1 Introduction

The purpose of this course is to explore software design.

It is the third course in the "design" series. The first course introduced you to the basics of design, and the first useful abstractions, including data abstraction, and functional abstraction. The second course introduced the notion of class abstraction.

Here, we focus on the even larger picture and try to address the problem of designing software. This of course includes everything you have learned until now, so all of it still applies. Additionally, however, we will worry about all of those ideas being put toghehter into a full piece of software. So our scope is slightly more large scale than before.

This problem of scale for designing software brings with it a host of problems that do not necessarily arise for smaller problems. And do not let anyone fool you—good design is hard. Why it is hard, I believe, is due to three reasons, a fundamental one, a technical one, and a pragmatic one.

(1) Aesthetics

The first reason why software design is hard is basically because design has an artistic component to it. When you thnk of design, or even the now popular notion of design patterns (we will see these later), we can go back to architecture. Architects have had design patterns for a long time—see Christopher Alexander's "Notes on the Synthesis of Form". And no one who knows any architect can deny that architects are in most part artists. Oh, they do know their science—good architects have a good knowledge of materials science—but beyond that, some of their decisions are in large part aesthetic.

Designing software is much the same. You will need to master the science—the algorithms, the data structures—and the techniques—design recipes, design patterns—and that will bring you far, but you will also need to develop an aesthetic sense for code. This is harrd to teach, as you can imagine, and is acquired mostly through training. I like to think of the series of courses you have had in your freshman year as starting to evelop that aesthetic sense. We will continue doing so here.

(2) The Abstraction Barrier

The second reason why software design is hard is because you need to repeatedly cross various abstraction barriers. We will have much more to say about this starting next lecture, but for the time being, we can think of it this way.

Every piece of software has, or should have, an abstraction barrier that splits the world in two: the clients and the implementors.

- The clients are those who use the software. They do not need to know how the software works, they just trust it.
- The implementors are those who build the software. They *do* need to know how it works, and their job is to distrust it. As an implementor, you should always assume that there are bugs in the software you are implementing, and you should search for them using testing, walkthroughs, and formal verification.

Unfortunately, it is not always straightforward to determine whether you are a client or an implementor. When you implement software, you are also using other software, for instance, libraries, so you are an implementor of some software but a client of other software. Things get even more funky when you consider that programmers often end up writing different parts of a big piece of software at the same time, meaning that they often switch back and forth between different parts of the program, and therefore have to switch from being a client to being an implementor and back to the being a client in shorts periods of time. This sort of context switching is quite difficult, psychologically speaking.

(3) Evolving Requirements

The third reason why software design is hard is because the design tends to be much more volatile. It does have to be this—rather, this is a consequence of how the software industry has evolved. (It may also have something to do with the fact that software is physically quite different than other products.) But a common experience in the industry is that customer specs (that is, what a customer will tell you they need, and against which you design your software) change regularly, especially as a customer sees previews of the software.

The main upshot of this state of affairs is that the design must adapt, evolve, as the specs change. And evolve in such a way that existing working code does not break. This calls for both methods for evolving or modifying a design, and for sofrware testing methods that can cope with an evolving code base. We will see that too.

This Course

In this course, we will study principles to help us design software, and examine how those principles are embodied in programming languages. The main concepts to recur in the course of the term include;

(1) Encapsulation and modularity

(2) Correctness

(3) Code reuse

(4) Design patterns

The emphasis in this course will be on object-oriented concepts. Partly, because this is a natural and useful way to approach the design and programming of a piece of software, and to tackle encapsulation and code reuse.

I should point out that *object-oriented* has a more general meaning than that commonly ascribed to it in the programming literature. In particular, by an object, I mean a rather vague notion. Roughly, for our purposes, *an object is just a value to which you can associate some behaviors*. By a value, I just mean it can be manipulated like one manipulates an integer, or a floating point number. It can be passed around to functions, it can be stored, if can be acted on. If you remember your first course here, you learned that functions can also be treated as values (we often call them first-class functions in that context), and doing so tremendously simplifies the programming model. Something similar happens with objects. As soon as you treat objects simply as values, they're nothing special, and the whole programming model becomes much simpler.

We will be programming in Scala in this course—previous versions of the course used Java—because Scala is very good at treating *everything* as an object. Java is pretty good in that respect, but not perfect (there are many values in the language that are not objects in our sense), and C++ is pretty abysmal in that same respect—the language forces you to be aware of many distinctions between objects and other entities in the language such as pointers to objects. This makes programming in C++ much more difficult and error-prone than it should be.