0:14

When designing a building, architects create sketches to visualize and experiment with various designs. Sketches are quick to produce and an intuitive way to communicate the design to their client but these sketches are simply not detailed enough for the builders. When architects communicate with the people who will be constructing the building, they provide detailed blueprints which contain exact measurements of various components. These extra details allow the builders to construct exactly what the architect envisions. For software, developers use technical diagrams called UML Diagrams to express their designs. This is what we will be learning. To recap, in previous lessons, we have only been doing conceptual design in the software development process through CRC cards. In the same way, architects use sketches to visualize and experiment with various designs. CRC cards are only good for prototyping and simulating higher level designs. To guide implementation, you need a technique that would be more like a blueprint. A UML Class Diagram, or just Class Diagram for short, allows you to represent your design in more detail than CRC cards can but it's still visual. Class Diagrams are much closer to the implementation and can easily be converted to classes in code. Abstraction, which you may recall, is the idea of simplifying a concept in the problem domain to its essentials within some context. Abstraction allows you to better understand a concept by breaking it down into a simplified description that ignores unimportant details. You can first apply abstraction at the design level using UML Class Diagrams then eventually convert the design into code. In this lesson, you are going to learn how to abstract concepts as a class in a Class Diagram. You will see that additional details can be represented in a Class Diagram compared to a CRC card. Don't get me wrong, CRC cards still have their place in prototyping and simulating various designs, but UML Class Diagrams are more suited for communicating the technical design of the software's implementation. By the end of this lesson, you will be able to convert Class Diagrams into code and even convert code into Class Diagrams with ease. So let's get started. Think for a moment on how you would abstract a food item in the context of a grocery store using a CRC card. You would have a food component with the responsibility of keeping track of its grocery ID, name, manufacturer, expiry date and price. It also needs to know if the item is on sale or not. Although CRC cards represent components, if you remember early on, the goal of design is for the components, when they are refined enough, to become functions, classes or collections of other components. Since we use Java in this course, where an abstraction is formed in a class, we focus on classes. So how would the food class look in a Class Diagram? This is the Class Diagram representation of the food class. Each class in the Class Diagram is represented by a box. Each box is divided in three sections much like a CRC card. The top part is the Class Name. This would be the same as the class name in your Java class. The middle part is the Property section. This would be equivalent to the member variables in your Java class and defines the attributes of the abstraction. And finally, the bottom part is the operations section which is equivalent to the methods in your Java class and defines the behaviors of the abstraction. Properties, which are equivalent to Java's member variables, are mainly composed of the variable name and variable type. Variable types, much like in Java, can be classes or primitive types. Operations, which are equivalent to Java's methods, are mainly composed of the operation name, parameter list and return type. For example, a food object could have a method to return if it is on sale or not. To show this, we write a method called isOnSale. This method will return a boolean to represent if it is on sale. A boolean value is either true or false. The isOnSale operation takes no parameter, so we don't include a parameter list. Suppose isOnSale takes a date parameter and returns true if the food item is on sale on the given date, our Class Diagram will now look like this. Note how the parameter list follows the same format as the Class Diagrams properties. Now, if we compare the CRC card to our Class Diagram, you might notice how some of the responsibilities on the card turned into properties in the Class Diagram. Some, specifically isOnSale, became an operation. You could certainly use CRC cards for abstracting an object such as a grocery food item but there are simply too many ambiguities that prevent a programmer from translating a CRC card to code. One ambiguity is that a CRC card does not show a separation between properties and operations. They are all listed together. Now that we have a Class Diagram representation, let's finally implement it in Java. Class Diagrams are very close to implementation, making the translation to Java very easy. Class name in Class Diagram turns into a class in Java. Properties in the Class Diagram turn into member variables. And finally, Operations turn into methods. You will probably notice that everything is public. We will assume that for now. Later, you will learn about access modifiers of member variables and methods in Java. Converting code to Class Diagram is also straightforward. Consider, for instance, this code. To convert this code to a Class Diagram, we identify ClickCounter as the class name since that's what the class is named in the code. We, then, set the member variable, count, as a property. This property has a type, int. Finally, the methods setClickCount and getClickCount become operations. setClickCount takes a parameter. Therefore, we will include the parameter list specifying the parameter name and its type. getClickCount has a return value. Therefore, we also have to specify the return type. Despite the extra details that the Class Diagram can provide, they still can't replace CRC card for simulating and prototyping different designs. CRC cards are cheap and small. It is easy to play with different designs with your team in the physical world and the fact that it is far from code, makes you focus on the problem and not the implementation. Class Diagrams, on the other hand, are much closer to code as you have seen. This is great if you want to clearly communicate your technical design to the developers but since you have to specify code-specific things like parameter lists and return values, these are too detailed for conceptual design. The details would be a distraction and time-consuming to describe when creating

your initial designs. You now have learned to represent abstractions at a UML Class Diagram. Although CRC cards still have their place for prototyping and simulating designs, the Class Diagram gives you a technical description of what the implementation looks like. But we have only scratched a very tiny surface of UML Class Diagrams. We will continue to use Class Diagrams in many areas of this specialization. In the other lessons, we will expand on the Class Diagram by revisiting the other object-oriented design principles.