Improving System Understanding via Interactive, Tailorable, Source Code Analysis

Vladimir Jakobac[1], Alexander Egyed[2], and Nenad Medvidovic[1]

[1] Computer Science Department, University of Southern California,
941 W 37th St, Los Angeles, CA 90089, USA
{jakobac, neno}@usc.edu
[2] Teknowledge Corporation, 4640 Admiralty Way, Suite 1010,
Marina Del Rey, CA 90292, USA
aegyed@teknowledge.com

Abstract. In situations in which developers are not familiar with a system or its documentation is inadequate, the system's source code becomes the only reliable source of information. Unfortunately, source code has much more detail than is needed to understand the system, and it disperses or obscures high-level constructs that would ease the system's understanding. Automated tools can aid system understanding by identifying recurring program features, classifying the system modules based on their purpose and usage patterns, and analyzing dependencies across the modules. This paper presents an iterative, user-guided approach to program understanding based on a framework for analyzing and visualizing software systems. The framework is built around a pluggable and extensible set of clues about a given problem domain, execution environment, and/or programming language. We evaluate our approach by providing the analysis of our tool's results obtained from several case studies.

1 Introduction

Adding new functionality to an existing software system starts with a process of understanding the system's architecture, i.e., its structure, behavior, and key non-functional properties [12, 13]. This becomes difficult in the case of large systems for which the documentation does not exist or is outdated. Many low-level details in the source code obstruct the process of creating a system's high-level, architectural abstraction, which aids in reasoning about the system.

A number of software "clustering" techniques have been developed to cope with this problem [9, 10, 11, 14] but these techniques fail to provide much rationale behind the architecture. This becomes particularly important if we consider that the source code may actually contain accidental or emergent functionality and relationships which are not intended by the system's developers. Furthermore, clustering approaches are not always effective tools for performing architectural recovery. For example, our experience [9] has shown that in layered systems these approaches do not actually recover the layers, but tend to "slice" across them since the clustering is usually based on the existence of strong coupling (inter-layer) relationships.

For this reason, we posit that architectural recovery, and software clustering in particular, need to be accompanied by a system understanding activity, which includes the use of semantic information before any syntactic dependencies are considered, and whose goal is to help engineers control the architectural recovery process, and identify and correct any inconsistencies therein. Various representations can be used to describe successive levels of system's abstractions. Incited by Perry and Wolf's observation [12] that the key architectural elements of a software system are (1) processing, (2) data, and (3) connecting, we have developed ARTISAn, a tool-supported, pluggable framework intended to aid program understanding and, ultimately, architectural recovery.[1] Our approach is based on both structural and semantic analysis, where various design- and implementation-level constructs, termed *clues*, are used to classify, label, and collapse the system's elements (e.g., classes) into the three major categories.

The ARTISAn framework is tailorable. It comprises replaceable components to accommodate the exact programming environment. For example, the framework is instantiated with different components for various programming languages or off-the-shelf "utility" technologies such as middleware. ARTISAn provides a rich, interactive web of information to an engineer, allowing her to add, remove, or change both the clues and other analysis rules (and then reapply them), manually relabel any analysis results (and then observe how that new information is affecting the rest of the system), enact "what if" scenarios to identify key relationships and dependencies in the system, all the while being able to "undo" any changes. ARTISAn can also be further tailored for situations in which the division of system elements into processing, data, and connection may be overly general.

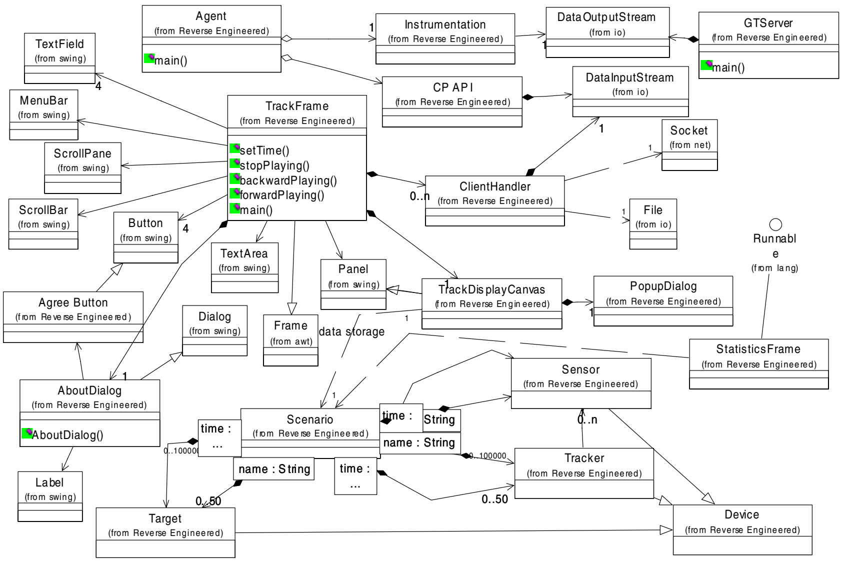
We have developed a prototype of ARTISAn targeted at Java systems. The tool is integrated with IBM Rational Rose®. We have applied ARTISAn on a number of third-party software applications to date, and report on those results.

This paper is organized as follows. Section 2 introduces an example application used to explain the approach, which is described in Section 3. In Section 4 we provide an evaluation of our approach. Section 5 presents related work and Section 6 summarizes our contributions and opportunities for future work.

2 Case Study

In this paper we are using a case study to illustrate our approach. The ANTS case study (Autonomous Negotiating agent TeamS) is an embedded agent negotiation system in which multiple, intelligent (software) agents negotiate over the best use of available resources (radars) to track a series of targets [2, 3]. The system was implemented in Java, and comprises over 200 classes developed by

[1] ARTISAn stands for Architectural Recovery via Tailorable, Interactive Source-code Analysis.



Agent (from Reverse Engineered) main()
Instrumentation (from Reverse Engineered)
DataOutputStream (from io)
GTServer (from Reverse Engineered) main()
TextField (from swing)
MenuBar (from swing)
ScrollPane (from swing)
ScrollBar (from swing)
Button (from swing)
TrackFrame (from Reverse Engineered) setTime() stopPlaying() backwardPlaying() forwardPlaying() main()
CP API (from Reverse Engineered)
DataInputStream (from io)
Socket (from net)
ClientHandler (from Reverse Engineered)
File (from io)
Runnable (from lang)
TextArea (from swing)
Panel (from swing)
TrackDisplayCanvas (from Reverse Engineered)
PopupDialog (from Reverse Engineered)
Agree Button (from Reverse Engineered)
Dialog (from swing)
Frame (from awt)
data storage
StatisticsFrame (from Reverse Engineered)
Sensor (from Reverse Engineered)
AboutDialog (from Reverse Engineered) AboutDialog()
Scenario (from Reverse Engineered)
time : ...
time : String
name : String
Tracker (from Reverse Engineered)
Label (from swing)
Target (from Reverse Engineered)
Device (from Reverse Engineered)
0..100000
0..50
0..n

Fig. 1. UML class diagram for the ANTS Visualizer system

several organizations. The main components of the system are *Agent, GTServer, CPAPI* and a real-time *Visualizer*. While we used the entire ANTS system to evaluate our approach, the illustrations used in this paper are limited to its non-trivial visualization subsystem. Figure 1 depicts the class diagram of the ANTS Visualizer subsystem, where Agent, GTServer, and CPAPI components are depicted as single classes due to their complexity. These components communicate over the network via TCP/IP. There are three different types of input devices: *Sensor, Target,* and *Tracker.* While sensors track targets, trackers use sensor data to estimate targets' locations. Each data item is stored in a new *Device* instance and it is the responsibility of the *Scenario* class to keep track of both the current state and the change history of all devices. Finally, *TrackFrame* is used with other GUI-based classes to process and visualize the data.

3 Approach

Our approach (Figure 2) comprises three steps that are initially performed sequentially but may then be revisited in any order by the user. The first step, termed *initial labeling*, results in a classification of individual elements into *processing* (P), *data* (D), and *communication* (C) [12] based on ARTISAn's *clues.* The result obtained during the initial labeling phase and a pluggable set of propagation *rules* provide input to the *propagation labeling* step. During this phase, some non-labeled elements become labeled (i.e,. classified as P, D, or C), based on the recognition of structural patterns and relationships with other, already labeled elements. Furthermore, this step also identifies possible structural inconsistencies among labeled elements and alerts the user about them. Initial labeling and propagation labeling result in an interpretation of the system that suggests the *purpose* of each of the system's individual elements.

Finally, during the *def-use analysis* phase, regions of related elements are identified based on invocation and inheritance relationships. The obtained regions distinguish between elements that are shared across regions and those that are exclusive to a region. The result of this phase is a system's *usage* view representation, which provides information on parts of the system that could be grouped together based on their usage scenarios.

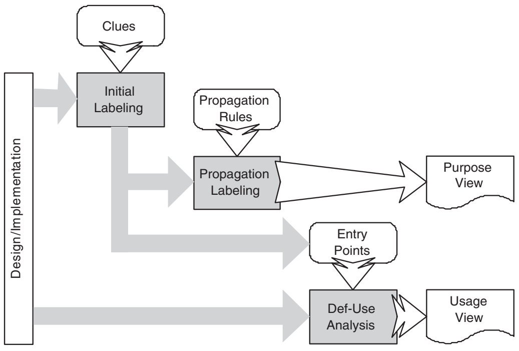


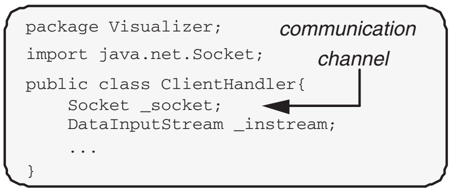
Fig. 2. The ARTISAn framework

Individually, the purpose and usage views provide the user with a classification of elements and their grouping based on usage analysis, respectively. These two views also complement each other. For example, if some unlabeled elements from the purpose view end up belonging to the same region with labeled elements of a single type, then one can surmise the purpose of the unlabeled elements. In total, our approach gives the user a better understanding of the system, and an opportunity to faster locate its parts that are of particular interest (e.g., for maintenance purposes).

The remainder of this section provides the rationale of our approach and describes each of the steps depicted in Figure 2 in more detail.

3.1 ARTISAn Clues and Initial Labeling

At the most general level, software systems integrate *processing* elements that exchange *data* via *communication (connecting)* elements [12]. By determining the type of a system element, one can distinguish elements with application-specific functionality from those with application-independent functionality. Typically, processing elements provide application-specific functionality as they implement the system's requirements. On the other hand, communication elements typically provide application-independent interaction facilities. In Java, for example, classes interact by invoking each other's methods and/or sharing data through public variables, regardless of the classes' functionality. In addition, a number of off-the-shelf communication elements (e.g., middleware) are available. A useful starting point in understanding the source code of a system is thus in the reusable, application-independent nature of its communication elements. Similarly, data elements only contain the information that is used or transformed by processing elements. Therefore, by identifying and then abstracting away data



```
package Visualizer;
import java.net.Socket;
public class ClientHandler{
    Socket _socket;
    DataInputStream _instream;
    ...
}
```

Fig. 3. Communication clue

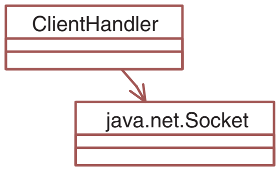


Fig. 4. An excerpt from the ANTS Visualizer class diagram

elements, the reasoning about the system is improved (e.g., applications built using the pipe-and-filter architectural style).

Software systems are generally described by their design or implementation models (e.g., class diagrams). Often, the models are too detailed, so that their understanding becomes obscured. In ARTISAn, constituent elements of these models (e.g., classes) are at first classified into the three aforementioned categories (Processing, Communication, and Data), providing an engineer the opportunity to quickly gain an overview on the purpose of individual elements and the structure of their composition. The process of classifying system elements into one of the three categories is termed *labeling*. The labeling is based on various design and implementation snippets, termed *clues*. Clues carry syntactic, semantic, and possibly domain-specific information, which is searched for in a system's model.

Figure 3 depicts a segment of Java source code from the ANTS system that illustrates how clues are identified. In the example, there is an attribute _socket that declares a use of the standard network socket library *java.net.**. This information is a clue to the existence of a communication channel, which is directly used by this class. We should note that clues could also be identified from a system's graphical model representation (e.g., its class diagrams), which enables the potential easy integration of our approach with already available visualization tools. For example, the same communication clue exists in Figure 4, which represents an excerpt from the ANTS Visualizer class diagram.

Each clue is represented as a 4-tuple: (1) *Impact*: if found, what is the meaning of a clue, i.e., is the element of type P, or C, or D? (2) *Base*: describes the software artifact in which we expect to find a clue (e.g., method, class, procedure); (3) *Condition*: a condition that must be satisfied for a clue to be found (e.g., a class whose name starts with "*java.net*"); (4) *Language*: the programming language to which clue applies. For example, the Java "Socket" clue, described above, is defined as *(C, class, class.name = "java.net.Socket", Java)*.

Although it is difficult to automatically understand the exact purpose of a processing element, it is possible to recognize such an element's existence through source code declarations. There are several clues that could be used in the detection of processing elements. For example, all classes that implement the static method *main*, or inherit from the library class *java.lang.Thread*, or implement the *java.lang.Runnable* interface are likely to be processing elements. Applied

Table 1. Domain-independent clues

Impact	Base	Condition
P	class.method	name="main"
	class	implements="java.lang.Runnable"
		extends="java.lang.Thread"
		extends.startsWith("java.awt")
C	class	name.startsWith("java.io")
		name.startsWith("java.net")
D	class	parent.type="D"
		no methods other than constructor(s)
		extends="java.lang.Exception"
		name="java.util.Vector", "java.util.Hashtable", ...
		implements="java.net.Serializable"

to our case study example, this means that all classes in a model that have the "main" method are classified as processing classes, such as *TrackFrame*, *Agent*, and *GTServer* in Figure 1. In a similar way, *ClientHandler* is labeled as a processing element since it implements the *Runnable* interface. Additionally, system elements that provide the GUI functionality are considered as a subcategory of processing elements. They are easily recognized based on the use of dedicated GUI libraries (e.g., *java.awt.**). Similarly, ARTISAn defines data-element clues. For example, all classes with only a constructor method and non-empty attribute list are likely to serve as data stores.

The clues described above all belong to ARTISAn's extensible and pluggable set of clues. We expect each programming paradigm and language, domain, and/or application to have their own set of clues. Those clues would be identified by language and domain experts and integrated into the framework. ARTISAn distinguishes between the following clue categories: (1) *Domain-independent* clues, such as the *Socket* class being classified as C, or a class with no methods being recognized as D; (2) *Domain-specific* clues, e.g., in case a system is built on top of a known middleware platform (e.g., an element of the *Siena* middleware is classified as C and the classes having access to the *Siena* are appropriately marked); and (3) *Application-specific* clues, such as a class of name "*jigsaw.Resource*" in the Jigsaw Web server being recognized as D.

Table 1 lists the Java-based clues that we have used to evaluate our approach. All the clues listed in Table 1 fall into the category of domain-independent clues. They can be applied to a wide range of (Java) software systems and can be naturally complemented by the more narrowly applicable domain- or application-specific clues.

We should note that, in general, ARTISAn does not require application-specific source code to follow any pre-defined naming conventions. However, ARTISAn provides support for using naming conventions in situations where such a collection of rules is available, such as with standardized libraries.

ARTISAn uses different colors to represent different classes' labels on the class diagram, or combinations of these colors if a class has more than one label.

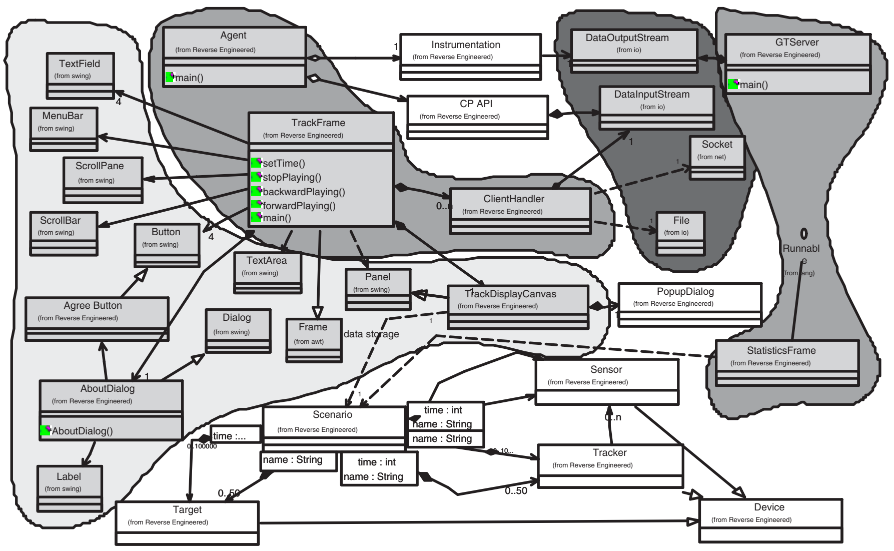


Fig. 5. The resulting diagram after the initial labeling step

Unlabeled classes remain transparent. However, in order to increase the readability of the illustrations in this paper, we additionally edited the diagrams by gray-scaling the labeled classes and drawing filled boundaries around classes of the same color. Figure 5 depicts the ANTS Visualizer diagram obtained after the initial labeling step is performed. Classes such as *Agent*, *TrackFrame,* or *GTServer*, which are inside the medium-gray boundary, indicate processing components. Classes such as *DataOutputStream* or *Socket,* inside the dark-grey boundary, indicate communication-based connectors. Finally, classes bounded by the light-grey shape indicate GUI elements, i.e., a subcategory of processing elements.

It should be noted that the clues are designed in such a way that applying them may identify one or more categories that an element belongs to, but also one or more categories to which the element does not belong. We refer to the former as an inclusion set, and to the latter as an exclusion set. For example, the *Socket* class will have C in its inclusion set, and P and D in its exclusion set. In other words, while this element is labeled as communication element, we also know that it cannot be processing or data. This information is of particular importance during the propagation labeling phase.

3.2 ARTISAn Rules and Propagation Labeling

It is very likely that not all elements in a system can be labeled based on ARTISAn clues. However, the existing knowledge about a system could be used to reason additionally about the system. Information obtained from clues can be

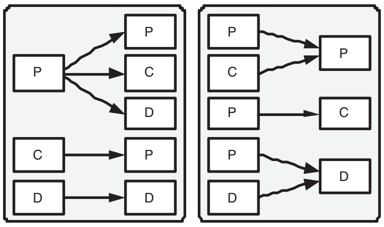


Fig. 6. Propagation scenarios

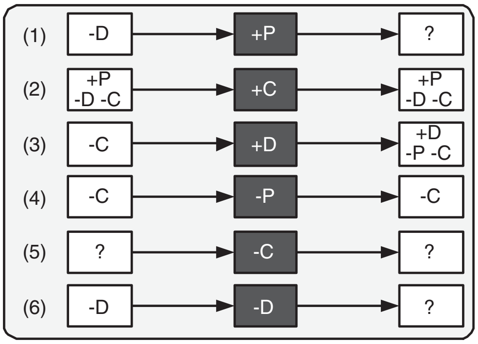


Fig. 7. Propagation rules

propagated from labeled elements to their neighboring elements (both labeled and unlabeled) when certain conditions are satisfied.

For example, as a result of an application of a communication clue, the ANTS *DataOutputStream* class is labeled as C, while no clue could be applied to the *Instrumentation* class (5). However, vital to the understanding of *Instrumentation* is its relationship to *DataOutputStream*. This relationship is a UML association and it indicates that *Instrumentation* uses *DataOutputStream*. Based on this observation we can deduce that *Instrumentation* cannot be a data element because a data element, by its definition, is not capable of such processing (i.e., a data element is perhaps allowed to do minor processing such as data checking, conversion, and storage, but not application-wide communication). Furthermore, *Instrumentation* is not an off-the-shelf communication element because we do not expect two such elements in a domain-independent system, such as ANTS, to be able to integrate and call each other directly. Thus, it follows that *Instrumentation* must be a processing element.[2]

We refer to this kind of reasoning as *clue propagation*. Clue propagation serves as a basis for the ARTISAn *propagation rules*. The pluggable set of propagation rules and the result obtained during the initial labeling phase provide input to the *propagation labeling* step (Figure 2). During this phase, some non-labeled elements become labeled, based on the application of the propagation rules. Propagation rules are derived from structural and interaction patterns involving different types of elements. Figure 6 illustrates these patterns.

The left-hand side of Figure 6 shows that a processing (P) element could call other processing, communication (C), and data (D) elements. In other words, there are no restrictions on what type of elements might be called by a processing element. On the other hand, our experience has shown that in case of domain-independent applications off-the-shelf communication elements usually

[2] This discussion is based on the understanding that there are no application-specific communication elements in the ANTS system. If there are, then they would be recognized as processing elements using ARTISAn's existing clues.

do not invoke any other element (e.g., socket-based communication) but if they do, the invoked elements could only be processing elements (e.g., COM-based communication element). We should note here that some technologies that are used to bridge across different computing platforms (e.g., the Java to COM bridge) may involve communication elements calling other communication elements. However, in those cases we would be dealing with specialized solutions that would allow us to recognize such situations on a case-by-case basis. Furthermore, these cases would be amenable to capture by specialized domain- or application-specific propagation clues and rules, which would result in an appropriate identification and labeling of all such elements. Finally, data elements are expected to be passive entities that may perform some rudimentary internal processing, but are otherwise not interacting with processing or communication elements. To describe the propagation rules in a more formal way, we will use the right-hand side of Figure 6, which is a transpose of its left-hand side (e.g., only P or C can call P).

Based on the caller-callee relationships in Figure 6, we can deduce six propagation rules, which are depicted in Figure 7. For example, rule 1 in Figure 7 states that if an element is known to be a processing element (denoted by +P in the middle box), then all elements that call it (its callers) cannot be data elements (denoted by D). The rationale for this is as follows: from the right-hand side of Figure 6 we know that either a P or a C can call another P. This implies that the caller cannot be D. Since we do not know whether the actual caller is P or C, we only write that it is not D. In this way we avoid having to make an early (but possibly incorrect) decision. The question mark in the right-hand column of rule 1 indicates that we cannot say anything about the elements being called by that element (its callees). Similarly, if an element is known not to be a processing element (-P), as in rule 4, then neither the caller nor the elements being called can be communication elements. This rule is again derivable from Figure 6. If an element is not P then it is either C or D. We know that C can be called only by P, and that D can be called by P or D. It follows that C or D can be called by at most the union of their callers, which is P or D. Since we do not know whether it is P or D, we simply write that it is not C. All other rules can be derived in a similar way.

As a result of the propagation labeling step, two additional classes in the ANTS Visualizer were recognized as processing elements: *Instrumentation* and *CPAPI.*

The algorithm for applying propagation rules is based on the changes in the inclusion and the exclusion sets for each of the system's elements. All elements are being processed, and as long as there are changes in any of the two sets (e.g., an unlabeled element becomes labeled, or a processing element becomes classified as non-connecting element), an appropriate propagation rule is run. Since the inclusion and exclusion sets for each element are finite, it is obvious that this algorithm terminates. Its running time is linear, because no decisions are ever undone (no backtracking).

This step also provides support for identifying any potential rule conflicts. For example, if a class is identified as a processing element through one propagation rule, but also as not a processing element using another rule, then either the clue or one of propagation rules was erroneous. Conflicts are easily identifiable due to their simple implementation representation (+P and -P) and ARTISAn reports all inconsistencies to the user. At that point, the user has the choice to manually label the elements if they are of a known type, ignore the discovered conflict (e.g., in case when a helper class of known functionality has conflicting labels), or use that information to modify the set of clues, and rerun the propagation labeling step. In the last case, both the user and the tool are "learning" about new clues that could be used for other systems.

3.3 Def-Use Analysis

The next step in our approach is the identification of regions, i.e., groups of system elements that are closely related, or independent of other parts of the system. To this end, we adapt *def-use analysis*. Def-use analysis is an approach that has already been used in literature and illustrates the use of dominance analysis for identification of regions of related modules [8, 10]. These regions indicate parts of a system that are exclusively used by its other part(s) and those that are shared. Each of the identified regions has an *entry point*, which is a module where processing starts (e.g., a class with the main() method). Entry points in ARTISAn are obtained from the initial labeling step (Figure 2). Those are all elements that satisfy the "main" clue, but also include elements that are able to create a new processing thread. The rationale for this lies in the fact that systems often spawn their own subsystems by creating separate processing threads. We can identify spawning using clues which were discussed previously.

In addition, ARTISAn supports a richer set of relationships among elements by analyzing class inheritance together with class association and dependency relationships. For example, the *Tracker* class inherits from the *Device* class, which makes the *Tracker* able to invoke the methods of the *Device*. Furthermore, if there is a class that declares a variable of type *Device*, that variable is then able to hold either an instance of *Device* or any of its subclasses (e.g., *Tracker*). This means that the variable holder class can invoke any of subclasses' methods, which is interpreted in ARTISAn as another type of calling relationship.

The information about regions enables an engineer to more easily recognize system elements that belong together. The *usage view* thus complements the *purpose view* by combining information about high-level functionality of individual elements with information about regions of related elements.[3]

3.4 Intervention of a Knowledgeable User

Since program understanding is an activity that inherently involves humans, it is vital for a tool such as ARTISAn to provide support for user intervention. AR-

[3] The example is omitted due to space limitation and can be found in [5].

TISAn is built with the premise that the information about the system provided by the user can be used by the tool to provide a richer set of results.

For example, in the case of the ANTS Visualizer system, the labeling phases were unable to classify the classes *Tracker*, *Target*, *Sensor*, *Scenario*, and *Device*. The result of a def-use analysis shows that these classes form a region, but the purpose of this entire region is still unknown (Figure 5). Yet, if a user knows that *Device* is a data class then she may provide this information to the ARTISAn tool. This information is then instantly propagated to other elements of the system that have a relationship with the *Device* class, which results in all subclasses of the *Device* class (*Tracker*, *Target*, *Sensor*) being labeled as data classes. Furthermore, since the *Scenario* invokes data elements, we know that it cannot be a communication element (because its exclusion set contains C).

Moreover, the users have an opportunity to add/remove clues, and update the elements' labels (both inclusion and exclusion sets) as well as information about entry points. All the changes are performed immediately, i.e., the tool does not expect the user to restart and repeat the whole analysis. In addition, changes can be undone to further support "what if" scenarios.

4 Evaluation

This section evaluates our approach by discussing our tool's ability to label all the classes in a system (recall rate) and do so correctly (precision rate).

Table 2 lists a representative subset of several case studies (applications) that we have used to evaluate the approach to date. The meaning of each column header and value is described throughout the section. Since our tool currently supports the object-oriented paradigm, we chose to analyze various Java applications that span different domains, including middleware, such as MobiKit, which is built on top of Siena [15]. The first two case studies listed in the table have already been described in the paper. The Jigsaw web server was built by a third party and is available as open source. In all cases we either used the existing design model, if it was available, or reverse engineered its class diagram from the implemented system.

Table 2. Evaluation results

Case study	Classes	Initial labeling		Propagation labeling		
		Initial recall rate	FP	Total recall rate	Total FP	Inconsistencies
Visualizer	37	75%	0	81%	0	0
ANTS	211	67%	0	69%	0	0
DeSi	64	68%	0	93%	0	0
TimeWeaver	120	55%	4	60%	4	0
MobiKit	34	32%	3	58%	3	0
Jigsaw	1009	25%	?	47%	?	4

There are two columns in the table that show the measure of completeness of our approach, one for each of the two labeling steps (*initial* and *total recall rate*). The values in these columns range from 25% (initial labeling in the Jigsaw case study) to more than 90% (after the propagation labeling in DeSi case study).

To validate the correctness of labeled classes we looked at the number of false positives produced (denoted by *FP* in the table), for each of the two steps. All system classes being labeled incorrectly are considered to be false positives. As a reference set to which we compared the ARTISAn-generated labels, we used the labels obtained from the programming environment's chief developer (in case such a person was available), or the results obtained by conducting a survey. We created a collection of over 50 randomly chosen Java source code classes from 4 of the case studies, and asked 20 graduate-level computer science students who are proficient in Java to manually inspect and label the classes into the four categories: P, D, C, and "don't know". Each of the classes had 12 votes on average and we found that our tool produced a low number of false positives (0 to 4), and that their number did not increase from one labeling step to another. ARTISAn correctly labeled 72% of classes that were given to students. We also asked the students to provide a rationale for their decisions. We noticed that the classes for which the students' answers were unanimous and which our tool was unable to label were predominately application-specific processing classes. For example, the classes identified by the students as processing elements had implemented complex algorithms internally, or had mnemonic names (including method names), which all served as a rationale for classifying them. This type of information is currently outside ARTISAn's scope, but can be embedded in additional (domain- or application-specific) clues and rules.

The propagation labeling phase added a significant advantage to labeling as the total recall rate rose to an average of 68% compared to 53% of initial labeling. To validate the set of propagation rules, we compared the ARTISAn's results to the students' responses and also observed the number of inconsistencies discovered by the tool after the propagation-labeling step (the last column in the table). Our results showed that, except in one case, there were no inconsistencies in any of the conducted case studies.

While we believe that these results are already encouraging, we found that several of the identified problems could be avoided to a large degree through better reverse engineering. We relied on the off-the-shelf IBM Rational Rose tool for analyzing the source code, but found early on that it did not discover all class relationships well. Therefore, in some cases we had to manually investigate the code to add missing relationships, which is a labor-intensive and error-prone activity. We also found that Rose did not distinguish between class invocations and class references. This caused inconsistencies in Jigsaw where variable references and calling references did not always coincide. Finally, we found that Rose did not capture calling relationships among methods that belong to different classes. This was a problem whenever a class was identified as, say, both a processing and a data element, i.e., in cases when some of its methods indicated it to be a processing and other methods a data element. This problem then led to inconsistencies within the "data is not allowed to call processing" rule since our tool could not distinguish whether the processing methods of the class invoked the other class or the invocations originated from the data methods.

It should also be noted that we only used a set of domain-independent clues (Table 1) in our case studies. This is because we wanted to use only the clues that are applicable to all case studies and keep their number as small as practical. We found that we could have improved the total recall measure if we had extended our set of clues with other domain- or application-specific clues, such as those based on the use of middleware solutions and naming convention. For example, this way the *jigsaw.Resource* or *Siena.Notification* classes (and all their subclasses) could be recognized as data elements. However, since we were not involved in the development of, nor are we intimately familiar with, any of the mentioned domain-specific case studies, we decided to present results obtained only from using the domain-independent clues.

5 Related Work

Among the numerous program understanding techniques that have been proposed in the literature (i.e., inspection, visualization [6, 7], reading), our work is mostly related to those that achieve the goal of better program understanding and visualization through various architecture recovery methods. This section focuses on this area.

X-ray [10] is an exploratory reverse engineering approach which aids programmers in recovering architectural runtime information from existing software artifacts of a distributed system. Similarly to ARTISAn's notion of clues, X-ray allows the definition of syntactic program patterns, and an associated pattern-matching mechanism. Although the search of program patterns in X-ray would result in the recognition of a more abstract program feature, there is an obvious trade-off in terms of the generality of the approach, the richness of its set of rules, precision, and hit rate. For example, unlike ARTISAn, an interaction mechanism in the form of shared data might not be able to be recognized by X-ray. Furthermore, the lower abstraction level of clues in ARTISAn resulted in its inherent support for "what if" scenarios. The main similarity between ARTISAn and X-ray is in the recognition of program entry points, followed by the application of the study of the dominance relation (usage or reachability analysis), which is well-known and has been used elsewhere in the literature [8].

Similarly to ARTISAn, Lanza and Ducasse [7] propose a categorization of classes, based on class blueprints, as a way to visualize the internal structure of classes. All methods and attributes are distributed among five layers (initialization, interface, implementation, accessor, and attribute) and categorized based on their blueprints. However, this categorization does not try to understand the functionality of a class, but just its static structure.

ManSART [4, 17] is a Software Architecture Recovery Tool that uses special query language routines, called recognizers, to extract and analyze style information from an abstract syntax tree representation of the source code. Similarly to ARTISAn, the result is given as a collection of different architectural views. Architectural representation in ManSART is obtained by manipulating and combining (e.g., merging) different views or, like in ARTISAn, by finding connected subsets of a view.

ACT [1] is an architecture recovery method that combines clustering with pattern-based techniques. Similarly to ARTISAn, it proposes the use of architectural clues that serve as footprints of the high-level design of a system. However, the clues in ACT are small structural patterns (e.g., Façade) that refer to architectural patterns (e.g., Client-Server), which makes them less frequently present and more difficult to recognize, mainly because of their higher complexity and granularity.

Rigi [14] is a program-understanding tool that provides support for the discovery and hierarchical representation of subsystems. Subsystem composition, based on artifacts that are extracted and then stored in an underlying repository, depends on the purpose, audience, and domain [11]. For program understanding purposes, the approach uses low coupling and strong cohesion; alternatively, components can be identified by maintenance personnel based on their experience or qualifications. Unlike ARTISAn, the composition criterion depends on the application that is being re-documented. The use of domain knowledge is unavoidable and the recovery is usually done by persons who are familiar with the application (e.g., its developers).

DiscoTect [16] is a technique for solving the problem of dynamic architectural recovery by mapping low-level implementation style constructs to more abstract architectural operations when predefined run time patterns are recognized. However, the patterns used for search, unlike in ARTISAn, are often very specific, and depend on the application or the environment under inspection.

6 Conclusion and Future Work

This paper discussed ARTISAn, an exploratory and tailorable framework that helps in program understanding tasks. The framework comprises replaceable components to accommodate the exact programming environment and supports developers in understanding large-scale, multi-language source code. The approach is twofold: it provides both a high-level functionality view (i.e., purpose) and a usage view of system elements. In tandem, these views provide the user with a better understanding of the system, and an opportunity to faster locate the parts that are of particular interest (e.g., for maintenance purposes). The first two steps of our approach are evaluated by providing the analysis of the tool's results obtained from several case studies. To evaluate the def-use analysis step, or the correctness of the resulting set of rules, more formal methods are needed. For example, the former can be achieved by comparing the usage view with results of other similar approaches). Finally, determining the overall correctness of the approach requires a deep understanding of the functionality and behavior implemented by each element of a system, which is beyond the capabilities of a light-weight approach, such as ARTISAn.

There are numerous ways to improve our technique. Some of them include the use of reliability metrics that would depend on the reliability of each of the clues and rules applied, and then be used to (automatically) resolve any of possible inconsistencies that result from the labeling process. The other direction of improvement is in providing a richer set of domain- and application-independent clues. For example, the fact that delegating classes act as facades or wrappers to other classes, might turn up to be useful in recognizing communication-processing relationships. Furthermore, the presented rule set can be extended by additional rules that support subcategories of the three major element groups (P, C, and D), such as GUI (P) and interruptible communication (C) type elements. Such a richer propagation rule set would lead to a better understanding of the purpose of a system's elements.

References

1. M. Bauer and M. Trifu, “Architecture-Aware Adaptive Clustering of OO Systems,” in *Proc. of the Eighth European Conference on Software Maintenance and Reengineering* (CSMR 2004), Tampere, Finland, March 24-26, 2004
2. A. Egyed, “Compositional and Relational Reasoning During Class Abstraction,” In *Proceedings of the 6th International Conf. on the UML*, Oct. 2003, San Francisco.
3. A. Egyed, B. Horling, R. Becker, and R. Balzer, “*Visualization and Debugging Tools,” Distributed Sensor Networks: A multiagent perspective*, pp. 33 - 41, editors: Victor Lesser, Charles Ortiz, and Milind Tambe, Kluwer Academic Publishers, 2003
4. D. R. Harris, A. S. Yeh, and H. B. Reubenstein, “Extracting Architectural Features from Source Code,” In *Automated Software Engineering* 3, 1996, pp. 109-138.
5. V. Jakobac, A. Egyed, and N. Medvidovic, “ARTISAn: An Approach and Tool for Improving Software System Understanding via Interactive, Tailorable Source Code Analysis”, TR USC-CSE-2004-513, December 2004, USC, USA
6. D.F. Jerding and S. Rugaber, ”Using Visualization for Architectural Localization and Extraction,” In *Proc. of the Fourth WCRE,* pp. 56-65, Oct. 1997
7. M. Lanza and S. Ducasse, “A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint,” In *Proceedings of the 2001 ACM OOPSLA*, October 14-18, 2001, Tampa, Florida, USA
8. T. Lengauer and R. E. Tarjan, “A Fast Algorithm for Finding Dominators in a Flowgraph,” *ACM Trans. on Programming Languages and Systems,* Vol. 1, No. 1, pp. 121-141, July 1979
9. N. Medvidovic and V. Jakobac, ”Using Software Evolution to Focus Architectural Recovery,” In *J. of Automated Software Engineering*, To appear. 2005
10. N. Mendonca and J. Kramer, “An Approach for Recovering Distributed System Architectures,” In *J. of Automated Software Engineering,* vol. 8, pp. 311-354, 2001
11. H. A. Müller, K. Wong, and S. R. Tilley “Understanding Software Systems Using Reverse Engineering Technology,” In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994
12. E. Perry and A. L. Wolf, “Foundations for the Study of Software Architecture,“ *ACM SIGSOFT SOFTWARE ENGINEERING NOTES,* vol 17 no 4 Oct 1992
13. M. Shaw and D. Garlan, “*Software Architecture: Perspectives on an Emerging Discipline*“ Prentice-Hall, 1996

14. K. Wong, S. Tilley, H. A. Müller, and M. D. Storey, “Structural Redocumentation: A Case Study,” *IEEE Software*, Jan. 1995, pp. 46-54.
15. Siena: A Wide-Area Event Notification Service, http://serl.cs.colorado.edu/∼carzanig/siena/
16. H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, “DiscoTect: A System for Discovering Architectures from Running Systems,” In *Proc. Intl'l Conf. Soft. Eng.,* Edinburgh, Scotland, United Kingdom, May 23-28, 2004
17. S. Yeh, D. R. Harris, and M. P. Chase, “Manipulating Recovered Architecture Views,” In *Proc. Intl'l Conf. Soft. Eng.,*May 17-23, 1997 Boston, pp. 184-194.