Tree Component alternatives
to the
Composite Design Pattern

Arun Sudhir

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
In partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science and Applications

Dr. Gregory Kulczycki, Chair
Dr. Ing Ray Chen
Dr. Chang Tien Lu

December 3rd, 2008
Falls Church, Virginia, USA

Keywords: Design Patterns, Tree, Indexed Tree, Composite Pattern, Formal Reasoning, Tako, Java

Copyright © 2008, Arun Sudhir
Tree Component alternatives to the
Composite Design Pattern

by Arun Sudhir

Abstract

The Composite design pattern is commonly employed in object-oriented languages to design a system of objects that form a part-whole hierarchical structure with composite objects formed out of primitive objects. The client does not differentiate between a composite object and a primitive object. The composite hierarchy effectively forms a tree-like hierarchical grouping of objects. From a software engineering perspective, there are at least two problems with the Composite pattern. First, it does not maintain a separation of concerns between the structure of the objects in a system and the objects themselves. The objects that comprise the system contain information about their relationship to other objects. This limits the ability of programmers to reuse the system’s structural information. Secondly, there is no mechanism for encapsulating the system as a whole. This makes it difficult to specify and reason about global system properties. This thesis presents two tree components that can be used as alternatives to the Composite design pattern in systems that are traditionally implemented with the pattern. Both components are data structures that can contain arbitrary objects and maintain the structure of those objects as an ordered-tree. Since the components encapsulate only the tree structure, they only need to be specified and verified once, and they are available for black-box reuse. The first component is a traversable tree that maintains a conceptual “cursor” position. Methods are provided for inserting and removing objects at the cursor position, and for moving the cursor throughout the tree. The second component extends the traversable tree. A formal specification for each tree component is presented in the Tako language – a Java-like language with alias avoidance that is designed to facilitate specification and verification. A case study is presented that shows how the indexed tree can be used and reasoned about in an application – a text-based adventure game. Finally, a similar application is developed in Java, once using the composite pattern and once using the indexed tree data structure, and object-oriented metrics are given for both systems.
# Table of Contents

List of figures and tables

Acknowledgments

1. Introduction

2. The Tree Component
   2.1 Specification of the Tree
   2.2 The Tree Interface
   2.3 Mathematical Model
   2.4 Data Modification Methods
   2.5 Cursor Movement Methods
   2.6 Query Methods
   2.7 Application

3. The Indexed Tree Component
   3.1 Specification of the Indexed Tree
   3.2 The Tree Interface
   3.3 Mathematical Model
   3.4 Cursor Movement Methods
   3.5 Data Modification Methods
   3.6 Query Methods
   3.7 Application

4. The text-based adventure game – A case study in Tako.
   4.1 Overview
   4.2 Adventure Game Architecture
   4.3 Game object specification
   4.4 Game world specification
   4.5 The setObjectProperty method
   4.6 Trace of setObjectProperty (Implementer View)
4.7 Trace of setObjectProperty (Client View) 34
4.8 Discussion 35

Java case study – comparison and metrics 37 - 43

5.1 The Composite Pattern Version 37
5.2 The Indexed Tree Version 37
4.3 Metrics 39
4.4 Discussion and Comparison 40

Conclusion 44 - 45
Bibliography 46 - 47
List of Figures

Figure 1.1 Composite design pattern 1
Figure 2.1 Tree methods and representation 5
Figure 2.2 Alternative view of the tree 6
Figure 2.3 Tree Specification 7
Figure 2.4 Mathematical model of the tree 7
Figure 2.5 Data modification methods 9
Figure 2.6 Cursor movement methods 10
Figure 2.7 Query methods 12
Figure 3.1 An illustration of the indexed tree methods 15
Figure 3.2 The indexed tree interface 16
Figure 3.3 Model and constructor for indexed tree 17
Figure 3.4 Cursor movement methods 19
Figure 3.5 Data Modification methods 22
Figure 3.6 Query methods 23
Figure 4.1 Adventure Game Architecture 27
Figure 4.2 The GameObject specification 28
Figure 4.3 The GameWorld specification 29
Figure 4.4 Pre-state and Post-state of the setObjectProperty call 31
Figure 5.1 Architecture of the Composite Pattern version 38

List of Tables

Table 4.1 Trace of setObjectProperty (implementer view) 32
Table 4.2 Trace of setObjectProperty (client view) 35
Table 5.1 Metrics for the Composite Pattern version 41
Table 5.2 Metrics for the Indexed Tree version 41
Table 5.3 Metrics for the adventure.universe package in the Composite Pattern version 41
Table 5.4 Metrics for the adventure.universe package in the Indexed Tree version 42
Acknowledgments

Foremost, I would like to express my heartfelt gratitude to my advisor, Gregory Kulczycki, for his ideas, support, and time for the time I was a graduate student under him. I would not have gained so much knowledge and deep insight into the subject without him. His ideas, insight and suggestions went a long way in laying the foundation of the thesis. He has guided me patiently all the way along my tenure as a graduate student and has provided me a meaningful direction to my research and shaping the thesis. His ideas have left an indelible impression on me and have had profound influence on my thinking and his methods have made me strive for perfection. I wish I can apply the learning that I have got from him in my career after graduation. He has been a truly magnificent person to work with both as a professor and as a person. I would also like to express my gratitude to Ing-Ray Chen and Chang-Tien Lu for being on my thesis committee and for their patience for helping me with completing my thesis. I am greatly indebted to all my professors for the knowledge that I have gained as a graduate student.

To my parents – Joseph Sudhir and Geetha Sudhir, and my fiancée Reno – I dedicate this thesis to you. I also thank all my friends for the motivation and self-belief that they have given me. Your belief in my ability and your unlimited encouragement and support has made me achieve whatever I have. Even when I was desperate, you calming influence, love and affectionate words have helped me regain confidence in myself. Thanks for being there.
Chapter 1

Introduction

The Composite Pattern comprises composites out of primitive objects and enables them to be used in the same way as primitives can be used. From a broader perspective, it is a hierarchical design pattern and groups objects into a tree-like hierarchy. The structure of the composite is shown in Figure 1.1. Both the Composite objects and the Leaf objects inherit from the Component interface, which contains the method `operation`. When the `operation` method is called on a Component object, the client does not care if the object is a Leaf or a Composite. Often, a composite-object method such as operation will simply iterate through its child components and call the same method on each. The composite pattern is a popular design pattern. Typical example applications of where it could be used are in a compiler and in a drawing program. In a compiler, the composite can be used to represent an abstract syntax tree. Typical composite objects would include declarations, statements, and expressions; typical leaf objects would include operators, keywords, and identifiers. In a drawing program (for example, PowerPoint), leaf objects would be shapes (such as ovals and rectangles), text, and lines. Composite objects would be any collections of drawing primitives designated by a client as forming a group (for example, by selecting multiple shapes and clicking the “group” button in PowerPoint).

![Figure 1.1 Composite Design Pattern](wik08)
Despite its popularity, the Composite pattern has at least two important drawbacks. The first is that the structure of the objects in the pattern is tightly coupled with the objects themselves. Composite objects include a list of their children, and methods such as add, remove, and getChild are called for each composite rather than for the system as a whole. Contrast this to a typical List component. The List object has methods that control its structure, such as add and remove. The objects it contains are arbitrary, so clients can create lists of integers, lists of files, or lists of processes, simply by instantiating the list with objects of a different type. This allows the code for the list component to be reused without changing anything. For the Composite, the pattern may be applied to different applications, but there is no code reuse.

A tree component separates the structure from the object it contains, so the structure may be reused without modifying any code.

Another drawback of the composite pattern is the lack of a mechanism that allows designers to easily specify the behavior of global system properties. For the same reason, clients that use the composite pattern cannot easily reason about global properties. The problem of specifying and reasoning about the composite design pattern was featured as a challenge problem in the 2008 SAVCBS workshop (Specification And Verification of Component-Based Systems). The following paragraph is an excerpt from the challenge problem section of the call for papers [Sav 08]. An invariant is an assertion about an object that should be true before and after each method call.

“Consider the class, Composite, that wishes to keep the simplest invariant: that each object, c, of type Composite has a count of the number of objects within the sub-tree rooted at c. Therefore, whenever a component is added as a child of c all of c’s ancestors must have their count updated. While c can own (using any of the various ownership methodologies) its children, it cannot own its parent. Yet adding a child to c risks invalidating its parent’s invariant.” [Sav 08]

A tree component maintains all structural invariants within a single object – the tree itself. While it may be complicated to verify that the code of a tree implementation is correct with respect to the class invariants, it only has to be done once for each implementation, because the code can be reused without modification.

Object-based component libraries typically lack tree components. Those that do have them usually limit their use to search trees. The Java class library [Jav 02] does not contain a tree data structure. It does contain a JTree class, but “a JTree object does not actually contain your data; it simply provides a view of the data” [Wal 01]. The C++ Standard Template Library contains list-like structures such as lists, stacks, queues, sets, and maps, but
no tree-like data structure. The Booch library [Boo 91], which was originally designed for Ada but eventually ported to C++, has three tree data structures: a binary tree, an AVL tree, and a multi way tree. All of these are search trees. The first tree we present in this thesis, the traversable tree, is based loosely based on the nested-list data structure found in [Ogd 04]. The traversable tree turned out to work well for application in which a client typically walked the entire tree, such as in the implementation of an abstract syntax tree for a compiler. The second tree, the indexed tree, works well when random access to tree nodes is needed, as in the object tree for a drawing application.

This thesis makes the following contributions to software engineering:

- We present an alternative formal specification for a traversable tree component that is loosely based on the nested-list component in [Ogd 04]. In contrast to the previous specification, our specification has a functional form in which the outgoing values of variables in post-conditions are represented as functions of their incoming values.

- We present an interface and formal specification for an indexed tree component. The specification is based on the well-known mathematical model for ordered trees found in Cormen et al. [Cor 01].

- We demonstrate the use of the indexed tree component in a non-trivial application. We present a tracing table for a method in the application that shows how a client of the indexed tree component can reason about it based on its specification.

- We demonstrate that the indexed tree component can be used in place of the composite pattern in traditional object-oriented languages by implementing two versions of an application in Java – one using the composite pattern and one using the indexed tree component.

The thesis outline is as follows. Chapter 1 introduces the traversable tree component and gives a mathematical model for it. It gives the interface of the tree and shows how the methods affect the mathematical model. Chapter 2 introduces the indexed tree component. The interface and methods of the indexed tree component are examined in detail. Chapter 3 explains how the indexed tree component is used to implement a non-trivial program in the Tako language. We show how to reason about the indexed tree by using a tracing table. Chapter 4 is a comparison of the same program implemented with the indexed tree component as well as with the Composite design pattern in Java. Metrics of the two programs are evaluated and compared. Finally, our observations are summarized in the final chapter.
Chapter 2

The Tree Component

2.1 Specification of the Tree

This section presents a traversable tree component. The tree components in this section and the next are given in the Tako programming language [Vas 06]. Tako is a Java-like language with alias avoidance that is designed to simply specification and reasoning of object-oriented components. The Tako language was designed with specification and verification of component-based systems in mind. Specification and Verification are two important principles of Formal Verification [Boe 82]. Formal Verification of programs is a technique which uses formal reasoning [Kal 80] techniques to verify if programs are implemented in accordance with their specification. The ultimate aim of formal verification is to build a verifying compiler – one in addition to compiling programs, would be able to verify using mathematical techniques – as to whether the program implementation complies with its specification.

The tree component can be viewed as a nested list – a list of other lists. Each list is labeled with an object, and the objects make up the contents of the tree. The tree has a conceptual cursor position. Tree traversal is accomplished by moving the cursor. A client can move the cursor to different parts of the tree using the cursor movement methods. They can insert and remove objects to the right of the cursor.

Figure 2.1 gives an example of how one might go about constructing a tree of tokens. The first statement, ast := new Tree() creates a new tree. A tree is a nested list, so the new tree is denoted as an empty list “< >” with a cursor position, denoted by “|”, that is inside the list. The second statement, ast.insert([minus, “-”]), inserts an empty list into the tree to the right of the cursor position. The empty list is labeled with the minus token that is given as a parameter. The third statement, ast.enter(), puts the cursor inside the list inserted in the previous step. The next statement ast.insert([number,”5”]); inserts a list labeled by the numeric token 5 inside the list labeled with the minus token. This is conceptually equivalent to inserting it as a child of the node labeled with the minus token. The next statement ast.advance() advances the cursor position to the right of the newly inserted node. Finally, the ast.reset() operation resets the cursor position to the beginning before the node labeled minus.
In the representation in Figure 2.1, an empty node is represented by <>. And the cursor is represented using the pipe symbol -“|”. Thus, <|> represents an empty node with the cursor in it. Figure 2.2 presents an alternative view of the tree.
The alternative view in Figure 2.2 shows the hierarchy of nodes in the tree and the conceptual cursor position for the same set of operations as in Figure 2.1. Nodes with similar indentation represent those at the same level in the tree (which were members of the same list in the earlier representation) and are siblings of each other.

2.2 The Tree Interface

Figure 2.3 describes the interface of the tree. It lists all methods in the tree specification. The tree methods can be broadly classified into cursor movement, data modification and query methods. The advance and retreat methods move the cursor back and forth at the same level. The enter method makes the cursor enter into the list immediately to its right. The exit method makes the cursor go up one level. The insert and delete methods insert and delete a node at the cursor position respectively. Query methods return values regarding the state of the tree. For example, the hasNext method checks if there is a sublist immediately after the cursor. We will see how these methods are implemented in detail in the sections to follow.

```java
public interface Tree {
    // Cursor movement methods
    public void advance();
    public void retreat();
    public void enter();
    public void exit();
    public void reset();
    public void toFront();
    public void toEnd();

    // Data modification methods
    public void insert(alters Object label);
    public void remove(replaces Object label);
    public void swapLabel(Object label);
    public void swapRem(Tree t);

    // Query methods
    public Boolean hasNext();
    public Boolean hasPrevious();
```
public interface Tree {

  Definition Site = {
    siteLabel: Object;
    PrecTS: MathString(MathTree);
    RemTS: MathString(MathTree);
  }

  model MathString(Site) Path;
  model MathString(MathTree) Prec;
  model MathString(MathTree) Rem;

  public Tree( );
    ensures Path = <> and Prec = <> and Rem = <>;
}

Figure 2.4 Mathematical model of the tree
The tree constructor in Figure 4 creates a tree containing a list labeled as null. Conceptually, this means that a tree is created with a root node (an empty list). The conceptual cursor position would be in between the preceding and remaining parts of the list.

2.4 Data Modification 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.

The data movement methods modify the structure of the tree or the labels of individual lists (labels of nodes) in the tree. They do so by inserting nodes, removing nodes or by swapping values of the node labels. The insert method shown in figure 2.5 shows how to insert a new node into the tree immediately after the conceptual cursor position. In the ensures clause, a variable with a hash, such as #Path, refers to its original (or old) value, and a variable without a hash refers to its current (or new) value. A label argument of type Object is supplied to the insert method. As the insertion of a new node does not affect the Path or the nodes before it, Path and Prec remain unaltered. The Rem sequence will now include the newly inserted node as its first node. The new node formed by Make_Tre(#label,<>) is concatenated with the incoming value of Rem for the outgoing value. Now, since the label object is inside the tree, Tako's value semantics necessitate that there be no reference to the object outside the tree. So label – the identifier which previously held the object – is now a valid, but unspecified value of its type.

```java
public void insert(alters Object label);
    ensures Path = #Path and
            Prec = #Prec and
            Rem = < Make_Tre(#label, < >) > o #Rem and
            label = ??;
```

```java
public void remove(replaces Object label);
    requires Rem ≠ < > and Subtrees(First(Rem)) = < >;
    ensures Path = #Path and Prec = #Prec and
            Rem = All_But_First(#Rem) and label = First(#Rem);
```
The remove method removes a node at the cursor position. The node immediately after the cursor (the first element of Rem) is removed. The post-condition indicates that there should be at least one node after the cursor (in Rem) for this method to execute. It is also specifies that the node to be removed should be a leaf node and should not contain any children. Once these conditions are met, the node is removed from the tree and its label returned as the input argument label's value. The outgoing value of Rem is the same as the incoming value except that the first node is removed.

The swapLabel method swaps the value of the node immediately after the cursor. It requires that there be a node after the cursor. Changing the label of a node after the cursor does not modify nodes before it or the structure of the Tree till the cursor, Path and Prec remain unchanged. Rem changes to include a new node with the new label in place of the old node.

SwapRem swaps the Rem part of the Tree with the Rem part of the tree given as argument. As a result, the Prec and Path parts of both the trees are unaffected and just the Rem parts get swapped.

2.5 Cursor Movement Methods

The cursor movement methods move the position of the conceptual cursor within the tree. They do not change the structure of the tree or the values of nodes in the tree. Figure 6 shows the cursor movement methods. These do not affect the structure of the tree but move the cursor to different places within the tree.

The remove method removes a node at the cursor position. The node immediately after the cursor (the first element of Rem) is removed. The post-condition indicates that there should be at least one node after the cursor (in Rem) for this method to execute. It is also specifies that the node to be removed should be a leaf node and should not contain any children. Once these conditions are met, the node is removed from the tree and its label returned as the input argument label's value. The outgoing value of Rem is the same as the incoming value except that the first node is removed.

The swapLabel method swaps the value of the node immediately after the cursor. It requires that there be a node after the cursor. Changing the label of a node after the cursor does not modify nodes before it or the structure of the Tree till the cursor, Path and Prec remain unchanged. Rem changes to include a new node with the new label in place of the old node.

SwapRem swaps the Rem part of the Tree with the Rem part of the tree given as argument. As a result, the Prec and Path parts of both the trees are unaffected and just the Rem parts get swapped.
Figure 2.6 Cursor movement methods

The reset method moves the cursor to just beneath the root node. So the Path variable, which represents how many levels down in the tree the cursor is, becomes empty. Prec is also empty as there are no nodes before the cursor position. Rem would be everything inside the incoming value of prec and rem zipped together as all the nodes in the tree are now inside the rem part of the tree.

The advance method advances the cursor by one node at the same level. It requires that there be a node to advance to. That means that the cursor should not be placed as the last node at a level. Since advance advances the cursor and the cursor would always be between Prec and Rem, the new Prec would now have the node that
the cursor advanced over. And that value would have been the first value in the incoming value of Rem. Since this occurs at the same level, Path remains unchanged.

The retreat method performs the exact opposite of what the advance method does. It retreats the cursor to one node before where it was. For this to happen, Prec should at least have one node for the cursor to move behind. The last node of the Prec part would now move over as the first node in Rem. So Prec loses its last node and Rem gains it as its first node.

The toFront method retreats the cursor before the first node at that level. So it performs a series of retreats till it reaches the first node and no more retreats are possible. The toEnd method does just the opposite. It advances the cursor to the last node at the same level.

The enter method puts the cursor inside the node that it preceded before the call. So the enter method requires that the cursor be before a node (that Rem should not be empty) which ensures that the cursor has a node to enter into. Since this changes the level on the tree that the cursor is in, another site gets added to the Path with the label of the entered node and the PrecTS set to the incoming value of Prec and RemTS set to the incoming value of Rem minus the node itself. The cursor should not be on top of the tree (at the root). That is indicated by the ensures clause.

The exit method does just the opposite of what the enter method does. The cursor ascends one level in the tree and moves before the parent of the node it was stationed at. The Path variable is modified to exclude the last variable it had.

### 2.6 Query methods

The query methods are listed in figure 2.7. These neither change the structure of the tree nor do they move the cursor. These are querying methods which return boolean and integer values based on the tree properties.

The hasNext method returns true if there are nodes after the current cursor position at that level and false otherwise. The hasPrevious method returns true if Prec is not empty meaning that there are Nodes at the current level before the cursor position.

The hasOuter method returns true if a node has a parent. This method returns true for all nodes except the root. The isEmpty method checks if the tree has only the root when it is created and not any other nodes on it.

The Preclength and Remlength methods return the number of nodes in the tree before and after the cursor position respectively. Pathlength indicates what level or how deep inside the tree the cursor is.
public Boolean hasNext();
ensures (Rem ≠ < >);

public Boolean hasPrevious();
ensures (Prec ≠ < >);

public Boolean hasOuter();
ensures (Path ≠ < >);

public Boolean isEmpty();
ensures Path = < > and Prec = < > and Rem = < >;

public Integer precLength();
ensures result = |Prec|;

public Integer remLength();
ensures result = |Rem|;

public Integer pathLength();
ensures result = |Path|;
}

Figure 2.7 Query methods

2.7 Application

Applications which warrant the use of a tree data structure can effectively make use of the tree specification discussed here. This tree structure can be used to implement designs based on design patterns such as the Visitor pattern and the Composite pattern which are difficult to reason about and are bound to introduce a significant amount of aliasing when implemented in conventional programming languages like C++ and Java. Such designs, when done in a language based on value semantics rather than reference semantics are easier to reason about and can be formally verified in a trivial manner. The alias avoidance and value semantics model of the Tako language makes it an ideal platform to implement designs based on hierarchies of objects using this tree specification.

This model of the tree can be successfully utilized in an abstract syntax tree. Further research involves modeling the abstract syntax tree for the bootstrap version of the Tako compiler based on this tree model.
The Indexed Tree Component

3.1 Specification of the Indexed Tree

This section presents a custom component called the Indexed tree. In implementing and using the tree in place of the composite design pattern, there is sometimes the need to traverse the hierarchical collection of objects in a random fashion and to find and modify a particular node in the tree. Instead of traversing the tree in a hierarchical fashion form the root, there is the need to go to a specific node. For that purpose, the indexed tree component is designed so as to facilitate lookup of a node from a key. The elements of an indexed tree are organized as an ordered tree. An ordered tree contains a root node, which is the ancestor of all the other nodes in the tree. Except that root node, every other node has a parent node. Children of the same node are siblings of each other. Siblings of the indexed tree are ordered with the first child being the eldest sibling, and subsequent children being younger siblings. Therefore, the last node inserted for a parent node is the youngest sibling of all the other nodes at that level. Similar to the tree component, the indexed tree component has a conceptual location known as the cursor position. We conceptualize the cursor position as a distinguished node in the ordered tree. Just like the tree component, the indexed tree component has cursor movement methods and data modification methods. As in the tree component, insertion and removal of nodes from a tree occurs to the right of the cursor. All nodes in the tree are indexed or labeled with a unique identifier and hence the name indexed tree. The indexing feature allows individual nodes to be accessed directly.

A graphical representation of various states of an indexed tree object is shown in Figure 3.1. It also shows the method calls that cause the transitions from one state to another. The first tree in this figure represents an initialized indexed tree. It has two nodes – a root node and a cursor. It is the tree that is created when the default constructor is called. The cursor is the child of the root. The call insert(KITCHEN, kitchen_obj) inserts a new node to the right of the cursor. The new node is the cursor’s younger sibling. In the method call, the first argument represents the enum type KITCHEN that is used to represent the node in the indexed tree and the second argument kitchen_obj is the actual kitchen object. The enum type object gets to be the node label and the label and the actual object that it corresponds to are stored in the map. To store the key and the actual value in the map, a replica or a deep copy of that Enum object is created. As it is not a heavy object, the deep copy incurs little cost. Thus the node labeled by KITCHEN is associated with the object kitchen_obj in the map. A
second call to insert with label BOB and object bob_obj inserts a new node to the right of the cursor, just before the KITCHEN node. The call advance() advances the cursor past its next node, BOB, and enter() causes the cursor to enter the subtree induced by its next node, KITCHEN. A somewhat more sophisticated method call, moveSubtreeToCursor(BOB), causes the subtree induced by BOB to be moved just to the right of the cursor. It requires that the BOB be in the tree and that the cursor is not a descendant of BOB.

3.2 The Indexed Tree Interface

The indexed tree interface is listed in figure 3.2. Like the tree interface discussed in the previous section, the
indexed tree methods too can be broadly classified into cursor movement methods, data modification methods and query methods. The categories have similar meanings to the corresponding categories discussed earlier for the tree. The cursor movement methods offer the flexibility of cursor movement using the advance, retreat, enter and exit methods. Additionally, the moveBefore method also allows cursor movement to the node specified as its argument. The data modification methods alter the structure of the tree by inserting or deleting nodes form it or by swapping the values stored at the nodes. Instead of the swapRem functionality presented for the tree component of the earlier section, the moveSubtreeToCursor method allows for moving an entire subtree rooted at a different node in the indexed tree to the node after the cursor. The query functions return boolean values relating to the properties of the tree and its nodes.

```java
public interface IndexedTree {
    //Cursor movement methods
    public void advance();
    public void retreat();
    public void enter();
    public void exit();
    public void moveBefore(restores Enum key);
    public void reset();

    //Data modification methods
    public void insert(restore Enum key, clears Object val);
    public void remove(restore Enum key);
    public void swapValue(Object label);
    public void moveSubTreeToCursor(alters Tree t);

    //Query methods
    public Boolean hasYoungerSibling(Enum key);
    public Boolean hasElderSibling(Enum key);
    public Boolean hasParent(Enum key);
    public Boolean isYoungerSibling(Enum node1, Enum node2);
    public Boolean isElderSibling(Enum node1, Enum node2);
    public Boolean isParent(Enum node1, Enum node2);
    public Boolean isEmpty();
}
```

Figure 3.2 The indexed tree interface
3.3 Mathematical Model

Figure 3.3 gives the mathematical model for the indexed tree component. It contains four model variables that specify how an indexed tree object is modeled [Che 05]. The first three variables are based on the mathematical model for ordered trees given in Cormen et al. [Cor 01], 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 variable nodes denotes the vertex set, and edges denotes the edge set. The edges are represented by ordered pairs. As the tree is modeled as an undirected graph, 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 eldest child of a parent is the first child when the children are ordered form left to right. The rightmost child is the youngest sibling. The children are ordered in ascending order from eldest to youngest. Thus, 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.

```java
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, CURSOR: nodes;
    constraints /* no cycles */
}
```

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

Figure 3.3 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 CURSOR 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 indexed tree constructor creates and indexed tree with the ROOT and the CURSOR as its child. Both root
and cursor have an order of 1 and map to null objects.

3.4 Cursor Movement Methods

A key feature of the indexed tree is the flexibility of cursor movement. Figure 3.4 specifies the cursor movement methods.

```java
public void advance();

requires hasYoungerSibling(CURSOR);
ensures nodes = #nodes and edges = #edges and
        order(x) = ( #order(x) - 1 ) if x = #next(CURSOR);
        #order(x) + 1 if x = CURSOR;
        #order(x) otherwise ) and
        contents = #contents;

public void retreat();

requires hasElderSibling(CURSOR);
ensures nodes = #nodes and edges = #edges and
        order(x) = ( #order(x) + 1 ) if x = #prev(CURSOR);
        #order(x) - 1 if x = CURSOR;
        #order(x) otherwise ) and
        contents = #contents;

public void enter();

requires hasYoungerSibling(CURSOR);
ensures nodes = #nodes and
        edges = #edges minus { (#parent(CURSOR), CURSOR) }
        union { (#next(CURSOR), CURSOR) } and
        order(x) = ( 1 if x = CURSOR;
        #order(x) - 1 if isYoungerSibling(x, CURSOR);
        #order(x) + 1 if isChild(x, #next(CURSOR));
        #order(x) otherwise
        ) and
        contents = #contents;

public void exit();

requires parent(CURSOR) != ROOT;
ensures nodes = #nodes and
```
edges = #edges minus { (#parent(CURSOR), CURSOR) } union { #parent(#parent(CURSOR)), CURSOR } and order(x) = ( 1 if x = CURSOR;  #order(x) - 1 if #isYoungerSibling(x, CURSOR);  #order(x) + 1 if #isParent(CURSOR) or #isParent(#parent(CURSOR)) and #isElderSibling(#parent(CURSOR))  ) and #order(x) otherwise ) and contents = #contents;

public void moveBefore(restores Enum key);

    requires key in nodes;
    ensures nodes = #nodes and
    edges = #edges minus { (#parent(CURSOR), CURSOR) } union { #parent(key), CURSOR } and
    order(CURSOR) = (  
        #order(#key) - 1 if #isYoungerSibling(key, CURSOR);
        #order(#key) otherwise
    ) and
    order(key) = (  
        #order(#key) if #isYoungerSibling(key, CURSOR);
        #order(#key) + 1 otherwise
    ) and
    order(x ≠ CURSOR, key) = (  
        #order(x) - 1 if #isYoungerSibling(x, CURSOR) and not #isYoungerSibling(#key, x);
        #order(x) + 1 if #isYoungerSibling(x, #key) and not #isYoungerSibling(CURSOR, x);
        #order(x) otherwise
    ) and
    contents = #contents;

public void reset();
    ensures nodes = #nodes and
    edges = #edges minus { (#parent(CURSOR), CURSOR) } union { (ROOT, CURSOR) } and
    order(x) = (  
        1 if x = CURSOR;
        #order(x) - 1 if #isYoungerSibling(x, CURSOR);
        #order(x) + 1 if #isChild(x, ROOT);
        #order(x) otherwise
    ) and
    contents = #contents;

    Figure 3.4 Cursor movement methods
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. The only change in the state is that the cursor and its immediate younger sibling, \#next(CURSOR), get their orders swapped. The retreat method retreats the cursor to the previous node on the same level. It requires that the cursor have an elder sibling to retreat beyond. The only change in the state is that the cursor and its immediate elder sibling, \#prev(CURSOR), get their orders swapped.

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 exit method makes the cursor the eldest sibling of its parent. It requires that the cursor have a parent and that is not the root node. The nodes and contents remain unchanged. The original edge involving the cursor is replaced by an edge from the cursor’s original grandparent to the cursor. The original younger siblings of the cursor get their orders decremented. The cursor jumps one level up to the previous 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 orders of the cursor’s younger siblings are decremented, and the orders of key’s younger siblings are incremented. However, if a node is a younger sibling of the cursor and key, its order remains unchanged. The contents map remains unchanged. The \texttt{restores} parameter mode for key indicates that the value of key remains unchanged, even though this is not explicitly stated in the ensures clause.

The reset method makes the cursor the eldest sibling of the root. The nodes and contents remain unchanged. The original edge involving the cursor is replaced by an edge from the root to the cursor. The original younger siblings of the cursor get their orders decremented. The cursor becomes the eldest child of the root, so its order is 1, and the other children of the root node get their orders incremented.

3.5 Data Modification Methods
The data modification methods insert or remove nodes from the tree or change the swap the values of the node labels. Figure 5 lists the data modification methods.

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 delete method deletes a node identified by KEY from the tree. The method requires that key be already in the indexed tree. key is removed from the node set, and the edge involving the key and its parent is deleted from the edge set. The order of nodes following key are decremented. The mapping involving key is also deleted from the contents map.

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.

The moveSubtreeToCursor method swaps the subtree under the cursor's next node with that rooted at the node associated by KEY. It requires that the cursor have a younger sibling and that the node identified by KEY is actually there on the tree. It ensures that the node set, and contents remain unchanged. The edge between the cursor's original younger sibling is replaced with that with the KEY node. And the cursor's original sibling now replaces the KEY node at the same position in the tree. So both KEY and the cursor's younger sibling swap orders.

```java
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(cursor) + 1 if x = #key;
            #order(x) + 1 if #isYoungerSibling(x, csr);
            #order(x) otherwise 
        ) and
        contents = #contents union { (#key, #val) };
```


public void delete(restores Enum key);
    requires key in nodes and hasChildren(key) = false and key != ROOT
    ensures nodes = #nodes minus { key } and
    edges = #edges minus { (#parent(key), key) } and
    order(x) = (
        order(x) - 1 if isYoungerSibling(x, key);
        order(x) otherwise
    ) and
    contents = #contents minus { (#parent(key), key) };

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

public void moveSubTreeToCursor(updates Enum key);
    requires hasYoungerSibling(cursor) and key in nodes;
    ensures nodes = #nodes and
    edges = #edges minus { (parent(key), key), (parent(cursor), x) }
    union
    { (parent(cursor), key), (parent(key), x) } if isYoungerSibling(x, cursor) and
    order(x) = (
        order(cursor) + 1 if x = key
        order(key) if isYoungerSibling(x, cursor)
        order(x) otherwise
    ) and
    contents = #contents

Figure 3.5 Data Modification methods.

3.6 Query methods

The query methods are listed in figure 3.6. Like in the tree discussed earlier, These neither change the structure
of the tree nor do they move the cursor. And return boolean values based on the tree properties.

The hasYoungerSibling returns true if there are nodes after the node identified by key and false otherwise. The
hasElderSibling method returns true if there are Nodes at the current level before key. The hasParent method
returns true if the node key has a parent. This method returns true for all nodes except the root. The isEmpty
method checks if the tree has only the root and cursor nodes when it is created and not any other nodes on it.
public Boolean hasYoungerSibling(Enum key);
ensures x > 0 where (x=child(#parent(key) and order(x) > order(#key));

global

public Boolean hasElderSibling(Enum key);
ensures x > 0 where (x=child(#parent(key) and order(x) < order(#key));

global

public Boolean hasParent(Enum key);
ensures #parent(key) ≠ null;

global

public Boolean isYoungerSibling(Enum node1, Enum node2);
requires #parent(node1) = #parent(node2)
ensures order(#node1) > order(#node2)

global

public Boolean isElderSibling(Enum node1, Enum node2);
requires #parent(node1) = #parent(node2)
ensures order(#node1) < order(#node2)

global

public Boolean isParent(Enum node1, Enum node2);
ensures #parent(node2) = #node1;

global

public Boolean isEmpty();
ensures nodes={ROOT,CURSOR} and edges={(ROOT,CURSOR)}

global

Figure 3.6 Query methods.

3.7 Application

Gamma et al [Gamma 01] describe a trivial graphics program in their discussion of the composite design pattern. The program can draw primitive shapes and then build drawings which are composites based on these primitive shapes. The indexed tree component discussed in this section is ideally suited to implementing such a program by using the indexed tree as a centralized point of control for the drawing application in place of the composite design pattern. When implemented with a alias-avoidance and value semantics based language such as Tako, the program would be formally specifiable and verifiable then its composite pattern implementation.

Another non-trivial application which is implemented in a later section is a text-based adventure game. Again, in a reference-based object-oriented language, one would be inclined to use the composite pattern for the design
of the game. However, a design utilizing the indexed tree data structure and value semantics would be much more easier to reason about and lends itself to easier formal verification. There is also the advantage of using the tree component's interface out-of-the box to easily construct the game tree. Reusing a tree component is easier than re-using the composite design pattern as there is good separation of the tree hierarchy design from the contents of the tree in the tree component as opposed to the composite pattern where there is little separation between design and content. So reusing the composite pattern as a black box is tougher to achieve then reusing the tree.
Chapter 4

The text-based adventure game – A case study in Tako.

4.1 Overview

This section introduces a text-based adventure game project. The adventure game is a non-trivial application which is loosely based on text-based adventure game systems such as TADS[Rob 06] and Inform[Nel 01]. The adventure game, if implemented in Java, would have been an ideal candidate for a Composite pattern based implementation. However, here it is implemented in the Tako [Vas 06] language. Tako is a Java-like language which facilitates value semantics by incorporating alias-avoidance [Min 96]. This implementation of the adventure game in Tako makes use of the Indexed Tree interface which has been discussed in the earlier section. We chose the adventure game as an example implementation, because it is sufficiently complex and non-trivial and preferred it over the Drawing application described in Gamma et.al [Gam 94] mainly because, implementing the latter would mean developing and using a drawing toolkit similar to the Java Swing [Wal 01] library. That would mean a lot of programming focused on the GUI design, which is really not the purpose of the program. The adventure game is console-based and has no GUI design involved and also, uses a design which makes use of the Composite design pattern.

The paradigm shift involved in using the Tako language was reflected in the way methods were implemented. 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. Statements which would have possibly used reference copying in Java were replaced by statements that used swapping. This was facilitated by the swapping operator :=: which was part of the Tako language and enabled swapping of entities on either side of it. Instead of obtaining references to objects inside containers like the tree, objects were swapped out of the containers, modified and then swapped back in. This was in conformance with the alias-avoidance model of the Tako language. The implementation was also different for methods. It is not uncommon for a Java method to have side-effects. This meant that in addition to returning a value, the method would implicitly modify an object for which it had a reference. Such side-effecting functions were implemented as procedures (methods devoid of return values) in Tako and the return value was passed out as a parameter. For methods which would not have side-effects, the return value was stored in a distinguished variable called result.

4.2 Adventure Game Architecture
Figure 4.1 portrays the architecture of the adventure game with the relevant game classes and their relationships. The lifecycle of the game starts with the Initializer class whose initialize method sets up the game world with all the necessary game objects on the game tree. The dictionary is also populated. Every entry in the dictionary is a set of game objects which have a string key. Thus, a baseball object and a beach ball object would be in the dictionary in a set keyed by the “ball” key. Under a scenario where, it is required to disambiguate which object was being referred by “ball”, the Resolver class comes into play. A Resolver component tries to determine what game object is intended when the player enters ambiguous text. The game accepts text inputs from the player, which are usually simple imperative sentences, such as “look at ball” or “put the beach ball in the box”. The command entered by the user is then processed by the Processor, which in turn, calls the Parser component. The Parser component parses the input based on a supplied grammar and dictionary of game objects. Every text command entered by the user is broken down by the Parser into a four-part Command object. “Bob, put the ball in the box” would have the action as put, the subject as Bob and object1 and object2 would be the ball and the box respectively. Suppose the subject name is not specified, then the player character is assumed to be the subject. Having got the Command object from parsing the text command, the Processor now calls the respond method passing the Command object as its argument. It also might call the respond method discussed in section [3b] on a GameObject that is affected by the command. The respond method prints an appropriate response string to the user. The GameWorld component tracks the state of the game. It is the GameWorld class which encapsulates a game tree. The tree is implemented using the Indexed Tree component discussed in the previous section. The nodes of the tree are identifiers that are mapped to game objects. When the player inputs a command, the application updates the GameWorld, which in turn, modifies the game tree accordingly and generates a text response pertaining to the action that the user has performed.
4.3 Game object specification

Since the nodes of the game tree are mapped to actual objects in the game – each of which is an instance of the GameObject class, it is imperative to look at the specification of the GameObject class in more detail to understand how the tree modeled the hierarchy of game objects. Figure 4.2 shows the specification of GameObject.

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

    public GameObject();
        ensures id = void and properties = { };
```
Figure 4.2 The GameObject specification.

Every game object has a unique ObjectID and a set of properties associated with it. The nature of properties associated to objects differ from object to object. An object identified by KITCHEN of type Room might include the property LIGHT so that players can see objects in the room. An object identified by BOB 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. For this purpose, one will have to add the particular property like the OPEN property to the set of properties of an object identified by BOX the setProperty method given in the specification.

Checking for whether a box is open involves calling the hasProperty method with the open property as parameter. This would return true if it were contained in the box’s set of properties of false otherwise.

The respond method is included in the specification to incorporate any object-specific response to a particular command. For example the command "look at ball" would simply return the description of the ball. However, a "look at Mary" command might need a response of "Mary sees you looking at her and goes away" This is a case of needing an object-specific response for a command. This is facilitated by having a respond method inside GameObject.

4.4 Game world specification

The GameWorld is implemented by the indexed tree. During game play, nodes of the tree represent game objects in the game world. The game world is updated by user commands which either change the configuration of the tree or the contents of nodes (game objects) of the tree. The specification of the Game World class is shown in Figure 3.
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 – { cursor } and
    conc.edges = gameTree.edges – {(parent(cursor), cursor) } and
    forall x in conc.nodes,
        conc.order(x) = (gameTree.order(x) – 1 if isYoungerSibling(x, cursor);
                        gameTree.order(x)) otherwise ) and
    conc.contents = gameTree.contents – { (cursor, gameTree.contents(cursor)) } }

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 */
} }
younger siblings of the cursor node in the indexed tree (all nodes which come after the cursor at the same level in the indexed tree) have their orders decremented in the game world. The contents are the same except that there would be no mapping involving the cursor.

4.5 The setObjectProperty method

The method setObjectProperty in the GameWorld class is used in the game to add properties to game objects. It requires that the objectID supplied as an argument actually represents an existing node on the tree. 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 new property is also added to the set of properties of that game object. In the following section, we discuss the implementation of the setObjectProperty method and try to verify it against its specification that is listed in Figure 3.

Let us look at the implementation for a call to setObjectProperty in detail. Suppose the method call is setObjectProperty (BOX, OPEN). The implementation first creates a new GameObject called rec. It then moves the cursor before the desired game object (BOX in this case) in the game tree using the moveBefore method of the tree. Now it swaps out the value of the existing game object which represents the box into rec with a call to gameTree.swapValue(rec). The new property (OPEN) is added to the swapped out game object using the addProperty method. Then the rec object is swapped back into the game tree with the new property added to it. So the BOX is now OPEN. Figure 4 graphically depicts the pre-state and post-state of a call to setObjectProperty(BOX,OPEN) method showing how the underlying tree is modified and also how the client (game world) sees it. Properties of an object are shown in curly braces next to the corresponding node featuring the object.
The implementer view shows the game tree which is implemented as an indexed tree using the model described in Section 3.3. The implementer view features the cursor as a separate node (labeled in red) along with other nodes in the game tree. The client view is essentially the game world which is obtained using the correspondence described in Figure 3 from the game tree. Hence, the cursor and the edge involving it are absent in the client view.

4.6 Trace of setObjectProperty (Implementer View)

The tracing table in Table 4.1 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 4.4. The game tree has five nodes: ROOT, CURSOR, KITCHEN, BOB, and BOX; and four edges, corresponding to those shown in Figure 4.4. In the tracing table, we represent game objects as sets of properties, omitting the redundant identifiers for brevity. Nodes in the game tree are listed inside the nodes set. The edges are listed as ordered pairs inside the edges set. The order mapping gives the appropriate sibling ranking of the nodes. The contents mapping maps nodes to game objects. The last line of each fact contains the values of the formal parameters and local variables (obj, prop and rec in this case). State 0 begins after the local variable rec is declared.

Table 4.1 Trace of setObjectProperty (implementer view)

<table>
<thead>
<tr>
<th>State</th>
<th>Facts</th>
</tr>
</thead>
</table>
| 0     | nodes = { ROOT, CURSOR, KITCHEN, BOB, BOX }  
edges = { (ROOT, CURSOR), (ROOT, KITCHEN), (KITCHEN, BOB), (KITCHEN, BOX) }  
order = { (ROOT, 1), (CURSOR, 1), (KITCHEN, 2), (BOB, 1), (BOX, 2) }  
contents = { (ROOT, {}), (CURSOR, {}), (KITCHEN, {LIGHT}), (BOB, {PERSON, MALE}), (BOX, {BIN}) }  
obj = BOX and prop = OPEN and rec = {} |
| 1     | nodes = { ROOT, CURSOR, KITCHEN, BOB, BOX }  
edges = { (ROOT, KITCHEN), (KITCHEN, BOB), (KITCHEN, CURSOR), (KITCHEN, BOX) }  
order = { (ROOT, 1), (KITCHEN, 2), (BOB, 1), (CURSOR, 2), (BOX, 3) }  
contents = { (ROOT, {}), (CURSOR, {}), (KITCHEN, {LIGHT}), (BOB, {PERSON, MALE}), (BOX, {BIN}) }  
obj = BOX and prop = OPEN and rec = {} |
gameTree.swapValue(rec);

2 | nodes = { ROOT, CURSOR, KITCHEN, BOB, BOX } 
   | edges = { (ROOT, KITCHEN), (KITCHEN, BOB), (KITCHEN, CURSOR), (KITCHEN, BOX) } 
   | order = { (ROOT, 1), (KITCHEN, 2), (BOB, 1), (CURSOR, 2), (BOX, 3) } 
   | contents = { (ROOT, {}), (CURSOR, {}), (KITCHEN, {LIGHT}), (BOB, {PERSON, MALE}), (BOX, { }) } 
   | obj = BOX and prop = OPEN and rec = {BIN} 

rec.addProperty(prop);

3 | nodes = { ROOT, CURSOR, KITCHEN, BOB, BOX } 
   | edges = { (ROOT, KITCHEN), (KITCHEN, BOB), (KITCHEN, CURSOR), (KITCHEN, BOX) } 
   | order = { (ROOT, 1), (KITCHEN, 2), (BOB, 1), (CURSOR, 2), (BOX, 3) } 
   | contents = { (ROOT, {}), (CURSOR, {}), (KITCHEN, {LIGHT}), (BOB, {PERSON, MALE}), (BOX, { }) } 
   | obj = BOX and prop = OPEN and rec = {BIN, OPEN} 

gameTree.swapValue(rec);

4 | nodes = { ROOT, CURSOR, KITCHEN, BOB, BOX } 
   | edges = { (ROOT, KITCHEN), (KITCHEN, BOB), (KITCHEN, CURSOR), (KITCHEN, BOX) } 
   | order = { (ROOT, 1), (KITCHEN, 2), (BOB, 1), (CURSOR, 2), (BOX, 3) } 
   | contents = { (ROOT, {}), (CURSOR, {}), (KITCHEN, {LIGHT}), (BOB, {PERSON, MALE}), (BOX, {BIN, OPEN}) } 
   | obj = BOX and prop = OPEN and rec = { }
Upon execution of the statement `gameTree.moveBefore(obj)`, the cursor now changes its position in the tree to be right before `BOX` to be its immediate younger sibling. To check if the moveBefore operation is permissible, we must look at the requires clause of the indexed tree's moveBefore method given in Figure 3.4. It requires that key (the formal parameter that corresponds to obj) be in the node set. The value of key just before the call to moveBefore is made is `BOX`, which is an existing element of the node set in state 0 just before the call. Thus, the precondition is satisfied. Now, we proceed to write out the facts of state 1. For this, we apply the ensures clause of the moveBefore method of the indexed tree obtained from Figure 3.4 to the facts in state 0.

From the ensures clause for the moveBefore method, we know that the only variables in the program state that change are edges and order. The edge from `ROOT` to `CURSOR` is replaced by an edge from `KITCHEN` to `CURSOR`. `BOX` is now a younger sibling of `CURSOR`, so the order of `CURSOR` becomes 2 (the old order of `BOX`), and `BOX`'s order is incremented to 3. `KITCHEN` was originally a younger sibling of `CURSOR`, so its order is decremented. The order of all other nodes is unchanged.

Upon examination of the ensures clause for the swapValue method, we find that the swapValue method swaps the values of `rec` and the game object at the node identified by `BOX` since the `CURSOR` is before that node. This means that the structure of the tree is unchanged. The nodes, edges and the order of nodes all remain unchanged. The only change here is in the contents mapping for the `BOX` identifier. That changes to the previous value of `rec` and `rec` now gets the game object previously held by the `BOX`.

The following call to `rec.addProperty(prop)` would add the `OPEN` property to `rec`. This does not affect the game tree at all in any way since the operation affects only `rec`. The final `gameTree.swapValue(rec)` swaps the value of `rec` back into the `gameTree`. Since the cursor position is still before `BOX`, the value held by `rec` is now swapped with the game object mapped by `BOX`. Thus, the `BOX` game object now has properties `BIN` and `OPEN`. This is how the call to `setObjectProperty(BOX,OPEN)` is implemented using the game tree.

4.7 Trace of setObjectProperty (Client View)

To complete our verification process as to whether the implementation of setObjectProperty is correct with respect to its specification, for this particular start state, we need to translate the first and last states from the implementer view in Table 1 to the client view using the correspondence clause, and then see if they conform to the specification of setObjectProperty. Table 4.2 is a tracing table for setObjectProperty after this translation. State 0 in Table 4.2 is derived from the initial state (state 0) in Table 1 and state 1 is derived from the final state (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 4.2 Trace of setObjectProperty (client view)

<table>
<thead>
<tr>
<th>State</th>
<th>Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>conc.nodes = { ROOT, KITCHEN, BOB, BOX }</code></td>
</tr>
<tr>
<td></td>
<td><code>conc.edges = { (ROOT, KITCHEN), (KITCHEN, BOB), (KITCHEN, BOX) }</code></td>
</tr>
<tr>
<td></td>
<td><code>conc.order = { (ROOT, 1), (KITCHEN, 2), (BOB, 1), (BOX, 2) }</code></td>
</tr>
<tr>
<td></td>
<td><code>conc.contents = { (ROOT, { })}, (KITCHEN, {LIGHT}), BOB, {PERSON, MALE}), (BOX, {BIN}) }</code></td>
</tr>
<tr>
<td>1</td>
<td><code>setObjectProperty(BOX, OPEN);</code></td>
</tr>
<tr>
<td></td>
<td><code>conc.nodes = { ROOT, KITCHEN, BOB, BOX }</code></td>
</tr>
<tr>
<td></td>
<td><code>conc.edges = { (ROOT, KITCHEN), (KITCHEN, BOB), (KITCHEN, BOX) }</code></td>
</tr>
<tr>
<td></td>
<td><code>conc.order = { (ROOT, 1), (KITCHEN, 2), (BOB, 1), (BOX, 2) }</code></td>
</tr>
<tr>
<td></td>
<td><code>conc.contents = { (ROOT, { }), (KITCHEN, {LIGHT}), BOB, {PERSON, MALE}), (BOX, {BIN, OPEN}) }</code></td>
</tr>
</tbody>
</table>

4.8 Discussion

We have seen how the adventure game was designed in Tako using the game world as a centralized point of control. Many object-oriented programming practices prefer distributed control to a centralized one [Fow 03]. A good way to implement a version with distributed control would be to implement a Composite pattern version of the adventure game in Java. That would form an interesting comparison with the Tako version. A tree-based implementation in Java would also make for an interesting implementation and comparison against a composite pattern version also implemented in Java, which is precisely what is studied in the next section. In a language which uses reference semantics like Java, references to the actual game objects can be stored in the indexed tree instead of enumeration types. Also, the need for swapping objects out, modifying them and putting them back in is also eliminated as one can modify the object using an alias [Hog 92]. The swapping out, modification and swapping back in sequence had to be carefully implemented in Tako. Care had to be taken not to modify the conceptual cursor position – which, if modified, led to errors in the tree operation. The errors were easily fixed, but represented an example of errors that would not occur in Java as the objects are never swapped in and out of the tree in the first place. References can easily be obtained to game objects and they can be modified via these handles. Although implementing composite pattern version in Tako is possible, 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. Formally reasoning about such programs with distributed control is also a
major challenge which is a definite area for further research.

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 were implemented from scratch using the pointer component. This further increased the coding effort required for the adventure game in Tako. The pointer component was only used in these low level data structure classes; while the higher design level classes did not require the use of the pointer component.

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.
Chapter 5

Java case study – comparison and metrics

5.1 Overview

This section presents a comparison of the text-based adventure game with a minimal set of classes implemented as a composite pattern version as well as one based on an indexed tree. For this purpose, the indexed tree was implemented in Java. The Architecture of the indexed tree version of the adventure game is the one shown in figure 4.1. The major packages in both the versions include:

- **adventure.action:** Contains the Action class and classes for all the actions (corresponding to the verbs like put, look etc),
- **adventure.command:** The Command class which is populated for every user command
- **adventure.scanners:** Contains the Parser and the Dictionary classes.
- **adventure.main:** Has a game initializer class (Initializer) and the game's entry point class (Game) containing the main method.

The composite pattern version and the indexed tree version have exactly the same classes in these packages. However, they differ in the following packages:

- **adventure.universe:** Contains the game objects. In the composite pattern version, game objects have a deep inheritance structure from primitives to composites with the GameWorld on top of the hierarchy and primitive objects like ball and box at the bottom. Composite objects like a kitchen or a drawing room are comprised of primitive objects
- **adventure.gameplay:** Containing the response class which forms the game response for user commands is also implemented differently in the two versions.

In the composite pattern version, the indexed tree version additionally has the adventure.util package containing the game tree which was the centralized point of control in the tree version. A description of the salient features of both the versions is presented followed by a metrics comparison and discussion highlighting the pros and cons of the two versions.

5.2 The Composite Pattern Version
The composite pattern version had a total of 21 classes in 7 packages. It took about 9 man hours to code this version and about 9 man hours to debug it. Since the control was on individual objects, bugs in every wrongly coded object had to be debugged and rectified which required significant effort in tracking down the object which had the bug and then correcting.

The Composite Pattern version features a primitive class called GameObject from which all other objects extend. Figure 5.1 shows this inheritance hierarchy for game objects in the composite pattern.

![GameObject Inheritance structure](image)

**Figure 5.1 The GameObject Inheritance structure**

The CompositeGameObject class extends GameObject and adds a list of children for a composite game object with methods to manage the children—who by themselves can be composites or primitive objects. Composites like a player character Alice and locations like the Drawing Room and Kitchen. The GameWorld is another composite object made out of rooms like drawing room and kitchen. Notice that unlike the tree version, the game world does not encapsulate the game tree. So the number of methods in the Game World is lesser than those in the indexed tree version.

Managing the children of a composite object is the responsibility of that composite. So there is no centralized point of control like a tree, but each composite objects in the hierarchy holds its child objects. Primitive game objects like the ball and box are at the bottom of the hierarchy (similar to the leaf nodes in the indexed tree).

Apart from the game world, other parts of the game like the action package, the command class, the parser and the dictionary are all unchanged and exactly like in the tree version. When the user types in a command like
“put ball in box”, the command is parsed by the parser using the dictionary and a command object with four variables – action, subject, object1 and object2 is formed. The command class is then fed into the response class which finds the actual object affected by the command– composite or primitive in the hierarchy and modifies it accordingly. There is no hierarchical traversal mechanism like the tree for getting to an object in the hierarchy. One has to get handles to the particular objects from references stored in the Dictionary. Suppose, the ball object is to be put inside a box, a reference to the ball object is obtained from the dictionary and then this reference was added to the list of children of the box object. Thus the ball was now “inside” the box. The response class then sends the appropriate response to the user.

5.3 The Indexed Tree Version

The indexed tree version had a total of 23 classes in 8 packages. It took about 12 man hours to code this version and about 5 man hours to debug it. The coding time was larger because it involved more effort to code the indexed tree. The debugging time was lesser than the composite pattern version because once the tree was set up, it provided a means of centralized control and objects were updated in accordance with user commands accordingly and fewer bugs were encountered. Any bug was easier to track down as the tree provided centralized control. Another point of note is that the tree nodes were references to actual game objects instead of being enum types like in the Tako version. This was because we could copy pointers in Java. Since the game objects were in the tree, a traversal mechanism was in place which could traverse down from the GameWorld to any object in the tree.

The tree version also featured the primitive class called GameObject from which all other objects extend. The game tree was composed of GameObject nodes. But there were no composite classes like in the composite pattern version. Nodes which contain other nodes simply had children in the tree. Primitive objects like a ball were leaf nodes and other composite nodes were higher up in the game tree. The GameWorld encapsulated the game tree and managed it. So the number of methods in the Game World is more than those in the composite pattern version.

Managing all objects in the game is the responsibility of the game tree. All changes to the game world structure were done through the game tree.

When the user types in a command like “put ball in box”, the command is parsed by the parser using the dictionary and a command object with four variables – action, subject, object1 and object2 is formed. The command class is then fed into the response class which calls GameWorld, which in turn, finds the affected nodes in the game tree and modifies them. The response class then sends the appropriate response to the user.
5.4 Metrics

Both the versions were run against a metrics evaluation program – ckjm [Chi 07] which consisted of the following metrics:

Weighted methods per class (WMC): A class's weighted methods per class metric measures the sum of the complexities of its methods. The ckjm program assigns a complexity value of 1 to each method, and therefore the value of the WMC is equal to the number of methods in the class.

Number of Children (NOC): A class's number of children (NOC) metric simply measures the number of immediate descendants of the class.

Response for a Class (RFC): The metric called the response for a class measures the number of different methods that can be executed when an object of that class receives a message (when a method is invoked for that object). Ckjm calculates a rough approximation to the response set by simply inspecting method calls within the class's method bodies.

Coupling between object classes (CBO): The coupling between object classes (CBO) metric is a measure of the number of classes coupled to a given class (effluent couplings, Ce). This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.

Depth of Inheritance Tree (DIT): The depth of inheritance tree metric provides for each class a measure of the inheritance levels from the object hierarchy top. In Java it is at least 1 (all classes inherit Object).

These set of metrics were run against two packages in both the versions – the adventure.scanners package which is the same for both the versions and the adventure.universe package which is different for both the versions.

Tables 5.1 and 5.2 shows the results of running the metrics against the adventure.scanners package for both the versions. The results for both the packages are the same. Both these packages had the exact same set of classes. So, they yielded the same results. The DIT parameter shows the depth of inheritance. It was high in these classes as they used a lot of classes like ArrayList form the java.util package which had a deep inheritance tree. The parser class also had a high CBO parameter as it uses other classes and the dictionary to parse the string that the user inputs on the command line. The RFC parameter is high for Parser indicative of the fact that the number of executable methods is higher and the class needs a lot of testing. Both had no children, so the NOC parameter is zero.
Table 5.1 Metrics for the adventure.scanners package in the Composite Pattern version

<table>
<thead>
<tr>
<th>Class</th>
<th>WMC</th>
<th>NOC</th>
<th>RFC</th>
<th>CBO</th>
<th>DIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>adventure.scanners.Parser</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>29</td>
</tr>
<tr>
<td>adventure.scanners.Dictionary</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.2 Metrics for the adventure.scanners package in the Indexed Tree version

<table>
<thead>
<tr>
<th>Class</th>
<th>WMC</th>
<th>NOC</th>
<th>RFC</th>
<th>CBO</th>
<th>DIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>adventure.scanners.Parser</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>29</td>
</tr>
<tr>
<td>adventure.scanners.Dictionary</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>13</td>
</tr>
</tbody>
</table>

Tables 5.3 and 5.4 shows the results of running the metrics against the adventure.universe package for both the versions. Notice that the composite pattern version has the CompositeGameObject class and the composite classes are its descendants. The WMC parameter is roughly the same for both the versions, however, the GameWorld class in the Indexed Tree version has a lot of methods for controlling the tree. So its WMC is quite high. The CompositeGameObject class has a lot of methods specific to composite objects giving it a high WMC too. The NOC parameter in the tree version has a high value for GameObject class as all other GameObjects are its immediate children. In the composite version, NOC is only 2 for GameObject as it has only the ball and the CompositeGameObject class as its immediate children. The RFC parameter is high for GameObject and GameWorld classes in the indexed tree version (As they are method-heavy), but RFC is distributed between GameObject, CompositeGameObject and its descendant Container in the composite pattern version. CBO values for the composite pattern version indicate a fair degree of coupling between all the classes in the universe. The high CBO for GameObject is because it uses classes form the Java standard library. The figures are the same for the tree version too. The DIT parameter is quite high for GameObject in both cases as the standard library classes are utilized there too. However, the CompositeGameObject class also has a high DIT value as it creates the list of children of a node and uses the Java library implementations of the List to do so.

Table 5.3 Metrics for the adventure.universe package in the Composite Pattern version

<table>
<thead>
<tr>
<th>Class</th>
<th>WMC</th>
<th>NOC</th>
<th>RFC</th>
<th>CBO</th>
<th>DIT</th>
</tr>
</thead>
</table>

Table 5.4 Metrics for the adventure.universe package in the Indexed Tree version

<table>
<thead>
<tr>
<th>Class</th>
<th>WMC</th>
<th>NOC</th>
<th>RFC</th>
<th>CBO</th>
<th>DIT</th>
</tr>
</thead>
</table>
### Table 5.4 Metrics for the adventure.universe package in the Indexed Tree version

<table>
<thead>
<tr>
<th>Class</th>
<th>WMC</th>
<th>NOC</th>
<th>RFC</th>
<th>CBO</th>
<th>DIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>adventure.universe.GameObject</td>
<td>17</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>adventure.universe.Box</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>adventure.universe.Ball</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>adventure.universe.Alice</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>adventure.universe.DrawingRoom</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>adventure.universe.GameWorld</td>
<td>17</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>18</td>
</tr>
</tbody>
</table>

#### 5.5 Comparison and Discussion

The comparison of both versions clearly shows that in the indexed tree version, there is a centralized control with the game world encapsulating the game tree and controlling the game from a single point. In the composite pattern version the control is distributed and the response class delegates control to the appropriate class in the composite pattern. The tree implementation also shows that the tree component can be easily re-used with its interface for any design demanding a hierarchy of objects. It is easier to code the composite pattern but is relatively tougher to re-use it compared to the indexed tree data structure as its contents and structure are not separated from each other. The whole universe is an inheritance hierarchy and selecting out
classes for reuse form that is not a trivial task. However, the composite pattern has distributed program control spread over many classes compared to the tree version which is heavily dependent on the GameWorld class for modifying the game tree. Development of the tree version took longer time too as the tree had to perfectly implemented and tested thoroughly. However, runtime bug tracking was easier in the tree version compared to the composite pattern version.
In this thesis we have looked at the composite pattern and found the inherent problems in using it with regard to formally reasoning about it. The approach to eliminate that is to use alternative ways of designing using the two tree data structures that are studied in great detail in this thesis. We also introduced the indexed tree component and saw how it can be used to design a verifiable non-trivial application – a text-based adventure game in Tako – a Java-like language based on value semantics. We also looked at how to verify a method in the adventure game by using formal verification techniques like the tracing table. We also looked at how we can apply the indexed tree interface to design the adventure game in Java. A comparison of the two designs was also presented and it was seen that the indexed tree model – although takes longer to implement, could be reused easily as it had a clean separation between design and content. This was seen from the metrics analysis performed on both the designs.

Looking at the indexed tree implementation and the composite pattern implementation of the adventure game in Java, it is clear that both the versions have their own pros and cons. The advantage of using the indexed tree is the specifiability, ease of reasoning, re-usability and clean separation of content and design. On the other hand, using the indexed tree implementation makes the game design centralized on the indexed tree and the game world which encapsulates the tree. All of the major game logic revolves around modifying the game tree and there is no distribution of control. As such, centralized control is not a desirable property to have in software design. The composite pattern version has distributed control. But it is difficult to reason about and the design pattern itself cannot facilitate code re-use.

We have already successfully implemented the adventure game using the indexed tree in Tako. This was presented as a case study in the Specification and Verification of Component-Based Systems (SAVCBS) 2008 conference [Sav 08]. The indexed tree specification in this paper was in the Tako language. Going forward, we propose to present tree alternatives to the composite pattern, with a lightweight, Java-based specification of the indexed tree at the Software Practice and Experience (SP&E) event. As the Tako project moves forward, we plan to implement the bootstrap version of the Tako Compiler. The abstract syntax tree will be implemented based on the indexed tree specification in this thesis.

It must be remembered though, that the tree components presented in this thesis are alternatives, not replacements to the Composite pattern. As we have seen, there are definite advantages and disadvantages to using either the composite pattern or the indexed tree in applicable cases. One has to decide on the choice to be
made based on the type of application needed, the development time at hand and the degree of re-use desired for the application.
Bibliography


[Ogd 00] Ogden, W.F. *The Proper Conceptualization of Data Structures*. The Ohio State University, Columbus, OH, 2000.


