Geoffrey Colman

CMSI 402

20 February 2019

**HW #2**

**5.1 What's the difference between a component-based architecture and a service-oriented architecture?**

*Component-based* architectures are considered to be sets of loosely-coupled sections of code (perhaps in separate source files, but all compiled to a single executable) that each perform highly specific roles and perform their services for each other by communicating through specified interfaces. Like service-oriented architectures, they allow large portions of the codebase to be developed and maintained independently of one another, but each component communicates directly with all the others (without need for external interfaces) and cannot run in isolation from the others.

*Service-oriented* architectures regard these components instead as “services,” which can be (and often are) implemented separately from one another. Each individual service runs in isolation from the others, and performs some function, or *service*. Web applications, which consist of multiple components (client, one or more different servers) interacting with one another over various types of network connections, are an example of this type of architecture.

**5.2 Suppose you're building a phone application that lets you play tic-tac-toe against a simple computer opponent. It will display high scores stored on the phone, not in an external database. Which architectures would be most appropriate and why?**

Since the application will run locally on the phone without need for a database, a component-based architecture seems like it would be the most appropriate design for the overall application as there is no need for any external interfaces. The computer opponent’s action set would likely be implemented using a rule-based system to determine moves based on those chosen by the human player (I’m fairly certain Tic-Tac-Toe was fully “solved” at some point in the 1950s or ‘60s).

**5.4 Repeat question 3 [after thinking about it; it repeats question 2 for a chess game] assuming the chess program lets two users play against each other over an Internet connection.**

Assuming each instance of the chess application runs on some type of local machine, the UI would likely be best implemented using an event-driven (there’s no point in forcing the machine to waste resources redrawing the board until a move takes place, e.g.) component-based architecture. It would likely be “best practice” to implement each element of the UI (board, pieces, move action, etc.) as a discrete component, with its own behaviors, and there are ample opportunities to reuse code for common properties (such as each “class” of piece and its legal behaviors, many of which are also shared with other classes of piece) through inheritance.

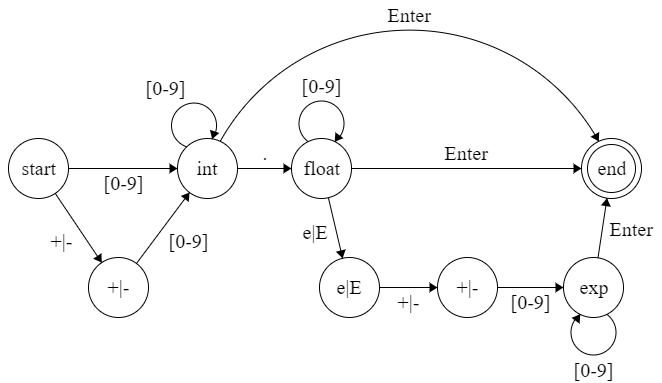
The requirement for network connectivity imposes a necessity for a distributed model of some sort, most likely taking an event-driven (as before, there’s no point in wasting bandwidth until a move actually occurs) multitier approach in handling players’ moves back and forth across the network. A client/server architecture would be particularly useful in synchronizing players’ game clocks if time is to be involved as it is in competitive chess matches. Depending upon the level of sophistication expected, an application instance’s functionality could also be distributed across multiple cores of the local machine - perhaps one to handle the UI, and one to handle network processes.

**5.6 What kind of database structure and maintenance should the ClassyDraw application use?**

The author explicitly states that he feels that storing this application’s drawings in a database would be “overkill.” That being said, the drawings should obviously be saved in *some* form of simple local data file.

Most likely, this would take the form of some text source enumerating the pertinent properties for each object in the application’s canvas (e.g., Cartesian coordinates of origin, length of each axis/radius as relevant to each object class, color) in some object format like XML or JSON so that they could be recreated upon subsequent invocations of the application. Without a “real” DBMS, maintenance would be minimal if not outright nonexistent, amounting to little more than expecting users to manage their own creations as they would with any other file format (documents, photos, etc.).

**5.8 Draw a state machine diagram to let a program read floating point numbers in scientific notation as in +37 or -12.3e+17 (which means -12.3 x 1017). Allow both E and e for the exponent symbol.**



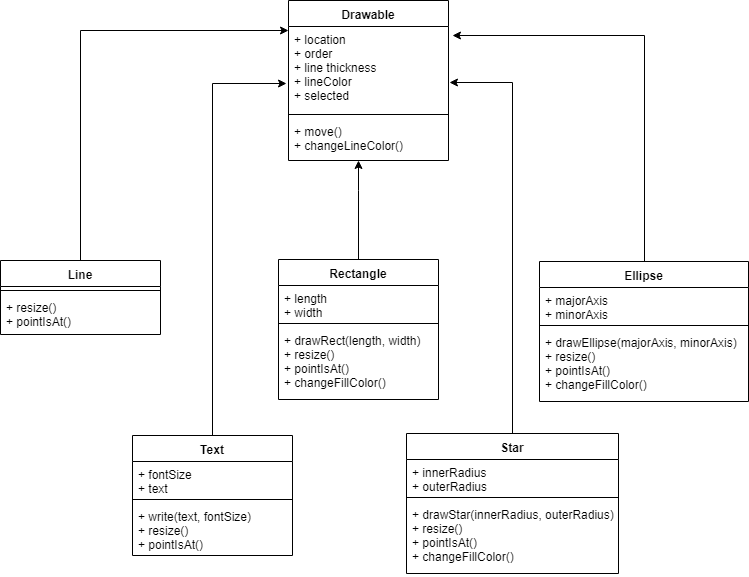
**6.1 Consider the ClassyDraw classes Line, Rectangle, Ellipse, Star, and Text. What properties do these classes all share? What properties do they not share? Are there any properties shared by some classes and not others? Where should the shared and nonshared properties be implemented?**

Every class shares a notion of location (perhaps implemented as a vertex or point of origin in the application’s implementation of Cartesian space), position in the stacking order (front, back, and everything in between), line thickness, color (though the classes other than Line could potentially allow for separate line and fill colors), and (depending upon the precise implementation of the UI) whether or not it is currently selected. The methods to modify these properties are therefore also shared across all of these shape classes.

While they all share some notion of size which defines their presence at a given pixel, the implementation (pointIsAt()) will vary from class to class: for example, Line is defined only by length, and (depending upon implementation) Text can be defined either by length and width, or only by font point. Even though Rectangle and Ellipse (as distinct from “circle”) are both defined by 2 dimensions - length and width for Rectangle, and (depending upon the underlying graphical implementation) either major/minor axis or F1/F2 for Ellipse - the approaches needed to generate the respective shapes from these figures will differ considerably. Star seems like it could be defined solely by a single radius, but a flexible approach could also utilize both “inner” and “outer” radii to define the length of the points - or even specify it directly. By the same token, the method to resize each of these shapes will vary depending upon each shape’s implementation of size.

In its most general form, these size parameters define virtually everything else about the shape classes. To this way of thinking, although any properties common to subsets of these shape classes would depend upon implementation details not shown in the text, their overlap would be relatively minimal beyond linear measurement such that all of these properties would likely have to be implemented individually for each class. It *could* be convenient to think of Line, Rectangle, and Star shapes in particular as something akin to SVG (collections of Lines, with Line itself a somewhat degenerate case), though that approach seems like it would be even more difficult to work with than simply defining specific implementations of pointIsAt() for each class. Thus, the shared properties should be implemented in the Drawable parent class, and the others in each individual shape class.

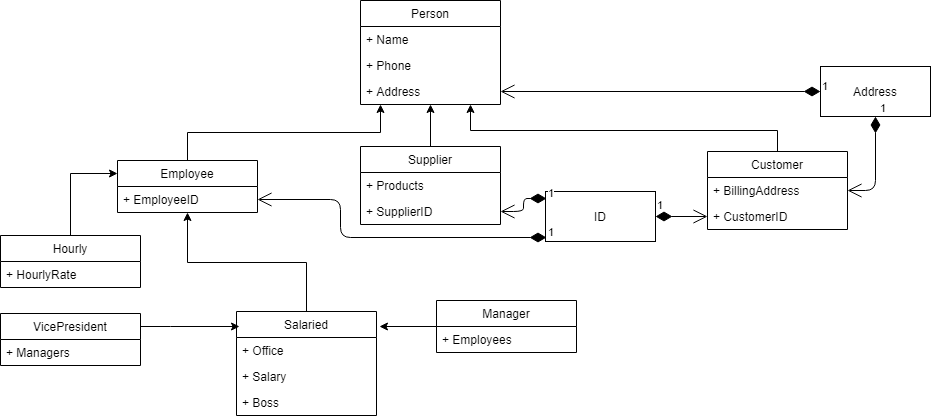
**6.2 Draw an inheritance diagram showing the properties you identified for Exercise 1. (Create parent classes as needed, and don't forget the Drawable class at the top.)**



**6.3 The following list gives the properties of several business-oriented classes.**

* **Customer — Name, Phone, Address, BillingAddress, CustomerID**
* **Hourly — Name, Phone, Address, EmployeeID, HourlyRate**
* **Manager — Name, Phone, Address, EmployeeID, Office, Salary, Boss, Employees**
* **Salaried — Name, Phone, Address, EmployeeID, Office, Salary, Boss**
* **Supplier — Name, Phone, Address, Products, SupplierID**
* **VicePresident — Name, Phone, Address, EmployeeID, Office, Salary, Managers**

**Assuming a Supplier is someone who supplies products for your business, draw an inheritance diagram showing the relationships among these classes. (Hint: Add extra classes if necessary.)**



I’m making the unrealistic and potentially counterfactual assumption in this diagram that VicePresident is not necessarily a child class of **Manager** since that relationship isn’t explicitly supported by the slate of properties as current enumerated. Clearly, since **Manager** is a subclass of **Employee** (and thus any field that takes an **Employee** should also accept a **Manager**), that parent/child relationship would, in fact, be semantically correct; I suppose they even *could* be implemented as the same class with a “Job Title” property to make the distinction explicit. But both of those solutions seemed like unneeded complexity for this case, particularly as the author encourages simpler (shallower) inheritance hierarchies.

**6.6 Suppose your company has many managerial types such as department manager, project manager, and division manager. You also have multiple levels of vice president, some of whom report to other manager types. How could you combine the Salaried, Manager, and VicePresident types you used in Exercise 3? Draw the new inheritance hierarchy.**

As I implied in my previous discussion of assumptions and caveats, one of the best ways to handle the **Manager** and VicePresident distinction (and probably the best way to make the hierarchy more parsimonious) is to treat them as a single class with a generalized “Subordinates” field and an additional explicit “Job Title” field (or, depending upon the level of sophistication desired, additional boolean fields “isManager” and “isVicePresident”). Certainly, common sense suggests that this system *should* store employees’ job titles if it’s also storing information on their pay rates and whether or not they’re managers or vice presidents.. Though this approach perhaps violates some Best Practice (intuitively, it feels mildly “unclean,” like leaving something as “NULL”), all 3 of these classes can even be condensed into a single one by simply leaving the “Subordinates” field as an empty array/list/set (or however it may be implemented) for salaried employees in non-managerial roles.

