Lightweight method for software product line features management

Jakub Perdek

Inštitút informatiky, informačných systémov a softvérového inžinierstva
Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave

20. Október, 2021

Abstrakt

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 [1]. Product derivation can be tailored to requested variability with use of aspect orientated approach, especially AspectJ language [8], 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 [4].

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 (examples of different types can be found in [5]) 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. 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.

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 about features, then from relationships among features and relationships with software artifacts implementing these features. This known actual knowledge sources are used for systematic reuse introduces by software product line engineering [6]. 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 modify process applying change to procedure that generates these components [7].

3 Overview of sections

Actual state and design of Battleship game is described in section 4. Applied techniques for solution refactoring are presented in section 5. In section 6 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 7 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 8. 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 9 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 10. We describe some existing approaches solving similar problems that we have.

Our conclusions and next work we mentioned in section 11.

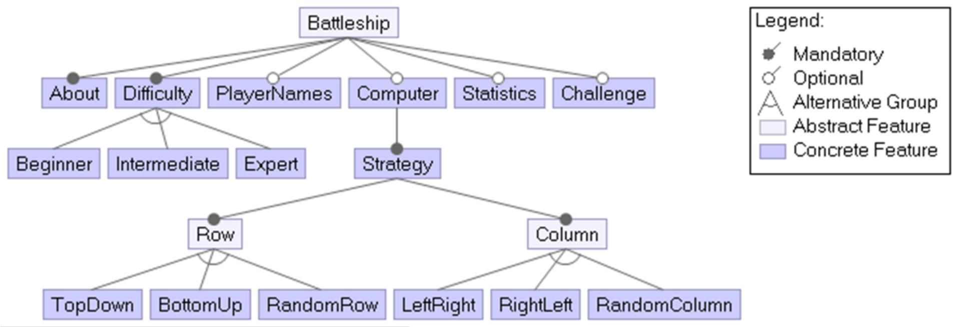
4 Actual state and design of solution in domain application

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. Battleship game is simple game consisting of 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.

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.

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



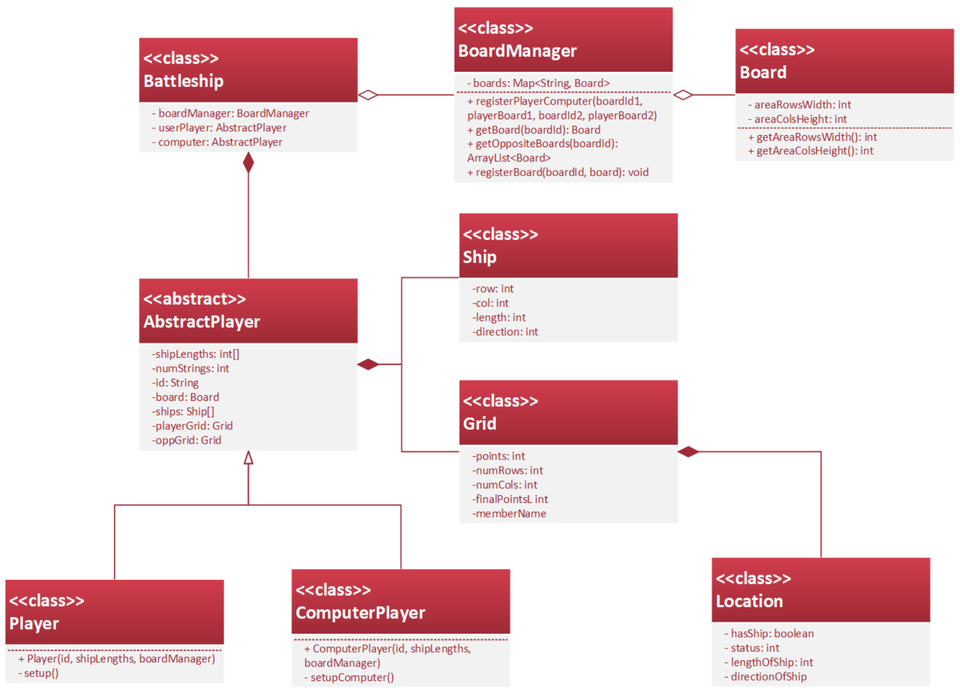
Obr. 1: Feature diagram for Java Battleship game

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

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 papares focused on refactoring of existing solutions based on aspects, mainly on AspectJ language. Examples are database system [3] 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 [4].

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 [4]. 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



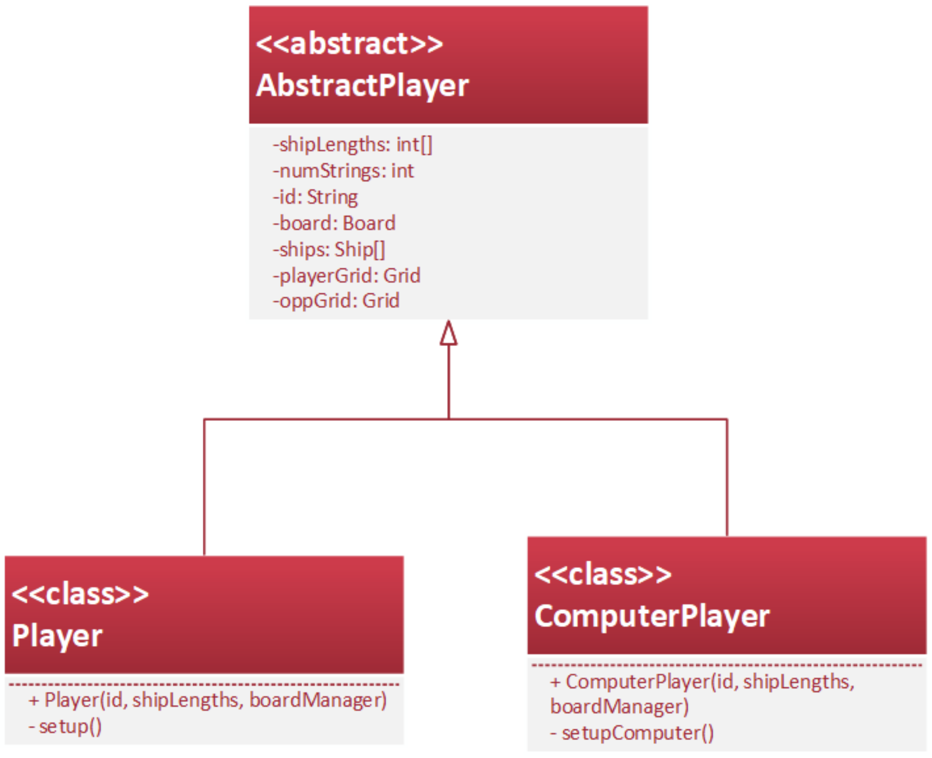
Obr. 2: Class Diagram of modified solution

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 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.

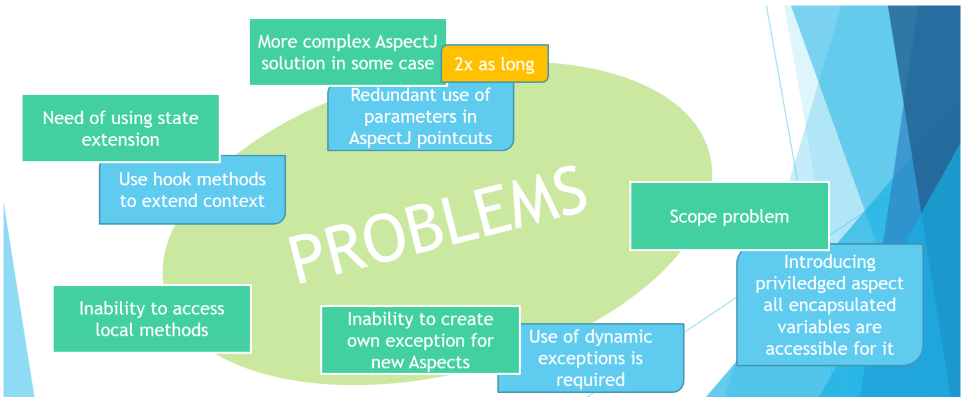
We found that aspects well suit on configuration of settings in application. Different implementations of one type of aspect can be used to apply specific configuration which can easily support customer needs. In our product line we only replace parameters of overweighted methods with values from the 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 necesary 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.



Obr. 3: Player abstraction

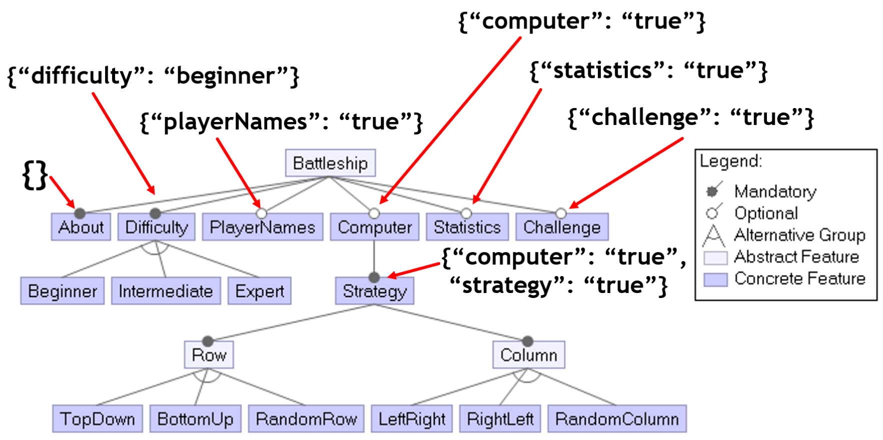
Po 4.


More complex AspectJ solution in some case
2x as long
Redundant use of parameters in AspectJ pointcuts
Need of using state extension
Use hook methods to extend context
PROBLEMS
Scope problem
Introducing priviledged aspect all encapsulated variables are accessible for it
Inability to access local methods
Inability to create own exception for new Aspects
Use of dynamic exceptions is required

Obr. 4: Problems of using AspectJ language

6 Derivation of given Battleship game

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 on picture 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.



Obr. 5: Derivation rules

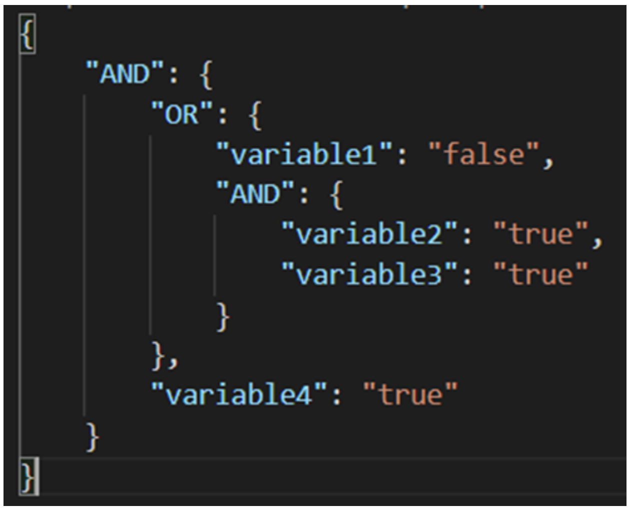
We specified three types of annotations:

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



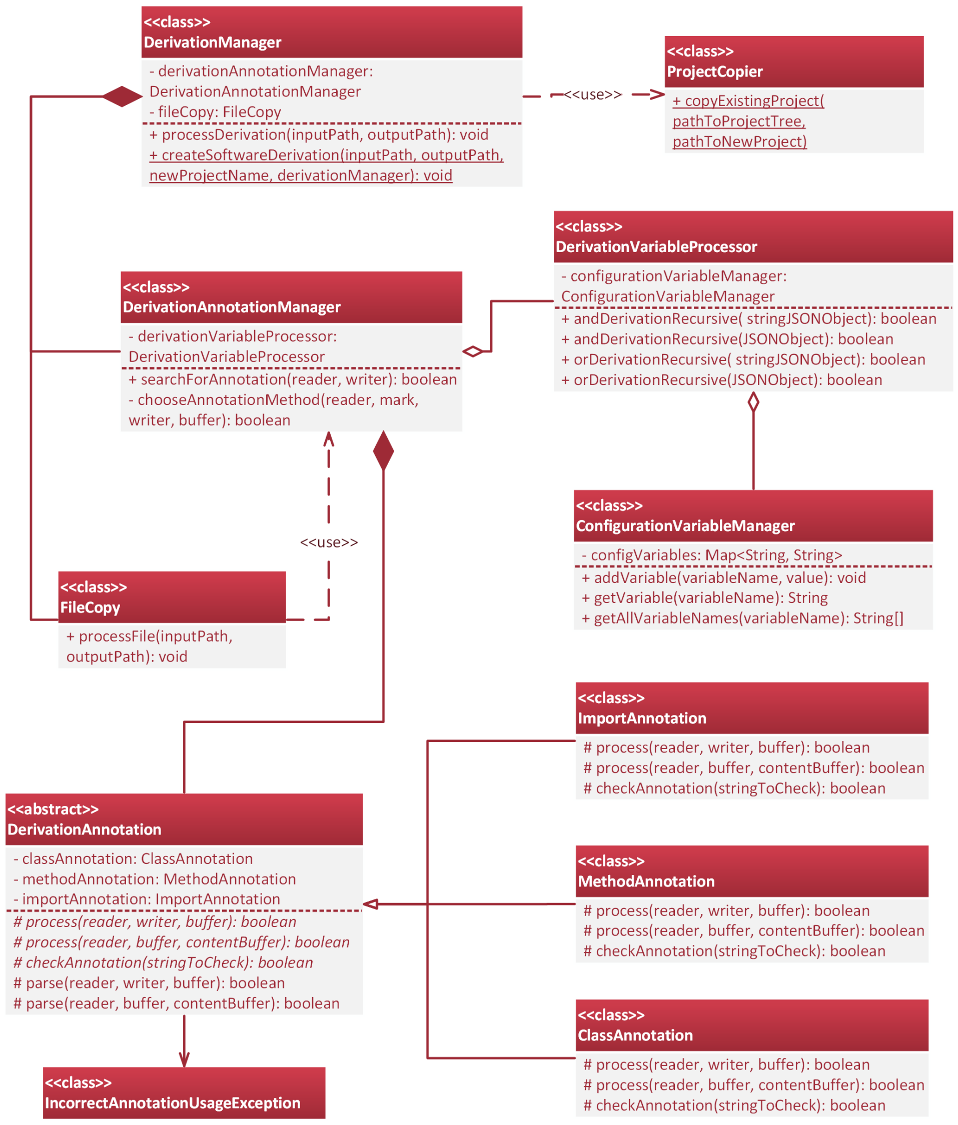
```
{
    "AND": {
        "OR": {
            "variable1": "false",
            "AND": {
                "variable2": "true",
                "variable3": "true"
            }
        },
        "variable4": "true"
    }
}
```

Obr. 6: Complex derivation rule

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.

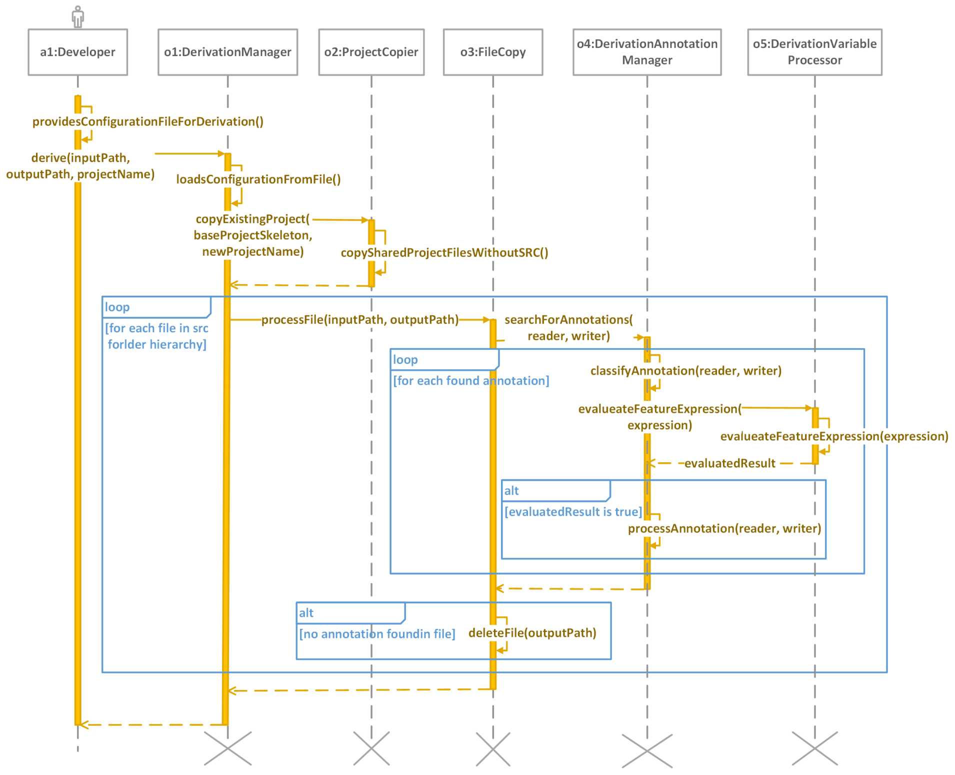
For making more complex condition we enable to use AND and OR operators to specify relation between variables as is described in picture 6. We use recursion to evaluate them. If no operators are used, expression is evaluated as AND statement. For AND operator all children should be true. For OR operator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

Class diagram for given solution is shown in picture 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 coped 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.


<<class>>
DerivationManager
- derivationAnnotationManager: DerivationAnnotationManager
- fileCopy: FileCopy
+ processDerivation(inputPath, outputPath): void
+ createSoftwareDerivation(inputPath, outputPath, newProjectName, derivationManager): void
<<use>>
<<class>>
ProjectCopier
+ copyExistingProject(pathToProjectTree, pathToNewProject)
<<class>>
DerivationVariableProcessor
- configurationVariableManager: ConfigurationVariableManager
+ andDerivationRecursive(stringJSONObject): boolean
+ andDerivationRecursive(JSONObject): boolean
+ orDerivationRecursive(stringJSONObject): boolean
+ orDerivationRecursive(JSONObject): boolean
<<class>>
DerivationAnnotationManager
- derivationVariableProcessor: DerivationVariableProcessor
+ searchForAnnotation(reader, writer): boolean
- chooseAnnotationMethod(reader, mark, writer, buffer): boolean
<<use>>
<<class>>
ConfigurationVariableManager
- configVariables: Map<String, String>
+ addVariable(variableName, value): void
+ getVariable(variableName): String
+ getAllVariableNames(variableName): String[]
<<class>>
FileCopy
+ processFile(inputPath, outputPath): void
<<class>>
ImportAnnotation
# process(reader, writer, buffer): boolean
# process(reader, buffer, contentBuffer): boolean
# checkAnnotation(stringToCheck): boolean
<<abstract>>
DerivationAnnotation
- classAnnotation: ClassAnnotation
- methodAnnotation: MethodAnnotation
- importAnnotation: ImportAnnotation
# process(reader, writer, buffer): boolean
# process(reader, buffer, contentBuffer): boolean
# checkAnnotation(stringToCheck): boolean
# parse(reader, writer, buffer): boolean
# parse(reader, buffer, contentBuffer): boolean
<<class>>
MethodAnnotation
# process(reader, writer, buffer): boolean
# process(reader, buffer, contentBuffer): boolean
# checkAnnotation(stringToCheck): boolean
<<class>>
ClassAnnotation
# process(reader, writer, buffer): boolean
# process(reader, buffer, contentBuffer): boolean
# checkAnnotation(stringToCheck): boolean
<<class>>
IncorrectAnnotationUsageException

Obr. 7: Class 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

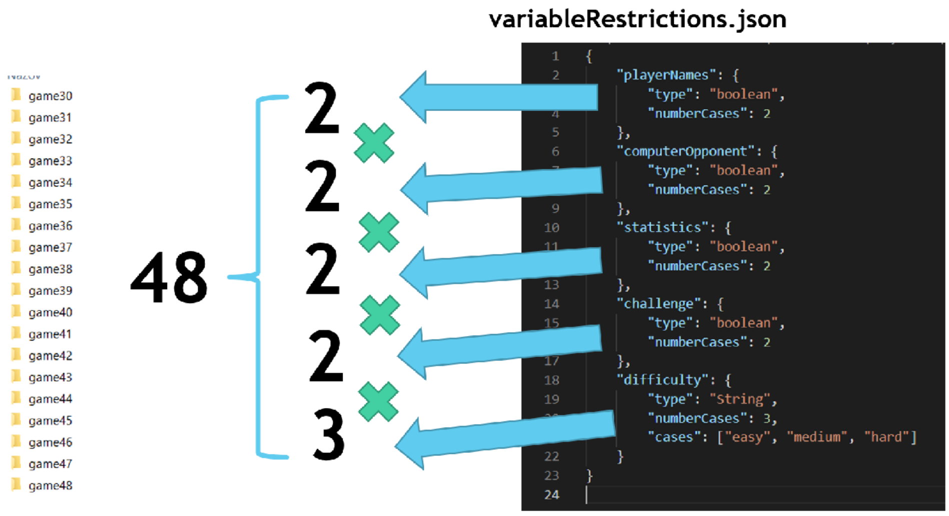

a1:Developer
o1:DerivationManager
o2:ProjectCopier
o3:FileCopy
o4:DerivationAnnotation Manager
o5:DerivationVariable Processor
providesConfigurationFileForDerivation()
derive(inputPath, outputPath, projectName)
loadsConfigurationFromFile()
copyExistingProject(baseProjectSkeleton, newProjectName)
copySharedProjectFilesWithoutSRC()
loop
[for each file in src forlder hierarchy]
processFile(inputPath, outputPath)
searchForAnnotations(reader, writer)
loop
[for each found annotation]
classifyAnnotation(reader, writer)
evalueateFeatureExpression(expression)
evalueateFeatureExpression(expression)
evaluatedResult
alt
[evaluatedResult is true]
processAnnotation(reader, writer)
alt
[no annotation foundin file]
deleteFile(outputPath)

Obr. 8: Sequence diagram for product derivation

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.

7 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 them. For example variable type and number of possible cases. Variable can have boolean type and all possible cases are two, 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



```
{
    "playerNames": {
        "type": "boolean",
        "numberCases": 2
    },
    "computerOpponent": {
        "type": "boolean",
        "numberCases": 2
    },
    "statistics": {
        "type": "boolean",
        "numberCases": 2
    },
    "challenge": {
        "type": "boolean",
        "numberCases": 2
    },
    "difficulty": {
        "type": "String",
        "numberCases": 3,
        "cases": ["easy", "medium", "hard"]
    }
}
```

Obr. 9: Derivation of all Battleship game types

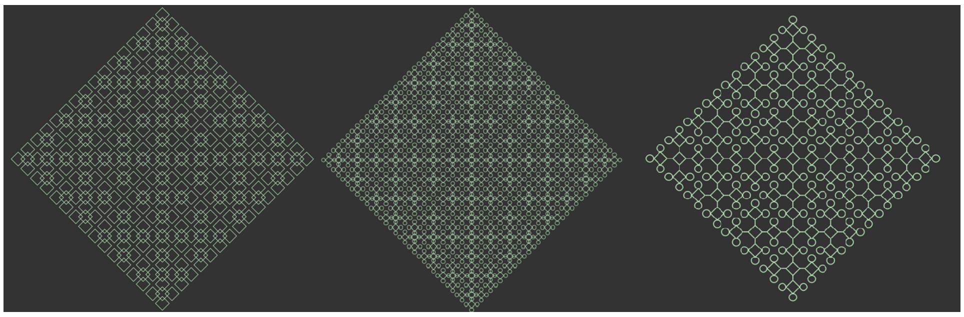
configuration file as variableRestrictions.json. Generator will generate all configuration for all cases of given variables. Described derivation method will be called for each configuration. In our project we created 2 * 2 * 2 * 2 * 3 = 48 derivations as depicts picture 9.

8 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 [5], 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.



Obr. 10: Different anklet fractal shapes

9 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.

9.1 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.

9.2 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 to separate them. 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.

10 Recent work

Kód kt

11 Conclusions and next work

Analyzo

Literatúra

[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] 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.

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

[5] R. Pelánek. *Programátorská cviebnice.* Computer press, 1. vydání edition, 2012.

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

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

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