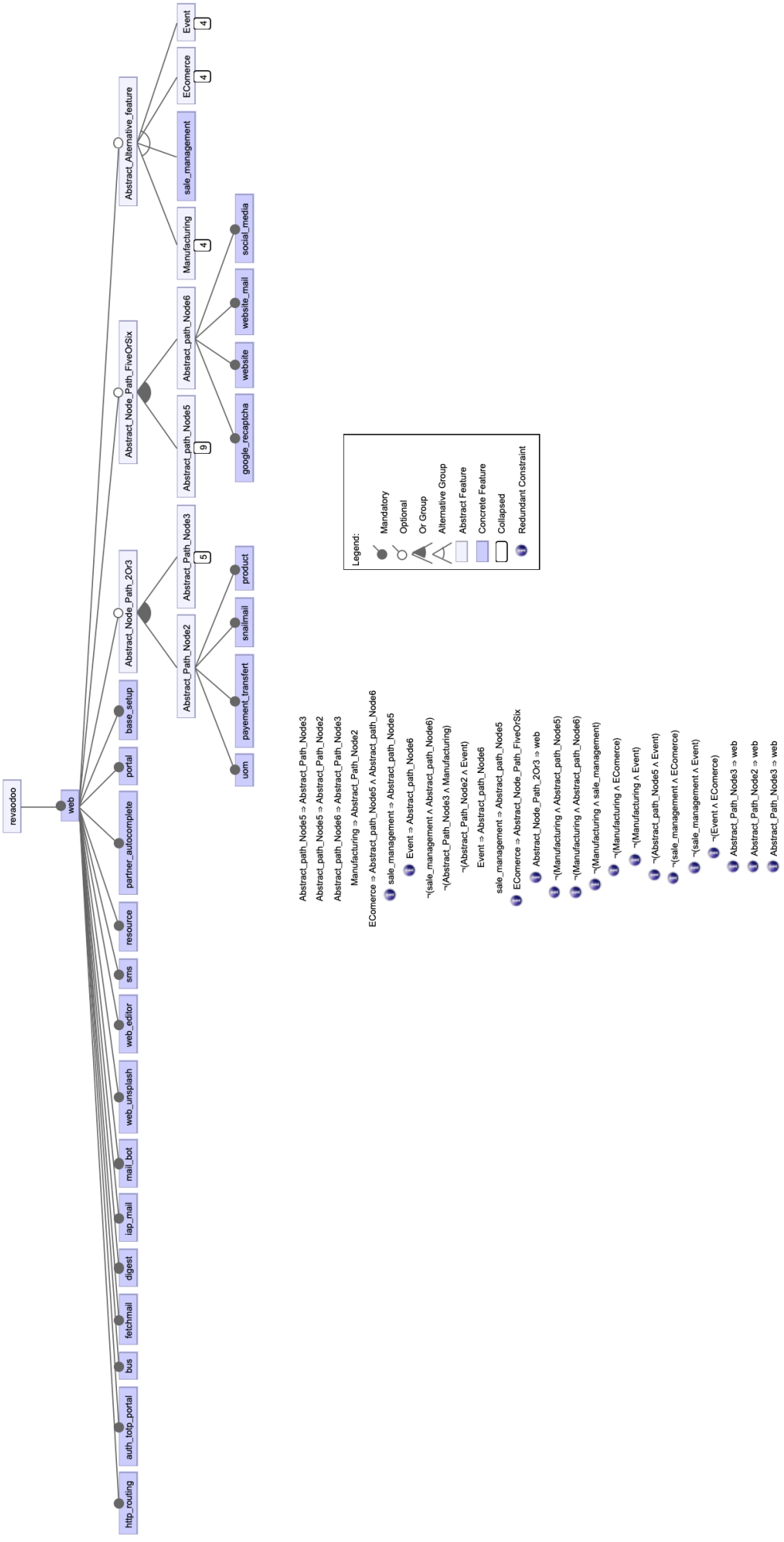
[2]https://www.featureide.de/

[3]To present the feature model with its constraints, we collapsed all the features attached to an abstract feature, but the complete FM is in Listing A.2

Algorithm 6: Identifying hierarchical representation.

```
Input: Sets of Dependencies
       (OR, AND, Require, Exclude, Alternative) and Mandatory
       and Optional Components (MC, OC)
Output: A Tree (tree)
$Adding mandatory components to the root node$
tree.root.addChildren(MC);
$Adding AND groups to the root node$
for each and ∈ AND do
    tree.addChild(and);
end
$Adding OR groups to the tree$
for each or ∈ OR do
    for each node ∈ or do
        $Check if the OR contains a member composed of an
        AND group$
        if AND.isContiant(node) then
            $Remove the AND from the root and add it as a
            child in the OR $
            tree.remove(node);
            nodeOR.addChildren(node);
        else
            nodeOR.addChildren(node);
        end
        tree.addChild(nodeOR);
    end
end
$Adding alternative groups to the tree$
for each alt ∈ Alternative do
    for each node ∈ alt do
        $Check if it is a member in any already added group$
        if OR.isContiant(node) then
            break;
        else if AND.isContiant(node) then
            tree.remove(node);
            nodeAlt.addChildren(node);
        else
            nodeAlt.addChildren(node);
        end
        tree.addChild(nodeAlt);
    end
end
$Add the rest of the components as optional$
tree.addChildren(OC.getRemainingOptional());
$add cross tree dependencies$
tree.addExcludeCrossTree(Exclude);
tree.addRequireCrossTree(Require);
return tree
```

FIGURE 5.5: Identifying hierarchical representation [26]


revaodoo
web
http_routing
auth_totp_portal
bus
fetchmail
digest
iap_mail
mail_bot
web_unsplash
web_editor
sms
resource
partner_autocomplete
portal
base_setup
Abstract_Node_Path_2Or3
Abstract_Path_Node2
uom
payement_transfert
snailmail
product
Abstract_Path_Node3
5
Abstract_Node_Path_FiveOrSix
Abstract_path_Node5
9
google_recaptcha
Abstract_path_Node6
website
website_mail
social_media
Abstract_Alternative_feature
Manufacturing
4
sale_management
EComerce
4
Event
4
Abstract_path_Node5 ⇒ Abstract_Path_Node3
Abstract_path_Node5 ⇒ Abstract_Path_Node2
Abstract_path_Node6 ⇒ Abstract_Path_Node3
Manufacturing ⇒ Abstract_Path_Node2
EComerce ⇒ Abstract_path_Node5 ∧ Abstract_path_Node6
sale_management ⇒ Abstract_path_Node5
Event ⇒ Abstract_path_Node6
¬(sale_management ∧ Abstract_path_Node6)
¬(Abstract_Path_Node3 ∧ Manufacturing)
¬(Abstract_Path_Node2 ∧ Event)
Event ⇒ Abstract_path_Node6
sale_management ⇒ Abstract_path_Node5
EComerce ⇒ Abstract_Node_Path_FiveOrSix
Abstract_Node_Path_2Or3 ⇒ web
¬(Manufacturing ∧ Abstract_path_Node5)
¬(Manufacturing ∧ Abstract_path_Node6)
¬(Manufacturing ∧ sale_management)
¬(Manufacturing ∧ EComerce)
¬(Manufacturing ∧ Event)
¬(Abstract_path_Node5 ∧ Event)
¬(sale_management ∧ EComerce)
¬(sale_management ∧ Event)
¬(Event ∧ EComerce)
Abstract_Path_Node3 ⇒ web
Abstract_Path_Node2 ⇒ web
Abstract_Path_Node3 ⇒ web
Legend:
Mandatory
Optional
Or Group
Alternative Group
Abstract Feature
Concrete Feature
Collapsed
Redundant Constraint

5.2 Discussion

At first impression, the resulting feature model has a flat hierarchy. However, it represents the variability of architecture and the decisions related to product design. Concerning the constraints of the resulting FM, the featureIDE tool enables determining constraints that are considered redundant, and we can ignore these redundant constraints without losing FM consistency. Also featureIDE calculate that there is 12 twelve valid products for this feature model. So far, the RE approach guided by FCA allowed the production of an FM based on the dependencies of the product variants. However, it remains interesting to try to produce another FM by exploiting information given by exploitation of the homemade inheritance mechanism of the Odoo framework and comparing the two resulting FMs to combine them with the hope of giving the FM a more hierarchy. Another thing to consider is testing the limitations of the proposed approach; our example was reduced to four product variants, but we have tested the approach, and it is effective for up to six product variants. It should be noted that the lattice generated based on seven product variants is not clear, so we need to automate the extraction to test the limits of the approach in a concrete way with multiple configurations.

Chapter 6

Conclusion

In this dissertation, first in Chapter 2, we have provided a state of the art that exposes all the notions we required to conduct the dissertation, including a general theory on SPL as well as the importance of SPLE methodology, the existing mechanisms of implementing variabilities in the industry, the RE process and its various approaches, and the FCA notions. Then in Chapter 3, we presented the Odoo framework as a commercial product, its general architecture, and the internal architecture of the add-on modules, which wrap up the features offered by Odoo. Also, Chapter 4, based on the result of a manual exploration of the source code, illustrated some mechanisms that Odoo uses to implement variability. The particularity of the Odoo framework case study is that its back-end is written in Python, which is an interpreted programming language. While in the literature, research and contributions in reverse engineering a feature model are mostly made on systems based on static programming languages such as Java or C/C++. This exploration allowed us to understand the internal Odoo core operation mode and identify that variability is mainly achieved through inheritance and framework technology. Finally, in chapter 5, we have executed an RE approach on four product variants based on the FCA properties; this heuristic has resulted in an FM that describes the variability based on the architectural dependencies that link the Odoo framework modules.

Appendix A

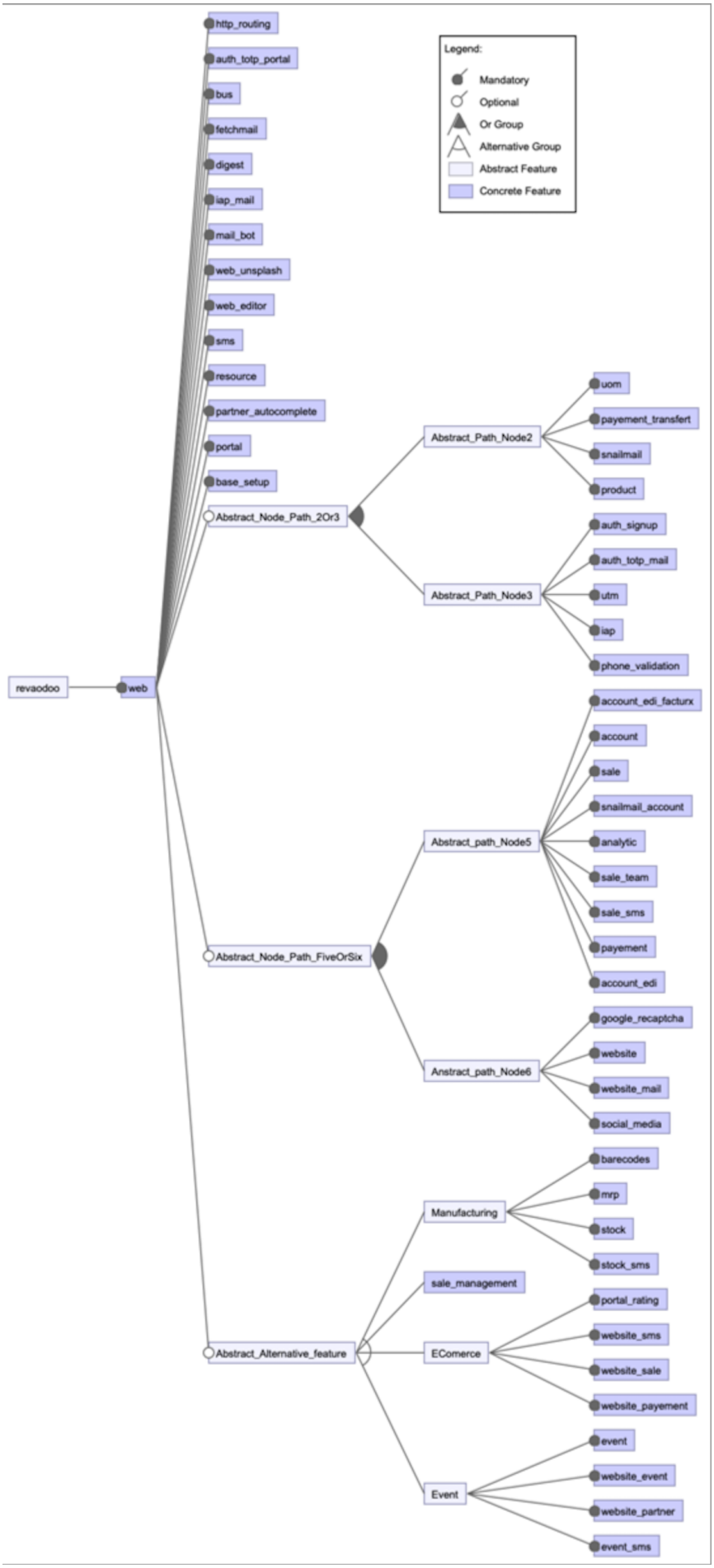
A.1 Console log output when when installing the E-commerce module

```
2022-02-21 01:37:57,213 70430 INFO zak odoo.addons.base.models.ir_module: ALLOW access to module.button_immediate_install on ['eCommerce']
to user admin #2 via 127.0.0.1
2022-02-21 01:37:57,213 70430 INFO zak odoo.addons.base.models.ir_module: User #2 triggered module installation
2022-02-21 01:37:57,216 70430 INFO zak odoo.addons.base.models.ir_module: ALLOW access to module.button_install on ['eCommerce']
to user admin #2 via 127.0.0.1
2022-02-21 01:37:57,442 70430 INFO zak odoo.modules.loading: loading 1 modules...
2022-02-21 01:37:57,451 70430 INFO zak odoo.modules.loading: 1 modules loaded in 0.01s, 0 queries (+0 extra)
2022-02-21 01:37:57,474 70430 INFO zak odoo.modules.loading: updating modules list
2022-02-21 01:37:57,475 70430 INFO zak odoo.addons.base.models.ir_module: ALLOW access to module.update_list on [] to user __system__ #1
via 127.0.0.1
2022-02-21 01:37:57,649 70430 WARNING zak odoo.modules.module: Missing `license` key in manifest for 'om_todoList', defaulting to LGPL-3
2022-02-21 01:37:58,133 70430 INFO zak odoo.modules.loading: loading 6 modules...
2022-02-21 01:37:58,139 70430 INFO zak odoo.modules.loading: 6 modules loaded in 0.01s, 0 queries (+0 extra)
2022-02-21 01:37:58,157 70430 INFO zak odoo.modules.loading: loading 48 modules...
2022-02-21 01:37:58,157 70430 INFO zak odoo.modules.loading: Loading module social_media (2/48)
2022-02-21 01:37:58,209 70430 INFO zak odoo.modules.registry: module social_media: creating or updating database tables

2022-02-21 01:37:58,283 70430 INFO zak odoo.modules.loading: Loading module uom (3/48)
2022-02-21 01:37:58,326 70430 INFO zak odoo.modules.registry: module uom: creating or updating database tables

2022-02-21 01:37:58,528 70430 INFO zak odoo.modules.loading: Loading module base_setup (7/48)
2022-02-21 01:37:58,576 70430 INFO zak odoo.modules.registry: module base_setup: creating or updating database tables
2022-02-21 01:37:58,695 70430 INFO zak odoo.modules.loading: Module base_setup loaded in 0.17s, 169 queries

2022-02-21 01:37:58,695 70430 INFO zak odoo.modules.loading: Loading module bus (8/48)
2022-02-21 01:37:58,745 70430 INFO zak odoo.modules.registry: module bus: creating or updating database tables
2022-02-21 01:37:58,874 70430 INFO zak odoo.modules.loading: Module bus loaded in 0.18s, 102 queries

2022-02-21 01:37:58,874 70430 INFO zak odoo.modules.loading: Loading module http_routing (9/48)
2022-02-21 01:37:58,942 70430 INFO zak odoo.modules.registry: module http_routing: creating or updating database tables
2022-02-21 01:37:59,028 70430 INFO zak odoo.modules.loading: Module http_routing loaded in 0.15s, 104 queries

2022-02-21 01:37:59,028 70430 INFO zak odoo.modules.loading: Loading module resource (10/48)
2022-02-21 01:37:59,083 70430 INFO zak odoo.modules.registry: module resource: creating or updating database tables
2022-02-21 01:37:59,490 70430 INFO zak odoo.modules.loading: Module resource loaded in 0.46s, 514 queries

2022-02-21 01:37:59,491 70430 INFO zak odoo.modules.loading: Loading module utm (11/48)
2022-02-21 01:37:59,546 70430 INFO zak odoo.modules.registry: module utm: creating or updating database tables
2022-02-21 01:37:59,977 70430 INFO zak odoo.modules.loading: Module utm loaded in 0.49s, 384 queries

2022-02-21 01:37:59,977 70430 INFO zak odoo.modules.loading: Loading module google_recaptcha (14/48)
2022-02-21 01:38:00,071 70430 INFO zak odoo.modules.registry: module google_recaptcha: creating or updating database tables
2022-02-21 01:38:00,191 70430 INFO zak odoo.modules.loading: Module google_recaptcha loaded in 0.21s, 71 queries

2022-02-21 01:38:00,191 70430 INFO zak odoo.modules.loading: Loading module iap (15/48)
2022-02-21 01:38:00,279 70430 INFO zak odoo.modules.registry: module iap: creating or updating database tables
2022-02-21 01:38:00,463 70430 INFO zak odoo.modules.loading: Module iap loaded in 0.27s, 157 queries

2022-02-21 01:38:00,463 70430 INFO zak odoo.modules.loading: Loading module mail (16/48)
2022-02-21 01:38:00,617 70430 INFO zak odoo.modules.registry: module mail: creating or updating database tables
2022-02-21 01:38:02,913 70430 INFO zak odoo.modules.loading: Module mail loaded in 2.45s, 3272 queries

2022-02-21 01:38:02,913 70430 INFO zak odoo.modules.loading: Loading module analytic (17/48)
2022-02-21 01:38:02,995 70430 INFO zak odoo.modules.registry: module analytic: creating or updating database tables
2022-02-21 01:38:03,323 70430 INFO zak odoo.modules.loading: Module analytic loaded in 0.41s, 409 queries

2022-02-21 01:38:03,323 70430 INFO zak odoo.modules.loading: Loading module auth_signup (18/48)
2022-02-21 01:38:03,457 70430 INFO zak odoo.modules.registry: module auth_signup: creating or updating database tables
2022-02-21 01:38:03,632 70430 INFO zak odoo.modules.loading: Module auth_signup loaded in 0.31s, 191 queries

2022-02-21 01:38:03,632 70430 INFO zak odoo.modules.loading: Loading module auth_totp_mail (19/48)
2022-02-21 01:38:03,720 70430 INFO zak odoo.modules.registry: module auth_totp_mail: creating or updating database tables
2022-02-21 01:38:03,812 70430 INFO zak odoo.modules.loading: Module auth_totp_mail loaded in 0.18s, 92 queries

2022-02-21 01:38:03,812 70430 INFO zak odoo.modules.loading: Loading module fetchmail (20/48)
2022-02-21 01:38:03,905 70430 INFO zak odoo.modules.registry: module fetchmail: creating or updating database tables
2022-02-21 01:38:04,074 70430 INFO zak odoo.modules.loading: Module fetchmail loaded in 0.26s, 189 queries

2022-02-21 01:38:04,074 70430 INFO zak odoo.modules.loading: Loading module iap_mail (21/48)
2022-02-21 01:38:04,190 70430 INFO zak odoo.modules.loading: Module iap_mail loaded in 0.12s, 28 queries

2022-02-21 01:38:04,190 70430 INFO zak odoo.modules.loading: Loading module mail_bot (22/48)
2022-02-21 01:38:04,274 70430 INFO zak odoo.modules.registry: module mail_bot: creating or updating database tables
```

```
2022-02-21 01:38:04,385 70430 INFO zak odoo.modules.loading: Module mail_bot loaded in 0.20s, 139 queries

2022-02-21 01:38:04,385 70430 INFO zak odoo.modules.loading: Loading module phone_validation (23/48)
2022-02-21 01:38:04,482 70430 INFO zak odoo.modules.registry: module phone_validation: creating or updating database tables
2022-02-21 01:38:04,711 70430 INFO zak odoo.modules.loading: Module phone_validation loaded in 0.33s, 179 queries

2022-02-21 01:38:04,711 70430 INFO zak odoo.modules.loading: Loading module product (24/48)
2022-02-21 01:38:04,879 70430 INFO zak odoo.modules.registry: module product: creating or updating database tables
2022-02-21 01:38:05,959 70430 INFO zak odoo.modules.loading: Module product loaded in 1.25s, 1392 queries

2022-02-21 01:38:05,959 70430 INFO zak odoo.modules.loading: Loading module rating (25/48)
2022-02-21 01:38:06,074 70430 INFO zak odoo.modules.registry: module rating: creating or updating database tables
2022-02-21 01:38:06,319 70430 INFO zak odoo.modules.loading: Module rating loaded in 0.36s, 310 queries

2022-02-21 01:38:06,319 70430 INFO zak odoo.modules.loading: Loading module sales_team (26/48)
2022-02-21 01:38:06,443 70430 INFO zak odoo.modules.registry: module sales_team: creating or updating database tables
2022-02-21 01:38:06,849 70430 INFO zak odoo.modules.loading: Module sales_team loaded in 0.53s, 525 queries

2022-02-21 01:38:06,849 70430 INFO zak odoo.modules.loading: Loading module web_editor (27/48)
2022-02-21 01:38:06,975 70430 INFO zak odoo.modules.registry: module web_editor: creating or updating database tables
2022-02-21 01:38:07,208 70430 INFO zak odoo.modules.loading: Module web_editor loaded in 0.36s, 241 queries

2022-02-21 01:38:07,208 70430 INFO zak odoo.modules.loading: Loading module partner_autocomplete (28/48)
2022-02-21 01:38:07,314 70430 INFO zak odoo.modules.registry: module partner_autocomplete: creating or updating database tables
2022-02-21 01:38:07,486 70430 INFO zak odoo.modules.loading: Module partner_autocomplete loaded in 0.28s, 189 queries

2022-02-21 01:38:07,486 70430 INFO zak odoo.modules.loading: Loading module portal (29/48)
2022-02-21 01:38:07,606 70430 INFO zak odoo.modules.registry: module portal: creating or updating database tables
2022-02-21 01:38:07,953 70430 INFO zak odoo.modules.loading: Module portal loaded in 0.47s, 437 queries

2022-02-21 01:38:07,953 70430 INFO zak odoo.modules.loading: Loading module sms (30/48)
2022-02-21 01:38:08,195 70430 INFO zak odoo.modules.registry: module sms: creating or updating database tables
2022-02-21 01:38:08,698 70430 INFO zak odoo.modules.loading: Module sms loaded in 0.75s, 621 queries

2022-02-21 01:38:08,698 70430 INFO zak odoo.modules.loading: Loading module snailmail (31/48)
2022-02-21 01:38:08,822 70430 INFO zak odoo.modules.registry: module snailmail: creating or updating database tables
2022-02-21 01:38:09,179 70430 INFO zak odoo.modules.loading: Module snailmail loaded in 0.48s, 376 queries

2022-02-21 01:38:09,179 70430 INFO zak odoo.modules.loading: Loading module web_unsplash (32/48)
2022-02-21 01:38:09,300 70430 INFO zak odoo.modules.registry: module web_unsplash: creating or updating database tables
2022-02-21 01:38:09,399 70430 INFO zak odoo.modules.loading: Module web_unsplash loaded in 0.22s, 91 queries

2022-02-21 01:38:09,400 70430 INFO zak odoo.modules.loading: Loading module auth_totp_portal (33/48)
2022-02-21 01:38:09,525 70430 INFO zak odoo.modules.registry: module auth_totp_portal: creating or updating database tables
2022-02-21 01:38:09,608 70430 INFO zak odoo.modules.loading: Module auth_totp_portal loaded in 0.21s, 75 queries

2022-02-21 01:38:09,609 70430 INFO zak odoo.modules.loading: Loading module digest (34/48)
2022-02-21 01:38:09,737 70430 INFO zak odoo.modules.registry: module digest: creating or updating database tables
2022-02-21 01:38:10,226 70430 INFO zak odoo.modules.loading: Module digest loaded in 0.62s, 308 queries

2022-02-21 01:38:10,226 70430 INFO zak odoo.modules.loading: Loading module portal_rating (35/48)
2022-02-21 01:38:10,426 70430 INFO zak odoo.modules.registry: module portal_rating: creating or updating database tables
2022-02-21 01:38:10,630 70430 INFO zak odoo.modules.loading: Module portal_rating loaded in 0.40s, 110 queries

2022-02-21 01:38:10,630 70430 INFO zak odoo.modules.loading: Loading module account (36/48)
2022-02-21 01:38:10,941 70430 INFO zak odoo.modules.registry: module account: creating or updating database tables
2022-02-21 01:38:15,832 70430 INFO zak odoo.modules.loading: Module account loaded in 5.20s, 6510 queries

2022-02-21 01:38:15,832 70430 INFO zak odoo.modules.loading: Loading module website (37/48)
2022-02-21 01:38:16,303 70430 INFO zak odoo.modules.registry: module website: creating or updating database tables
2022-02-21 01:38:20,697 70430 INFO zak odoo.modules.loading: Module website loaded in 4.86s, 6033 queries

2022-02-21 01:38:20,697 70430 INFO zak odoo.modules.loading: Loading module account_edi (38/48)
2022-02-21 01:38:20,919 70430 INFO zak odoo.modules.registry: module account_edi: creating or updating database tables
2022-02-21 01:38:21,269 70430 INFO zak odoo.modules.loading: Module account_edi loaded in 0.57s, 318 queries

2022-02-21 01:38:21,269 70430 INFO zak odoo.modules.loading: Loading module payment (39/48)
2022-02-21 01:38:21,499 70430 INFO zak odoo.modules.registry: module payment: creating or updating database tables
2022-02-21 01:38:22,890 70430 INFO zak odoo.modules.loading: Module payment loaded in 1.62s, 1109 queries

2022-02-21 01:38:22,890 70430 INFO zak odoo.modules.loading: Loading module snailmail_account (40/48)
2022-02-21 01:38:23,264 70430 INFO zak odoo.modules.registry: module snailmail_account: creating or updating database tables
2022-02-21 01:38:23,532 70430 INFO zak odoo.modules.loading: Module snailmail_account loaded in 0.64s, 133 queries

2022-02-21 01:38:23,532 70430 INFO zak odoo.modules.loading: Loading module website_mail (41/48)
2022-02-21 01:38:23,773 70430 INFO zak odoo.modules.registry: module website_mail: creating or updating database tables
2022-02-21 01:38:23,937 70430 INFO zak odoo.modules.loading: Module website_mail loaded in 0.41s, 38 queries

2022-02-21 01:38:23,937 70430 INFO zak odoo.modules.loading: Loading module website_sms (42/48)
2022-02-21 01:38:24,175 70430 INFO zak odoo.modules.registry: module website_sms: creating or updating database tables
2022-02-21 01:38:24,315 70430 INFO zak odoo.modules.loading: Module website_sms loaded in 0.38s, 81 queries

2022-02-21 01:38:24,315 70430 INFO zak odoo.modules.loading: Loading module account_edi_facturx (43/48)
2022-02-21 01:38:24,717 70430 INFO zak odoo.modules.registry: module account_edi_facturx: creating or updating database tables
2022-02-21 01:38:24,926 70430 INFO zak odoo.modules.loading: Module account_edi_facturx loaded in 0.61s, 64 queries

2022-02-21 01:38:24,926 70430 INFO zak odoo.modules.loading: Loading module payment_transfer (44/48)
2022-02-21 01:38:25,272 70430 INFO zak odoo.modules.registry: module payment_transfer: creating or updating database tables
2022-02-21 01:38:25,434 70430 INFO zak odoo.modules.loading: Module payment_transfer loaded in 0.51s, 116 queries

2022-02-21 01:38:25,435 70430 INFO zak odoo.modules.loading: Loading module sale (45/48)
2022-02-21 01:38:25,739 70430 INFO zak odoo.modules.registry: module sale: creating or updating database tables
2022-02-21 01:38:27,496 70430 INFO zak odoo.modules.loading: Module sale loaded in 2.06s, 2449 queries

2022-02-21 01:38:27,496 70430 INFO zak odoo.modules.loading: Loading module website_payment (46/48)
2022-02-21 01:38:27,868 70430 INFO zak odoo.modules.registry: module website_payment: creating or updating database tables
2022-02-21 01:38:28,113 70430 INFO zak odoo.modules.loading: Module website_payment loaded in 0.62s, 190 queries
```

```
2022-02-21 01:38:28,113 70430 INFO zak odoo.modules.loading: Loading module sale_sms (47/48)
2022-02-21 01:38:28,649 70430 INFO zak odoo.modules.loading: Module sale_sms loaded in 0.54s, 48 queries

2022-02-21 01:38:28,649 70430 INFO zak odoo.modules.loading: Loading module website_sale (48/48)
2022-02-21 01:38:29,094 70430 INFO zak odoo.modules.registry: module website_sale: creating or updating database tables
2022-02-21 01:38:30,995 70430 INFO zak odoo.modules.loading: Module website_sale loaded in 2.35s, 2699 queries

2022-02-21 01:38:30,995 70430 INFO zak odoo.modules.loading: 48 modules loaded in 32.84s, 30813 queries (+0 extra)
2022-02-21 01:38:31,719 70430 INFO zak odoo.modules.loading: Modules loaded.
2022-02-21 01:38:31,725 70430 INFO zak odoo.modules.registry: Registry loaded in 34.315s
```

A.2 The Complete Generated Feature Model



Bibliography

[1] Sven Apel, Don Batory, Christian Kästner, et al. *Feature-Oriented Software Product Lines Concepts and Implementation / by Sven Apel, Don Batory, Christian Kästner, Gunter Saake.* eng. 1st ed. 2013. Berlin, Heidelberg, 2013. ISBN: 3-642-37520-0.

[2] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software Product Line Engineering Foundations, Principles and Techniques / by Klaus Pohl, Günter Böckle, Frank J. van der Linden.* eng. 1st ed. 2005. Berlin, Heidelberg: Springer Berlin Heidelberg : Imprint: Springer, 2005. ISBN: 3-642-06364-0.

[3] Dirk MUTHIG and Thomas PATZKE. "Generic implementation of product line components". eng. In: *Lecture notes in computer science*. Berlin: Springer, 2003, pp. 313–329. ISBN: 3540007377.

[4] Frank Linden, Eelco Rommes, and Klaus Schmid. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. eng. 1. Aufl. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 9783540714361.

[5] Sven Apel and Christian Kästner. "An Overview of Feature-Oriented Software Development". eng. In: *Journal of object technology* 8.5 (2009), p. 49. ISSN: 1660-1769.

[6] Kyo Kang, Sholom Cohen, James Hess, et al. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-021. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231.

[7] J van Gurp, J Bosch, and M Svahnberg. "On the notion of variability in software product lines". eng. In: *Proceedings Working IEEE/IFIP Conference on Software Architecture*. IEEE, 2001, pp. 45–54. ISBN: 0769513603.

[8] Andreas Metzger and Klaus Pohl. "Software Product Line Engineering and Variability Management: Achievements and Challenges". In: *Future of Software Engineering Proceedings*. FOSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 70–84. ISBN: 9781450328654. DOI: 10.1145/2593882.2593888.

[9] Betty H. C Cheng, Rogério de Lemos, Paola Inverardi, et al. *Software Engineering for Self-Adaptive Systems edited by Betty H. C. Cheng, Rogério de Lemos, Paola Inverardi, Jeff Magee.* eng. 1st ed. 2009. Programming and Software Engineering ; 5525. Berlin, Heidelberg: Springer Berlin Heidelberg : Imprint: Springer, 2009. ISBN: 3-642-02160-3.

[10] Bo Zhang, Slawomir Duszynski, and Martin Becker. "Variability Mechanisms and Lessons Learned in Practice". eng. In: *2016 IEEE/ACM 1st International Workshop on Variability and Complexity in Software Design (VACE)*. ACM, 2016, pp. 14–20. ISBN: 9781450341769.

[11] Xhevahire Tërnava and Philippe Collet. "On the Diversity of Capturing Variability at the Implementation Level". eng. In: *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*. SPLC '17. ACM, 2017, pp. 81–88. ISBN: 1450351190.

[12] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. "A taxonomy of variability realization techniques". eng. In: *Software, practice experience* 35.8 (2005), pp. 705–754. ISSN: 0038-0644.

[13] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, et al. "Variability mechanisms in software ecosystems". eng. In: *Information and software technology* 56.11 (2014), pp. 1520–1535. ISSN: 0950-5849.

[14] Critina Gacek and Michalis Anastasopoules. "Implementing product line variabilities". eng. In: *Proceedings of the 2001 symposium on software reusability*. SSR '01. ACM, 2001, pp. 109–117. ISBN: 1581133588.

[15] T Patzke. *Sustainable evolution of product line infrastructure code*. eng. 2011.

[16] A Rashid, T Cottenier, P Greenwood, et al. "Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe". eng. In: *Computer (Long Beach, Calif.)* 43.2 (2010), pp. 19–26. ISSN: 0018-9162.

[17] J Bosch. "Superimposition: a component adaptation technique". eng. In: *Information and software technology* 41.5 (1999), pp. 257–273. ISSN: 0950-5849.

[18] Don Batory, Peter Höfner, and Jongwook Kim. "Feature interactions, products, and composition". eng. In: *SIGPLAN notices* 47.3 (2012), pp. 13–22. ISSN: 0362-1340.

[19] Marko Rosenmüller, Norbert Siegmund, Sven Apel, et al. "Flexible feature binding in software product lines". eng. In: *Automated software engineering* 18.2 (2011), pp. 163–197. ISSN: 0928-8910.

[20] Thomas Patzke, Martin Becker, Michaela Steffens, et al. "Identifying improvement potential in evolving product line infrastructures: 3 case studies". eng. In: *Proceedings of the 16th International Software Product Line Conference*. Vol. 1. SPLC '12. ACM, 2012, pp. 239–248. ISBN: 9781450310949.

[21] M.D Ernst, G.J Badros, and D Notkin. "An empirical analysis of c preprocessor use". eng. In: *IEEE transactions on software engineering* 28.12 (2002), pp. 1146–1170. ISSN: 0098-5589.

[22] Jörg Liebig, Sven Apel, Christian Lengauer, et al. "An analysis of the variability in forty preprocessor-based software product lines". eng. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 1. ICSE '10. ACM, 2010, pp. 105–114. ISBN: 9781605587196.

[23] CharlesW Krueger. "Easing the Transition to Software Mass Customization". eng. In: *Lecture notes in computer science*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 282–293. ISBN: 9783540436591.

[24] Axel Halin, Alexandre Nuttinck, Mathieu Acher, et al. "Yo Variability! JHipster: A Playground for Web-Apps Analyses". eng. In: 2017, pp. 44–51.

[25] Ra'Fat Ahmad Al-Msie'Deen, Marianne Huchard, Abdelhak-Djamel Seriai, et al. "Reverse Engineering Feature Models from Software Configurations using Formal Concept Analysis". eng. In: *11th International Conference on Concept Lattices and Their Applications, CEUR-Workshop*. Vol. 1252. 2014.

[26] Anas Shatnawi, Abdelhak-Djamel Seriai, and Houari Sahraoui. "Recovering software product line architecture of a family of object-oriented product variants". eng. In: *The Journal of systems and software* 131 (2017), pp. 325–346. ISSN: 0164-1212.

[27] Kun Chen, Wei Zhang, Haiyan Zhao, et al. "An approach to constructing feature models based on requirements clustering". eng. In: *13th IEEE International Conference on Requirements Engineering (RE'05)*. IEEE, 2005, pp. 31–40. ISBN: 0769524257.

[28] A Braganca and R.J Machado. "Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines". eng. In: *11th International Software Product Line Conference (SPLC 2007)*. IEEE, 2007, pp. 3–12. ISBN: 0769528880.

[29] Ra'Fat Ahmad Al-Msie'Deen, Abdelhak-Djamel Seriai, and Marianne Huchard. *Reengineering Software Product Variants into Software Product Line: REVPLINE Approach*. eng. Lambert Academic Publishing (LAP), 2014. ISBN: 3659511250.

[30] Yiming Yang, Xin Peng, and Wenyun Zhao. "Domain Feature Model Recovery from Multiple Applications Using Data Access Semantics and Formal Concept Analysis". eng. In: *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 215–224. ISBN: 9780769538679.

[31] Steven She, Rafael Lotufo, Thorsten Berger, et al. "Reverse engineering feature models". eng. In: *2011 33rd International Conference on Software Engineering (ICSE)*. ICSE '11. ACM, 2011, pp. 461–470. ISBN: 9781450304450.

[32] Mathieu Acher, Anthony Cleve, Gilles Perrouin, et al. "On extracting feature models from product descriptions". eng. In: *Proceedings of the Sixth International Workshop on variability modeling of software-intensive systems*. VaMoS '12. ACM, 2012, pp. 45–54. ISBN: 9781450310581.

[33] Evelyn Nicole Haslinger, Roberto Erick Lopez-Herrejon, and Alexander Egyed. "On Extracting Feature Models from Sets of Valid Feature Combinations". eng. In: *Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 53–67. ISBN: 364237056X.

[34] K Czarnecki and A Wasowski. "Feature Diagrams and Logics: There and Back Again". eng. In: *11th International Software Product Line Conference (SPLC 2007)*. IEEE, 2007, pp. 23–34. ISBN: 0769528880.

[35] Steven She, Uwe Ryssel, Nele Andersen, et al. "Efficient synthesis of feature models". eng. In: *Information and software technology* 56.9 (2014), pp. 1122–1143. ISSN: 0950-5849.

[36] Roberto E Lopez-Herrejon, Lukas Linsbauer, José A Galindo, et al. "An assessment of search-based techniques for reverse engineering feature models". eng. In: *The Journal of systems and software* 103 (2015), pp. 353–369. ISSN: 0164-1212.

[37] Mathieu Acher, Anthony Cleve, Philippe Collet, et al. "Extraction and evolution of architectural variability models in plugin-based systems". eng. In: *Software Systems Modeling* 13.4 (2013), pp. 1367–1394. ISSN: 1619-1366.

[38] Jabier Martinez, Tewfik Ziadi, Tegawende F Bissyande, et al. "Bottom-Up Technologies for Reuse: Automated Extractive Adoption of Software Product Lines". eng. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 67–70. ISBN: 9781538615898.

[39] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis Mathematical Foundations / by Bernhard Ganter, Rudolf Wille.* eng. 1st ed. 1999. Berlin, Heidelberg: Springer Berlin Heidelberg : Imprint: Springer, 1999. ISBN: 3-540-62771-5.

[40] "Concept data analysis; theory and applications". eng. In: *Scitech Book News* 29.2 (2005), p. 24. ISSN: 0196-6006.

[41] Jonas Poelmans, Paul Elzinga, Stijn Viaene, et al. "Formal Concept Analysis in Knowledge Discovery: A Survey". eng. In: *Conceptual Structures: From Information to Intelligence*. Lecture Notes in Computer Science (), pp. 139–153. ISSN: 0302-9743.

[42] Tom Huysegoms, Monique Snoeck, Guido Dedene, et al. "Visualizing Variability Management in Requirements Engineering through Formal Concept Analysis". In: *Procedia Technology* 9 (2013). CENTERIS 2013 - Conference on ENTERprise Information Systems / ProjMAN 2013 - International Conference on Project MANagement/ HCIST 2013 - International Conference on Health and Social Care

Information Systems and Technologies, pp. 189–199. ISSN: 2212-0173. DOI: https://doi.org/10.1016/j.protcy.2013.12.021.

[43] B. A Davey and H. A. (Hilary A.) Priestley. *Introduction to lattices and order / B.A. Davey, H.A. Priestley.* eng. 2nd ed. Cambridge, U.K. ; New York, N.Y.: Cambridge University Press, 2002. ISBN: 0521784514.

[44] *Adapting our open source license*. fr-FR. publisher: Odoo S.A. Feb. 2015. URL: https://www.odoo.com/blog/notre-blog-5/adapting-our-open-source-license-245 (visited on 05/12/2022).

[45] *Release Notes | Odoo*. fr-FR. publisher: Odoo S.A. URL: https://www.odoo.com/page/release-notes (visited on 05/12/2022).

[46] Serhiy A Yevtushenko. “System of data analysis" Concept Explorer"”. In: *Proc. 7th National Conference on Artificial Intelligence (KII'00)*. 2000, pp. 127–134.

[47] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, et al. *Introduction to Algorithms, Third Edition*. eng. 3rd ed. Cambridge: MIT Press, 2009. ISBN: 9780262533058.