Viewing an ERP as an SPL: Exploratory study

Yenisei Delgado Verdecia and María Cecilia Bastarrica

Computer Science Department, University of Chile
{ydelgado,cecilia}@dcc.uchile.cl

Abstract: Software Product Lines have been established as one of the best ways to promote reuse. Enterprise Resource Planning (ERP) are systems composed of a set of functional modules, whose implementation involves their configuration and customization in order to satisfy a customer’s needs. This paper focuses on integrating both concepts by viewing an ERP as an SPL by proposing an approach to extract an SPL from an existing ERP. To this end, we conducted an exploratory study of the Odoo ERP suite, illustrating how our approach is applied. As a result, we could recover the ERP domain artifacts and we could derive ERP implementations. Although the proposed approach was able to configure ERP implementations, we found that it cannot be used to customize assets or to define reusable assets not yet considered as part of the existent ERP.

Keywords: Software Product Line, SPL, Enterprise Resource Planning, ERP, Open Source ERP, Odoo ERP

1 Introduction

The need for integrated support for the operational management of an organization motivates the evolution of business information systems towards comprehensive solutions called Enterprise Resource Planning that integrate and facilitate, through a platform, the management of an organization [2, 6, 17, 22]. This platform is composed of interdependent units called modules that support the processes that are carried out in each department of the organization [3, 13]. The process of selecting and integrating the modules to meet the needs of the organization is known as ERP implementation. In this way, ERP systems can be implemented either altogether or addressing incrementally different functions, departments or modules of the organization operation.

A product line is a set of products that together address a particular market segment or fulfill a particular mission. In that sense, a Software Product Line (SPL) allows for the generation of a set of software products for a specific domain, based on the reuse of a well-defined infrastructure of core assets [20]. Core assets constitute all reusable elements of the SPL, e.g., features, software components, architecture, performance modeling and analysis, test cases, test plans, and documentation. This massive asset reuse reduces cost and time required for building each software product.
The work presented in this paper is motivated by the idea of viewing an ERP as an SPL and thus take advantage of the SPL product derivation strategy for implementing an ERP. Therefore, our main objective in this paper is to explore to what extent an ERP can be considered as an SPL. For this, we defined an approach to extract the core assets defined for any SPL from an existing ERP. To illustrate the feasibility of the proposed approach, we conducted an exploratory study using Odoo, an open source ERP. We found that our approach can manage ERP configuration but not other implementation operations such as customization or bolt-ons. However, it is worth stating that these modifications are not directly addressed by SPLs either.

The paper is organized as follows. Section 2 addresses the basic concepts of SPL and ERP. Section 3 describes our approach. We present the exploratory study in Section 4, its application and its results. Some related work is discussed in Section 6. Finally, Section 7 concludes our work and suggests future work.

2 Background Concepts

In this section we detail mainly those background concepts of SPL and ERP that are relevant for a clear understanding of the presented approach.

2.1 Software Product Line (SPL)

A SPL is a strategy for developing related systems from a set of common assets. Clements and Northrop [5], focus on the software product family:

“a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way”.

Pohl, Klaus and Böckle [18] define SPL as activity:

“a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization”.

Here “platform” refers to a base of technologies on which other technologies or processes are built [18] and “mass customization” is the large-scale production of goods tailored to individual customers’ needs [7]. In the SPL context, platform is a collection of reusable assets that are used in a consistent and systematic way to build products and to facilitate mass customization.

Reusable assets include different kinds of software development artifacts, such as requirements models, architectural models, software components, test plans, and test designs, among others. In order to allow for the development of custom applications that reuse predefined assets, the variability of the domain needs to be identified. In SPL, common features are part of every product, in exactly the same form, and variable features are only part of some products.

To build a robust platform as well as creating specific applications, SPL define two processes [12,18]: Domain engineering and Application engineering.
Domain Engineering [18] is the process that establishes the reuse infrastructure (upper part of Figure 1). Its main goals are: (1) To state the scope of the SPL; (2) To ensure that the variability is appropriately identified in order to generate all the products within the defined scope, and (3) To define and build the mandatory and reusable assets. This process is in turn involves three sub-processes:

**Domain analysis:** covers all the activities required to document the common and variable features as well as their relationships.

**Domain design:** establishes a high-level structure for the SPL, i.e., common and variable components for the development of specific applications based on a reference architecture.

**Domain realization:** provides the detailed design and the implementation of reusable software assets, based on the reference architecture.

There must be a correspondence between features, architecture components and implemented components, but this might not be exhaustive. For example, not all components defined in the reference architecture need to be realized upfront, they can be implemented only when needed and then they can be reused.

Application Engineering [18] Also know as product engineering, is the process that elicits specific requirements from end user and develops the product by reusing assets developed in the domain engineering (bottom part of Figure 1). Its main goals are: (1) To achieve the highest possible reuse of the domain assets, exploiting the commonalities and the variability of the SPL, and (2) To document the application artifacts, i.e., application requirements, architecture, components, and tests, and relate them to the domain artifacts. It involves three sub-processes:

**Application analysis:** covers all activities to define an application requirements specification. All deltas between application requirements and the available capabilities are identified.
**Application design:** encompasses the activities for producing the application architecture. This sub-process selects and configures the variable parts of the reference architecture and also incorporates application specific adaptations.

**Application realization:** uses reusable and application specific assets to build the intended application. Its main concerns are to select and configure reusable components and to realize application specific assets.

### 2.2 Enterprise Resource Planning (ERP)

An ERP is a set of information systems that allows for the integration of the operations of an organization such as management control, inventory, human resources and so forth. According to Jacobs and Bendoly [10], ERP can be defined as both a concept and a system. The conceptual definition refers to the integration of business processes within an organization. However, the definition of the ERP as a system is the technological infrastructure designed to support and take advantage of the capabilities of the tools used by an organization [10]. In this sense, an ERP considered as a system can be characterized as packaged software developed to meet the general needs of organizations [3,13].

Basically, an ERP combines several functions of management by means of packages related to the main areas of the organization. So ERP systems are designed to be modular, allowing for the gradual installation of packages. In this sense, the packages can be classified into three types:

(i) Basic or elementary packages: those that must be installed (mandatory) since they are necessary for any implementation of the ERP to work.

(ii) Additional or optional packages: selected depending on the specific needs of the organization complementing the basic packages.

(iii) Tailor-made packages: designed to meet specific organizational needs that are not contemplated in the basic or additional packages.

When an ERP system is implemented in the organization, ERP system packages are selected according to the characteristics of the organization and the processes that need support. In addition, some modifications to the ERP system may be necessary, *e.g.*, customization, configuration or costume-made packages. We now briefly describe these three types of ERP modifications:

(i) **Customization:** implies the modification of an ERP software package to match the existing business processes of the organization. Therefore, changes to the functionality or the source code of the ERP packages are made [9,19].

(ii) **Configuration:** involves configuration of parameters in the packages, to choose between different executions of processes and functions. In this type of modification, the ERP system code is not modified [3,9].

(iii) **Costume-made:** are extensions to the ERP, developed by either the implementers of an independent software provider to satisfy the needs of a particular client. In particular, when these packages are implemented by third parties, they are called bolt-ons [3,8].
### 3 Approach

Our purpose is to conceptualize the ERP as an SPL. The starting point is a particular ERP in the form of source code, whose structure is in principle not known. Therefore, it is necessary to reconstruct the architecture of the ERP to derive applications tailored to the specific customers’ needs. With this aim, our approach, as visualized in Figure 2, defines two processes: Recover domain and Derive application. We next describe each of these processes.

#### 3.1 Recover Domain (RD)

RD process (upper part of Figure 2) conceptualizes the ERP as an SPL in order to build the Domain Engineering. Thus, this process aims to reconstruct the reusable assets i.e., software components, architecture, features and the involved variability. To this end, we defined three sub-processes:

**Recover domain realization:** uses reverse engineering to reconstruct the software modules involved in the ERP system. It involves two phases:

1. Identify packages: set of packages that group the functionalities of the ERP modules. In this phase, the names of the packages are associated to the names of the ERP modules.

2. Differentiate generic and specific packages: define which ERP modules are generic and specific. In this phase, the elementary ERP modules are mapped to common packages in the SPL and optional ERP modules that are mapped to packages that may not exists in all products. Therefore, it is necessary to review the basic documentation of the ERP, in order to know how the ERP packages are organized and what are the basic and optional ERP packages.

**Recover domain design:** identifies software packages and their dependencies using a module viewtype [1]. This is the starting point for recovering the product line architecture (PLA). Its elements are modules i.e.,
the implementation of software units that provide a coherent set of responsibilities. It also considers three types of relationships: is part of, depends on, and is a. The mandatory and variable packages identified in the Recover domain realization need to be mapped to mandatory and variable modules in the PLA. To carry out this step, it may be useful to check the ERP documentation whenever available.

**Recover domain requirements:** identifies and describes the features that the products address. These features may be either mandatory or variable, and they must be consistent with the variability identified for the PLA. In this sub-process the scope of the domain, as well as the parameterizations points, are also identified. We defined two phases for this sub-process:

(i) Identify major features: identifies the scope of the domain. To express variability in terms of common and variable features, we may use different notations such as feature models [4] or OVM [15], among others, even though it is common in industry to use just spreadsheets. In this work we use feature models. We thus determine two key points to define the feature model:

- **Feature Relationship:** to identify relations between features, especially hierarchy, order, and grouping.
- **Constraints:** to identify restrictions among features and represent valid feature combinations, *e.g.*, ‘feature X requires feature Y’ and ‘feature X excludes feature Y’.

Considering these key points, the feature diagram is built, so the results of this phase is a feature model.

(ii) Identify parameterization\(^1\): recognizes the features that can be parameterized taking into account the characteristics of the organization. The result of this phase is the location of the parameterization point of the feature as well as the definition of the corresponding parameters.

### 3.2 Derive Applications (DA)

**DA process** (bottom of Figure 2) describes how to develop applications based on the recovered domain. To achieve that purpose we defined three sub-processes:

**Analyze requirements:** map the customer’s requirements to those identified earlier during Recover domain requirements. Thus, the requirements analysis includes the selection of existing features, setting parameters, modifying features that do not match exactly what is needed, and identifying features not considered yet. This results in a valid application within the SPL scope.

**Define architecture:** use the PLA to instantiate the application architecture. In this phase the selected and parameterized features are mapped to the corresponding modules in the PLA. The result of this phase is the specific application architecture.

\(^1\) ERP characteristic that allows that allows the system to adjust to the terminology of the business rules of each organization *e.g.*, language; the parameterization of the product when defining *e.g.*, the Spanish language.
Assemble application: compose the product assembling the packages obtained in the Recover domain process. Thus, a product can be assembled using reusable artefacts associated with the obtained architecture. The result of this phase is the product.

4 Exploratory study

We conducted an exploratory study that is defined in three phases: i) selecting the ERP to apply our approach, ii) recover domain of the ERP and iii) derive applications based on the recovered domain.

4.1 Phase 1 - ERP selection

Currently in the market we find two types of ERP systems: proprietary and open source ERP. Among proprietary ERPs we can find Oracle, SAP and Microsoft Dynamics. There are also several open source ERP systems, flexible and rich in functions, such as Adempiere, Openbravo or Odoo.

We selected Odoo for our exploratory study that is an open source ERP system. It was created by Odoo S.A. (formerly OpenERP S.A.) founded in 2004. It has an active community of developers, technical documentation and a cycle of publication of new versions of 12 months. It is a modular system that has official modules and others developed by third parties. It allows for the implementation and integration of costume-made modules to be able to adapt it to the specific needs of the organization. Its main modules are Sales, Customer Relationship Management, Human Resources, Purchase, Inventory, Manufacturing, Point of Sale, Accounting, Project, Timesheet, Payroll and Website.

In this exploratory study, we use Odoo 11.0, in its Community Edition\(^2\). It is developed in Python 3.5 and uses PostgreSQL 9.5 as a database management system. Odoo uses a client/server architecture in which clients are web browsers accessing the Odoo server via RPC Protocol\(^3\). This architecture permits that the business logic and extensions are generally performed on the server side, although supporting features can be added to the client, e.g., new data representation such as interactive maps. Moreover, it is designed to be multi-platform, e.g., Windows, Linux or Mac. It also follows the Model-View-Controller (MVC) architectural pattern. Then, Odoo modules contain in its directory, the models that define the data structure, the views that define the user’s interfaces and the controllers that support the business logic of the applications.

Odoo has a modular structure, where each module adds new features or modifies existing ones for building applications. In Odoo, modules are called ‘addon’ and each one is defined in a directory that in turn contains a number of directories described in Table 1. In addition, an Odoo module is declared by its manifest or descriptor file, called `manifest.py`, which specifies the module

\(^2\) Official Community packages with all relevant dependency requirements are available at \texttt{https://www.odoo.com/es_ES/page/download}

\(^3\) See \texttt{https://www.odoo.com/documentation/11.0/webservices/odoo.html}
<table>
<thead>
<tr>
<th>Directories</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>Contains demo and data xml</td>
</tr>
<tr>
<td>models</td>
<td>Contains models definition</td>
</tr>
<tr>
<td>controllers</td>
<td>Contains controllers (HTTP routes)</td>
</tr>
<tr>
<td>views</td>
<td>Contains the views and templates</td>
</tr>
<tr>
<td>static</td>
<td>Contains the web assets, separated into css/, js/, img/, lib/</td>
</tr>
<tr>
<td>Optional</td>
<td></td>
</tr>
<tr>
<td>wizard</td>
<td>Regroups the transient models and their views</td>
</tr>
<tr>
<td>report</td>
<td>Contains the reports (models based on SQL views (for reporting)), Python objects and XML views</td>
</tr>
<tr>
<td>tests</td>
<td>Contains the Python/YML tests</td>
</tr>
</tbody>
</table>

metadata. The manifest file\(^4\) contains a single Python dictionary, where each key specifies module metadata.

### 4.2 Phase 2 - Recover domain

The main objective of the Recover domain process is to reconstruct the reusable assets of the domain engineering e.g., components, PLA and features. As stated in Section 3, we first recover the realization, then the design and finally the requirements model.

**Realization recovery:** First, we identify packages based on the Odoo source code. According to the Odoo documentation, addons folder contains all ERP modules. We found 292 addons in that directory. Each module is classified by functional categories (defined in the ‘category’ key of the manifest file). In total, Odoo defines 29 functional categories to organize modules in areas of interest\(^5\).

As it can be seen in Figure 3, Accounting is a functional category that contains several addons inside. The relations between the addons were defined taking into account the ‘depends’ key of the manifest file of the module account. It depends on the product, analytic, portal, base-setup and web-planner modules. In turn, each of these modules has their own dependencies.

Our second objective is to identify generic and specific packages. According to Odoo documentation, the only generic module i.e., mandatory, is the so called base module. All other modules are considered as specific i.e., optional.

**Design recovery:** At this point, dependencies between packages are identified. In Odoo, when a module is installed, all of its dependencies are also installed, and

---

\(^4\) Each manifest fields is described in detail at [https://www.odoo.com/documentation/11.0/reference/module.html#reference-module-manifest](https://www.odoo.com/documentation/11.0/reference/module.html#reference-module-manifest)

\(^5\) The functional categories are defined at [https://github.com/odoo/odoo/blob/11.0/odoo/addons/base/module/module_data.xml](https://github.com/odoo/odoo/blob/11.0/odoo/addons/base/module/module_data.xml)
similarly during module loading. The ‘depends’ key in the manifest file defines existing dependencies between Odoo modules.

We selected only some areas of interest defined by Odoo for the exploratory study e.g., Accounting, Sales, Purchase, Inventory and Subscriptions. However, we will not consider Subscriptions for our case study as it is a module of the Odoo Enterprise Edition that includes 20% more modules.

Based on the manifest file corresponding to addons of the selected areas of interest, we can start to build the PLA. Figure 4 illustrates how we obtain a part of the PLA and shows the Module view of a part the PLA.

**Requirements recovery:** The first objective of this phase is to identify features, variability and constraints in terms of a feature model. This, in turn, defines the set of valid products. For this, we considered the areas of interest
shown in Figure 4. To create a feature diagram we used the FeatureIDE\textsuperscript{6} tool \cite{23}. Figure 5 depicts the feature model created. In the tree, nodes represent features and edges describe relations. A single root node, here ERP, represents the domain concept being modeled. Optional features are decorated with a hollow circle, \textit{e.g.}, Accounting. In this case, three constraints were defined, \textit{e.g.}, the first one, \textit{Purchase} $\Rightarrow$ \textit{Inventory}, reads ‘Purchase requires Inventory’. Similarly, for the other constraints.

The second objective of this phase is to identify the parameterization points. Depending on the business, there are different parameterizations to be made \textit{e.g.}, customer terminologies, local codes, formats (separators, date and time), currency structure, \textit{etc.} These parameterization points are directly related to the features. In Odoo, coin, date, time formats or language correspond to general parameterization points applied to all modules. This level of parameterization is defined in two types of files \textit{.po} and \textit{.pot} which are in the \textit{I18n} folder of each module. The \textit{.po} files are the actual translation files whereas \textit{.pot} is the template for the translation.

4.3 Phase 3 - Derive application

The main goal of this phase is to derive applications based on the recovered domain. So, our first objective is to map the application requirements (the ERP implementation) to the features of the domain and select features needed for a customized software system, resulting in a configuration. To exemplify this phase, we select \textit{Purchase} as an area of interest to derive an implementation of the ERP (as SPL).

As an application configuration is a decision process to get a valid feature combination, where the interdependencies of all features are considered \cite{18}, we used the FeatureIDE editor to configure an application according to the feature model. For this, we start from the obtained feature model (see Figure 6.\textsuperscript{1}), which has a configuration view as shown in Figure 6.\textsuperscript{2}, that allows an iterative feature (de)selection. Hence, based on the features selected, we can obtain a valid product that complies with the combinations of features and constraints defined in a feature model \textit{e.g.}, the configuration of \textit{Purchase} shown in Figure 6.\textsuperscript{3}.

Our second objective is to obtain the concrete architecture of the application. Based on the application configuration obtained (see Figure 7.\textsuperscript{1}), we can map features to architecture modules as well as their relationships. So, we derive the architecture presented in Figure 7.\textsuperscript{2}. Note that relations among modules is obtained from the PLA recovered during Design Recovery (see Figure 4).

The final objective is to compose the product. Considering the modules identified as part of the architecture -in this case for \textit{Purchase}- their realization is taken into account for assembling the application. Table 2 describes all the components that form part of these modules. Similarly, modules correspond to those identified during Realization Recovery (see Figures 3 and 4).

\textsuperscript{6} FeatureIDE is an open-source framework for feature-oriented software development (FOSD) based on Eclipse. See \url{https://featureide.github.io/}
5 Discussion

We now present the result of the case study and discuss some threats to validity.

5.1 Results

Mapping the functional areas of the ERP with features as well as defining the constraints, allow us to verify if a particular set of features are valid in the SPL. Constraints were defined based on the existing dependencies between modules corresponding to the functional areas considered in the Derive application phase. However, if a customer’s requirement cannot be mapped to one or more existing features, some strategy should be followed: either using an existing bolt-on that covers these needs or building a new module, but our approach does not address this issue yet. Moreover, once the feature model is conceptualized, it is easy to derive the product by reusing the defined product line architecture. Additionally, the traceability of the analysis, design and implementation elements is established by construction, and thus the derivation of particular products is straightforward whenever the feature model contains all the desired requirements.

Table 2. Components included in the actual application

<table>
<thead>
<tr>
<th>Modules</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purchase</td>
<td>purchase, stock_account</td>
</tr>
<tr>
<td>Inventory</td>
<td>stock, product, barcodes, web_planner</td>
</tr>
<tr>
<td>Accounting</td>
<td>account, portal, web, base, mail, product, web_planner, web_tour, http_routing, bus, base_setup, analytic, decimal_precision</td>
</tr>
</tbody>
</table>
5.2 Threats to validity

As in any exploratory study, there are limitations that must be considered when interpreting its results. We are aware that our study only considered one ERP, thus generalization will require further case studies. Furthermore, we used a particular type of ERP, in this case, an open source ERP (Odoo 11.0), thereby others ERP e.g., proprietary, could have lead to different results. Also, the possibility of properly recovering the elements of the domain depends to a large degree on the quality of the ERP documentation, both in identifying the modules implemented and in determining the variability. Fourth, identify at a code level, the definition of modules dependencies was crucial in view of being able to derive the architecture. In addition, the dependencies indicate the order of component implementation and; Fifth, the constraints defined are based on existing dependencies among modules corresponding to functional areas. However, despite limitations of our study, our results demonstrate that is possible viewing an ERP as an SPL.

6 Related work

Most studies relating ERP focus on modeling, but very few on considering ERP implementation as an SPL process. We have observed that publications on ERP modeling are quite old while SPL is still an area of intense research and interest. However, there are a few recent works that relate these two topics [14, 16].

Nöbauer et al. [16] present an industrial case study to evaluate a semi-automatic tool-based approach to identify reusable features in customized ERP products. Here, customers’ requirements are defined as reusable features that are used to customize ERP systems but not subsequent phases.

Wu et al. [24] define industry-oriented ERP (IERP) as an ERP software designed for specific industry sectors. They present a framework for developing IERP where business process modeling and software reuse are primary methods to improve the operation of component-based IERP. The framework consists of server layer, teamwork supporting layer, IERP construction and customization layer, reusable assets and tool-set layer, and the IERP system instance layer. In this sense, ERP and SPL concepts are implicitly related.

Soffer et al. [21] present a generic reverse engineering process, for modeling the alternatives at different application levels of an ERP system. They suggest that this generic process serves for comprehensive ERP modeling, as well as for obtaining a model of other process-supportive off-the-shelf systems that are of generic and configurable nature. This work focuses on implementing process variants based on the values of certain attributes of the ERP database. We do not claim to be as general as this work, but we also include application derivation.

Kouamou [11] presents the implementation of a service oriented ERP. He uses reverse engineering for reusing software and migrate the ERP towards service oriented architecture (SOA). The purpose is to implement a software package that respects the principles of SOA by reusing the existing solution. Here, SPL concepts are implicit and product derivation is not addressed.
Mani et al. [14] propose an approach for automatically generating the domain-specific rules based on user requirements by using the feature model. The feature model bridges the concepts of a domain model in ontology form and the business process. Again, the focus is only at the feature level.

7 Conclusion

In this paper, we defined the processes to reuse an ERP for deriving new products. For this, our approach proposes two processes: Recover domain and Derive applications which are composed of sub-processes. Our first process Recover domain conceptualizes the ERP as an SPL in order to build the Domain Engineering of the ERP and our second process Derive applications describes how to develop applications based on the recovered domain, i.e., the Application engineering. In an effort to illustrate how our approach can be put into practice, we conducted an exploratory study. We selected the Odoo ERP and we showed how each process defined in our approach can be carried out.

In summary, the results of this exploratory case study show that only an ERP can be seen as a product line following this approach if its code is accessible. Besides, the implementation of an ERP generally involves, not only the configuration of existing modules and the use of bolt-ons. It also requires internal customizations of the modules to add or modify functionality. This customization of modules in turn could eventually be considered as an SPL for each module, which adds a level of complexity: SPL of SPLs.

Although the automatic assembly of a specific product through the selection of features is a great challenge because it must guarantee the adequate integration of the components associated with the features, it still requires to formally model all SPL artifacts involved as well as the traceability among them.

References