Proceedings of the 6th International Conference on the Unified Modeling Language (UML), San Francisco, USA, October 2003

Compositional and Relational Reasoning During Class Abstraction

Alexander Egyed

Teknowledge Corporation, 4640 Admiralty Way, Suite 1010, Marina Del Rey, CA 90292, USA, aegyed@acm.org

Abstract. Class diagrams are among the most widely used object-oriented design techniques. They are effective in modeling the structure of software systems at any stages of the software life cycle. Still, class diagrams can become as complex and overwhelming as the software systems they describe. This paper describes a technique for abstracting lower-level class structures into higher-level ones by 'collapsing' lower-level class patterns into single, higher-level classes and relationships. This paper is an extension to an existing technique that re-interprets the transitive meaning of lower-level classes into higher-level relationships (relational reasoning). The existing technique is briefly summarized. The extensions proposed in this paper are two-fold: This paper augments the set of abstraction rules to also collapse class patterns into higher-level classes (compositional reasoning). While this augmentation is simple and in sync with traditional views of refinement and abstraction, it has drawbacks in defining class features like methods and attributes. This paper thus also demonstrates how to filter low-level class features during abstraction. Our approach requires some human guidance in deciding when to use compositional or relational reasoning but is otherwise fully automated. Our approach is conservative in its results guaranteeing completeness but at the expense of some false positives (i.e., the filter errs in favor of not eliminating in case of doubt). The proposed technique is applicable to model understanding, inconsistency detection, and reverse engineering.

1 Introduction

Software systems are often compositions of high-level components. During the design, those high-level components are gradually refined into lower-level components. In most cases, a refinement step implies an increase in the number of model elements. This benefits the gradual exploration of a problem space that does not require complete, initial knowledge about a software system. Instead knowledge is added incrementally in response to decisions made along the way (i.e., spiral model [1]).

One of the most widely used design languages is the class diagram [12] because it can model a software system throughout its life cycle. Early on, classes and relationships can describe high-level components and their interactions. Refining classes and relationships gradually extends class structures. Over time, class structures may reach a refinement level that allows direct code generation.

Nonetheless, there are problems associated with the use of class diagrams. The refinement process is a complicated task and becomes increasingly difficult with the size of the design. This cannot be avoided since refinement steps tend to increase the number of elements in the class structure. As a consequence, understanding a class structure and reasoning in its presence becomes more complex over time – naturally in response to the larger size but also because of fragmentation. Fragmentation occurs because refinement is not limited to the addition of new elements but it may also replace or substitute existing elements. For instance, a single design element may be replaced by several new design elements to more precisely describe its meaning. As a result, single higher-level elements can only be recognized within clusters of low-level elements. This loss of simplicity is a severe problem for large-scale software development because developers lose oversight more easily leading to confusion, inefficiency, and error.

It is sometimes attempted to maintain separate class structures that describe the same system at different levels of detail. This solution is flawed due to maintainability concerns (i.e., inconsistencies) since evolutionary changes have to be performed on all class structures separately. This is a time consuming and costly task [4], and it is flawed since human error may result in inconsistent updates. Developers thus cannot rely on separately maintained class structures.

Our approach extends an existing technique [4] to avoid the above maintenance problem. It does so by translating (abstracting) lower-level class structures into higher-level ones on demand. It requires some manual guidance in defining the cluster of related, lower-level model elements. A combination of reasoning techniques is then used to analyzes the low-level class structure to translate its elements into a higher-level class structure according to the clusters defined. The result is a new, simpler, higher level class structure that is consistent with the lower-level class structure.

The approach can be used to periodically 'zoom-out' of arbitrary complex class diagrams to investigate and understand their bigger picture. It is lightweight and computationally not very expensive. It is oblivious to evolutionary changes since new abstractions can be generated on demand and discarded thereafter. Roughly a dozen complex, real-world class diagrams were used to evaluate its usefulness. The result of this evaluation showed that generated abstraction results were reliable in most cases although errors did occur. Manual inspection and intervention is thus required to make abstraction results more reliable (i.e., semi automation).

2 Case Study

This paper uses a case study to motivate the need for abstraction and to demonstrate the effectiveness of our abstraction technique. The case study is an existing, embedded, agent negotiation system where multiple, intelligent (software) agents negotiate over the best use of available resources (radars) to track a series of targets [5]. The targets and sensors can be simulated via the RadSim2 components or they can be observed via real sensors and targets (infrastructure was built under the DARPA's ANTS project).

The system's main components are Agent, RadSim2 (sensor and target simulator), and a real-time Visualizer (note: we omit the real hardware for brevity). These components communicate through a network where the Visualizer is on the receiving end and the other components are transmitting. There is also communication going on between Agents and RadSim2 that is omitted here. Figure 1 depicts a high-level class structure of the Visualization component. The Visualizer has an input interface (class *Network Input*) to handle data coming from the network (*Agents* and *RadSim2*) and it has a storage facility for incoming data (class *Data*). A visualization component to display the incoming data (class *Visualization Display*), data properties (class Property Display), and the about box (classes *About Display*) make up the user interface. Data can be stored in a file (class *File Access*).

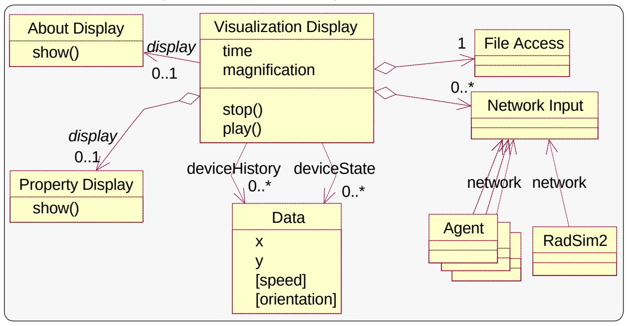

Figure 1. Design Class Structure of Visualizer

The Agents, RadSim2, and Visualizer are implemented software components [5]. Figure 2 shows a refinement of Figure 1 depicting the real, low-level class structure of the Visualization component (implementation-level implies that there is a one-to-one mapping between the class structure and the actual code). A network is used to gather data in near real-time speed (class *DataInputStream*). Gathered data may come from one out of three different types of input devices: *Sensor*, *Target*, and *Tracker*. Sensors track targets and trackers use sensor data to estimate target location. All input devices have properties like a time stamp, a name, and a history of changes. Data received from devices is usually information about activities or state changes. Each data is stored in a new device instance and the class *Scenario* is responsible for keeping track of the current state of all devices (sensors, targets, and trackers) as well as their entire change history. The heart of the visualization component is the class *TrackFrame* which processes and visualizes the data. *TrackFrame* also has an user interface with elements like button, text area, or canvas to graphically display the devices and their states (i.e., device properties are visualized via icons and other graphical clues). A *PopupDialog* is used to display properties of a device.

The class structure in Figure 2 is already simplified since numerous methods and attributes are omitted. Also, the RadSim2 and Agent components are only indicated through single classes. Naturally, they are complex components themselves and consist of numerous classes. We had to omit their details here to make the example more concise. Nonetheless, the depicted, low-level class structure is not trivial despite its relative small size (33 classes).

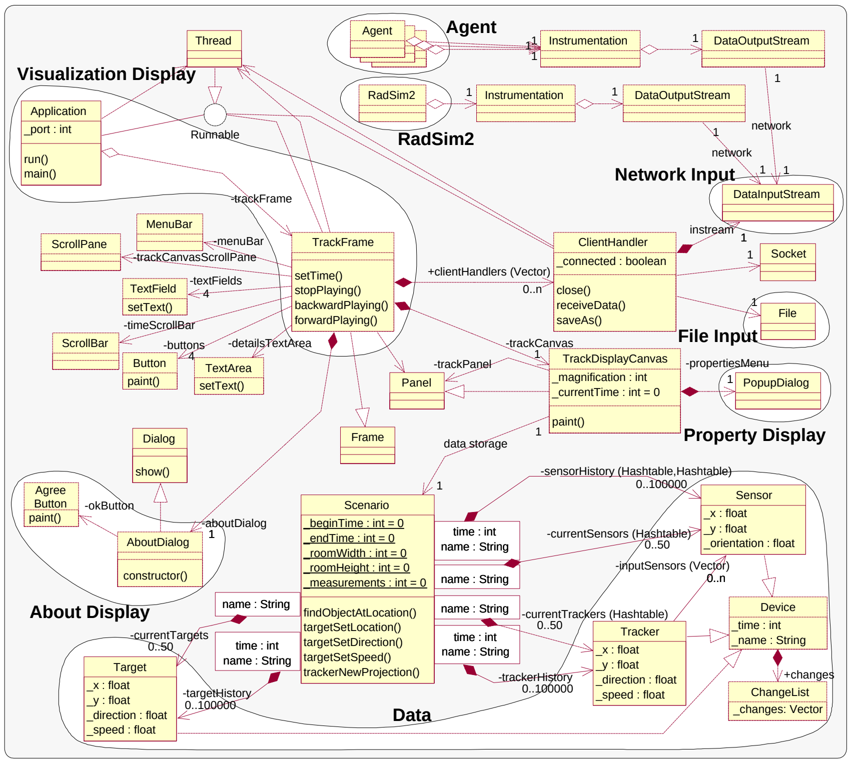

Figure 2. Implementation Class Structure of Visualizer only (contours indicate clusters of design classes; e.g., Agree Button and AboutDialog)

3 The Challenge of Abstraction

For software developers, there is a benefit in having both the high-level class structure (Figure 1) and the low-level class structure (Figure 2). The high-level class structure is easier to understand because it uses less model elements (8 classes versus 33) and it may preserve high-level features (i.e., for model evolution as in [10]). The low-level class structure is more complex but it is also more informative since it gives clues on how to implement the system (i.e., code generation). Despite the desire to keep both class structures for future use, one might lose them:

1) We usually do not know the “best” high-level abstractions in advance. An abstraction is a summary of low-level information and it is often dependent on particular points of views (suppress information irrelevant to a particular view).
2) We cannot presume that the high-level class structure remains consistent with the low-level one over time if they are maintained separately. We have no factual proof that Figure 1 is indeed an abstraction of Figure 2 making it less trustworthy.

Although it is highly beneficial to capture class structures at different levels of abstraction for later reference, potential inconsistencies between them negates their benefits. Inconsistencies are caused primarily through evolutionary changes a system may undergo. For instance, if an implementation is changed (i.e., because of a requirements change) then the design may become inconsistent with the implementation unless extra, manual effort is spent in correcting the inconsistency. Inconsistencies are also caused through human error and become more likely the more complex a class structure is. Maintaining abstractions of class structures manually is associated with high effort and we found that the effort increases non-linearly with the size of the class structure [4]. Here, automation can help in reducing the complexity of this problem.

The following discusses two complementary forms of abstractions. The first one, called relational abstraction, translates sets of low-level relationships (with their classes) into individual high-level relationships. The second one, called compositional abstraction, translates other sets of low-level classes (with their relationships) to individual, high-level classes. Human guidance is required to distinguish between compositional abstraction and relational abstraction. The translation is fully automated and tool supported.

4 Transitive Reasoning

We developed a transitive reasoning technique in collaboration with Rational Software [6]. The technique can take arbitrary complex class patterns to infer transitive relationships between its classes. We define a transitive relationship to be the semantic equivalent of a collection of normal relationships. For instance, if two normal calling relationships exist (e.g., UML associations) so that A calls B and B calls C then, transitively, we can infer that A calls C. Transitive relationships are thus indirect relationships between classes.

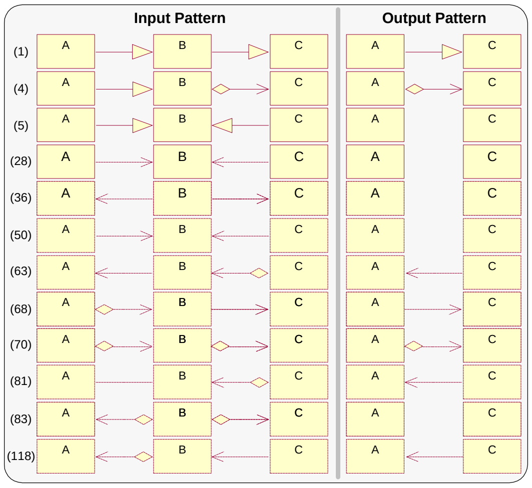

Figure 3. Subset of Transitive Abstraction Rules [6]

A transitive relationship is always the result of a collection of direct relationships. By composing the properties of a collection of direct relationships one can infer properties of the transitive relationship. Properties of relationships include direction of call, type of relationship, or cardinality of association ends. If, say, two relationships have the same type and the same calling direction then transitively the two relationships can be composed into a single one of the same type and direction (see Rule 70 in Figure 3). Transitive relationships are thus a form of abstraction where the transitive relationship is semantically equivalent or weaker (less constrained) than the direct relationships it composes. Figure 3 gives an excerpt of about 121 transitive relationships defined in [3]. For instance, rule 1 states that if A inherits from B and B inherits from C (input pattern) then, transitively, A inherits from C (output pattern). Or Rule 118 states that if C depends on B, A is a part of B (diamond head), and A is called by B (arrowhead) then, transitively, C depends on A.

The given transitive abstraction rules are simple in nature. Most rules describe a collection of two input relationships that can be composed into a single output relationship. What makes this abstraction technique powerful is the large number of simple rules (121 rules for three types of class relationships and various properties). Given the simplicity of the rules, the abstraction algorithm is very fast (see empirical studies in [3]); however, at the expense of precision. UML relationship semantics are not well-treated in the current UML specification which may lead to uncertainties during transitive reasoning (e.g., A calling B and B calling C may not imply A calling C always; see validation in Section 7).

As input, the algorithm takes an arbitrary complex class structure and a list of 'important classes' (the cluster). The list of important classes is needed to distinguish between classes that may be abstracted and classes that may not. In Figure 3 the classes A and C are important and the class B is not important as it gets replaced (together with its relationships) by a higher-level relationship. A human has to make the decision on what classes are important as it depends on the circumstances and usage of the higher-level output.

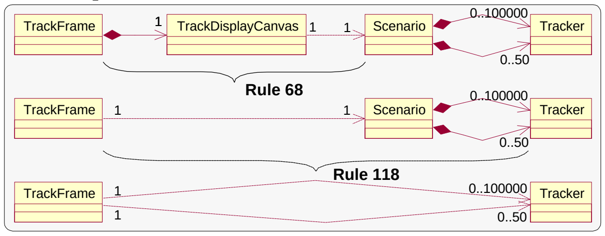

Figure 4. Transitive Relationship between Classes

Figure 4 shows the use of transitive (relational) reasoning in understanding the relationship between the user-defined, important, low-level classes *TrackFrame* and *Tracker* (from Figure 2). Although the two classes are not directly related to one another, a transitive relationship can be derived by eliminating the helper classes *TrackDisplayCanvas* and *Scenario*. Figure 4 shows that the application of Rule 68 eliminates the class *TrackDisplayCanvas* whereas the subsequent application of Rule 118 eliminates the class *Scenario*. This results in two high-level relationships between the two important classes which, together, are semantically equivalent to the replaced classes and all their relationships.

In summary, transitive reasoning merges low-level classes and relationships into higher-level relationships. This form of abstraction is necessary in cases where lower-level classes are the result of refining a relationship. For instance, the low-level class *Scenario* in above example is an index for visualization data. It does not actually contain any data. It therefore does not belong conceptually to the high-level class *Data* and neither does it belong to the high-level class *TraceFrame* since it does not provide user interface services. On a high-level it thus belongs to the relationship that binds the classes *Visualization Display* and *Data*.

5 Compositional Abstraction

While transitive reasoning is important in abstracting away low-level classes that belong conceptually to relationships, there are still other classes that are low-level but belong conceptually to high-level classes and not relationships. For example, the low-level classes *Device*, *Tracker*, *Target*, *Sensor*, and *ChangeList* capture data. They all belong conceptually to the high-level class *Data*. It would be incorrect to apply some form of transitive reasoning here. Compositional Abstraction, discussed in this section, merges low-level classes that belong to single high-level classes.

The shading used in Figure 2 indicates clusters of classes that a user defined to be conceptually related. Compositional abstraction is applicable to each such cluster. Figure 5 demonstrates this on the cluster containing the low-level classes *Agree Button* and *AboutDialog* (the about dialog displays a copyright that has to be acknowledged through a special agree button). Since the two low-level classes were defined by a user to belong conceptually to the single high-level class *About Display* (see name given to cluster in Figure 2), a new high-level class *About Display* can be created (note that we do not alter any existing class structures during abstraction but instead generate a new, separate class structure; see also Figure 5 bottom).

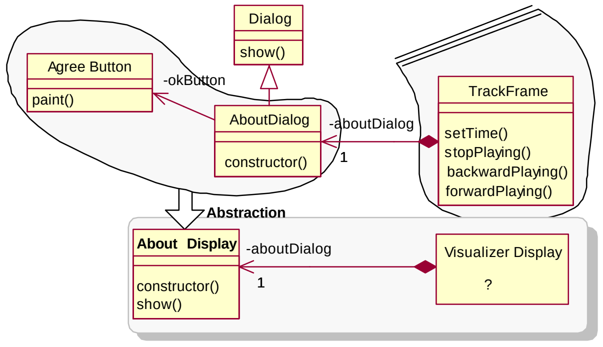

Figure 5. Scenario for Compositional Abstraction

Creating higher-level classes to represent a cluster of lower-level classes is an intuitive and straightforward activity since the higher-level elements can be seen as placeholders for the lower-level ones (refinement is often the exact reverse of this activity where a single high-level element is broken up into several low-level ones). However, a class is much more than a box. A class also captures structure through class attributes and methods and it captures behavior through relationships.

services of class = structure and behavior of class

For a cluster of low-level classes to consistently implement a single high-level class, both have to exhibit the same structure and behavior. The structure and behavior of a high-level class is obvious since such a high-level class clearly defines its attributes, methods, and relationships. On the other hand, a cluster of low-level classes belonging to a single high-level class *distributes* its structure and behavior among the many lower-level classes. To complicate matters, it is generally incorrect to assume that the sum of the services of low-level classes is equivalent to the services of the high-level class. The rationale:

- a low-level class may offer partial, low-level services
- a low-level class may offer generic, rich services that are partially used

Both cases above imply that individual low-level classes may provide services that are not being offered by its high-level abstraction. As a consequence:

services of high-level class Í sum of the services of lower-level classes

Unfortunately, it is not possible to automatically determine which services of low-level classes are low-level and which ones are high-level. However through another observation we can provide a more meaningful approximation.

a low-level class cannot provide high-level services if it is not accessible to classes from other clusters

A class is accessible by another class if a direct or indirect (transitive) calling dependency exists (expressed through relationships). If a low-level class is accessed by another class then its services are being used. If that low-level class is accessed by low-level classes of the same cluster only then its services are internal within that cluster. Contrary, if that low-level class is accessed by low-level classes of other clusters then its services are publicly available. Only public services can be high-level services.

isEntryPoint(e:Class) $\leftrightarrow \exists$ c $\in$ classes, c $\in$ clusters.classes,
e $\in$ clusters.classes, c.cluster $\neq$ e.cluster, existsDirectRelations(e,c) or existsTransitiveRelations(e,c)

The above definition states that a given class is a public entry point if it is part of a cluster and there exists another class part of another cluster that has a relationship to the given class.

Figure 5 (top) continues on the previous example. It also depicts the only two neighboring low-level classes that directly interact with the *About Display* cluster. One of the neighboring classes is not part of any cluster (*Dialog*) whereas the other neighboring class is part of the *Visualizer Display* cluster (*TrackFrame*). Abstracting the *About Display* cluster combines the two low-level classes *Agree Button* and *AboutDialog* but it does not combine their services. The low-level class *Agree Button* is a support class in that only *About Dialog* can interact with it. Neither one of the other neighboring classes *Dialog* or *TrackFrame* can access it without going through *AboutDialog*. This indicates that services defined in *Agree Button* are inaccessible to classes outside the cluster and need not be abstracted. The opposite is true for the class *AboutDialog*. It is directly accessible by a class of another cluster which implies that its services are publicly available (isEntryPoint is true). Services of *AboutDialog* must be abstracted which also includes the inherited services of *Dialog*.

The definition of the entry point has another stipulation in that either direct relations (existsDirectRelations(e,c)) or transitive relations (existsTransitiveRelations(e,c)) are used to determine entry points. Why is a distinction being made here? The relationship between the cluster *About Display* and *Visualizer Display* is a direct one since the classes of both clusters have a direct relationship defined (aggregation between *AboutDialog* and *TrackFrame*):

existsDirectRelations(a,b) $\leftrightarrow$

$\exists$r $\in$ relations, (r.origin=a and r.destination=b) or (r.destination=a and r.origin=b)

This definition states that there exists a direct relationship between two classes if a relationship exists that has both classes as origin or destination. In the low-level class structure in Figure 2 there are many direct relationships (any arrow between classes) but there is only one direct relationship between classes belonging to different clusters. Note that we are not interested in direct relationships between classes of the same cluster (e.g., the relationship between *Application* and *TrackFrame* is not relevant).

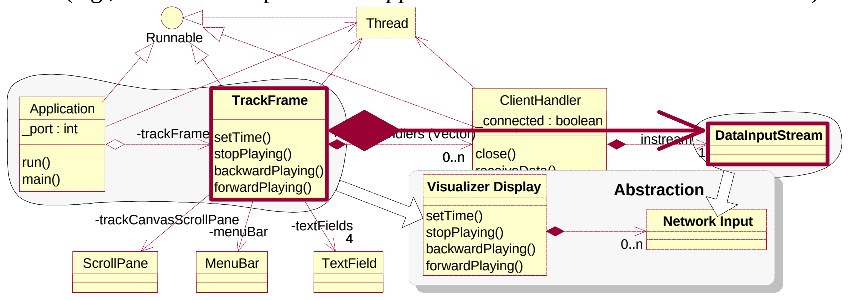

Figure 6. Transitive Relationship between Classes without direct Relationships

The lack of direct relationships between classes of different clusters does not imply that those classes cannot access one another. As an example, consider Figure 6 with the two clusters *Visualizer Display* and *Network Input*. Both clusters do not directly relate to one another because the low-level class *ClientHandler* separates them. Still, a calling dependency exists because class *TrackFrame* (part of cluster *Visualizer Display*) can call methods of class *ClientHandler* (implied through arrowhead in aggregation relationship) and those methods, in turn, can call methods of class *DataInputStream* (part of cluster *Network Input*). This implies transitively that class *TrackFrame* can call class *DataInputStream* via the 'helper class' *ClientHandler*. This transitive relationship defines *TrackFrame* and *DataInputStream* as entry points for their respective clusters.

existsTransitiveRelations (a,b) $\leftrightarrow$ deriveTransitiveRelations(a,b) $\neq \emptyset$

Since clusters are user-defined sets of low-level classes that conceptually belong to single, high-level classes, it follows that all remaining classes not defined in clusters belong to single, high-level relationships. Transitive reasoning must only use classes not part of any cluster since the others already belong to higher-level elements (i.e., higher-level classes). It follows that there is no transitive relationship between *Application* and *DataInputStream* through the 'helper classes' *TrackFrame* and *ClientHan-*

dler. Furthermore, there are also no transitive relations going through *Runnable* or *Thread* because no transitive abstraction pattern applies. For a proof on these matters please refer to [4].

The previous two examples of abstracting clusters were cases of two or more classes per cluster with single public classes. Abstracting those clusters is relatively simple since the services of the high-level classes are essentially the services of the entry point classes. Figure 7 discusses an example of a cluster with multiple entry points. It shows the previously mentioned cluster *Data* and its five implementation classes *Target*, *Tracker*, *Sensor*, *ChangeList* and *Device*. Using the definition for isEntryPoint one can determine that *Target*, *Tracker*, and *Sensor* are entry points but *Device* and *ChangeList* are inaccessible (that there are transitive dependency relationships between low-level class *TrackFrame* and the three entry point classes).

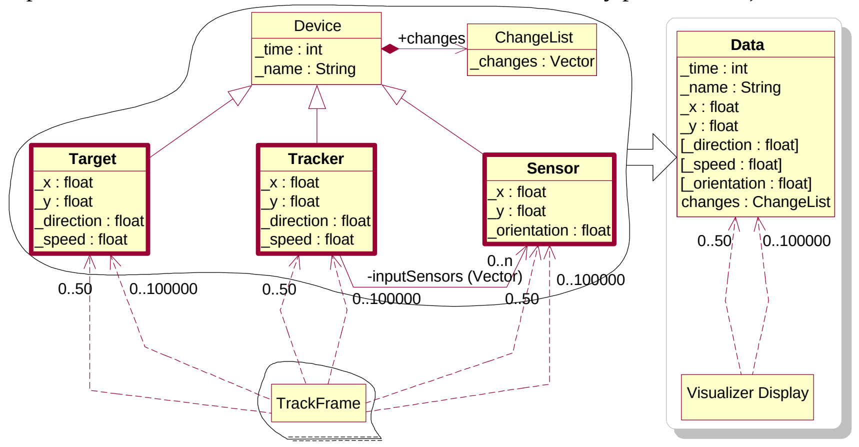

Figure 7. Compositional Abstraction of Multiple Entry Point Classes

Abstracting the *Data* cluster requires the creation of a new class called *Data* with the combined services of the entry point classes. *Data* must have the attributes ‘x’ and ‘y’ because *TrackFrame* can access those attributes regardless what entry point class is chosen. Note that each relationship, in principle, denotes a distinct access to an entry point class. Since some attributes are not associated with all entry point classes (e.g., *orientation* is only part of *Sensor*) it follows that not all attributes of all entry point classes are accessible through all relationships. For abstraction this implies that some attributes are mandatory and other attributes are optional. Mandatory attributes are the intersection of all attributes in entry point classes. The three entry points *Tracker*, *Target*, and *Sensor* all have in common the attributes ‘x’ and ‘y’ (mandatory attributes). Since the three entry points also inherit the attributes ‘time’ and ‘name’ from *Device* those attributes also become mandatory (recall a similar exception involving inheritance relationships in Figure 5). Optional attributes are the union of all attributes in entry point classes (whether inherited or not) minus mandatory attributes. For example, an instance of *Data* might have attributes like ‘direction’ or ‘orientation’ but likely not both. Optional attributes are denoted with rectangular brackets (i.e., [direction : float]).

Methods can be treated like attributes. Mandatory methods are methods used in all entry point classes whereas optional methods are all methods used in all entry point classes minus mandatory ones. Relationships will be discussed later.

```
for all cl ∈ clusters
    classesInCluster = classes.collect( c : class | c.cluster = cl)
    entryPoints = classesInCluster.collect(c | isEntryPoint(c))
    allMethods = ∅
    for all e ∈ entryPoints
        allMethods = allMethods +
            e.methods.collect(m | m. isPublic=true) +
            e.parents.methods.collect(m | m.isPublic=true)
    mandatoryMethods = allMethods
    for all e ∈ entryPoints
        mandatoryMethods = mandatoryMethods.
            intersection(e.methods+e.parents.methods)
    optionalMethods = allMethods - mandatoryMethods
    allAttributes = ∅
    for all e ∈ entryPoints
        allAttributes = allAttributes +
            e.attributes.collect(a | a.isPublic=true)
            e.parents.attributes.collect(a | a.isPublic=true)
    mandatoryAttributes = allAttributes
    for all e ∈ entryPoints
        mandatoryAttributes = mandatoryAttributes.
            intersection(e.attributes+e.parents.attributes)
    optionalAttributes = allAttributes – mandatoryAttributes
    cl.abstraction = new Class(name = cluster.name)
    cl. abstraction. mandatoryMethods = mandatoryMethods
    cl. abstraction. optionalMethods = optionalMethods
    cl. abstraction.mandatoryAttributes = mandatoryAttributs
    cl. abstraction.optionalAttributes = optionalAttributes
end //for all
```

Figure 8. Compositional Abstraction Algorithm

Figure 8 summarizes the algorithm for compositional abstraction as was discussed in this section. For each cluster, a new, high-level class is created with the name of the cluster and the mandatory and optional services derived out of entry point classes. Entry point classes are identified using the isEntryPoint definition we gave earlier. The command 'collect' traverses a list of classes part of the same cluster in 'classesInCluster' and returns only those for which the given condition holds (e.g., isEntryPoint is true). The variable 'allMethods' captures the union of all public methods in entry points (note: only public methods are accessible from the outside). The variable 'mandatoryMethods' is the intersection of all methods in entry points and the variable 'optionalMethods' is the union of all methods in entry points minus mandatory methods. Mandatory and optional attributes are computed in the same fashion.

6 Relational Reasoning

Relational reasoning extends compositional reasoning in that relationships may play two distinct roles during abstraction: Relationships may be either associated with abstract classes or with abstract relationships. An obvious case is the relationship between *Scenario* and *TrackDisplayCanvas* which is used for transitive reasoning and, consequently, is associated with an abstract relationship (see Figure 4). A counterexamples is the relationship between *Device* and *ChangeList* in Figure 7 which is associated with clusters and consequently with abstract classes.

```
abstractableRelations = ∅
for all cluster ∈ clusters
    for all c1 ∈ cluster.classes
        for all c2 ∈ classes, c2.cluster ≠ ∅, c2.cluster ≠ cluster
            abstractableRelations = abstractableRelations +
                relations(c1, c2) + transitiveRelations(c1, c2)
        end //for all
    end //for all
    entryPoints = cluster.classes.collect(c | isEntryPoint(c))
    relationRelevant = abstractableRelations.refinements
    classRelevant = entryPoints. parents.outgoingRelations +
        entryPoints.outgoingRelations - relationRelevant
    allRelations = ∅
    for all e ∈ entryPoints
        allRelations = allRelations +
            e. outgoingRelations + e.parents.outgoingRelations
    mandatoryRelations = allRelations
    for all e ∈ entryPoints
        mandatoryRelations = mandatoryRelations. intersection(e. outgoingRelations +
            e.parents.outgoingRelations)
    optionalRelations = allRelations - mandatoryRelations
    cluster.abstraction.mandatoryAttributes = cluster.abstraction.mandatoryAttributes +
        convertToAttributes(mandatoryRelations.collect( r:Association | r.connection(r.destination).isPublic)
    cluster.abstraction.optionalAttributes = cluster.abstraction.optionalAttributes +
        convertToAttributes(optionalRelations.collect( r:Association | r.connection(r.destination).isPublic)
end //for all
for all r ∈ abstractableRelations
    abstraction = new Relation(properties like r)
    abstraction.origin = r.origin.cluster.abstraction
    abstraction.destination = r.destination.cluster.abstraction
end //for all
```

Figure 9. Relational Abstraction Algorithm

Relationships that are associated with abstract relationships are only needed to derive transitive relationships. Relationships that are associated with abstract classes (clusters) might be used to derive abstract attributes. UML association and aggregation relationships define attributes implicitly. For instance, the relationship from *Device* to *ChangeList* is an aggregation (a variation of an association). This aggregation uses the role name "+changes" to reference instances of *ChangeList*. The role name is essentially a unique identifier within *Device* which corresponds semantically to a global variable (an attribute). An association or aggregation thus may cause an abstract attribute if it is originating from an entry point class and if it is public. In Figure 7, the aggregation "changes" causes an attribute in *Data* because it satisfies both above conditions: (1) it is accessible because an entry point inherits it and (2) it is public as indicated through the plus symbol in front of the name (plus is public and minus is private). Other relationships like the one between *TrackFrame* and *MenuBar* do not become attributes because they are either inaccessible or private. Using the name of the association as the attribute name and the name of the destination class as the attribute type converts an association into an attribute.

convertToAttribute (a:Association) ↔

    new Attribute(name = a.name, type=a.destination.name)

Figure 9 gives the algorithm for relational reasoning using the transitive relationship technique discussed before. First, the algorithm locates all direct or transitive relationships between clusters (relationships between classes outside clusters are not of interest here) and stores them into the variable 'abstractableRelations.' Next, the entry points are calculated analogous to Figure 8. Relationships associated with abstract relations are computed and stored in 'relationRelevant.' Then relationships associated with classes are computed by looking at outgoing relationships of entry points or their parents minus the ones that are associated with relationships. If a relation is accessible by all entry points then that relation might become a mandatory attribute; otherwise it might become an optional attribute (analogous to Figure 8). Whether or not a relationship becomes an attribute is determined next by converting only those relationships that are associations (note: aggregations are also associations) and that have a publicly accessible destination. In a last step, all abstractable relations are made into abstract relations by taking over their properties and defining their abstract origins and destinations which are the abstract classes created in Figure 8.

Figure 10 shows the result of abstracting the implementation class in Figure 2. In comparison with the existing design class structure in Figure 1, the abstraction is indeed similar but has a few differences. These differences are possible inconsistencies.

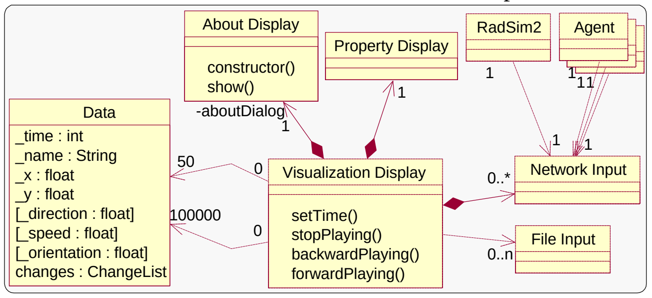

Figure 10. Abstracted Implementation Class Structure

7 Validation

Abstraction aims at reversing the refinement process that classes undergo. The process of abstraction is analogous to a magnification glass with which one may 'zoom out' any degree to inspect a complex class structure. User input is required in defining clusters of related, low-level classes. Finding clusters in a complex class structure varies in difficulty. The main problem of finding clusters is finding their precise boundaries. The full automation provided by our approach has the advantage that cluster boundaries may be explored iteratively through trial-and-error.

Our approach is based on transitive reasoning. This has the advantage that certain properties can be guaranteed, Most importantly, our approach to transitive reasoning is conservative in identifying all possible transitive relationships [4]. However because of this conservative nature, it may also report additional, false transitive relationships. Extensive empirical studies [4] have shown this error to be little (<5% of all reported results).

As a result, the identification of entry points is correct if it is based on direct relationships but potentially incorrect if it is based on transitive relationships. However, due to the conservative nature of transitive reasoning it is guaranteed that all entry points are identified with a low likelihood of additional, wrong ones (<5%).

In summary, our approach is guaranteed to identify all high-level classes, relationships, and class services (i.e., attributes and methods). False positives may exist which implies that the true abstraction result is a subset of the reported abstraction result. Human intervention is required to determine this subset which requires investigating the results of transitive reasoning and the services of entry point classes.

8 Related Work

Many techniques have been proposed to aid the understanding of complex class structures. There are reading techniques like inspection [7] that use group effort to cope with complexity. Most of these techniques are manual and involve high effort and manpower. Using multiple views is an effective form of separating concerns [14]. Class structures can be subdivided into viewpoints [9] where partial and potentially overlapping portions of the structure are depicted. The sum of all diagrams is the complete class structure itself. Multiple views make use of the fact that one does not need access to all classes to understand a particular concern. Still, multiple views cannot avoid the problem of class replacement and substitution (fragmentation discussed earlier) and although multiple views can make classes belonging to individual concerns more understandable, they generally cannot be used to project a high-level, simpler abstraction of a complex class structure.

Slicing [13] is another technique to cope with software complexity. Although it is primarily used on source code, one could imagine its use on design languages like class diagrams. Slicing uses some property or rule and investigates how such a rule might become valid. For instance, a ‘slice’ could display all possible design elements that might affect a variable directly or indirectly. Slicing, like the separation of concerns, can be very effective in understanding individual concerns within complex class structures but it does not improve the overall understanding (abstraction) of a complex system.

Techniques have been proposed to formalize UML class diagrams [2]. Those techniques are important to improve the precision and meaning of model elements like classes and relationships. Although, a more precise definition of class diagrams can significantly aid its understanding and use (i.e., code generation, verification, etc.) it itself does not provide abstraction. Still, a more precise definition may make it easier to devise more reliable abstraction techniques.

We see the key to class abstraction in automated transformation. For example, the approach proposed by Fahmy and Holt [8] provides a set of transformation rules that can be applied onto class-like structures. Indeed there is some similarity between some of their rules and our abstraction technique (e.g., lifting). The main problems we see are that their rules are too few in number for comprehensive abstraction and they expect the rules to be applied manually. Racz and Koskimies [11] devised a technique to evaluate transitive relationships (indirect relationships spanning multiple classes in class structures). Their technique is very similar to our approach to transitive reasoning but their technique is only semi-automated. Also, as was shown in this paper, understanding transitive relationships between classes only provides limited abstraction and is not sufficient to derive abstract properties of class structures in general.

Lieberherr et al. [10] defined class transformation methods to capture evolution. They argue that class evolution is inevitable and result in new class models that, preferably, should be as consistent as possible with earlier versions. Although, one could argue that evolution is a form of refinement, we take a narrower stance. For us, refinement has to maintain consistency within a given model. Their work thus addresses evolutionary "refinement" and "consistency issues" that are considered outside the scope of this paper. Nonetheless, one can envision a strong need for our approach to be combined with theirs so that model refinement and abstraction can be complemented with model evolution.

9 Conclusions

Understanding a complex class structure does not necessarily require the understanding of every single class. Clusters of classes like buttons, windows, text fields, or scroll bars are integral parts of user interfaces and in some cases it is not necessary to know their details. This paper introduced a technique on how to abstract low-level classes into higher-level classes using these kinds of clusters to direct the abstraction. The technique composes classes within clusters into abstract classes; and classes outside clusters into abstract relationships. As a result low-level properties can be reinterpreted as higher-level properties. Our approach is a lightweight, easy-to-use, and iterative way of simplifying the complexity of class structures. Abstraction results can improve the understanding of a model or ease the navigation between its elements. We also used the approach for consistency checking and reverse engineering.

Our approach was validated on numerous real applications. We found that our approach produces useable abstractions very fast. The quality of results is determined by the quality (correctness) of the clusters and the approach tends to error in favor of showing relationship in case of doubt. We observed that this only becomes a problem if the degree of abstraction is very high (1:10 abstraction ratio or higher). Given the iterative nature of our technique, we also found it easy to reason about correct cluster boundaries using a trial-and-error-like approach. If an abstraction does not match a mental model then a simple adjustment of the cluster boundary may resolve the problem. If the problem persists then the difference may indicate an inconsistency.

It is future work to improve the presented abstraction technique by integrating it with other (UML) diagrams. Since other diagrams may embody additional modeling data, it may be used to make abstractions stronger and more reliable. It is also future work to investigate how the semantics of a relationship is affected if it is composed of low-level classes.

Acknowledgements

We wish to thank Philippe Kruchten, Barry Boehm, Cristina Gacek, Paul Grunbacher, Nenad Medvidovic, DaveWile, and the anonymous reviewers for insightful comments. This work was support by DARPA through contracts F30602-00-C-0218 and F30602-99-1-0524.

References

[1] Boehm B., Egyed A., Kwan J., and Madachy R.: Using the WinWin Spiral Model: A Case Study. IEEE Computer, 1998, 33-44.

[2] Bourdeau R. H. and Cheng B. H. C.: A Formal Semantics for Object Model Diagrams. IEEE Transactions on Software Engineering (TSE), 1995.

[3] Egyed, A.: "Semantic Abstraction Rules for Class Diagrams," Proceedings of the 15th IEEE International Conference of Automated Software Engineering (ASE), Grenoble, France, September 2000.

[4] Egyed A.: Automated Abstraction of Class Diagrams. ACM Transaction on Software Engineering and Methodology (TOSEM) 11(4), 2002, 449-491.

[5] Egyed, A., Horling, B., Becker, R., Robert Balzer: Visualization and Debugging Tools, In Distributed Sensor Networks: A multiagent perspective, edited by Victor Lesser, Charles L. Ortiz Jr., and Milind Tambe. Boston, Kluwer Academic Publishers, 2003.

[6] Egyed, A. and Kruchten, P.: "Rose/Architect: A Tool to Visualize Architecture," Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS), January 1999.

[7] Fagan M. E.: Advances in software inspections. IEEE Transactions on Software Engineering (TSE) 12(7), 1986, 744-751.

[8] Fahmy, H. and Holt, R. C.: "Using Graph Rewriting to Specify Software Architectural Transformations," Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE), Grenoble, France, September 2000, pp.187-196.

[9] Finkelstein A., Kramer J., Nuseibeh B., Finkelstein L., and Goedicke M.: Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. International Journal on Software Engineering and Knowledge Engineering, 1991, 31-58.

[10] Lieberherr K. J., Hursch W. L., and Xiao C.: Object-Extending Class Transformations. Journal Formal Aspects of Computing 6(4), 1994, 391-416.

[11] Racz, F. D. and Koskimies, K.: "Tool-Supported Compression of UML Class Diagrams," Proceedings of the 2nd International Conference on the Unified Modeling Language (UML), Fort Collins, CO, October 1999, pp.172-187.

[12] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, 1999.

[13] Snelting G. and Tip F.: Understanding Class Hierarchies Using Concept Analysis. ACM Transactions on Programming Languages and Systems, 2000, 540-582.

[14] Tarr, P., Osher, H., Harrison, W., and Sutton, S. M. Jr.: "N Degrees of Separation: Multi-Dimensional Separation of Concerns," Proceedings of the 21st International Conference on Software Engineering (ICSE 21), May 1999, pp.107-119.