2013 15-214 INTRODUCTION TO DESIGN LECTURE NOTES

In this lecture we will take a first look at the software design process, and consider the concept of design patterns that come up again and again in software designs.

A BASIC APPROACH TO DESIGNING A DRAWING EDITOR

Let us explore design through a design exercize. In this exercize, we will sketch the design of an application for editing drawings, providing functionality similar to that seen in applications such as PowerPoint or Visio (our actual design reflects ideas in the JHotDraw application). A real drawing application would likely have many features, but here we will consider a simplified version. Consider the following set of sketched requirements:

R1: create a drawing and figures to go in it
R2: each figure should have a movable position within the drawing
R3: display the drawing on the screen
R4: support multiple kinds of figures, including lines and rectangles
R5: the kinds of figures should be extensible

While these requirements are simplified, they will be sufficient for now. We will ignore the user interface, except for the ability to display drawings on the screen.

Let's start by creating a _domain model_, representing the concepts in the requirements. We want to give each concept in the requirements a name, define it, and describe its properties. Some things in the domain model will turn into interfaces and classes in the software design--but for now let's think in the problem domain.

It's often helpful to examine the requirements specification and look for nouns and verbs. The nouns are often concepts that belong in our domain model. The verbs are often operations that will become methods when we translate the domain model into a concrete software design. In this case, we have the following nouns and verbs:

Nouns: drawing, figure, position, screen, line, rectangle
Verbs: create, move, display

Let us sketch the relationships between these. Which are concepts, and which are attributes of some other concept? A design guideline is that if we think of something as text or a number in the real world, it is likely an attribute, whereas if it has more complex structure or properties, it is likely to be a concept. In the domain model, we will draw the concepts as classes and the attributes as fields.

The domain model may also have specialization/generalization, analogous to inheritance in class diagrams. Is there a specialization/generalization among some of the concepts we found above?

Finally there may be relationships among elements in the model. For example, one concept may refer to another. Are there examples of relationships in the example above?

Let us now consider the concrete design of the software. We should ask whether the concepts we identified need to be represented as classes in the design. If the program needs to store information about the concepts, we will typically need them; that is true for most of the concepts in our example.

If we need a concept, we should think about representing its attributes as fields and think about what operations it should define as methods. Here we can take the verbs we identified before and find appropriate classes to add them to. Some verbs, such as create, may become constructors.

How do we decide where to add methods? A common solution is known as responsibility-driven design. Each class should have a set of responsibilities: typically the responsibility to know certain things and to carry out certain tasks. For example, a figure should be responsible for knowing its position, and for drawing itself on the screen.

Extending a domain model to a software design sometimes involves adding new classes and making design decisions. One design decision concerns the figure concept: do we need a figure class or interface that is separate from particular figures such as lines and rectangles?

To answer this question, consider the fifth requirement (R5) above. This requirement is not like the others. It does not describe functionality in the application. Instead, it is a _quality attribute_ of the design. It implies that the design (and corresponding implementation) should be easily extensible to support new figures in the future. How can we provide this extensibility in our design? Could the figure concept help with this?

If so, how should we represent figure? Should it be an interface or an abstract class? Might it be useful to have both?

From this example we can gain some initial insight into the design process. Our first goal is to represent the problem that is expressed in the requirements and domain model within the software system, so that it can be solved. However, there are multiple solutions to the design problem, and some are better than others. We have seen three considerations already: extensibility, flexibility, reuse, and complexity. We can provide extensibility by providing an interface that multiple implementations can implement. An interface also provides flexibility to implement a concept in completely independent ways. An abstract class decreases flexibility by fixing some design decisions, but provides code that can be reused across multiple classes. Having both an interface and an abstract class gives us both flexibility and reuse, but increases the complexity of the design, thus making it (marginally) more difficult to understand and implement.

EXTENDING THE DESIGN TO SUPPORT GROUPED FIGURES

Let us consider a design problem that comes up naturally in the domain of drawing editors. Many drawing editors, such as PowerPoint, allow figures to be grouped into a larger composite figure. Once grouped, the composite figure can be manipulated (moved, resized, etc.) as if it were an atomic figure itself. Moving the composite automatically moves all of the parts.

R6: group small figures into large ones, and manipulate larger group as if it were a figure

We want to keep our user interface code simple, and to accomplish that, we want to make sure that most of the design code doesn't have to know the difference between composite figures and primitive figures. In other words, even at the code level, we want to be able to manipulate a group of figures as if it were just a figure itself. How can we solve this problem?

After a class discussion of this design problem, students carry out a related in-class design exercize.

We then look at the lecture slides for today, considering the composite design pattern on slides 3-4, and studying the concept of design patterns and their benefits in slides 5-17. Where have we seen the composite design pattern before?

EXTENDING THE DESIGN TO UPDATE LAYOUT AND REDRAW AFTER A RESIZE

Let us examine another design problem, this one motivated by the interaction between displaying figures on the screen and functionality such as resizing a figure. We expect that different figures will need to implement resize in different ways. For example, a circle will simply change its radius, while a rectangle will change it length and width. A composite figure will have to change its size and then lay out its components within it again.

One thing all figures have to do, however, is re-draw themselves on the screen after the resize operation. We could include a call to updateScreen() in all implementations of the resize method. However, this duplicates one line of code in every method; is there a way we can reuse the method call so we only have to write it once? Furthermore, if we include a call to updateScreen() in each implementation, there is always the danger that when we add a new figure (which R5 suggests we will do a lot), we might forget to make the call, introducing a quite subtle bug into the implementation.

After a class discussion of this design problem, we look at the Template Method design pattern in lecture slides 30-31. Where have we seen this design pattern before?

The template method design pattern is very often necessary to achieving good code reuse. Remember that geting as much code reuse as possible is one of the most critical elements of the Virtual World assignment you are currently completing.