Tracy Mann

CSE210

Bro. Parrish

Section A8

Inheritance barrows heavily from its non-IT meaning. In this classes explanation the terms super or base class was used for the parent, and the terms sub or derived were used for the child classes. Similar, but not exactly the same as genetics children classes take on the traits of the parent class. In this case, all the non-private traits of the parent class are in the children automatically and these traits are the descriptors and methods. A new protection level is introduced specifically for inheritance. The protected security allows child classes to have access to the parent class traits defined as such, without leaking that access external to classes that are not descended from the parent class. This reinforces our ability to maintain encapsulation as we get each object more specific to a detailed individual responsibility.

The two greatest advantages I see with this is to reduce duplication of code and the isolation of distinctly different but similar classes. The reduce of duplication is done by allowing the child class to automatically use permitted descriptors and methods allowed to be used as if defined locally without having to do so. This allows for the common elements between two distinctly different classes to still have that common definition from the same source while maintaining the protection of their individuality of their internal differences. The parent will not know of the differences and neither will the siblings. This strengthens encapsulation in a family of classes.

An example to use inheritance in this manner is two classes, like dog and cat. They are distinctly different, but still have some things in common. They are both Mammals. In this example, we would define a parent class of Mammal to contain all of the traits needed for both dogs and cats, like hair color and hunt. Then the specific traits that belong to dog in the dog class and respectively the same for the cat class. Something like, bark or fetch would be a dog trait and meow or climb a tree would be a cat trait.

In my code I use the activity class as the parent with three children classes including: Breathing Activity, Reflection Activity, and Listing Activity. I inherit the parent class with the ‘:’ operator as follows in the Breathing Activity class, “public class BreathingActivity : Activity”. In the Activity class I defined a common trait called \_duration as available to be used by all children as follows, “protected int \_duration = 0;”. In turn, the prompt for duration method is defined in the activity class and populates the duration descriptor while being also made available to be used by all children classes as follows, “protected void PromptForDuration(){…}”. In the child class listing activity the class specific method of run listing activity uses the inherited prompt for duration method as follows, “public void RunListingActivity() {…; PromptForDuration(); …}”. Later in the same function, this class uses the common descriptor of \_duration to drive how only to run this process for, like this, “DateTime done = dateTime.AddSeconds(\_duration);”. Similarly, the other two child classes call these two common components in their own way when they need it for their purposes unrelated to this class. At the same time, the \_timesUsed descriptor is not used directly by any of the children and thus is declared as not inherited, like this, “private int \_timesUsed;”. The children have no need to know about this descriptor nor how it works. Instead, they use the inherited method Report Usage that does this work for them, as such, “protected void ReportUsage(int duration, DateTime? dateTime = null) {…; \_timesUsed++; … }”.

Demonstrating that Abstraction, Encapsulation and Inheritance support each other to provide flexibility in targeting very specific cases while protecting the internal working of the objects.