Modularization of Legacy Features by Relocation and Reconceptualization: How Much is Enough?

Andrzej Olszak and Bo Nørregaard Jørgensen
The Maersk Mc-Kinney Møller Institute
University of Southern Denmark, Odense, Denmark
{ao, bnj}@mmmi.sdu.dk

Abstract—As programs become larger and start evolving, they often need to be split into modules, in order to facilitate independent evolution of end-user features and consolidate reusable core abstractions. Achieving this for legacy object-oriented software is, however, problematic due to scattering and tangling of feature implementations. While relocation of classes among packages can be used to significantly reduce these phenomena, achieving complete separation of features requires reconceptualization of existing classes. In this paper, we investigate the tradeoffs between relocation and reconceptualization of classes during a migration of the NDVis neuroscience application to the NetBeans Module System. We do this by comparing the manually modularized version of NDVis with three automatically optimized designs that exhibit various degrees of relocation and reconceptualization. The obtained findings shed new light on the actual degree and impact of relocation and reconceptualization during modularization of legacy features.

Keywords: features; restructuring; design; modularity

I. INTRODUCTION

Feature is a unit of user-identifiable functionality of a program [10]. The concept of features provides a practical means of correlating functional user-oriented specifications of a program with a program’s source code. This property becomes especially useful during software evolution and maintenance, where legacy code needs to be modified in a feature-centric fashion due to feature-centric requests from the users. Hence, it becomes vital that programmers can efficiently locate, understand and modify the fragments of source code implementing individual functional concerns [3].

However, in many object-oriented legacy systems the correspondences between features and structural units of source code are neither explicit nor direct. This misalignment between features and structural units manifests itself as the phenomena of scattering, which describes delocalization of a feature among structural units, and tangling, which describes the number of features overlapping on a structural unit [11].

In monolithic applications, usage of scattered and tangled functional concerns as the units of program modification leads to cognitive overhead known as delocalized plans [13] and interleaving [14]. Such misalignment was also reported to be one of the primary factors contributing to change effort and error-proneness during software evolution [4][12][5].

The phenomena of scattering and tangling become especially significant during migrations of legacy monolithic applications to module systems, such as OSGi or NetBeans Module System [9]. Dividing a monolithic code base into explicitly isolated modules can by itself result in new kinds of evolutionary pressures [19] and it can amplify the above mentioned problems. If not modularized properly, crosscutting feature implementations are bound to hinder module-wise work division, increase change propagation among modules, and effectively prevent modular addition and deployment of single features. Hence, migration of a monolithic application to a module system often involves designing new structural units as the basis for modules, rather than just reusing the legacy ones.

Restructuring of a legacy application to localize and separate feature implementations in terms of structural units is known as feature-oriented refactoring [15] or feature-oriented remodularization [16]. One of the important conceptual differences among the existing approaches to doing so is the treatment of legacy classes. In these approaches, such as FOR [15] that aims at extraction of software product lines, classes are being manually split into feature-specific fragments, often using advanced mechanisms for separation of concerns [2]. We call this process reconceptualization. On the other end of the spectrum, there are approaches that preserve legacy classes and automatically move them among packages to reduce scattering and tangling [16]. We call this process relocation.

A question remains, what are the maximum achievable impacts of relocation and reconceptualization on modularity of features, as schematically visualized in Fig. 1. Furthermore, one needs to identify the parts of this scale that tend to be occupied by real-world cases of manual feature-oriented remodularizations. These issues need to be addressed in order to better understand the role and importance of relocation and reconceptualization during migrations to module systems. In particular, one needs these answers to identify an optimal degree of reconceptualization, which, while enabling additional

Figure 1. A hypothetical impact of relocation and reconceptualization.
separation of features, tends to be more complex and more difficult to automate than relocation.

In this paper, we investigate the degree and impact of class reorganization and relocation during a case of feature-oriented migration to a module system. The reported study is centered on manual migration of a monolithic Java application for visualizing neurological databases, called NDVis [6], to the NetBeans Module System. By contrasting the results of the manual restructuring with three other automatically-optimized designs, we compute the maximum achievable impacts of reorganization and relocation of classes. We use these results as a scale for evaluating the degree and discussing the implications of the manually performed restructurings.

This paper is organized as follows. In Section II, we motivate feature-oriented remodularization and migration to module systems. In Section III, we discuss the concepts of relocation and reorganization. Section IV introduces the NDVis application, which is being restructured in Sections V, VI, VII and VIII. Section IX and X discuss the obtained results and threats to validity. Finally, Section XI presents the related work and Section XII concludes the paper.

II. FEATURE-ORIENTED MIGRATION TO MODULE SYSTEMS

As legacy monolithic applications grow in size and functionality, there emerges a need for managing complexity by dividing their source code into explicit parts known as modules or components. Ideally, such modules are made suitable for independent comprehension, change and work division, so that a change to one module does not propagate to others and that two different changes affect only their own respective modules. The technical means of dividing source code into modules are provided by so-called module systems.

Similarly to plain JAR files, module systems are the means of dividing programs into separate units of development, compilation and deployment. In addition, module systems make it possible to declare a version number for a module, to separate exported packages from private ones and to establish explicit dependencies among modules. Moreover, most module systems make it possible to load and unload module at runtime and provide a mechanism for dynamic discovery of new services by means of locating available implementations of a given service-provider interface [9].

Having the technical means of creating modules, one needs to decide how to use them, i.e. one needs to design a beneficial division of monolithic code base. As discussed by Lehman [19], failure to arrive at a beneficial division of source code can result in new evolutionary forces, which reduce the advantages of using a module system. However, designing such a beneficial division is difficult, since it can rarely be achieved by promoting the legacy structural units, i.e. Java packages, to the status of modules. This is because modules are the explicit units of evolution, composition and deployment, which is a fundamentally different purpose from that of packages. Hence, migration to a module system is bound to involve restructuring of a legacy source code.

In order to design a structural division that is beneficial during software evolution, one needs to adapt it to dimensions of change, customization and reuse that will be enforced by forthcoming change requests.

One of the dimensions being crucial for many applications is the dimension of user-identifiable functional features. As stated by the sixth law of software evolution [20], change requests originating from software users are an important driver of evolutionary changes. Since these requests tend to be formulated in terms of features, they impose features, rather than other technical concerns, as the units of evolutionary change, customization, reuse and work division.

Since most legacy object-oriented programs do not localize and separate feature implementations in terms of packages, they need to undergo a restructuring during their feature-oriented migrations to a module system. Such restructuring, known as feature-oriented remodularization, is a behavior-preserving transformation of source code aiming at untangling, localizing and making the feature implementations structurally explicit. As a result, it becomes possible to create modules dedicated to individual features and a set of reusable core modules. The outcomes of doing so include better confinement of feature-centric changes, facilitation of feature-wise division of work, feature-wise code reuse and deployment-time customization of functionality by inclusion/exclusion of feature modules.

III. RELOCATION VERSUS RECONCEPTUALIZATION

One of the important decisions during feature-oriented remodularization, and during any restructuring in general, is the approach to relocating and reconceptualizing code units.

For the needs of this discussion, we introduce the following terminology. Computational unit is a unit of computation in a program. We distinguish five types of computational units ordered according to their granularity: modules, packages, classes, methods, and instructions. Granularity $N$ is a granularity level corresponding to a given computational unit. Accordingly, granularity $N+1$ corresponds to the following finer level of granularity than $N$ (e.g. methods (N+1) are the following finer granularity than classes (N)). Similarly, granularity $N-1$ stands for the following coarser level of granularity. We say that a computational unit is single-feature when it is used exclusively by implementation of one feature. We say that a computational unit is multi-feature when it is used by implementations of more than one feature.

Relocation at granularity $N$ is the process of moving a computational unit at granularity $N$ from its containing computational unit at granularity $N-1$ to another computational unit at granularity $N-1$. As relocation does not involve modifying a computational unit being relocated, it preserves its original semantics. An example of relocation at granularity of classes is the move class refactoring [21].

Reconceptualization at granularity $N$ is the process of modifying a computational unit at granularity $N$ by changing or relocating one or more of its contained computational units at granularity $N+1$. As a result of modifying a computational unit, reconceptualization is likely to modify its semantics as well. An example of reconceptualization at granularity of classes is using the move method refactoring [21].
In order to separate and localize implementations of features at granularity N, relocation has to be performed at least at granularity N+1 and reconceptualization has to be performed at least at granularity N.

In the case of separating features in terms of classes (N=class), existing classes (N) have to be reconceptualized. This can be done by splitting classes through relocating all their feature-specific methods (N+1) to new feature-specific classes. If a class contains any multi-feature methods, then a complete separation cannot be achieved at granularity of classes (N) and it also needs to be also performed at granularity of methods (N+1). This is done by reconceptualizing methods through relocation of their single-feature instructions (N+2) to new single-feature methods (N+1). As instructions are the lowest level of granularity, which makes further reconceptualization impossible, the remaining multi-feature instructions can only be separated among features by creating a duplicate for each feature involved. While this might be an acceptable tradeoff in the case of separating method bodies, it becomes more problematic when applied to fields and variables, as introducing state duplication requires providing additional means of synchronizing state among duplicates.

In the case of localizing features in terms of packages (N=package), existing packages (N) have to be reconceptualized. This can be done by creating one feature-specific package for each feature and relocating to it all its feature-specific classes (N+1). If there exist multi-feature classes, then a complete localization cannot be achieved at granularity of packages (N) and separation needs to be performed at granularity of classes (N+1). By following the process that we have discussed earlier it is possible to split multi-feature classes into new feature-specific classes, which can be relocated to their appropriate single-feature packages.

From the presented discussion it follows that relocation at granularity N manifests itself as reconceptualization at granularity N-1. This describes well the process of splitting a multi-feature class into several single-feature classes (reconceptualization of a class) by moving its methods (relocation of methods). Furthermore, it can be seen that the applicability of relocation to separating and localizing features at granularity N is determined by the degree of separation of features present at granularity N+1. Hence, reconceptualization at finer granularities acts as an enabler of relocation at coarser granularities. For instance, a class can be cleanly split among features using relocation of methods only if the features are already split at method granularity.

The mentioned need for refining the granularity of relocation and reconceptualization, while beneficial, is also problematic for several reasons. Firstly, refining the granularity of syntactic separation often creates a need for more advanced mechanisms for separation of concerns (e.g. aspects [2] or derivatives [15]) than the ones available in mainstream object-oriented programming languages. Secondly, the finer the granularity of restructuring, the more complex, error-prone and difficult to automate the restructuring process becomes. Thirdly, usage of reconceptualization to split legacy classes into feature-specific fragments forces one to invent new abstractions to describe the semantics of the created class-fragments, and thereby to delocalize the implementations of the domain concepts that the original classes represented.

As the mentioned issues contribute to the overall cost of restructuring, it is important to understand the need for and implications of relocation and reconceptualization of classes.

IV. THE CASE OF NDVis

NDVis by VisiTrend is a 15 KLOC Java-based tool for visualization and analysis of large multi-dimensional neurological datasets [6]. After completing the initial development of the tool, the owning company decided to migrate the project to the NetBeans Rich Client Platform and modularize it using the NetBeans Module System [9]. The high-level aim was to facilitate independent evolution and deployment of features, and to enable code reuse across multiple project branches as well as a larger portfolio of applications being developed at VisiTrend.

Having initially no knowledge of the design, implementation or problem domain of NDVis, we have joined the project as external contributors. Our aim was to provide it with the properties envisioned by the owners by means of feature-oriented remodularization. In the following sections of this paper, we report on our manual efforts to restructure NDVis and we show how we assessed the degree and impact of manually performed class relocation and reconceptualization.

A. Design of the Study

In order to evaluate the usage of relocation and reconceptualization at the granularity of classes during manual modularization of NDVis, we investigate three properties of the performed restructurings.

Firstly, we compare the absolute impact of the manual relocation and reconceptualization of classes on scattering and tangling of features to the results of automated relocation of classes. By doing so, we assess whether the manual process can be improved by adopting the automated approach.

Secondly, we isolate and assess the absolute impact of the reconceptualization of classes performed during the manual restructuring. Thereby, we compare the effects of manual reconceptualization to the effects automated relocation.

Thirdly, we identify the relative degree of the manually performed reconceptualization by placing it on a scale ranging from none to maximum reconceptualization of classes. By doing so, we identify the degree to which reconceptualization potential has been exploited during the manual restructuring.

We formulate these investigations as three research questions:

Q1: What is the absolute impact of the manually performed partial relocation and reconceptualization on the scattering and tangling of features, as compared to that of sole maximum relocation?

Q2: What is the absolute impact of the manually performed reconceptualization on the scattering and tangling of features?

Q3: What is the relative degree of the manually performed reconceptualization?
In order to address these research questions, we use five alternative designs of NDVis, exhibiting different degrees of relocation and reconceptualization at the granularity of classes. These designs, as well as their associations with the research questions, are summarized in Table I.

### Table I. Investigated Designs of NDVis

<table>
<thead>
<tr>
<th>Design</th>
<th>Extent of Applied Refactoring</th>
<th>Related Question</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Class Relocation</td>
<td>Class Reconceptualization</td>
</tr>
<tr>
<td>Original</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Manual</td>
<td>Partial</td>
<td>Partial</td>
</tr>
<tr>
<td>Automatic</td>
<td>Maximum</td>
<td>None</td>
</tr>
<tr>
<td>Manual+Automatic</td>
<td>Maximum</td>
<td>Partial</td>
</tr>
<tr>
<td>Automatic split</td>
<td>Maximum</td>
<td>Maximum</td>
</tr>
</tbody>
</table>

The original design is the legacy monolithic design of NDVis based on the MVC pattern, which we use as a frame of reference for the other designs. As we have made no modifications to this legacy design, we say that it exhibits no class relocation and reconceptualization.

The manual design is the result of our efforts to migrate features of NDVis to the NetBeans Module System. Since class relocation and reconceptualization was used here for the purpose of feature-oriented migration, the achieved extents of these restructurings reflect the actual extent sufficient for that very purpose. We report on this restructuring in Section V.

The automatic design was obtained using an approach that automatically optimized the original design according to scattering, tangling, cohesion and coupling using only relocation of classes among packages. Such optimized relocation will be referred to as maximum relocation. We describe construction of this design in Section VI.

The manual+automatic is the design created by applying the automated class-relocation-based approach on the manual design. Thereby, the resulting design is made to exhibit maximum degree of relocation and the degree of reconceptualization of the manual design. This design is described in Section VII.

Lastly, the automatic split design is the design in which we use the manual design as an input to simulating the maximum possible degrees of reconceptualization and relocation of classes. We discuss how this is done in Section VIII.

Once we discuss construction of these designs, we will use Section IX for presenting the performed measurements and answering our research questions.

### B. Establishing Traceability Links

As a prerequisite to restructuring and comparing alternative designs of NDVis, we have established traceability links between features and source code of the program.

Firstly, we have recovered a set of feature specifications of NDVis. This was done by inspecting the functionality provided in the GUI of the program and interviewing the lead developer of the project. In Table II, we list the names and summarize the descriptions of all the identified features. Since we were dealing with migration of a monolithic application to a module system, we found it unnecessary to create a full feature model, which is required for creating software product lines [15].

### Table II. Features of NDVis

<table>
<thead>
<tr>
<th>Feature</th>
<th>Summary description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjust image</td>
<td>Zooming, panning, enlarging, and resetting image.</td>
</tr>
<tr>
<td>Color mapper</td>
<td>Mapping data to colors on the image using SQL queries, managing, editing and bookmarking queries.</td>
</tr>
<tr>
<td>Database connectivity</td>
<td>Configuring and connecting to a database.</td>
</tr>
<tr>
<td>Image</td>
<td>Creating and rendering an image-based representation of data, hover-sensitive preview of data values.</td>
</tr>
<tr>
<td>Import data</td>
<td>Importing data from a CSV file.</td>
</tr>
<tr>
<td>Optimize</td>
<td>Optimizing how multiple data parameters are being displayed as two-dimensional images.</td>
</tr>
<tr>
<td>Parameters</td>
<td>Displaying and manually reordering data parameters.</td>
</tr>
<tr>
<td>Program startup</td>
<td>Initializing the program.</td>
</tr>
<tr>
<td>Save image</td>
<td>Persisting created images.</td>
</tr>
<tr>
<td>Simulator</td>
<td>Simulating the behavior of neurons based on data.</td>
</tr>
</tbody>
</table>

Secondly, we have used the recovered names of the features to apply our dynamic feature-location mechanism described in [16]. This approach to feature location requires annotating so-called feature-entry points in the source code of an investigated program in order to guide dynamic analysis. Feature-entry points are the methods through which the execution of a program enters the implementations of features. In the case of NDVis, in which features are triggered through GUI elements, feature-entry points were most often associated with the actionPerformed methods of event-handling classes. Based on such annotations, which we parameterize with textual identifiers of features, we traced the activations of individual features during a program’s execution. Our tracing agent registered packages, classes, objects and methods involved in execution of individual features and then saved this information as a set of feature-trace files.

The hereby established traceability links are used throughout the remainder of this paper.

### V. “Manual” Design

In this section, we report on usage of a mixture of relocation and reconceptualization of classes during a manual feature-oriented migration of NDVis to the NetBeans Module System. Using three features as examples, we discuss the most important challenges, design decisions and feature-oriented analysis techniques involved.

Our plan for using remodularization to establish the properties desired for NDVis by its owners was based on two kinds of modules. We used independent feature modules to group code units specific to features, whereas we used core modules to group multi-feature code units that are to be reused by multiple features. The intent was to facilitate independent evolution and deployment of features by increased separation of features in terms of feature modules, while at the same time facilitating reusability of essential domain concepts and utilities by placing them in explicit core modules.

We adhered to the mechanisms offered by the NetBeans Module System and the standard specification of the Java...
language as the technical means of modularizing features. While existing mechanisms of advanced separation of concerns could enable additional separation of features, they were excluded due to additional learning curve and unconfirmed interoperability with the NetBeans Module System.

A. Initial Feature-Centric Analysis of the Original Design

Firstly, we performed a top-down investigation of how feature implementations fit into the original design of NDVis. Thereby, we aimed at estimating the extent of the forthcoming restructuring and at identifying potential hotspots exhibiting the most complex feature-code relations.

The analytical view used for this purpose, as well as the two following views in this section, were produced from the traceability links gathered earlier by the Featureous tool for feature-centric analysis [8]. This view, being presented in Fig.2, is a colored bar plot of scattering and tangling [11] of the features found in NDVis, shown in terms of packages. The coloring scheme assigns a gradient of colors between light green for single-feature classes towards dark blue for most tangled classes.

![Figure 2. Characterization of feature-package relations.](image)

By studying the information on feature scattering, we learned that an average feature of NDVis is using 6 out of 19 packages. From the scattering profiles we confirmed the lack of localization of single-feature classes in single-feature packages.

Through analyzing the two most tangled packages, namely .gui and .app, we discovered strong differences in the ways they were reused among features. While .gui consists of a set of simply moveable single-feature classes, .app contains a highly tangled class NDVis. Later on, this class turned out to be a major hot-spot that we needed to reconceptualize to remove dynamic dependencies among features (see Section V.C).

**Identifying core domain concepts as the most tangled classes.** By refining the granularity of the presented view to the level of classes, we identified the classes that participate in most of the features. We observed that these classes (apart from the NDVis class) constitute the implementations of the central domain concepts of the application. These classes were: ImagePanel, DataInfo, Parameters, ParametersUtils and ColorEngine. Early recognition of these concepts helped us to understand the essential domain model of NDVis and to establish a set of candidate classes for forming core modules (see Section V.D).

B. Modularizing Features by Relocation – “Optimize”

One of the features modularized by means of class relocation was “Optimize”. The “Optimize” feature is concerned with optimizing how multiple dimensions (Parameters) of data (DataInfo) are used to form a two-dimension representation of the dataset.

The way “Optimize” shares classes with two other features of interest is presented in Fig. 3. Here we can see that the implementation of “Optimize” is related to five feature-specific classes and a number of core classes identified earlier.

![Figure 3. Correlation of selected features and classes.](image)

Good separation of this feature at the granularity of classes made it possible to avoid extensive reconceptualization of existing classes. This feature was modularized by mostly relocating its feature-specific classes (green classes in the first column in the figure) to its dedicated feature module. The remaining highly tangled classes of “Optimize” were used later on for forming the core modules.

**Identifying classes exhibiting under-reuse.** Interestingly, we have observed that one of the feature-specific classes of “Optimize”, namely PermutorCombinator, was originally placed in the .utils package of NDVis. This indicates that this class was originally intended to be reused as a utility class among multiple features. However, as we can see, this class has not been reused by features other than “Optimize” and hence ended up being inappropriately placed in the original package structure of NDVis.

**Identifying classes exhibiting over-reuse.** A situation may occur, where there exist static dependencies in terms of code reuse between seemingly unrelated features. Such dependencies are excessive, as they do not correspond to logical relations among features in an application’s problem domain. Hence, we ensured that the resulting “Optimize” module neither exposes any API classes nor depends on any other feature modules. As a consequence, it became possible to transparently remove or add “Optimize” to the application simply by (un)installing its dedicated module.

C. Decentralizing Initialization – “Program Startup”

“Program startup” was the feature, or rather the technical concern, responsible for initializing the whole application. We
found this centralized initialization model to be a major obstacle to run-time independence of features.

The dynamic role of “Program startup” is well depicted in Fig. 4, which presents a graph of object-wise producer-consumer dependencies among features. As shown, all other features use objects instantiated by “Program startup” feature.

**Figure 4.** Dynamic dependencies between features.

In order to understand how “Program startup” initialized features in the original design of NDVis, we have investigated the central hot spot of its implementation being the NDVis class. We learned that the NDVis class was a god class not only instantiating, but also composing together essential parts of implementations of most features. In the context of software evolution, this meant that the NDVis class had to be modified every time a feature was added or removed from the application. This hindered functional extensibility and customizability of the application.

To address this issue, we have reconceptualized the NDVis class to redistribute the initializations of individual features among the features themselves. Thereby, we moved away from the centralized model of feature initialization and composition towards a decentralized model. In this new model, pluggability of features is facilitated by making each feature initialize and integrate itself with the rest of the application.

As a tangible outcome of the extensive refactorings performed, the NDVis class was reduced by 330 lines of code (out of 490 initial lines). The remains of “Program startup” were placed in one of the application’s core modules.

**D. Consolidating the Reusable Core – “Parameters”**

The goal for creating an application core was to establish an explicit bundle of reusable code that would form a common base for all feature implementations. In principle, such a core should capture all the classes implementing important domain concepts that features commonly operate on.

“Parameters”, being one of the essential features of NDVis, provides the concept of dimensional parameters (Parameters) of a data and handles their table-based visualization in the GUI.

In order to modularize “Parameters” we performed a series of extensive reconceptualizations on the NDVis class in connection with the dismantling of the “Program startup” feature. This allowed us to isolate classes belonging to the “Parameters” feature (these included: Parameters, ParametersController and ParametersUtils). We have relocated these classes in a dedicated module for “Parameters”. However, the module of “Parameters” was decided to be made a mandatory core module of NDVis due to the mandatory nature of its functionality and due to incoming dependencies from multiple feature modules.

In order to facilitate the reusability of the “Parameters” module, we have ensured that this module does not depend on any non-core modules. Lack of such a dependency is furthermore required by the NetBeans Module System, which prohibits circular dependencies between modules. Given that all feature modules were already depending on core modules, including “Parameters”, no opposite dependencies were permitted. In the context of software evolution, independence of an application’s core from user features reduces inbound ripple effects of feature-oriented changes and facilitates reuse of core modules independently from feature modules.

**E. Resulting Design**

As a result of iterative modularization of features into independent feature modules and consolidation of reusable multi-feature core modules, we have arrived at a new package structure of NDVis, divided into modules shown in Fig. 5. Achieving this result required 35 man-hours of work and 20 intermediate commits.

**Figure 5.** Division of NDVis into explicit modules.

Each of the top feature modules implements a single feature and depends only on core modules. By employing the service discovery mechanism of the module system, we made NDVis customizable by enabling exclusion of any feature module without changing a single line of code in the application. Moreover, the established set of core modules can be easily reused across a larger portfolio of applications, since it does not depend on any feature modules. In Section IX, we quantify the package-level localization and separation of features exhibited by these new modules of NDVis.

**VI. “AUTOMATIC” DESIGN**

To assess the impact of the manual restructuring on the confinement of features, we restructured the original design of NDVis by optimizing feature confinement using only relocation of classes.

The used approach, introduced and discussed in full detail in [16], is based on multi-objective optimization of a set of package-granularity design criteria by means of class relocation. This is done by encoding assignments of classes to packages as chromosomes in a multi-objective genetic algorithm that optimizes the population of chromosomes according to a set of objectives expressed as metrics.
The objectives we use to determine the optimal relocation of class are to minimize scattering and tangling of features in packages. In addition, maximization of package cohesion and minimization of inter-package coupling is used to ensure that new packages created by relocation possess these qualities and to ensure that any classes not covered by feature traces will be relocated to the packages that are most strongly related to it. By using these four objectives, we aim at optimizing the design of NDVis to localize feature implementations and separate them from each other, while keeping the resulting packages cohesive and loosely coupled.

The precise formulations of the four objectives we use (i.e. total scattering of features among packages, total tangling of features in packages, average cohesion of packages and total coupling of packages) are listed in Fig. 6.

\[
\text{tfsca}(P) = \sum_{f \in P} \text{fsc}(f), \text{ where: } \text{fsc}(f) = \left[ \frac{|\{ p \in P : f \Rightarrow p \}| - 1}{|P|} \right]
\]

\[
\text{ttang}(P) = \sum_{f \in P} \text{ttang}(p), \text{ where: } \text{ttang}(p) = \left[ \frac{|\{ f \in F : f \Rightarrow p \}| - 1}{|F|} \right]
\]

\[
\text{apcoh}(P) = \frac{\sum_{f \in P} \text{coph}(p, T)}{|P|},
\text{ where: } \text{coph}(p, T) = \sum_{I \in mp} \sum_{I \in mp} |D_{I1_{II2}} \cup DM_{I1_{II2}}|
\]

\[
\text{tpcou}(P) = \sum_{f \in P} \text{pcou}(p, T),
\text{ where: } \text{pcou}(p, T) = \sum_{I \in mp} \sum_{I \in mp} |D_{I1_{II2}} \cup DM_{I1_{II2}}|
\]

Figure 6. Multi-objective formulation of feature-oriented remodularization.

Based on the formulation proposed in [11], total scattering of features among packages (tfsca) measures the average number of packages \( P \) that contribute to implementing program features \( F \) (i.e. packages that fulfill the \( \Rightarrow \) “implemented by” relation with features). This value is furthermore normalized according to the number of packages found in the program. The value of tfsca needs to be minimized to reduce delocalization of feature implementations.

Based on the formulation proposed in [11], total tangling of features in packages (ttang) is a measure complementary to tfsca, as it quantifies the average number of features \( F \) that use packages \( P \). Our approach minimizes the value of ttang to reduce interleaving of features in packages.

The definitions of average package cohesion (apcoh) and total coupling of packages (tpcou) are based on cohesion and coupling measures proposed in [23] and [18]. They are based on the notions of interactions between data declarations (DD-interactions) and interactions between data declarations and methods (DM-interactions) and the \( \Rightarrow \) operator for specifying the containment relations between classes and packages. Our formulation of package coupling corresponds to a sum of the ACAIC, OCAIC, ACMIC, and OCMIC coupling measures defined in [18], whereas our formulation of cohesion is the package-level version of the RCI metric proposed in [23].

The actual calculation of the two feature-oriented metrics is done based on the feature-code traceability links established earlier. The metrics of cohesion and coupling are calculated on a dependency model representing static relations between classes, which we extract using static analysis of source code.

We have applied this approach to automatically optimize the original design of NDVis. We obtained the new design (automatic) by repeated execution of the genetic algorithm on a population of 300 alternative randomized designs (genes) for 500 epochs with 5% mutation probability. We use the best structuring obtained as the maximum optimization of the original design achievable by means of sole class relocation, without using any reconceptualization.

VII. “MANUAL+AUTOMATIC” DESIGN

To isolate and assess the impact of the manually performed class reconceptualization on scattering and tangling of features, we applied the automatic class relocation approach discussed previously on the manual design of NDVis.

The resulting design optimizes the relocation of classes in the manual design, thereby achieving maximum relocation, while preserving the level of class reconceptualization achieved originally by manual restructuring.

VIII. “AUTOMATIC SPLIT” DESIGN

To determine the relative degree of the manually performed class reconceptualization, we need to establish a scale consisting of a design exhibiting no class reconceptualization with maximum relocation and a design exhibiting maximum possible class reconceptualization with maximum relocation. The former of these is the automated design of NDVis discussed in Section VI. The latter we create by simulating maximum reconceptualization of classes and applying the automated relocation approach.

We simulate maximum class reconceptualization of NDVis by applying the following procedure to its manual design. For each multi-feature class, we identify all its single-feature methods and we remove the traceability links between these methods and their corresponding features. From the point of view of the metrics that we use for evaluation in Section IX, doing so corresponds to the effects of the move method refactoring that would be used by a programmer to relocate such a method to another class related to its corresponding feature. In the case of multi-feature classes whose methods are used disjointly by multiple features, applying this approach simulates the process of cleanly splitting the class into multiple single-feature classes, which are then relocated to their respective single-feature packages.

By applying this approach to the manual design of NDVis and then applying the automated maximum relocation approach, we obtained a design exhibiting maximum degrees of both class relocation and reconceptualization. By applying the described simulated reconceptualization, we managed to completely detach a feature from a whole class only in 6 cases, which suggests a relatively low potential of NDVis for reconceptualization at the granularity of classes.

IX. RESULTS

In order to address the research questions formulated in Section IV, we have measured and compared the feature-
oriented quality of the five discussed designs of NDVis. This was done using the metrics of scattering (fsca) and tangling (ftang) introduced in Fig. 6 in Section VI.

In addition to taking the presented measurements, we measured the values of coupling and cohesion and used them as sanity checks for the obtained designs. By doing so, we confirmed that the produced designs do not exhibit extremely high coupling or low cohesion values and thereby follow these elementary design practices. In fact, we have observed the generated designs to slightly improve these properties in comparison to original and manual designs of NDVis.

A summary of the measurements of scattering and tangling is shown in Fig. 7, whereas their detailed distributions are shown in Fig. 8. Based on this, we address our research questions.

Figure 7. Impact of investigated designs on feature tangling and scattering.

Q1: What is the absolute impact of the manually performed partial relocation and reconceptualization on the scattering and tangling of features, as compared to that of sole maximum relocation?

By comparing the original design of NDVis with the manual design and the automatic design, we can see that both of the restructuring reduced scattering and tangling of features. However, the design based purely on automatic relocation of classes achieved significantly bigger improvements (49% and 64% respectively) than the design based on manual relocation and reconceptualization (16% and 27% respectively).

We observed this difference to be caused by effort-intensity and substantial cognitive complexity of the involved refactorings. This suggests that incorporating the automatic approach as a part of the manual restructuring would not only significantly improve the results but also reduce the amount of manual work involved. Thereby, manual work could be limited to only reconceptualizing classes and to ensuring that the program is configured to run in the module system.

Q2: What is the absolute impact of the manually performed reconceptualization on the scattering and tangling of features?

To assess the impact of the manually performed reconceptualization on scattering and tangling of features, we compare the automatic design based on optimized class relocation with the manual+automatic design based on manual reconceptualization and optimized relocation. As we can see from the obtained results, the additional reductions of scattering and tangling introduced by manual reconceptualization surpass the results of pure relocation by only 7% and 14% respectively.

This result suggests that the reconceptualization performed during the manual restructuring had only a minor effect on reducing scattering and tangling of features. However, as shown in the scattering plot in Fig. 8, the performed reconceptualization was highly efficient in modularizing selected features, such as the “Program startup” feature, which became nearly completely localized. As can be further seen in Fig. 8, this was not achievable by means of sole automated relocation of classes.

Q3: What is the relative degree of the manually performed reconceptualization?

To establish a scale against which the degree of reconceptualization performed in the manual+automatic design can be assessed, we use the automatic design exhibiting no reconceptualization and the automatic split design exhibiting maximum reconceptualization of classes. By doing so, we can see that the manual reconceptualization of the manual+automatic design exploits most of the achievable potential for reconceptualization of classes.

This indicates that migration to a module system required a high relative degree of reconceptualization of classes. Furthermore, it can be seen that the maximum achievable improvement offered by reconceptualization remains significantly smaller than the one offered by automated relocation. A closer inspection revealed that this was caused by a relatively high tangling of methods in NDVis, which limited the effects of reconceptualization at the granularity of classes and created the need for further reconceptualization at the granularity of methods.

X. DISCUSSION AND THREATS TO VALIDITY

In the presented study, we have relied on several assumptions that could be refined in the future replications to potentially grant additional insights into our findings.

Firstly, one could focus on differentiating between and quantifying multiple kinds of class-level reconceptualization, such as move method, pull up method, push down method and move field refactorings [21]. While in the presented study we treated them uniformly, we consider it worthwhile to aim at refining the investigation in these directions.

Secondly, the impact of reconceptualization at lower granularities (i.e. methods and instructions) would be an interesting topic for investigation from the perspective of additional modularization of features, effort required, and impact on comprehension of source code.

Thirdly, the fact that the authors of this work performed the manual remodularization of NDVis has to be seen as a threat to internal validity of the presented results. However, we believe that this factor had a negligible impact on the results, since the
manual remodularization was originally performed as a part of separate work, months before the idea of using this data for investigating relocation and reconceptualization was born.

Finally, replicating our study on a larger population of programs is necessary for generalizing our findings. Hence, we consider it a contribution of this work to report the initial insights and to demonstrate how similar studies can be performed. We hope that additional studies will lead to evaluation of the generality of our findings and to investigate different types of remodularizations (e.g. migrating to a software product line).

XI. RELATED WORK

The approach that relates to our manual remodularization of NDVis is the work of Mehta and Heineman [1]. The authors presented a method for locating and refactoring features into fine-grained reusable components. Features are located by test-driven gathering of execution traces. Feature implementations are then analyzed, and refactored into components according to a proposed component model. The authors apply both relocation and reconceptualization of classes for separating stateful and stateless methods among components. However, details of using these refactorings are not reported.

Kästner et al. [22] investigated the issue of granularity of feature separation in Software Product Lines (SPLs). They compared their annotative approach based on C/C++-like preprocessor with AHEAD – a step-wise-refinement approach to composing template-encoded class and method refinements. The authors report on the drawbacks of the template-based composition, including limited granularity and the inability to extend to statement, expressions and method signatures. On the other hand, the annotative mechanisms were found to obfuscate the code and to lack modularity mechanisms.

Liu et al. [15] proposed the feature-oriented refactoring (FOR) approach to restructuring legacy programs to forming feature-oriented SPLs. The aim of this approach is achieving complete separation of feature implementations in the source code by means of advanced separation of concerns. This relies on creating class fragments through maximum manual reconceptualization and relocation at the granularities of classes, methods and instructions. This is done by creating base modules, which contain classes with so-called introductions, and sequentially-ordered derivative modules for features.

Lopez-Herrejon et al. [17] reported on their experience in manually refactoring features of two legacy application towards forming feature-oriented SPLs. The authors identify eight refactoring patterns that describe how to extract the elements of features into separate code units. The proposed patterns constitute relocation at granularities of instructions, methods and classes, and reconceptualization at granularities of methods, classes, and packages. Furthermore, the authors recognize the problem of feature tangling at the granularity of methods and use heuristic-based disambiguation to decide on a destination feature that such methods should be relocated to.

Murphy et al. [2] explored the tradeoffs between three policies of splitting tangled features implementations: a lightweight class-based mechanism, AspectJ, and Hyper/J. By manually separating a set of independent features at different levels of granularity, the authors confirm the limited potential for tangling reduction of the lightweight approach. In the case
of AspectJ and Hyper/J, they have discovered that usage of these makes certain code increments difficult to understand in isolation from one another. Furthermore, aspect-oriented techniques were found to be sensitive to the order of composition, which resulted in undesirable coupling of the implementations of features to each other.

XII. CONCLUSION

In this paper, we analyzed the degree and impact of class-granularity relocation and reconceptualization during a feature-oriented migration of NDVis to the NetBeans Module System.

We discussed the role of features during migration of monolithic applications to module systems and formalized the notions of relocation and reconceptualization. We used this conceptual framework as a basis for formulating three research questions. We have addressed these questions by designing and performing a study of five designs of NDVis exhibiting different degrees of relocation and reconceptualization.

Based on the collected measurements, we determined that the maximum possible impact of class relocation on scattering and tangling of features in NDVis is significantly higher than that of class reconceptualization. Nevertheless, we observed that the manual efforts were more efficient in resolving selected specific cases of feature scattering and tangling. Moreover, we assessed the degree of manual class relocation and reconceptualization performed during the actual remodularization of NDVis.

We believe that this novel evidence provides an important insight into the role of class relocation and class reconceptualization during migrations of legacy monolithic applications to module systems. Furthermore, we hope that the proposed evaluation procedure can be reused in other contexts (i.e. other applications, other types of migrations) to provide additional insights and generalize the presented findings.

The tool that we used for supporting manual modularization of features, measuring the results and generating the automated designs, as well as the results of the manual remodularization of NDVis can be found on their respective websites [7] and [6].

REFERENCES