Feature-Based Product Line Instantiation using Source-Level Packages*

Arie van Deursen[1], Merijn de Jonge[1], and Tobias Kuipers[2]

[1] CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
[2] Software Improvement Group
Kruislaan 419, 1098 VA Amsterdam, The Netherlands
Arie.van.Deursen@cwi.nl, http://www.cwi.nl/~arie
Merijn.de.Jonge@cwi.nl, http://www.cwi.nl/~mdejonge
tk@software-improvers.com, http://www.software-improvers.com/

Abstract. In this paper we discuss the construction of software products from customer-specific feature selections. We address variability management with the Feature Description Language (FDL) to capture variation points of product line architectures. We describe feature packaging which covers selecting and packaging implementation components according to feature selections using the `autobundle` tool. Finally, we discuss a generic approach, based on the abstract factory design pattern, to make instantiated (customer-specific) variability accessible in applications.
The solutions and techniques presented in this paper are based on our experience with the product line architecture of the commercial documentation generator DocGen.

1 Introduction

This paper deals with three key issues in software product lines: variability management, feature packaging, and product line instantiation. It covers these topics based on our experience in the design, implementation, and deployment of DocGen, a commercial product line in the area of documentation generation for legacy systems.

Like many product lines, DocGen started out as a single product dedicated to a particular customer. It was followed by modifications of this product for a series of subsequent customers, who all wanted a similar documentation generator, specialized to their specific needs. Gradually a kernel generator evolved, which could be instantiated to dedicated documentation requirements. DocGen is still evolving, and each new customer may introduce functionality that affects the kernel generator. This may involve the need for additional variation points in DocGen, or the introduction of functionality that is useful to a broad range of customers meriting inclusion in the standard DocGen setup.

The business model we use to accommodate such an evolving product line specialized to the needs of many different customers is based on subscription: customers can

* This research was sponsored in part by the Dutch Telematica Instituut, project DSL.

install a new DocGen version on a regular basis, which is guaranteed to be compatible with that customer's specializations. For the customer this has the advantage of being able to benefit from DocGen extensions; For the supplier it has the advantage of a continuous revenue flow instead of harder to predict large lump sums when selling a particular product.

The technological challenges of this business model and its corresponding evolutionary product line development are significant. In particular, product line evolution must be organized such that it can deal with many different customers. In this paper, we cover three related issues that we experienced as very helpful to address this problem.

Our first topic is managing the variation points, which are likely to change at each release. When discussing the use of DocGen with a potential customer, it must be clear what the variability of the current system is. The new customer may have additional wishes which may require the creation of new variation points, extending or modifying the existing interfaces. Moreover, marketing, customer support, or the product manager may come up with ideas for new functionality, which will also affect the variability. In Section 3 we study the use of the *Feature Description Language* FDL described in [8] to capture such a changing variability.

The second issue we cover is managing the source code components implementing the variability. The features selected by a customer should be mapped to appropriately configured software components. In Section 4 we describe our approach, which emphasizes developing product line components separately, using internally released software *packages*. Assembling a product from these packages consists of merging the sources of these packages, as well as the corresponding build processes – a technique called *source tree composition* [15]. Packages can either implement a feature, or implement functionality shared by other packages.

The third topic we address is managing customer code, that is, the instantiated variability. The same feature can be implemented slightly differently for different customers. In Section 5 we propose an extension of the abstract factory to achieve appropriate packaging and configuration of customer code.

In Section 6 we summarize our approach, contrast it with related work, and describe future directions. Before we dive into that, we provide a short introduction to the DocGen product line in Section 2.

2 A Documentation Generation Product Line

In this section we introduce the product line DocGen [9, 10], which we will use as our case study throughout the paper. Our discussion of DocGen follows the format used by Bosch to present his software product line case studies [5].

2.1 Company Background

DocGen is the flagship product of the Software Improvement Group (SIG), an Amsterdam, The Netherlands based company offering solutions to businesses facing problems with the maintenance and evolution of software systems. SIG was founded in 2000, and is a spin-off of academic research in the area of reverse and re-engineering conducted

at CWI from 1996 to 1999. This research resulted in, amongst others, a prototype documentation generator described by [9], which was the starting point for the DocGen product family now offered by SIG.

2.2 Product Family

SIG delivers a range of documentation generation products. These products vary in the source languages analyzed (such as SQL, Cobol, JCL, 4GL's, proprietary languages, and so on) as well as the way in which the derived documentation is to be presented.

Each DocGen product operates by populating a repository with a series of facts derived from legacy sources. These facts are used to derive web-based documentation for the systems analyzed. This documentation includes textual summaries, overviews, various forms of control flow graphs, architectural information, and so on. Information is available at different levels of abstraction, which are connected through hyperlinks.

DocGen customers have different wishes regarding the languages to be analyzed, the specific set of analyses to be performed, and the way in which the collected information should be retrieved. Thus, DocGen is a software product line, providing a set of reusable assets well-suited to express and implement different customized documentation generation systems.

2.3 Technology

At present, DocGen is an object-oriented application framework written in Java. It uses a relational database to store facts derived from legacy sources. It provides a range of core classes for analysis and presentation purposes. In order to instantiate family members, a Java package specific to a given customer is created, containing specializations of core classes where needed, including methods called by the DocGen factory for producing the actual DocGen instantiation. DocGen consists of approximately 850 Java classes, 750 Java classes generated from language definitions, 250 Java classes used for continuous testing, and roughly 50 shell and Perl scripts.

In addition to the key classes, DocGen makes use of external packages, for example for graph drawing purposes.

2.4 Organization

Since SIG is a relatively small company, the development team tries to work as closely together as possible. For that reason, there is no explicit separation between a core product line development team and project teams responsible for building bespoke customer products. Instead, these are roles, which are rotated throughout the entire team.

For each product instantiation, a dedicated person is assigned in order to fulfill the customer role. This person is responsible for accepting or rejecting the product derived from DocGen for a particular customer.

2.5 Process

The construction of DocGen is characterized by evolutionary design (DocGen is being developed following the principles of extreme programming [4]). DocGen started as a research prototype described by [9]. This prototype was not implemented as a reusable framework; instead it just produced documentation as desired by one particular customer. As the commercial interest in applications of DocGen grew, more and more variation points were introduced, evolving DocGen into a system suitable for deriving many different documentation generation systems.

With the number of customer configurations growing, it is time to rethink the way in which DocGen product instantiations are created, and what sort of variability the DocGen product line should offer.

3 Analyzing Variability

3.1 Feature Descriptions

To explore the variability of software product lines we use the Feature Description Language FDL discussed by [8]. This is essentially a textual representation for the feature diagrams of the Feature Oriented Domain Analysis method FODA [16].

A feature can be *atomic* or *composite*. We will use the convention that names of atomic features start with a lower case letter and names of composite features start with an upper case letter. Note that atomic and composite features are called features, respectively, subconcepts in [7].

An FDL definition consists of a number of *feature definitions*: a feature name followed by "`:`" and a *feature expression*. A feature expression can consist of

- an atomic feature;
- a composite feature: a named feature whose definition appears elsewhere;
- an optional feature: a feature expression followed by "`?`";
- mandatory features: a list of feature expressions enclosed in `all( )`;
- alternative features: a list of feature expressions enclosed in `one-of( )`;
- selection of features:[1] a list of feature expressions enclosed in `more-of( )`;
- features of the form `...`, indicating that a given set is not completely specified.

An FDL definition generates all possible feature configurations, also called *product instances*. Feature configurations are flat sets of features of the form `all(`$a_1, \ldots, a_n$`)`, where each a_i denotes an atomic feature. In [8] a series of FDL manipulations is described, to bring any FDL definition into a *disjunctive normal form* defined as:

`one-of( all(`$a_{11}, \ldots, a_{1n_1}$`),..., all(`$a_{m1}, \ldots, a_{mn_m}$`))`

By bringing an FDL definition in disjunctive normal form, a feature expression is obtained that lists all possible configurations.

Feature combinations can be further restricted via *constraints*. We will adopt the following constraints:

[1] Called "or-features" in [7].

- `A1 requires A2`: if feature `A1` is present, then feature `A2` should also be present;
- `A1 excludes A2`: if feature `A1` is present, then feature `A2` should not be present;

Such constraints are called *diagram constraints* since they express fixed, inherent, dependencies between features in a diagram.

DocGen **:**
  all(Analysis**,** Presentation**,** Database**)**
Analysis **:**
  all(LanguageAnalysis**,** *versionManagement***?,** *subsystems***?)**
LanguageAnalysis **:**
  more-of(Cobol**,** *jcl***,** *sql***,** *delphi***,** *progress***, ...)**
Cobol **:**
  one-of(*ibm-cobol***,** *microfocus-cobol***, ...)**
Presentation **:**
  all(Localization**,** Interaction**,** MainPages**,** Visualizations**?)**
Localization **:**
  more-of(*english***,** *dutch***)**
Interaction **:**
  one-of(*static***,** *dynamic***)**
MainPages **:**
  more-of(ProgramPage**,** *copybookPage***,** StatisticsPage**,** *indexes***,** *searchPage***,**
        *subsystemPage***,** *sourcePage***,** *sourceDifference***, ...)**
ProgramPage **:**
  more-of(*annotationSection***,** *activationSection***,** *entitiesSection***,** *parametersSection***, ...)**
StatisticsPage **:**
  one-of(*statsWithHistory***,** *statsNoHistory***)**
Visualizations **:**
  more-of(*performGraph***,** *conditionalPerformGraph***,** *jclGraph***,** *subsystemGraph***,**
        *overviewGraph***, ...)**
Database **:**
  one-of(*db2***,** *oracle***,** *mysql***, ...)**

Fig. 1. Some of the configurable features of the DocGen product line expressed in the Feature Description Language (FDL).

3.2 DocGen Features

A selection of the variable features of DocGen and some of their constraints are shown in Figures 1 and 2. The features listed describe the variation points in the current version of DocGen. One of the goals of constructing the FDL specification of these features is to search for alternative ways in which to organize the variable features, in order to optimize the configuration process of DocGen family members. Another goal of the FDL specification is to help (re-) structuring the implementation of product lines.

The features listed focus on just the Analysis and Presentation configuration of DocGen, as specified by the first dependency of Figure 1. The Database feature in that dependency will not be discussed here.

%% Some constraints	
subsystemPage	**requires** *subsystems*
subsystemGraph	**requires** *subsystems*
sourceDifference	**requires** *versionManagement*
%% Some source language constraints	
performGraph	**requires** *cobol*
conditionalPerformgraph	**requires** *cobol*
jclGraph	**requires** *jcl*
%% Mutually exclusive features	
static	**excludes** *annotationSection*
static	**excludes** *searchPage*

Fig. 2. Constraints on variable DocGen features.

The Analysis features show how the DocGen analysis can be influenced by specifying source languages that DocGen should be able to process. The per-language parsing will actually populate a data base, which can then be used for further analysis.

Other features are optional. For example, *versionManagement* can be switched on, so that differences between documented sources can be seen over time. When a system to be documented contains subsystems which need to be taken into account, the *subsystems* feature can be set.

The Presentation features affect the way in which the facts contained in the repository are presented to DocGen end users. As an example, the Localization feature indicates which languages are supported (English, Dutch, ...). At compile time, one or more supported languages can be selected; at run time, the end user can use a web-browser's localization scheme to actually select a language.

The Interaction feature determines the moment the HTML pages are generated. In *dynamic* interaction, a page is created whenever the end-user requests a page. This has the advantage that the pages always use the most up-to-date information from the repository and that interactive browsing is possible. In *static* mode, all pages are generated in advance. This has the advantage that no web-server is needed to inspect the data and that they can be easily viewed on a disconnected laptop. As we will see, the Interaction feature puts constraints on other presentation features.

The MainPages feature indicates the contents of the root page of the derived documentation. It is a list of standard pages that can be reused, implemented as a many-to-one association to subclasses of an abstract "Page" class. Of these, the ProgramPage consists of one or more sections.

In addition to pages, presentation includes various Visualizations. These are all optional, allowing a customer to choose to have plain (HTML) documentation (which requires less software to be installed at the client side) or graphically enhanced documentation (which requires plug-ins to be installed).

3.3 DocGen Feature Constraints

Figure 2 lists several constraints restricting the number of valid DocGen configurations of the features listed in Figure 1.

The pages that can be presented depend on the analyses that are conducted. If we want to show a *subsystemGraph*, we need to have selected *subsystems*. Some features are language specific: a *jclGraph* can only be shown when *jcl* is one of the analyzed languages.

Last but not least, certain features are in conflict with each other. In particular, the *annotationSection* can be used to let the end-user interactively add annotations to pages, which are then stored in the repository. This is only possible in the dynamic version, and cannot be done if the Interaction is set to *static*. The same holds for the dynamic *searchPage*.

3.4 Evaluation

Developing and maintaining a feature description for an existing application, gives a clear understanding of the variability of the application. It can be used not only during product instantiation, but also when discussing the the design of the product line. As an example, discussions about the DocGen feature description have resulted in the discovery of several potential inconsistencies, as well as suggestions for resolving them.

One of the problems with the use of feature descriptions is that feature dependencies can be defined in multiple ways, for instance as a composite feature definition or as a combination of a feature definition and constraints. We are still experimenting with using these different constructs in order to develop heuristics about when to use which construct.

4 Software Assembly

4.1 Source Tree Composition

Source tree composition is the process of assembling software systems by merging reusable source code components. A *source tree* is defined as a collection of source files, together with *build instructions*, divided in a directory hierarchy [15]. We call the source tree of a particular part of a product line architecture a *source code component*. Source code components can be developed, maintained, tested, and released individually.

Source tree *composition* involves merging source trees, build processes, and configuration processes. It results in a single source tree with centralized build and configuration processes [15].

Source tree composition requires abstractions over source code components which are called *package definitions* (see Figure 3). They capture information about the component such as its dependencies on other components, and its configuration parameters. Package definitions as the one in Figure 3 are called *concrete* package definitions because they correspond directly to an implementing source code component. *Abstract* package definitions, on the other hand, do not correspond to implementing source code

```
package
identification
          name=docgen
        version=2.4
       location=http://www.software-improvers.com/packages
            info=http://www.software-improvers.com
   description='DocGen, documentation generation tool.'
     keywords=docgen, documentation generation, core
configuration interface
customer'name identifying customer-specific issues'
requires
   html-lib    1.5   with xml-support=on
   gnuregexp 1.1.4
   junit         3.5
```

Fig. 3. Example of a package definition for the DocGen core package. The 'configuration interface' section lists configurable items together with corresponding short descriptions. The 'requires' section defines package dependencies and their configuration.

components. They only combine existing packages and set configuration options. Abstract package definitions are distinguished from concrete package definitions by having an empty location field (see Figure 5).

After selecting components of need, a software *bundle* (containing all corresponding source trees) is obtained through a process called *package normalization*. This process includes package dependency and version resolution, build order arrangement, configuration distribution, and bundle interface construction. We refer to [15] for a complete description of source tree composition.

Package definitions are stored centrally in *package repositories*. They can be accessed by developers to search for reusable packages. They are also accessed by the tool `autobundle` (see below) to resolve package dependencies automatically when assembling product instances.

Source tree composition is supported and automated by `autobundle`[2] [15]. Given a set of package names, it (i) obtains their package definitions from package repositories; (ii) calculates the transitive closure of all required packages; (iii) calculates a (partial) configuration of the individual packages; (iv) generates a self-contained source tree by merging the source trees of all required packages; (v) integrates the configuration and build processes of all bundled packages.

4.2 Source Tree Composition in Product Lines

With the help of `autobundle`, assembling products on a product line can become as simple as selecting the necessary packages. `Autobundle` is used to bundle them together with required packages into self-contained customer-specific source distributions.

[2] `autobundle` is free software and available for download at http://www.cwi.nl/~mdejonge/autobundle/.

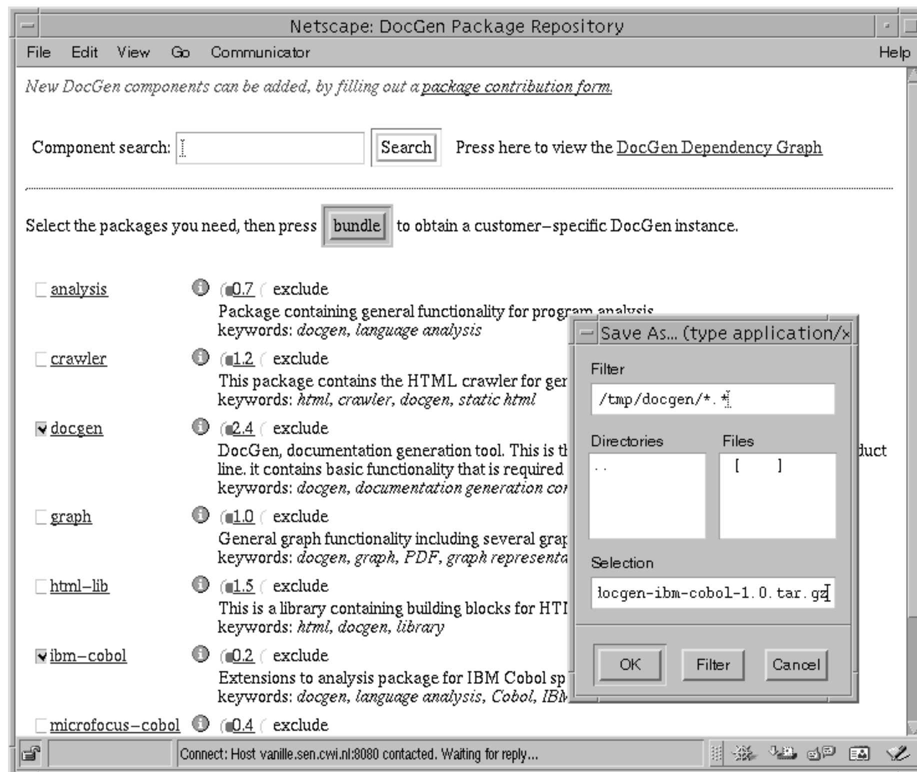

Fig. 4. Screenshot of an experimental online package repository from which implementing DocGen packages can be selected to assemble DocGen product instances.

Package selection can be performed by developers by accessing (online) package repositories (see Figure 4) and selecting the packages of need. By pressing the “bundle” button, `autobundle` is instructed to generate the desired software bundle.

By manually selecting concrete packages from an online package repository, a developer maps a selection of features to a corresponding selection of implementing source code components. This is called product configuration. Manual selection of packages forms an implicit relation between features (in the problem space) and implementation (in the solution space).

This relation between problem and solution space is a many-to-many relation: a single feature can be implemented in more than one source code package; a single source code package can implement multiple features. Selecting a feature therefore may yield a configuration of a single source code package that implements multiple features (to turn the selected feature on), or it may result in bundling a collection of packages that implement the feature together.

The relation between problem and solution space can be defined more explicitly in abstract package definitions. Abstract package definitions then correspond directly to features according to the feature description of a product line architecture (see Figure 5).

```
package
identification
      name=c3-project
    version=1.7
   location=
        info=http://www.software-improvers.com/c3
 description='DocGen instance for customer C3.'
  keywords=customers, c3
configuration interface
requires
 docgen  2.4 with customer=c3
 analysis 1.2 with language=cobol
```

Fig. 5. Example of an *abstract* package definition which forms a mapping from the problem to the solution space.

The 'requires' section of abstract packages defines dependencies upon abstract and/or concrete packages. The latter define how to map (part of) a feature to an implementation component. During package normalization, all mappings are applied, yielding a collection of implementation components.

Like concrete package definitions, the definitions of abstract packages can also be made available via online package bases. Package bases then serve to represent application-oriented concepts and features. Assembling product instances then reduces to selecting the features of need.

4.3 Source Tree Composition in DocGen

To benefit from source tree composition to easily assemble different product instances, the source tree of the DocGen product line needs to be split up. Different parts of the product line then become separate source code components. Every feature as described in the feature description languages should be contained in a single package. This package can be either abstract or concrete. These packages may depend on other core or library packages that do not implement an externally perceivable feature, but implement general functionality.

Implementing each feature as a separate package promises a clean separation of features in the source code. Whether one feature (implementation) depends on another can be easily seen in the feature description. Currently the coupling between the *feature* selection and the selected *packages* is an informal one. More experience is needed to be able to decide whether this scheme will always work.

The source code components that implement a feature or general functionality are internally released as *source code packages*, i.e., versioned distributions of source code components as discussed in Section 4.1. These packages are subjected to an explicit release policy. Reuse of software in different product instances is based only on released source code components. This release and reuse policy allows different versions of a component to coexist seamlessly. Furthermore, it allows developers to control when to upgrade to a new version of a component.

Apart from the packages that implement the individual features, there are several packages implementing general functionality. Of these, the `docgen` package implements the user interface of the software analysis side of DocGen, as well as things like the infrastructure to process files and read them from disk. It also implements the Application Service Provider interface where customers offer their sources over the Internet.

In order to generate the final presentation of DocGen in HTML, a package `html-lib` provides us with the grammar of HTML and a number of interfaces to generate files in HTML. The various graphical representations used in DocGen are bundled in the package `graph` which knows how to present generic graphs as PDF files.

The DocGen source code components are stored in the DocGen package repository (see Figure 4) from which customer-specific source trees are assembled. Compilation of such assembled source trees is performed at the Software Improvement Group to obtain customer products in binary form. The so obtained products are then packaged and delivered to our customers.

4.4 Evaluation

Source tree composition applied to a product line such as DocGen results in a number of benefits. Probably the most important one is that using source tree composition, it is much easier to exclude functionality from the product line. This may be necessary if customers only want to use a "low budget" edition. Moreover, it can be crucial for code developed specifically for a particular customer: such code may contain essential knowledge of a customer's business, which should not be shared with other customers.

A second benefit of using packages for managing variation points is that it simplifies product instantiation. By using a package repository as derived by `autobundle`, features can be easily selected, resulting in the correct composition of appropriately configured packages.

Another benefit is that by putting variable features into separate packages, the source tree is split into a series of separate source code components that can be maintained individually and independently. This solves various problems involved in monolithic source trees, such as: i) Long development/test/integration cycles; ii) Limited possibilities for safe simultaneous development due to undocumented dependencies between parts of the source tree; iii) No version management and release policy for individual components. Explicitly released packages having explicitly documented dependencies help to resolve these issues.

Special attention should be paid to so-called *cross cutting* features. An example is the aforementioned localization feature, which potentially affects any presentation package. Such features result in a (global) configuration switch indicating that the feature is switched on. Observe that the implementation of these features can make use of existing mechanisms to deal with cross cutting behavior, such as aspect-oriented programming [17]: the use of source tree composition does not prescribe or exclude any implementation technique.

```
package docgen;
public class Layout
{
  ...
  String backgroundColor = "white";
  ...
}
```

Fig. 6. Part of the default `Layout` class

5 Managing Customer Code

5.1 Customer Packages

Instantiating the DocGen product line for a particular customer amounts to:

- Selecting the variable features that should be included;
- Selecting the corresponding packages and setting the appropriate configuration switches;
- Writing the customer-specific code for those features that cannot be expressed as simple switches.

As the number of different customers increases, it becomes more and more important to manage such product instantiations in a controlled and predictable way. The first step is to adopt the source tree composition approach discussed in Section 4, and create a separate *package* for each customer. This package first of all contains customer-specific Java code. Moreover, it includes a package definition indicating precisely which (versions of) other DocGen packages it relies on, and how they should be configured.

5.2 Customer Factories

Customer package definitions capture the package dependencies and configuration switches. In addition to this, the Java code implementing the packages should be organized in such a way that it can easily deal with many different variants and specializations for different customers. This involves the following challenges:

- DocGen core functionality must be able to create customer-specific *objects*, without becoming dependent on these;
- The overhead in instantiating DocGen for a new customer should be minimal;
- It must be simple to keep the existing customer-code running when new DocGen variation points are created.

A partial solution is to adopt the *abstract factory* design pattern in order to deal with a range of different customers in a unified way [11]. Abstract factory "provides an interface for creating families of related or dependent objects without specifying their concrete classes". The participants of this pattern include:

- An *abstract factory* interface for creating abstract products;

```
package docgen.customers.greenbank;
public class Layout extends docgen.Layout
{
  String backgroundColor = "green";
}
```

Fig. 7. The `Layout` class for customer Green Bank

- Several *concrete factories*, one for each customer, for implementing the operations to create customer-specific objects;
- A range of *abstract products*, one for each type of product that needs to be extended with customer-specific behavior;
- Several *concrete products*: per abstract product there can be different concrete products for each customer.
- The *client* uses only the interfaces declared by the abstract factory and abstract products. In our case, this is the customer-independent DocGen kernel package.

The abstract factory solves the problem of creating customer-specific objects. Adoption of the pattern as-is to a product line, however, is problematic if there are many different customers. Each of these will require a separate concrete factory. This can typically lead to code duplication, since many of these concrete factories will be similar.

To deal with this, we propose *customer factories* as an extension of the abstract factory design pattern. Instead of creating a concrete factory for each customer, we have one customer factory which uses a customer *name* to find customer-specific classes. Reflection is then used to create an instance of the appropriate customer-specific class.

As an example, consider the class `Layout` in Figure 6, in which the default background color of the user interface is set to "white". This class represents one of the *abstract products* of the factory pattern. The specialized version for customer `greenbank` is shown in Figure 7. The name of this class is the same, but it occurs in the specific `greenbank` package. Note that this is a *Java* package, which in turn can be part of an `autobundle` package.

Since the `Layout` class represents an abstract product, we offer a static `getInstance` factory method, which creates a layout object of the suitable type. We do this for every constructor of `Layout`.

As shown in Figure 8, this `getInstance` method is implemented using a static method located in a class called `CustomerFactory`. A key task of this method is to find the actual class that needs to be instantiated. For this, it uses the customer's name, which is read from a centralized property file. It first tries to locate the class

```
docgen.customers.<current-customer-name>.Layout
```

If this class does not exist (e.g., because no customization is needed for this customer) the Layout class in the specified package is identified.

Once the class is determined, Java's reflection mechanism is used to invoke the actual constructor. For this, an array of objects representing the arguments needed for object instantiation is used. In the example, this array is empty.

```
package docgen;
public class Layout
{
  ...
  public static Layout getInstance()
  {
    return (Layout)CustomerFactory.getProductInstance(
                        docgen.Layout.class, new Object[]{});
  }
  ...
}
```

Fig. 8. Factory code for the `Layout` class.

5.3 Evaluation

The overall effect of customer packages and customer factories is that

– One `autobundle` package is created for each customer, which exactly indicates what packages are needed for this customer, and how they should be configured;
– All customer-specific Java code is put in a separate Java package, which in turn is part of the `autobundle` package of the customer;
– Addding new customers does *not* involve the creation of additional concrete factories: instead, the customer package is automatically searched for relevant class specializations;
– Turning an existing class into a variation point permitting customer-specific overriding is a *local* change: instead of an adaptation to the abstract factory used by all customers, it amounts to adding the appropriate `getInstance` to the variation class.

A potential problem of the use of the *customer factory* pattern is that the heavy use of reflection may involve an efficiency penalty. For DocGen this has proven not be a problem. If it is, *prototype instances* for each class can be put in a hash table, which are then cloned whenever a new instance is needed. In that case, the use of reflection is limited to the construction of the hash table.

6 Concluding Remarks

6.1 Contributions

In this paper we combined three techniques to develop and build a product line architecture. We used these techniques in practice for the DocGen application, but they might be of general interest to build other product lines.

Feature descriptions live at the design level of the product line and serve to explore and capture the variability of a product line. They define features, feature interactions,

and feature constraints declaratively. A feature description thus defines all possible instances of a product line architecture. A product instance is defined by making a feature selection which is valid with respect to the feature description.

Feature descriptions are also helpful in understanding the variability of a product during the development of a product line architecture. For instance, when migrating an existing software system into a product line architecture, they can help to (re)structure the system into source code components as we discussed in Section 4.3.

To assist the developer in making product instances, an interactive graphical representation of feature descriptions (for instance based on feature diagrams or Customization Decision Trees (CDT) [14]), would be of great help. Ideally, constructing product instances from feature selections should be automated. The use of `autobundle` to automatically assemble source trees (see below), is a first step in this direction. Other approaches are described in [14, 7].

Automated source tree composition At the implementation level we propose component based software development and structuring of applications in separate source code components. This helps to keep source trees small and manageable. Source code components can be developed, maintained, and tested separately. An explicit release policy gives great control over which version of a component to use. It also helps to prevent a system from breaking down when components are simultaneous being used and developed. To assist developers in building product instances by assembling applications from different sets of source code components, we propose automated source tree composition. The tool `autobundle` can be used for this. Needed components can be easily selected and automatically bundled via online package repositories.

Package definitions can be made to represent features of a product on a product line. By making such (abstract) packages available via online package repositories, product instantiation becomes as easy as selecting the necessary features. After selecting the features, a self-contained source tree with an integrated build and configuration process is automatically generated.

Customer configuration When two customers want the same feature, but require slightly changed functionality, we propose the notion of customer specializations. We developed a mechanism based on Java's reflection mechanism to manage such customer specific functionality.

This mechanism allows an application to consist of a core set of classes, after source tree composition, that implement the features as selected for this particular product instance. Customer specificity is accomplished by specializing the classes that are deemed customer specific based on a global customer setting. When no particular specialization is needed for a customer, the system will fall back on the default implementation. This allows us to only implement the actual differences between each customer, and allows for maximal reuse of code.

We implemented the customer specific mechanism in Java. It could easily be implemented in other languages.

6.2 Related Work

RSEB, the *Reuse-driven Software Engineering Business* covers many organizational and technical issues of software product lines [13]. They emphasize an iterative process, in which an application family evolves. *Variation points* are distinguished both at the use case and at the source component level. Components are grouped into *component systems*, which are similar to our *abstract packages*. In certain cases component systems implement the *facade* design pattern, which corresponds to a facade package in our setting containing just the code to provide an integrated interface to the constituent packages.

FeatuRSEB is an extension of RSEB with an explicit domain analysis phase [12] based on FODA [16]. The feature model is used as a *catalog* of feature commonality and variability. Moreover, it acts as *configuration roadmap* providing an understanding of what can be combined, selected, and customized in a system.

Generative programming aims at automating the mapping from feature combinations to implementation components through the use of generator technology [7], such as C++ template meta programming or GenVoca [3]. They emphasize features that "cross cut" the existing modularization, affecting many different components. Source tree composition could be used to steer such generative compositions, making use of partially shared configuration interfaces for constituent components.

Customization Decision Trees are an extension of feature diagrams proposed by Jarzabek *et al.* [14]. Features can be annotated with *scripts* specifying the architecture modifications needed to realizing the variant in question. In our setting, this could correspond to annotating FDL descriptions with package names implementing the given features.

Bosch analyzes the use of object-oriented frameworks as building blocks for implementing software product lines [5]. One of his observations is that industrial-strength component reuse is almost always realized at the source code level. "Components are primarily developed internally and include functionality relevant for the products or applications in which it is used. Externally developed components are generatelly subject to considerable (source code) adaptation to match, e.g., product line architecture requirements" [5, p. 240]. *Source tree composition* as proposed in our paper provides the support required to deal with this product line issue in a systematic and controlled way.

In another paper, Bosch *et al.* list a number of problems related to product instantiation [6]. They recognize that it is hard to exclude component features. Our proposed solution is to address this by focusing on source-level component integration. Moreover, they observe that the initialization code is scattered and hidden. Our approach addresses this problem by putting all initialization code in the abstract factory, so that concrete factories can refine this as needed. Finally, they note the importance of design patterns, including the abstract factory pattern for product instantiation.

The use of design patterns in software product lines is discussed by Sharp and Roll [18]. Our paper deals with one design pattern in full detail, and proposes an extension of this abstract factory pattern for the case in which there are many different customers and product instantiations.

The abstract factory is also discussed in detail by Vlissides, who proposes *pluggable factories* as an alternative [19, 20]. The pluggable factory relies on the *prototype* pattern

(creating an object by copying a prototypical instance) in order to modify the behavior of abstract factories dynamically. It is suitable when many different, but similar, concrete factories are needed.

Anastasopoulos and Gacek discuss various techniques for implementing variability, such as delegation, property files, reflection and design patterns [1]. Our abstract factory proposal can be used for any of their techniques, and addresses the issue of *packaging* the variability in the most suitable way.

The use of *attributed* features to describe configured and versioned sets of components is covered by Zeller and Snelting [21]. They deal with configuration management only: an interesting area of future research is to integrate their feature logic with the feature descriptions of FODA and FDL.

6.3 Discussion and Future Work

There is a relation between the features as selected for a particular product instance and the corresponding composition (and configuration) of implementing source code components. We are still investigating how abstract packages can be used to define this relation and how `autobundle` can be used to perform product configuration automatically.

Also the use of online package repositories to assist a developer in assembling product instances is subject to research. From online package repositories containing only abstract packages (which directly correspond to features), developers can select the features of need. With automated source tree composition, the corresponding implementing components are determined and bundled in the intended product instance. It is not clear yet how to handle feature interactions and feature constraints.

The techniques we used in this paper cover customization by means of static and dynamic configuration. Customization can also be based on generative techniques [2, 14, 7]. Incorporating generative techniques (such as frame technology [2, 14]) in the build process of DocGen for customer-specific customization would be interesting future work.

References

1. M. Anastasopoulos and C. Gacek. Implementing product line variabilities. In *Proceedings of the 2001 Symposium on Software Reusability*, pages 109–117. ACM, 2001. SIGSOFT Software Engineering Notes 26(3).
2. P. Bassett. *Framing Software Reuse. Lessons From the Real World*. Yourdon Press (ISBN 0-13327-859-X), 1997.
3. D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *International Conference on Software Reuse*, volume 1844 of *LNCS*, pages 117–136. Springer-Verlag, 2000.
4. K. Beck. *Extreme Programming Explained. Embrace Change*. Addison Wesley, 1999.
5. J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.
6. J. Bosch and M. Högström. Product instantiation in software product lines: A case study. In *Proceedings of the Second International Symposium on Generative and Component-Based*

Software Engineering (GCSE 2000), volume 2177 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

7. K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools, and Applications*. Addison-Wesley, 2000.
8. A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 2001.
9. A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society Press, 1999.
10. Automatic Documentation Generation; White Paper. Software Improvement Group, 2001. http://www.software-improvers.com/PDF/DocGenWhitePaper.pdf.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
12. M. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85. IEEE Computer Society, 1998.
13. I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.
14. S. Jarzabek and R. Seviora. Engineering components for ease of customization and evolution. *IEE Proceedings – Software*, 147(6):237–248, December 2000. A special issue on Component-based Software Engineering.
15. M. de Jonge. Source tree composition. In *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *LNCS*. Springer-Verlag, 2002.
16. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, november 1990.
17. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
18. D. Sharp and W. Roll. Pattern usage in an avionics mission processing product line. In *Proceedings of the OOPSLA 2001 Workshop on Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems*, 2001. See http://www.cs.wustl.edu/~mk1/RealTimePatterns/.
19. J. Vlissides. Pluggable factory, part I. *C++ Report*, November/December 1998.
20. J. Vlissides. Pluggable factory, part II. *C++ Report*, February 1999.
21. A. Zeller and G. Snelting. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398–441, 1997.