Using DITA for Documenting Software Product Lines

Oscar Díaz  
oscar.diaz@ehu.es  
Felipe I. Anfurrutia*  
felipe.anfurrutia@ehu.es  
Jon Kortabitarte  
j.kortabitarte@ehu.es  
ONEKIN Research Group  
University of the Basque Country  
San Sebastián, Spain

ABSTRACT
Aligning the software process and the documentation process is a recipe for having both software and documentation in synchrony where changes in software seamlessly ripple along its documentation counterpart. This paper focuses on documentation for Software Product Lines (SPLs). A SPL is not intended to build one application, but a number of them: a product family. In contrast to single-software product development, SPL development is based on the idea that the distinct products of the family share a significant amount of assets. This forces a change in the software process. Likewise, software documentation development should now mimic their code counterpart: product documentation should also be produced out of a common set of assets. Specifically, the paper shows how DITA process and documents are recasted using a feature-oriented approach, a realization mechanism for SPLs. In so doing, documentation artifacts are produced at the same pace and using similar variability mechanisms that those used for code artifacts. This accounts for three main advantages: uniformity, separation of concerns, and timely and accurate delivery of the documentation.

Categories and Subject Descriptors
D.2.2 [Software Engineering]: Design Tools and Techniques, Management; D.3.3 [Programming Languages]: Language Constructs and Features; I.7.2 [Document and Text Processing]: Document Preparation

General Terms
Design, Documentation, Experimentation

Keywords
documentation, DITA, Software Product Lines, Feature-Oriented Programming

*The corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DocEng '09, September 16–18, 2009, Munich, Germany.
Copyright 2009 ACM 978-1-60558-575-8/09/09 ...$10.00.

1. INTRODUCTION
Preventing mismatches between the application and its documentation counterpart calls for an alignment of the application and documentation processes. As stated in [14], “documentation development is not a standalone process, but one that depends on the software development process... A combined process needs to identify not only documentation’s parallel workflows, but also how the documentation and software workflows integrate”. This implies that a change in the software process should accordingly lead to a change in how documentation is obtained. This is the case of Software Product Lines (SPLs).

SPL is the paradigm for developing a diversity of similar software applications (a.k.a. product family) at low cost, in short time, and with high quality. SPLs emerge as a reuse approach where a set of software applications overlap in their functionality. Clements et al. define a SPL as “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” [5]. For the purpose of this work, a key aspect should be highlighted: the SPL process clearly distinguishes between i) the product line as such (i.e. the core assets development), which defines and realizes the commonality and variability (i.e. features) of the product family, and ii) the product per se, which is derived from the core assets based on the features to be exhibited by this product. This two-step process in building software products of a family is a main departure from single-product approaches.

According to our previous principle (i.e. aligning application and documentation practises), such departure should also impact how product documentation in SPL is produced. If we just follow traditional documentation practises, we will end up with a mismatch between the SPL product and its documentation counterpart [12]. Therefore, the “new software process” brought by SPLs should lead to a change in how SPL documentation is developed. To this end, this paper introduces the Feature-Oriented Documentation Process (FODP).

FODP rests on two main principles:

1. documentation should be managed as any other kind of artifact of the SPL. This principle states that similar mechanisms and processes should be used to handle both code and documentation artifacts. In this way, variability is handled uniformly throughout the artifact spectrum regardless of the type of artifact (i.e. software vs. documentation). This is a main differ-
ence from the current situation where distinct reuse techniques and processes are used to outcome either software products or documentation products.

2. *Features should be the guiding principle to conceive, organise and write documentation.* Even if no document variability exists (e.g., documentation is always targeted to developers in a Web format), how this single document is produced should somehow mimic how the software product is developed. Being in a SPL setting, this mainly implies *documentation should also be built from core assets.*

Therefore, FODP strives to align software and documentation practices. Such alignment can proceed in two ways. The first option is extending the existing software workflow to include documentation workers and activities. A second alternative is creating a new documentation workflow that is tightly integrated with multiple software workflows [19]. This work follows the first approach. Hence the challenge is to include document practises as part of the SPL process.

There exist distinct approaches to both SPL development [5], [16], [23] and documentation practises [19], [9], [21]. Specifically, this work focuses on the *Algebraic Hierarchical Equations for Application Design (AHEAD)* [3] as the SPL approach, and the *Darwin Information Typing Architecture* (DITA) [17] as the documentation practice.

The former, AHEAD, follows an additive approach to variability management where a product is synthesised by composing existing artifact fragments. It was selected due to the core role played by the notion of feature throughout the development cycle, and its extensibility to cope with new types of artifacts.

As for DITA, it is an XML-based architecture for authoring, producing, and delivering technical information. Originally developed by IBM, it was published as an OASIS specification in May 2005. DITA was chosen by its wide popularity and its concern for reusability and variability of product documentation, both aspects highly regarded by the SPL community. However, DITA is meant for traditional, single-product development. It does not contemplate SPLs.

This paper’s contribution is twofold. First, we characterise the role of documentation in SPL where the stress is on variability management rather than on how to describe the documentation itself. Second, we adapt DITA to an additive approach. Hence, the documentation is described using DITA, but the process of obtaining DITA artifacts is achieved *à la AHEAD*, i.e., DITA artifacts are produced by composing existing DITA document fragments. Both code and documentation are handled uniformly and simultaneously. The paper begins with a brief on SPLs.

2. **A BRIEF ON SOFTWARE PRODUCT LINES**

A SPL is not intended to build a single application, but a number of them: a product family. This forces a change in the engineering process where a distinction between domain engineering and application engineering is introduced.

**Domain engineering** (a.k.a. core asset development) determines the commonality and the variability of the SPL. On the other hand, application engineering (a.k.a. product development) produces concrete products out of the core assets. Doing so, the construction of the reusable assets (i.e. core assets) and their variability (a.k.a. variants) is separated from production of the product-line applications.

Variability is a central concept in product family development. Variability permits the generation of different products of the family by reusing core assets. Variability is captured through *features*, i.e. a product characteristic that customers feel is important in describing and distinguishing between members within a family. This feature model is captured during domain engineering and defines the scope of the family. The feature model gets instantiated (a.k.a. configuration model) during application engineering. This configuration model describes the concrete features to be exhibited by a single product.

As a running example, consider the *CurrencyConverter* SPL. The products of the SPL are Web applications for currency conversion\(^1\). The variability offered by this SPL is captured through the feature model depicted in figure 1 using FODA notation [11]. Variations are admitted along three aspects, namely:

- **Functionality**, which includes the variants *dateRate* and *bankRate*. The former allows end users to introduce a date to make the currency conversion with the rates at the given date. As for *bankRate*, it indicates the possibility of adding this concern.

- **Customisation**, which provides the ability for customary users to set some default parameters. Specifically, the *defaultCurrencies*, the *defaultDateFormat* and the *defaultBankRates*. Moreover, the *defaultDateFormat* and the *defaultBankRates* parameters only make sense if the *dateRate* or the *bankRate* features are selected, respectively. This fact is denoted as an existential dependency between *customisation* and *functionality* features. Subsection 6.2 addresses how feature dependencies impact not only software but documentation as well.

- **Audience**, which captures documentation variability based on the targeted reader.

These features can be combined to obtain different currency converters. Figure 2 provides a screen-shot of one such product that exhibits four features. Other converter applications can be obtained by selecting a different set of features. Specifically, seven different converters can be produced. It is worth noticing that a SPL is not just a single product that is feature-based configurable using *ifdef* statements. Such an approach will lead to a thick application that will have an important communication and possible efficiency penalty. By contrast, SPL first produces the core assets, and next, generates a product out of these core assets.

2.1 **SPL documentation**

In comparison with single-product documentation, SPL documentation exhibits four distinctive characteristics:

\(^1\)See [www.oanda.com/convert/classic](http://www.oanda.com/convert/classic) as a working example.
Higher complexity. SPLs aim to build a family of products. Hence, the volume and coupling of artifacts is necessarily larger than if the focus were on a single product, and so is it the documentation.

Longer lifecycles. Being used in different products, SPL designs need to live longer than single-artifact designs which should “only” cater for a single product evolution. Moreover, being re-used many times in distinct products, artifacts must be documented and automated (while single-product design might deserve less stringent requirements).

Wider audiences. SPL-centered organisations tend to be distributed [4], i.e. SPLs introduce a main distinction between domain engineers, responsible for managing the core assets (e.g. the platform and the feature assets), and application engineers, in charge of developing the product out of the core assets. In this setting, platform engineers, feature engineers and application engineers collaboratively produce and exchange documentation which must be understood by all stakeholders involved in the development of a product line and product members.

Larger impact. If single-product documentation is mis-guided, the impact will be restricted to a product. If documentation for a core artifact is inaccurate, the repercussion could affect any product reusing this core artifact. One of the benefits of improving the accuracy of the user documentation is reduction of costs arising from lawsuits. However, cost/benefit analyses for quality-related decisions are often complex, and a careful analysis could not payoff for small developments [10]. But, this is not the case for SPLs where planned reuse ensures that documentation cost will be spread along distinct products. The wider scope of SPLs safeguards the return-of-investment for documentation.

Therefore, the potentially broad number of artifacts, the tangling relationships between these artifacts, the lasting life of these artifacts, and the common separation between domain engineers and application engineers, all put stringent demands on how documentation is managed for SPLs.

3. PROBLEM STATEMENT

Variability is a key notion in SPLs. Distinct studies have addressed how to engineering artifacts for variability. A first distinction is between the subtractive and additive approaches [8]. The former is a successor of the IFDEF presence conditions found in code artifacts where instructions can be conditioned through checks of a feature’s presence/absence in the features being selected (i.e. the configuration model). A single artifact is then “customised” by moving away all those instructions whose conditions are not met by the current configuration model. This is the approach followed by DITA for documentation processing (see below).

By contrast, the additive approach strives to support each feature in a separate artifact. In this way, we distinguish between base and fragment artifacts. Base artifacts account for the commonality of the SPL. On the other hand, fragment artifacts realized a single feature. At production time, base artifacts are composed with those fragment artifacts according to the configuration model. Feature Oriented Programming (FOP) follows this second approach [3] (see below).

This divergence in how variability is handled, i.e. subtractive-based for documentation vs. additive-based for code, hinders a homogeneous approach to software production which includes both code and documentation. Assets, no matter their type (.class, .html or .dita), should be simultaneously produced as part of a homogeneous process that yields the many-faceted representations of a software product. Following subsections provide a brief on both FOP and DITA.

3.1 Feature Oriented Programming

A feature is an increment in program functionality that leverages the final product with the feature. From this perspective, a SPL is characterised by the set of features it supports, i.e. the statement $M = \{f, h, i, j\}$ is meant to describe a SPL $M$ with features $f$, $h$, $i$ and $j$. AHEAD, a framework for FOP, distinguishes features as constants or functions. Constants represent base programs (i.e. the commonality). Feature functions represent program refinements that extend an input program with the feature functionality. At production time, feature functions are applied to constants to deliver the product that exhibits the desired features. This illustrates an additive approach to variability where features (i.e. the variable part) are separately specified from the common part.

AHEAD follows this approach whereby a complex program is developed from a simple program by adding features incrementally using function composition:

\[ i \bullet x \] // adds feature $i$ to base program $x$
\[ j \bullet x \] // adds feature $j$ to base program $x$

where $\bullet$ denotes function composition.
Features are realized using a mixin-like mechanism. Figure 3a shows a base artifact Foo defining variable members \((x, y)\), and methods \((\text{get}X\) and \(\text{get}Y\)\). This base artifact can now be incrementally extended (i.e. refined in AHEAD parlance) by adding a new method \(\text{reset()}\) that extends the functionality of the base with a new feature Feature1. Figure 3b shows such extension using the Jak language [3]. The expression Feature1+Base returns a Java artifact which holds feature Feature1 (see Figure 3c). Likewise, Feature2+Feature1+Base stands for the base being enhanced with features Feature1 and Feature2, where the order of feature composition (i.e. from right to left) can matter.

At first sight, this resembles regular inheritance. Notice however, that Feature1 is not realised through a subclass of Foo. Rather, the very same class Foo is being extended. There are not two classes but a single class that is being incrementally extended to hold a new feature. Furthermore, the class being extended is not fixed at compile time (like in regular inheritance) but decided at composition time. In this way, a feature function behaves like a mixin inheritance, i.e. a class whose super class is parametrised. Since the super class is not fixed until composition, distinct feature functions on different (and unpredictable) order may be composed to yield a class.

However, a software product is more than a set of classes. A product is multi-faceted, i.e. it admits multiple representations (e.g., Java classes, testing cases, SQL scripts, HTML pages, configuration files, etc.). And a feature can need to leverage one or several of those representations to inlay the desired functionality into the base program. Therefore, a feature realization is a composite, i.e. a unit of enhancement (i.e. a functional increment) but decided at composition time. In this way, a feature function behaves like a mixin inheritance, i.e. a class whose super class is parametrised. Since the super class is not fixed until composition, distinct feature functions on different (and unpredictable) order may be composed to yield a class.

![Figure 3: Refinement of a class: a) base feature; b) Feature1; c) Feature1●base composition](image)

So far, AHEAD TS does not address product documentation as part of the development process. There is then a mismatch between the stepwise way of producing code artifacts à la SPL, and the one-shot, afterthought manner of handling documentation at the end of the project. To overcome this situation, documentation should be handle as any other core artifact. Rather than providing our own artifact formats, we strive to adapt existing documentation practices into AHEAD. This is where DITA comes about.

### 3.2 DITA

DITA is an architecture that encapsulates best practises around reuse by reference\(^3\). For the purpose of this paper, these best practises encompass three main principles. First, DITA encourages authors to write in topic-sized chunks. A **topic** is “the smallest independently maintainable unit of content. Topics must be able to stand alone so that they can be understood when they are encountered out-of-context, for example when a user finds the topic through search, an index, or by following a link”\(^{18}\). The notion of topic then brings two main benefits: i) it meant that authors could work independently on their own topics without impeding one another; and ii) it also favours topic reuse across many different publications. Figure 5 shows a task topic definition in DITA.

\(^{18}\)A reference implementation containing a toolkit is available at [http://dita-ot.sourceforge.net](http://dita-ot.sourceforge.net)

\(^3\)A software product is more than a set of classes. A product is multi-faceted, i.e. it admits multiple representations (e.g., Java classes, testing cases, SQL scripts, HTML pages, configuration files, etc.). And a feature can need to leverage one or several of those representations to inlay the desired functionality into the base program. Therefore, a feature realization is a composite, i.e. a unit of enhancement (i.e. a functional increment) but decided at composition time. In this way, a feature function behaves like a mixin inheritance, i.e. a class whose super class is parametrised. Since the super class is not fixed until composition, distinct feature functions on different (and unpredictable) order may be composed to yield a class.

![Figure 4: Composing features as directories](image)
For complex scenarios, the subtractive approach leads to brittle artifacts, cluttered with lots of presence conditions, and feature realization being scattered throughout the artifacts.

This work aims at bringing the additive-approach also to DITA artifacts. Besides homogeneity, this approach accounts for modularity, which in turn, eases the handling of feature interaction (see subsection 6.2).

4. AN ADDITIVE APPROACH TO HANDLING VARIABILITY IN DITA

DITA follows a subtractive approach to variability. However, the additive option is being used to handle variability for SPL code artifacts with clear advantages over the subtractive approach. This work addresses the use of an additive approach for DITA documents in a SPL setting. Such endeavour encompasses distinct issues, namely:

1. variability description, which is achieved through feature models (see [11]).
2. variability support, which is realised through a wide range of techniques (see [7]).
3. variability administration, which describes the processes and staff roles involved (see [4]).

Therefore, an SPL way to document SPLs implies foremost to take variability as the pivotal role also for documentation. And describing, supporting and administering this documentation variability using SPL-like approaches. Following sections address these issues for user documentation (i.e. describing to users the way in which the software product is to be used in order to obtain the desired results), though the results can be extrapolated to other types of documentation (e.g. requirements, design, test).

5. DESCRIBING DOCUMENTATION VARIABILITY

If documentation is to be handled as another facet of a product, documentation variability should be described using the same means that the rest of the SPL, i.e. using feature modeling. Variability concerns for documentation rest on the product itself, the delivery format (PDF versus Web versus embedded help), audience (developer versus administrator versus tester), or learning activity (background learning versus task failure recovery versus troubleshooting). In this way, the very same content can be arranged sequentially (i.e. in a book-like way) or as a hypertext using automatic, context-dependent link generation. Previous examples include one of these metadata, audience, which is set to “developer” (in bold in figure 5 and 6). Both topics and maps can be conditionally processed to include/exclude those parts that do not match with the current profile. At processing time (i.e. application engineering), this profile is specified as a property file, e.g.

\[
\text{prop att="audience" val="developer" action="exclude"}
\]

At runtime time, the processing of topics and maps will exclude those aspects whose audience equals “developer”.

DITA’s conditional processing follows the so-called subtractive approach to variability handling whereby a “customised document” is worked out by moving away all those steps (or topicref) whose conditions are not met by the current profile.
6.1 Feature-oriented structuring of DITA’s topics

The feature model can serve as a guiding principle for topic identification. From this perspective, the self-contained nature of DITA’s topic seems to be well aligned with that of feature. A topic focuses on covering a particular subject, or answer a particular question, regardless of the various places where the topic might end up being read. Likewise, a feature provides a given functionality independently of the final product that will exhibit the feature.

However, topics can serve distinct aims. DITA specialises the notion of topic as follows:

- **concept** topics, which answer “What is...” questions. Concepts provide background that helps readers understand essential information about a product, interface, or task. A feature is a product characteristic that customers feel is important in describing and distinguishing members within a family. From this perspective, each feature can be mapped to a concept topic that describe what the feature is about.

- **task** topics, which address “How do I?” questions, and have a well-defined structure that describes how to complete a procedure to accomplish a specific goal. Features can now be regarded as atomic increments in program functionality. And, a task topic mainly describes step-by-step instructions e.g. how a product can be leveraged to exhibit a given feature or how a feature should be used, that depends on the audience. The task topic includes sections for describing the context, prerequisites, steps, expected results, and other aspects of a task. Feature dependencies can be documented as task prerequisites.

- **reference** topics, which provide quick access to facts. Information needed for deeper understanding of a reference topic or to perform related procedures should be provided in a concept or task topic. Examples include ingredients for food recipes, bibliographic entries for books, API libraries for software products, etc.

The separation of content from context becomes particularly important for SPLs. Indeed, SPL core assets play the role of content to be reused in different products (i.e. the context). From this perspective, topics are well-aligned with the philosophy of core assets.

6.2 Impact of feature dependencies into DITA’s topics and maps

If topics are the documentation counterparts of features, feature dependencies should be considered. Specifically, existential dependencies (denoted by the `<requires>` stereotype in the feature model) can evidence that features are not so independent as initially thought. Our sample case illustrates this situation where the “customisation” feature depends on the “dateRate” feature. Should each feature be mapped to a distinct topic? Or, should both features be merged into a single topic where “customisation” provides additional details about the “dateRate” topic?

This situation can be currently addressed in DITA in different ways. A possibility is to provide a single topic for “customisation + dateRate” where metadata attributes are used to distinguish those fragments that correspond to each of them. These fragments will be included or excluded depending on the feature selection. Figure 7a) illustrates this situation where the single topic `uM` (i.e. userManual) contains a base content which can eventually be enlarged with fragments for features `dateRate` and `customisation`. Note that the part of `customisation` fragment included within the `dateRate` fragment produces a dependency. This approach resembles the `ifdef` solution for handling code variants. The main drawback is that a single author needs to write both topic descriptions whereas the code counterpart can still be assigned to two different teams. This would produce a mismatch between code and documentation.

Another alternative is depicted in figure 7b). Here, the process goes along three steps: 1) a topic with the base description is introduced (e.g. `uMbase`); 2) a topic is defined that supplements the base description (e.g. `uMdateRate` and `uMcustom`) where an inclusion reference is made to the base (i.e. `conref`), thus promoting reuse; and 3) a map is used to...
arrange for the different assembling options through metadata attributes (e.g. product).

This solution moves conditional processing from the topics to the maps. Although it allows for the topic definition to be distributed along the feature teams, in this context it artificially increases the complexity of map authoring as the map is polluted with topic variability. Additionally, the simultaneous inclusion of features dateRate and customisation results in base content being duplicated in the synthesised document. Note also that the part of customisation fragment included within the dateRate fragment is not solved yet.

DITA is based on the premise that “topics must be able to stand alone so that they can be understood when they are encountered out-of-context”. The topic is then a unit of delivery as far as documentation is concerned. However, this does not imply that topics must be written in “a single shot” but they can be gradually described in a stepwise manner as new features increase the awareness about the different shades of the topic. In our sample case, the implementation of the customisation feature is not mapped into a single feature module. Rather, it is split into three features: the base feature named customBase and two derivatives: customDateRate and customBankRate features, which describe additional information about its customisability.

A derivative (in AHEAD parlance) is a feature that is needed only when a particular combination of other interacting features are present in a program [13]. A derivative is not an end-user visible feature, but an internal one. For example, the customDateRate derivative is only needed if and only if customisation and dateRate features are selected. This situation is depicted in figure 7c): topic uM is defined through stepwise refinements base, dateRate and customisation. Each of these topic deltas (i.e. refinements) can be provided by a different team: uM_base by the platform engineers, and uM_dateRate and the pair of uM_customBase and uM_customDateRate by developers in charge of features dateRate and customisation, respectively. At composition time, user • customisation • dateRate • base will deliver the complete topic (i.e. uM). Figure 2 shows the result of this composition. This allows documentation construction to mimic code construction.

Notice that topic delta does not change DITA’s notion of topic but how a topic is constructed. Next section delves into topic delta and map delta.

6.3 Applying feature-oriented programming to DITA’s topics and maps

Stepwise refinement is a way to incrementally define an artifact. When the artifact is source code, a class refinement can introduce new data members, methods and constructors to a target class, as well as extend or override existing methods and constructors of that class. But, what is meant to refine a topic or a map artifact? AHEAD TS does not provide a way to refine topics and maps. However, Batory et al. state the Principle of Uniformity whereby “when introducing a new artifact (type), the tasks to be done are (1) to support inheritance relationships between instances and (2) to implement a refinement operation that realizes mixin inheritance”[3]. Following paragraphs describe how this principle has been realised for DITA artifacts based on our XML composition tool named XAK [1]. XAK provides both an XML composition tool integrated into AHEAD TS and a lightweight XML vocabulary (i.e. namespace) in order to represent a delta (i.e. refinement) in XML documents.

Artifact composition (a.k.a. synthesis) starts with a base artifact and applies deltas (i.e. refinements) to progressively incorporate new features to the artifact. Thus, they are two kinds of artifacts: base artifacts and delta artifacts.

Base artifact. Any topic/map can be a base. The only distinction is that a base provides three additional annotations (see figures 8a and 9a), namely: 1) @xak:module, which serves to indicate those elements that play the role of modules (i.e. fragments that are affected by the variability), and hence, liable to be refined. The value of this attribute is the module identifier, and the element content is the module implementation (i.e. the variant); 2) @xak:feature, which specifies the name of the feature being supported. This attribute is added to the document element (i.e. the root). As a convention, base documents, include the “base” value; and 3) @xak:artifact, which provides the file name of the artifact that is being incrementally defined. Figure 8a defines the base content of the functionalityTask.xml artifact, a task topic. The base states that the only element liable to be refined is <steps> which is identified as the mSteps module. Other elements of this artifact can not be refined.

Delta artifact. A delta artifact embodies a meaningful unit of change, i.e. it encapsulates a set of changes in a base topic or map that should be accomplished unitedly. This delta is specified through the <xak:refines> and <xak:keep-content> tags (see figures 8b and 9b). The former is the root element of a delta document. Its content describes a set of module deltas (i.e. elements annotated with the xak:module attribute) over a given base document (i.e. the xak:artifact attribute). Moreover, the xak:feature attribute specifies to which feature this delta belongs to. Finally, <xak:keep-content> dictates to inherit the content of the module being refined. Back to our sample case, figure 8b) shows the dateRate delta, which refines the functionalityTask.xml artifact. This delta extends the mSteps module (which is set by the base) with new steps. At composition time, dateRate • base yields the final task topic (see figure 8c).

An important remark. Composition cares about the content but not the order. For software artifacts this is normally not a problem: the order of methods within a class conveys no meaning. However, order matters in rendering. Documentation is to be consumed by users, and users care about the order in which content is presented to them. When content is rendered through hypertextual means (e.g. HTML), this could not be an important drawback, since the user is free to browse along the links in any order. However, when the rendering order is fixed (e.g. PDF) this is an important nuisance.

Specifically, the composition of DITA maps restricts additions to be made at either the beginning or the end of the topicref content, but not in between, just in the same way that new lines of code cannot be added to a method body when specialised. This black-box approach is a mandate for code specialisation, but it can lead to convoluted rendering.

This situation is illustrated in figure 2 for the product described through the following composition equation: user • customisation • dateRate • base. Its user manual counterpart will output “What is customisation...” after “Contact
Figure 8: Topic composition in XAK: a) base topic, b) dateRate delta, c) dateRate•base composition.

Figure 9: Map composition in XAK: a) base map, b) dateRate delta, c) dateRate•base composition.

Figure 10: JDeveloper directory structure: base and dateRate feature implementation.

Table 1: Incorporating DITA responsibilities into the SPL organisational chart.
information ...” whereas it would have been more intuitive the other way around.

This situation is due to the base already establishing "Introduction ...”, “Functionality ...”, “Contact Information ...” as a block of topicrefs (see figure 9). Different mechanisms can be envisioned to overcome this situation: refining any of these topicrefs, adding a new topicref at the beginning of this block, or adding a new topicref at the end of this block, but no topicref can be inserted in between.

6.4 Processing of topic/map delta

Once base and deltas are specified in DITA, next step is to integrate these artifacts into AHEAD TS. This implies: 1) extending the SPL directory structure to account for DITA artifacts; and 2) overloading AHEAD’s polymorphic operator with DITA types. As for the former, AHEAD TS does not preclude any directory structure. For our sample case, the default structure provided by JDeveloper is extended with a Documentation folder which in turn, comprises a folder for each different type of documentation (e.g. System and User). These folders hold DITA artifacts (see figure 10). Finally, AHEAD’s operator has been extended to account for DITA types. Now, when the AHEAD engine is fed with a feature equation (e.g. dateRate • base), the outcome is a JDeveloper directory that includes both code artifacts and DITA artifacts ready for deploying both the software product and its documentation counterpart. Figure 9(c) shows the outcome for the DITA sample. It is a normal DITA document. The difference stems from: 1) code and documentation are simultaneous and unitedly obtained. Code can not be generated without the DITA artifacts, and vice versa. The documentation flow and the application flow are merged into a single process; and 2) DITA artifacts are now incrementally obtained by composing deltas on top of base. In this way, DITA engineers mimics AHEAD engineers: different artifact types but the same refinement-based pattern for handling variability.

7. ADMINISTERING DOCUMENTATION VARIABILITY

Synchronising the lifecycle of software and documentation is not just a technical issue. Introducing documentation concerns in SPLs implies to address how documentation roles are framed within SPL roles. Specifically, roles and responsibilities related to DITA include the following [6]:

- **type architect**, which analyses topic types needed to accommodate content being produced, and defines new topic types if needed.

- **topic writer**, which writes and edits topics, according to the topic-type standards established for the project by the XML architect.

- **information architect**, which analyses the overall structure of the content, groups it into topic collections, and defines maps that describe the relationship of topics to each other.

- **build developer**, which processes the DITA source topics into various formats, as needed for product deliverables.

- **information designer**, which establishes the “look and feel” of the output presentation.

These roles need to be integrated into the SPL organisational chart. Different organisational models have been described with different trade-offs [4]. When feature-oriented programming is used for SPL development, the following roles can be distinguished:

- **platform engineers**, which are responsible for setting the common platform shared by all the products of the SPL. This includes not only the technological platform (e.g. DBMS, Web Server and the like), but also programming frameworks, code artifacts, SQL scripts, XML schemas, and in general, any core asset to be reused by all the SPL products. According to the AHEAD terminology, these assets conform the base function (i.e. the values) on top of which feature functions are applied.

- **feature engineers**, which leverage the base with the corresponding feature. As the product is multi-faceted, a feature can need to address distinct facets to accomplish the desired results. Hence, they should master how assets can be refined.

- **application engineers**, which specify the product as a feature equation. They should care about feature incompatibilities although feature grammars are been proposed to facilitate this task [2].

Basically, AHEAD roles can be regarded as a reuse-oriented way to distribute responsibilities. A concern is assigned to either a platform engineer, a feature engineer or an application engineer based on whether this concern affects all, some or just one product of the SPL, respectively. On these grounds, DITA responsibilities can be distributed as indicated in table 1: i) topic types and the general structure of the topics is set at the platform level; ii) rendering mechanisms and aesthetic guidelines are also considered of general interest and thus establish for the whole platform; iii) topic authoring and specialised maps for particular contexts are the duty of the feature engineer; and iv) the final delivery of the document as well as documentation review are left to the application engineer that knows the features to be exhibited by the final product.

8. RELATED WORK

To the best of our knowledge, SPL documentation has been seldom addressed. An exception is the work at [22] where documentation pattern are introduced to facilitate the use of Commercial Off-The-Shelf (COTS) components in SPLs. The motivation rests on software integrators having “difficulties in finding out the capabilities of components, because components are not documented in a standard way. Documentation is often the only way of assessing the applicability, credibility and quality of a third-party component”. To this end, documentation patterns offer guidelines and structure for component documentation while identifying the documentation roles. The stress is more on what to document rather than on how to document. By contrast, our approach strives to align and extend documentation and software development under the same SPL umbrella.

This concern of integrating the lifecycle of both software and documentation is also risen in [20] but for component-based development. Their idea is to document a component by reusing documentation from other abstract compo-
ments while keeping consistency of changes by literate programming. The documentation becomes another characteristic of the component, and documentation reuse is achieved through traditional OO techniques. By contrast, our approach decouples documentation from the software artifact by introducing the feature as a mediator. A code artifact does not have an explicit relationship with the corresponding documentation. But the relationship is implicitly stated by being qualified by the same feature. Furthermore, our approach illustrates how feature-oriented programming can outperform OO by separating content from context which is most important in documentation.

There are also several approaches related to connecting program documentation to program code and maintaining them synchronised. Hartmann et al in [9] and Pierce et al in [15] describe an integrated automated approach to documenting software systems using commercial reverse engineering tools. Their differences are that Hartmann et al produces personalised graphical views for distinct audience and make use of standardised representations such as XML and commonly-available tools, whereas Pierce et al uses UML and Rational Rose as a tool. By contrast, Priestley and Utt propose integrating documentation and development processes in a single-product development by incorporating program documentation as another activity in the Rational Unified Process [19]. We follow this last approach, but our approach integrates a documentation development process in an SPL Engineering process (a.k.a. domain- and application-engineering) using a standardised specification (i.e. DITA) for documentation artifacts and FOP approach as an organisation of those artifacts.

9. CONCLUSIONS

There is increasing evidence that accurate and timely delivery of documentation rests on tightening together documentation and software development processes. This work addresses how current feature-oriented practices for SPL development can be extended to documentation artifacts and roles. The approach is realised by integrating IBM’s DITA into AHEAD Tool Suite. Since software documentation generation is part of the building process, it is guaranteed to be consistent with the system actually implemented. The result is that documentation is no longer an after-thought, but the SPL is documented as new features are added, and documentation assembling mimics code assembling.

Acknowledgements

This work was co-supported by the Spanish Ministry of Science & Innovation under contract TIN2008-06507-C02-01. We thank to Salvador Trujillo and Maider Azanza for their comments on earlier drafts.

10. REFERENCES