Lightweight aspect orientated method for software product line feature management

Jakub Perdek

Inštitute of informatics, information systems and software engineering
Faculty of informatics and information technologies
Slovak university of technology in Bratislava

12. December, 2021

Abstract

Generating product variants helps with creation of products from given domain effectively and cheaper. Not only one product is derived, but also easy adaptable products from different environments, languages and according other requirements.

Product derivation can be tailored to requested variability with use of aspect orientated approach, especially AspectJ language, which opportunities we want to observe during fulfilling this task. We chose use of AspectJ language because it enables separate concerns such as logging, caching, pooling, but mainly is possible to exclude given aspects from execution, specific features of application based on them. Its also possible to exclude dependency on this language too.

We are interested in applying this language by enabling configuration of certain features in code, and generating specific software product variant with possible request of implementation independence on this used language. We evaluated benefits and qualities of aspect oriented approach applied on given implementation. Then we compared them with identified problems and benefits mentioned in varying range of reimplementations of existing software systems such as scientific calculator or database system. Provided analysis suggest importance of domain choice during software product creation. Last but not least because of such importance we evaluated influence of given domain on its implementation.

We redesigned existing Battleship game to enable managing features based on existing feature model. Many of depicted features were not implemented. The main goal is analyze possibilities of software product derivation and extent of its dependencies introduced by using AspectJ language.

Next application for analysis is derivation wide range of fractals from original algorithm realized by intervention of aspects into its execution. We assume that task of designing features for given application is determined by value of fractal appearance. For software product

line creation we can request different demands for feature model such as generating all possible derivations for next aesthetic evaluation and identification of the best candidates. Because of it we evaluated potential importance of aspect oriented application for derivation of specific fractals.

1 Introduction

Creation of software products needs to create project and develop given solution according requirements. This is often expensive and needs analysis for every such project. One possible way is to implement software product line to make possible derive such resulting products or components and evolve then on basis of domain knowledge. Some features can be too expensive for customer to pay for them and because of it they are omitted. Others collide with necessary ones and needs to be omitted too.

Appropriate domain analysis is required to design mandatory and voluntary features of the given system to know which products should be possible to generate and if it brings value to customer and his changing requirements. For this purpose we used feature models. In reality many of these models can exist, but only few them can support these changing needs [1]. We think that few relations in feature model can be expressed and solved by aspect and classes almost similarly for each group of them. But sometimes features can collide and some features can't be used together. Because of mentioned problem, its necessary to think about derivation of given product and developing certain feature without colliding ones.

In our work we try to focus on applying aspect oriented language to develop program on feature model basis to serve as software product line. During our work we analyze applied ways how to do it in AspectJ language and its consequences on code quality, encapsulation, modularity benefits and dependencies which arise with using AspectJ language. Its an extension of object orientated language Java and is using join-point model [7]. We used hints and advices from papers which provided refactoring on many systems using aspect orientated languages, especially previously mentioned AspectJ.

Provided software product line and our development techniques can be later applied on fractal construction. New shapes and patterns applied at given positions during recursion drawing of given fractal are voluntary features of given model. To perceive potential aesthetic feeling from some of them is necessary to derive such products. Having appropriate product line enabling generate full range of products for their next validation is the main requirement for this specific task.

Actual state and design of Battleship game is described in section 3. Applied techniques for solution refactoring are presented in section 4. In section 5 we present a game derivation based on actual configuration settings and introducing annotations and expression which helps to choose content

to copy into result project. Section 6 describes how is possible to create all possible derivations based on feature model of given solution. We are using Battleship game again with appropriate feature model for this purpose.

We mentioned possible applications for deriving fractals in section 7. Making the best design of software product line which enables derivation in all possible cases to enrich our perception and produce patterns for next validation is main goal here.

In section 8 we conclude impacts on modularity. We also analyze these impacts using aspect oriented programming and a given case of software product line approach.

Work which is done in given area is described in section 9. We describe some existing approaches that solve similar problems as we have.

Our conclusions and next work we mentioned in section 10.

2 Software product line and domain analysis

Variability is the main part of software product development and products are build by resolving it in a way that can build customer specific products [2]. Software products often emerge from success of market with different needs that can be provided by actual knowledge determined by features, relationships among them and between them and software artifacts which providing implementation of these features. This known actual knowledge sources are used for systematic reuse introduces by software product line engineering [9]. Evolving products then depends on its real word applications which should provide flexible way how to apply changing needs. It can occur that given feature affects several places in models and code which can cause problems during its adaptations with other already included features. Interaction of features can cause a need to generate tailored software artifacts or software artifacts for their next modification for final application. Easier is to configure process by applying change to code in place where components are generated [11].

Aspect-oriented model-driven software product line development provides easier implementation of features as variability representatives. Even their management and ability to assign them to given functionality thanks to the appropriate level of abstraction leads to much greater control and ease of use. This approach is using feature models to separate features and aspects to compose them on module level [10]. Models provides more abstract views on features to be separated and be more effectively managed in comparison with code level basis. They are even better traced according to customer requirements. These requirements then can be captured on domain level of problem space model, which consists on business requirements and end user concerns. This model should be mapped to solution space model. Implementation details are main concepts of solution models. Finally, the

resulting transformations are generated by running the application [10].

AspectJ provides wide range of functionality often much more then other aspect orientated languages. "Its simple, powerful and has dynamic join point model" is characteristics taken from [5]. Also it helps to tailor variability for software product derivation process [13]. Few existing projects tried this language to make parts of software more modular and tested its functionality to evaluate benefits and problems of such language. They are mentioned and summarized in [3]. We chose this language for its benefits and to compare the quality of created software product line after refactoring solution to support given feature model.

Main focus of our work is to provide lightweight method to derive wide range of products from implemented feature model based on using aspect oriented language AspectJ. Our aim is to separate parts of code which are necessary for requested functionality in resulting derivation. We are introducing lightweight method based on annotations to separate such parts and run or enhance them as another application without any other constraints. Derivation can be done by running application. Only modular design is required including separation of concerns where aspects really helps. Our solution should help in case to generate all types of products without much effort to develop such products on their own or evaluate them in some way.

Our method can be applied for both revolutionary and evolutionary development of software product line. It suits more in case of already applied domain analysis to construct feature model. According it we should be able to explicitly formulate expression for each feature if it should be included or not. In revolutionary development developing modular application with all its features is necessary. All products can be derived after specifying mentioned expressions for each feature. Using evolutionary development requires constructing modules for solution incrementally and configure expression for it after its creation. If there is already created application we can choose both development approaches again. There is only need to correctly formulate expressions for features. If we not annotate some, then they will not appear in resulting derivation.

3 Actual state and design of solution in domain application

To provide the most appropriate feature model which can capture potential needs of customer and domain knowledge to serve as basis for our later analysis we choose already provided one. Mentioned feature model is a part of the Battleship game in repository [1] for which depicts its features and is also shown in figure 1. Battleship game is simple game consisting of

[1] `https://github.com/juletx/BattleshipFeatureIDE`

two players trying to destroy ships to each other. Lot of improvements are possible to include as additional features to make the game more interesting. We think that given feature model is based on performed domain analysis and identified many features with decision of their optionality.

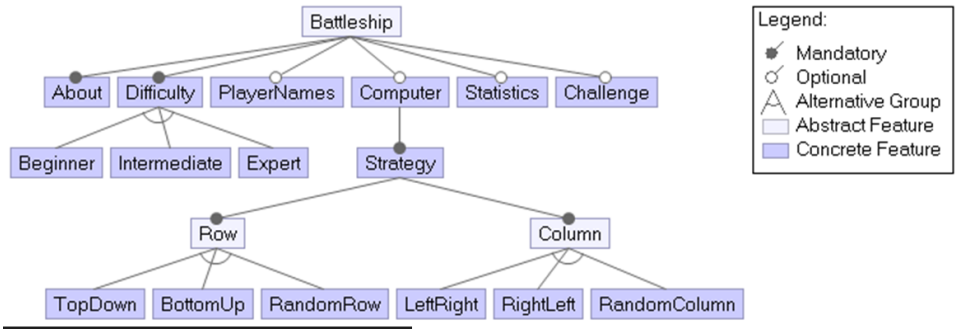

Figure 1: Feature diagram for Java Battleship game

Firstly, we need to mention some problems of provided implementation of Java Battleship game. Only some of mandatory features were provided in unconfigurable way. From feature model we can name features like player names, collecting statistics, computer strategy subtree and challenge as unimplemented. Many features are hardcoded and cannot easily be enhanced without code refactoring. Other aspect of such implementation is that user can't choose one option from given options without modifying code. For example choosing difficulty option in the game. The same code is repeated in many classes and methods. It lacks of some object oriented programming practices. No encapsulation resulting in free access to public variables from any class. Another identified problem was that concerns were not fully separated. For example setup of player was not included in player class. Some static method for its association to given instance should be converted to dynamic ones. This game use only console and not have any graphic user interface. We let console interface as is. Implementation not provides any possibility of product derivation at all. Potential refactoring was required then.

Mentioned Battleship game is not large and not have many features, but can serve as potential board game domain application. We can evaluate its benefits on using aspects and creation of software product line with final product derivation. Separation of concerns is the main benefit observed from development enabling to divide components as separated classes and aspects during product derivation into separated files for each feature and making development easier. Because of aspects, new features often do not need to change existing code and require only to specify new functionality with pointcuts and managing collisions. This functionality can be easily

managed using configuration file with static variables which are used in if condition of given advice. If condition is true, then functionality will be applied otherwise not. Not only extending existing or adding new features are possible using aspects. But configuring existing classes changing their parameters is possible too. These aspects can be also replaced by other aspects which use different strategy. Solution should be more configurable, modular and easily can respond to changes. Not all changes are possible, but with modularity enabling easy product derivation these can be developed separately.

4 Refactoring of domain application using AspectJ language to be based on feature tree

We mentioned necessary changes for our domain application, Battleship game. We applied object oriented principles to make solution modular as possible, extensible, configurable and encapsulated. After changing visibility of variables, adding configuration file for base state of application and moving appropriate content to newly created classes we need to design configurable features. Resulting class diagram for project refactoring is shown in figure 2. During design of such configuration we tried to apply the same approaches which are described in papers focused on refactoring of existing solutions based on aspects, mainly on AspectJ language. Examples are database system [4] and scientific calculator [2]. We also focused on reaching maximum modularity using ApsectJ languages based on approaches from examples presented in book AspectJ in action [6].

Original game contains only main functionality consisting of mandatory features with not configurable game with computer. We added other mandatory features in object oriented way using classes and interfaces. Providing information about authors is example of introduced mandatory feature as one method in main class. For optional features we observed that is better implement them using an aspect oriented way resulting in code scattering prevention and modularity of created software product line.

We based optional features mainly on AspectJ because of extensible functionality. For example feature for setting player names requires add another method before creating players functionality with added player names. If we not used aspects we have to use conditions and it will result in code scattering [6]. There is also not possibility add player name variable to Player class only for this case. Aspects provides modular solution and enables to add new variables and method. It is also possible to configure or exclude them according configuration which can be loaded when application starts or changing values on runtime. Using aspects we only need to specify pointcut which provides mapping of certain location where players are created to executed method. In this aspect method we asked for their names all in one

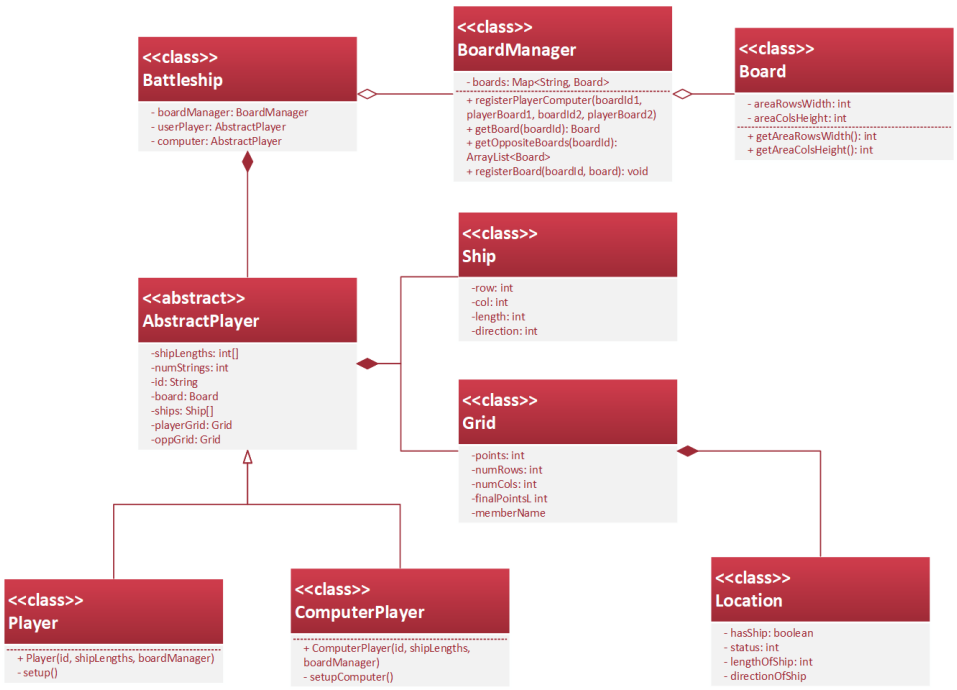

Figure 2: Class Diagram of modified solution

aspect class.

Collecting statistics is another optional domain product feature. We used aspects and classes to implement it. Aspects with pointcuts are used to specify where information should be collected and methods to collect this information. In ordinary classes we implement counter based on hashmap with strings as keys and objects implementing VariableObject interface as values. We created implementation for Integer values as counter for player hits and moves. Then we provided methods and pointcuts to count such variables during playing the game.

We found that aspects well suit on configuration of settings in application. Different implementations of one type of aspect can be used to apply specific configuration which can easily support customer needs. In our product line we only replace parameters of overweighted methods with values from the configuration before calling these methods. We call these overweighted methods with more parameters to apply as much variables for configuration as is possible, but in other application it can depend on specific customer needs.

Optional configuration with computer as player which replaces normal player requires to apply aspects to efficiently separate concerns of computer player from rest of the application in case to not provide such functionality

and to easily segregate such part later. We applied Cuckoo's egg pattern in this case by replacing Player class with Abstract player one. Classes for player and computer player inherited from this abstract class shared functionality which is depicted in figure 3. Using abstract class we can use instances of these classes to represent player and its functionality. If variable computerPlayer is set to true then certain aspect is called to prevent creation of original player instance from Player class by blocking calling constructor of this class. This aspect creates instance of ComputerPlayer class and returns it as result in around method. It serves as substitution to original Player class. Its necessary to mention that applying this functionality requires to add aspect to manage priority because of collisions with difficulty features. Managing difficulty must be applied first because of setting values for next creation of player instance. After it happens then from these given arguments one of possible player instance can be created according configuration or customer needs.

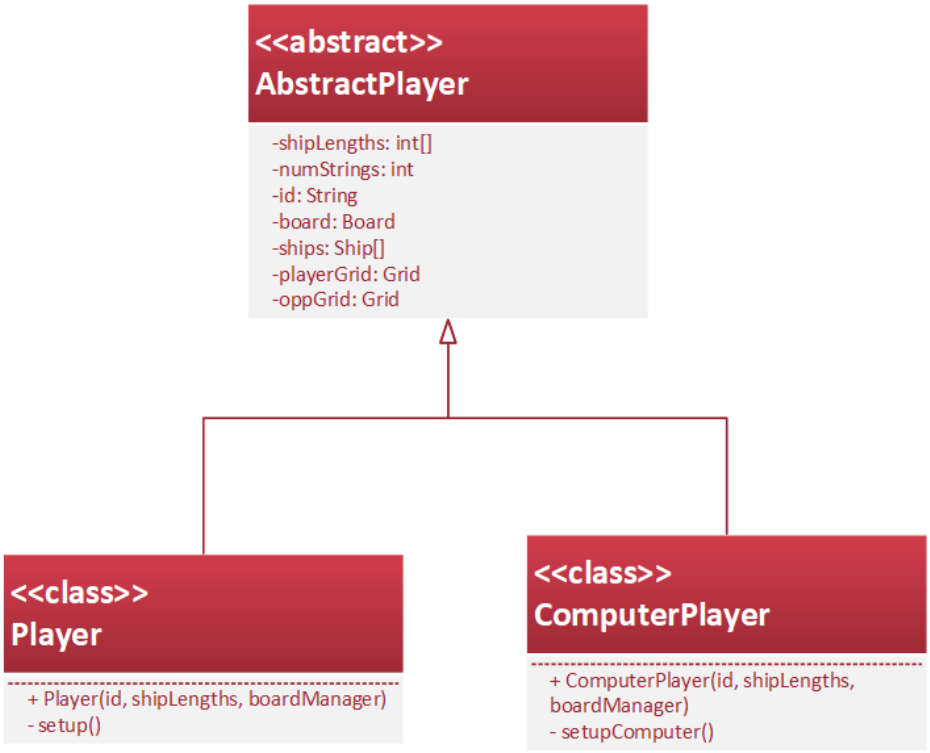

Figure 3: Player abstraction

Evaluation quality of code should show how powerful aspect oriented programming through AspectJ language is. We were inspired by approaches from refactoring papers which applied AspectJ in real world application

and also AspectJ publications. We observed the same benefits, but also problems and complications with using AspectJ as were mentioned by these publications. The main problem of AspectJ is requirement to have certain naming conventions of methods and classes [6], dividing application into appropriate parts for changing their behaviour and hook methods to serve only to hang up aspect on them [4]. All these things point to explicit use of AspectJ language for refactoring of an application. Other issues can arise with certain types of applications. For example in database system using its necessary to use priviledged aspects to access non public variables, but it means also violate encapsulation rules from given class [4]. Aspect behaviour can modify and use of these variables. Aspects improves modularity but it often requires mentioned supervision on design of final solution according with aspect otherwise solution will be less maintainable. The consequences of such a result mean susceptibility to error and complexity [12]. Identified problems on code quality which arise during implementation are shown in figure 4.

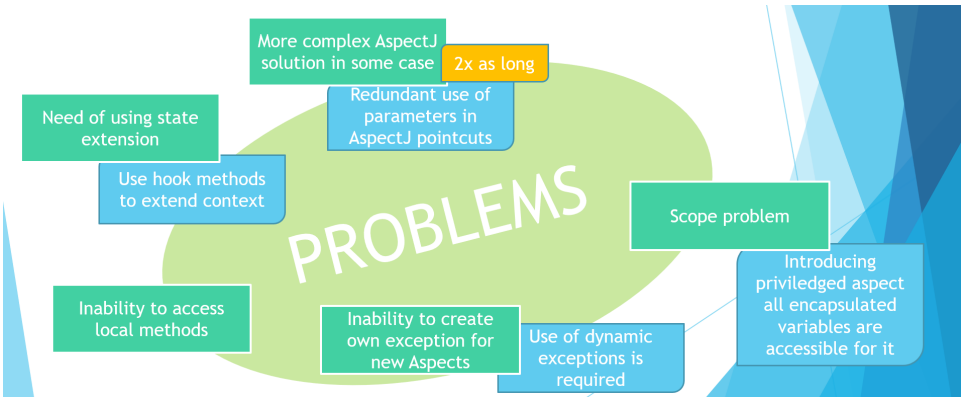

Figure 4: Problems of using AspectJ language

5 Derivation of given software product

If classes and aspects are available there should be question how to separate given features from others except only to configure aspects. We come with idea, that we can copy files or part of them which represents required features. It will be based on feature models as previous part. Mandatory features from the top of the three always should be copied. Other features deeper in feature tree often require from their parents to be included. This creates new problem how to copy such files. We needed rules which can be mappped to features. We again come with idea of creating annotations with expression enabling specify condition when such feature should be copied and when not. Mapping of some features with given values to feature tree

can be seen in figure 5. For example if variable playerNames is true, then all necessary (annotated) classes for this feature will be copied in final solution. If rule will be empty then will be evaluated as true. If any file will be without annotation or all rules have its annotations evaluated as false, then file will not be copied into final solution.

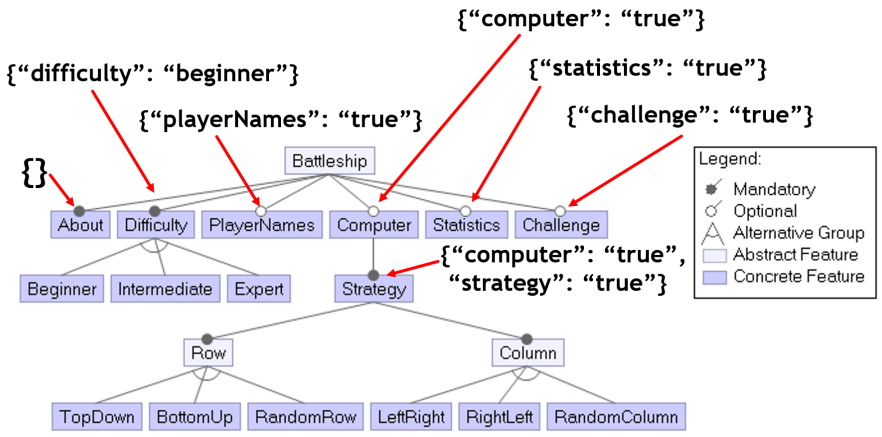

Figure 5: Derivation rules

We specified three types of annotations:

1. The first annotation is //@{}. This annotation can be used to annotate classes, interfaces or aspects to copy whole file. Additional check can be added to program, that one of these descriptive keywords should be included after annotation.

2. The second annotation is //#{}. This annotation is suitable to include or exclude given methods. If file should be copied then at least one condition of annotation should be true. It should not be mixed with first one.

3. The third annotation is //%{}. Using this annotation some imports (line of code) can be included or excluded according including or excluding given method from file. It can be used with second type of annotation.

We mainly use the first type of annotation, but in some cases remaining two should be used. Because that solution is modular it is possible to copy all files for each feature and make derived product functional. We can say so in

situation where splitting some files with many features occurring at one place can be solution, but destroying such entities and their responsibilities at all. Because of it remaining two annotations are used. Content can be managed effectively then. Because we include or exclude methods and they can import classes which in some cases will not be part of final solution its necessary to annotate them using third annotation to be included or excluded. Typical example is functionality for setting names. Two methods are annotated with second annotation. Both of them expects that variable setNames will be true if they should be included. Only second method for setting computer name has another condition in annotation that computerOpponent variable should be set to true, otherwise method will not be copied.

For making more complex condition we enable to use AND and OR operators to specify relation between variables as is displayed in figure 6. We use recursion to evaluate them. If no operators are used, expression is evaluated as AND statement. For AND operator all children should be true. For OR operator at least one should be true.

Figure 6: Complex derivation rule

Class diagram for given solution is shown in figure 7. DerivationManager class manages software product derivation and contain all necessary tools for this operations. For example project copier to copy empty project without classes or FileCopy class to manage reading and writing of given file as stream. During copying it is necessary to find if file should be copied or not. For this purpose program needs to find annotation and evaluate its expression. Searching is done by DerivationAnnotationManager which is using abstract class DerivationAnnotation with all of possible derivation annotation classes which extends this mentioned abstract class and deals with

its own type of processing annotation. Available are methods working with StringBuilder to save string for later use or Writer to write values immediately. Each annotation should check if some keywords are included. For example aspect, interface or class for the first type of annotation. DerivationVariableProcessor class helps DerivationAnnotationManager to evaluate expression associated with annotation. Provided implementation should evaluate expressions with AND and OR operators too. It needs to know used variables and their values to decide mentioned expression correctly. Mapping is provided by ConfigurationVariableManager.

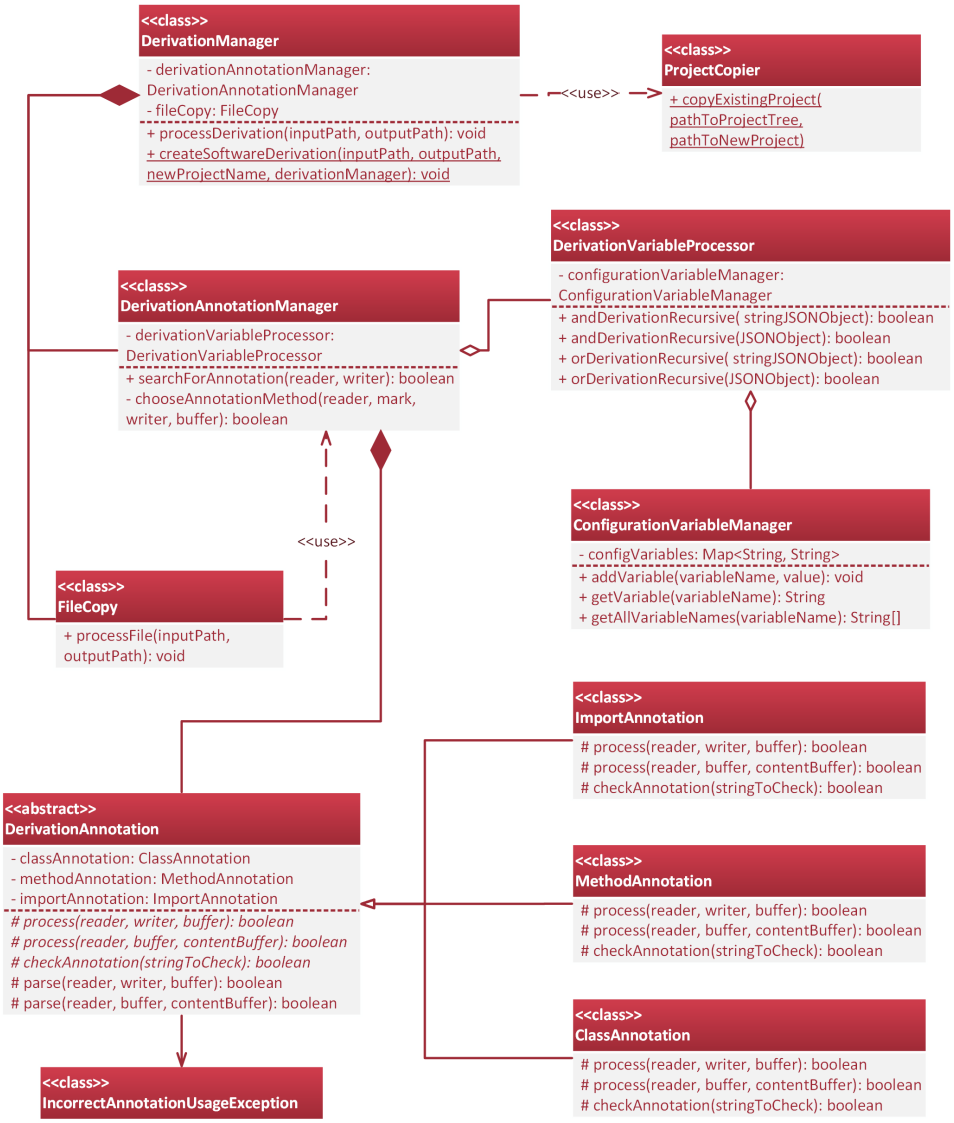

Figure 7: Class diagram for product derivation

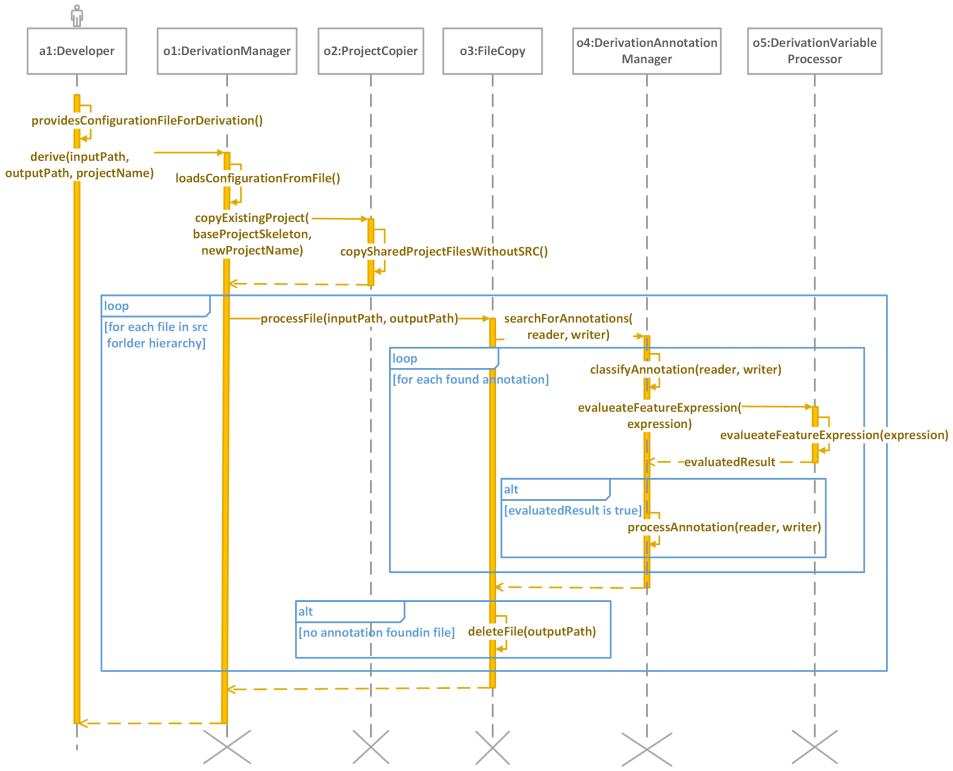

Figure 8: Sequence diagram for product derivation

We also model dynamic behavior for this generator as is shown in figure 8. At the beginning developer chooses features by setting values to configuration file. After the method for derivation is called and configuration file is applied then DerivationManager class firstly copy empty project to specified result place. FileCopy class is used then. For each file in source src directory tries to find annotation using DerivationAnnotationManager. If annotation is found then DerivationVariableProcessor class evaluates condition. If condition is true, annotation is processed using given implementation for given type of annotation. If file does not have annotations or all expressions result in false result then the file will be deleted.

6 Derivation of all types of Battleship games

Feature models help to determine variability in solution. According them it is possible to apply given derivation. We used variables from configuration too, but also other classes that generate all possible derivations. Changing configuration variables was necessary. We need to create another configuration file where variable names are specified with other information about

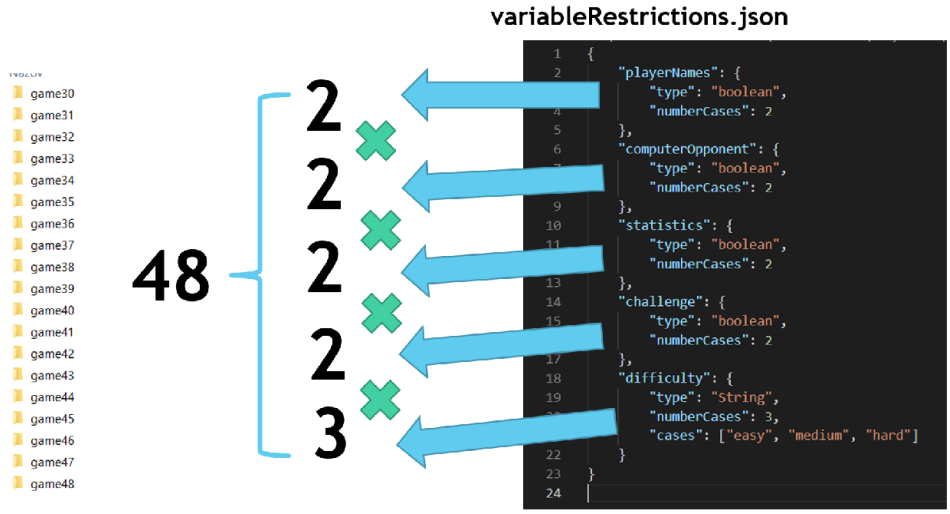

Figure 9: Derivation of all Battleship game types

them. For example variable type and number of possible cases. Variable can have boolean type and all possible cases are two, namely true and false. if variable have string type, cases should be named explicitly in array. For example if we can include different difficulty settings in each derived product then possible cases will be three. Namely easy, medium and hard. We named configuration file as variableRestrictions.json. Generator will generate all configurations for all cases of given variables. Described derivation method will be called for each configuration. In our project we created 2 * 2 * 2 * 2 * 3 = 48 derivations as depicts figure 9.

7 Software product line for fractals

Generating fractals should be interesting application in software product line design. Not only mathematical approach, but also user requirements should be applied during development. Feature models will then vary according these requirements. To evaluate resulting shapes and use their variable basis is necessary to derive such software enabling to draw fractals. All possible derivations should be created for their aesthetic evaluation to see which of them provides higher value to our perception. After this phase its possible to continue with development using certain derivations and enhance their structure.

During fractals analysis we constructed many types of fractals. Some of them are created dividing certain shapes to some extent like Sierpinskeho fractal [8], others extends to space using given iteration parameter. The

length of line not changes during iterations for some fractals. Examples are w-curves or Krishna anklet. Other types of fractals are created using rotation of lines in space. For example Koch snowflake or hexagram. Mentioned algorithm type used to generate fractal can be enhanced by adding more shapes on user request. Not only parameters of algorithm will apply there. Lot of parameters are possible there too. For example choosing radius of circles, or different length of given shapes, width of the line, omitting certain recursion calls or drawing unfilled lines as exchange for given line. With unfilled lines its possible to manage diameter between these lines. We applied given approach drawing w-curves and results shows interesting shapes apart from typical w-curve shape using high value for this diameter. Software product lines can help in such situations to generate all types easily and only evaluate given result remains when launching application is the only one remaining requirement. Figure 10 is showing 3 instances of one type of fractal (anklet fractal) created using different parameters.

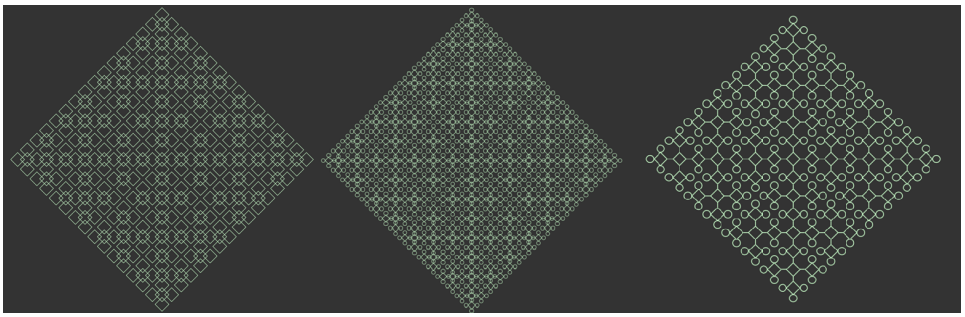

Figure 10: Different anklet fractal shapes

8 Resulting modularity and derived products

Aspects helps to produce modular solutions and enhance functionality without modification of existing code. In case of separating concerns using them can prevent code scattering. Large parts of interacting code for logging, exception handling, third party functionality which is used because of code improvements can be moved to separated classes and let main logic for each concern remain in given place without such code to be directly included. Mentioned parts of such concerns are included during weaving. In case of product line approach we found that is possible to introduce mechanism to easy copy certain parts of given project and let project to be functional and possibly prepared for next development phase.

8.1 Modification of functionality and use of AspectJ language

Developed game using AspectJ shows these mentioned benefits as crucial for purpose of this application. Other benefits using AspectJ enabling easier configuration provided by changing method calls with values provided by chosen approach. This approach can be changed only by adding or removing given aspects from execution. Developing such aspect requires only changing core of few methods of one type of given aspect copy to reach this goal. Aspects well suits for required features too. They well cover variable features. Adding such feature needs to prepare given aspects and join points for them. Application should be developed in respect to changing requirements. Provided code then should be parametrized to enable use of many features and possibility to configure application.

Using aspects not brings only benefits to development and refactoring of existing solution. AspectJ requires to include certain refactoring and code modifications to hang certain aspects on these modified parts. Aspects pointcuts are twice as long and lot of provided parameters are repeated many times. If these pointcuts are modified only in few places their copy is unreadable and is hard to distinguish amongst them. Other crucial problem is need to create empty methods called hooks to hang up certain aspect on them, because aspect oriented methods for changing behaviour of code inside method are missing. In many cases aspects are applied only on one method, not groups of methods. In larger applications it can happen that private variables are exposed to privileged aspects and all encapsulation benefits are lost. AspectJ is not only language witch supports aspect oriented programming, but in my opinion its most powerful way for software product line development including product derivation.

8.2 Generating derivation according to feature model

During our work we created annotations which enable selection of given classes, interfaces and aspects in final derivation. We prepared and connected these classes with configuration file to make such selection to be based on actual configuration. Changing one variable and applying derivation results in creation of different result. We let to generate all possible derivations for all implemented variable features from feature model to prove its functionality. We need only three type of basic annotations to mark classes for next separation. We also developed algorithm to evaluate expression which enables copy given files to result project. Expression consists of variables which identifies features of given feature model which serves as basis for software product line application.

9 Related work

Aspect-oriented model-driven software product line development is extensive approach to derive products according feature models. It uses open source model-driven openArchitectureWare (oAW) toolkit that is integrated into Eclipse and which provides great support for Eclipse Modeling Framework (EMF) [10]. This toolkit includes family of specialized languages such as those for model verification with using constraints (Checks language), artifact generation (Xpand language), model modifications and transformations (Xtend language). Each of these languages has common expression language and type system [10]. Models, especially EMF, are stored in .XMI files. Applying such tools seems to be complex.

Our method is simpler, because it only needs to load configuration file for features and appropriately annotate given classes, aspects and other parts of code. For final decision if class, aspect or method should be included in final derivation, expression in annotation with given variables should be explicitly mentioned with appropriate values. This is explicit step which should be done correctly otherwise resulting solution can be dysfunctional in case of missing functionality not appended to it such as missing classes and methods. In comparison with described approach using oAW toolkit our solution is focused on development explicitly based on feature models. We are not doing any implicit transformation between problem space and solution space. Our solution is based on developing application according feature model in modular way and after it is necessary to appropriately annotate created classes, methods and aspects for next automatic derivation. Logic associations from feature tree are explicitly used here and usually follows simple pattern from feature model hierarchy with possibility to include other conditions for colliding features.

10 Conclusions and future work

In our work we introduced approach which can be used to simple derive wide range of software products as other runnable applications for their next improvements or validation. Method is simpler then other analyzed approaches but modular design and separation of concerns must be explicitly managed. Aspect oriented programming helps with these problems, but the end solution applications can be highly dependent on it. We created example using already prepared feature model to design application which can derive requested derivations according previous configuration. The derivation process is supporting previous implementation of most types of relations from this feature model.

Final application includes configuration of features, product derivation mechanism based on annotations applied by user on certain classes and

methods, implementation of features according feature model and possibility to generate all given software product types. We generated all 48 possible derivations of product example from given domain according feature model and tested their functionality. Its also possible to launch certain features directly using base project. Setting configuration values in configuration file is required for this operation.

Our next focus is to apply this introduced product line to derive all possible derivations for given fractals and validate them according metrics which support aesthetic perception of given users or using other assumptions. For example about symmetry or forming patterns in space. We assume that product line should support derivation of all such features or can serve as next basis for another product line to implement contradictory features efficiently. Main benefit of provided approach is to develop and derive software products fast and effectively using aspect oriented way which should have potential in such type of application. Already provided aspect orientated solutions also can be fastly enhanced to support extraction of such features using annotations.

References

[1] D. Beuche and M. Dalgarno. Software product line engineering with feature models. page 7, 2006.

[2] G. Botterweck, K. Lee, and S. Thiel. Automating product derivation in software product line engineering. page 6, 2009.

[3] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor, and F. Dantas. Evolving software product lines with aspects: An empirical study on design stability. pages 261–270, 01 2008.

[4] C. Kastner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *11th International Software Product Line Conference (SPLC 2007)*, pages 223–232. IEEE, 2007.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In G. Goos, J. Hartmanis, J. van Leeuwen, and J. L. Knudsen, editors, *ECOOP 2001 — Object-Oriented Programming*, volume 2072, pages 327–354. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. Series Title: Lecture Notes in Computer Science.

[6] R. Laddad. *AspectJ in action: practical aspect-oriented programming*. Manning, 2003. OCLC: ocm53049913.

[7] M. Marin, A. Deursen, L. Moonen, and R. Rijst. An integrated crosscutting concern migration strategy and its semi-automated application to jhotdraw. *Autom. Softw. Eng.*, 16:323–356, 06 2009.

[8] R. Pelánek. *Programátorská cvičebnice*. Computer press, 1. vydání edition, 2012.

[9] I. Reinhartz-Berger, A. Sturm, T. Clark, S. Cohen, and J. Bettin, editors. *Domain Engineering, Product Lines, Languages, and Conceptual Models*. Springer, 2013.

[10] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *11th International Software Product Line Conference (SPLC 2007)*, pages 233–242. IEEE, 2007.

[11] V. Vranic and R. Táborský. Features as transformations: A generative approach to software development. 13(3):759–778, 2016.

[12] R. Yokomori, H. Siy, N. Yoshida, M. Noro, and K. Inoue. Measuring the effects of aspect-oriented refactoring on component relationships: Two case studies. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, AOSD '11, page 215–226, New York, NY, USA, 2011. Association for Computing Machinery.

[13] T. J. Young and B. Math. Using AspectJ to build a software product line for mobile devices. page 73, 1999.