Striffs: Architectural Component Diagrams for Code Reviews

Muntazir Fadhel
mfadhel@hadii.ca
HADII Technology
Toronto, Canada

Emil Sekerinski
emil@mcmaster.ca
McMaster University
Hamilton, Canada

Abstract

Despite recent advancements in automated code quality and defect finding tools, developers spend a significant amount of time completing code reviews. Code understandability is a key contributor to this phenomenon, since engineers need to understand both microscopic and macroscopic level details of the code under review. Existing tools for code reviews including diffing, inline commenting and syntax highlighting provide limited support for the macroscopic understanding needs of reviewers. When reviewing code for architectural and design quality, such tools do not enable reviewers to understand the code from a top-down lens which the original architects of the code would have likely used to design the system. To overcome these limitations and to complement existing approaches, we introduce structure diff (striff) diagrams. Striffs provide reviewers with an architectural understanding of the incoming code in relation to the existing system, allowing reviewers to gain a more complete view of the scope and impact of the proposed code changes in a code review.

Keywords: code review, software design, object oriented programming, code comprehension, graphs

1 Introduction

Code reviews constitute a widespread practice employed by software engineers to maintain high software quality and share project knowledge [25]. The formal code review was first defined by Fagan in 1976 [14] as a software inspection practice in which source code is reviewed for correctness, often conducted by multiple reviewers. More recently, code reviews have grown in scope to bring about many benefits including knowledge transfer, increased team awareness, and collaboration resulting in alternative solutions to problems.

While code understandability is a primary component of code reviews, it is a significant source of difficulty for reviewers [34, 35, 39]. Despite the advancements of existing static analysis and program differencing tools, such tools focus on the microscopic understanding needs of reviewers and ignore their macroscopic code understandability needs to a large degree. Moreover, when reviewing code for architectural and design quality, such tools do not enable reviewers to understand the code from a top-down lens which the original architects of the system would have used to design it.

The large cognitive load placed on reviewers during code reviews has numerous impacts, both on the time it takes to understand the code, impact, and scope of the change, but also on the quality of the code review itself. Numerous studies have showed that conducting proper code reviews is time consuming [32] and that developers can spend up to 15% of their time completing them [13]. Additionally, researchers have also provided strong empirical evidence that the outcome of the code review process is erratic and often unsatisfying or misaligned with the expectations of participants [4, 6, 29]. This problematic outcome is a result of the high cognitive demands of code reviews [3], whose outcome is primarily a result of the time and ability of the involved reviewers.

To this end, we present striff diagrams, which visually communicate the architectural effect of an incoming code change on a software system using UML-like class diagrams. Striffs provide reviewers with an architectural understanding of the incoming code in relation to the existing system, thereby satisfying reviewer information needs related to the scope and impact of the proposed code changes in a code review.

To appear in 2021 International Conference on Code Quality (ICCQ), pages 69–78, 2021. IEEE

2 System Comprehension in Code Reviews

A code review is typically completed when a modification needs to be made to a project, either through the implementation of a new feature or a bug fix. Popular collaborative code hosting platforms like GitHub [1] encapsulate the workflow of introducing changes to a software project using Pull Requests. As part of the peer code review in this workflow, a code reviewer must protect the quality and integrity of the code repository from incoming code submissions. The code reviewer typically has two primary artifacts that can be used to conduct the code review.

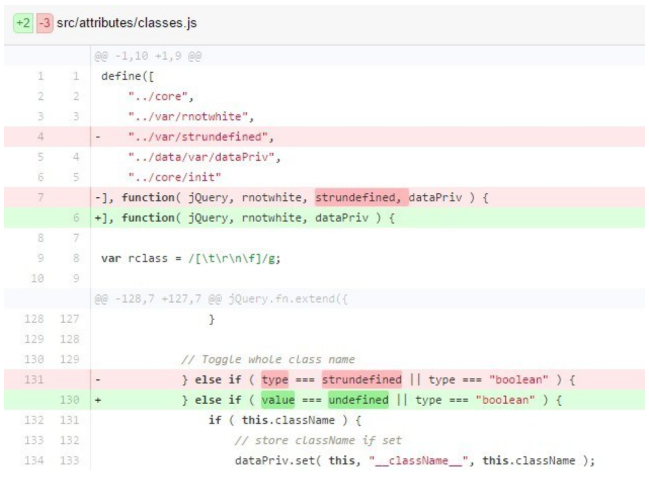
1. **Code**: Source files corresponding to the original software project and the set of changes being introduced into the system as a result of the Pull Request.
2. **Continuous Integration (CI) Artifacts**: The results of various builds, test executions, and static analysis checks as required, typically completed by a CI server.

It is well-known that program understanding plays a pivotal role in performing code reviews and most software-related maintenance activities [2, 9, 33, 38]. In code reviews, reviewers need to understand both the microscopic details of the code and the macroscopic conceptual structure [10]. The microscopic level of abstraction includes the mechanics of classes and methods, which can be examined in the text of the code. The macroscopic level of abstraction includes concrete higher-level concepts including modules, systems and conceptual structures that are not manifest directly in the code. It is the articulation and comprehension of code at this level that continues to take up a significant amount of time in code reviews [6].

Strong evidences from dozens of open-source and commercial projects reveal that typical design flaws are introduced by developers and are responsible for poor maintenance quality and high long-term costs [26, 27, 36, 43]. The limitation is that these design flaws usually have already caused significant loss to the system when they can be detected by existing approaches.

Given that reducing the cognitive load of reviewers improves their code review performance [32], our study is motivated by the following question: "What additional code reviewing artifacts would help reduce the cognitive load placed on reviewers in code reviews?" All code review tools that are widely used today only address the microscopic understanding needs of reviewers – providing features such as diffing capabilities, inline commenting, or syntax highlighting. Reviewers lack important details at a macroscopic level in reviewing the change if they depend only on such tools [21].

Previous research has investigated differencing algorithms over source code [12, 16, 17] which seek the shortest edit distance between two source files represented as Abstract Syntax Trees (AST). Such algorithms extend string differencing algorithms [28] to compute an edit sequence composed of three basic edit operations, delete, insert and relabel. These operations are used to transform one tree into another with minimum cost, where the cost is given by the sum of the cost of each involved edit operation. The cost of the edit sequence with minimum cost is called the edit distance, which has been shown to be extremely beneficial in comprehending the nature of a code change [15].


```
+2 -3 src/attributes/classes.js

@@ -1,10 +1,9 @@
 define([
     "../core",
     "../var/rnotwhite",
-    "../var/strundefined",
     "../data/var/dataPriv",
     "../core/init"
-], function( jQuery, rnotwhite, strundefined, dataPriv ) {
+], function( jQuery, rnotwhite, dataPriv ) {

 var rclass = /[\t\r\n\f]/g;

@@ -128,7 +127,7 @@ jQuery.fn.extend({
                 }

             // Toggle whole class name
-            } else if ( type === strundefined || type === "boolean" ) {
+            } else if ( value === undefined || type === "boolean" ) {
                 if ( this.className ) {
                     // store className if set
                     dataPriv.set( this, "__className__", this.className );
```

Figure 1. A sample unified diff on GitHub.

At an architectural level, LSdiff [21] and DESIGNDIFF [40] leverage differencing algorithms at a macroscopic level to reduce the cognitive burden of analyzing architectural properties of code in code reviews. LSdiff proposes a program differencing technique that automatically identifies systematic structural differences as logic rules. DESIGNDIFF models the high level design differences resulting from every code revision in a software project. However, both tools express the differences they uncover in a textual format, which limits the ability of developers to discuss and collaborate on improvements that can be made to the system in a visual way. Relo [37] uses visual diagrams to illustrate code architecture and enables users to navigate code architecture in a dynamic fashion. However, it does not support comparing two versions of code and was not specifically designed for code reviews.

3 Striff Diagrams

To overcome difficulties faced by reviewers in understanding code at a macroscopic level, we introduce striff diagrams. Striffs leverage the basic premise surroundings the utility of line-wise code diffs at an architectural level, and encourage a more natural understanding of code changes through a "top-down" approach which more closely resembles the lens from which the system was designed and intended to be understood. The cognitive implications of striff diagrams are manifold: diagrams support communicating, capturing attention and grounding conversations [11]. They also reduce the cognitive burden of evaluating a design or considering new ideas [18]. Moreover, numerous empirical studies [10, 22] have also demonstrated benefits of using visual models in the context of software maintenance. A sample striff is illustrated in Figure 2.

Striffs extend the UML class diagram specification to help reviewers understand the context and impact of code submissions in code reviews. Standardized by the Object Management Group (OMG) in 1997, the Unified Modeling Language (UML) emerged as a de-facto industry standard for analysis and design modeling of software systems [31]. In most projects, UML models are the first artifacts to systematically represent a software architecture [23]. Moreover, studies have shown that developers prefer to communicate software design using graph and UML-like diagrams [10, 22].

Coloring Scheme: In popular code collaboration platforms like GitHub, unified diffs as illustrated in Figure 1 outline the differences between two files, often between an original file and proposed version of the file. In this format, any lines that have undergone modification are depicted beside unchanged lines both before and after. New lines of source code and deleted lines of source code as a result of the proposed file are annotated in green and red colors, respectively. Context to the patch is provided by the inclusion of unchanged lines and serves as a reference to locate the patch's position in a modified file.

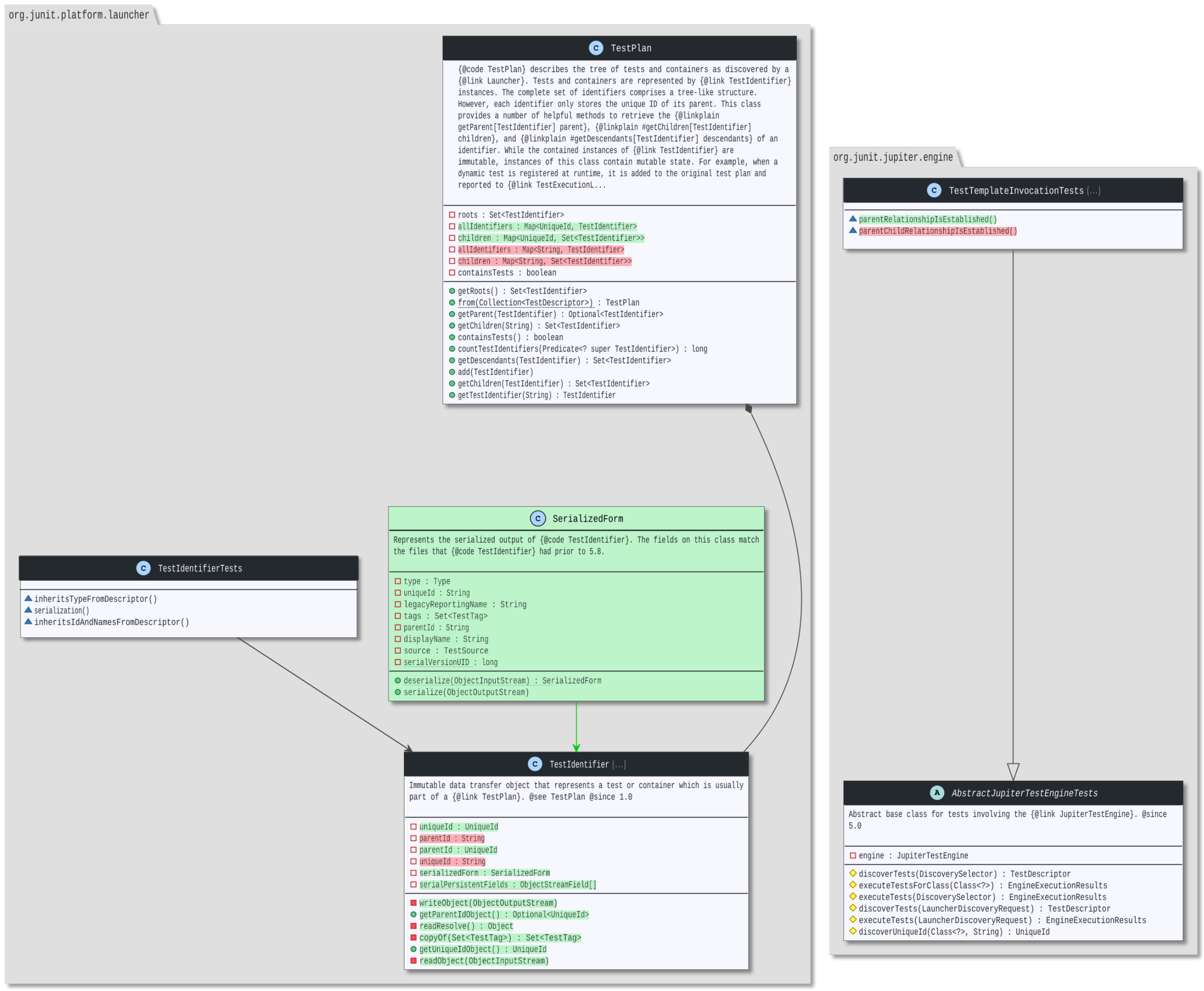
A striff depicts entities entering and leaving the system at an architectural level using the same coloring scheme. UML relationships, methods, fields, and objects that enter and leave the system as a result of the proposed code changes are annotated in green and red colors, respectively. Like regular line-wise diffs, unchanged entities are not annotated with any color and provide context to the visualization.

3.1 Managing Understandability in Striffs

Although different ways of organizing diagrams don't change the actual design, they influence how easy the model is to understand and interpret [24]. In the context of architectural component views, understandability is a critical aspect, as one of the main purposes of software architecture is to "... enable designers to abstract away fine-grained details that obscure understanding and focus on the "big picture: system structure, the interactions between components, ..." [19]. Architectural diagrams must successfully convey all the relevant concerns of the system at a high level to facilitate the understanding of relationships between the low level code and the higher level design of the system [5, 8]. To manage understandability in striffs, we impose the following constraints on diagrams:

1. Given a soft input size constraint ℓ, there can be at most $\ell + \ell/2$ components in the striff. Figure 2 is the result of running the proposed reverse engineering algorithm detailed in Section 4 with $\ell = 6$.
2. For components for which the set of child methods and fields exceed an input constraint m on the number of child components, only those child methods and fields which have been modified will be displayed, and the component name will be suffixed with "(...)" to indicate some details are intentionally left out. This parameter prevents striffs from overwhelming the reader with long lists of component children that are not essential. Figure 2 is the result of running the proposed reverse engineering algorithm detailed in Section 4 with $m = 15$.
3. Given that gaining a high-level context to unfamiliar code is a major difficulty in code reviews, object components in striffs can include documentation on their high level design where appropriate. This allows reviewers to understand the purpose and design of components in a code review more quickly.

When discussing striff diagrams, determining which context is important to the viewer while minimizing the size of the diagram constitute two opposing concerns. Without any context, reviewers will not understand what role the modified components play in the larger



```
org.junit.platform.launcher

TestPlan
{@code TestPlan} describes the tree of tests and containers as discovered by a
{@link Launcher}. Tests and containers are represented by {@link TestIdentifier}
instances. The complete set of identifiers comprises a tree-like structure.
However, each identifier only stores the unique ID of its parent. This class
provides a number of helpful methods to retrieve the {@linkplain
getParent[TestIdentifier] parent}, {@linkplain #getChildren[TestIdentifier]
children}, and {@linkplain #getDescendants[TestIdentifier] descendants} of an
identifier. While the contained instances of {@link TestIdentifier} are
immutable, instances of this class contain mutable state. For example, when a
dynamic test is registered at runtime, it is added to the original test plan and
reported to {@link TestExecutionL...
roots : Set<TestIdentifier>
allIdentifiers : Map<UniqueId, TestIdentifier>
children : Map<UniqueId, Set<TestIdentifier>>
allIdentifiers : Map<String, TestIdentifier>
children : Map<String, Set<TestIdentifier>>
containsTests : boolean
getRoots() : Set<TestIdentifier>
from(Collection<TestDescriptor>) : TestPlan
getParent(TestIdentifier) : Optional<TestIdentifier>
getChildren(String) : Set<TestIdentifier>
containsTests() : boolean
countTestIdentifiers(Predicate<? super TestIdentifier>) : long
getDescendants(TestIdentifier) : Set<TestIdentifier>
add(TestIdentifier)
getChildren(TestIdentifier) : Set<TestIdentifier>
getTestIdentifier(String) : TestIdentifier

SerializedForm
Represents the serialized output of {@code TestIdentifier}. The fields on this class match
the files that {@code TestIdentifier} had prior to 5.8.
type : Type
uniqueId : String
legacyReportingName : String
tags : Set<TestTag>
parentId : String
displayName : String
source : TestSource
serialVersionUID : long
deserialize(ObjectInputStream) : SerializedForm
serialize(ObjectOutputStream)

TestIdentifierTests
inheritsTypeFromDescriptor()
serialization()
inheritsIdAndNamesFromDescriptor()

TestIdentifier (...)
Immutable data transfer object that represents a test or container which is usually
part of a {@link TestPlan}. @see TestPlan @since 1.0
uniqueId : UniqueId
parentId : String
parentId : UniqueId
uniqueId : String
serializedForm : SerializedForm
serialPersistentFields : ObjectStreamField[]
writeObject(ObjectOutputStream)
getParentIdObject() : Optional<UniqueId>
readResolve() : Object
copyOf(Set<TestTag>) : Set<TestTag>
getUniqueIdObject() : UniqueId
readObject(ObjectInputStream)

org.junit.jupiter.engine

TestTemplateInvocationTests (...)
parentRelationshipIsEstablished()
parentChildRelationshipIsEstablished()

AbstractJupiterTestEngineTests
Abstract base class for tests involving the {@link JupiterTestEngine}. @since
5.0
engine : JupiterTestEngine
discoverTests(DiscoverySelector) : TestDescriptor
executeTestsForClass(Class<?>) : EngineExecutionResults
executeTests(DiscoverySelector) : EngineExecutionResults
discoverTests(LauncherDiscoveryRequest) : TestDescriptor
executeTests(LauncherDiscoveryRequest) : EngineExecutionResults
discoverUniqueId(Class<?>, String) : UniqueId
```

Figure 2. A sample striff diagram[1].

system. Given too much context, the diagram becomes too large and the reviewer is overwhelmed. Platform likes GitHub approach this problem in line-wise diffs by simply adding three lines of context code before and after any modified line as illustrated in Figure 1. This paper solves the issue of balancing context and diagram size by modelling the diagram as a weighted graph and applying a community detection algorithm recursively on the graph to eventually produce a suitable set of subgraphs representing more readable diagrams.

4 Algorithm

Striff-lib[2] is an open source Java library that implements the proposed algorithm detailed in this section to reverse engineer striff diagrams from source code.

Striff-lib can be integrated as a browser plugin or as part of a CI pipeline to generate striff diagrams with the goal of facilitating engineers in performing code reviews. A high level outline of the algorithm is depicted in Figure 4. Given two sets of source files corresponding to the original and proposed software systems in the context of a code review, striff-lib generates a set of graphs representing the structural differences between the two systems. These structural differences are then

[1]Generated for GitHub pull request at https://github.com/Zir0-93/junit5/pull/1/files.

[2]Source code available at https://github.com/hadii-tech/striff-lib.

visualized using PlantUML[3], a Java library for drawing UML diagrams.

Step 1. Generate Source Code Models. The first step leverages the clarpse Java library[4] to parse source code of the original and proposed versions of the software with the goal of constructing a high level, object oriented based abstraction of entities that realize the technical design and implementation of the system. The result of this parsing step is referred to as a *source code model*, which is a collection of high level object and interface components. We refer to such components as "*major*" components for the rest of this paper. Every major component in clarpse maintains a collection of child components which correspond to field and method components, referred to as "*minor*" components. In total, clarpse uses four main types of components in representing source code models: objects, interfaces, methods, and fields. The benefit of this approach is realized in the potential for clarpse to parse any language that supports object oriented design into a common, polyglot format represented by source code models over which software engineering tools can be developed in a consistent manner. For this reason, striff-lib was developed using clarpse, and is currently able to generate striffs over code written in a variety of programming languages including Java and Go.

A source code model in clarpse is a collection of major components which can be represented as a set of trees $T = \{t_1, t_2, ..., t_n\}$. Each tree is represented by $t = (C, A)$, where $c \in C_t$ is a vertex and represents a component generated by clarpse while $a \in A_t$ is an edge. We denote C_t as the vertex set and A_t as the edge set of a tree t in T. Every vertex c identifies a component as a function of its structural representation based on the source code. As a result, each vertex is unique amongst the set of components of each tree in T as expressed by the following:

$$\forall t \forall t' (t' \in T \mid t' \neq t \implies C_t \cap C_{t'} = \emptyset) \tag{1}$$

Additionally, we assume that the root node r of every tree in T always corresponds to a major component, and must exist in every edge in that tree such that $\forall a(a_{ij} \in A_t \implies c_i = r \oplus c_j = r)$. Because major components in clarpse only contain a single level of direct children corresponding to minor components, the maximum height of any given tree is given by the function $h : T \to \{0, 1\}$. Consequently, each tree in T represents a major component along with it's child components. The output of this step are two source code models represented as sets of trees T_o and T_p, corresponding to the set of trees representing the original and proposed versions of the software system respectively. Note that in the context of a code review, we expect that many of the components in the trees of T_o and T_p to be equivalent since they are a function of the structural position of that component based on the source code.

Step 2. Generate Graph Representations. Components in clarpse store a list of a type dependencies on other components which can be analyzed to generate a list of UML compliant class relationships for the entire source code model. Given a set of trees T from the previous step, we create an undirected graph $G = (V, E, \mu, \xi)$ where $v \in V$ is a node and $e \in E$ is an unordered pair representing an edge. The function μ maps each vertex to a tree $\mu : V \to T$. Similarly, ξ is a function $\xi : E \to U$, where U denotes the set of edge types corresponding to the set of UML class relationships as specified in Table 1. Let *isRelated* be a function that indicates whether or not a given pair of major components represented by two trees are related by a UML class diagram relationship. The set of edges E in V are then constrained to represent such relationships between pairs of trees in T by the expression $\forall e(e_{ij} \in E \implies isRelated(\mu(v_i), \mu(v_j)))$. The output of this step are graphs G_o and G_p that represent the graph formed from analyzing source code models T_o and T_p respectively.

Step 3. Diff Graph Representations. We first generate a set of trees T_m which represents the result of merging T_o and T_p together as detailed in Algorithm 1. Let q be a vertex and T a set of trees. We maintain a function $\delta(q, T)$ which returns a set X consisting of all the trees in T for which q is the root vertex. Therefore, X is a subset of T as given by $X \subseteq T$, and each tree in X satisfies the expression $\forall x(x \in X \implies r_x = q)$, where r_x is the root node of the tree x. Because each vertex is unique in any given set of trees T as expressed by (1),

[3]https://plantuml.com

[4]Source code available at https://github.com/hadii-tech/clarpse.

Type	Realization	Generalization	Composition	Aggregation	Association
Weight	6	6	4	3	1

Table 1. A list of UML class relationships and their associated weights implemented in the proposed algorithm.

the output set of this functions is maximally equal to one: $|\delta(q, T)| \leq 1$.

Algorithm 1: Merging T_o into T_p

```
Result: $T_m$
Input: $T_o$, $T_p$
for $v \in V_o$ do
    for $t_o \in \mu_o(v)$ do
        $Z := \delta(r_{t_o}, T_p)$
        if $Z = \emptyset$ then
            $T_p := T_p \cup \{t_o\}$
        else
            $C_z := C_z \cup C_{t_o}$
        end
    end
end
$T_m := T_p$
```

We now define a graph $G_m = (V_m, E_m, \mu_m, \xi)$, where $V_m = V_p \cup V_o$, $E_m = E_p \cup E_o$, and μ_m is a function which maps each vertex to tree $\mu : V_m \longrightarrow T_m$ and is defined as:

$$\mu_m(v) = \begin{cases} \emptyset & \text{if } v \notin V_o \wedge v \notin V_p \\ t_o & \text{if } v \in V_o \wedge v \notin V_p \\ t_p & \text{otherwise} \end{cases} \tag{2}$$

Graph G_m now represents the result of merging G_o into G_p. As a result, the set of inserted and removed edges between G_o and G_p corresponds $E_i = E_p - E_o$ and $E_r = E_o - E_p$ respectively. Additionally, we define C_i and C_r which correspond to all the inserted and removed vertices by the following equations:

$$C_i = \{c | c \in \cup_{C_{tp}, t_p \in T_p} \wedge \ c \notin \cup_{C_{to}, t_o \in T_o}\} \tag{3}$$

$$C_r = \{c | c \notin \cup_{C_{tp}, t_p \in T_p} \wedge \ c \in \cup_{C_{to}, t_o \in T_o}\} \tag{4}$$

Step 4. Filter Graph. At this point in the algorithm, G_m represents the merged architectural view of the original and proposed source code models, while C_i, C_r, E_i, and E_r represent components and relationships that enter and exit the system. These entities are sufficient for drawing a striff diagram; however, there is currently no limit on $|V_m|$ which allows for extremely large diagrams. Graph visualization and automatic layout are important issues that have tremendous impact on the

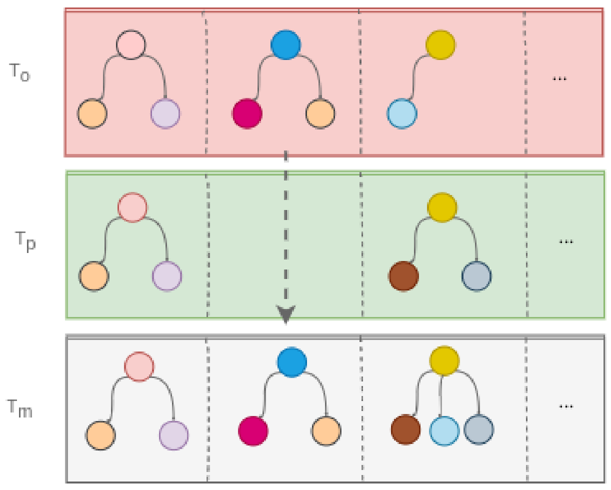


Figure 3. A visualization of merging T_o into T_p as detailed in Algorithm 1.

readability of diagrams [10, 41, 42]. To solve this problem, we generate a sets of graphs D given a soft limit ℓ according to the following definition:

$$\forall d(|d| \leq (\ell + \frac{\ell}{2}) \wedge V_d \subseteq V_m \wedge E_d \subseteq E_m) \tag{5}$$

We also ensure that every graph in D has at least one key entity within it by satisfying one or both of (6) or (7) below.

$$\exists v_d(|c_{\mu_m(v_d)} \cap (C_i \cup C_r)| \geq 1) \tag{6}$$

$$\exists e_d(e_d \in (E_i \cup E_r)) \tag{7}$$

We leverage a community detection algorithm to split G_m into a set of subgraphs D. One of the classic approaches leveraged by such algorithms is to develop a partition of the vertex set that maximizes an optimization function, of which the modularity function [30] is a well-known example. In this function, the strength of an edge between two vertices is calculated based on the probability of finding such an edge in a random model maintaining the same degree distribution as the original graph.

Based on this understanding of modularity, the strength of a given community does not increase significantly when an edge between two vertices with high degrees is discovered in comparison to an edge discovered between vertices of low degrees. This is because the latter

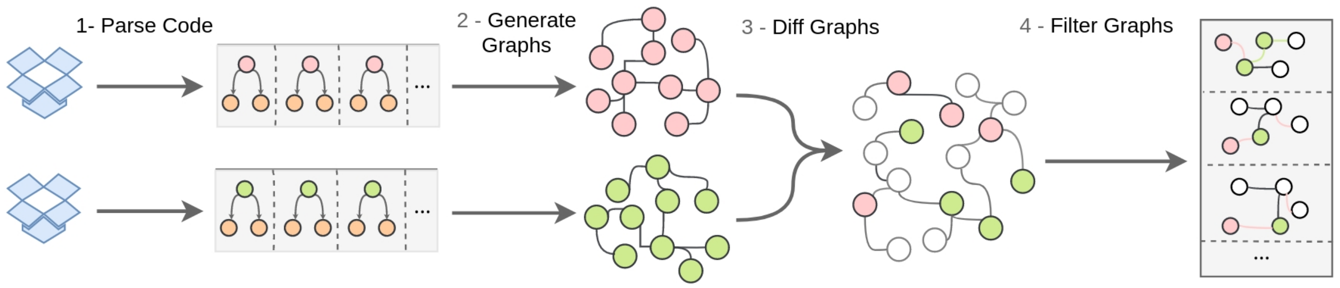


Figure 4. The proposed algorithm for generating striff diagrams.

type of edge is much more surprising to find, and therefore contributes to the modularity and strength of its community more significantly than the former. Formally, modularity is a scalar value which is defined as:

$$Q(C) = \frac{1}{2m} \sum_{ij} [A_{ij} - \frac{k_i k_j}{2m}] \delta(c_i, c_j) \qquad (8)$$

where A_{ij} represents the edge weight between vertices i and j. The sum of the weights of the edges attached to nodes i and j are represented by k_i and k_j, while the sum of all the edge weights in the graph is denoted by m. Finally, c_i and c_j are the communities of the nodes and δ is Kronecker delta function.

The Louvain algorithm [7] is an approach based on modularity optimization that is employed in this paper due to its applicability to numerous types of graphs and wide spread availability in the form of open source implementations. Initially, the Louvain algorithm considers each vertex as a community, and consists of two main steps:

- **Step A**: Vertices are moved between communities so that the modularity of G is increased by each move. Step A is started by creating $|V_m|$ communities where each vertex is in its own community. As long as the modularity can be increased, vertices are sequentially reassigned to new communities. The choice of communities for a vertex is based on the community that yields the largest modularity increase.
- **Step B**: A new graph is created from the newly partitioned graph. Identified communities form the set of new vertices, and edges between vertices in different communities are aggregated into a new set of edges. Step A is then applied to this newly created graph. The two steps above are repeated until no further improvement in modularity is obtained.

Given that the Louvain algorithm produces subgraphs that maximize edge density with respect to other graphs, the weighting function $\omega(e) \rightarrow \chi(e) + \zeta(e)$ employed in our study assigns edge weights according to two important properties:

- χ is a function that maps edge e to the associated weight of that edge type based on Table 1. Therefore, edge weights are assigned based on the level of relatedness implied by the UML relationship represented by that edge. For example, an edge representing a UML realization relationship between components is given a much higher weighting than an edge representing a basic association relationship between two components. The reason behind this weighting scheme is a product of a key goal of striff diagrams in conveying a high level architectural overview of code. As a result of generating subgraphs that maximise edge-density with respect to other graphs, the Louvain algorithm will produce graphs representing striff diagrams that focus on the important object oriented properties of the code.
- ζ is a function that outputs a value based on whether or not e represents an edge in which either or both of its vertices map to a component in C_i or C_r as given by:

$$\zeta(e_{ij}) = \begin{cases} 2 & \text{if } \mu_m(v_i) \in C_i \bigwedge \mu_m(v_i) \in C_r \\ 1 & \text{if } \mu_m(v_i) \in C_i \oplus \mu_m(v_i) \in C_r \\ 0 & \text{otherwise} \end{cases} \qquad (9)$$

Therefore, this weighting scheme encourages the Louvain algorithm to produce subgraphs that maximize the number of vertices representing modified components as well.

Through the use of the Gephi Java library[5], the Louvain algorithm is recursively executed to produce a set

[5]Source code available at https://github.com/gephi/gephi.

of subgraphs that satisfy expressions (5), (6) and (7). That is, in each iteration, any graphs that do not satisfy one of (6) or (7) after running the Louvain algorithm are discarded. For the remaining graphs in the result set that do not satisfy the size constraints expressed by (5), the algorithm is run once again on those graphs in a similar fashion until suitable graphs are obtained.

5 Discussion

Line-wise diffs of source code can be found in almost every code collaboration platform, and is a tool most developers' employ in reviews. Given the success of such diffs in analyzing microscopic details of source code, striffs are built upon the same fundamental idea, but focus on the macroscopic level details of code. Figure 2 illustrates a striff generated as part of our case study by the proposed algorithm for a GitHub pull request in the Junit5[6] repository. The author's intention was to reduce the memory footprint of test executions in the project. To accomplish this, a new *SerializedForm* class was introduced to serialize test execution data, and the existing $TestIdentifier$ and $TestPlan$ classes were refactored as required to support storing and fetching executions. Additionally, unrelated to the goal of reducing the memory footprint, the author also made modifications to the interface of the $TestTemplateInvocationTests$, all of which is clearly captured by the striff in Figure 2.

We hope to empirically evaluate the ability of striffs to satisfy the high level information needs of reviewers in code reviews in the future. Additionally, many of the parameters discussed in the proposed algorithm can be tuned according to the reviewers needs. The impact on varying these parameters on generated striffs needs to be researched further. For now, we believe the constraints outlined in Section 3.1 on striff diagrams constitute a sufficient starting point for further research in improving the understandability and usefulness of striffs. First, striffs contain high level component documentation when available, which is why the $TestPlan$ components' documentation can be read in Figure 2. The UML class diagram specification does not allow for such information to be included in diagrams, which places a higher cognitive burden on reviewers having to figure out important design details of components. Next, the $TestIdentifier$ and $TestTemplateInvocationTests$ class have over 15 child methods and fields which have been purposefully excluded from the diagram. This makes the diagram more readable and prevents reviewers from being overwhelmed with too many details.

The use of community detection algorithms in generating readable diagrams was also proposed in this study. Reverse engineering algorithms that produce visualizations of code have typically suffered from containing too much information, both of a primary and contextual nature [10]. By weighting graph edges representing UML relationships according to their importance in an object oriented context, the proposed algorithm generates diagram layouts that optimize the display of context and key architectural properties of the software system. Additionally, by running the algorithm recursively on the output set of graphs, graphs that are sized desirably within a set threshold are eventually produced. Moreover, this approach can be applied to generate graphs that are meaningful in domains outside of object oriented systems. For example, it would be possible to leverage the UMLsec [20] specification to assign edge weights based on the level of security between the two source code components connected by a given edge. Consequently, the proposed algorithm might generate diagrams that represent areas of the source code that are less secure from a security perspective.

[6]https://github.com/junit-team/junit5

References

[1] [n.d.]. Where the world builds software. https://github.com/

[2] K. K. Aggarwal, Y. Singh, and J. K. Chhabra. 2002. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318)*. 235–241. https://doi.org/10.1109/RAMS.2002.981648

[3] T. Baum, K. Schneider, and A. Bacchelli. 2017. On the Optimal Order of Reading Source Code Changes for Review. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 329–340. https://doi.org/10.1109/ICSME.2017.28

[4] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern Code Reviews in Open-Source Projects: Which Problems Do They Fix?. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 202–211. https://doi.org/10.1145/2597073.2597082

[5] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. 1994. Program Understanding and the Concept Assignment Problem. *Commun. ACM* 37, 5 (May 1994), 72–82. https://doi.org/10.1145/175290.175300

[6] Christian Bird and Alberto Bacchelli. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. In *Proceedings of the International Conference on Software Engineering* (proceedings of the international conference on software

engineering ed.). IEEE. https://www.microsoft.com/en-us/research/publication/expectations-outcomes-and-challenges-of-modern-code-review/

[7] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (Oct 2008), P10008. https://doi.org/10.1088/1742-5468/2008/10/p10008

[8] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543 – 554. https://doi.org/10.1016/S0020-7373(83)80031-5

[9] Celia Chen, Reem Alfayez, Kamonphop Srisopha, Lin Shi, and Barry Boehm. 2016. Evaluating Human-Assessed Software Maintainability Metrics. In *Software Engineering and Methodology for Emerging Domains*, Lu Zhang and Chang Xu (Eds.). Springer Singapore, Singapore, 120–132.

[10] Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. 2007. Let's Go to the Whiteboard: How and Why Software Developers Use Drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '07)*. Association for Computing Machinery, New York, NY, USA, 557–566. https://doi.org/10.1145/1240624.1240714

[11] Herbert H. Clark and Edward F. Schaefer. 1989. Contributing to discourse. *Cognitive Science* 13, 2 (1989), 259 – 294. https://doi.org/10.1016/0364-0213(89)90008-6

[12] Georg Dotzler and Michael Philippsen. 2016. Move-Optimized Source Code Tree Differencing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 660–671. https://doi.org/10.1145/2970276.2970315

[13] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik. 2019. Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 49–60. https://doi.org/10.1109/SANER.2019.8668024

[14] M. E. Fagan. 1976. Design and Code Inspections to Reduce Errors in Program Development. *IBM Syst. J.* 15, 3 (Sept. 1976), 182–211. https://doi.org/10.1147/sj.153.0182

[15] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[16] B. Fluri, M. Wursch, M. Pinzger, and H. Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

[17] M. Hashimoto and A. Mori. 2008. Diff/TS: A Tool for Fine-Grained Structural Change Analysis. In *2008 15th Working Conference on Reverse Engineering*. 279–288. https://doi.org/10.1109/WCRE.2008.44

[18] Edwin Hutchins. 1996. *Cognition in the Wild (Bradford Books)*. The MIT Press.

[19] J. R. Josephson and M. C. Tanner. 1994. The Role of Explanatory Relationships in Strategies for Abduction. *IEEE Intelligent Systems* 14, 03 (May 1994), 54–59. https://doi.org/10.1109/64.311280

[20] Jan Jürjens. 2002. UMLsec: Extending UML for Secure Systems Development. In *UML 2002 — The Unified Modeling Language*, Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 412–425.

[21] Miryung Kim and David Notkin. 2009. Discovering and Representing Systematic Code Changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 309–319. https://doi.org/10.1109/ICSE.2009.5070531

[22] Rainer Koschke. 2003. Software Visualization in Software Maintenance, Reverse Engineering, and Re-Engineering: A Research Survey. *Journal of Software Maintenance* 15, 2 (March 2003), 87–109. https://doi.org/10.1002/smr.270

[23] C. F. J. Lange, M. R. V. Chaudron, and J. Muskens. 2006. In practice: UML software architecture and design description. *IEEE Software* 23, 2 (2006), 40–46. https://doi.org/10.1109/MS.2006.50

[24] S. Lee, Y. Kwon, and J. Hwa. 2009. Hierarchical Understandability Assessment Model for Large-Scale OO System. In *2009 16th Asia-Pacific Software Engineering Conference (APSEC 2009)*. IEEE Computer Society, Los Alamitos, CA, USA, 11–18. https://doi.org/10.1109/APSEC.2009.60

[25] L. MacLeod, M. Greiler, M. Storey, C. Bird, and J. Czerwonka. 2018. Code Reviewing in the Trenches: Challenges and Best Practices. *IEEE Software* 35, 4 (July 2018), 34–42. https://doi.org/10.1109/MS.2017.265100500

[26] R. Mo, Y. Cai, R. Kazman, and L. Xiao. 2015. Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. In *12th Working IEEE/IFIP Conference on Software Architecture*. 51–60. https://doi.org/10.1109/WICSA.2015.12

[27] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. 2019. Architecture Anti-patterns: Automatically Detectable Violations of Design Principles. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2910856

[28] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* 1, 2 (1986), 251–266. https://doi.org/10.1007/BF01840446

[29] M. V. Mäntylä and C. Lassenius. 2009. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering* 35, 3 (2009), 430–448. https://doi.org/10.1109/TSE.2008.71

[30] M. E. J. Newman and M. Girvan. 2004. Finding and evaluating community structure in networks. *Physical Review E* 69, 2 (Feb 2004). https://doi.org/10.1103/physreve.69.026113

[31] OMG. 2011. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1. http://www.omg.org/spec/UML/2.4.1

[32] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and Alberto Bacchelli. 2018. Information Needs in Contemporary Code Review. *Proceedings of the ACM on Human-Computer Interaction* 2 (2018), 1 – 27.

[33] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. 2012. How do professional developers comprehend software?. In *2012*

34th International Conference on Software Engineering (ICSE). 255–265. https://doi.org/10.1109/ICSE.2012.6227188

[34] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto. 2019. Automatically Assessing Code Understandability. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2901468

[35] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. 2017. Automatically assessing code understandability: How far are we? In *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 417–427. https://doi.org/10.1109/ASE.2017.8115654

[36] R. Schwanke, L. Xiao, and Y. Cai. 2013. Measuring architecture quality by structure plus history analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 891–900. https://doi.org/10.1109/ICSE.2013.6606638

[37] Vineet Sinha, David Karger, and Rob Miller. 2006. Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange, eclipse'05*, 187–194. https://doi.org/10.1145/1117696.1117701

[38] Mathupayas Thongmak and Pornsiri Muenchaisri. 2011. Measuring Understandability of Aspect-Oriented Code. In *Digital Information and Communication Technology and Its Applications*, Hocine Cherifi, Jasni Mohamad Zain, and Eyas El-Qawasmeh (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 43–54.

[39] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, and B. Vasilescu. 2018. "Automatically Assessing Code Understandability" Reanalyzed: Combined Metrics Matter. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 314–318.

[40] X. Wang, L. Xiao, K. Huang, B. Chen, Y. Zhao, and Y. Liu. 2020. DesignDiff: Continuously Modeling Software Design Difference from Code Revisions. In *2020 IEEE International Conference on Software Architecture (ICSA)*. 179–190. https://doi.org/10.1109/ICSA47634.2020.00025

[41] Colin Ware, David Hui, and Glenn Franck. 1993. Visualizing Object Oriented Software in Three Dimensions. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1* (Toronto, Ontario, Canada) *(CASCON '93)*. IBM Press, 612–620.

[42] Colin Ware and Peter Mitchell. 2008. Visualizing Graphs in Three Dimensions. *TAP* 5 (01 2008). https://doi.org/10.1145/1279640.1279642

[43] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng. 2016. Identifying and Quantifying Architectural Debt. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 488–498. https://doi.org/10.1145/2884781.2884822