Often there exist relationships between the custom data types we create in a programming project. In the context of a graphical user interface (GUI), a window may consist of a collection of widgets, where a widget could be a button, scrollbar, or edit control. While it is possible to design a GUI from a set of disconnected objects, this design would not be leveraging the similarities of the objects and would have a great deal of redundant code.

This paper will help us to identify the basic types of object relationships, learn design tools enabling us to identify and visualize these relationships, and discover guidelines to help us find the best way to capture these relationships in our data structure designs.

**Union Design Patterns**

On the simplest level, an object can be built from primitives (built in types such as integers and floats) or from other objects (another class). When an object is defined in terms of another object, two possible relationships exist: “has-a” or “composition” where one object consists of a collection of subordinate objects, and “is-a” or “derivation” (also called “inheritance”) where one object is a manifestation or type of a parent object.

**Has-A**

The simplest type of relationship is when one object is defined in terms of a collection of other objects. Back to our GUI example, a scrollbar consists of two buttons (up and down arrows), the moveable button (slider), and the track (where the moveable button slides). This is commonly called a “has-a” relationship because the scrollbar “has a” down-arrow button, it “has an” up-arrow button, “has a” slider, and “has a” track.

Note that in composition relationships, one object has a distinct and persistent relationship with the parents. The up-arrow button, for example, behaves exactly the same in the scrollbar as it would if it were by itself.

**Is-A**

A more complex relationship between objects is when one object inherits many (but not necessarily all) of the properties of a parent. In many cases, it is a “type of” the parent. Back to our GUI example, a window has a collection of widgets. Every widget has a collection of properties, including the position \((x, y)\), size \((x\text{size}, y\text{size})\), and some draw functionality. A button “is a” widget. It inherits the position and size properties from its widget parent, but it also has a collection of other properties (text label, click behavior, and enabling state to name a few). An edit control is another widget. It also inherits the position and size properties from its widget parent, but has a different collection of properties (user data, cursor location, and click and keyboard behaviors for example). Both the button and the edit control exhibit an “is-a” relationship with the widget because they inherit common properties and have a collection of other properties unique to them.

**UML**

Possibly the best way to visualize object relationships is through UML class diagrams. Through this tool, we can represent both composition (has-a) and inheritance (is-a) relationships.
UML for Has-A

The composition relationship between classes is demonstrated in UML in a hierarchy by drawing arrows to the child (or newly built class) from the parent (what we are building from) using an open arrow (→) with dashed lines (- - -). We typically also specify the number of instances (called multiples) a given parent object is used in the child. This is important because composition relationships can include a large number of parents. This could also be a range (1-4) or unlimited (*). Common multiples include: 0..1 (no instances or one instance), 1 (exactly one instance), 0..* (zero or more instances), and 1..* (at least one).

In this example, the ScrollBar object has a Slider, a Track, and two Buttons. Observe the four member variables (slider, track, upButton, downButton) typical of “has-a” relations. There are actually four flavors of “has-a” relationships: dependency, association, aggregation, and composition. In the context of this class, we will treat them all the same.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependency</td>
<td>Objects of one class work briefly with objects of another class.</td>
<td>A local variable in a method</td>
</tr>
<tr>
<td>Association</td>
<td>Objects of one class work with objects of another class for a prolonged amount of time.</td>
<td>A private member variable</td>
</tr>
<tr>
<td>Aggregation</td>
<td>One class owns but shares a reference to objects of another class</td>
<td>A public member variable</td>
</tr>
<tr>
<td>Composition</td>
<td>One class contains objects of another class</td>
<td>The ScrollBar example above.</td>
</tr>
</tbody>
</table>

(Miles & Hamilton, 2006, p. 84)
UML for Is-A

The inheritance relationship between classes is demonstrated in much the same way except a closed arrow (► or ◄) is used. Note that you can’t have more than one instance of a parent because, unlike composition relationships, inheritance is essentially the elaboration on a single type.

In this example, there are three manifestations of a Widget: the Button, the Edit control, and the Scrollbar. Every child contains all the elements of the base Widget class. In this example, the rec property and the draw() function are in each of the derived classes.

There are two types of “is-a” relationships. The first is when the child adds properties and methods to the parent. This makes the child an elaboration of the parent. In the above example, Button adds properties (label and hotkey) and methods (pressEvent, setEnabling, getEnabling, setHotkey, and getHotkey) to Widget.

The second type is when the child redefines a method from the parent. This is called polymorphism or “multiple shapes.” In our example above, Button, Edit, and Scrollbar each have a special version of draw(). We signify this type of relationship by italicizing the method draw().
Example

Both composition and derivation relationships can be specified in a single UML diagram. Consider the following example consisting of all the GUI elements discussed above.

In this example, a Window has a TitleBar and an unbounded collection of Widgets. There are three types of widgets: a Button, an Edit control, and a ScrollBar. While the Button and Edit controls are not composed of any other objects (ignoring the Rectangle type for the sake of simplicity), the ScrollBar is composed of four other objects (upArrow and downArrow SimpleButtons, the track, and slider).
Design Considerations

In order to minimize the chance that our object relationship design will become unwieldy or overly-complicated, a few design rules are in order: manage variant and invariant attributes carefully, be conscious of layers of abstraction, and be aware of the tradeoffs between universalizing vs. abstracting.

Variant and Invariant attributes

When working with “is-a” relationships, it is important to draw the line between the properties invariant between the children (every child and the parent share the same property), and the properties that are designed to be variant (many children are likely to have a different implementation). In one extreme, *every* property is variant. This begs the question of whether the children really should be related at all. If the children have nothing in common, why specify a relationship that does not truly exist or is meaningless? On the other extreme, if every property is invariant, then there are no differences between the children. If this is the case, why do we specify different children if there truly are no differences?

When designing an “is-a” relationship, specify in the parent or base-class which properties are designed to be variant and which are designed to be common between all children.

Layers of Abstraction

A superclass or parent class represents a higher level of abstraction than the child. In our GUI example above, a *Widget* is more abstract than a *ScrollBar* because a widget can be one of many different types of controls. A *ScrollBar*, in turn, has many components. The *Slider* is one component, representing an object less abstract than either the *ScrollBar* or the *Widget*. As you can imagine, the number of levels of abstraction can be quite large. One design consideration when working with object relationships is to try to make the levels of abstraction consistent and meaningful.

Possibly the best example of this is the major taxonomic levels of living things. This taxonomy, called the Linnaean taxonomic system (from Carolus Linnaeus 1707-1778 who described it in 1735), consists of seven major divisions: {Kingdom, Phylum, Class, Order, Family, Genus, Species}. There are two aspects of this taxonomy: hierarchy and rank. **Hierarchy** is preserved because the lower you go in the taxonomy the more similar the organism. Members of a Species, for example, are guaranteed to be more similar than members of a Genus. The second, **rank**, relates to the layers of abstraction. The level of specificity for definition of a Species is consistent across the taxonomy. In other words, the requirements for creating a new Species is the same across all Genera of the taxonomy.

Pay careful attention to maintaining a consistent abstraction (or rank) level across your design (Wong, Union Design Pattern, 2007). Changes made to the hierarchy contrary to the layers of abstraction are likely to compromise any benefit of the abstraction in the first place.
Universalizing and Abstracting

There are two basic approaches to representing object relationships: universalizing and abstracting. **Universalizing** is the process of making a single type containing all possible attributes. In our GUI example above, this would entail creating a single object containing a superset of the properties of every widget type. This would mean that the vast majority of the properties would not be meaningful in a given context. The property `cursorPosition` might be very useful in the `Edit` widget, but have no meaning in the `Button` widget. The approach of creating one object representing everything is called universalizing.

**Abstracting**, on the other hand, is the process of creating an entity capturing the *essence* of relationship and allowing specific implementations to describe their details. In this scenario, only the pertinent properties are available for the object; no irrelevant attributes exist.

Object oriented design principles lean heavily on the abstracting approach over the universalizing approach in an effort to reduce irrelevant data and develop more cohesive data types. Be mindful of temptations to create universal designs; a better abstract design is probably available.

Works Cited


