Component-Based Design in Tako: A Case Study

Arun Sudhir
Virginia Tech
Falls Church, VA
aruns@vt.edu

Gregory Kulczycki
Virginia Tech
Falls Church, VA
gregwk@vt.edu

Jyotindra Vasudeo
Virginia Tech
Falls Church, VA
vasudeo@vt.edu

ABSTRACT
Tako is an object-oriented language similar in many respects to Java, but is designed to support alias avoidance and thereby simplify both formal and informal reasoning. Aliasing in Java occurs mainly due to reference assignment, which Tako replaces with alternative data assignment mechanisms such as copying, swapping, and initializing transfer. Though the changes are syntactically minor, their effect on component design and design patterns is not. This paper examines a non-trivial program designed and implemented in Tako, and discusses how and where the design differs from a typical Java program. We look at how the design impacts specification and reasoning. We found that while many design decisions were unaffected by the emphasis on alias avoidance, there were certain design issues that Java programmers would need to adjust to. A key component in the example program is a tree data structure that would likely be implemented using a composite pattern in Java.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features.

Keywords
Alias avoidance, design patterns

1. INTRODUCTION
Tako is an object-oriented language with Java-like syntax that supports alias avoidance, thereby simplifying both formal and informal reasoning [1]. Though most alias-avoidance languages, including Tako, do support limited aliasing, they differ from alias-control languages in that aliasing is the exception rather than the rule. Tako's main use to date has been as an instructional tool to help teach students how to reason formally about their code. The students are taught how to read and write specifications, and how to use those specifications to trace through code based on specific input. They also learn how to construct symbolic reasoning tables – generalized, user-friendly, tracing tables that can be used to generate the verification conditions needed for correctness proofs. Tako simplifies writing specifications and reasoning about code because programmers do not need to keep track of the indirection that pervades traditional object-oriented languages.

Many of the alias-avoidance techniques found in Tako have their origins in the Resolve language [2, 3]. Resolve is an integrated programming and specification language intended to support full, heavyweight program verification. Central to the approach of both Resolve and Tako to facilitate alias avoidance is the use of alternative data assignment operators such as swapping. Some researchers have raised concerns about whether the paradigm associated with this approach "can mesh well with mainstream object-oriented programming techniques" [4, 5].

This paper examines a non-trivial program designed and implemented in Tako, and discusses how and where the design differs from a typical Java program. We found that while many design decisions were unaffected by the swapping paradigm and the emphasis on alias avoidance, there were certain design issues that Java programmers would need to adjust to.

Section 2 gives a brief overview of the Tako language, emphasizing how it differs from Java. Section 3 describes the architecture of the program we designed – a simple text-based adventure game. Section 4 describes and partially specifies a key data structure used in the program, an indexed tree, which has the features of both a tree and a map. Section 5 describes how the indexed tree is used in the program and demonstrates how to trace through a portion of code based on the indexed tree specification. Section 6 raises other design issues that distinguish Tako from Java. Section 7 provides some concluding thoughts on the subject.

2. OVERVIEW OF TAKO
The main difference between Tako and Java is that Tako includes alias avoidance features. This allows programmers to view variables directly as objects rather than as references to objects. The following subsections give a few important differences.

2.1 No primitive types
In Java, there are two kinds of types: primitive types and reference types. Primitive types are built-in to the language and their variables denote values. Some reference types are built-in to the language, but most are user-defined. Variables of a reference type denote references to objects. In Tako, all types are value types. Some are built-in and others are not, but variables in Tako always represent objects, no matter what type they are from. In addition, no two variables ever represent the same object.

In Tako, as in Java, some types that are built-in to the language have special syntax. These include Booleans, Integers, Strings, and Arrays. In general, if a type has special syntax in Java, its corresponding type in Tako will probably have it also.

Sometimes we talk about replicable types in Tako. A type is replicable if it has a *replica* operation. Some common types like Booleans, Integers, and Strings, already include a *replica* operation. Programmers can make any type replicable by simply adding a *replica* operation themselves.

2.2 Initial values
In Java, the compiler will report an error if you try to use a variable before you have initialized it. In Tako, all variables get

initial values when they are declared. Tako uses the default constructor for this purpose. As in Java, Tako programmers are encouraged to provide default constructors for all objects. If no default constructor exists for a type, a newly declared variable of that type will get a null value. Since null values are not consistent with viewing variables directly as objects, omitting a default constructor is discouraged.

2.3 Alternative data assignment

Java's assignment operator introduces aliasing because it copies references. Tako is designed to avoid aliasing, so it requires alternative mechanisms to assign objects to variables. Here is a brief overview of the alternatives.

2.3.1 Swapping

Swapping is the primary means of data assignment in Tako. When two variables are swapped, the variables simply exchange objects. Swapping does not introduce aliasing because if the variables denote distinct objects before the operation, they still denote distinct objects after the operation. Swapping is also a constant time operation, because the compiler implements it by swapping memory locations. However, swapping is a symmetric operation, so both variables need to have the same type before they can be swapped.

2.3.2 Initializing transfer

The initializing transfer operation in Tako "<–" transfers an object from one variable to another and gives the first variable an initial value. The transfer operation is fairly efficient, but if the variable receiving the object already had a different one, its original object will become garbage and will have to be deallocated eventually.

2.3.3 Function assignment

Another way of getting an object into a variable is by assigning the result of an function to the variable. The function assignment operator in Tako ":=" always expects a variable on its left-hand side and an expression on its right. If the compiler sees anything other than a variable on the left-hand side, it will complain. If it sees a variable rather than an expression on the right-hand side, as in "max := n", the compiler tries to replicate the variable, as in "max := n.replica()". If no replica operation is found, it reports an error.

2.4 In-out parameter passing

By default, parameter passing in Tako is in-out. In other words, argument values are transferred to the formal parameters, the method is executed, and formal parameter values are transferred back to the arguments.

In-out parameter passing allows Tako programmers to keep functions and procedures distinct. A function has a return type (non-void) and a procedure does not. By convention, functions should not have side-effects. A function has side-effects if it changes the value of a variable. An example of a side-effecting function is a pop method in a Java stack, as in the assignment x = s.pop(). It is a function because it returns a value, and it has a side-effect because it changes the current stack object. If a Tako programmer wants an operation to change the state of the program, they should write it as a procedure rather than a function, so that it would be called as "s.pop(x)".

2.5 Result variable

In Tako functions (non-void methods), the result of the function is returned through a special result variable. This guarantees that the object returned is unique and is not an alias to any existing object. The result variable has the same type as the return type of the function. The compiler treats the result variable as if its declaration is the first statement of the method. So a getter method for a private attribute length would be written as "**public** Integer getLength( ) { **result** := length; }" and interpreted by the compiler as "**public** Integer getLength( ) { Integer **result**; **result** := length; }". The result variable is initialized when it is declared, so even a function with no statements would return an initial value.

2.6 Pointer component

Despite the fact that the design of Tako is focused on avoiding general aliasing, we understand that there are circumstances when programmers will need pointers and references to efficiently implement certain classes. For this purpose, Tako has a pointer component that is specifically designed to aid in the implementation of linked data structures such as lists and trees.

3. ADVENTURE GAME ARCHITECTURE

To experience first hand the paradigm shifts involved in programming a non-trivial application in Tako, we undertook the development of a text-based adventure game. The game was initially developed in Java, but with the intent that it would eventually be ported to Tako. Figure 1 shows the general architecture of the application in the form of a UML class diagram. It is loosely based on traditional text-based adventure game development systems such as Inform [6] and TADS [7].

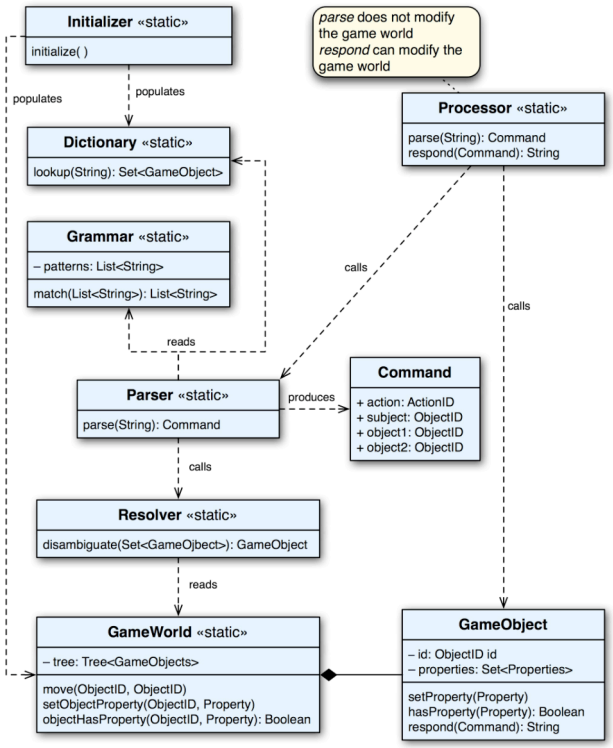

Figure 1. Adventure game architecture

The game accepts text inputs from the player, which are usually simple imperative sentences, such as "take the chess piece" or "put the red queen on the chess board". A Parser component parses the input based on a supplied grammar and dictionary of game objects. A Resolver component tries to determine what game object is intended when the player enters ambiguous text. The GameWorld component tracks the state of the game. It is a cross between a tree-like data structure and a database that stores all the game objects. When the player inputs a command, the application updates the GameWorld accordingly and generates a text response.

The program contains about 50 classes and consists of over 4,000 lines of code. It required approximately 85 man hours to code the game in Tako based on the Java version of the game. Table 1 gives process metrics for the conversion from Java to Tako. The time spent on the conversion is shown during and after the translation process. During the translation process, most of the time went into translating statements that used reference copying into statements that used swapping. Part of this process involved direct substitution of the reference copy operator with the swap operator. Part of it also involved swapping objects from containers. In both cases, there was the possibility that objects had to be swapped back, as illustrated in section 5. A fair amount of time was also spent in converting methods with return values to their equivalent in Tako. If the methods had side-effects, then they where changed to procedures (methods without return values) in Tako, and the return value was passed out through a parameter. If the methods did not have side-effects, then the methods where changed to appropriate functions (methods with return values) in Tako. In Tako functions, the distinguished **result** variable is used to store the return value. Some time was also spent for converting Java enumeration types to static integer variables. Enum types are supported in Java 1.5 but not in Tako. The remaining time was spent in copying and pasting code from one language to another. This was possible due to the similarities in Java and Tako syntax.

Table 1. Process metrics for conversion from Java to Tako

Description	Hours
Time spent during translation	
Conversion of Enum types to static integer variables	2
Converting side-effecting functions	10
Converting non-side-effecting functions	5
Translating code with aliasing to swapping methodology	15
Simple translations (copy and paste)	5
Time spent after translation	
Debugging errors due to erroneous translation	30
Debugging errors already present in Java version	18

Debugging code after the translation took up the majority of time. The debugging process metrics were divided into two parts: time spent in debugging errors that occurred due to erroneous translation, and time spent in errors that where present in the original version of the Java code. Nearly 30 hours were spent in debugging the translation errors. We had expected this part of the process would take the most time since this was our first attempt at such a translation. The other 18 hours spent in debugging could have been avoided if the original Java version had been tested thoroughly.

4. INDEXED TREE COMPONENT

The two most sophisticated components in the adventure game are the Parser and the GameWorld. The Parser takes an imperative sentence typed by the player and converts it to a four-part command. The parser as implemented in Tako is not very different from the parser as implemented in Java. This is probably due to the fact that the Parser is designed to essentially encapsulate a single, though complex, method – parse. The Tako GameWorld component does have significant differences with the Java GameWorld component. Therefore, we spend this section and the next discussing it. The GameWorld component is based on a custom data-structure called an IndexedTree.

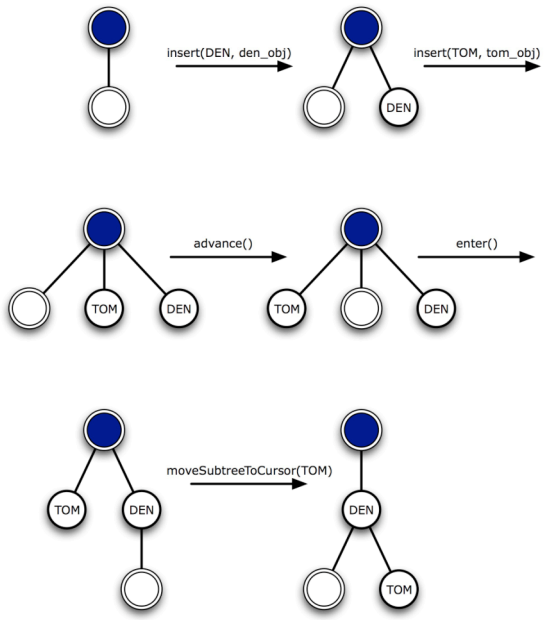

Figure 2. Tree methods

The elements of an indexed tree are organized as an ordered tree [8]. An ordered tree contains a root node, which is the ancestor of all the other nodes in the tree. Every node except for the root has a parent. Nodes with the same parent are siblings. In an ordered tree, the siblings are ordered. There is a first, or eldest, sibling; and there is a last, or youngest, sibling. The indexed tree data structure is traversable. That is, a tree has a conceptual location known as the cursor position. We conceptualize the cursor position as a distinguished node in the ordered tree. The tree component provides various cursor movement methods that can be used to easily change the cursor's location in the tree. Insertion and removal of nodes from a tree occurs to the right of the cursor. The component is called an *indexed* tree because all tree nodes are indexed, or labeled, with a unique identifier. This allows individual nodes to be accessed directly.

Figure 2 gives a graphical representation of various states of an indexed tree object and shows the method calls that cause the transitions from one state to another. The first tree in this figure represents an initialized tree. It is the tree that is created when the default constructor is called. It has two nodes – a root node and a cursor. The cursor is the child of the root. The call insert(DEN, den_obj) inserts a new node to the right of the cursor. The new node is the cursor's younger sibling. The new node DEN is associated with the object den_obj. A second call to insert with

label TOM and object tom_object inserts a new node to the right of the cursor, just before the DEN node. The call advance() advances the cursor past its next node, TOM, and enter() causes the cursor to enter the subtree induced by its next node, DEN. A somewhat more sophisticated method call, moveSubtreeToCursor(TOM), causes the subtree induced by TOM to be moved just to the right of the cursor. It requires that the TOM be in the tree and that the cursor is not a descendent of TOM.

4.1 Mathematical Model for Indexed Tree

Figure 3 gives the mathematical model for the indexed tree component. It contains four model variables that specify how an indexed tree object is modeled [9]. The first three variables are based on the mathematical model for ordered trees given in Cormen et al. [8], in which graphs are modeled as two sets – a vertex set and an edge set – and a tree is an acyclic, undirected, connected graph. The keyword model indicates that a variable is part of the mathematical model. The variable *nodes* denotes the vertex set, and *edges* denotes the edge set. The vertex pairs in the edge set are unordered, so edge (3, 1) is the same as edge (1, 3). The variable *order* denotes the order for children of the same parent from eldest to youngest. The order of the eldest child is 1, the second eldest child is 2, and so on. The final model variable, *contents*, maps nodes to objects.

```
public interface IndexedTree {
   model nodes:     set of Enum;
   model edges:     set of pair of Enum;
   model order:     function from Enum to Integer;
   model contents:  function from Enum to Object;
   defines ROOT, CSR: nodes;
      constraints /* no cycles */

      public IndexedTree( );
         ensures nodes = { ROOT, CSR } and
            edges = { (ROOT, CSR) } and
            order = { (ROOT, 1), (CSR, 1) } and
            contents = { (ROOT, null), (CSR, null) };
```

Figure 3. Model and constructor for indexed tree

The defines clause defines two distinguished variables that belong to the set *nodes*. Conceptually, ROOT is the root node, and CSR is the cursor. A class invariant (given by the constraints clause) asserts that no cycles exists in the undirected graph represented by the nodes and the edge set. The assertion is given here informally.

The constructor creates an indexed tree with ROOT and CSR as nodes, and an edge connecting ROOT to CSR. Both root and cursor have an order of 1 and map to null objects.

4.2 Cursor Movement Methods

A key feature of the tree data structure is the flexibility of cursor movement. Figure 3 specifies some selected cursor movement methods. For the other cursor movement methods, see [10].

The advance method advances the cursor to the next node on the same level. It requires that the cursor have a younger sibling to advance past. In the ensures clause, a variable with a hash, such as #nodes, refers to its original (or old) value, and a variable without a hash refers to its current (or new) value. The only change in the state is that the cursor and its immediate younger sibling, #next(CSR), get their orders swapped.

```
public void advance();
   requires hasYoungerSibling(CSR);
   ensures nodes = #nodes and edges = #edges and
      order(x) = (  #order(x) – 1   if x = #next(CSR);
                    #order(x) + 1   if x = CSR;
                    #order(x)       otherwise ) and
      contents = #contents;

public void enter();
   requires hasYoungerSibling(CSR);
   ensures nodes = #nodes and
      edges = #edges  minus { (#parent(CSR), CSR) }
                      union { (#next(CSR), CSR) } and
      order(x) = (  1                 if x = CSR;
                    #order(x) – 1     if #isYoungerSibling(x, CSR);
                    #order(x) + 1     if #isChild(x, #next(CSR));
                    #order(x)         otherwise ) and
      contents = #contents;

public void moveBefore(restores Enum key);
   requires key in nodes;
   ensures nodes = #nodes and
      edges = #edges  minus { (#parent(CSR), CSR) }
                      union { #parent(key), CSR) } and
      order(CSR) = (
         #order(#key) – 1  if #isYoungerSibling(key, CSR);
         #order(#key)      otherwise ) and
      order(key) = (
         #order(#key)      if #isYoungerSibling(key, CSR);
         #order(#key) + 1  otherwise ) and
      order(x ≠ CSR, key) = (
         #order(x) – 1     if #isYoungerSibling(x, CSR) and
                              not #isYoungerSibling(#key, x);
         #order(x) + 1     if #isYoungerSibling(x, #key) and
                              not #isYoungerSibling(CSR, x);
         #order(x)         otherwise ) and
      contents = #contents;
```

Figure 4. Cursor movement methods

The enter method makes the cursor the first child of its next node. It requires that the cursor have a younger sibling. The nodes and contents remain unchanged. The original edge involving the cursor is replaced by an edge from the cursor's original next sibling to the cursor. The original younger siblings of the cursor get their orders decremented. The cursor advances to the next level and becomes the eldest child of its new parent, so its order is 1, and the cursor's new younger siblings get their orders incremented.

The moveBefore method takes key as an argument. It requires that key be in the node set. It ensures that the cursor will be moved directly before key. That is, the cursor will become key's immediate older sibling. The node set does not change. The original edge to the cursor is replaced by an edge from key's parent to the cursor. If the cursor is the younger sibling of key, the cursor's order becomes one less than the original order of key and key's order stays the same. Otherwise, the cursor's order becomes the original order of key and key's order is incremented. For all other nodes, the order of the cursor's younger siblings are decremented, and the order of key's younger siblings are incremented. However, if a node is a younger sibling of both the cursor and key, its order remains unchanged. The contents map remains unchanged. The **restores** parameter mode for key indicates that the value of key remains unchanged, even though this is not explicitly stated in the ensures clause.

4.3 Insert and SwapValue Methods

Inserting elements into and removing elements from data structures affects how programs are designed in Java and Tako. In Java, updating an element inside a data structure means getting a handle to the element and updating the handle. This updates the element inside the data structure because the handle is an alias to it. In Tako, such aliasing is avoided. Therefore, the element must be removed from the data structure, updated, and put back into the data structure in the same place it was at originally.

```
public void insert(restores Enum key, clears Object val);
   requires key not_in nodes;
   ensures nodes = #nodes union { key } and
      edges = #edges union { (#parent(CSR), #key) } and
      order(x) = (
         #order(CSR) + 1    if x = #key;
         #order(x) + 1          if #isYoungerSibling(x, CSR);
         #order(x)              otherwise ) and
      contents = #contents union { (#key, #val) };

public void swapValue(updates Object val);
   requires hasYoungerSibling(CSR);
   ensures nodes = #nodes and
      edges = #edges and order = #order and
      contents = #contents override { (#next(cursor), #val) } and
      val = #contents(#next(cursor));
```

Figure 5. Insert and swap methods

The methods shown in Figure 5 modify the tree by inserting nodes and swapping values from it. We do not discuss how to remove nodes, but a description can be found in [10].

The insert method inserts a node into the tree as the immediate younger sibling of the cursor. key becomes the new node, and val is the contents of that node. The method requires that key is not already in the indexed tree. key is added to the node set, and an edge to key is added to the edge set. The order of key is one more than the order of the cursor, and the order of the nodes following key are incremented. The **clears** parameter mode for val indicates that val has an initial value after the call. Since the val object is inserted into the tree, the val parameter must hold a different object after the call. Were it to have a *restores* parameter mode, like key, it would force the implementer to perform a deep copy of val, which could be a potentially expensive operation. The key object is also inserted into the tree, but key is a small object (an Enum) so copying it is inexpensive.

The swapValue method swaps the contents of cursor's next node with val. It requires that the cursor have a younger sibling. It ensures that the node set, edge set and order map remain unchanged. The existing contents of the node gets the original object in val, and val gets the original contents of the node. The **updates** parameter mode indicates that the value of val is updated.

5. USING THE INDEXED TREE

5.1 The GameWorld and its GameObjects

The IndexedTree component is used in the implementation of the GameWorld component. The game world is an indexed tree whose nodes are identifiers that are mapped to game objects. A game object inherits from the GameObject class. The GameObject class includes two fields: one for a unique identifier, and another for a set of properties, as shown in Figure 6. Both ObjectID and Property are enumeration types.

```
public interface GameObject {
   model id:          ObjectID;
   model properties:  set of Property;

   public GameObject( )
      ensures id = VOID and properties = { };

   public void addProperty(restores Property p)
      ensures properties = #properties union { #p };

   /* other operations */
}
```

Figure 6. GameObject specification

An object identified by DEN of type Room might include the property LIGHT so that players can see objects in the room. An object identified by TOM of type Actor might include the property PERSON so that the player can talk to it, and the property MALE so that the game's printer knows what pronoun to use when referring to the object. An object identified by BOX might include the property BIN so that players can place other objects inside it. If it has the property OPEN a player may be able to see its contents.

```
public class GameWorld {
   model nodes:    set of ObjectID;
   model edges:    set of pair of ObjectID;
   model order:    function from ObjectID to Integer;
   model contents: function from ObjectID to GameObject;
   defines ROOT: nodes;
      constraints /* no cycles */

   /* gameTree maps ObjectID to GameObject */
   private IndexedTree gameTree;

   correspondence
      conc.ROOT = ROOT and
      conc.nodes = gameTree.nodes – { CSR } and
      conc.edges = gameTree.edges –
         { (parent(CSR), CSR) } and
      forall x in conc.nodes,
         conc.order(x) = (
            gameTree.order(x) – 1 if isYoungerSibling(x, CSR);
            gameTree.order(x)     otherwise ) and
      conc.contents = gameTree.contents –
         { (CSR, gameTree.contents(CSR)) }

   public void setObjectProperty(  restores ObjectID obj,
                                   restores Property prop)
      requires obj is_in nodes;
      ensures nodes = #nodes and
         edges = #edges and order = #order and
         contents = #contents override
            { (#obj, [  #contents(#obj).id,
                        #contents(#obj).property union {#prop} ] ) }
   {
      GameObject rec;
      gameTree.moveBefore(obj);
      gameTree.swapValue(rec);
      rec.addProperty(prop);
      gameTree.swapValue(rec);
   }

   /* other operations */
}
```

Figure 7. GameWorld class

The game world component is implemented with an indexed tree. During game play, each game object is represented by node in the tree. Commands input by the player can potentially update the game world, either by updating the configuration of the tree, or updating the properties of the game objects inside the tree. A portion of the GameWorld class is shown in Figure 7.

The conceptual model of the game world shares the same model variables as the indexed tree, but where the indexed tree is modeled using generic Enums and Objects, the game world uses ObjectID and GameObject types. Also, the game world model does not require a cursor node.

GameWorld contains a single field, the game tree, which is an indexed tree assumed to contain game object identifiers as keys and game objects as values. The correspondence clause, also known as the abstraction relation, describes how to derive the state of the conceptual game world from the state of the internal game tree. The game world is very similar to the game tree except that there is no cursor node and no edge involving the cursor. The order of all younger siblings of the cursor in the game tree are decremented to get their new orders in the game world. The contents are the same except that there is no mapping involving the cursor.

The method setObjectProperty is used in the game to add properties to game objects. It ensures that the structure of the game tree – the nodes, edges, and order – remains unchanged. The new property addition is reflected in the change to the contents variable. The next subsection traces through a particular call to the method, showing how the implementation meets its specification for a particular input.

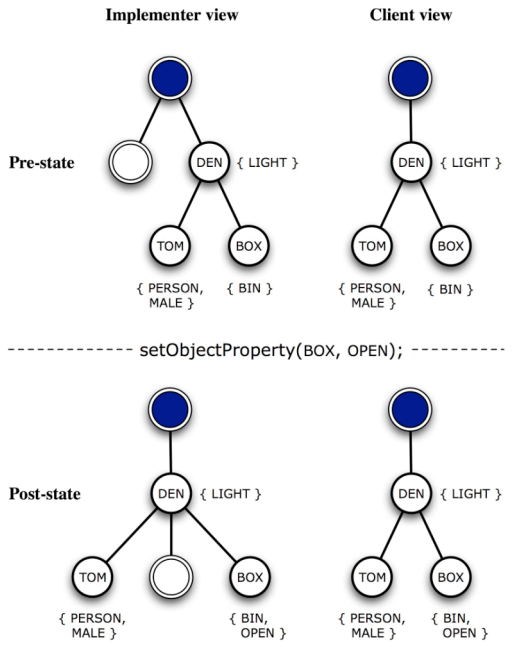

Figure 8. Pre-state and post-state of setObjectProperty call

5.2 Tracing Through a Method

Figure 8 shows a pre-state and the resulting post-state for the call to setObjectProperty(BOX, OPEN). Two views are shown: The implementer view and the client view. The implementer view shows the game tree, which includes a cursor, and the client view shows the game world, in which no cursor is present. Properties for each game object are given next to their corresponding nodes in the trees.

The tracing table in Table 2 shows the state of the program while stepping through the statements in the implementation of the setObjectProperty method. The initial state, state 0, describes the game tree that corresponds to the implementer view of the pre-state in Figure 8. The game tree has five nodes: ROOT, CSR, DEN, TOM, and BOX; and four edges, corresponding to those shown in Figure 8. The order mapping gives the appropriate sibling ranking of the nodes. The contents mapping maps nodes to game objects. In the tracing table, we represent game objects as sets of properties, omitting the redundant identifiers for brevity. The last line of each fact contains the values of the formal parameters and local variables. State 0 begins after the local variable rec is declared.

Table 2. Trace of setObjectProperty (implementer view)

St	Facts
0	nodes = { ROOT, CSR, DEN, TOM, BOX } edges = { (ROOT, CSR), (ROOT, DEN), (DEN, TOM), (DEN, BOX) } order = { (ROOT, 1), (CSR, 1), (DEN, 2), (TOM, 1), (BOX, 2) } contents = { (ROOT, { }), (CSR, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, {BIN}) } obj = BOX **and** prop = OPEN **and** rec = { }
gameTree.moveBefore(obj);	
1	nodes = { ROOT, CSR, DEN, TOM, BOX } edges = { (ROOT, DEN), (DEN, TOM), (DEN, CSR), (DEN, BOX) } order = { (ROOT, 1), (DEN, 2), (TOM, 1), (CSR, 2), (BOX, 3) } contents = { (ROOT, { }), (CSR, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, {BIN}) } obj = BOX **and** prop = OPEN **and** rec = { }
gameTree.swapValue(rec);	
2	nodes = { ROOT, CSR, DEN, TOM, BOX } edges = { (ROOT, DEN), (DEN, TOM), (DEN, CSR), (DEN, BOX) } order = { (ROOT, 1), (DEN, 2), (TOM, 1), (CSR, 2), (BOX, 3) } contents = { (ROOT, { }), (CSR, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, { }) } obj = BOX **and** prop = OPEN **and** rec = {BIN}
rec.addProperty(prop);	
3	nodes = { ROOT, CSR, DEN, TOM, BOX } edges = { (ROOT, DEN), (DEN, TOM), (DEN, CSR), (DEN, BOX) } order = { (ROOT, 1), (DEN, 2), (TOM, 1), (CSR, 2), (BOX, 3) } contents = { (ROOT, { }), (CSR, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, { }) } obj = BOX **and** prop = OPEN **and** rec = {BIN, OPEN}
gameTree.swapValue(rec);	
4	nodes = { ROOT, CSR, DEN, TOM, BOX } edges = { (ROOT, DEN), (DEN, TOM), (DEN, CSR), (DEN, BOX) } order = { (ROOT, 1), (DEN, 2), (TOM, 1), (CSR, 2), (BOX, 3) } contents = { (ROOT, { }), (CSR, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, {BIN, OPEN}) } obj = BOX **and** prop = OPEN **and** rec = { }

The statement gameTree.moveBefore(obj) moves the cursor to the position just before BOX. To check if the operation is permissible, we must look at the requires clause of the indexed trees

moveBefore method given in Figure 4. It requires that key (the formal parameter that corresponds to obj) be in the node set. Here, key = BOX, which is in the node set in state 0 just before the call is made, so the precondition is satisfied. To get the facts in state 1, we apply the ensures clause of moveBefore to the facts in state 0.

From the ensures clause for moveBefore we know that the only variables in the program state that change are edges and order. The edge from ROOT to CSR is replaced by an edge from DEN to CSR. BOX is not a younger sibling of CSR, so the order of CSR becomes 2 (the old order of BOX), and BOX's order is incremented. DEN is originally a younger sibling of CSR, so its order is decremented. The order of all other nodes is unchanged. The effects of the other statements are reasoned about similarly.

To verify that the implementation is correct with respect to the specification for this particular start state, we need to translate the first and last states from the implementer view to the client view using the correspondence clause, and then see if they conform to the specification of setObjectProperty. Table 3 is a tracing table for setObjectProperty after this translation. State 0 in Table 3 is derived from state 0 in Table 2 and state 1 is derived from state 4. Applying the ensures clause of setObjectProperty to the facts in state 0 results in the facts in state 1, so the implementation is correct in this instance.

Table 3. Trace of setObjectProperty (client view)

St	Facts
0	**conc**.nodes = { ROOT, DEN, TOM, BOX } **conc**.edges = { (ROOT, DEN), (DEN, TOM), (DEN, BOX) } **conc**.order = { (ROOT, 1), (DEN, 2), (TOM, 1), (BOX, 2) } **conc**.contents = { (ROOT, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, {BIN}) }
setObjectProperty(BOX, OPEN);	
1	**conc**.nodes = { ROOT, DEN, TOM, BOX } **conc**.edges = { (ROOT, DEN), (DEN, TOM), (DEN, BOX) } **conc**.order = { (ROOT, 1), (DEN, 2), (TOM, 1), (BOX, 2) } **conc**.contents = { (ROOT, { }), (DEN, {LIGHT}), (TOM, {PERSON, MALE}), (BOX, {BIN, OPEN}) }

6. OTHER DESIGN ISSUES

6.1 Singleton Design Pattern

A design pattern that was used extensively in the Java version of the game, but could not be reproduced in the Tako version was the singleton pattern, whose intent, according to the patterns book of Gamma et al. [11], is to "Ensure that a class only has one instance, and provide a global point of access to it" (p. 127). Note that the statement says nothing about references and nothing about aliasing. However, a typical implementation of the singleton permits aliasing everywhere, as shown in Figure 9.

In this implementation, every singleton variable is a reference to the same object. In practice, however, there is never more than one singleton variable in a class, and there is no benefit from sharing the same object through references over using the object itself. One way to use the same object without aliases is through a global variable. Gamma et. al. note that global variables provide access but criticize global variables for two reasons: They do not prevent multiple instances, and they pollute the global namespace. The first criticism can easily be addressed in Java using the singleton class in Figure 10 in which the constructor is only usable from inside the class itself. The second criticism seems to imply that it is more difficult to reason about the singleton client in Figure 10 than the singleton client in Figure 9 because global variables make reasoning difficult. But this criticism seems odd to us since aliased variables require reasoning about the global heap, a structure as complicated as any variable in a typical program.

The current Tako compiler does not implement static import variables, so a class whose sole purpose was to hold global variables was constructed, and we were disciplined about not declaring instances of, for example, GameWorld, anywhere but inside that class. Clearly this is not a satisfactory solution, so the next version of the Tako compiler will have the ability to implement the singleton as illustrated in Figure 10.

```
class GameWorld {
   GameWorld world = new GameWorld();
   private GameWorld() { /* constructor body */ }
   public static GameWorld getInstance () { return world; }
   /* remainder of class */
}

import GameWorld;
class Resolver {
   GameWorld world = GameWorld.getInstance();
   /* class body uses 'world' variable */
}
```

Figure 9. Typical singleton implementation

```
class GameWorld {
   public static GameWorld world = new GameWorld();
   private GameWorld() { /* constructor body */ }
   /* attributes and methods */
}

import static GameWorld.world;
class Resolver {
   /* class body uses global 'world' variable */
}
```

Figure 10. Singleton implemented with single instance global

6.2 Tako-Java Integration

Tako is syntactically similar to Java, and the current Tako compiler translates to Java, so the current version of Tako is closely tied to the Java language. Given this, we want to ensure that Tako and Java are as compatible as possible. The Tako implementation of the adventure game uses Java Swing components for its graphical user interface. Our initial experience in integrating Java and Tako components helped us come up with a few basic rules, some of which have been applied in the current adventure game, and some of which will have to wait for the next version of the Tako compiler.

A Tako class can use a Java class. Currently, a Tako class simply imports the Java class, but future versions of the compiler should require a special import statement such as "import java". The Tako compiler should translate the Java method call as is. A Java method should not take non-Java arguments. If a Tako variable x is found, the compiler will view it as the function call x.toJava_TypeName() where TypeName is the name of the Java type expected. Obviously, such a function will return a value of type TypeName.

If a programmer wants to write a Java class that uses Tako code, they should write a Java interface and implement that interface with a Tako class. Java methods should have Java parameters and return Java values. Tako classes will need to access Tako types that can convert to and from the Java types needed in the interface methods. For example, if there is a Java method whose signature is String processText(String x), then the Tako Text class should have a method String toJava_String() and a constructor Text(String x).

7. DISCUSSION AND CONCLUSION

A typical Java version of the adventure game might use the composite design pattern to implement the game world. While the composite pattern can be implemented in Tako, the recursive type structures involved raise the possibility that null references will be assigned to variables, which, in general, is something we prefer to avoid. The design of the Tako program uses the game world as a point of centralized access and control for the tree structure and all of its contents. Many object-oriented programmers prefer designs using distributed control rather than centralized control [12]. The Tako language does not preclude designs with distributed control, but formally reasoning about designs can be a challenge. We plan to more explore this topic more thoroughly in future research.

Since all the game objects are stored in a tree like data structure, accessing these objects in Tako meant swapping them out of the structure, examining them, and then swapping them back in. In the Java version, programmers modify a game object through a handle or reference to the object. The swapping in Tako initially lead to errors while inserting them back in the tree as the conceptual cursor position in the tree was unexpectedly modified. The error was easily fixed, but it represents one example of an error that would not occur in Java as the object is never removed in the first place.

In the Java version of the game, the container classes like stacks, hash maps, queues, and lists were already provided by the java.util package. But in the Tako version, these util classes where implemented from scratch using the pointer component. The pointer component was only used in these low level classes; while the higher design level classes did not require the use of the pointer component. This supported our conjecture that the programmer, at higher level of design, can make do without using references.

In this particular program the distinction between object identity and name identity did not play a major role. We used hash maps to store the game objects and each game object had a unique key associated with it. In both the Java and Tako version, it was these unique keys rather than the language dependent object identity that was used for uniquely identifying the objects.

Overall, we found that the paradigm shifts involved were not very difficult to adjust to though they required some alertness in terms of the swapping paradigm. This experience is consistent with that of Hollingsworth et al., who discuss a sizeable commercial application developed in C++ using the Resolve discipline [13]. In their report, they concluded that swapping worked well with most object-oriented techniques.

We would like to implement the adventure game in Java (or perhaps Tako) using the composite design pattern to further explore the benefits and drawbacks of using a tree data structure rather than a composite pattern. Ultimately, we would like to bootstrap the Tako compiler using a design based on formally specified data types. The source code for the current Tako compiler and for the adventure game can be found in the takocompiler project on Sourceforge [14].

8. REFERENCES

[1] Kulczycki, G. and Vasudeo, J. Simplifying Reasoning about Objects with Tako. In *Proceedings of the Specification and Verification of Component-Based Systems 2006* (2006).

[2] Sitaraman, M. and Weide, B. W. Component-Based Software using Resolve. *ACM Software Engineering Notes*, 19, 4 1994), 21-76.

[3] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S. and Hollingsworth, J. E. Reasoning about Software Component Behavior. In *Proceedings of the International Conference on Software Reuse* (2000). Springer-Verlag.

[4] Hogg, J., Lea, D., Wills, A., deChampeaux, D. and Holt, R. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger*, 3, 2 1992), 11-16.

[5] Clarke, D. *Object Ownership and Containment*. University of New South Wales, 2001.

[6] Nelson, G. and Rees, G. *The Inform Designer's Manual: 4th Edition*. Dan Sanderson (pub.), 2001.

[7] Roberts, M. J. *TADS - The Text Adventure Development System*. City, 2006.

[8] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. *Introduction to Algorithms: 2nd Edition*. MIT Press, 2003.

[9] Cheon, Y., Leavens, G., Sitaraman, M. and Edwards, S. Model Variables: Cleanly Supporting Abstraction in Design by Contract. *Software - Practice and Experience*, 35, 6 2005), 583-599.

[10] Kulczycki, G. *The Tako Component Library*. City, 2008.

[11] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Adison-Wesley, 2003.

[13] Hollingsworth, J. E., Blankenship, L. and Weide, B. W. Experience Report: Using Resolve/C++ for Commercial Software. In *Proceedings of the International Symposium on Foundations of Software Engineering* (2000).

[14] Kulczycki, G. and Vasudeo, J. *The Tako Compiler Project*. City, 2006.