Modularization Compass
Navigating the White Waters of Feature-Oriented Modularity

Andrzej Olszak and Bo Nørregaard Jørgensen
The Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Odense, Denmark

Keywords: Software Evolution, Measurement, Modularity, Features.

Abstract: Successful software systems have to adapt to the evolving needs of their users. However, adding and extending functional features often becomes more difficult over time due to aggregated complexity and eroded modularity. This erosion can be quantified by measuring scattering and tangling of feature implementations in the source code, to track long-term regressions and to plan refactorings. This paper argues that the traditional usage of only the absolute values of modularity metrics is, however, insufficient and proposes to use their relative values instead. These relative values are referred to as the drift of feature-oriented modularity, and are defined as the distance between the actual metric values for a given source code and their values achievable for the source code's ideally modularized counterpart. The proposed approach, called modularization compass, computes the modularity drift by optimizing the feature-oriented modularization of source code based on traceability links between features and source code. The optimized modularizations are created automatically by transforming the groupings of classes into packages, which is guided by a multi-objective grouping genetic algorithm. The proposed approach was evaluated by application to long-term release histories of three open-source Java applications.

1 INTRODUCTION

Incorporating changes requested by the users during software evolution is non-trivial, because it requires a deep understanding of the relations between the software's problem domain and its solution domain (Turner et al., 1999). Doing so is difficult because the problem domain is centered around user-observable units of functionality, the so-called *features* (Turner et al., 1999)(Tarr et al., 1999), whereas the solution domain is arranged around source-code units such as modules, packages, classes, methods and instructions. Hence, during modification of a feature in response to a particular change requested by users, it is important to be able to efficiently map the feature to the concrete source-code units that need to be inspected, modified and tested. Furthermore, in order to aid software inspection and modification, one needs to properly modularize the implementations of features into source-code modules (Parnas, 1972).

Unfortunately, it is common that implementations of features are not explicitly represented in the organizations of software into source-code modules. Instead, the organization of software traditionally focuses on separating technical concerns such as model, view, controller or persistence into separate architectural layers each represented by one or more source-code modules. As a result, the implementations of features become *scattered* over multiple source-code modules and *tangled* with one another, as each feature typically crosscut multiple architectural layers. These relations between feature specifications and source-code modules affect software evolution in several ways:

- *Scattering* denotes the delocalization of the implementation of a feature over several source-code units of an application (Turner et al., 1999) and corresponds to the software comprehension phenomenon of *delocalized plans* (Letovsky and Soloway, 1986). The presence of delocalized plans is known to make it difficult to identify the relevant source-code units during change tasks (Letovsky and Soloway, 1986)(Eaddy et al., 2008).
- *Tangling* of features denotes the *interleaving* of the implementation of multiple features within a single module of source code (Rugaber et al., 1995). Such interleaving is known to make it difficult to understand how multiple features

Olszak A. and Nørregaard Jørgensen B..
Modularization Compass - Navigating the White Waters of Feature-Oriented Modularity.
DOI: 10.5220/0005092800480059
In *Proceedings of the 9th International Conference on Software Engineering and Applications* (ICSOFT-EA-2014), pages 48-59
ISBN: 978-989-758-036-9

relate and how they reuse fragments of each others' implementations (Benestad et al., 2009).

Apart from software comprehension, the representation gap between features and source-code modules makes it more difficult to modify source code. Due to scattering, modification of one feature may require understanding and modification of several seemingly unrelated source-code modules. Due to tangling, a modification intended to affect only one feature may cause accidental change propagation to other features that happen to use the source-code module being modified.

Due to the evolutionary implications of scattering and tangling, it is important to keep track of the development of their values over subsequent evolutionary releases of software. The erosion of feature-oriented modularity, as indicated by increasing scattering and tangling, has to be observed to provide feedback on the extent of the development overhead that they may incur. Ultimately, such knowledge can be used to inform planning of feature-oriented remodularization efforts.

This paper proposes an approach called *modularization compass* that quantifies the so-called drift of feature-oriented modularity in software. We define the *drift in feature-oriented modularity* as the distance between the scattering and tangling metric values for the actual source code of a software release and counterpart that is ideally modularized with respect to the metrics of interest. The idealized counterparts are created through a remodularization process that optimizes the grouping of classes into packages according to feature-oriented criteria using a multi-objective grouping genetic algorithm. To compute the values of scattering and tangling, the approach assumes availability of traceability links between features and source-code units, as obtainable from several existing feature-location approaches. Based on the measurements of scattering and tangling drifts, the modularization compass approach provides so-called *compass views* that depict the evolution of drift of feature-oriented modularity over an application's lifetime.

This approach was implemented for the Java programming language and evaluated using long-term release histories of three open-source Java applications. There, the drift information from the modularization compass views was used to identify the development periods in which the potential benefits from restructuring the code would have been largest, and to determine whether this restructuring effort should have focused on reducing the scattering or the tangling of features. Apart from demonstrating the approach, a number of observations were made regarding the nature of drift of feature-oriented modularity.

The remaining part of the paper is structured as follows. Section 2 describes the state of the art of feature-oriented modularity. Section 3 presents the modularization compass approach. Section 4 evaluates the approach. Finally, Section 5 concludes the paper.

2 STATE OF THE ART

There exist several works that investigate evolution of features and the modularity of their implementations over time.

Hsi and Potts (Hsi and Potts, 2000) proposed to use three views: morphological view, functional view and object view to study the co-evolution of the representation of features in the UI, their textual specifications and their implementation in three releases of Microsoft's Word text processor. The presented qualitative analysis shows that the features providing the core functionality experience little change and tend to stabilize over time. This is because they tend to become more entangled with associations as new features are added. As a result, newer features are observed to be added on the periphery of the main functionality of the application in either small extensions or larger clumps.

Fischer and Gall (Fischer and Gall, 2004) designed a visualization of feature co-evolution based on the logical coupling between source files created during adoption of change requests. This approach is used to uncover hidden dependencies among features and thereby to identify potential occurrences of architectural deterioration in directory structures of programs. The authors apply their approach to a four-year revision history of the Mozilla web browser to uncover unanticipated dependencies and co-evolution of features.

Hou and Wang (Hou and Wang, 2009) analyzed the evolution of features related to usability in the Eclipse IDE. This was done by both qualitative and quantitative manual analyses of the change logs of the project. The authors identify the majority of the changes as gradual refinements or incremental additions well accommodated by the project's architecture. The authors observe the usability-related features to be the largest component of work in the project, with a shift over time towards focusing on features concerned with integration of other features and their automation. The observed incremental, rather than punctuated, growth of features of Eclipse is believed to be enabled by the stability of the architecture.

Greevy et al. (Greevy et al., 2005) focused on qualitative assessment and visualization of evolutionary changes in implementations of features. Using the proposed visualization, the authors are able to reason about functional specialization of classes over time, extension of existing features with new classes and refactorings performed to features. The presented results depict an increase of feature count and addition of feature-specific classes over time.

In an earlier work, Olszak and Jørgensen (Olszak and Jørgensen, 2012) developed an approach to bi-directional remodularization of existing Java applications to improve the modularization of features in source code. There, feature location was performed using an annotation-driven dynamic analysis mechanism, and new feature-oriented package structures were created automatically using a multi-objective genetic algorithm aiming at reducing scattering, tangling, coupling and increasing cohesion. The observed improvements suggested that the modularizations produced by this approach to be good starting points when migrating applications to feature-oriented designs.

3 THE APPROACH

Implementing a feature inherently requires a mixture of technically diverse classes. In particular, each non-trivial feature encompasses some forms of (1) interfacing classes, which allow users to activate the feature and see the results of its execution, (2) logic and domain model classes, which contain the essential processing algorithms, and (3) persistence classes, which allow for storing and loading the results of the feature's execution. Hence, features can be viewed as implicit vertical slices that crosscut the common horizontal layers of an application's architecture. These implicit slices consist of graphs of collaborating classes that end up *scattered* and *tangled* within individual layers (Van Den Berg et al., 2006). This is depicted in Figure 1.

Our approach quantifies these two facets of modularity of features using the following measures, based on formulations proposed by Brcina and Riebisch (Brcina and Riebisch, 2008):

- *Scattering of a feature is quantified* as the number of packages that contribute to implementing that feature. The average of these values computed for all features in a system is referred to as *FSCA*. The formulation of FSCA is described in detail in Section 4.3.
- *Tangling of a package is quantified* as the number of features that the package contributes to. The average of these values computed for all packages in a system is referred to as *FTANG*. The formulation of FTANG is described in detail in Section 4.3.

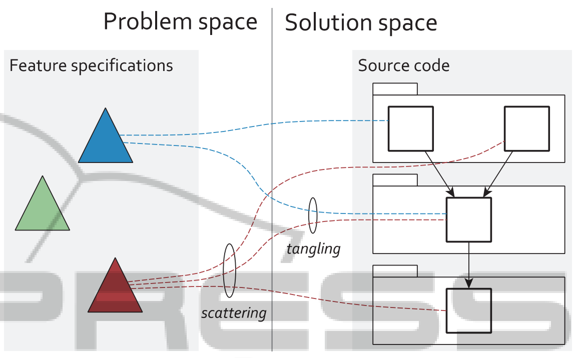


Figure 1: Relations between feature specifications and units of source code.

The extent to which scattering and tangling of features is minimized is a measure of how well features are modularized within source-code units. We refer to this as the degree of *feature-oriented modularity* of software.

In order to measure feature-oriented complexity of evolving features in terms of scattering and tangling, relations between the source-code units and individual features have to be identified. The process of identifying the relations between source-code units and observable functionality of a system is known as *feature location* (Wilde et al., 1992). This work assumes that traceability links are readily available or are recovered for an application using one of the feature-location approaches available in the literature. In particular, for the evaluation purposes, Section 4 uses an existing feature-location approach based on source-code annotation and dynamic analysis.

3.1 Evolution of Feature-Oriented Modularity

The essence of how features of software applications evolve is well expressed by the laws of *continuing growth* and the law of *increasing complexity* formulated by Lehman (Lehman, 1980). According to the first, software applications need to expand and enhance their features over time in order to remain useful to their users. The second postulates that these expansions will lead to increasing complexity

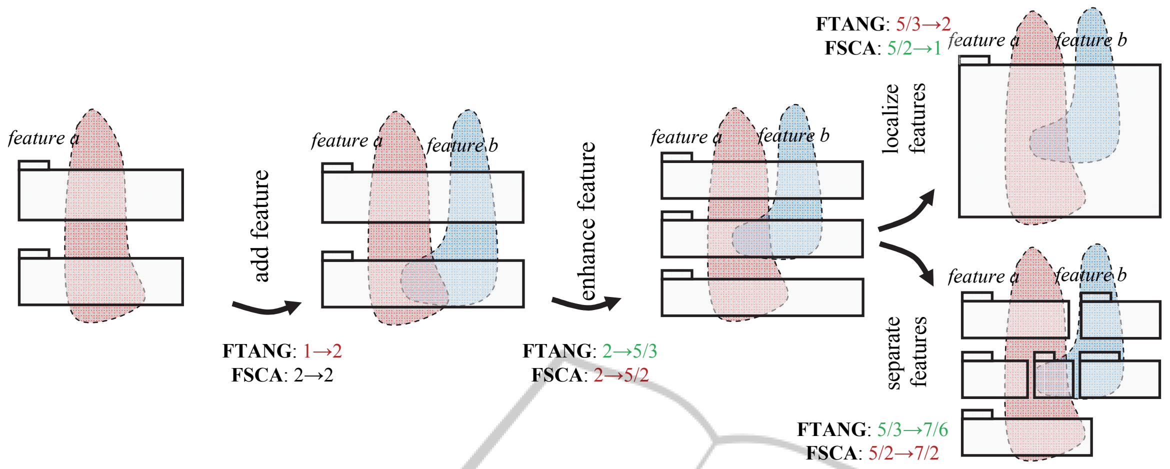


Figure 2: Example impact of evolutionary changes on feature-oriented modularity.

of the source code, unless work is done to reduce it. One of the facets of the increasing complexity is the increasing complexity of how features are modularized in source code, as will be exemplified in the following.

The example application schematically depicted in Figure 2 initially provides only one feature that is implemented by two layered modules. Hence, the initial average tangling FTANG in the application equals 1 (initially each module implements one feature), and the initial average scattering FSCA equals 2 (the feature is implemented by two modules).

The first change scenario depicts the effects of adding a new feature to the application without modifying the structure of the source code. Such a functional extension will naturally tend to increase the tangling of the application's modules, as a result of reusing parts of existing code among features.

The second scenario shows the effects of enhancing one of the existing features. Because the enhancement is implemented as a new module in the application (a realistic example of doing so would be adding persistence capabilities), the scattering of the feature increases.

Thereafter, depicted are two possible contrasting scenarios of source-code restructurings undertaken to improve modularization of features. One of them is based on the merging of existing modules to minimize the scattering of features. As can be seen, this causes features to be more tangled with one another. The other restructuring reduces feature tangling by dividing existing modules along the boundaries of features. As a side effect, the scattering of features increases.

Based on this simple example, two important observations can be made:

Addition and enhancement of user functionality will tend to increase the tangling and scattering of features. Accordingly, the difficulties of code comprehension and change propagation associated with these phenomena should be expected to increase as well.

Restructuring the source code to minimize only one of the two properties of feature-oriented modularity (i.e. scattering or tangling) will tend to degrade the other property. Hence, in order to achieve a simultaneous optimization of both these conflicting criteria, a middle-ground restructuring needs to be devised. As for the presented toy example, this could be done by simply enumerating all possible modularizations, but it would certainly not be feasible for larger systems, since the number of all possible distributions of N classes among M modules is equal to M^N.

3.2 The Drift of Modularity

There are multiple factors that have to be considered when planning a feature-oriented restructuring of an application. Fundamentally, undertaking a restructuring is only worthwhile if the costs of doing so are regained by lower development costs for subsequent releases. The costs of a restructuring include factors such as the actual effort required, the impact on time-to-market of the product, changes to design documentation, etc. On the benefits side, one should expect improvements of changeability and understandability of feature implementations during subsequent releases and hence a reduction of development costs. Unfortunately, estimating these benefits remains difficult without knowing how much the modularization of features can actually be improved by means of restructuring.

Hence, to make informed feature-oriented restructuring decisions, one should be able to foresee the consequences of performing a feature-oriented restructuring. In practice, this boils down to being able to foresee how much the current values of feature scattering and tangling can be reduced in course of restructurings.

Unfortunately, the achievable benefits of restructurings cannot be estimated by simply computing the distance between the current values of scattering and tangling metrics and their numerical minima. This is because the numerical minima of these and other metrics often do not correspond to realistic optimal modularizations of non-trivial applications, e.g. tangling equal to 1 requires no code sharing among features; scattering equal to 1 requires each feature to be fully contained in a single module; coupling equal to 0 requires no dependencies among modules, etc. The situation is further complicated by the presence and the type of normalization factors embedded in each metric.

In order to identify the maximum possible improvements of feature-oriented modularization, it is therefore necessary to actually construct its optimized modularization, on which the reference scattering and tangling values can be measured. Assuming that doing so is possible with sufficient accuracy and in an automated manner (which assumption will be expanded on in the next section), it would be possible to calculate the distance between the current values of scattering and tangling and their optimized values achievable, if the application is restructured according to feature-oriented criteria.

Based on this, we define the *drift of feature-oriented modularity* in an application as the distances between the absolute and the optimal values of the scattering, measured here using FSCA, and of tangling, measured here using FTANG.

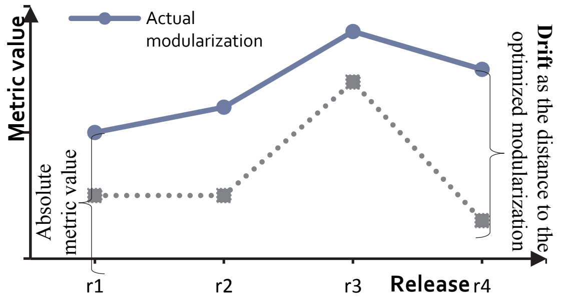


Figure 3: Relativity of metric drift.

As schematically depicted in Figure 3, the drift of feature-oriented modularity can be plotted over time for a given application to serve as a metaphorical *compass* that indicates how much the modularization of features diverges from the optimum with each subsequent release. Observing the drift trends can be used in several ways by developers to determine the need for initiating feature-oriented restructurings of the next releases of their applications.

The compass views of scattering and tangling drifts can be used to identify periods in which restructuring efforts would be most beneficial. Types of such periods include the ones in which the drift constitutes a large portion of the absolute metric value. An example of such a period is the release *r4* in Figure 3, where there is a large potential for reducing the absolute metric value by improving modularization of features. Moreover, in the release *r4* the drift increased significantly with relation to the previous release, and therefore restructuring could be considered in *r4* to prevent further divergence of the application's modularization from the optimum in the next release.

Furthermore, by contrasting the drift plots for scattering and tangling, one can determine the character of restructuring most needed at a given point in time. For instance, large drift of scattering indicates a need for improving localization of individual features within modules, which may require reducing the overall number of modules. In contrast, large drift of tangling indicates a need for improving separation of features within modules, which may require increasing the overall number of modules.

3.3 Calculating Drift using Optimization

As demonstrated by Murphy et al. (Murphy et al., 2001), there exist tradeoffs between the known approaches to improving modularization of features. Firstly, Murphy et al. found pure class-based refactorings to have a limited potential for separating tangled features. In contrast, approaches based on AspectJ and Hyper/J were found to have a better separation potential, but also more difficult to apply and making some of the resulting isolated code fragments difficult to understand. In addition, it was demonstrated that aspect-oriented techniques are sensitive to the order of composition, which resulted in coupling of features to one another.

Given the known technical characteristics and automation potentials of the existing methods for separating features, the modularization compass approach is based on regrouping classes in terms of packages to reduce scattering and tangling of features. While this purely class-based approach has limits to the level of feature separation that it can achieve, it has the important property from the point of view of this work that it allows for complete automation of searching for desired feature-oriented package structures and subsequently establishing them in source code by using refactorings.

In order to calculate the drift of feature-oriented modularity, the modularization compass approach uses the so-called feature-oriented remodularization. *Feature-oriented remodularization* is the process of multi-objective optimization of the distribution of classes among packages, which aims at identifying Pareto-optimal package structures that minimize both scattering FSCA and tangling FTANG metrics (Olszak and Jørgensen, 2012).

In addition, this formulation encompasses two traditional object-oriented objectives that govern the inter- and intra-module dependencies among class, i.e. the objectives of maximizing cohesion in packages and minimizing coupling among packages. Formalized definitions of the four metrics used as evaluation criteria for the mentioned optimization objectives are listed in Figure 4. There, the set of all features in an application is denotes as *F,* the set of all packages that contribute to at least one feature as P_F, and the set of all types as *T*.

$$\boldsymbol{FSCA}(F) = \sum_{f \in F} \frac{|\{p \in P_F : f \rightsquigarrow p\}|}{|F|} \qquad \rightarrow \textbf{min}$$

$$\boldsymbol{FTANG}(P_F) = \sum_{p \in P} \frac{|\{f \in F : f \rightsquigarrow p\}|}{|P_F|} \qquad \rightarrow \textbf{min}$$

$$\boldsymbol{PCOH}(P) = \frac{\sum_{\substack{p \in P \\ T \Rightarrow p}} pcoh(p,T)}{|P|}, \qquad \rightarrow \textbf{max}$$

$$\boldsymbol{pcoh}(p,T) = \frac{\sum_{\substack{t1 \in T \\ t1 \Rightarrow p}} \sum_{\substack{t2 \in T \\ t2 \Rightarrow p}} |DD_{t1,t2} \cup DM_{t1,t2}|}{\sum_{\substack{t1 \in T \\ t1 \Rightarrow p}} \sum_{\substack{t2 \in T \\ t2 \Rightarrow p}} |MaxDD_{t1,t2} \cup MaxDM_{t1,t2}|}$$

$$\boldsymbol{PCOUP}(P) = \sum_{\substack{p \in P \\ T \Rightarrow p}} pcoup(p,T), \qquad \rightarrow \textbf{min}$$

$$\boldsymbol{pcoup}(p,T) = \sum_{\substack{t1 \in T \\ t1 \Rightarrow p}} \sum_{\substack{t2 \in T \\ t2 \not\Rightarrow p}} |DD_{t1,t2} \cup DM_{t1,t2}|$$

Figure 4: Objectives for optimizing modularity of features.

The definitions of FSCA and FTANG correspond to the ones mentioned earlier and are simplified versions of the metrics proposed by Brcina and Riebisch (Brcina and Riebisch, 2008) that are defined based on the $\rightsquigarrow$ (i.e. "implemented by") relation between features and packages. The reformulation made in this work removes the additional normalization factors and makes the metrics correspond directly to the numbers of features tangled in a package, and packages that a feature is scattered over. Doing so allows for easier interpretation of the metric values, and is possible due to the modularity drift calculation being independent of metric normalization, as discussed earlier.

The cohesion metric PCOH is the package-level version of the RCI metric based on data-data (*DD*) and data-method (*DM*) relations proposed by Briand et al (Briand et al., 1998). In its essence, this metric computes for the set of packages P the average quotient of the actual number of intra-package static dependencies among classes and the maximum possible number of such dependencies. In turn, the package coupling metric PCOUP corresponds to a sum of the ACAIC, OCAIC, ACMIC, and OCMIC coupling measures, as defined by the same authors in (Briand et al., 1999), and thereby constitutes the sum of all inter-package static dependencies in an application.

The actual process of optimizing the application's modularity with respect to all the metrics is performed using a tailored formulation of a genetic algorithm that we refer to as *multi-objective grouping genetic algorithm* (MOGGA) (Olszak and Jørgensen, 2012). The multi-objectivity is achieved by exploiting the notion of Pareto-optimality, whose efficiency in optimizing modularization of software systems according to multiple conflicting criteria was demonstrated by Harman and Tratt (Harman and Tratt, 2007). The grouping nature of the problem is exploited by using a set of tailored genetic operators based on the work of Seng et al. (Seng et al., 2005), who demonstrated their significant effect on improving the efficiency of traversing the search space of alternative modularizations. Hereby, MOGGA constitutes a composition of these two well-established approaches that is aims at leveraging their respective advantages.

In its essence, MOGGA evolves a population of individuals by means of *selection*, *reproduction* and *mutation* driven by the score of the individuals with respect to a fitness function. Each individual represents a particular distribution of classes among packages, expressed by an array of integers. Within this array, classes are represented by indexes in the arrays, and their assignment to packages is represented by the values of the corresponding array cells.

MOGGA adapts two genetic operators that exploit the grouping-based nature of the remodularization problem. First, the crossover operator that forms two children from two parents is made to preserve packages as the building blocks of modularizations. Secondly, a mutation operator is defined to randomly perform one of three actions: merge two packages with the smallest number of classes, split the largest package into two packages, and adopt an *orphan class* (Tzerpos and Holt, 2000) being alone in a package into another package.

Evaluation of the fitness of the individual modularization alternatives is done by computing the four metrics of FSCA, FTANG, PCOH and PCOUP. In order to appropriately represent the regions of the four-dimensional search space that the individual modularizations in the population occupy, MOGGA adopts the concept of Pareto-optimality. Hence, the fitness of each individual becomes a tuple consisting of four independent metric values. Such a multi-modal fitness is used for comparing individuals based on the *Pareto-dominance* relation, which states that one out of two individuals is better than the other individual, if all of its fitness values are not worse, and at least one of the values is better. Thereby, it becomes possible to partially order individuals and to determine the set of non-dominated individuals in a population, i.e. the so-called Pareto-front.

Starting with an initial population consisting of 98% randomized individuals and 2% of the individuals from the original modularization, a predefined number of evolutionary iterations are executed. Then the last Pareto-front is used to select a single individual being the optimization result. This is done by ranking the individuals in the obtained four-dimensional Pareto-front with respect to each metric separately, and then choosing the individual that is ranked best on average. Please note that while this method is used here, existing literature defines a range of diverse methods for choosing a single solution out of a Pareto-front.

4 EVALUATION

We have implemented the presented remodularization approach as part of the freely available Featureous tool for feature-oriented analysis of Java software (Featureous). The code transformations required for establishing the source-code modularizations were implemented using the Recoder code transformation library (Recoder). Furthermore, as will be discussed later, this evaluation relies on a dynamic feature-location approach provided by Featureous.

The goal of the study presented in this section is formulated as follows:

To evaluate whether drift-based metrics bring new insights into the evolution of feature-oriented modularity of applications, as compared to using their absolutes values.

This is done by applying the approach to long-term release histories of three open-source Java applications that were chosen based on their size, maturity and availability of the historical revisions. The used applications are: *RText* – a text editor for programmers (17 releases spanning, 3 years) (RText), *FreeMind* – a mind-mapping tool (13 releases, 5 years) (FreeMind) and *JHotDraw Pert* – a diagramming application being a showcase for the JHotDraw framework (11 releases, 8 years) (JHotDraw).

4.1 Results of Feature Location

While the modularization compass approach does not impose any constraint on the feature-location approach to be used, we have chosen to use the dynamic feature-location approach provided by the Featureous tool. This feature-location approach identifies code units involved in implementing individual features by tracing the execution of an instrumented program during its interaction with a user. The tracing agent used for this purpose is guided by annotations that have to be placed by a programmer at appropriate starting methods of each feature. Apart from the use of annotations and user-driven feature triggering, this approach remains analogous to other dynamic approaches, such as software reconnaissance (Wilde et al., 1992). An extensive discussion of the conceptual and technical details of the used feature-location approach can be found in (Olszak and Jørgensen, 2012).

The part of the feature-location process that was most sensitive to human interpretation was the recovery of feature specifications for each release of the three investigated applications. We have performed this recovery by inspecting the available user documentation and by listing the functionality exposed in the user interfaces of the applications. Table 1 lists the identified features and the release in which they were added to the systems, if they were added during the investigated periods.

Table 1: Investigated releases and their identified features.

Application releases	Identified features

RText
 	
<u>Releases</u>:
0.8.0; 0.8.1;
0.8.2; 0.8.3;
0.8.4; 0.8.5;
0.8.6; 0.8.7;
0.8.8; 0.8.9;
0.9.0; 0.9.2;
0.9.3; 0.9.4;
	Display text, Document properties (0.9.0), Edit basic, Edit text, Exit program, Export document (0.8.7), Init program, Modify options, Customize text (0.9.0), Multiple documents, Navigate text, New document, Open document, Playback macro (0.9.0), Print document, Record macro, Save document, Show documentation, Source browser (added in 0.8.8 and removed in 0.9.0), Undo redo, Plugins (0.9.0)

FreeMind
 	
<u>Releases</u>:
0.0.2; 0.0.3;
0.1.0; 0.2.0;
0.3.0; 0.3.1;
0.4.0; 0.5.0;
0.6.0; 0.6.1;
0.6.5; 0.6.7;
0.7.1
	Browse mode (0.3.0), Cloud node (0.7.1), Display map, Show documentation (0.2.0), Edit basic, Edit map, Evaluate (0.3.0), Exit program, Export map (0.5.0), File mode (0.1.0), Icons (0.6.7), Import/export branch (0.2.0), Init program, Link node (0.0.3), Modify edge, Modify node, Multiple maps (0.0.3), Multiple modes (0.1.0), Navigate map, New map, Open map, Print map (0.03), Save map, Zoom

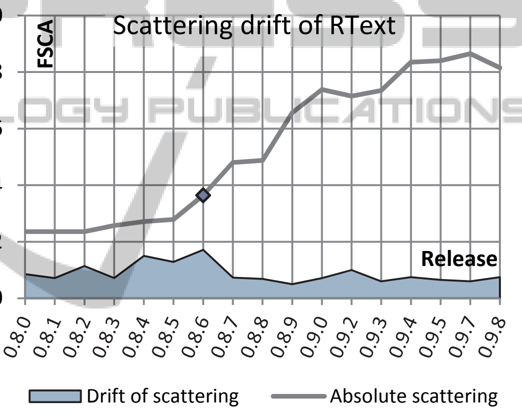
JHotDraw Pert <u>Releases</u>: 5.2; 5.3; 5.4b1; 6.0b1; 7.0.7; 7.0.8; 7.0.9; 7.1; 7.2; 7.3; 7.3.1	Align, Dependency tool, Edit basic, Edit figure, Exit program, Export drawing (7.0.7), Group figures, Init program, Line tool (removed in 6.0b1), Modify figure, Multiple windows (7.0.7), New drawing, Open drawing, Order figures, Save as drawing, Selection tool, Snap to grid, Task tool, Text tool, Undo redo (5.3), Zoom (7.0.7)

4.2 Results of Feature Drift Measurement

In this evaluation, the drift of feature-oriented modularity was calculated by executing MOGGA on each release of the three applications. Based on observations from a series of pilot executions of the MOGGA on the target applications, we arrived at the following configuration of the algorithm that reduces the overall execution times while preserving high optimization level of the resulting modularizations. MOGGA was executed for a population of 300 individuals for 500 evolutionary iterations with mutation probability of 5%. This configuration of the algorithm was applied to each release ten times to reduce the impact of non-determinism of genetic computation. The best of the solutions found was used as the final result for each release. It is worth mentioning that while this configuration of MOGGA was observed to produce Pareto-optimal solutions in acceptable timeframes for all the investigated releases (i.e. in the order of magnitude of days), further adjustments to the algorithm parameters could lead to reducing these times even further.

The results of measuring the drift of feature-oriented modularity using MOGGA are presented in the form of compass views in Figure 5 for RText, in Figure 6 for FreeMind, and in Figure 7 for JHotDraw Pert. For each application, two plots are shown – one for evolution of scattering and one for evolution of tangling. In the plots, the absolute metric values are displayed as a line, whereas the calculated drift is displayed as an area at the bottom of the plots. This is aimed at simplifying the observation of development and relation of the drift to the absolute metric value.

The scattering drift plot for RText, shown in Figure 5, can be divided into two distinct periods. The first period, ranging from the release 0.8.0 to the 0.8.6, is a period of overall growth of the scattering drift. Despite of minor reductions observed in a few intermediate releases (i.e., 0.8.1, 0.8.3 and 0.8.5), the drift value doubled in this first period.



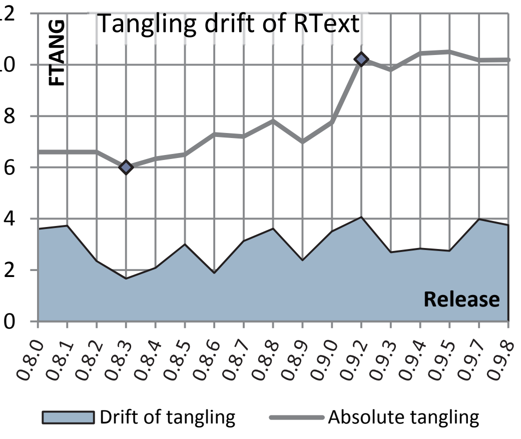


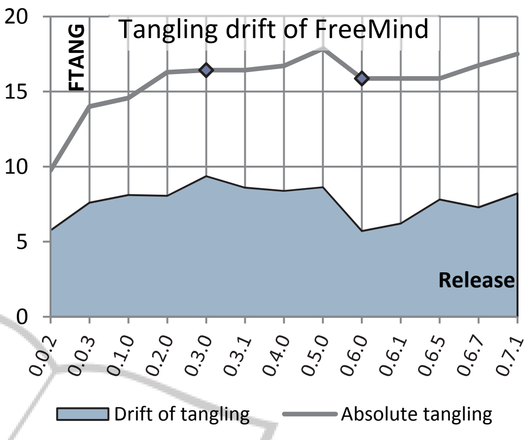
Figure 5: Drift measurements for releases of RText.

This was also the period, in which the drift increased together with the absolute scattering and constituted on average 42% of the scattering's value. During the second period, between the releases 0.8.6 and 0.9.8, the drift was initially decreased, and thereafter maintained a relatively constant level. Interestingly, this was achieved despite of an over twofold increase in the absolute scattering of the application. This indicates that the modularization decisions of the developers with regard to confining features to a small number of packages were close to optimum in this period.

The tangling drift plot for RText, shown in Figure 5, contains three interesting periods. Firstly, the period between the releases 0.8.0 and 0.8.3 is the period of sharp decreases of drift and absolute tangling and a decrease of the relative contribution of drift to the absolute tangling value. Secondly, between the releases 0.8.3 and 0.9.2, both the drift and the absolute tangling were increasing at a similar rate. Despite the overall growth, the drift appears here to be periodically reduced by the developers. Lastly, in the period 0.9.2 to 0.9.8 both the drift and the absolute tangling remain fairly constant. It is also this period, where the relative contribution of the drift is the lowest. However, it remains significantly higher than the relative contribution observed earlier of the scattering drift. Together, this data indicates that the features of RText were better localized than separated from one another in terms of packages.

The scattering drift plot for FreeMind, shown in Figure 6, depicts several oscillations of the scattering drift over time. Initially, the oscillations are stronger but they eventually weaken over time. In comparison, the value of the absolute scattering of the application increases sharply between the releases 0.0.2 and 0.1.0, and thereafter remains approximately constant over the next 10 releases. This suggests that the application structure established at release 0.1.0 served well for the purpose of adding new features and extending the existing ones in a localized fashion.

The tangling drift plot for FreeMind, shown in Figure 6, can be divided into three periods: the period of increasing drift and increasing absolute tangling (0.0.2 – 0.3.0), the period of decreasing drift and stabilized absolute tangling (0.3.0 – 0.6.0), and the period of continued growth in both the drift and the absolute tangling. It can be seen that the overall changes of tangling drift and the absolute tangling reflect each other over time; only a minor difference in the growth rates can be observed, i.e. in the release 0.0.2 the drift constitutes 59% of the absolute tangling value, whereas in release 0.7.1. it constitutes 47% of the absolute tangling value. This high contribution indicates that FreeMind has a relatively high potential for improving the separation of features through source code restructuring.



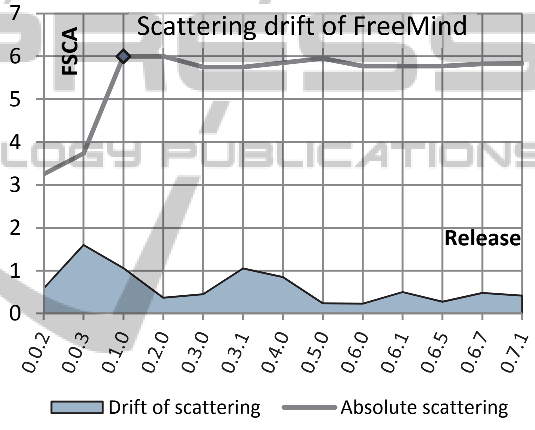


Figure 6: Drift measurements for releases of FreeMind.

A potential trace of such efforts undertaken by the FreeMind developers is the transition from the release 0.5.0 to 0.6.0, where the drift of tangling was reduced by 34%.

In both the scattering and tangling drift plots for JHotDraw Pert, shown in Figure 7, it can be seen that the feature-oriented evolution of the application underwent a dramatic shift after release 6.0b1. Up till then, both the drifts and the absolute values of scattering and tangling were generally increasing. Starting from the release 7.0.7, these trends have changed. During the transition from 6.0b1 to 7.0.7, the drift of scattering was reduced almost completely, despite an increase in the absolute scattering, and both the drift and the absolute value of tangling were decreased significantly. Thereafter, both scattering and tangling drifts experienced only very small increases, whereas the absolute scattering value continued to rise and the absolute tangling value continued to slightly decrease.

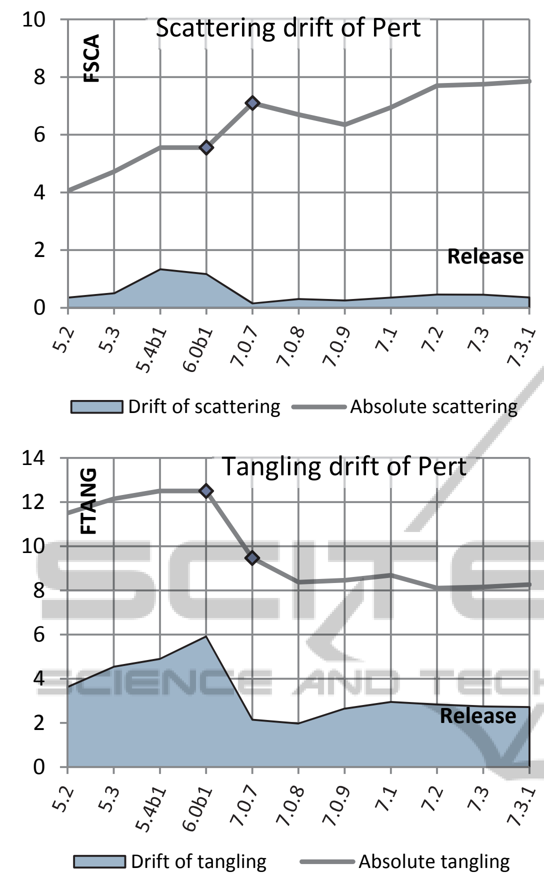


Figure 7: Drift measurements for releases of JHotDraw Pert.

It turns out that these observations find their reflection in the types of work on the application that the developers undertook in the period preceding the 7.0.7 release. The release notes from that period mention a large-scale architectural refactoring of the underlying JHotDraw framework. While it is difficult to tell whether improving the separation of individual features of Pert was among the intentions of these refactorings, it certainly became one of the results. Furthermore, the obtained reductions for both the drifts and the absolute value of scattering and tangling have shown to remain fairly stable after the source code refactoring – especially if compared to the rapid developments prior to the refactoring. Interestingly, the absolute value of tangling began to decrease over a longer period, which is a behavior unseen in the two other investigated software applications.

4.3 Discussion

The reported study applied the modularization compass approach to three real-world Java applications. The measured drift values were observed to evolve over the subsequent releases of the three applications in ways that were not trivially related to evolution of the absolute metric values. This indicates that for the study subjects, the drift measurements add a new type of information about the evolution of the applications' modularity over time.

The obtained drift measurements were used as an input to formulating a number of hypotheses about the reasons for the observed changes of the applications' feature-oriented modularity over time and a number of restructuring recommendations.

Overall, in all the investigated applications the tangling drift constituted a significantly higher portion of FTANG than the scattering drift did for FSCA. This suggests that it is the separation of features from one another, rather than their confinement in few packages, that should be the primary restructuring goal for the three investigated applications. While at this point it is not possible to judge whether the insufficient separation of features is a common trait of layered object-oriented architectures, we see it as a viable hypothesis for further investigation.

Furthermore, periodical oscillations of the drift were observed in several cases that were not observed on the absolute metric values. This initial observation appears possibly be related to the observations of Anton and Potts (Antón and Potts, 2003) about the burst-like nature of adding new features. In a 50-year evolution of a telephone system, they observed new features to be introduced in discrete bursts, i.e. they exhibit punctuated rather than incremental or gradual evolution. These bursts were typically followed by periods of retrenchment that merged similar features and phased out older versions of new features. In our context, burst-like additions or enhancements of features could have resulted in rapid increases of drift, which were thereafter reduced during retrenchment periods.

There are several threats to validity of the obtained results, as well as several aspects of the presented study that can be improved in the future.

Firstly, in order to strengthen the internal validity of the results, a separate systematic exploration of MOGGA configuration parameters can be performed to improve the configuration of the algorithm. Even though care was taken during the configuration process to obtain a set of parameters that produces the best observable solutions, it remains interesting to compare the performance of different configurations of MOGGA and other optimization approaches. Performing such a systematic follow-up comparison, similarly as done by Mitchell and Mancoridis (Mitchell and Mancoridis, 2007), would be important to learning about the characteristics of MOGGA.

Secondly, it is worthwhile to equip the remodularization approach with method-level refactorings (e.g. move method, extract method, etc.), so that drift at the granularity of methods can also be detected. Nevertheless, such a refinement is expected to have only a limited impact on the results presented in this paper, according to an earlier work of the authors showing that method-level refactorings only have a minor effect on scattering and tangling optimization (Olszak and Jørgensen, 2012).

Lastly, while the presented work was motivated by influence of modularization of features on evolvability of software, it remains possible to apply the modularization compass approach to other characteristics of software design. This can be done as long as these 'other characteristics' are quantifiable and can be shaped by means of source-code restructuring. In practice, the presented approach can be re-purposed by replacing the metrics that are used to drive the MOGGA. Guidelines for doing so can be found in the work of Harman and Clark (Harman and Clark, 2004).

5 CONCLUSION

The ability to change is both a blessing and a burden to software. On the one hand, it allows systems to adapt to changing requirements imposed by users. On the other hand, changing existing source code is often difficult and the adoption of repetitive changes tends to erode the original structure of source code.

The work presented in this paper focused on the drift of feature-oriented modularity during the evolution of software applications. The proposed approach called modularization compass measures this type of drift by comparing the original version of an application to its automatically remodularized counterpart. The remodularization process is performed by using a multi-objective grouping genetic algorithm that uses metrics of scattering, tangling, cohesion and coupling as the objectives for package structure optimization.

The approach was implemented in Java, and applied to three open-source Java applications. The obtained compass views showed the significant differences between the evolution of absolute values of scattering and tangling and the evolution of their drifts. Based on the analysis of drifts over subsequent releases, we were able to identify when restructuring brings the largest improvement in feature modularity, and to determine that the restructuring effort for all three applications should focus on separating features from one another to reduce the significant drifts of their tangling.

Finally, the design and the evaluation of the approach resulted in several promising directions for future research and provided several preliminary observations about the general nature of evolution of software features.

REFERENCES

Recoder, http://recoder.sourceforge.net/

Hou, D. and Wang, Y. 2009. An empirical analysis of the evolution of user-visible features in an integrated development environment. In Proceedings of the 2009 Conference of the Center For Advanced Studies on Collaborative Research, CASCON '09, 122-135.

Antón, A. I. and Potts, C. 2003. Functional Paleontology: The Evolution of User-Visible System Services. IEEE Trans. Softw. Eng. 29, 2, 151-166.

Hsi, I. and Potts, C. 2000. Studying the Evolution and Enhancement of Software Features. ICSM'00: In Proceedings of the International Conference on Software Maintenance, pp. 143.

Fischer, M. and Gall, H. 2004. Visualizing feature evolution of large-scale software based on problem and modification report data: Research Articles. J. Softw. Maint. Evol. 16, 6 (Nov. 2004), 385-403.

Greevy, O., Ducasse, S., Girba, T. 2005. Analyzing feature traces to incorporate the semantics of change in software evolution analysis. ICSM '05: Proceedings of the International Conference on Software Maintenance, pp. 347-356.

Turner, C. R., Fuggetta, A., Lavazza, L. and Wolf, A. L. 1999. A conceptual basis for feature engineering. Journal of Systems and Software, vol. 49, no. 1, pp. 3-15.

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. Communications of the ACM, vol. 15, no. 12, pp. 1053-1058.

Rugaber, S., Stirewalt, K. and Wills, L. M. 1995. The interleaving problem in program understanding. WCRE'95: In Proceedings of 2nd Working Conference on Reverse Engineering, pp. 166-175.

Letovsky, S. and Soloway, E. 1986. Delocalized plans and program comprehension. IEEE Software, vol. 3, no. 3, pp. 41-49.

Eaddy, M., Zimmermann, T., Sherwood, K. D., Garg, V., Murphy, G. C., Nagappan, N. and Aho, A. V. 2008.

Do crosscutting concerns cause defects?. IEEE Transactions on Software Engineering, 34, 497-515.

Benestad, H. C., Anda, B. and Arisholm, E. 2009. Understanding cost drivers of software evolution: a quantitative and qualitative investigation of change effort in two evolving software systems. Journal of Empirical Software Engineering, 15(2), 166-203.

Van Den Berg, K., Conejero, J. M. and Hernández, J. 2006. Analysis of crosscutting across software development phases based on traceability. In EA'06: Proceedings of the 2006 international workshop on Early aspects at ICSE, 43–50.

Brcina, R. and Riebisch, M. 2008. Architecting for evolvability by means of traceability and features. In 23rd International Conference on Automated Software Engineering - Workshops, pp. 72-81.

RText, http://fifesoft.com/rtext/

FreeMind, http://freemind.sourceforge.net/

JHotDraw, http://www.jhotdraw.org/

Wilde, N., Gomez, J.A., Gust, T., Strasburg, D. 1992. Locating user functionality in old code. ICSM'92: In Proceedings of the 1992 International Conference on Software Maintenance, pp.200-205.

Tarr, P., Osher, H., Harrison, W., Sutton, S. M. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. ICSE'99: In Proceedings of the 21st International Conference on Software Engineering, pp 107-119.

Featureous, http://featureous.org/

Briand, L. C., Daly, J. W. and Wüst, J. 1998. A unified framework for cohesion measurement in object-oriented systems. Journal of Empirical Software Engineering, vol. 3, no. 1, pp. 65-117.

Briand, L. C., Daly, J. W. and Wüst, J. 1999. A unified framework for coupling measurement in object-oriented systems. IEEE Transactions on Software Engineering, vol. 13, no. 2, pp. 115-121.

Seng, O., Bauer, M., Biehl, M. and Pache, G. 2005. Search-based improvement of subsystem decompositions. In Proceedings of the 2005 conference on Genetic and evolutionary computation , 1045-1051.

Olszak, A. and Jørgensen, B. N. 2012. Remodularizing Java Programs for Improved Locality of Feature Implementations in Source Code. Science of Computer Programming, Vol. 77, no. 3, pp. 131-151.

Lehman, M.M. 1980. Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68(9), 1060-1076.

Harman, M. and Tratt, L. 2007. Pareto Optimal Search Based Refactoring at the Design Level. GECCO'07: In Proceedings of the 9th annual conference on Genetic and evolutionary computation, 1106-1113.

Tzerpos, V. and Holt, R. C. 2000. ACDC: An Algorithm for Comprehension-Driven Clustering. WCRE'00: In Proceedings of Seventh Working Conference on Reverse Engineering, 258-267.

Olszak, A. and Jørgensen, B. N. 2012. Modularization of Legacy Features by Relocation and Reconceptualization: How Much is Enough? CSMR'12: In Proceedings of the 16th European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, pp. 171-180.

Harman, M. and Clark, J. 2004. Metrics Are Fitness Functions Too. METRICS'04: In Proceedings of the IEEE International Symposium on Software Metrics, 58-69.

Mitchell, B. S., Mancoridis, S. 2007. On the evaluation of the Bunch search-based software modularization algorithm. Journal of Soft Computing. 12, 1, 77-93.

Murphy, G. C., Lai, A., Walker, R. J. and Robillard, M. P. (2001). Separating features in source code: an exploratory study. In ICSE'01: Proceedings of the 23rd International Conference on Software Engineering, 275-284.