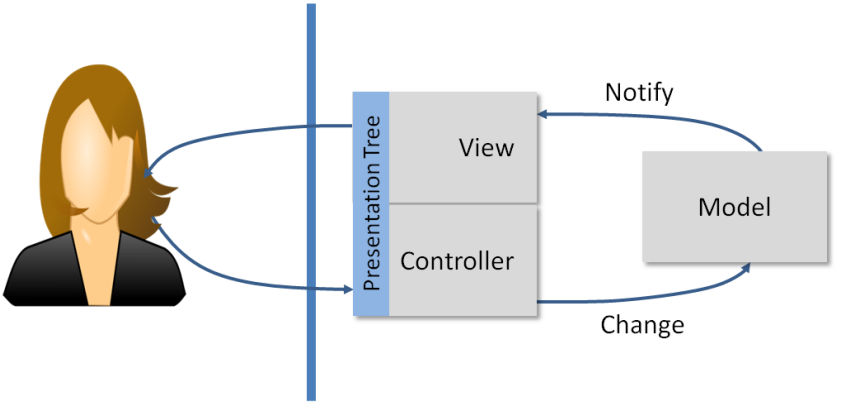
Data Trees

Figure 2-1 shows a modified form of the Model-View-Controller architecture. In this approach the view and controller are merged and both of them work in terms of a presentation tree. The presentation tree represents the drawing on the screen. Input events are interpreted by the controller based on what drawing primitives are being presented by the view.



2-1 – Model View Controller

The model also is frequently represented as a tree. In such an architecture it is the responsibility of the view to transform the model tree into a presentation tree. It is the responsibility of the controller to transform selections on the presentation tree into references of objects in the model tree that must be changed. If we add a persistent data store such as a file system or web service, their communications are also represented as trees of data objects. Many styling tools that add color, shadows and shape to basic presentations are defined in terms of tree transformations. Trees and their transformations occur everywhere in interactive architectures.

In current web browsers, the HTML is parsed into a tree called the DOM (Domain Object Model). JavaScript then provides access to this tree. As JavaScript code makes modifications to the DOM, the browser will automatically change what is displayed. The DOM tree is fundamental to how the modern browser UI functions. In addition many web page models communicate with their host using either XML (eXtensible Markup Language) or JSON (JavaScript Object Notation). Both of these formats are trees. Presentation trees have long been part of the architecture of 3D rendering systems.

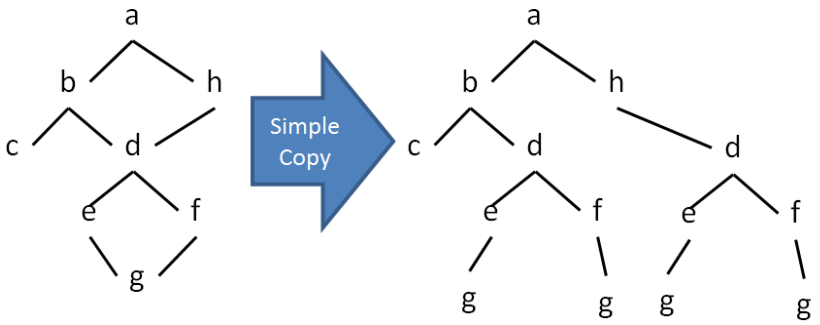
Why trees?

Trees as a data structure have a number of properties that make them very easy to use. It is simple to copy a tree with a recursive algorithm that only contains a few lines of code (Figure 2-2).

```
Tree copy(Tree t)
{   Tree nt = new Tree();
    foreach (c childof t)
    {   nt.add(c); }
    return nt;
}
```

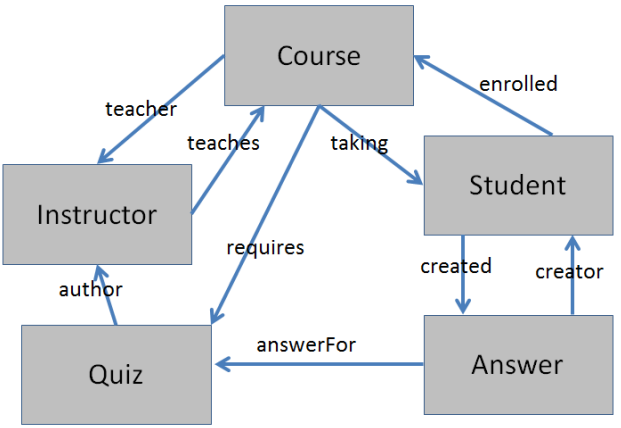
2-2 – Simple Tree Copy

Using data structures other than trees can cause problems for such simple algorithms. A directed acyclic graph (DAG) expands with many copies as shown in Figure 2-3. If the structure has cycles then this simple algorithm will generate an infinite tree. It is not that we want to make a lot of copies. It is just that for trees, many algorithms that we need are as simple as the copy algorithm and get rather harder with more complex structures. Algorithms for comparing, matching and conversion to a file all fall into this category.



2-3 – Simple copy of a DAG

In the real world relationships among data objects are more complex than trees. Take for example a system where instructors author quizzes and students take quizzes providing answers. We need to link all these together to create a structure that represents our problem. These relationships are shown in Figure 2-4.



2-4 – Quizzing data structure

This is not a very complicated model but everything relates to many other things, and there are many cycles in this relationship. Omitted from Figure 2-4 is all of the information about these kinds of objects should as names, semesters taught, year in school, etc. In representing

the real world, trees are inadequate. A simple model that reduces this to trees is called a key/value store. Every course, instructor, student, quiz and answer is given a unique identifier (key). The keys are generally text because that is easiest to send over the network and to debug. Keys are easy to generate. A single part of the program has a function to generate them. A very robust technique is to generate a random 10-20 digit number and form a key such as "K0265372273". We then check our key/value store to see if it is already used. If it is, then we generate another one. If a Student object wants to reference a Course object, it stores the object's key rather than a pointer. In the chapter on persistent data we will discuss this more in detail. We can treat a key/value store as another object for which the keys are attribute names. For our purposes, every object is now a tree and has an identifier or key. This this we can build any structure we want while retaining the tree as our basic data structure. Note that the keys need only be unique within our application or within a single document.

This key/value store with tree objects is the representation we will use throughout the rest of the book. The trees are easy to convert strings and ship over the network as our application integrates with the rest of the world. The big advantage for us is that all of our algorithms become as simple as our tree copy.

Spark

Throughout the book we will discuss the development and enhancement of an interactive toolkit called Spark. The name is purely arbitrary. We will use Spark because it is designed to allow us to look under the hood and show how many of the modern interactive tools and applications are put together. Spark is written in Java and is available as a JAR file at XXXXX. The source to Spark is also available at XXXXX. As we work through the concepts and show how to implement them in Spark we will then show how these same concepts are found in HTML/CSS, JQuery, Swing, C#, Microsoft's WPF, Adobe Flash and many others. A simple system that we can dissect and study will illustrate the

concepts in these others without the complexity of industrial-strength tools.

SON (Spark Object Notation)

With so many trees floating around our discussions we need a simple notation that we can use. There are many notations and many more will be generated before this book needs to be rewritten. XML was developed from SGML (Standard Generalized Markup Language) and both are closely related to HTML (HyperText Markup Language). As markup languages they became highly influenced by projects that wanted correct and validated markup. As such the notation became increasingly verbose.

As JavaScript grew in power within web browsers, its object notation JSON (JavaScript Object Notation) grew in favor. It is a very simple direct notation. JSON is very simple. It consists of numbers, strings, true, false, objects and arrays. Objects are simple name value pairs such as:

```
{ size:20, name:"George", distance: 2.0,
   address:{ street:"Fair View Dr.",
      number:1074
   }
}
```

Arrays are structured similarly with square brackets instead of curly braces, such as:

```
[ 1, true, { begin:2, end:4 }, "Ralph" ]
```

Combinations of objects and arrays can produce any tree structure that we might need. It is a very simple notation and does almost everything that we need. These simple object/array structures are found in many modern programming languages such as Python and Flash. Historically they have not been used in major programming languages because though they are very flexible they has some inefficiencies. However, these inefficiencies have been resolved by Moore's law and advanced

complication techniques that make them more than adequate for our needs.

However, for many of our discussions we will need objects with classes so that we can interface with lower-level tools and we can talk about what kind of object this actually is. SON extends JSON with the concept of a class name. Objects can optionally have a class, such as:

```
Line{ x1:7, y1:22, x2:7, y2:30, style:thick}
```

Independent of the efficiencies of strongly typed objects, it just helps to know that the object above is a Line and it does what Lines do rather than the more generic object representation. SON is basically JSON with the added ability to give objects a class.

SON to Java

As we work through our code discussions, the language will be Java. There are many other languages that would work just as well, but I picked Java. The goal is to illustrate the concepts rather than endorse a language. I do strongly recommend that you use a language with good garbage collection facilities. Life is just too short to muddy up your software trying to deallocate storage and prevent memory leaks. This rules out C and C++. Just walk away and use a language that will save much time a grief. Today there are many garbage collected languages to choose from. This will also help prevent a number of hacking exploits if your application is going to work over the network.

Spark objects are built around three classes: SV, SO and SA. SV is a Spark Value that can contain any long, double, boolean, string, object or array. It is our basic container object. SO is a Java interface for accessing an object using key value pairs. The generic implementation of SO is the class SObj. An SObj essentially stores key/value pairs of anything. SA is the interface for arrays for which the generic implementation is SArray. So as not to impede our discussion of interactive architecture, the methods and constructors for SV, SO, and SA are found in appendix A.

Their usage is very straightforward. As they are used in the book you will quickly get the drift. Appendix A can resolve any confusions.

The SV class has several methods that greatly simplify moving back and forth between SON and Java.

```
SV.son(string ) – parses a String as SON and returns and
SV of what was parsed
sv.outSON() – will write the contents of sv to
System.out
SV.url(url ) – accepts a file: or http: reference to a
.son file and returns the parsed value as an SV
```

These methods make it very easy to convert Java objects to and from SON representations. This will allow us to easily look at the data we are manipulating and also provide us a simple way to read and write data from files and web services.

The methods above require class names to be fully qualified and will allow any Spark object class to be parsed. They can be a security hole and the fully qualified names are tedious. There are three additional methods that take an array of Java package names as a parameter. These will only parse objects from the listed packages and will use the class's simple name rather than the fully qualified name.

```
SV.son(packages, string ) – parses a String as SON and
returns and SV of what was parsed
sv.outSon(packages ) – will write the contents of sv to
System.out
SV.url(packages, url ) – accepts a file: or http:
reference to a .son file and returns the parsed value as
an SV
```

Programming Language Reflection

Reflection is a programming language concept where information about the language's classes, fields and methods are available at run-time. Sometimes this is called introspection. For example, we can get a Class object that describes the class of any data object. This Class object can be used to find the name of the class and a variety of other information. The kinds of information will vary from language to language but for our

purposes we want the name of the class, any public fields and any public methods. We also need the ability to use reflection to access and set fields as well as to invoke methods. The reflection capability allows us to access information about objects more dynamically. Access through reflection is slower, but opens up a variety of opportunities. Java, C#, Swift and other modern programming languages have reflection capabilities.

In Spark we have the class SOReflect. This class uses Java reflection to make normal Java objects with their methods and fields function as SO objects. Java reflection allows one to access information about a class's field and methods at runtime and access their contents.

Suppose we wanted to create our Line class as a Java class so that we could add code that would allow it to paint itself on the screen. The declaration would be:

```
public class Line extends SOReflect
{   public double x1;
    public double y1;
    public double x2;
    public double y2;
    public SO getStyle() { . . . . }
    public void setStyle(SO style) { . . . }
}
```

The SOReflect class has all of the SO methods. However, they are implemented so as to use Java reflection to discover and expose fields and methods from any of SOReflect's subclasses. In our Line example the fields x1,y1,x2 and y2 are exposed as if they were Spark attributes. The "style" attribute is also exposed when SOReflect discovers the "getStyle()" and "setStyle()" pair of methods. Using a method pair to define an attribute allows us to do more computation than simply setting and getting a field. We will discuss the need for this later when we look at change notification techniques. Some languages such as C# and Swift formalize this get/set method combination. In C# the style field would be encoded as follows:

```
private string myStyle;
public string style
    {  get { . . . . }
       set { . . . .  }
    }
```

Instead of discovering a pair of methods with special names as in Java the language specifically supports creating such methods and then allows style to be used like any other field, with the compiler taking care of the difference. We will use the reflection capabilities of SOReflect throughout the book to create various kinds of objects that conform to the Spark data model and yet have specific implementations that we need.

Referencing Objects

Because our objects are all trees, we can reference any part of an object using a path. This is an identical concept to a full file name in a folder or directory-based file system. A path is just a series of attribute or index selections starting from the root of the tree. Consider the following object.

```
Person{   name:"Phred", age:35,
    address:{
       number: 1032, street: "Winding way",
       city:"Boonieville", state: "New Jersey",
       zip: 12345
    }
    phones:[
       { kind:"mobile", number:"822-123-4567"},
       { kind:"home", number:"833-987-6543" }
    ]
}
```

In this object we can access the home phone number using an array of selectors, such as:

```
["phones", 1, "number"]
```

It is just a sequence of selections as we work down the tree. If the person was stored as an SV object the path in code would be:

```
person.get("phones").get(1).get("number");
```

Frequently we do not have a path, what we have is some object known to be part of a tree. In a tree every object except the root has a parent. This leads to a simple algorithm to retrieve a path for some object.

```
SA getPath(SV o)
{   if (o.myParent()==null)
    {   return new SArray();} // empty array
    else
    {   SA rslt = getPath(o.myParent());
        SV child = whichChild(o.myParent(),o);
        rslt.set(rslt.size(),child);
        return rslt;
    }
}
SV whichChild(SV parent,SV o)
{   if (parent.isSO())
    {   SO po = o.getSO();
        for(String att:po.attributes())
        {   if (po.get(att).equals(o))
                return new SV(att);
        }
    }
    else if (parent.isSA())
    {   SA pa = o.getSA();
        for (int i=0;i<pa.size();i++)
        {   if (pa.get(i).equals(o))
                return new SV(i);
        }
    }
    return null;
}
```

2-5 – Deriving a path for an object in a tree

Representing Object Change

There are essentially four things that one can do to an object. These are represented by the acronym CRUD (Create, Read, Update, and Delete). Creating an object does not generate a need for change notification until that object is placed into some data structure, such as a key/value store or some part of another object. Reading does not cause any change notification, so we will not worry about that here. There are three ways to update an object: setting an index in an array, inserting a new element into an array or setting an attribute in an object. We can

also delete array elements by index and delete attribute values in an object.

We can easily represent such actions in three data objects as shown in figure 2-6.

```
Set{ path:[ "address", "zip"], to:65432 }
Delete{ path: [ "phones", 0 ] }
Insert{ path: ["phones",0],
   as: { kind:"mobile", number:"829-123-4567"}
   }
```

2-6 – Data change representations

It is frequently the case that multiple changes have occurred. This is handled by using an array of change objects with the changes performed in sequence. With our path notation and the three special objects Set, Delete and Insert we can represent any series of changes to any of our data trees.

There are also times when we want to delete and insert multiple value. This particularly happens when working with arrays. The object modifications in figure 2-7 can handle those issues.

```
Delete{ path: ["phones"], from:0, to:1 }
Insert{ path: ["phones", 0], all:
   [   { kind:"home", number:"899-123-4567"},
       { kind:"mobile", number:"824-102-9384"}
   ]
}
```

2-7 – Deleting and inserting multiple values

Though we have represented change as data objects (which will be useful) we might also have represented change as set(), delete() and insert() methods defined on objects and arrays. Representation as methods simplifies their use in code. In some cases such methods will generate the change objects rather than performing the change directly.

Being able to represent data change in a general way is useful for notifying other parts of our architecture about changes that were made, representing changes in a change management system. If we add

attributes with the identity of who made the change, we can provide support for collaborative work, and we can send changes records over a network to tell a server how to update a database. If we add information about the previous value before the change, we can use these to store data for an undo/redo facility.

Summary

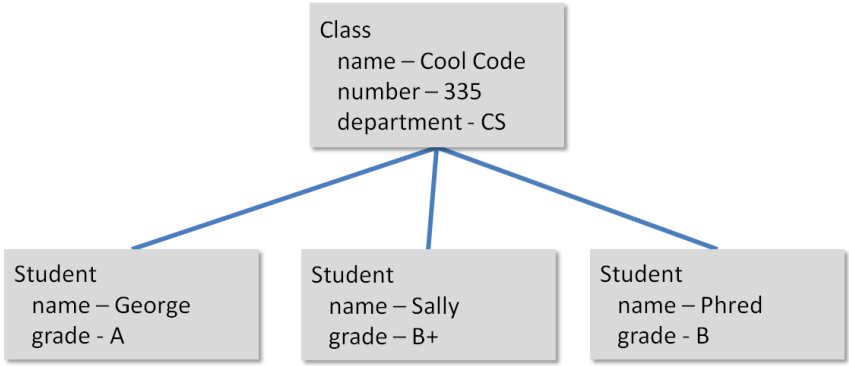
Data manipulation is fundamental to the model-view-controller architecture. We must transform model trees into presentations as well as selections of portion of the presentation into changes in the model. Though models for real applications require interrelationships among objects, we can transform such interconnected structures into trees by assigning key names to objects and storing the key names rather than pointers. Conversion of arbitrary data structures into trees simplifies a variety of data manipulation algorithms that we will need.

We also introduced Spark and its object notation SON. This is modeled after JSON with the addition of class identification on objects. The class identification will greatly clarify our discussions later. Spark also provides a simple object/array model for representing trees , which is patterned after Java, Python and others. Spark also provides the SOReflect class that makes it easy to create specialized object classes that integrate with the general data model.

Lastly we created a path notation that allows us to reference any object in a tree as well as a change notation that allows us to represent modifications to a tree. The change notation will be useful through our software architecture discussion.

Problems

1. Convert the tree shown in figure 2-8 into SON.



2. Given the following KeyStore, draw a diagram that represents the data in the store.

```
KeyStore{
    K123: Server:{ name:"Data", size:1024,
           connectsTo:[ "K99","K42" ]
        }
    K99:  Server:{ name:"Web", size:2048,
           connectsTo:["K123"]
        }
    K42:   Server:{ name:"DNS", size:64,
           connectsTo:["K123"]
        }
}
```

3. Write a path representation that will reference the size of the Web server in problem 2.
4. Write a data change description that will convert the tree in problem 2 into the tree below.

```
KeyStore{
    K123: Server:{ name:"Data", size:1024,
           connectsTo:[ "K99","K42" ]
        }
    K99:  Server:{ name:"Web", size:2048,
           connectsTo:["K123"]
        }
    K42:   Server:{ name:"DNS", size:64,
           connectsTo:["K123", "K17"]
        }
    K17:   Server:{ name:"Identity", size:128,
           connectsTo: [ "K42" ]
        }
```

```
}
```