Marea: a Semi-automatic Decision Support System for Breaking Dependency Cycles

Andrea Caracciolo
SCG, University of Bern
3012 Bern, Switzerland
http://scg.unibe.ch

Bledar Aga
Swiss Aviation Software
4002 Basel, Switzerland
http://www.swiss-as.com

Mircea Lungu
University of Groningen
9747 Groningen, The Netherlands
m.f.lungu@rug.nl

Oscar Nierstrasz
SCG, University of Bern
3012 Bern, Switzerland
http://scg.unibe.ch

***Abstract*—Dependency cycles are commonly recognized as one of the most critical quality anti-patterns. Cycles compromise the modularity of a system, prevent proper reuse and increase the cost of maintenance and testing. Many tools are capable of detecting and visualizing package cycles existing within software projects. Unfortunately, detecting cycles is only half of the work. Once detected, cycles need to be removed and this typically results in a complex process that is only partially supported by current tools. We propose a tool that offers an intelligent guidance mechanism to support developers in removing package cycles. Our tool, Marea, simulates different refactoring strategies and suggests the most cost-effective sequence of refactoring operations that will break the cycle. The optimal refactoring strategy is determined based on a custom profit function. Our approach has been validated on multiple projects and executes in linear time.**

I. Introduction

Dependency cycles are a typical symptom of bad design [1], [2], [3] and are often linked to architectural erosion [4] and defect-proneness [5]. Empirical studies show that cycles can be found in almost any medium to large object-oriented software system [6], [7]. Dependency cycles introduce deployment constraints, forcing developers to bundle packages that are logically uncoupled, and generally increase maintenance costs. Excessive coupling amongst packages reduces the overall modularity of the project, precluding the possibility to homogeneously distribute the development effort between the members of the team. Scarce modularity has also a negative impact on testability, since isolating the functionality of low granularity units becomes more complicated. Martin defines the Acyclic Design Principle [2] as one of the rules that govern the structure of object-oriented software systems.

Given the proven importance of this architectural anti-pattern, many tools have been developed to detect dependency cycles. Most of them are commercial tools (*e.g.*, Structure101[1], Lattix LDM[2], SonarGraph[3]) and are commonly used by industrial practitioners. The main functionality offered by most of these tools consists in presenting a rich visualization of the package cycles existing within a given project. Other tools (*i.e.*, JooJ [8]), prevent the introduction of new cycles by monitoring the development environment and offering real-time warnings.

One fundamental limitation of all the existing techniques is the absence of a convenient support for removing the detected cycles. Automatic fixes and refactoring suggestions are highly appreciated by developers [9], [10]. Based on our interaction with practitioners, we found that developers are often forced to undergo multiple stages in order to eliminate a cycle. Refactoring actions are repeatedly interleaved with reverse engineering steps, during which the user checks the impact of the applied modification. This can lead to a highly ineffective non-linear process that contributes to frustration and higher maintenance costs.

Some tools (*e.g.*, Pasta [11]) have tried to cope with this limitation by introducing support for simulating basic refactoring operations over a reverse engineered model of the analyzed project. Users can drag and drop code elements (*i.e.*, classes, methods) from one container (*i.e.*, package, class) to another and immediately see how this impacts package-level dependencies. Unfortunately, this kind of process is only a slight improvement over the previously described one. In fact, the user still needs to perform subjective choices with little guarantee that the outcome of his action will eventually lead to the complete removal of the cycle. In addition to that, the refactoring operations supported by these tools are very elementary. Other more sophisticated techniques (*i.e.*, dependency injection) often used in practice are simply ignored.

In this paper, we propose a decision support system for removing package dependencies. Our tool, Marea, computes multiple refactoring sequences and suggests the strategy that offers the best results (*i.e.*, low number of dependency cycles and high structural quality) at lowest cost (*i.e.*, low number of required refactoring steps).

In our current implementation, we support 4 types of refactoring (see subsection II-B). These refactoring strategies are applied to create mutations of a model originally extracted from the system's source code. By simulating these operations, we create a decision tree where each node (*i.e.*, refactoring step) has a weight calculated through a custom profit function. Our approach also offers a more complete and automated solution to cycle removal. The presented technique has been evaluated on two open source projects and exhibits linear scalability. It could be integrated into any existing cycle detection tool and would improve them by extending their diagnostic features with a more complete set of reactive quality improvement capabilities.

The paper is structured as follows: In section II, we define a common terminology and describe the refactoring operations that are used in our approach. We then introduce our approach (III), report on its evaluation (IV), and discuss its applicability (V). Finally, we conclude (sections VI-VII).

[1]https://structure101.com
[2]http://lattix.com
[3]https://www.hello2morrow.com/products/sonargraph

II. Basic Concepts

In this section we briefly introduce the main concepts characterizing the domain of application of Marea. In order to make the description as concrete as possible, we choose to restrict the scope of the discussion to systems developed in Java.

A. Terminology

Marea has been designed with the purpose of detecting and removing package-level dependency cycles. This form of cycle is detected by representing a system as a graph where the nodes are packages and the edges are the dependencies between them. Such a graph might contain strongly connected components (SCCs) that are composed by one or more cycles. A cycle is a closed walk with no repetitions of vertices and edges allowed.

If an entity x depends on another entity y, we write $x \rightharpoonup y$.

We categorize the dependencies in a system as follows:

- Class Dependency (CD): concrete dependency relating a class to another. $A \rightharpoonup B$, if the class A contains a reference to the class B.
- Package Dependency (PD): conceptual dependency between packages resulting from the aggregation of one or more CDs. In more concrete terms, suppose class A in package P_A depends on class B in package P_B. Then P_A depends on P_B, and the CD $A \rightharpoonup B$ *is contained in* the PD $P_A \rightharpoonup P_B$.
- Shared Package Dependency (SPD): package dependency that is present in more than one cycle.

Class dependencies are further classified into:

- Inheritance Dependency: $A \rightharpoonup B$, if A is a direct subclass of B or A implements B (in case B is an interface).
- Reference Dependency: $A \rightharpoonup B$, if
  - a class field of A is of type B (Class Field).
  - a class field of A is initialized with an object of type B (Initialized Class Field).
  - a variable defined in a method in A is of type B (Local Variable).
  - a variable defined in a method in A is initialized with an object of type B (Initialized Local Variable).
  - a parameter of a method in A is of type B (Parameter).
  - the return type of a method in A is of type B (Return Type).
- Invocation Dependency: $A \rightharpoonup B$, if a method in A invokes a method in B.

B. Refactoring Strategies

In our approach we use 4 distinct refactoring strategies. In the remainder of this section we describe the different strategies and their applicability constraints.

Move Class (MC)

This refactoring strategy moves a class from one package to another. This refactoring strategy has previously been used by Shah *et al.* [12] to untangle dependency cycles.

In Figure 1 (Before), the package *components* $\rightharpoonup$ *control* (we will ignore the specifics of this dependency in the interest of simplicity) and the class *Button* $\rightharpoonup$ *Light*, which induces the reverse dependency *control* $\rightharpoonup$ *components*. This cycle could be broken by simply moving *Button* to the package *components*. This refactoring strategy is very simple but may not be semantically consistent with the overall design of the system. In fact, in this case, the class *Button* should not be moved to *components*.

We will use the notation "MC: A to P" to describe the operation where class A is moved to package P.

Move Method (MM)

This refactoring is similar to Move Class. It has been previously investigated by Tsantalis *et al.* [13] as a means to remove Feature Envy bad smells.

Let's assume that the method *Button.press()* depends on the class *Light* (in Figure 1 (Before)). This invocation dependency could be removed by moving the method from its original class (*Button*) to the target of the dependency (*Light*).

We will use the notation "MM: M to A" to describe the operation where method M is moved to class A.

Abstract Server Pattern (ASP)

This refactoring, described by Martin [14], is inspired by the following principle: "Depend upon Abstractions. Do not depend upon concretions".

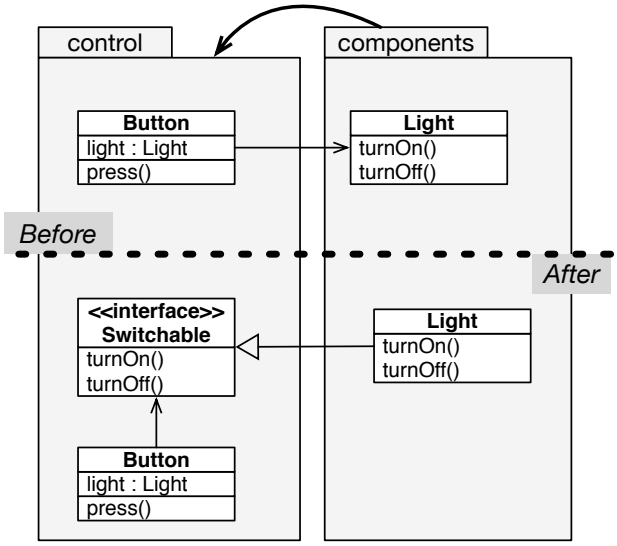


Fig. 1. The 'Abstract Server Pattern' refactoring strategy

The refactoring can be used to invert the direction of a dependency in case its target is a concrete class. In Figure 1, *Button* has a field of type *Light*. This dependency can be inverted by creating a new interface, *Switchable*, in the package containing the class from which the dependency is originated. This interface will then be implemented by the class on the other end of the dependency (*Light*). By applying this simple operation we inverted the dependency from *control* to *components*, and we eliminated the cycle.

We use the notation "ASP: *SourceElement* for type *TargetClass*" to describe an instance of this refactoring (in our example, we would use: ASP: control.Button.light for type

components.Light).

Abstract Server Pattern + Dependency Injection (ASP+DI)

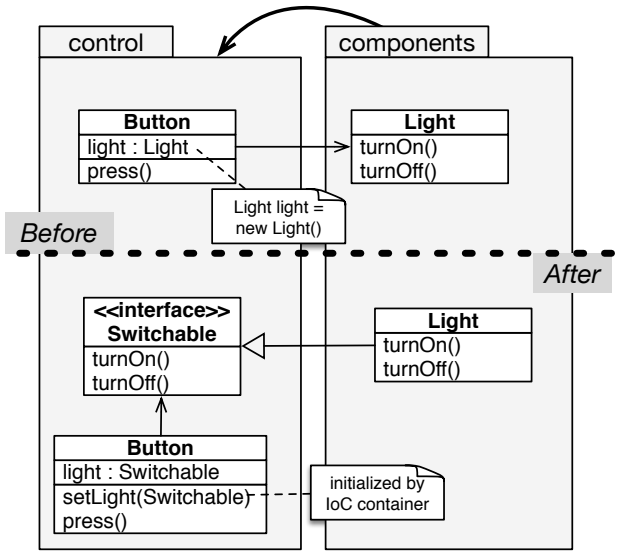


Fig. 2. The 'Abstract Server Pattern + Dependency Injection' refactoring strategy

This refactoring[4] is an extension of the previously introduced ASP refactoring. We use dependency injection to eliminate initialization code responsible for dependencies. In our example (Figure 2), we first apply the ASP refactoring. Since Button.light is originally initialized using the default constructor of Light, we remove the assignment in the field declaration (Light light = new Light()) and declare a new method (setLight()). This method will eventually be invoked by an Inversion of Control (IoC) container (*e.g.*, Spring[5]) when the object is created. The actual value used for the initialization could be defined in a configuration file and the field might be annotated with a framework-specific annotation.

Using this technique has a positive impact on maintainability, as dependencies can be easily changed to adapt to new requirements or a different runtime environment. On the other side, the code will contain implicit references and will require some sort of configuration logic in order to function properly. Fortunately, many applications (*e.g.*, J2EE, Spring) already rely on a dependency injection framework and therefore would not be required to introduce drastic architectural changes in their system.

Since ASP+DI is essentially a more refined version of ASP, this refactoring can be used to handle all the cases that are handled by its simpler counterpart. Nevertheless, every refactoring technique presented in this section has a different cost (*i.e.*, implementation time) and semantic validity associated to them. The choice between one refactoring or another cannot be automatically determined and must be taken by the user. In our approach we simply test the applicability of each refactoring in the context of the analyzed system.

We will use the notation "ASP+DI: *SourceElement* for type *Class*" to denote an instance of this refactoring.

[4]http://www.martinfowler.com/articles/injection.html

[5]http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html

C. Strategy Applicability

Each refactoring strategy can be applied to break a variable number of class dependencies. We provide a exhaustive overview showing the applicability of the various strategies for each type of dependency (See Table I).

dependency type	MC	MM	ASP	ASP + DI
Inheritance	✓	✗	✗	✗
Class Field	✓	✗	✓	✓
Init. Class Field	✓	✗	✗	✓
Local Variable	✓	✓(*)	✓	✓
Init. Local Variable	✓	✓(*)	✗	✓
Parameter	✓	✓(*)	✓	✓
Return Type	✓	✓(*)	✓	✓
Invocation	✓	✓(*)	✗	✓(*)

TABLE I. APPLICABILITY OF THE REFACTORING STRATEGIES (ASTERISK STANDS FOR LIMITED APPLICABILITY)

The MM refactoring strategy is not applicable in case the dependency is caused by an inheritance relationship (*e.g.*, *Button* extends *Light*) or a class field dependency (*e.g.*, *Button.light* is of type *Light*). In those cases the refactoring cannot be applied as no method is involved in the dependency.

The ASP refactoring strategy cannot be used in case the dependency is caused by an initialized class or local variable (*e.g.*, *Button.light* is initialized to a concrete instance of *Light*). In those cases, one should opt for the ASP+DI refactoring.

The ASP+DI refactoring cannot be used in presence of an inheritance dependency (*e.g.*, if *Button* is a subclass of *Light*). In this case, the refactoring operation would break desired properties deriving from the dependency (*e.g.*, behavior reuse).

Special conditions apply in the circumstances marked with an asterisk (Table I). MM cannot be applied to remove the indicated dependencies if the method presents one of the following properties:

- The method is a constructor.
- The method returns *this*.
- The method accesses a variable with class-scope.
- The method has an invocation to a static method defined in the same class.

Furthermore, ASP+DI cannot be used if the dependency is caused by an invocation and the invoked method is static, a constructor, or *super()*, as none of these can be defined in an interface.

III. OUR SOLUTION

Marea executes in three phases (see Figure 3): (A) Initially it analyzes the input system to detect cycles; (B) then it explores all possible refactoring sequences; (C) and finally it suggest the most cost-effective refactoring sequence to the user. In the remainder of this section we describe each phase in more detail. The prototype is available for download on the web[6].

A. Analyze cycles

Detect cycles – In order to start the process we have to identify the dependencies in the target system. We do so by

[6]http://smalltalkhub.com/#!/~caracciolo/Marea/

running a fact extractor[7] based on Eclipse JDT[8]. The extractor visits all the nodes of the AST reconstructed by the underlying Eclipse platform and uses that information to build a FAMIX [15] model. This model will later be imported in Moose [16], an analysis platform designed to simplify the task of querying and manipulating FAMIX models. Once imported, the model is ready to be analyzed. To detect the cycles, we define a graph G = (V,E), where V are all the classes contained in the obtained model and E are the package dependencies (as defined in subsection II-A) existing among them. We use Tarjan's algorithm [17] to detect the strongly connected components (SCC) in the graph. Each SCC is subsequently untangled into individual cycles.

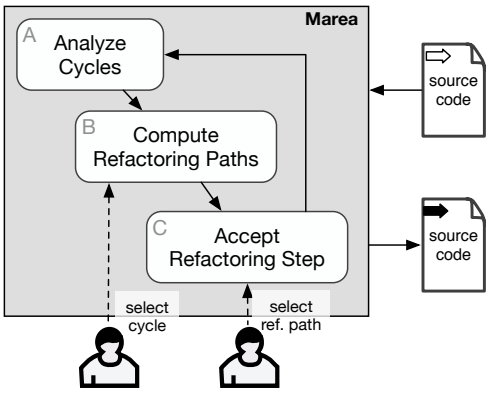


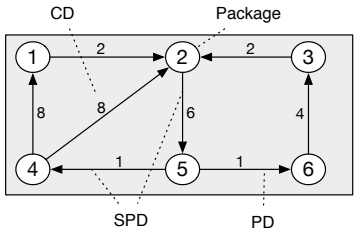
Fig. 3. The three main process phases of our solution

This phase can be executed in linear time (*i.e.*, the computational complexity of Tarjan's algorithm is $O(|V| + |E|)$).

Rank cycles – A typical project may contain a high number of cycles. To help the user to optimize the refactoring effort, we rank the cycles based on their overlapping degree and their size. This means that smaller cycles containing dependencies that are shared among more cycles will be presented before larger ones with fewer shared dependencies. To define our ranking, we sort all the cycles contained in the system (in descending order) based on the following formula:

$$rank(cycle) = \frac{|SPD_{cycle}|}{|CD_{cycle}| * |PD_{cycle}|} \quad (1)$$

where CD_{cycle}, PD_{cycle}, SPD_{cycle} are the sets of all the CDs, PDs and SPDs contained in *cycle*.



c1= {1,2,5,4}
rank(c1) = 2 / (17*4) = **0.03**

c2= {2,5,4}
rank(c2) = 2 / (15*3) = **0.04**

c3= {2,5,6,3}
rank(c3) = 1 / (13*4) = **0.02**

Fig. 4. Example: ranking of 3 cycles

[7]https://gforge.inria.fr/projects/verveinej/
[8]http://www.eclipse.org/jdt/

In the example illustrated in Figure 4, we have three cycles. Cycle 2 ranks best because it has two SPDs and a relatively low number of PDs and CDs.

Rank package dependencies – Cycles are formed by at least two package dependencies (PD). PDs are aggregations of multiple class dependencies and can be sorted based on the following function:

$$rank(pd) = \frac{|\ \{cycle\ |\ pd \in SPD_{cycle}\}\ |}{|CD_{pd}|} \quad (2)$$

where CD_{pd} is the set of all the CDs contained in the package dependency pd and SPD_{cycle} are the sets of all the SPDs contained a *cycle*.

This ranking function favors package dependencies that are shared among many cycles and are composed by a low number of class dependencies. These dependencies have the highest probability to break multiple cycles at the lowest cost (*i.e.*, refactoring steps are usually proportional to the number of CDs). An alternative method for ranking package cycles is discussed in a recent work by Falleri *et al.* [18]. In their approach, they assigning a higher rank to cycles that involve packages that are more distant in the package containment tree (*e.g.*, rank({x, x.y.z}) > rank({x.y, x.y.z}); where x, x.y, x.y.z are the packages forming the ranked cycles). They also assume that cycles with PDs with a lower number of CDs are more likely to have been introduced accidentally and therefore should be ranked higher. The last assumption is also reflected in our function.

B. Compute Refactoring Paths

User Action: Select cycle – When the analysis of the target system is complete, the user is asked to select the cycle she wants to remove. Her choice may be strongly influenced by her expertise, past experience and priorities. For example, a cycle that involves a core module is most probably more important than one between utility packages.

Simulate refactoring paths – To find the best refactoring sequence, we build a decision tree where each node represents a mutation of the model originally extracted from the system (Phase A). Each model variation is actually computed by cloning an existing model and applying the modifications prescribed by one of the supported refactoring strategies. The memory footprint of every new model is limited by the fact that only the changes are saved (and not a full new model).

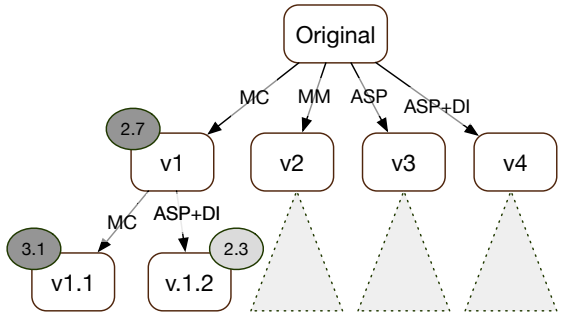


Fig. 5. Decision tree representing possible refactoring sequences

The example in Figure 5 illustrates a situation in which the user selected a cycle in which the top ranked package

dependency contains a class dependency that can be broken using all four refactoring strategies[9]. Since all refactorings are applicable, we create 4 variations of the original model (v1-4). For each new model we search again for cycles and recompute the dependencies that still need to be removed in order to eliminate the cycle. In the model v1, we might have to remove a dependency of type *initialized class field*. Since this dependency can only be removed using MC or ASP+DI, we obtain 2 new variations of v1 (v1.1-2).

The depth of the tree will grow depending on the level of accuracy that needs to be reached. Sometimes class dependencies might be impossible to break (*e.g.*, the suggested refactorings are semantically inconsistent with architectural design choices). In this case more elaborated design changes are required. In other cases, the removal of a class dependency may introduce new cycles in the system, causing an overall negative effect on the quality of the system. If this happens, the resulting refactoring sequence should be applied with caution or avoided.

The construction of a tree terminates when the cycle at hand has been completely removed. This means that every path connecting the root node to any leaf represents a complete refactoring sequence that leads to the elimination of the given cycle. The termination of the tree construction process is guaranteed to terminate, because at least one refactoring strategy (*i.e.*, MC) can always be applied. The process can be prematurely terminated for performance reasons by defining a maximal tree depth.

Suggest optimal path – To distinguish between effective and potentially harmful refactoring operations, we define a profit function that summarizes the gain obtained by applying that operation. The function should favor measurable quality improvements and should penalize high-effort refactoring operations. The function is defined as follows :

$$P = \frac{w_0}{cycles+1} + \frac{w_1}{depth} + w_2\frac{(1-I_{from})+(1-I_{to})}{2} + w_3\frac{A_{from}+A_{to}}{2} \quad (3)$$

where:

- *cycles* is the total number of cycles in the system.
- *depth* is the depth of the node within the decision tree.
- $I_{from/to}$ quantifies the instability [14] of the package from/to which the dependency is directed. This metric is an indicator of the package's resilience to change. The value can range from 0 to 1, where 0 indicates a completely stable package.
- $A_{from/to}$ quantifies the abstractness [14] of the package from/to which the dependency is directed. This metric indicates the percentage of abstract classes contained in the given package. The range for this metric is 0 to 1, with 1 indicating a completely abstract package. According to the Stable Abstractions Principle[10], "packages that are maximally stable should be maximally abstract".
- w_i are constants that can be tuned to assign less or more weight to the single components of the equation. In our experiments, we chose to have all weights set to 1.

The profit function was designed to guide the user towards the best result (*i.e.*, low number of dependency cycles and high structural quality) by minimizing operational costs (*i.e.*, low number of required refactoring steps). The profit value of a refactoring sequence is inversely proportional to the number of cycles existing in the system and the number of operations composing the given sequence. Improvements in structural quality (decrease of instability or increase in abstractness) contribute to increase the profit. Weights (*i.e.*, w_i) have been introduced to accommodate project/organization specific customization. These parameters could be tuned manually or automatically (based on previous interaction sessions).

This profit function is used in our approach to calculate the utility value of the single nodes composing our decision tree. As the tree is completely constructed, we will review all the leaf nodes and select the one with maximum profit. The refactoring sequence corresponding to the path connecting the root of the tree to this node is the sequence that will (according to our profit function) remove the cycle at the lowest cost. In our example (Figure 5), the best refactoring sequence is Original $\rightarrow$ v1 $\rightarrow$ v1.1. This refactoring sequence has a profit value of 3.1. In this example we assume that all the sub-trees below v2-4 lead to lower profit leaf nodes.

C. Accept Refactoring Path

User Action: Select refactoring path – The optimal refactoring sequence might be composed by a large number of refactoring steps. Some of these steps might be applicable to the system under analysis while others might be semantically inconsistent with the overall design of the project. To opportunely guide the user in the process, we present her only with the refactoring actions that compose the optimal path. Other sub-optimal refactoring sequences can be examined on request.

Update dependency graph – Once the user has chosen to apply a specific refactoring sequence, the process updates its internal model and starts again from the beginning (*i.e.*, phase A). All the selected refactoring operations are only simulated and not applied to the code. Refactoring sequences could be exported in a textual format and serve as an input for another tool that will perform the actual modifications to the source code.

IV. Evaluation

To test the applicability of our approach, we used our proof-of-concept prototype to analyze two projects (one open-source and the other commercial). In this section we describe the outcome of our experiments.

A. JHotDraw

JHotDraw[11] is an open-source project developed by Gamma *et al.*, often chosen as a reference object-oriented system for its sound design and rich use of design patterns. We analyzed version 6.0 beta 1, consisting of 485 Java files and a

[9]This might be the case when the class dependency is of the type: *local variable*; *parameter*; *return type*; or *invocation*. See Table I

[10]http://www.objectmentor.com/resources/articles/stability.pdf

[11]http://www.jhotdraw.org

total of 28,000 non-comment lines of code. During our analysis we detected 44 package cycles. Six of them could probably be considering irrelevant, as they only involve packages belonging to the test modules (*i.e.*, org.jhotdraw.test.**). The remaining ones are more likely to be considered harmful and often involve packages (*i.e.*, org.jhotdraw.contrib, org.jhotdraw.util) that, at first sight, should not belong to cycles. A subset of cycles encountered in our analysis is presented in Table II.

To test our prototype we followed the steps described in section III. Since we could not find an expert user who could help us discarding sub-optimal refactoring options, we blindly followed the suggestions offered by the tool without considering the semantic implications that this could have on the overall design of the system. As a result, we managed to discover a refactoring that allowed us to break all 44 cycles in 7 refactoring steps. This result was achieved through the following steps (described in detail in Table II):

Cycle #1 ({*contrib, samples.javadraw, contrib.zoom*}[12]): This cycle was removed by inverting the dependency *contrib* → *samples.jhotdraw*. This dependency has been selected as best candidate for removal because of its relatively smaller number of comprising class dependencies (2 compared to the 18 dependencies existing in the opposite direction) and its high number of dependency sharing (14 other cycles share the same dependency). The cycle was removed by moving the class *SVGDrawApp* from *contrib* to *samples.jhotdraw*. This was not only the optimal refactoring solution, but also the only one applicable. In fact, the class *SVGDrawApp* inherits from *org.jhotdraw.samples.javadraw.JavaDrawApp*. Therefore the only applicable refactoring strategy is Move Class (see Table I). Moving the class to *samples.jhotdraw* automatically removes the second dependency, consisting of an invocation to the parent constructor (using the super construct). After this step, the system still contained 26 unresolved cycles.

cycle #2-5 (2: {*contrib, contrib.zoom*}, 3: {*test, test.samples.pert*}, 4: {*standard, contrib, samples.javadraw, framework*}, 5: {*standard, contrib.dnd*}): As the analysis continues, more complex cycles are analyzed. Cycles 2, 3, 5 offer two refactoring options while cycle 4 offers only one. In all four cases, the MC refactoring appears to be the more convenient option (according to our profit function Equation 3). Proposed alternatives feature a profit score that is very similar to the chosen optimal counterpart. This means that, according to our profit function, all non-chosen refactorings would have been good candidates for the refactoring. In cycle #2, the second best refactoring option suggested by our tool was ASP+DI on *contrib.CustomSelectionTool.showPopupMenu([..])* for type *contrib.zoom.ZoomDrawingView*. This option scored 1.35 profit points, compared to the 1.36 of the chosen refactoring strategy. Also in cycle #3, the alternative solution (*i.e.*, MM test.AllTests.suite() to test.samples.pert.AllTests) had a similar score compared to the optimal solution (1.44 compared to 1.52). After this step, the system still contained 3 unresolved cycles.

cycle #6 ({*framework, util*}): The weaker package dependency in this cycle consists of nine class dependencies. For performance reasons, we decided to limit the depth of the simulation tree to a maximum of three. The suggested refactoring reported in Table II was obtained by combining the optimal paths computed during three subsequent simulation phases. During each simulation step, the tool evaluated an average of almost 30 different scenarios. The whole process had to be split into three phases because of the considerable memory requirements required to compute the individual simulation trees.

Cycle	Refactoring Sequence
1	MC: *contrib.SVGDrawApp* to *samples.javadraw*
2	MC: *contrib.CustomSelectionTool* to *contrib.zoom*
3	MC: *test.AllTests* to *test.samples.pert*
4	MC: *standard.StandardDrawingView* to *contrib*
5	MC: *contrib.dnd.DragNDropTool* to *standard*
6	MC: *framework.Handle* to *util*; MC: *framework.DrawingView* to *util*; ASP: *framework.HandleEnumeration.nextHandle()* for type *util.Handle*; ASP: *framework.DrawingEditor.getUndoManager()* for type *UndoManager*; MC: *framework.Tool* to *util*; MC: *framework.DrawingEditor* to *util*; MC: *framework.Locator* to *util*; MC: *framework.Figure* to *util*; MC: *framework.Connector* to *util*; MC: *framework.ConnectionFigure* to *util*; MC: *framework.Drawing.findFigureInsideWithout (int,int,Figure)* to *util*; MC: *framework.DrawingChangeEvent.Drawing ChangeEvent(Drawing,Rectangle)* to *util*; ASP: *framework.DrawingChangeListener.drawingInvalidated([...]).e* for type *framework.DrawingChangeEvent* ASP: *framework.DrawingChangeListener.drawingTitleChanged([...]).e* for type *framework.DrawingChangeEvent* ASP: *framework.DrawingChangeListener.drawingRequestUpdate([...]).e* for type *framework.DrawingChangeEvent* MC: *framework.Locator* to *util*;
7	MC: *util.UndoableCommand* to *standard* MC: *util.UndoableTool* to *standard* MC: *util.Figure* to *standard* MM: *util.ConnectionFigure.startFigure()* to *util.Figure* MM: *util.ConnectionFigure.endFigure()* to *util.Figure* ASP: *util.GraphNode.node* for type *standard.Figure* MC: *util.GraphLayout* to *standard* MC: *util.ConnectionFigure* to *standard* MC: *standard.RedoCommand* to *standard* MC: *standard.UndoCommand* to *standard* MC: *util.JDOStorageFormat* to *standard* MC: *util.UndoableHandle* to *standard*

TABLE II. JHOTDRAW: DETECTED CYCLES AND REFACTORING SEQUENCES APPLIED TO REMOVE THEM.

The refactoring steps reported in Table II show that our tool tried to break the dependency from *framework* to *util* by applying the MC and ASP refactorings in a precise sequence. The first operation consists in moving the class *framework.Handle* to *util*. This operation introduces a new dependency between the two packages *framework* and *util*, since *framework.HandleEnumeration.nextHandle()* has a return type *Handle*, and Handle is now in package *util*. To address this issue, the tool proposes to apply ASP on *framework.HandleEnumeration.nextHandle()* (operation 3). The subsequent operations contribute further in reducing the number of dependencies. Only the 8th operation (*i.e.*, MC: *framework.Figure* to *util*) appears to be problematic. In fact, by moving Figure into its new package, we introduce new dependencies. This happens because another class contained in *framework* is heavily coupled with *Figure* (*i.e.*, many methods return *Figure* or require arguments of type *Figure*) and moving *Figure* to *util*, we automatically add 23 new dependencies from *framework* to *util*. This contributes in increasing the effort required to remove the cycle. The new dependencies are slowly removed by further applying the MC and ASP refactorings.

[12]A dependency cycle is described as a set of packages, where the element in position N depends on N+1 and the last depends on the first. The common package name prefix "org.jhotdraw" has been removed for readability purposes.

After this step, the system still contained 1 unresolved cycle.

cycle #7 ({*framework, util*}): This last iteration also had to be split into multiple phases. The number of steps within the optimal refactoring sequence is relatively contained (11 refactoring operations), but the memory resources required to compute all the possible alternative paths was considerable. Also in this case the tool preferred to resort to MC in most of the steps. However, in few cases, it also suggested to use the MM and ASP refactoring. MM could be seen as a variant of MC with a lower impact on the design of the system (since a smaller part of functionality is moved across packages). The reason why MC is often preferred over MM is that it provides a means to remove dependencies in a smaller number of steps. If no other class depends on the moved class, then no negative side effect will result from the refactoring.

B. Industrial Project

To further evaluate our tool, we approached a team working in one of the largest IT companies in Switzerland. The team, composed of four people, was actively developing a module that was part of a larger project. The version that we could analyze consisted of 865 Java files distributed across 159 packages for a total of 50.000 non-comment lines of code.

Our contact person explained to us that the team prided itself for dedicating special care to good object-oriented design practices. The quality of their software was, according to them, superior to that of other modules developed within the same project. The team also regularly used SonarQube[13], a quality assessment tool that, among other things, offers a report of all the cycles contained in a project.

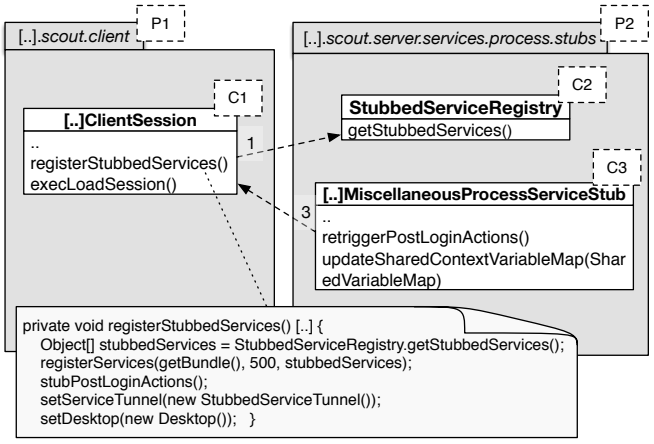


Fig. 6. One of the cycles found in the industrial project

During our analysis, Marea found 25 cycles formed across 22 packages. We asked our collaborator (a member of the development team) to select the most relevant cycles. From those, we chose to analyze the cycle represented in Figure 6. This cycle involves two packages (*i.e.*, *[..].scout.client* and *[..].scout.server.services.process.stubs*) and is caused by a total of four dependencies involving three classes. Any detail that may reveal the identity of the company has been omitted as explicitly requested. Packages and classes have been labeled for convenience.

[13]http://www.sonarqube.org

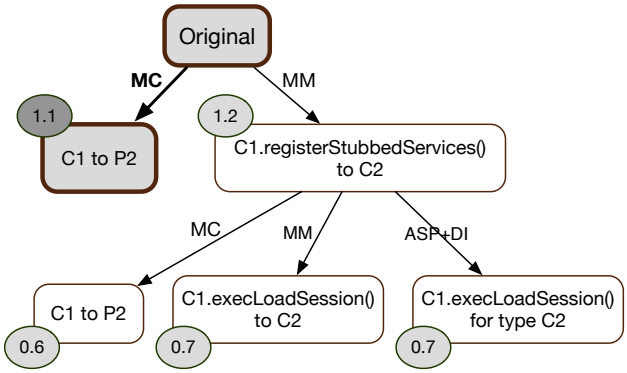


Fig. 7. Simulation tree for the cycle found in the industrial project

To break this cycle, Marea ranks the package dependencies based on Equation 2. Since the dependency between *C1* and *C2* is caused by one single invocation, the tool proceeds with the computation of all refactoring paths that may lead to its removal. The result of the simulation phase is shown in Figure 7. As we can see, Marea evaluates 4 alternative refactoring sequences. The first one consists in moving the class *C1* to the package *P2*. This simple refactoring not only eliminates the dependency from *P1* to *P2*, but also the one from *P2* to *P1*. In fact, at the end of the refactoring, all the classes involved in the cycle will be in one single package. This solution obtains a profit score of 1.1, which makes it the best refactoring sequence. An alternative solution, would have been to move the method *C1.registerStubbedServices()* to *C2*. This operation is similar to the previously described move class, with the difference that only the method involved in the dependency needs to be moved. Unfortunately, another method in *C1* (*i.e.*, *execLoadSession()*) depends on the moved method. Because of that, the dependency from *P1* to *P2* is still not broken and further steps must be taken. The tool evaluates 3 options: MC of *C1* to *P2*; MM of *C1.execLoadSession()* to *C2*; ASP+DI of *C1.execLoadSession()* for type *C2*. The first refactoring simply replicates the approach taken in the optimal solution. The second one moves the method that is now causing the dependency between the two packages to its target class. This solution breaks the cycle since *C1.execLoadSession()* has no incoming dependencies. The last refactoring inverts the dependency between the two packages by removing the explicit reference to *C2* in *C1.execLoadSession()* using ASP+DI.

To test the full applicability of the obtained solutions, we performed all the proposed refactoring operations using a common IDE. We initially opted for Eclipse[14], but eventually had to find for an alternative. In fact, Eclipse supports all the required refactoring operations but failed in performing MM of *C1.registerStubbedServices()* to *C2*. The reason is that the move method refactoring "cannot be used to move potentially recursive methods". Despite the fact that *C1.registerStubbedServices()* is by no means recursive, we could only complete the operation manually. In a second attempt, we chose to use IntelliJ IDEA[15]. This IDE correctly performed the MM refactoring by first adding a static modifier to *C1.registerStubbedServices()* and then moving it to its target class. All subsequent refactoring operations were also correctly applied. In the end we verified that all proposed solutions are

[14]https://www.eclipse.org
[15]https://www.jetbrains.com/idea/

sound and lead to the complete removal of the cycle.

We asked one of the developers to comment on the proposed solutions. After an attentive analysis of the code, he explained that:

- Solution 1 (MC), is simple and might be applicable if there were no restriction on how the two packages are deployed. Unfortunately client and server are deployed separately. Moving the class *C1* to the package *P2* would require to include client libraries into the server bundle.
- Solution 2 (MM + MC), is meaningless since it reaches the same result as in Solution 1 but in two steps instead of one.
- Solution 3 (MM + MM), is not a complete solution since the moved method (*execLoadSession()*) overrides a method of its superclass. The method also calls itself using the super keyword (*i.e.*, *super.execLoadSession()*). Moving the method to another class would break the functionality of the method.
- Solution 4 (MM + ASP/DI), is the solution of choice of our subject. This refactoring is more time consuming and complex compared to all other ones, but cleanly separates the concepts contained in the two packages. The functionality remains where it has been originally placed and a newly introduced interface serves as a contract between the two classes causing the dependency.

One additional solution, that might apply only in this particular case, suggested by our user was to move *P2* under *P1*. In fact, *P2* only contains lightweight classes with few dependencies and no business logic. These classes have been created for testing purposes and do not necessarily need to be deployed with any of the two packages. Making *P2* into a sub-package of *P1* would quickly remove the cycle and only introduce a minor semantic inconsistency. On one hand, the classes contained in *P2* should logically belong to the *scout.server* package hierarchy, as they relate to the server domain. On the other hand, moving them to *P1* would be acceptable because they are exclusively used for testing purposes. This solution would not completely remove the cycle, but, for our user, cycles contained inside architectural components (*i.e.*, *scout.server*, *scout.client*) are considered to be of secondary importance.

Solution 4 has been chosen as best refactoring strategy because it reflects a standard refactoring adopted by the team to remove dependencies. In fact, many classes deployed separately on client and server, share common interfaces. Concrete dependencies are resolved through a dependency injection mechanism based on AOP and the exchange of serialized classes over HTTP. The preference expressed by the user for this refactoring strategy could be explicitly factored into the profit function (*i.e.*, in Equation 3, add another term: $w_4 \times number\_of\_ASPDI$).

Solution 2 showed that the algorithm might lead to a meaningless solution when the proposed refactoring path reaches a previously explored subgraph. This point can be used to optimize the construction of the decision tree.

Solution 3 brought up another corner case that needs to be treated with special care. This case will be addressed by implementing a dedicated guard in the algorithm responsible for building the decision tree.

V. DISCUSSION

A. The Package Blending Problem

Our approach is always guaranteed to reach a solution, as long as enough time and memory are provided. In fact the MC refactoring can be applied in any circumstance, and will therefore always be used to approach an optimal solution. The problem is that the MC refactoring might often not represent the most desirable type of refactoring from a semantic point of view. The change implied by this refactoring is only justifiable if the behavior described in the moved class is consistent with the category described by the target package. If we ignore this reasoning and blindly move classes from one package to another (as we intentionally did in subsection IV-A), we will end up gradually dismantling the modularity of the system. This behavior may possibly lead, in the extreme case, to the complete blending of one package into another. If that happens we will have eliminated a package dependency causing a cycle but we would have significantly altered the high-level structure of the project. This contrasts with the separation of concerns design principle, which advocates the isolation of cohesive functionality.

This general tendency towards the unification of packages should primarily be prevented by the user. The profit function often suggests MC as an optimal refactoring strategy (See Table II), as it offers a fast and uncomplicated way to get rid of a dependency. Another approach to control the abuse of the MC refactoring strategy, could consist of extending the profit function (Equation 3) with an additional metric that measures package cohesion. One such metric could be an opportunely adapted variation of LCOM [19], a metric that quantifies the number of responsibilities of a given functional unit, or CRSS [20], a metric for good package design.

A further option to control the execution of the simulation algorithm could be to support the specification of structural invariants that define the boundaries of allowed refactorings. These invariants could explicitly forbid the relocation of classes contained in specific modules or the separation of entities sharing some semantically relevant property. The specification of the architectural components (in terms of sets of packages) would already be sufficient to prevent design breaking changes (*e.g.*, moving classes between a *client* and a *server* component).

B. Prototype Limitations and Tradeoffs

Simulating multiple refactoring scenarios, as we have seen in section IV, has its cost. One of the main issues encountered during our experimentation is related to the amount of memory required to store each evaluated simulation step. In our prototype we tried to optimize memory consumption by using in-memory object models that can be evolved by only saving the incremental changes that separate one version of the model from another. This was possible thanks to a third-party framework called Orion [21]. Despite the advantage of incremental change memorization, we still encountered several cases in which the simulation exhausted the available memory resources. This issue could probably be addressed by

improving the Orion framework or opting for a more scalable data management approach for storing our models (*e.g.*, a graph database). Another viable solution consists in optimizing the simulation process by enhancing the profit function (subsection V-D) or by taking into account complementary information (*e.g.*, architectural rules, build configuration files) that leads to the definition of structural constraints.

During our experiment we decided to cope with the above mentioned limitation by pruning the simulation tree at a prefixed depth. This choice might have a negative impact on the accuracy of our technique. In fact, a partial sub-optimal refactoring sequence could theoretically develop into an optimal solution in further steps of the simulation. By varying the maximum length of a simulation path, we can vary the tradeoff between the overall cost of the analysis and its level of precision.

Another significant aspect that needs to be considered when implementing our approach is the level of correctness of the simulated refactoring operations. In our prototype, as mentioned, we delegated the versioning of our simulation models to a dedicated framework. This framework allows us to evolve existing models by applying predefined basic change operations (*e.g.*, create/delete class, create/delete method). The refactoring strategies discussed in this paper (subsection II-B) had to be implemented by combining several of those change operations. This implies investigating and managing all possible edge cases, updating references and maintaining the system in a generally consistent state. This whole spectrum of complexity is well managed in commercial refactoring engines, but requires considerable effort to be implemented from scratch. In our prototype, we did our best to address all encountered issues and to handle the most recurring cases. Despite our effort, we recognize the fact that implementing a fully correct refactoring operation requires a considerable engineering effort. Furthermore, the completeness of the implementation also depends on the level of detail of the meta-model used to represent the system under analysis. If certain details of the system are only partially captured in the model, then those details are not guaranteed to be correctly updated to reflect the applied refactoring. We plan to address the above mentioned limitations by engaging in further experimentation and performing comparative studies with current implementations of refactoring algorithms (*e.g.*, Eclipse JDT refactoring engine[16]).

C. Refactoring application

As explained in subsection IV-B, applying the refactoring operations suggested by Marea is not always easy. Our approach simulates the refactoring operations on models that are only partially as complex as the reality they represent. This means that all the details necessarily omitted in our models may play a role in the actual applicability of the suggested operations. We try to prevent complex situations by analyzing the properties of the involved code elements (*e.g.*, MM cannot be applied on methods that contain invocations to other static methods defined in the same class). Our approach can be considered safe and unobtrusive. A possible evolution of our approach could be based on the speculative application of refactoring step in a code sandbox. This approach would be more pragmatic but might possibly require more computation time, as the analysis model (on which decisions are taken) needs to be reverse engineered after each iteration.

[16]http://www.eclipse.org/jdt/

D. Profit function

The profit function used in our approach and described in Equation 3, is a simple attempt at quantifying the effect of a refactoring strategy over a project. It was designed to guide the user in choosing a better solution based on objective measurements. Given that the optimal solution can only be decided by the user, we assume that this function should only be used as a general heuristic for comparing competing alternative solutions. The metrics employed in the calculation are well known indicators of the structural stability of software packages. The function could be improved by adding further metrics and changing their relative impact by modifying their weight coefficients. A more adequate combination of weights could be inferred by performing empirical studies and recording the paths typically chosen by the majority of the users.

VI. RELATED WORK

Providing automatic support for the removal of dependency cycles is a complex problem. We here report on the main research directions that have developed around this topic.

A. Refactoring Candidate Identification Heuristics

Some approaches are specialized in detecting the most critical elements in a dependency cycle. These approaches do not advise explicitly on how to remove a cycle, but rather provide hints on where to look in order to devise a proper refactoring strategy.

Melton *et al.* present Jepends [6], a tool that identifies the classes that should be refactored in order to remove a cycle. Each class in the system is analyzed and ranked based on its number of incoming/outgoing dependencies and on the number of cycles it is involved into. Classes that most contribute to cycles and with higher coupling are considered to be the best starting point for further inspection and consequent refactoring.

Jooj [8] is another tool by Melton *et al.* that warns the user about the existence of cycles as soon as they appear. The warnings are displayed within the IDE and further instructions may be provided to remove critical dependencies. The main assumption behind this solution is that, as long as the user is aware of the impact of his actions, new cycles will not be introduced. The authors declare their intent to implement refactoring suggestions based on patterns described by Lakos [22] (*e.g.*, escalation, demotion, dumb data, manager class). No further information is provided regarding the challenges involved in adding this specific feature.

Laval *et al.* introduce CycleTable [23], a visualization technique that should guide the user in the identification of critical code elements involved in cycles. CycleTable does not focus on a single solution to break cyclic dependencies. It rather groups classes based on their coupling profile. This approach could be compared to the one used in Jepends [6], as both aim at classifying code units based on structural metrics.

Laval *et al.* also present another tool, Ozone [24]. Orion suggests which dependencies should be removed in order to obtain a layered package structure. The target architecture is inferred automatically based on the analysis of the source

code and a set of optional user-defined dependency constraints. The refactoring steps required to concretely obtain the target architecture are left to the user to investigate.

All the mentioned approaches only provide small hints regarding what concretely needs to be done to break a dependency cycle. In fact, presenting a list of candidates for refactoring without further instructions on how to perform the actual refactoring task only partially contributes to solving the problem of cycle removal.

Oyetoyan *et al.* [25] propose a new heuristic metric that can be used to guide the removal of intra-class cycles. Their algorithm identifies an optimal refactoring strategy that maximizes a set of structural metrics (*e.g.*, coupling) and promotes specific implementation choices (*e.g.*, class attribute static modifier). The resulting refactoring is then applied to update an annotated graph model representing the system. The algorithm can be applied multiple times until all class cycles have been removed. The main difference between this approach and Marea is the fact that Marea explores multiple refactoring options and evaluates combinations of multiple refactoring strategies during each analysis step. Instead of simply applying a pre-configured default refactoring, Marea simulates complete refactoring scenarios and is therefore capable of measuring the impact of the overall refactoring instead of just focusing on devising a solution based on step-wise optimization. Marea also focuses on package cycles, instead of class cycles.

B. Refactoring Simulation

Many commercial tools provide support for removing cyclic dependencies through the simulation of refactoring operations.

In Structure101[17], the user can move classes, methods, fields and packages using drag-and-drop actions. No guidance is provided during the process. Refactoring operations, such as move method, seem to be always easily applicable without considering the effects that such an operation would involve when applied on the corresponding code elements. The dependency graph of the analyzed system is hard to navigate and cannot be reduced to isolate single cycles.

SonarGraph-Architect[18] allows the user to identify the dependencies that cause a cycle. These dependencies are ranked and described at the granularity level of classes. No hints are provided on how to remove the identified dependencies. the user can move classes from one package to another and observe the impact of the operation on the dependency structure of the system.

Lattix LDM[19] visualizes cyclic dependencies using a Dependency Structure Matrix [26]. Besides finding cycles, the tool also supports basic structural editing features such as the renaming, moving and deletion of packages. This limited set of operations may help removing certain cycles, but will also most probably encourage users to reduce the modularity of their system.

Pasta [11] is a tool developed by Compuware. Pasta, like Ozone [24], tries to derive the best layering configuration for a given system and presents the user with all the dependencies that need to be removed to implement that configuration. Pasta also offers a graphical interface that supports the simulation of several refactoring operations by drag-and-drop: move package, move class. Simulated changes can eventually be automatically applied to the code. The author claims that a future version of the application will also support advanced refactoring operations described by Martin [2].

All the presented tools deal with dependency cycles in the same terms as one would approach a graph problem. Each node composing the cycle can be moved around regardless of the complexity of its underlying implementation. Little or no guidance is provided on which operation may offer the best compromise between effort and benefit. The user has the responsibility to decide between many refactoring options that can only often be performed only on larger granularity elements (packages, class). The gap between the simulated change operations and the actual refactorings that might eventually be applied on the corresponding code elements remains large.

If the editing features provided by the current commercial solutions could be extended with more sophisticated refactoring operations and supported by an intelligent decision support system, we might have a complete solution for dealing effectively with cyclic dependencies.

VII. Conclusion

In this paper we introduce a novel approach to guide developers in the task of removing cyclic dependencies among packages. We propose a tool that simulates various refactoring operations and identifies the optimal change sequence based on a profit function. We also report on the challenges that we encountered during the implementation and evaluation of the tool.

We conclude that assisting a user during the removal of cyclic dependency is possible and worthwhile. Our prototype illustrates the basic phases that such a process should support. In the future we plan to improve the scalability of the tool by using an alternative framework for model versioning. We also plan to improve the consistency and reliability of the supported refactoring simulation operations. We believe that by addressing these two aspects we could provide a solution that effectively complements existing commercial solutions.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Agile Software Assessment” (SNSF project Np. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

References

[1] R. C. Martin, “Granularity,” 1996. www.objectmentor.com.

[2] R. C. Martin, “Design principles and design patterns,” 2000. www.objectmentor.com.

[3] M. Fowler, “Reducing coupling,” *IEEE Software*, vol. 18, no. 4, pp. 102–104, 2001.

[4] M. Mair and S. Herold, “Towards extensive software architecture erosion repairs,” in *Software Architecture* (K. Drira, ed.), vol. 7957 of *Lecture Notes in Computer Science*, pp. 299–306, Springer Berlin Heidelberg, 2013.

[5] T. Oyetoyan, D. Cruzes, and R. Conradi, “Can refactoring cyclic dependent components reduce defect-proneness?,” in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pp. 420–423, Sept. 2013.

[17] https://structure101.com

[18] https://www.hello2morrow.com/products/sonargraph

[19] http://lattix.com

[6] H. Melton and E. Tempero, "An empirical study of cycles among classes in java," *Empirical Software Engineering*, vol. 12, no. 4, pp. 389–415, 2007.

[7] H. Melton and E. Tempero, "Identifying refactoring opportunities by identifying dependency cycles," in *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ACSC '06, (Darlinghurst, Australia, Australia), pp. 35–41, Australian Computer Society, Inc., 2006.

[8] H. Melton and E. Tempero, "Jooj: Real-time support for avoiding cyclic dependencies," in *Thirtieth Australasian Computer Science Conference (ACSC2007)* (G. Dobbie, ed.), vol. 62 of *CRPIT*, (Ballarat Australia), pp. 87–95, ACS, 2007.

[9] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *International Conference on Software Engineering (ICSE)*, 2015.

[10] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pp. 672–681, IEEE Press, 2013.

[11] E. Hautus, "Inmproving Java software through package structure analysis," in *International Conference Software Engineering and Applications*, 2002.

[12] S. Shah, J. Dietrich, and C. McCartin, "Making smart moves to untangle programs," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 359–364, Mar. 2012.

[13] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *Software Engineering, IEEE Transactions on*, vol. 35, pp. 347–367, May 2009.

[14] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[15] S. Demeyer, S. Tichelaar, and S. Ducasse, "FAMIX 2.1 — The FAMOOS Information Exchange Model," tech. rep., University of Bern, 2001.

[16] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of Moose: an agile reengineering environment," in *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, (New York, NY, USA), pp. 1–10, ACM Press, Sept. 2005. Invited paper.

[17] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, pp. 146–160, 1972.

[18] J.-R. Falleri, S. Denier, J. Laval, P. Vismara, and S. Ducasse, "Efficient retrieval and ranking of undesired package cycles in large software systems," in *Objects, Models, Components, Patterns* (J. Bishop and A. Vallecillo, eds.), vol. 6705 of *Lecture Notes in Computer Science*, pp. 260–275, Springer Berlin Heidelberg, 2011.

[19] M. Hitz and B. Montazeri, "Measure coupling and cohesion in object-oriented systems," *Proceedings of International Symposium on Applied Corporate Computing (ISAAC '95)*, Oct. 1995.

[20] H. Melton and E. Tempero, "The crss metric for package design quality," in *ACSC '07: Proceedings of the Australian Computer Science Conference*, 2007.

[21] J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri, "Supporting simultaneous versions for software evolution assessment," *Journal of Science of Computer Programming (SCP)*, May 2010.

[22] J. Lakos, *Large Scale C++ Software Design*. Addison Wesley, 1996.

[23] J. Laval, S. Denier, and S. Ducasse, "Identifying cycle causes with cycletable," in *FAMOOSr 2009: 3rd Workshop on FAMIX and MOOSE in Software Reengineering*, (Brest, France), 2009.

[24] J. Laval, S. Ducasse, and N. Anquetil, "Ozone: Package layered structure identification in presence of cycles," in *9th BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)*, (Lille, France), 2010.

[25] T. D. Oyetoyan, D. S. Cruzes, and C. Thurmann-Nielsen, "A decision support system to refactor class cycles," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 231–240, IEEE, 2015.

[26] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *Proceedings of OOPSLA'05*, pp. 167–176, 2005.