PROPERTIES OF A FEATURE IN CODE-ASSETS: AN EXPLORATORY STUDY

Armaya'u Zango Umar[1] and Jaejoon Lee[2]

[1]Department of Mathematical Sciences, Al-Qalam University Katsina
[2]School of Computing Science, University of East Anglia

ABSTRACT

Software product line engineering is a paradigm for developing a family of software products from a repository of reusable assets rather than developing each individual product from scratch. In feature-oriented software product line engineering, the common and the variable characteristics of the products are expressed in terms of features. Using software product line engineering approach, software products are produced en masse by means of two engineering phases: (i) Domain Engineering and, (ii) Application Engineering. At the domain engineering phase, reusable assets are developed with variation points where variant features may be bound for each of the diverse products. At the application engineering phase, individual and customized products are developed from the reusable assets. Ideally, the reusable assets should be adaptable with less effort to support additional variations (features) that were not planned beforehand in order to increase the usage context of SPL as a result of expanding markets or when a new usage context of software product line emerges. This paper presents an exploration research to investigate the properties of features, in the code-asset implemented using Object-Oriented Programming Style. In the exploration, we observed that program elements of disparate features formed unions as well as intersections that may affect modifiability of the code-assets. The implication of this research to practice is that an unstable product line and with the tendency of emerging variations should aim for techniques that limit the number of intersections between program elements of different features. Similarly, the implication of the observation to research is that there should be subsequent investigations using multiple case studies in different software domains and programming styles to improve the understanding of the findings.

KEYWORDS

Software product line, Feature in the code-asset, Exploratory study, Adaptability of reusable code-assets

1. INTRODUCTION

Software product line engineering (SPLE) is a paradigm for developing a family of software products from the repository of reusable assets rather than developing each individual product from scratch. The driver of SPLE is pre-planned software reuse and within a specific problem area known as a domain. In the feature-oriented SPLE[1]–[3], the common and the variable characteristics of the products are expressed in terms of features. Thus, a *feature* is used as the key abstraction to distinguish between the members of the family. Consequently, the sets of products in the product line are said to have 'common' features and differ in 'variable' features.

1.1. Feature Model

A feature model [2] is a graphical tree structure in which product features of a product line are identified and organized with their types and their relationships. Feature type is one of the following broad categories:

- A mandatory type: a common feature that is manifested in all the products of a product line.
- A variable type: a feature of this type is either optional, is part of an alternative group (only one feature in the group can be selected), or is part of an inclusive OR group feature (more than one feature can be selected from the group). Unlike the alternative group, the OR group features are not mutually exclusive and can either contain all optional features or at least one of the features must be selected.

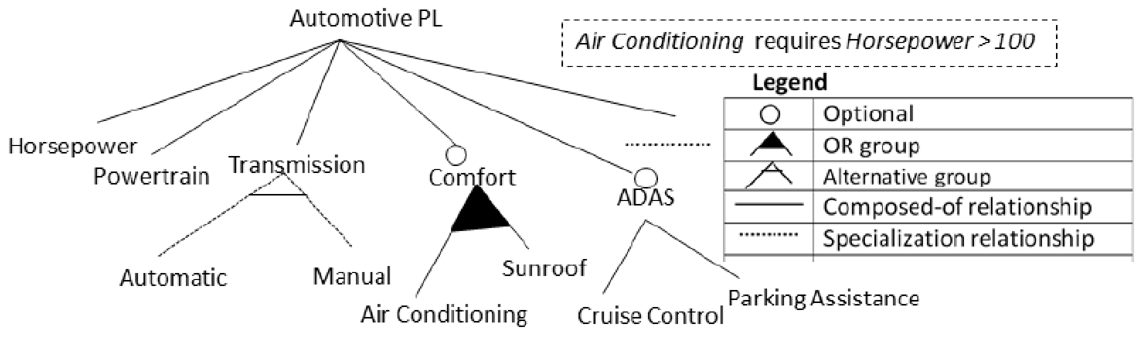


Figure 1: Feature Model of Automotive Product Line

Figure 1 depicts a hypothetical feature model of an automotive product line adapted from Kang *et al.* [3]. In the feature model, the *Powertrain* and *Transmission* features are mandatory because they must be available in every car. *Comfort* and *Advanced Driver Assistance (ADAS)* are optional features because they do not have to be in every vehicle produced. The *Automatic* and *Manual* are alternative features since the two cannot co-exist in the same car. The *Air Conditioning* and *Sunroof* are OR group features because a car can have zero or more of the features.

A selection of a valid combination of features is known as *product configuration*. For example, the following is a valid configuration of features from Figure 1: ***Horsepower 150, Powertrain, Automatic, Air conditioning***

Using software product line engineering approach, software products are produced *en masse* by means of two engineering phases:

(i) Domain Engineering (DE) and, (ii) Application Engineering (AE)

At the domain engineering phase, reusable assets are developed with variation points where variant features may be bound for each of the diverse products. At the application engineering phase, individual and customized products are developed from the reusable assets[4].

Figure 2 summarizes the relationship between feature space and the outputs of both domain engineering and application engineering activities. As shown in Figure 2, features in the feature space are mapped to the reusable assets (deliverable of the domain engineering activities). Similarly, each product or partial product, as the case may be, is derived from the reusable asset based on feature configurations from the feature space (i.e. valid selection of features). Lastly, a product-specific artefact may be fed back into the reusable assets in the form of feedback.

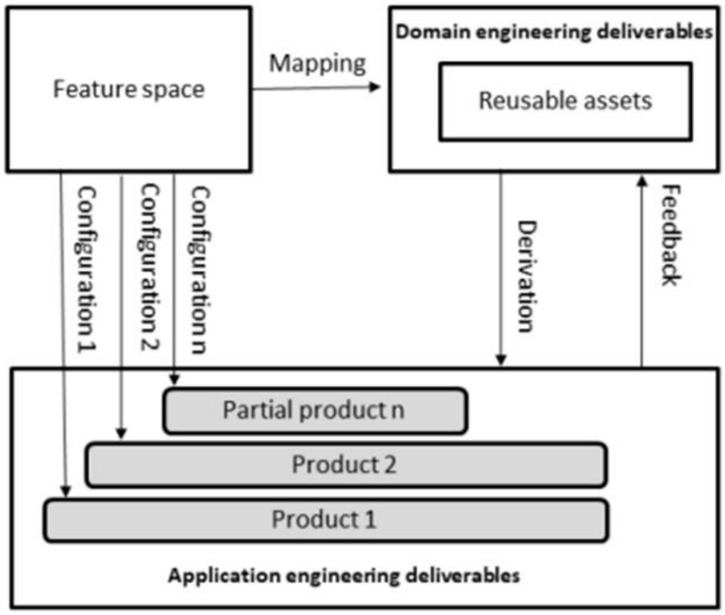


Figure 2: Relationship between feature space (top left) on one hand and the deliverables of both domain engineering (top right) and application engineering (bottom) on the other hand

1.2. Justification for the study

Modifiable assets are needed to support additional variations that were not planned beforehand to increase the usage context of the software product line (SPL) as a result of expanding markets. For example, Kastner [5] observed that, in Berkeley Database Engine (BDE) product line[6], features such as *Statistics* and *Transactions* were implemented as mandatory. However, such features will have to be made optional to make BDE configurable to other usage-contexts such as smartcard products because the product cannot afford the footprint of the extraneous feature. Therefore, to increase the usage context of BDE, its code-assets must be modified to make the *Statistics* and the *Transactions* optional. Failure to inject additional variations may lead to the delivery of product with extraneous code-assets - which is not desirable for lean memory applications and may also cause a problem especially if the product is to be integrated within other software products[7].

The objective of this paper is to explore the properties of features, in the code-asset, that may affect modifiability of injecting additional variations. Thus, the paper attempted to answer the following research question:

RQ. What are the characteristics of a feature in the code-asset that may potentially affect the modifiability of injecting additional variations?

The contribution of this paper is the exploration of properties of features at the implementation level and contributed with the description of the properties that may affect the flexibility of reusable assets.

2. STUDY DESIGN

This section provides explanations of the key design decisions for this study. The section begins with clarifications of the key terms that appear repeatedly in the paper.

Definition 1 (Code-asset, program element)

Code-asset (CA) is a pair <E; R> where E represents the set of program elements of CA, and R is the set of relationships between the program elements. A program element e can be an attribute, an operation, or a declaration. Let Att be the set of attributes of CA, Op be the set of operations of CA and Dec be the set of declarations of CA. The set E of program elements of CA is defined as $E = Att \cup Op \cup Dec$

Definition 2 (Feature module)

A feature module f1 consists of a set of program elements, Ef1, such that $Ef1 \subset E$.

2.1. Case Study: Oracle Berkeley Database Engine

Oracle Berkeley Database Engine (BDE), Java Edition [6], is an embedded storage engine designed to support integrating application logic and storage requirements of a software product in a single binary installation. It is specific for applications targeting Java Virtual Machine (JVM) and where no separate installation of a database server is required. Hence, the BDE runs in the same memory address space with the integrated application logic and thereby eliminating the overhead of process switching. Thus, a BDE can be embedded in a wide range of applications.

BDE has many features that are extraneous to the requirements of some applications in the domain. For example, features for concurrent access and atomic transaction are not required in an application with single access and simple data requirements. Similarly, by default, BDE gathers statistics of almost every operation of the database such as tree traversal and memory usage. The implementation of the statistics collection adds a significant footprint that may become a burden to some applications that do not require the collection of statistics. Therefore, those features should be optional and the code-asset should reflect their variability.

We selected the BDE as a case study because its legacy code-asset requires additional variations to derive customized products for the different applications and has been used in previous related researchers[5], [8].

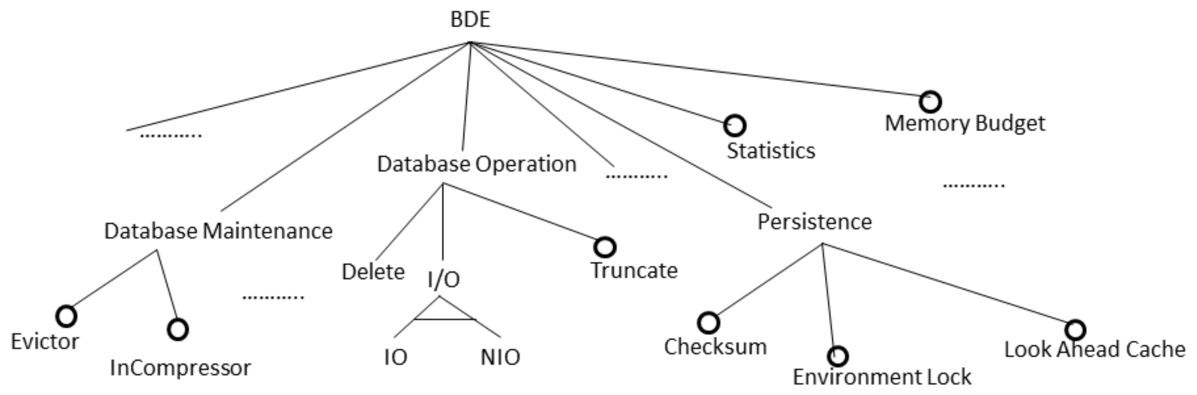


Figure 3: Selected BDE variations transformed into features.

Fig.3 depicts the partial feature model of BDE in which 11 features hitherto not optional but will have to be made so, to increase its usage-context. All the features were identified from previous studies[5], [8]and vary in size of lines of codes. We limited our selection to 11 features because the retrofitting of variations into features is tediously repetitive. Table 1 presents a brief explanation of the selected features.

Table 1: List of 11 features selected for the exploration

Feature name	**Explanation**
Checksum	Feature for calculating checksum for every database entry to be written to the log file and using the same for validation while reading back the entry
Delete	Feature implementing delete operation of the database.
Environment Lock	Feature responsible for locking the database to preserve its integrity on concurrent access.
Evictor	Feature responsible for maintaining memory consumption within a certain threshold specified in the database configuration.
Incompressor	Removes deleted entries and empty nodes from the tree
IO	Variant implementation of database input and output operation
Look ahead cache	Feature for maintaining cache on log files that are line-up for cleaning.
Memory Budget	Feature for monitoring overall memory usage of the database.
NIO	Variant implementation of database input and output operation.
Truncate	Feature for deleting the database and creating a new one.
Statistics	Feature responsible for gathering statistics about several operations of the database.

2.2. Case study exploration

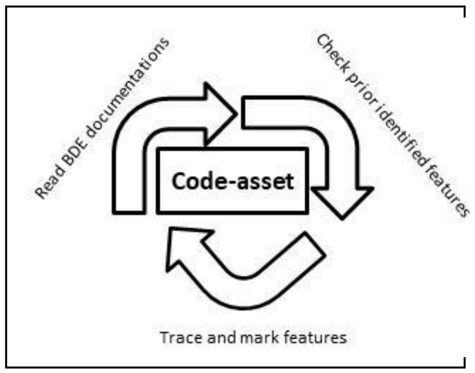
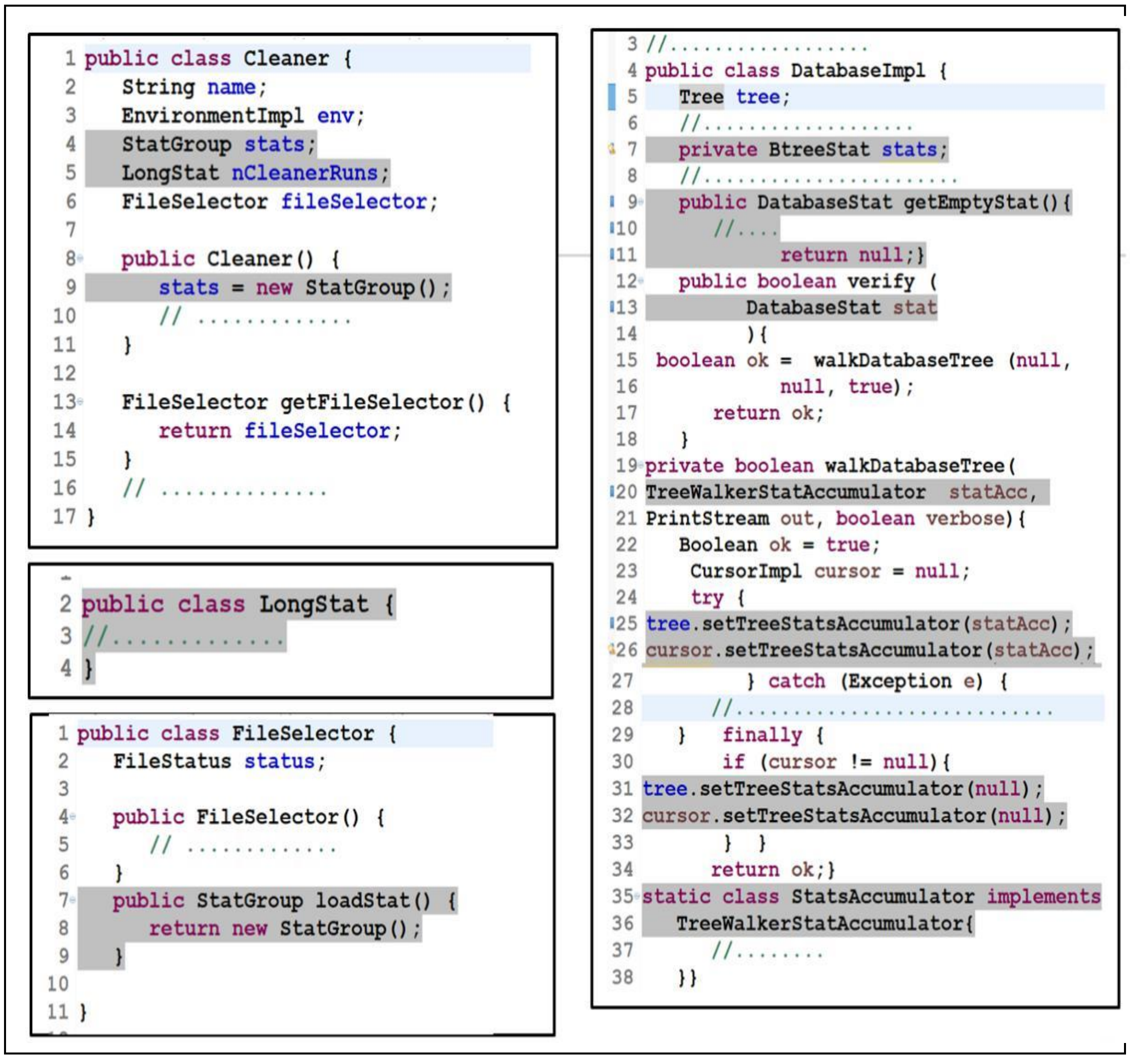


Figure 4: Iterative process for the exploration

As program elements of a specific feature are often scattered across the code-assets[9], to explore the properties of features in the code-asset, we followed an iterative process depicted in Fig.4. We started with consulting BDE documentation[6] to have reasonable domain knowledge. We checked the features already identified from the previous studies[5], [8]. We searched through the code-asset to trace the implementation of the identified features.

Statistics feature is one of the features that spread all over the code-assets. For example, Fig.5 illustrates the code-asset in which the program elements of the *Statistics* feature tangled with other program elements. In the figure, there are four Java classes: *Cleaner, LongStat, FileSelector,* and *DatabaseImpl.* Within the classes, the program elements of *Statistics* are shaded in grey.

In the *Cleaner* class at the top-left of Fig.5, there are two attributes, *stats* and *nCleanerRuns* (line 4-5) for the *Statistics* feature. In addition, there is another program element within the class constructor (line 9). The entire *LongStat* class, at the middle-left of Fig.5, is for the *Statistics* feature. In the *FileSelector* class, at the bottom-left of Fig.5, the method *loadStat()* (line 7-9) is also for the *Statistics* feature.



```
 1 public class Cleaner {
 2    String name;
 3    EnvironmentImpl env;
 4    StatGroup stats;
 5    LongStat nCleanerRuns;
 6    FileSelector fileSelector;
 7
 8    public Cleaner() {
 9       stats = new StatGroup();
10       // .............
11    }
12
13    FileSelector getFileSelector() {
14       return fileSelector;
15    }
16    // ..............
17 }
```

```
2 public class LongStat {
3 //.............
4 }
```

```
 1 public class FileSelector {
 2    FileStatus status;
 3
 4    public FileSelector() {
 5       // ............
 6    }
 7    public StatGroup loadStat() {
 8       return new StatGroup();
 9    }
10
11 }
```

```
 3 //..................
 4 public class DatabaseImpl {
 5    Tree tree;
 6    //..................
 7    private BtreeStat stats;
 8    //......................
 9    public DatabaseStat getEmptyStat(){
10       //....
11             return null;}
12    public boolean verify (
13          DatabaseStat stat
14          ){
15  boolean ok =  walkDatabaseTree (null,
16             null, true);
17       return ok;
18    }
19 private boolean walkDatabaseTree(
20 TreeWalkerStatAccumulator  statAcc,
21 PrintStream out, boolean verbose){
22    Boolean ok = true;
23     CursorImpl cursor = null;
24     try {
25 tree.setTreeStatsAccumulator(statAcc);
26 cursor.setTreeStatsAccumulator(statAcc);
27             } catch (Exception e) {
28       //...........................
29    }   finally {
30        if (cursor != null){
31 tree.setTreeStatsAccumulator(null);
32 cursor.setTreeStatsAccumulator(null);
33          }  }
34       return ok;}
35 static class StatsAccumulator implements
36    TreeWalkerStatAccumulator{
37       //.......
38    }}
```

Figure 5: Part of BDE code-assets; code-assets for *Statistics* (shaded in grey) feature are tangled with other program elements

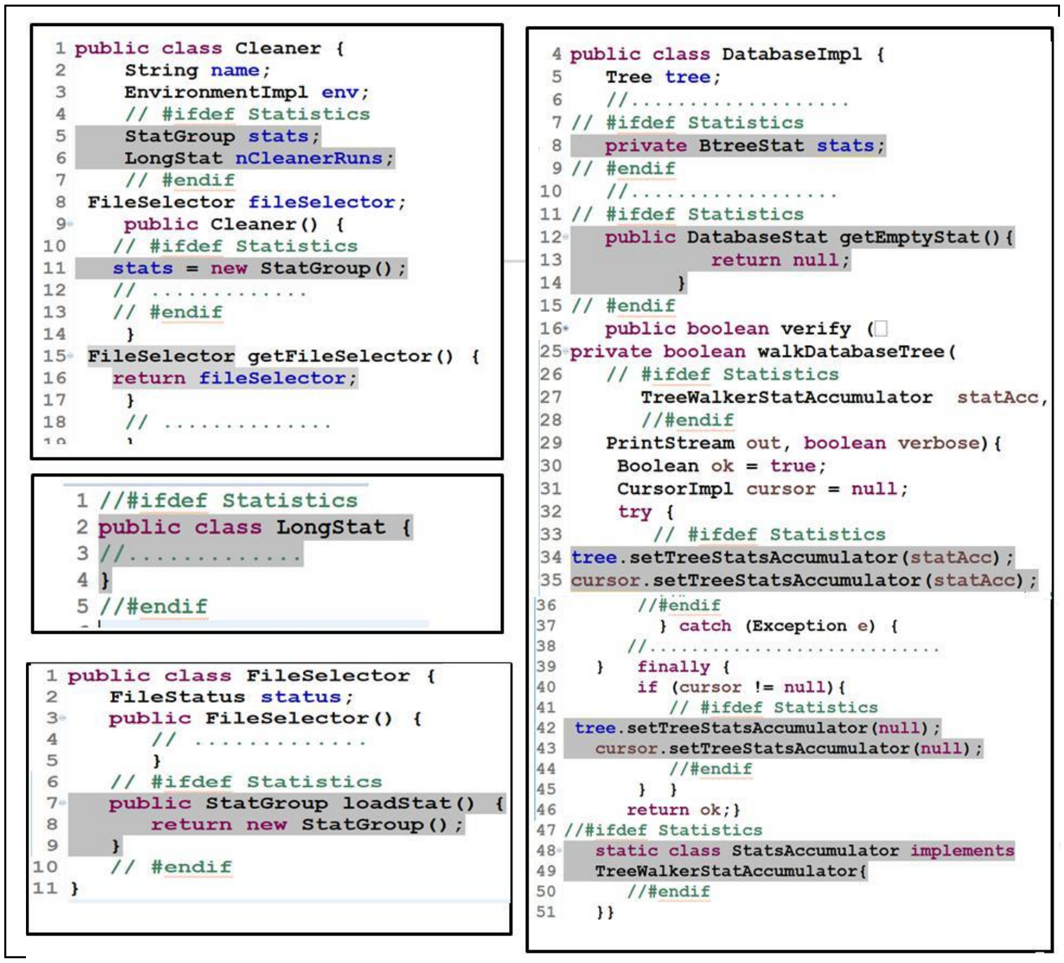
In the *DatabaseImpl* class at the right hand side of Fig.5, the attribute, *stats* (line 7), the method, *getEmptyStat()(*line 9-11), and the static inner class *StatsAccumulator*(line 35-36) are for the *Statistics* feature. In addition, program elements of The *Statistics* feature can be seen within the *walkDatabaseTree* method (line 25-26, line 31-32, and even within its parameters on line 20).

With the support of a tool, we marked program elements of the traced features with annotations. Annotation is the most popular means of enforcing variations in code-assets[10]. Fig.6, illustrates the use of the annotation to mark program elements of the *Statistics feature* in the four (classes) discussed earlier. In the figure, program elements of *Statistics* are annotated with *//#ifdef Statistics.... //#endif*. For the annotations, we used *Antenna*, a non-native annotation facility introduced in Java and integrated with a SPL implementation tool-suite[11].

3. Observation and Discussion

3.1. Observation

In the process of marking the implementation of the eleven (11) features, we observed that the code-asset is a union of feature modules and the program elements in the feature modules intersect with each other. Fig.7 depicts this observation in a Venn diagram.



```
public class Cleaner {
     String name;
     EnvironmentImpl env;
     // #ifdef Statistics
     StatGroup stats;
     LongStat nCleanerRuns;
     // #endif
 FileSelector fileSelector;
     public Cleaner() {
    // #ifdef Statistics
    stats = new StatGroup();
    // .............
    // #endif
     }
 FileSelector getFileSelector() {
    return fileSelector;
     }
     // ..............
```

```
//#ifdef Statistics
public class LongStat {
//.............
}
//#endif
```

```
public class FileSelector {
    FileStatus status;
    public FileSelector() {
        // .............
        }
    // #ifdef Statistics
    public StatGroup loadStat() {
        return new StatGroup();
    }
    // #endif
}
```

```
public class DatabaseImpl {
   Tree tree;
   //...................
// #ifdef Statistics
   private BtreeStat stats;
// #endif
   //..................
// #ifdef Statistics
   public DatabaseStat getEmptyStat(){
            return null;
         }
// #endif
   public boolean verify (□
private boolean walkDatabaseTree(
   // #ifdef Statistics
      TreeWalkerStatAccumulator  statAcc,
      //#endif
   PrintStream out, boolean verbose){
    Boolean ok = true;
    CursorImpl cursor = null;
    try {
       // #ifdef Statistics
tree.setTreeStatsAccumulator(statAcc);
cursor.setTreeStatsAccumulator(statAcc);
       //#endif
         } catch (Exception e) {
      //............................
   }   finally {
       if (cursor != null){
          // #ifdef Statistics
 tree.setTreeStatsAccumulator(null);
   cursor.setTreeStatsAccumulator(null);
          //#endif
       }  }
      return ok;}
//#ifdef Statistics
   static class StatsAccumulator implements
   TreeWalkerStatAccumulator{
      //#endif
   }}
```

Figure 6: Annotated code of *Statistics* feature

.

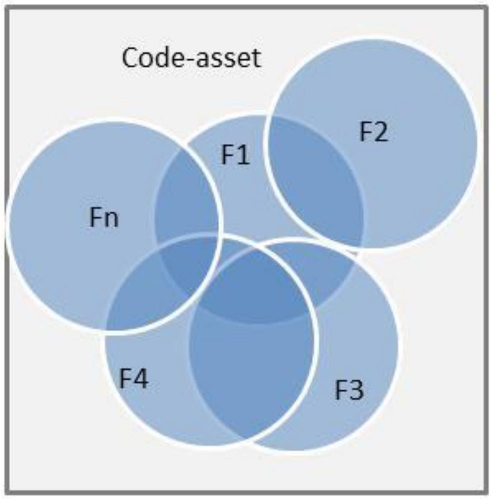


Figure 7: Union and intersection of feature modules in code-asset

In Fig.7, each of the feature modules, represented by F1...Fn internally contains program elements (attributes, declarations, and operations) that are only for the containing feature as well as program elements that are shared with other features. We refer to the program elements that are not shared with any other feature as exclusive elements to the given feature and the shared part as an intersection between features. Thus, we defined the exclusivity and intersection (semi-formally) in definition 3 and definition 4 respectively:

Definition 3 (Exclusivity)

Program element ef1 is said to be exclusive to a feature f1 if it satisfies a disjoint union relationship with other features in the code-asset CA. i.e.ef1 ∩ ef2 ∩ ef3 ∩ ef3∩ ….. efn =ø

Definition 4 (Intersection)

An intersection between programs elements of different features ef1 and ef2 is defined as ef1#ef2, which is a modification and or integration of ef1 and ef2 so that they work correctly together[12].[1]

Table 2: Intersecting features

Feature	**Intersections**
Delete	*Transaction, Cleaner*
Environmental Lock	*IO, NIO*
Evictor	*Incompressor*
Memory Budget	*Transaction, Evictor*
Statistics	*Transaction, Evictor, Incompressor, Cleaner, Log, Cache, Checkpointer*

[1] Batory and Kim [12] refer to this as feature interaction in the original definition. We use the term intersection to distinguish it from the interaction as a problem of unexpected side effects when two or more features are combined to work together [30]

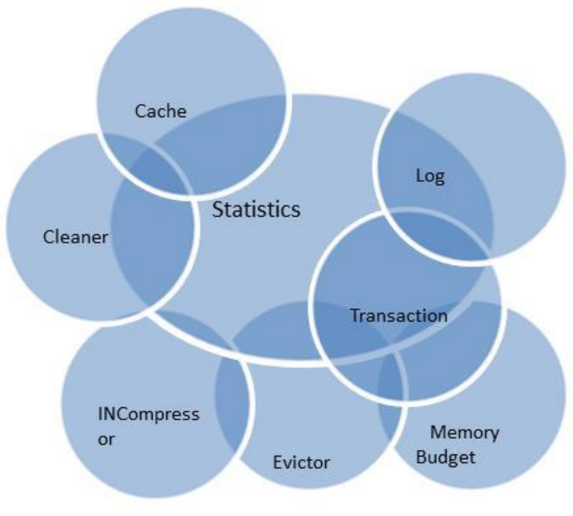


Figure 8: Illustration of intersecting features

For the 11 features explored in the study, Table 2 shows the summary of intersecting features. Note that, some of the intersecting features in Table 2 are not among the 11 selected features but were picked while annotating the 11 features. Also, note that the intersection is reflexive, i.e. if feature A intersects feature B, implies feature B also intersects feature A. Fig.8 depicts an illustration of 7 features intersecting the *Statistics* feature.

3.2. Discussion

The consequence of observation is that injection of additional variations requires decoupling of features in the code-asset: The program elements of the feature to be made optional have to be traced in the code-asset; Both the exclusive and the intersection program elements have to be separated; The separated program elements should be added to a product configuration only if the variant feature is selected.

We noticed this observation to be true within the context of Object-Oriented Development. For example, using various design patterns in OOP[13], a designer should be able to encapsulate features to some extent. For example, Keypad, Fingerprint, Remote control as OR group sub-features of Access Control feature in a smart home can be implemented using Strategy pattern. In that case, a programmer implements each feature as a separate strategy for accessing the smart home, and each of the strategies only contains program elements exclusive to one of the features. At the point of invocation, however, a request to access the smart home has to be resolved to one of the concrete strategies, and that is the point of intersection.

Similarly, when using class inheritance, as a form of polymorphism, to implement an optional feature as a subclass of one of the mandatory classes, the sub-class is the exclusive program element to the optional feature. In that case, intersection points are places where references are made to the subclass.

Thus, thinking in terms of exclusiveness and intersection can be beneficial. Intuitively, limiting the number of intersections may be better for a product line that is not stable and with the tendency of the emergence of new usage-contexts. Of particular note, the *Factory* pattern encapsulates a specific form of intersection- a point to create one of the variant objects.

4. RELATED WORK

In this section, we present similar studies from the literature to highlight the novelty of our approach. There is an approach of taking existing software assets and transforming them into reusable assets of a software product line. It is called *extractive* and there have been positive arguments and studies for it [14]–[16]. For example,[16] proposed a semi-automated and step-wise process to identify, map, and visualize features in the code-assets of a legacy system that is being transformed into software product line following the extractive approach. However, the focus was mostly on the benefits of the approach in easing the transition from single software product development to the software product line.

There are studies aimed at visualizing features in the code-assets[16], [17]. For example, [18] developed a tool for interactive visualization of features. It provided very easy support for visualising and locating features in the code-assets. The tool can also be used to answer questions such as which feature has the highest line of codes or which feature has the highest spread in the code-assets. However, the aim was to aid feature comprehension and not properties of features that may affect modifiability. Our description of properties of feature in the code-assets as well as the visualization is meant to give an insight of modifiability of the product line code-assets to introduce additional variations.

Tracing features in the code-assets is one of the common activities in software product line engineering[19] and tools[20]–[23] were developed to facilitate the activity. Nonetheless, less attention was paid to the actual properties of features that have the potential to affect the modifiability of the code-asset. Rather, studies on the exploration of features in the code-assets focused on getting insights that will improve feature traceability[24], the influence of varying project scope on the feature traceability, as well as effect of feature traceability on program comprehension[25].

May *et al* [26] conducted an exploration of features in the code-assets of micro-service-based implementation of a webshop. They even took a step further and re-engineered the code-assets and produced product line architecture; introduced variations using the principles of delta-oriented software product line[27]. In another exploratory study, Murphy *et al* [28] investigated the flexibility of three language-based techniques when used to untangle features from a code-asset. All the three techniques, Hyper/J, AspectJ[29], and the authors' own technique, were designed for advanced separation of concern. The authors qualitatively characterized the effect the different techniques had on the structure of the code-asset and also characterized how to restructure the code-asset to untangle features with each of the techniques. Nevertheless, these studies did not describe the properties of features, in the code-assets, that may affect the modifiability of code-assets.

In summary, this study attempted to provide insights into properties of features that may affect modifiability of code-assets to pave way for researches on comparable software product line implementation techniques that can be used to decouple features in the code-assets to introduce new variations.

5. FUTURE WORK AND CONCLUSION

In future researches, properties of feature in other domains and programming styles shall be explored and then compare newer software product line implementation techniques with respect to modifiability of existing code-assets to introduce variations that were not planned beforehand.

In this paper, we explored properties features in the code-asset using Berkeley Database Engine (Java edition), as a case study. In the exploration, we observed that program elements of disparate features formed unions as well as intersections that may affect the modifiability of the code-assets. The implication of this research to practice is that an unstable product line and with the tendency of emerging variations, should aim for programming style that limits the number of intersections between program elements of different features. Similarly, the implication of the observation to research is that, there should be subsequent investigations using multiple case studies in different software domains and programming styles in order to improve the understanding of the findings.

REFERENCES

[1] K. C. Kang, J. Lee, and P. Donohoe, “Feature-oriented product line engineering,” IEEE Softw., vol. 19, no. 4, pp. 58–65, Jul. 2002.

[2] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, “FORM: A feature-oriented reuse method with domain-specific reference architectures,” Ann. Softw. Eng., vol. 5, no. 1, pp. 143–168, 1998.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, “Feature-oriented domain analysis (FODA) feasibility study,” 1990.

[4] K. Kang, S. Kim, J. Lee, K. Kim, and E. Shin, “FORM: A feature-; oriented reuse method with domain-; specific reference architectures,” Ann. Softw., 1998, Accessed: Oct. 03, 2016. [Online]. Available: http://link.springer.com/article/10.1023/A:1018980625587.

[5] C. Kastner, S. Apel, and D. Batory, “A case study implementing features using AspectJ,” in 11th International Software Product Line Conference (SPLC 2007), 2007, pp. 223–232.

[6] Oracle Corporation, “Berkeley DB Java Edition Architecture,” 2006. http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/learnmore/bdb-je-architecture-whitepaper-366830.pdf (accessed Aug. 13, 2021).

[7] D. Garlan, R. Allen, and J. Ockerbloom, “Architectural mismatch: Why reuse is still so hard,” IEEE Softw., vol. 26, no. 4, pp. 66–69, 2009, doi: 10.1109/MS.2009.86.

[8] R. Andrade, M. Ribeiro, H. Rebêlo, P. Borba, V. Gasiunas, and L. Satabin, “Assessing idioms for a flexible feature binding time,” Comput. J., vol. 59, no. 1, pp. 1–32, 2015.

[9] L. Passos et al., “A Study of Feature Scattering in the Linux Kernel,” IEEE Trans. Softw. Eng., vol. 47, no. 1, pp. 146–164, Jan. 2021, doi: 10.1109/TSE.2018.2884911.

[10] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, An analysis of the variability in forty preprocessor-based software product lines, vol. 1. New York, New York, USA: ACM Press, 2010, p. 105.

[11] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, “FeatureIDE: An extensible framework for feature-oriented software development,” Sci. Comput. Program., vol. 79, pp. 70–85, 2014, doi: 10.1016/j.scico.2012.06.002.

[12] D. Batory, P. Höfner, J. Kim, D. Batory, P. Höfner, and J. Kim, "Feature interactions, products, and composition," in Proceedings of the 10th ACM international conference on Generative programming and component engineering - GPCE '11, 2011, vol. 47, no. 3, p. 13, doi: 10.1145/2047862.2047867.
[13] E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software. - Google Scholar. Pearson Education, 1995.
[14] P. Clements and C. Krueger, "Being Proactive Pays Off/Eliminating the Adoption Barrier. Point-Counterpoint article in," IEEE Softw., 2002.
[15] J. Krüger, S. Krieter, G. Saake, and T. Leich, "Extracting product lines from variants (explant)," in In Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems, 2020, pp. 1–2, doi: 10.1145/3377024.3377046.
[16] J. Krüger, L. Nell, W. Fenske, G. Saake, and T. Leich, "Finding lost features in cloned systems," ACM Int. Conf. Proceeding Ser., vol. 2, pp. 65–72, Sep. 2017, doi: 10.1145/3109729.3109736.
[17] S. Entekhabi, A. Solback, … J. S.-P. of the 23rd, and U. 2019, "Visualization of feature locations with the tool featuredashboard," in 23rd International Systems and Software Product Line Conference, Sep. 2019, vol. B, pp. 1–4, doi: 10.1145/3307630.3342392.
[18] A. Bergel, T. Berger, and M. R. V Chaudron, "FeatureVista : Interactive Feature Visualization," in 25th ACM International Systems and Software Product Line Conference, 2021, no. 21, doi: 10.1145/3461001.3471154.
[19] J. Krüger, T. Berger, T. L.-S. E. for Variability, and U. 2019, "Features and how to find them: a survey of manual feature location," Softw. Eng. Var. Intensive Syst., pp. 153–172, 2019.
[20] H. Jansson and J. Martinson, "HAnS: IDE-based editing support for embedded feature annotations," 2021.
[21] J. Pleumann, O. Yadan, and E. Wetterberg, "Antenna," 2010. http://antenna.sourceforge.net/wtkpreprocess.php.
[22] H. Abukwaik, A. Burger, B. K. Andam, and T. Berger, "Semi-automated feature traceability with embedded annotations," Proc. - 2018 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2018, pp. 529–533, Nov. 2018, doi: 10.1109/ICSME.2018.00049.
[23] T. Schwarz, W. Mahmood, and T. Berger, "A Common Notation and Tool Support for Embedded Feature Annotations," ACM Int. Conf. Proceeding Ser., vol. Part F164402-B, pp. 5–8, Oct. 2020, doi: 10.1145/3382026.3431253.
[24] A. SOLBACK, "Visualization of feature-traceability in variant-rich systems," 2019.
[25] J. Krüger, G. Alkl, T. Berger, T. Leich, and G. Saake, "Effects of explicit feature traceability on program comprehension," ESEC/FSE 2019 - Proc. 2019 27th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., pp. 338–349, Aug. 2019, doi: 10.1145/3338906.3338968.
[26] M. R. A. Setyautami, H. S. Fadhlillah, D. Adianto, I. Affan, and A. Azurat, "Variability management: re-engineering microservices with delta-oriented software product lines," ACM Int. Conf. Proceeding Ser., vol. Part F164267-A, p. 301, Oct. 2020, doi: 10.1145/3382025.3414981.
[27] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, "Delta-Oriented Programming of Software Product Lines," Springer Berlin Heidelberg, 2010, pp. 77–91.
[28] G. Murphy, A. Lai, and R. Walker, "Separating features in source code: An exploratory study," in In Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001, 2001, pp. 275–284.
[29] P. A. R. C. Xerox Corporation, "The AspectJtm Development Environment Guide," 2005. http://www.eclipse.org/aspectj/doc/released/devguide/index.html (accessed Aug. 13, 2021).
[30] S. Apel, J. Atlee, L. Baresi, P. Z.-D. Reports, and U. 2014, "Feature interactions: the next generation (dagstuhl seminar 14281)," 2014. doi: 10.4230/DagRep.4.7.1.