

UC Berkeley Computer Science

Subject: A Program to Define a FLEET
Date: February 22, 2006
From: Ivan Sutherland
UCIES #2006-is20

Reference:

UCIES# 2005-is14: FLEET – A One-Instruction Computer, Ivan Sutherland, 5 December 2005

INTRODUCTION

I have been working on a program to put out a definition of a FLEET. It has two output formats: the XML format used as input to Igor's simulator, and the format required by Greg for input to RAMP. The text-form output describes a FLEET in terms of individual components and the connections between them. The Java code will define the form of the FLEET in question. I have chosen to call the program "fleetmaker".

It seemed easy to write fleetmaker. There's not much to a FLEET. There are only a few Ships selected from a finite set of possible types. There are horn and funnel structures to deliver instructions to the Ships and to form the switch fabric. If we think of banks of connections, there are only a few: Ship outputs connect to the switch fabric; switch fabric outputs connect to the Ship inputs; instruction horn output connects to Ships; and the fetch unit connects to the instruction horn and the trunk. A simple program should suffice to define all these connections.

Like many things, however, making all this work has turned out to reveal flaws in my initial thinking. As I wrote the program I kept being tempted to include a pointer structure model of the FLEET. For example, having output a horn or funnel structure, the program must connect other things to it. How much must the program remember about such major sub-structures to connect them together? I was tempted to model my fleetmaker after the simulator's topology, a bad plan that introduces more complexity than necessary.

So I decided to write this memo. I hope it will serve two purposes. First, it will outline the code that works. Second, and more important, I hope it will help me understand what's important to do. The focus of the program must be on making an output text file. The program should avoid unnecessary internal topology.

HORNS AND FUNNELS

I wrote the code that prints out the horns and funnels in a package called "partmaker." A factory class called TreeFactory has recursive code for a depth-first walk through a binary tree. It uses the "visitor" pattern, with a visitor class, called NodeDoer that does the output. The TreeFactory provides a single tree-walking system for both horns and funnels.

Specialized sub-classes of NodeDoer make types of horns and funnels, producing the proper output for each type. A NodeDoer has three methods, one for the root of the tree, one for the middle part, and one for the leaf nodes. The TreeFactory walks

This document is a product of a collaboration between Sun Microsystems and the University of California at Berkeley.. The ideas contained herein are freely available for any academic purpose.

the tree depth-first, calling the appropriate methods in NodeDoer to produce output. TreeFactory provides unique names for the component parts and internal connections in the tree.

HornMaker has a private sub-class of NodeDoer called BranchDoer that uses Igor's addressable branch components to make a horn. It will make two horns: the instruction horn and the destination horn. FunnelMaker has a private sub-class of NodeDoer called MergeDoer that uses Igor's arbitrated merge units to make a funnel.

The code is recursive and simple. I got the code to work with little difficulty. It makes lots of output for even modest tree depths. A very satisfying Java experience.

THE SHIPS

It seems easy enough to assemble a list of Ships. Code for this can have a single call for each Ship that associates an arbitrary name for each Ship with its type. Several different Ships can be of the same type, but have different names. The list of Ships thus specified will have a set of input and output terminals defined as the sum of the inputs and outputs of each Ship as specified in the definition of that Ship type.

The hard part of this code is getting the specifications of the Ship types. Igor provides a text file in XML format that lists the available types of Ships, giving each a type name. With access to that library, fleetmaker need only associate a set of Ship names with the appropriate type name from the library. A set of text pairs should suffice to define a particular fleet!

Ah, but there's the COW. You all know what BULL is – a lot of words with little content. Igor's XML file defining the library is COW – a lot of content with few words. The fleetmaker needs the definitions of each type of available Ship including its type name and the names of its inputs and outputs.

Igor provided me with code that reads his XML "library" file to produce a "ClassRegistry" that describes the kit of available parts. With some effort, I got this code to work, and it provides me with a "ClassRegistry" of Ship types – the required COW.

I asked Igor how he knows that his XML library file corresponds exactly to the real Java definitions of his Ships. The raw truth is that the library file is a hand-written transcription of the Ship types. Maybe there's a way to check its correctness, but it's not automatically generated. The problem, Igor explains to me, is that his program has no way to enumerate all of the Java classes that define Ship types. Some types might enter service as JAR files only after compilation and Java is not sufficiently class-aware to do an enumeration. Never mind, over time the library XML file will come to represent truth.

WHAT TO REMEMBER

Let us assume success. Assume code to define a list of Ships and to define horns and funnels. What must the code that assembles these collections provide to the code that connects these collections together? How much of the internal topology of the collections must fleetmaker preserve for connecting them?

Maybe the program need preserve nothing. For example, having output information about a bunch of Ships, I could write Java code that connects their inputs individually to the destination horn's outputs. I could examine the list of Ships to get terminal names and examine the destination horn to get terminal names and then, for each such pair,

call a method that outputs a connection between them. To do so would require me to write at least one line of code for each and every individual connection.

I prefer to assemble groups of connections. With this in mind I wrote a class called "TerminalMap" to collect Terminals into groups. The TreeFactory code builds a TerminalMap for each horn and funnel. The code defining the set of Ships could also build a TerminalMap. All that need be preserved from building each sub-structure is a record of how to connect to it. The TerminalMap is almost that.

The TerminalMap fails in its task because it confounds inputs and outputs. The TerminalMap is adequate as the interface to a horn or funnel, because we treat all of the inputs or all of the outputs of these structures together. It is inadequate for the trunk, however, because the trunk gets input from the funnel, separate input for literals, and an output to the destination horn. TerminalMap also fails for the column of couplers that associate Ship outputs with instructions from the instruction horn and feeds the source funnel. This collection of couplers has two broad input interfaces and one broad output interface.

TERMINALS vs PINS

I also got confused about the role of a "Terminal." In Igor's simulator a Terminal is a topological class that provides a lot of capability. In fleetmaker, however, a Terminal can be much simpler. It need only record enough information so fleetmaker can output an attachment to it. That's so much simpler that I now think the concept deserves a different name, Pin. The new name will avoid confusion with Igor's Terminal.

A Pin will define the text required for Igor's simulator, or Greg's assembler, to identify where to attach a connection. That text consists of two parts, the name of some component of an assembly, and the name of a terminal of that part. Because fleetmaker will already have output the parts of the assembly, there's no need to remember anything about an assembly except the pins where one can connect to it. A Pin need not reference anything – it's just a text string with two parts: the name of a component previously output by fleetmaker, and the name of a terminal on that component.

Is the distinction of "input" and "output" Pins important? It might be important if we believe that a connection can be only between an "input" pin and an "output" Pin. It is much less important if we associate the idea of "input" and "output" with groups of Pins as suggested in the next section.

CABLE CONNECTORS

I'd like to have groups of Pins. TerminalMap confounds input and output Pins. Instead let's think of "Plugs" and "Sockets," each of which is an ordered list of closely related Pins. The number of Pins will be the "width" of the Plug or Socket. Let us associate a Plug with inputs and a Socket with output, following the standard 120 volt power convention.

Ordering the Pins in Plugs and a Sockets is important to simplify mass connections between them. Ordering substitutes indexing for the need to specify the names of individual Pins. A mass connection is a group of many individual connections produced in order on a Pin by Pin basis. Connecting a Plug and a Socket will produce a bunch of text defining the many connections between individual Pins.

What happens if we attach a Plug and a Socket that differ in width? The code must apply some rule and issue a warning. One such rule might leave the unused pins unconnected. A better rule might connect the unused pins to a suitable null element.

THE ADDRESS TABLES

The list of Ships will have a Plug for the Ship's inputs and a Socket for the Ship's outputs. Similarly the destination horn has a Socket for the data outputs from the switch fabric. When we connect the destination horn to the Ships, the order of the Pins in the Plug and Socket establish the destination addresses of the various Ships. The program can use that association to produce the destination address table.

One might think that the sequence of pins in the source funnel likewise determines the source addresses of the Ships. Not so, because the source funnel serves its inputs on demand. Instead, it is the sequence of outputs of the instruction horn that establishes the source addresses. The program can use the association of the instruction horn's output with the Ship's outputs to produce the source address table.

CONCLUSION

This memo has served my second purpose admirably. I now feel, after having written all this down, that I know how to proceed. I hope the reader may also have learned by reading this. I hope you have noticed not only what I say here, but also how writing this memo has, itself, guided my thinking.

I remain amazed at how much difference recording ideas makes. The record forces better thinking because recorded ideas are crisp and firm, whereas unrecorded ideas are all too often fuzzy, shapeless and vague.

Thank you for listening.