Experiences with Constructing and Evolving a Software Product Line with Delta-Oriented Programming

Michael Nieke
michael.nieke@mailbox.org
IT University of Copenhagen
Copenhagen, Denmark

Adrian Hoff
adho@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

Ina Schaefer
i.schaefer@tu-bs.de
TU Braunschweig
Braunschweig, Germany

Christoph Seidl
chse@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

ABSTRACT

A Software Product Line (SPL) captures families of closely related software variants. The configuration options of an SPL are represented by features. Typically, SPLs are developed in a feature-centric manner and, thus, require different development methods and technologies from developing software products individually. For developers of single systems, this means a shift in paradigm and technology. Especially with invasive variability realization mechanisms, such as Delta-Oriented Programming (DOP), centering development around configurable features realized via source code transformation is commonly expected to pose an obstacle, but concrete experience reports are lacking. In this paper, we investigate how DOP and cutting-edge SPL development tools are picked up by non-expert developers. To this end, we report on our experiences from a student capstone SPL development project. Our results show that participants find easy access to SPL development concepts and tools. Based on our observations and the participants' practices, we define guidelines for developers using DOP.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Software evolution**.

KEYWORDS

Delta Oriented Programming, Experience Report, Software Product Line, Evolution, Case Study, Observations, Guidelines

ACM Reference Format:
Michael Nieke, Adrian Hoff, Ina Schaefer, and Christoph Seidl. 2022. Experiences with Constructing and Evolving a Software Product Line with Delta-Oriented Programming. In *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems (VAMOS '22), February 23–25, 2022, Florence, Italy.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3510466.3511271

1 INTRODUCTION

Software Product Line (SPL) engineering aims at realizing families of closely related software systems [19]. Typically, SPLs are developed in a feature-centric manner: features represent configurable conceptual units whose relations are captured by a variability model, such as a feature model [10]. For each feature, code needs to be implemented which realizes the respective variable functionality. For instance, in Delta-Oriented Programming (DOP), delta modules are defined containing delta operations that transform existing code by adding, deleting, and modifying code elements [18]. For developers of single systems, switching to variability-aware programming means a shift in engineering paradigms and applied technology.

Highly invasive variability realization mechanisms, such as DOP, promise to quickly realize an SPL by using an existing product as base and changing its implementation according to (de)selected features. However, developing an SPL is typically expected to be particularly challenging due to the additional dimension of complexity induced by variability [3]. Especially the invasiveness of DOP poses new obstacles for developers. However, experience reports inspecting these expectations are lacking – especially for developers new to the concepts and technology of DOP [5].

To provide more insights in SPL engineering for non-expert developers and with DOP, we report on an educational SPL capstone project with 8 students. The participants implemented an SPL we call `NaviDeltaSPL` which consists of 46 features in 12 versions over a time frame of 3 weeks of full-time work. We provide insights into this project by highlighting how participants picked up the SPL development paradigm. Based on the participants' experiences and our own observations, we derive guidelines for improved development.

With this paper and its contributions, we aid other developers in implementing their SPLs. Additionally, we support the research community by providing an open-source repository of a medium-sized SPL.[1]

2 BACKGROUND

We provide a brief introduction to concepts and technology factoring into the project and our experience report:

Feature Models. Variable and common functionality of an SPL is represented by features. In a feature model, relations between features are captured to define the set of viable feature combinations

[1] https://github.com/TUBS-ISF/NaviSPL

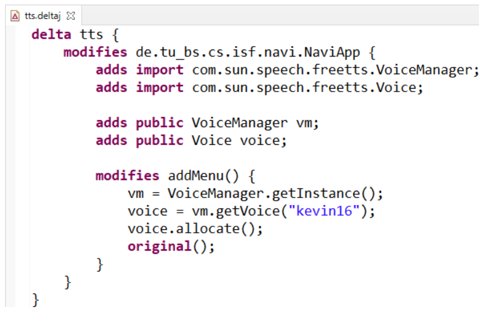
forming the *problem space* of an SPL. Features of a feature model are structured in a tree and each feature has a type: Provided that its parent feature is selected, an OPTIONAL feature *may* be selected whereas a MANDATORY feature *must* be selected [1, 10]. To express relations for sets of child features, OPTIONAL features can be grouped in OR and ALTERNATIVE groups permitting selection of at least one and exactly one feature, respectively. To express more constraints on possible feature combinations apart from the tree structure and types, *cross-tree constraints* can be defined which are Boolean expressions with features as variables.

Delta-Oriented Programming (DOP). Concrete variability realization mechanisms permit expressing variability on code/realization artifact level, i.e., in the *solution space* of an SPL. Three main categories of variability realization mechanisms exist: annotative (or negative), compositional (or positive), and transformational approaches [19]. Delta-Oriented Programming (DOP) is the primary representative for transformational approaches. In DOP, developers apply delta operations in delta modules that transform existing artifacts by adding, deleting, or modifying elements of the artifacts [18]. DOP is an invasive variability mechanism as it can change the inner structure of an existing implementation. To define delta modules and delta operations, dedicated delta languages are required for each target artifact type. For instance, DELTAJAVA allows to define delta modules for Java code.[2] Figure 1 shows an exemplary delta module Δ `tts`, which modifies the class `NaviApp` by adding two new imports and two new class variables. Additionally, the delta module modifies the method `addMenu()`. Method modification results in overwriting the original method body. In this case, three new statements are added. Using the `original()` keyword, the code of the original method before modification can be called. Similarly, a delta operation with the `removes` keyword can remove a classifier or a particular classifier member (constructor, method, field, or constant).

Mapping. A feature-artifact mapping, or short *mapping*, connects problem and solution space, e.g., by mapping combinations of features to realization artifacts, such as delta modules [19]. To this end, a mapping consists of an application condition, i.e., a Boolean expression over features, and a set of mapped realization artifacts. To derive a variant of an SPL, a configuration, i.e., a selection of features, must be defined. Using this configuration and the mapping, relevant realization artifacts are determined based on their satisfied application condition. The variant is then generated out of the determined realization artifacts, e.g., for DOP, by applying the delta operations of determined delta modules.

3 PROJECT SETUP

The intention of the capstone project we report on in this paper was twofold: first, for students to learn SPL engineering and to give proof of their development capabilities with a non-trivial practical subject system, and second, for us to gain insights into the concept and technology adoption process by observing the development of a realistic SPL project. We decided to use a real-world inspired system and to incrementally extend it by configurable functionality that is common in similar systems. In the following, we provide an overview of the capstone project by describing the subject system, the participant group, and their work organization.



```
delta tts {
    modifies de.tu_bs.cs.isf.navi.NaviApp {
        adds import com.sun.speech.freetts.VoiceManager;
        adds import com.sun.speech.freetts.Voice;

        adds public VoiceManager vm;
        adds public Voice voice;

        modifies addMenu() {
            vm = VoiceManager.getInstance();
            voice = vm.getVoice("kevin16");
            voice.allocate();
            original();
        }
    }
}
```

Figure 1: Exemplary delta module modifying the class `NaviApp`.

Subject System. The task for the project members was to implement a navigation system SPL based on an existing open-source map viewer JMAPVIEWER.[3] In its basic form, it allows for showing maps by querying OpenStreetMap, to zoom in and out, and to show markers. Overall, JMAPVIEWER has 4, 353 LOC.

During the project, JMAPVIEWER was extended by configurable features known from other maps and navigation services. In particular, we prompted participants to implement map features, such as a search history, available bus stops, points of interest (POIs), a location's weather, and suggestions for searches. Newly added navigation features are basic navigation, route distance, travel time, differentiation between a pedestrian/car route, navigation instructions via text-to-speech output, speed limits, and carpooling services. In summary, the configurable functionality of the resulting `NaviDeltaSPL` was a mix of easily implementable features, features querying external sources, and more complex features.

Participant Group. For most of the participants, the project was the first encounter with SPL engineering and its concepts. The group consisted of 8 Computer Science students: 7 Bachelor's students and 1 Master's student. In their Bachelor studies, each student participated in a Java programming course and a medium-sized software engineering project. Thus, general programming experience was available but far from practiced. The Master's student also took a lecture on SPLs with mostly theoretical concepts but also small programming tasks with different SPL technologies. Another student was an expert with the theoretical concepts of DOP as he implemented the most current version of DELTAJAVA used in the capstone project.

Tools. For developing the SPL, we prescribed to use the tools DARWINSPL [15] and DELTAJAVA. To provide basic orientation, we held a two-day workshop to introduce SPL engineering concepts and technologies explaining fundamentals of SPLs and feature models. As preparation for their use of DELTAJAVA, we also introduced DOP as variability realization mechanism. For feature modeling, we introduced the tool DARWINSPL with its integrated mechanism for tracking feature-model evolution. In multiple hands-on sessions, we exemplarily applied both tools together with the participants.

Work Organization. As part of the project's results, we wanted to learn which work organization works best for non-expert SPL developers. Thus, in general, participants should organize themselves independently. The project took three full-time weeks and

[2] https://deltajava.org/

[3] https://wiki.openstreetmap.org/wiki/JMapViewer

we expected the team to establish own best practices. We specified a rough Scrum-like development process, which could be adapted dynamically based on gathered experiences. Developers should work in teams of two to perform pair-programming. The role of Scrum master should be filled by one pair as well and rotate between the participants on a regular basis. The main tasks of the Scrum master team were to adapt the feature model as part of evolution, and to monitor the implementation process. Finally, we asked to use one Git branch per feature which would be merged by the Scrum master team. In order for us to evaluate project progression retroactively, we asked the participants to document, for each performed change, its nature, effects, and rationale.

4 OBSERVATIONS AND INSIGHTS

After completing the introductory workshop, the team started to implement the `NaviDeltaSPL` incrementally. In the following, we report on the resulting SPL, the process leading to it, and remarkable observations. We structure our observations by artifact space, i.e., the feature model, the mapping between feature model and delta modules, and the delta modules. For each observation, we define an identifier (for later reference), provide an example, and derive conclusions. These lead to our guidelines in Section 5.

4.1 Resulting Software Product Line

The project's output is a medium-sized open-source SPL. To better understand the SPL's extent, we describe meta data of the resulting feature model and the delta modules.

Feature Model. Overall, we asked to implement 37 features, distributed over multiple requirement requests. The resulting feature model has 12 versions and contains 5 (v1) – 46 (v12) features. The discrepancy between the required features and the actual features of the feature model stems from participants splitting up features to allow for more fine-grained configuration.

Figure 2 shows the feature model after the project's completion (v12). Participants structured the feature model by topics, i.e., one subtree for each of the functionalities for map handling, routing, information provision, and mobility services. The number of newly added features varied heavily with each version. For instance, versions v2, v7, and v9 were the most extensive ones with 6, 8, and 8 added features, respectively, whereas in version v3 only 1 feature was added and version v5 comprises only restructurings. Overall, the entire feature-model history contains 80 feature-model modifications. Apart from feature creations, 35 changes were performed, e.g., moving a feature/group or changing a feature/group type.

At version v12, the feature model contains 14 cross-tree constraints. Out of these, 11 cross-tree constraints modeled dependencies between features due to technical reasons. In addition, 2 constraints restrict variability to prevent non-sensible configurations due to conceptual reasons. In particular, the feature `InfoGroup` creates a visual component that shows information gathered by other features. Thus, it makes sense to select this feature only if at least one information providing feature is selected as well. As a result, the team defined two cross-tree constraints implying information features if `InfoGroup` is selected ("`InfoGroup` $\longrightarrow$ ..."). The remaining constraint ("`LocationImages` $\longrightarrow$ `!Orientation`") was introduced due to implementation reasons as the delta modules mapped to the respective features are incompatible. Overall, the number of valid configurations amounts to 52,860,672.

Delta Modules. The team implemented the `NaviDeltaSPL` with an overall of 45 delta modules (6,382 LOC Java core assets and 4,067 LOC DeltaJava delta modules). To better analyze delta modules, we classified different usage scenarios and defined according terminology. Most of the delta modules (36) are unique *feature delta modules*, i.e., they realize a single feature's functionality and are mapped directly in a 1-to-1 mapping. For similar features, it is sensible to factor out common code to a *shared code delta module*. Here, participants identified only one opportunity to reuse common code, i.e., for the features `ArrivalTime` and `DepartureTime` resulting in the delta module Δ `DepartureArrivalTimeHelper`. Interacting features require "glue code" to properly work together so that additional *feature interaction delta modules* are necessary. Participants implemented 8 of those delta modules, of which 6 were implemented to make certain information providing features (e.g., `TravelTime` or `Weather`) work with the `InfoGroup` feature, which contains GUI functionality to display their results.

4.2 Development Process

As highlighted in Section 3, the team was responsible for organizing itself. In general, team members reported that the feature-centric development paradigm was easy to understand and that thinking in feature increments as a conceptual and implementation unit was natural to them. The notion of variability (i.e., being able to turn certain features on or off) required no more explanation than in the introductory workshop (cf. Section 3).

Process Steps. Figure 3 shows the incremental and agile development process devised by the team. After stakeholders (i.e., we as supervisors) came up with new feature/product requests, the team decided in a common effort which features to implement next. Here, it is important to note that each feature was assigned to exactly one development pair. Subsequently, developers implemented the selected features while the Scrum master pair added the new features to the feature model. It strikes us that the new features were added to the feature model in parallel to implementing them. However, we hypothesize that this was possible only due to the small and quickly implemented feature increments. When a feature's delta module is implemented, it is added to the "integration queue" where it waits for integration by the Scrum master team. Throughout this process, we made the following noteworthy observations:

(Obs. 1) Placing a new feature in the feature model requires both domain and implementation knowledge. When the Scrum master team picked a new feature from the integration queue, the respective developers were consulted. Together, both pairs defined cross-tree constraints and the feature-delta mapping. As the Scrum masters did not know the new feature's implementation details, they were potentially not able to properly insert it in the feature model's hierarchy. Thus, collaboratively with the feature-implementing developer team, they adapted the feature model to put the new feature in the best fitting spot. While this procedure was possible for a medium-scale SPL, for larger and more mature SPLs, we expect that such frequent ad-hoc restructuring would be an indicator for bad planning.

(Obs. 2) Product sampling for testing is informed by domain knowledge, developer experience, and uncertainty of products containing

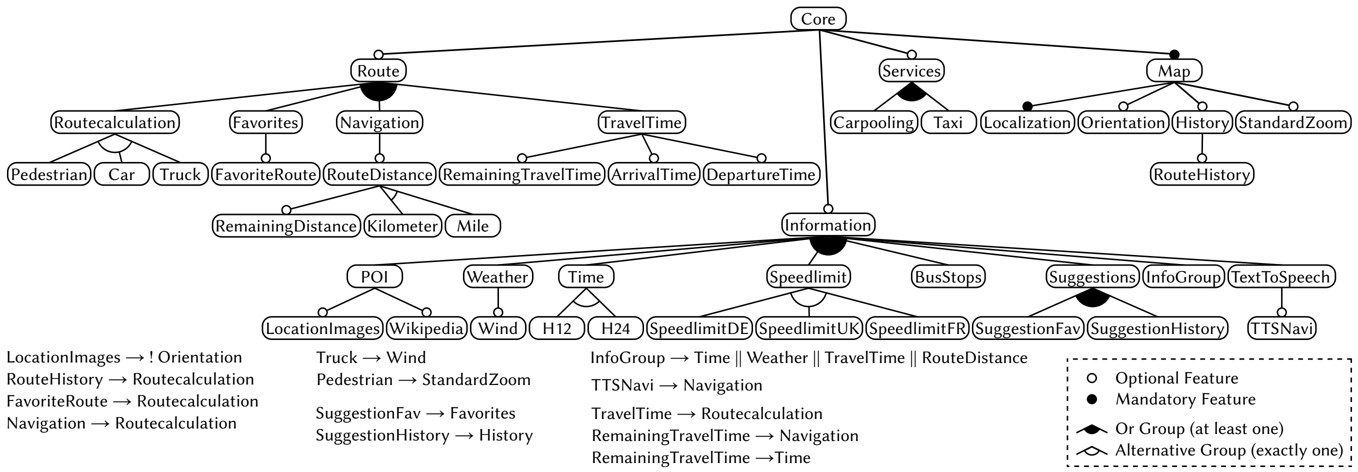


Figure 2: Feature model of the navigation system after the last evolution step.

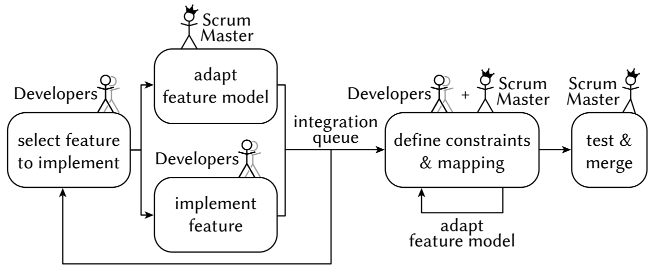


Figure 3: Development process devised by the team.

faults. After feature integration, the Scrum masters tested a new feature using sampled products. To this end, they picked configurations known for having many feature interactions or having exposed problems in the past. To catch other bugs, the Scrum master team also randomly selected several configurations. Finally, if the feature was free from known bugs, they merged the Git branch of the feature into the main branch.

Planning and Evolution. *(Obs. 3) For SPLs, adequate sprint duration is shaped by size and number of the features to be implemented.* In our workshop, we suggested to perform one feature-model evolution step a day. In the daily stand-up meetings, the team selected feature(s) to implement on that very day, which turned out to be a very compact agile process. In this scenario (also due to our short-notice feature requests), the team planned from day to day and, thus, one day resembled one sprint. Due to the high number of implemented features in a short period of time and the mostly small feature size, the sprint duration of one day proved as appropriate. In other projects, the number and size of features per sprint may differ – especially if more extensive quality assurance is performed. We believe that feature number and size are main factors for determining an adequate sprint duration.

Developing with DOP. *(Obs. 4) The concepts of DOP appeared to cause no obstacles for new developers.* In general, developers did not encounter conceptual obstacles with the DOP paradigm. The idea of transforming existing code via delta operations was directly clear to them. Moreover, due to the feature-centric and delta-oriented paradigms, modularization along variation points (i.e., each feature is mapped to its delta module) was also seen as natural.

(Obs. 5) Participants intuitively exploited DOP's capacity to develop features on product level. Developing a delta module on product-line level is challenging as the impact of its delta operations on all products needs to be considered. Thus, it is easier for developers to develop delta modules on product level. That means using a product as base, modifying it, and reversely extracting performed changes as delta operations. The transformational nature of DOP enables this procedure. However, this manual procedure is laborious and error-prone, thus, indicating potential for automation.

(Obs. 6) Implementation pace with DOP significantly exceeded our expectations. Originally, we planned for 24 features. This included smaller and larger features, and features that built upon each other. We anticipated that certain feature requests would require feature model restructurings or introduce feature interactions. After the introductory workshop, we expected the remaining 3 days of that week to be non-productive but dedicated to learning the concepts and tools. However, the team implemented all of our originally planned features within 4 days whereas we planned for 11 days of development. To foster further development, we devised new feature ideas resulting in 37 requested features.

4.3 Feature Model

During the project, the team realized that the feature model is the conceptual heart of an SPL, which is important to keep well-maintained and well-structured. Without this ongoing maintenance, overview is lost and feature dependencies cannot be properly represented by the feature-model structure. In the following, we describe measures the team took to maintain and structure the feature model. We illustrate each measure with an actual example from the `NaviDeltaSPL`'s feature model.

Degree of Variability. *(Obs. 7) There is a trade-off between feature granularity and increasing complexity of the feature model/delta modules.* In the project, we posed a feature request to show the elapsed and the remaining travel time during navigation. The team decided to split this up into two features, one for the elapsed travel time and one for the remaining travel time. As the former feature does not require the feature `Navigation`, they moved both features under the feature `Route` and added a cross-tree constraint "`RemainingTime` ⟶ `Navigation`". In summary, this resulted in a significantly more complex feature model than just adding a feature `TravelTime` under `Navigation`. While this may be desirable to have such a fine-grained SPL, it results in a high complexity and, consequently, worse maintainability.

(Obs. 8) Conservative extension of the configuration space reduces potential technical debt. Originally, team members modeled the feature `Suggestions` as child feature of `Route` assuming that there will be a dependency between those features. At a later point, they learned that this assumed dependency did not actually exist. Consequently, they moved the feature `Suggestions` under the feature `Information` (v11) to make it accessible even without selecting the feature `Route`. Without this conservative future planning of the team and if the dependency would arise as expected, developers would need to restructure the feature model, configurations would become invalid, and, potentially, implementation would need to change. The way the team approached this, they could relax the original constraints to allow for more configurations without impact on existing products. This shows the importance of conservative future planning to not introduce technical debt.

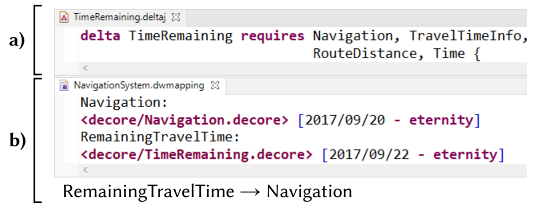
Effect of Evolution. *(Obs. 9) Changing the feature model, even via refactoring, may invalidate existing configurations.* As participants tested their SPL using a set of known critical configurations, they encountered this phenomenon. For instance, a configuration that selects the feature `Speedlimit` became invalid in v6 as an ALTERNATIVE child group was introduced. However, as selecting the new feature `None` reflects the same behavior as before evolution, with sufficient domain knowledge, the choice of an alternative configuration representing similar functionality is obvious. Yet, for real-world SPLs, this would affect configurations of customers. Thus, updating configurations after SPL evolution is required to keep them up-to-date. A method, such as *guided configuration evolution* [16], is required to define configuration update operations by domain engineers who know how configurations are affected by a change.

4.4 Connection of Problem and Solution Space

The mapping serves as main artifact for the connection between problem space and solution space. We gained several valuable insights both by observing the team connecting the two spaces and by retroactively analyzing the defined artifacts.

Structuring and Overview. *(Obs. 10) With an increasing number of features, maintaining the feature-artifact mapping grows significantly more complex.* The mapping file of `NaviDeltaSPL` has 142 LOC (94 LOC for mapping entries, 48 blank lines for structuring). Thus, when searching for mapping entries, the team struggled to find the correct ones. In the end, they concluded that mapping entries need a defined sorting order, but did not have any concrete suggestions. We analyzed the mapping file but did not identify any rules the team used for sorting.

Constraints Scattered Throughout Spaces. *(Obs. 11) Cross-tree constraints permit lifting technical incompatibilities to domain level.*



```
delta TimeRemaining requires Navigation, TravelTimeInfo,
                             RouteDistance, Time {
```

```
Navigation:
<decore/Navigation.decore> [2017/09/20 - eternity]
RemainingTravelTime:
<decore/TimeRemaining.decore> [2017/09/22 - eternity]
```

RemainingTravelTime ⟶ Navigation

Figure 4: Delta module dependency between Δ `TimeRemaining` and Δ `Navigation` is redundantly defined in a) Δ `TimeRemaining` and b) combination of the mapping and feature-model constraints.

Interestingly, not only the mapping served to connect artifacts between the spaces. The team also used cross-tree constraints between features whose mapped delta modules were incompatible. In particular, the delta modules of the features `LocationImages` and `Orientation` are incompatible. Thus, they added a cross-tree constraint preventing the selection of both features ("`LocationImages` ⟶ `!Orientation`", v7). This way, technical incompatibilities manifested on domain level via cross-tree constraints. However, in the cross-tree constraint file, it was not directly clear why this constraint existed as it was mixed with domain cross-tree constraints.

(Obs. 12) Delta-module dependencies may overlap with feature dependencies imposed by the feature model. Figure 4 shows the header of the delta module Δ `TimeRemaining`, which defines a dependency to the delta module Δ `Navigation`. The combination of the mapping shown in Figure 4 with the feature-model cross-tree constraint between the features `RemainingTravelTime` and `Navigation` (cf. Figure 2) results in the same dependency between the delta modules. Thus, this dependency is redundant and distributed over several SPL artifacts resulting in reduced maintainability.

(Obs. 13) Application conditions of mapping entries may overlap with constraints imposed by the feature model. The feature `Mile` does not have a directly mapped delta module (i.e., no 1-to-1 mapping). In fact, the delta module Δ `RouteDistanceMile` has the application condition "`RouteDistance && Mile`". As `RouteDistance` is the parent feature of `Mile`, this application condition is semantically equivalent to a 1-to-1 mapping to `Mile` (if the feature model structure is taken into account). Thus, in the current state of the SPL, this application condition is unnecessarily complex and, thus, reduces maintainability. When thinking about future evolution, this modeling may be sensible as the feature `Mile` may be even relevant for other features than `RouteDistance` so that it would be moved away from below `RouteDistance`. In the end, engineers defining feature models and mappings always need to keep such requirements in mind, and there is a trade-off between current complexity and future flexibility. In contrast to feature and delta module dependencies in Obs. 12, the overlapping constraints of application conditions and feature-model constraints may also contain exclusions of feature combinations.

Unmapped Features. *(Obs. 14) The notion of adding features to a configuration and the manifestation of variation points in DOP may result in a misalignment between features and their implementation.* We identified that certain features do not appear in the mapping. For the first identified unmapped feature `Truck`, we could indeed not find any implementation. One explanation is that `Truck` has been modeled as flag to force the selection of the feature `Wind` via the respective cross-tree constraint if `Truck` is selected. Two other unmapped features are `Car` and `Kilometer`. We found that the respective parent features (`Routecalculation` and `RouteDistance`) were implemented before their child features were even planned. The standard functionality of those features was to calculate a route for cars with kilometers as distance unit. When the child features were introduced, the standard functionality was already present so that only the functionality of alternative sibling features was implemented in new delta modules. That shows a paradigm difference between product-based development of DOP and the notion of feature models and configurations. In DOP, developers start with an existing product containing certain features not only adding new functionality but also *removing* functionality not demanded by a configuration due to delta module application. In configurations, features are selected that define functionality that should be *added* to a product but not removed. The same misalignment also exists for annotative (or negative) variability realization mechanisms.

4.5 Delta Modules

After deciding for features to be realized next, the development teams started to implement them in parallel with the Scrum master team adapting the feature model. To better understand how the team implemented the features, we first investigate the effort the participants took for implementing the individual features and, second, observed *how* they implemented them.

Implementation Effort. As measure for implementation complexity and effort, Figure 5 shows the required implementation time per delta module in days. First, we analyzed how many days the team took to implement a delta module's core functionality (i.e., the code necessary to make the delta module work, but without refactorings or fixes for bugs found later). Second, we analyzed the time span between feature creation in the feature model and implementation of the mapped delta module. The diagram shows that there is variation regarding the required time for implementing a feature, and between feature creation and finishing the core functionality.

To measure core functionality implementation time, we determined the first and last dates of the delta modules' core functionality Git commits. We assume that each delta module took at least 1 day for implementation, i.e., if the first and last core functionality commit were on the same day. As Figure 5 illustrates, for most delta modules, it took 1 day to implement the core functionality. However, several delta modules took significantly more time. For instance, Δ `History` took 7 days to implement. The required time only partially correlates with the size of the delta modules. For instance, it took only 4 days to implement the largest delta module Δ `CarPool`. In summary, the results show that there is significant variation in the time required to implement delta modules.

The time from feature creation to mapped delta module implementation is even more diverse than the pure core functionality implementation time. The largest time span (13 days) was for the delta module Δ `Speedlimit`. Even the child features of the feature `Speedlimit` were added before the delta module had been implemented. Another interesting fact is that some delta modules (Δ `FavoriteRoute`, Δ `GreaterZoom`) had been implemented one day before the mapped feature had been created, resulting in value 0 in Figure 5. We conclude that the reason for this variation is the parallel development process of the feature model, the delta modules, and the integration of queued feature implementations.

Invasiveness of DOP. *(Obs. 15) DOP's independence of explicitly defined variation points requires coordination in collaborative development.* DOP is an invasive variability realization mechanism. In general, each code element of a target language can be modified in order to realize features without explicit variability structures in the target language. Thus, module boundaries of the target language are not respected by delta modules. However, if multiple developers modify one code element or code affecting that element, negative feature interaction can occur. The team experienced that when they "destroyed" each delta modules/features maintained by other participants due to such negative feature interaction. They reported multiple times that this became apparent when testing and that additional bug fixes were necessary – often in collaboration with other development pairs who implemented the conflicting delta modules. Thus, DOP enables quick development of an SPL, but this comes at a price of a lot of testing and communication between developers.

(Obs. 16) DOP can transform only identifiable code elements. Non-identifiable elements require workarounds that break with DOP's principles. By design, delta operations can only transform (implicitly) identifiable slots, e.g., a Java method via its containing class' name and own signature. Elements such as particular statements within a method do not have a unique identifier. Thus, they cannot be addressed to be removed nor can a new statement be added between two existing statements. Similarly, members/variables declared in an original method body are not accessible by delta modules. This results in limitations regarding DOP's invasiveness. Adding, removing, or modifying arbitrary points in the code is not possible. These limitations were hard to understand for the students as, in general, DOP is very invasive and allows for fine-grained transformations.

The participants specifically devised constructs to overcome the limitations of addressable slots by delta operations. To access variables declared in method bodies, they turned many method variables to class variables in the base code. While this procedure enabled participants to access these variables in delta modules, it is non-arguably an anti-pattern.

To change a method's body more flexibly, the team introduced another workaround. With DOP, methods can be modified by exchanging their implementation. It is possible to call the original implementation as well, using a distinct keyword, such as `original()` (cf. Figure 1). As a result, new code can be added only before and after the original implementation but not at arbitrary locations of the original method's body. To overcome this limitation, the team added calls to methods with empty bodies at points in the original implementation where they wanted to add code. They called these empty methods *hook methods* and, by modifying them with delta modules, they were able to add code at predefined positions within an original method's body. This concept is similar to the *Template Method* pattern which is demonstrably used for variability modeling [20–22]. The team came up with this pattern on their own and made it an internal coding guideline. However, the concept of the *Template Method*, which requires to add template method calls at appropriate points in the original methods, and DOP, whose goal it is to leave the base product as-is and not to shape it towards an SPL, contradict each other.

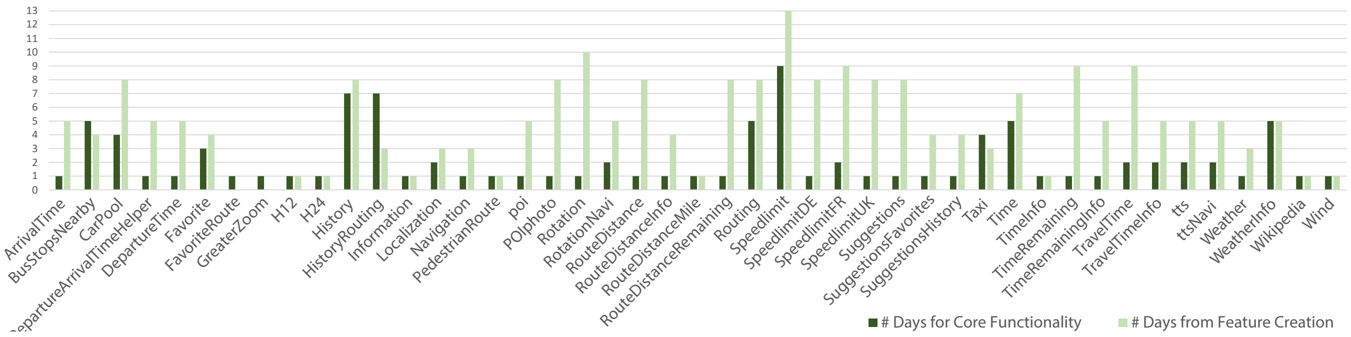


Figure 5: Development time of each delta module in days.

```
public void addMenu() {
    addMenu$Navigation();
    /* Add Menu Code */
}
private void addMenu$Navigation() {
    addMenu$Routing();
    /* Navigation Menu Code */
}
private void addMenu$Routing() {
    /* Routing Menu Code */
    addMenu$Localization();
}
private void addMenu$Localization() {
    /* Localization Menu Code */
}
```

Figure 6: Resulting variant code of three delta modules, each modifying the `addMenu()` method.

Tool Support and DeltaJava. *(Obs. 17) Automated product generation is a prerequisite for quality assurance.* Several challenges for the team were due to the implementation/tool support of DeltaJava. When testing the SPL, the generation of each product for the tested configurations needed to be triggered and executed manually. As a result, the team tested less. They argued that automatic product generation is necessary for effective and efficient testing.

(Obs. 18) Automatically generated products need to preserve code legibility, e.g., to be maintainable. Testing, debugging, and fixing bugs often requires to read and understand the code of an actual product. With DeltaJava, each delta modification of a method containing an `original()` call results in a new separate method in the generated product. For instance, Figure 6 shows the variant code in which three delta modules (Δ `Navigation`, Δ `Routing`, and Δ `Localization`) modify the method `addMenu()`, resulting in four methods. Final products become cluttered by generated methods stemming from delta modules. Consequently, reading and understanding the code of a product is cumbersome, making testing and fixing bugs an even harder task. The team reported that maintenance and identifying feature interactions were problematic due to non-legibility of the generated code. From this, we draw that a major factor for an SPL's quality is legibility of generated code – independently from the used variability realization mechanism.

(Obs. 19) Comfort functionality is an enabling factor for adopting SPL development on product-line level. To implement delta modules, DeltaJava provides its own code editor.Comfort functionality developers are used to from well-established IDEs, such as quick fixes, auto completion, etc., are currently missing in DeltaJava. As a consequence, developing delta modules is currently significantly more intricate than developing standard Java code. Participants frequently mentioned this lack and stated that such functionality is crucial for productive use.

5 GUIDELINES FOR DELTA-ORIENTED PROGRAMMING

Based on the project members' practices, and our own observations and insights (cf. Section 4), we conclude guidelines for beginners that can be used as orientation. For each guideline and advice, we refer to the relevant observation(s).

Feature-Model. *(Gui. 1) Assign Variability Master* Having an engineer with an overview on the entire SPL and its implementation is sensible to maintain the feature model. A dedicated role will help to add new features at the right place in the feature model. (Obs. 1)

(Gui. 2) Reflect on Feature Granularity Modeling features too fine-grained is a pitfall, especially for inexperienced SPL developers, as this may result in too much complexity for maintenance and testing. Additionally, feature-model evolution should be planned conservatively to not allow too many feature combinations which may have to be constrained later. Vice versa, anticipated feature dependencies may affect how the feature model is structured, but if these dependencies do not hold, unnecessary restrictions may exist. Continuously reflecting those design decisions and thorough planning is important to keep the SPL manageable. (Obs. 7, 8)

Problem and Solution Space Connection. *(Gui. 3) Manage the Mapping* Keeping an overview on the mapping is challenging and sorting mapping entries requires domain expertise. Deep integration of the feature model and the mapping editors could help, e.g., by showing all mapping entries for a certain feature via a context menu in the feature model editor. (Obs. 10)

(Gui. 4) Explicate Technical Dependencies/Incompatibilities Technically incompatible delta module combinations should be reflected on feature-model level via cross-tree constraints to model incompatibility of the mapped features. Thus, such incompatibilities are lifted to domain level to prevent the generation of faulty variants. However, the resulting redundancies between feature-model constraints and delta-module dependencies may result in problems during evolution. If respective constraints/dependencies change, all occurrences must be updated. To avoid inconsistencies, tools could provide support to link feature-model constraints with delta

module dependencies, which can be used to keep them synchronized. Moreover, a connection to a bug or ticket system is sensible as the respective constraints should be removed once the incompatibilities/dependencies are removed. (Obs. 11, 12)

(Gui. 5) Reduce Redundancy with Maturity To improve the mapping overview, redundancies between application conditions and feature-model constraints should be eliminated. However, as (during evolution) the respective feature-model constraints may be relaxed and application conditions would not be correct anymore, this should only be done for stable feature-model structures. (Obs. 13)

Delta-Oriented Programming. *(Gui. 6) Avoid Optional-Feature Code in the Core* In DOP, variants are created by transforming a core product. This core product already contains functionality which may also be the functionality of an OPTIONAL feature. Consequently, if the respective feature is selected for a variant, no transformations need to be performed to integrate that functionality. This results in features having no mapped delta modules. Only if such features are not selected, delta modules are applied that remove or alter the core functionality. That makes it hard to identify which code belongs to the feature as it is never explicitly added by a delta module. The respective code has to be identified by analyzing what its deselection would change. Thus, it would be cleaner to extract the respective functionality and add it to a distinct delta module. (Obs. 14)

(Gui. 7) Facilitate Product-Based Development The project participants implemented features on product level by creating a variant, changing it, and manually extracting these changes to delta modules. This procedure is inefficient but it was necessary to make the delta module creation more accessible. Productive usage of DOP requires tool support that automates this procedure. For instance, SiLift [11] retroactively extracts differences between variants and derives respective delta operations. Tools that directly integrate with the editor and transform changes to delta operations would render this additional difference computation obsolete. (Obs. 5)

(Gui. 8) Approach Monotonicity For small SPLs and less experienced developers, DOP seems to be a good method to start with as it is easy to understand. Once the set of valid configurations becomes stable, more complex SPLs could benefit from migrating to more modularized SPL techniques, such as plug-ins or a variable component architecture, to avoid problems stemming from the invasiveness of DOP. As a starting point for such a migration, delta modules could be refactored to achieve monotonicity [8], i.e., they only add code and do not modify or remove elements. (Obs. 15)

(Gui. 9) Teach/Train Good SPL Practices DOP is easy to learn, yet hard to master. Developers need to know bad and best practices as these severely impact the success of an SPL project and its maintainability. In particular, bypassing DOP's limitations to define even more invasive delta modules comes at the price of complex and error-prone code. Explicit variability-oriented development training is necessary for developers to implement *good* SPLs. (Obs. 16)

(Gui. 10) Generate Legible Products White-box testing, debugging, and delta module definition require reading generated products. Thus, apart from comfort functionalities (Obs. 17, 19), tooling for DOP should generate legible code. (Obs. 18)

With these guidelines, we aim at reducing barriers for engineers new to SPL development. Additionally, we identified potential for improvement regarding DOP and respective tools, e.g., DeltaJava.

6 RELATED WORK

SPL development is already common practice in industry, and a multitude of experience reports and case studies exist [4, 7, 12, 13, 17, 23, 24]. However, none of these articles provides insights regarding DOP nor inexperienced developers starting with SPL development. While, to the best of our knowledge, the above-mentioned projects were implemented by experienced developers with established habits and practices, we recruited project participants who were inexperienced with SPL engineering and taught them clean concepts. This enabled us to analyze how SPL engineering with DOP is picked up if no existing habits might influence it. The existing experience reports also lack insights regarding SPL evolution.

For DOP, there is a significant lack of case studies and, even more, publicly available data [5]. Behringer and Fey [2] present how their tool suite PEoPL can be applied to an artificial case study. Helvensteijn et al. [9] present the case study *Fredhopper*, which is realized using the delta modeling capacities of *ABS* [6]. With the *Body Comfort System* case study, Lity et al. [14] provide an extensive source for a delta-oriented SPL using state charts, architecture diagrams, textual requirements, and test cases. However, none of the aforementioned articles reports on the experiences of developing the SPL nor provides experiences on SPL evolution.

Most notably, Camargo et al. [5] report on experiences with using evolution templates for the safe evolution of two medium sized (~5.500 and ~7.500 LOC) SPLs, both implemented with DeltaJava (older version). However, they focus on the use of evolution templates and do not report on general observations on SPL development with DOP, nor on guidelines or suggestions for improvement.

7 CONCLUSION

In this paper, we presented our experiences with a capstone project to implement an SPL using DOP. The results and provided data can be used as basis for further research on constructing and evolving an SPL based on DOP. From our observations, we derived guidelines for developing an SPL with DOP – especially for inexperienced developers. The entirety of our observations leads us to the assumption that DOP seems to be a good method to start with a small or medium-sized SPL even for relatively inexperienced developers. However, the invasiveness of DOP suggests that more complex SPLs should be migrated to more modularized SPL techniques, such as plug-ins or a variable component architecture. With our identified potential for improvement, DOP tools, such as DeltaJava, can advance towards productive usability. In addition, the capacity of DOP to introduce variability to an existing software product paired with a possibility for migrating to other variability realization mechanisms as an SPL matures opens avenues for future research. For instance, how to support startups in customizing their software to different customer configurations rapidly while mitigating potential technical debt accrued by an invasive variability realization mechanism.

ACKNOWLEDGMENTS

We thank all participants of the capstone project, who provided extensive data. This work is partially supported by the DFF (Independent Research Fund Denmark) within the project ISA – Immersive Software Archaeology (0136-00070B).

REFERENCES

[1] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 7–20.

[2] Benjamin Behringer and Moritz Fey. 2016. Implementing Delta-Oriented SPLs Using PEoPL: An Example Scenario and Case Study. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 28–38. https://doi.org/10.1145/3001867.3001871

[3] David Benavides and José A. Galindo. 2014. Variability Management in an Unaware Software Product Line Company: An Experience Report. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 5, 6 pages. https://doi.org/10.1145/2556624.2556633

[4] Ross Buhrdorf, Dale Churchett, and Charles W. Krueger. 2004. Salion's Experience with a Reactive Software Product Line Approach. In *Proc. Int'l Workshop on Software Product-Family Engineering (PFE)*. Springer, 317–322.

[5] Leomar Camargo, Luisa Fantin, Gabriel Lobão, Thiago Figueiredo, Rodrigo Bonifacio, Karine Gomes, and Leopoldo Teixeira. 2021. Evolving Delta-Oriented Product Lines: A Case Study on Feature Interaction, Safe and Partially Safe Evolution. In *Brazilian Symposium on Software Engineering*. 95–104.

[6] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. 2010. Variability modelling in the ABS language. In *Proc. Int'l Symposium on Formal Methods for Components and Objects (FMCO)*. Springer, 204–224.

[7] James C. Dager. 2000. Cummins's Experience in Developing a Software Product Line Architecture for Real-time Embedded Diesel Engine Controls. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 23–45. https://doi.org/10.1007/978-1-4615-4339-8_2

[8] Ferruccio Damiani and Michael Lienhardt. 2016. Refactoring Delta-Oriented Product Lines to achieve Monotonicity. In *Proc. Int'l Workshop on Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*. 2–16. https://doi.org/10.4204/EPTCS.206.2

[9] Michiel Helvensteijn, Radu Muschevici, and Peter Y. H. Wong. 2012. Delta Modeling in Practice: A Fredhopper Case Study. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 139–148. https://doi.org/10.1145/2110147.2110163

[10] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-21. Software Engineering Institute.

[11] Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. 2012. Understanding Model Evolution Through Semantically Lifting Model Differences With SiLift. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 638–641.

[12] Charles W. Krueger, Dale Churchett, and Ross Buhrdorf. 2008. HomeAway's Transition to Software Product Line Practice: Engineering and Business Results in 60 Days. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. 297–306. https://doi.org/10.1109/SPLC.2008.36

[13] Hyesun Lee, Hyunsik Choi, Kyo C. Kang, Dohyung Kim, and Zino Lee. 2009. Experience Report on Using a Domain Model-Based Extractive Approach to Software Product Line Asset Development. In *Proc. Int'l Conf. on Software Reuse (ICSR)*. Springer, 137–149.

[14] Sasche Lity, Remo Lachmann, Malte Lochau, and Ina Schaefer. 2012. *Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study*. Technical Report. TU Braunschweig, Germany. Available at https://www.isf.cs.tu-bs.de/cms/team/lity/bcs_tubs_tech_rep_V1_4.pdf.

[15] Michael Nieke, Gil Engel, and Christoph Seidl. 2017. DarwinSPL: An Integrated Tool Suite for Modeling Evolving Context-aware Software Product Lines. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 92–99.

[16] Michael Nieke, Gabriela Sampaio, Thomas Thüm, Christoph Seidl, Leopoldo Teixeira, and Ina Schaefer. 2021. Guiding the Evolution of Product-Line Configurations. *Software and System Modeling (SoSyM)* (2021). https://doi.org/10.1007/s10270-021-00906-w

[17] Ulf Pettersson and Stan Jarzabek. 2005. Industrial Experience with Building a Web Portal Product Line Using a Lightweight, Reactive Approach. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 326–335. https://doi.org/10.1145/1081706.1081758

[18] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-Oriented Programming of Software Product Lines. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 77–91.

[19] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software Diversity: State of the Art and Perspectives. *Int'l J. Software Tools for Technology Transfer (STTT)* 14 (2012), 477–495. Issue 5.

[20] Sven Schuster, Christoph Seidl, and Ina Schaefer. 2018. Detecting and Describing Variability-Aware Design Patterns in Feature-Oriented Software Product Lines. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018*, Slimane Hammoudi, Luís Ferreira Pires, and Bran Selic (Eds.). SciTePress, 731–742. https://doi.org/10.5220/0006749307310742

[21] Christoph Seidl, Sven Schuster, and Ina Schaefer. 2015. Generative Software Product Line Development Using Variability-Aware Design Patterns. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/2814204.2814212

[22] Christoph Seidl, Sven Schuster, and Ina Schaefer. 2017. Generative software product line development using variability-aware design patterns. *Comput. Lang. Syst. Struct.* 48 (2017), 89–111. https://doi.org/10.1016/j.cl.2016.08.006

[23] Andreas Svendsen, Xiaorui Zhang, Roy Lind-Tviberg, Franck Fleurey, Øystein Haugen, Birger Møller-Pedersen, and Gøran K. Olsen. 2010. Developing a Software Product Line for Train Control: A Case Study of CVL. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 106–120.

[24] Weishan Zhang and Stan Jarzabek. 2005. Reuse without Compromising Performance: Industrial Experience from RPG Software Product Line for Mobile Devices. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. Springer, 57–69.