[0691] EXTRA: On Reuse, both DRY and SOLID

In order to understand Inheritance and Polymorphism better, it is helpful to understand several other things about what programmers actually do in "the real world."

Different Kinds of Programmers for Different Kinds of Programs

Possibly the most important thing to remember when reading about programming is that there are many, many different kinds of programs out there and, thus, many, many different kinds of programmers. For example:

- Full-Stack Web Developers - understand and program websites, typically using database systems to store and retrieve data.
- Scientific Programmers - create programs that gather and analyze scientific data.
- Mobile App Developers- create apps for various mobile devices
- Big Data Programmers - work with HUGE datasets
- Small Device Programmers - write code for memory-constrained environments
- O/S Programmers - write code for Operating Systems
- Game programmers - write code that's optimized for Graphics processors
- Advertising programmers - write code for interactive web advertisements
- Critical Systems programmers - write programs that people's lives depend on
- C# Book Authors - don't really write much code at all
- High-Performance Programmers - try to write the fastest programs possible (Bitcoin anyone?)
- Mainframe Programmers - still exist
- etc. etc. etc.

My point is that *when you read or hear someone else talking about programming, you need to make sure you understand what kind of programming they are talking about.* There really is no such thing as "General Purpose" programming anymore. Different programmers have different priorities and are focused on different aspects of programming as a result.

One of the most famous quotes in Computer Science is ***"Premature Optimization is the root of all evil."*** - Donald Knuth. (PLEASE take time to read **other quotes from Dr. Knuth (https://en.wikiquote.org/wiki/Donald_Knuth)** . They are super valuable!) In thinking about that quote, remember that "optimization" doesn't always mean "speed" optimization. It applies equally to all forms of optimization including...

Code Reuse and DRY

Code Reuse is often held up as the Holy Grail of programming. Don't get me wrong - done correctly, reusable code is extremely helpful/valuable/cool/etc. The key point of that sentence is "done correctly." Most of the time, reusable code is not done correctly. What do I mean? By "done correctly" I mean that the code is useful in a wide variety of situations, its function is easy to understand, it is natural and easy to use, and - most importantly - it (probably) doesn't need to be updated later. The great irony in computer programming is that the more a piece of code is reused, the harder it becomes to change it without breaking the stuff that depends on it.

I'm talking about code reuse here - in the inheritance chapter - because code reuse is held up frequently as one of the key benefits of inheritance. Again, done correctly, that can be true. However, it is not easy to do it correctly (and it is very easy to do it wrong).

Important: I'm not saying reusing your code is bad or wrong. Inside of *your program*, you should always be looking for opportunities to create useful methods that help make your code more compact and more readable. However, reuse is a "nice to have" quality for most programs. Personally, I prioritize readability, maintainability, scalability and "improvability" over reusability for the kind of programs I typically write.

In programming circles, the concept of DRY is very popular. **DRY stands for "Don't Repeat Yourself"** and is a direct appeal to programmers to increase their reuse of code. And again, within limits, I agree with that. However, there are many examples of people that have taken that concept too far and wound up with very brittle, hard to maintain, hard to understand code. Your goal should not be DRY for DRY's sake. It should be useful, understandable, maintainable code that might have several reusable parts.

An exception to this Advice: If you are writing code specifically intended for use by others - libraries, base classes, Microsoft APIs, etc. - then reuse your goal. It's your job. It's what you do. In that case, you should spend time getting really, really good at it. There are many techniques and concepts that class library designers need to employ that other programmers don't normally focus on and that's fine.

Bottom Line: Don't become a slave to the pursuit of reuse and DRY. These can become rat-holes that waste time and sabotage your program's design if you aren't careful.

SOLID Principles

SOLID is an acronym for a mindset one should have when doing Object Oriented Design with an eye towards maximizing reuse and maximizing general-purpose usefulness. It forces you to think critically about classes and the relationships between them. SOLID stands for:

- Single Responsibility Principle
- Open/Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

SOLID came out of the work of Robert C. Martin (AKA "Dr. Bob) and is discussed continuously (with large amounts of religious zeal) by programmers all over the Internet. Again, it is a set of principles, NOT hard and fast rules. Depending on the kind of program you are writing, the need for SOLID principles can vary. All things being equal, SOLID principles can help you decide which of two possible designs might work better. (All things are rarely equal however!)

Here's what each point is about:

Single Responsibility Principle

Every class should be responsible for a single "functional capability" of the program and that class should entirely encapsulate that capability. As Dr, Bob says "A class should have only one reason to change." In class, we talked about the need to keep the UI code separate from a program's logic - that is an example of this principle.

Open/Close Principle

"Classes should be open for extension but closed for modification." By "open for extension," we are referring to the ability to add additional fields of methods to the class. By "closed for modification," we mean that its public interface cannot be changed. In class, I said that you should keep everything private unless there was a really good reason and treat public stuff like a wedding vow. That's an example of the Open/Close Principle. Later, we will see that the use of abstract classes can help with Open/Closed designs too.

Liskov Substitution Principle

A fancy name for the concept that sub-types should work just like the types they are based on. Or, in our context, sub-classes should do everything that they base class can do (and more). To me, this is a "Duh" principle. Put another way, your subclass shouldn't screw around with stuff your base class does.

Interface Segregation Principle

"Clients should not be forced to depend upon interfaces that they do not use." - Robert C. Martin

If the methods in a class are used by some of the sub-classes but never by other sub-classes, you need to split out those methods using abstract classes and/or interfaces.

Dependency Inversion Principle

Use interfaces to eliminate dependencies between sub-classes and their parent classes and therefore increase reusability.

The Bottom Line

Creating truly reusable inheritance hierarchies is hard. In the rest of this class, you are only getting a taste of the complexity involved. Fortunately, USING the inheritance hierarchies that are already built into C# is easy.

More Info:

https://programmingwithmosh.com/object-oriented-programming/what-text-books-tell-you-about-inheritance-in-oop-is-wrong/ (https://programmingwithmosh.com/object-oriented-programming/what-text-books-tell-you-about-inheritance-in-oop-is-wrong/)

https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design (https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design)

http://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/ (http://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/)

https://team-coder.com/solid-principles/ (https://team-coder.com/solid-principles/)

"What Programmers do with Inheritance in Java" - **https://www.cs.auckland.ac.nz/~ewan/qualitas/studies/inheritance/TemperoYangNobleECOOP2013-pre.pdf (https://www.cs.auckland.ac.nz/~ewan/qualitas/studies/inheritance/TemperoYangNobleECOOP2013-pre.pdf)**