Abstract—A component is usually complemented with guidelines expressing its proper use, e.g. the appropriate order of calls that clients should conform to. During reuse, clients could easily alter such an order, and this could result in reused components that become unreliable, since architectural guidelines have not been honoured. Sometimes architectural guidelines are simply unknown, hence whether components are misused by clients is uncertain.

This paper proposes an approach to document the architectural guidelines that client classes should comply with when reusing a component. We empower component developers to provide such guidelines, conveying design decisions, along with the code of components in such a way to be apt to automatic checks. Then, clients compliance with architectural guidelines of reused components can be automatically checked by an aspect-based tool. As a result, proper usage of reused components can be ensured, and in turn the behaviour of components should be correct. This strengthen the reliability of the resulting system. Especially for continuous evolution, having automatic conformance checks is paramount for obtaining the correct behaviour of reused components.

I. INTRODUCTION

One of the difficulties for reusing existing object-oriented components is that developers need to find out the way to properly employ such components. While a component relies on some assumptions, i.e. some pre-conditions have to be satisfied for an operation to be invoked on the component, developers reusing such a component could be unaware of such conditions. Moreover, post-conditions, i.e. the outcomes that are considered to be the correct behaviour under normal conditions, after the use of some operations, could be unknown to developers reusing components [12].

A client application that accidentally forces the assumptions of a reused component is likely to experience negative effects on, at least, availability and reliability. However, the improper operations performed on a reused component might be hard to find and correct. This becomes more exacerbated when an application uses remote services residing e.g. on a cloud, because of the complexity brought by distribution and the unavailability of service source code.

Although specification and design documents of a reused component could detail architectural guidelines [7], they are not readily available [9]. I.e. during code and test phases, typical of agile methodologies, the developer is left alone and no automatic check is available. Hence, the developer could somehow unwittingly force an improper execution on the component unaware of unwanted side-effects. Generally, whether component usage within a larger software system occurs according to the original intent can not be automatically assessed. Some approaches use a model of components, however models may differ from real components and assessing whether clients are properly using a component requires model validation. Moreover, even several techniques are available to reconstruct the design of an existing system, the underlying design decisions can not always be unambiguously discovered from the code [15], [14], [18].

This paper aims at tackling the lack of support within object-oriented systems that would allow given design decisions to be enforced [13]. For this, we provide a way to express design decisions within the code of components, so that they can be handled by object-oriented languages, hence always up-to-date. Moreover, we devise an approach to automatically check whether such decisions have been honoured within the larger system reusing some components. Both design decisions and checking supports would be available during the development of a system reusing components, hence greatly helping developers to properly employ components.

Automatically checking whether components are used according to their designed aims is very valuable. This is in good agreement with agile methodologies stating that requirements should be executable, i.e. written as tests, hence automatically checked against the resulting software system.

The content of this paper is organised as follows. Section II provides an outline of the approach. Section III describes the design decisions we are interested in and how to express them. Section IV shows how design decisions can be automatically checked on client classes reusing components. Section V reports on related approaches. Finally, conclusions are drawn in Section VI.

II. APPROACH FOR STATING DESIGN DECISIONS

During design, software architects identify components and relations between such components. The software ar-
chitecture can be documented using appropriate languages, such as ACME [7], or notations, such as UML [16], and others [4]. Design documents complement and enrich the code of resulting components and sometimes are available to developers of applications reusing components. Some of the design decisions, that software architects embed into components are based on best object-oriented practices [2], [6], coding idioms [1], [5], advanced solutions for super-imposing design patterns roles into application classes [8], etc. However, software architects select desired properties and tailor choices for the component at hand. As a result, each component has its own design decisions that should be documented.

This paper, firstly, identifies a set of recurrent design decisions and provides a way to express these into components. We suggest ad-hoc Java annotations for expressing them. Java annotations are considered appropriate for their concise, customisable and unintrusive nature, and allow marking entities on the code. Our ad-hoc annotations are used to mark a target method, class, or a set of classes to indicate how each component has to be used by a client application, hence within the overall architecture. Such annotations express the assumptions underlying correct execution. Annotations are unintrusive since they are markers placed outside the code of methods, i.e. they do not intertwine with the algorithm of a method, however they are near the code, so they can be easily read and updated when their target method or class changes. Moreover, annotations are not processed by the compiler, so no early error messages are given, by an IDE or a compiler, while the code is still under construction.

Secondly, we propose a tool that checks design decisions by connecting with annotations. An additional benefit of using annotations is that when the tool need not be used anymore, annotations can be kept with the component, since these are simply ignored by the compiler, without risking to compromise nor add overhead during the execution of the final system. We propose to have a library that provides, along with annotations, the automatic assessment of design decisions on the overall application comprising the annotated reusable components and client classes (these need not include annotations). The provided tool uses aspect-orientation [10] to connect with annotated components and transparently check, according to the data provided at runtime by both clients and annotations, whether proper operations have been invoked on reused components. Such a library comprising software aspects can be included into an existing IDE for convenience, hence providing feedback while developing client applications.

As a result, within our approach, expressing design decisions by means of annotations and then having annotations that perform conformance checks allows an application to be automatically verified against a defined software architecture. This greatly enhances the reliability of the overall resulting system.
pipe. When this is the case, the client class would have inappropriately operated on the reused component, because against the pipe and filter style. Then, a warning will be issued.

Section IV describes how the check is performed.

B. Guidelines for control flow within a class

Generally, when reusing a class providing several methods, for the class to exhibit the correct behaviour, only some of all possible sequences of method calls should be allowed. This is because each method execution will change the state of the class instance, and the state resulting just after the method has finished could be compatible only with the execution of a few methods of the same class, not necessarily all available ones. Of course, before executing a method its preconditions could be checked, however often preconditions are not included into the code. When developing a component that will be reused, leaving such conditions unchecked could become a problem, because the usage pattern of possible client classes is unknown.

For expressing precedences of method calls, we provide annotations @CallOrder and @SetOrder to be used on methods, and taking as parameter a number. They respectively indicate the ranking of calls allowed, and the changes on the ranking to be performed after executing the method. Given some numbers \(x\) and \(y\) with \(x > y\), a method marked with annotation @CallOrder\((x)\) will have to be called only before other methods marked with annotation @CallOrder\((y)\) are called. Moreover, given a method \(m\) annotated with @SetOrder\((x)\), the execution of \(m\) enables methods annotated with @CallOrder\((x)\) to be executed after \(m\) has finished, independently of the ranking of \(m\). The same number can be passed to annotation @CallOrder used on different methods, meaning that any of such methods can be called, but after one other method of a lower order has been called. Methods without annotations can be called any time.

Figure 3 shows a class using annotations CallOrder and SetOrder. In such a class, methods reserve() and increasePrice() can be called only as long as method buy() has not been called yet (see annotation CallOrder\((1)\)). When method reserve() has been called, only method buy() can be called afterwards (see annotation SetOrder\((2)\)). Methods getState() and getPrice() can be called any time.

As another example, consider a class holding data and that has to behave as a stack. It suffices to annotate its method push() with annotation CallOrder\((1)\), while its method pop() with annotations CallOrder\((2)\) and SetOrder\((1)\). Note that with such annotations, multiple calls to the same method push() (or pop()) will be allowed, however method pop() cannot be called first.

More generally, such annotations allow the state of an object to pass only for well-known and desired transitions. Being the callee side the one regulating which transitions are allowed, as opposed to let callers decide, the behaviour of the reused component will be the correct one.

C. Guidelines for control flow for several classes

Similarly to the above order of calls for methods of a class, for a group of reused components to properly execute, it would be the case that a given order is chosen for methods to be called on different classes. A possible design decision is to constrain unknown client classes to perform only selected interactions with the group of reused components, in order to shape the overall architecture.

I.e. suppose that classes \(A\) and \(B\) are going to be reused, and that the methods of class \(B\) should be invoked by a client class \(C\) only when methods of class \(A\) have been previously called by \(C\). This would ensure that class \(C\), unknown to the reused classes, allow \(A\) to assume a proper state for \(B\) to rely on.

Annotation @ClassOrder, to be used as a marker for a class, let the developer express by means of a numerical parameter the ranking of use for a class with respect to other classes marked with the same annotation. Note that this annotation can be combined with the above annotations for the order of methods. Together they support the expressions of the ranking of method calls across several classes.

D. Guidelines for visibility of classes

When having developed a group of components to be reused, a few of such components can be made available...
to provide assistance to other components, however not to all possible clients. Such a rule conforms to design pattern Facade, for which a subsystem comprising a group of components is hidden and accessed only by a simplified interface. Moreover, this is useful for additional design patterns, such as e.g. for design pattern State, whereby only role Context should use roles ConcreteStates [6].

Deciding on the visibility of a class is a recurrent issue during development, which the usual language operators for methods, such as private or public are not fully able to express, since e.g. a public method will be available to any client class, not only a selected few; even package-level visibility can be considered too coarse grained.

We suggest to express the design decision whereby public methods of a given class are available only to some other classes. For this, annotation @Afferents is made available to be used on a class. This annotation takes as input parameter a list of classes, which are enabled to invoke public methods on the target class. E.g. for expressing that only classes A and B can be coupled with class C, thus limiting the use of the public methods of class C only to enabled classes A and B, class C would have to be annotated as follows

\[
@Afferents\{A, B\} \text{ public class } C \{ \ldots \}
\]

Suppose that classes A, B and C are provided as a group of classes to be reused on a client application, then only classes A and B will be available to other clients, whereas class C will remain hidden to clients. Note that, class C cannot be hidden by nesting it within another class, since it would otherwise be possible only for one class among A or B to use it.

E. Guidelines for dependency and performance

A component providing some services could have been developed in such a way that it would be convenient to have only a few client classes. However, such client classes are unknown during its development. I.e., the identity of the calling classes will be discovered later on, when the component will be reused. The constraint on the number of allowed calling classes derives from the concept of centrality of a component. A component having a high number of callers is of great importance for the execution of the whole system. However, the component could have been designed to require relevant computing resources that become overwhelming when called by many client classes. Additionally, design decisions could suggest that a low fan-in is desired.

In order to let the component developer limit the number of simultaneous calls to a class, we have devised annotation @MaxCallers. This takes as a parameter the number of invocations, on any method, that client classes are allowed to perform at the same time on any instance of the annotated class. Similarly, the additional annotation @MaxFanIn, taking as a parameter a number, allows developers to specify the maximum number of client classes that can be coupled with such a class.

Moreover, when monitoring applications to analyse their outputs and performances, operations are traced and then analysed to distinguish their characteristics and the extent of an operation on the overall cost. Operations of reused components could be classified in several ways. Here, we suggest to distinguish: (i) operations that mark the starting point for an elaboration phase and that bring a component to its initial state; (ii) operations that perform time consuming elaborations or read data from a disk, etc., whose execution could be started several times, and (iii) operations that bring a component to a final state. Such a classification is imbued with design decisions, which we want to be explicitly documented. Hence, correspondent annotations are needed to mark operations of reused components.

By uniformly marking operations, i.e. the same kind of operations are given the same annotation, application developers can then easily extract among the whole execution flow the operations of interest. In this case, the provided support aims at facilitating execution analyses. Then, application developers would have to check whether performances can be improved by choosing alternative implementations, different order of calls, or hardware configurations.

IV. Checking compliance with design decisions

Once a client application class calls methods of reused components that have been annotated, we are able to check usage constraints provided with design decisions. The outcomes of checks depend on the dynamic behaviour, i.e. a static analysis would miss the real interconnections and the behaviour that the application experiences on reused components.

We use aspect-oriented programming for incorporating checks. Aspects are modules that can express crosscutting concerns, avoiding the spread of related code into several classes [10]. Aspects allow developers to define pointcuts that express the parts of a program, called join points, that have to be captured. Once control has been captured, it will be handed to an advice that can execute some code. AspectJ is the most advanced aspect-oriented language [11], and a compiler is available to weave aspect code with classes.

In our approach, provided checking aspects are triggered by the annotations defined above, and introspect the running classes and instances. Our aspects are general, i.e. they work for any class playing one of the role defined by provided annotations and for any client application. They are clearly independent of both reused components and applications. Of

In general, the number of modules dependent on a given module is referred as afferent coupling.

3eclipse.org/aspectj
We have developed aspect CheckRolePipe to assess the application. The broader the coverage of cases that exercise the code for common (or all possible) actual operational situations. The aspect should be executed according to test cases that exercise the code for common (or all possible) actual operational situations. The broader the coverage achieved by test cases the better aspects can observe and assess the application.

In order to assess the above constraint whereby the same filter cannot use the same pipe both as a source and a sink, we have developed aspect CheckRolePipe, whose interactions with other classes are shown in Figure 4, whereas its listing is in Figure 5. The aspect works at follows. Calls to methods marked @Sink and @Source are captured together with the references to both caller and callee, i.e. a filter and a pipe, respectively. Pointcuts sinkCalls() and sourceCalls() define the join points that have to be captured (see lines 8 to 12, in Figure 5). Then, control is handed to correspondent advices.

The advice related with pointcut sinkCalls() (see lines 15 to 19), firstly obtains the pipes that have been used before by the captured filter (see line 16), then checks whether this filter has used the captured pipe as a source (see line 17). If this is the case, an error message will be printed. For each class playing role filter, which is discovered at runtime, the aspect holds and updates a list of references to both caller and callee, i.e. a filter and a pipe. Analogously, the advice related with pointcut sourceCalls() (see lines 22 to 26) checks whether the same filter has been using the pipe as a sink.

The above aspect bears many features of other aspects needed for checking the design decisions listed in Section III. Thus, for the sake of brevity only the significant steps of other aspects will be given in the following.

The aspects checking the order of calls, intercepts the methods annotated with @CallOrder, then reads the value of the annotation parameter and compares it with the last saved one for the same instance, whose methods have been captured. If the comparison results in a decreasing order then an error will be displayed. When the captured method bears annotation @SetOrder then the value of the parameter is saved as the last value. Otherwise the value of the parameter for @CallOrder will be saved.

The aspect checking classes whose calls have to be regulated according to annotation @ClassOrder is similar to the last one. The major differences are the following: (i) checks will be performed for any method of annotated classes, (ii) the saved value of the last call refers to the instance of a class, and (iii) instances of different classes (bearing the same annotation) will be related to the ranking. The aspect checking whether calling classes are allowed, has to capture methods of classes annotated with @Afferents, then compares the caller class with each element of the list of classes given as a parameter of the annotation. When no match has been found an error message will be displayed.

Similarly to the latter, the aspect checking compliance with @MaxCallers and @MaxFanIn checks the number of active executions from different instances, and the number of different classes, invoking the annotated component.

Figure 4. UML sequence diagram for aspect CheckRolePipe capturing calls and checking conformance with pipe and filter style

public aspect CheckRolePipe {
  private Dictionary<Object, List<Object>> readers =
    new Hashtable<Object, List<Object>>();
  private Dictionary<Object, List<Object>> writers =
    new Hashtable<Object, List<Object>>();

  pointcut sinkCalls(Object f, Object p);
  call (@Sink * *.() ) & & this(f) & & target(p);

  pointcut sourceCalls(Object f, Object p);
  call (@Source * *.() ) & & this(f) & & target(p);

  // check whether the caller is using the same Pipe as source
  before(Object f, Object p) : sinkCalls(f, p) {
    List<Object> lr = readers.get(f);
    if (lr != null & & lr.contains(p)) writeError(f, p);
    putInTable (writers , f , p);
  }

  // check whether the caller is using the same Pipe as sink
  before(Object f, Object p) : sourceCalls(f, p) {
    List<Object> lw = writers.get(f);
    if (lw != null & & lw.contains(p)) writeError(f, p);
    putInTable (writers , f , p);
  }

  private void writeError(Object f, Object p) {
    System.out.println ("Filter instance "+ f + 
      " is using Pipe " + p + 
      " both as sink and source");
  }

  // update the list of used pipes for a filter
  private void putInTable (Dictionary <Object, List<Object>> d, 
    Object f, Object p) {
    List <Object> lrw = d.get(f);
    if (lrw != null & & !lrw.contains(p)) lrw.add(p);
    else if (lrw == null) {
      lrw = new LinkedList<Object>();
      lrw.add(p);
    }
    d.put(f, lrw);
  }
}
V. RELATED WORK

The need of support for expressing design decisions has been clearly stated in [9]. However, the authors confirm that such decisions are usually not documented. In [17], authors propose an approach for recording architecture decisions and trace them to requirements, yet their approach is not helping to assess conformance with the resulting code.

Assessing design decisions, such as preconditions, by means of code within a component, is not always feasible, due to the limited knowledge that a component has of other parts, such as e.g. its clients. Moreover, relevant checks, when possible, are unrelated with component concern and the resulting code inevitably lowers expected performances.

Existing architectural languages and notations can not be of help for assessing conformance between design decisions and resulting code. Firstly, architecture description and code are separated from each other, hence it is difficult to maintain both up-to-date; secondly, existing architectural languages are meant to express decisions at an higher level of abstraction with respect to the code [4], [16], [7] and this makes it difficult or impossible to automatically verify the matching; thirdly, once the lack of correspondence between design and code has been assessed, the developer still has to determine which of the two is the newest or the appropriate one.

VI. CONCLUSIONS

We have identified design decisions that complement the code of reused components, and are fundamental for checking correctness of usage. Although such decisions could be documented separately from components, in our approach, and in good agreement with the principle of executable requirements, we have them together with reused components, as annotations. Support for assessing whether a client application conforms with the proper usage desired for reused components has also been provided. Checks rely on aspect-orientation, hence can be easily plugged during application development and test and unplugged for production. Checks are independent of client applications, hence reusable themselves. The approach is light, since it avoids bloating the code of reused components and has no effect on performance once checks are not needed anymore.

VII. ACKNOWLEDGMENT

This work has been supported by projects JACOS and Infinity Web Edition funded within POR FESR Sicilia 2007-2013 framework, and project PRISMA PON04a2 A/F funded by the Italian Ministry of University within PON 2007-2013 framework.

REFERENCES