Lecture 17 - The OO Paradigm - Inheritance, Polymorphism, and Dynamic Binding

Collin Roberts

November 9, 2023

Outline

Example

1. Consider a `File` class, with an `Open` method.
2. An instantiation of a `File` might be stored on
   1. hard disk,
   2. flash drive or
   3. tape,

   so the code inside the `Open` method must be different in each situation.
3. The `File` base class has derived classes
   1. `HardDiskFile`,
   2. `FlashDriveFile` and
   3. `TapeFile`,

   each having an `Open` method specific to its medium.
4. The `File` class has a dummy `Open` method.

Example

Definition 1

At run time, the system decides which `Open` *method to invoke. This is called* **dynamic binding**.

Definition 2

The `Open` *method is called* **polymorphic**, *because it applies to different sub-classes, differently.*

Example

5. Problems with Dynamic Binding/Polymorphism
    1. We cannot determine at compile time which version of a polymorphic method will be called at run time. This can make failures hard to diagnose.
    2. Similarly a S/W product that makes heavy use of polymorphism can be hard to understand and hence hard to maintain/enhance.

1. OO treats data and operations on that data together, with equal importance.
2. So a well-designed class does a good job of modelling some real-world entity.
3. A well-designed class also fosters **re-use**.
4. High cohesion + loose coupling → fewer regression faults.
5. Postdelivery maintenance is also improved.

1. In the 1960s and early 1970s, S/W Engineering was non-existent.
2. The **Code-And-Fix** model was the norm.
3. Hence the Classical model was most developers' first experience with S/W Engineering practices.
4. Adopting the Classical life-cycle model yielded major improvements in productivity and S/W quality at the time.
5. However as S/W products grew larger and more complex, the weaknesses of the Classical paradigm (which we have already discussed) became more pronounced, and the OO paradigm was proposed as a better alternative.

Problem: There is a **learning curve** associated with adopting the OO paradigm for the first time. The first project done with OO takes longer than doing the same project with the Classical paradigm. This is particularly pronounced if the project has a large GUI component. But after the initial project,

1. the re-use of classes in subsequent projects usually pays back the initial investment (again, this is more pronounced with a large GUI component) and
2. post-delivery maintenance costs are reduced.

Definition 3

Any change to the base class affects **all of its descendants**. *This phenomenon is known as the* **fragile base class problem**.

1. In the best case, all descendants need to be recompiled after the base class is changed.
2. In the worst case, all descendants have to be re-coded then re-compiled. This is bad!

To mitigate this, meticulously design all classes, especially parent classes in an inheritance tree.

1. Unless explicitly prevented, every subclass inherits **all** the Properties/Methods of its parent. The reason to create a subclass is to add Properties/Methods. Hence objects lower in the inheritance tree can quickly become large, leading to storage problems.
2. Recommendation: change our philosophy from "use inheritance whenever possible" to "use inheritance whenever appropriate".
3. Also explicitly exclude Properties/Methods from being inherited, where this makes sense.

1. This is especially true of programming in an OO language. OO languages have constructs that add unnecessary complexity to the S/W product when they are misused.
2. We must endeavour to produce high-quality code when working with the OO paradigm.

1. As mentioned earlier, the OO paradigm is certain to be superseded by some superior methodology in the future.
2. **Aspect Oriented Programming (AOP)** (covered in §18.1 in the text) is one possible candidate to replace the OO paradigm.