Card Creator

This document outlines some of the decisions and trade-offs made during development of the card creator project as well as how to set it up and run it.

Setup and compilation

The project should be easy to set up and compile. It uses the .NET framework 4.7.2 and C# 7.3 so, assuming you have the required SDKs etc installed, it should 'just work'. Make sure you build the solution first and get the necessary NuGet packages installed. Then make sure you select the 'CardCreator' project when you run/debug; the 'Startup project' setting might be set to another project.

Software design overview

The `CardCreator` project depends on the `IO`, `Result`, and `StringUtils` projects. These are all very small projects that serve a very specific purpose and could easily be decoupled from the main project.

Interfaces, interfaces everywhere

In this project I strove to separate concerns and decrease coupling as much as possible. This resulted in a very liberal use of interfaces as I, whenever it was feasible, opted against depending on concrete classes, instead depending on Interfaces.

This naturally constrains what each function can do and greatly aids in building a decoupled system. Furthermore, it also makes it very easy to create generic functions and interfaces that only care about what a parameter can do as per it's interface rather than how the actual implementation works.

`abstract class` vs `interface`: inheritance or implementation?

In most cases, I find an interface to be more useful than an abstract class, as it grants greater freedom to the implementing class. However, in some cases, inheritance is useful. As interfaces do not support default implementations (although that seems to be changing with C# 8.0, for better or worse) inheritance is currently the best way to go for code reuse.

In this project, some examples are the `Stat` class and also the `TabViewModel<T>` (itself implementing the `ITab<T>` interface) which is inherited by `CardTabViewModel` and `MoveTabViewModel`, allowing the tab view models to share JSON import and export code.

Favoring composition over inheritance

Often times, however, inheritance may not be what you want, and you can achieve your goal with composition. This is the approach taken in the `IOResult<T>` class. The `IIOResult<T>` interface implements the `IResult<T>` interface, so any implementation must provide the same functionality as an `IResult<T>`. In addition to this the `IIOResult<T>` also has a an `IResult<T>` property, which the `IOResult<T>` class can use in conjunction with its own properties to create a more accurate modeling of the result of an IO operation.

Inversion of Control

The `TabViewModel<T>` and its subclasses use the inversion of control principle to have their IO services and view models injected into them instead of creating them themselves. The same goes for other classes that need a version of an interface.

The `IStat` interface

When dealing with cards like this, stats are an important part of the process, but how the stat behaves might be different based on which stat it is. For instance, the level stat works in multiples of one and has a range from 1 to 100 inclusive, whereas the HP stat increments in multiples of 10 and has a range of 10 to 200. I ended up making these different instances of the abstract class `Stat`, implements the `IStat` interface, so that the models can simply use an `IStat`, but the stats can still share code where necessary.

On input validation

Due to how I structured the types in the view models for the card and move tabs, there was very little validation that had to be done outside of the basic validation that WPF gives you for free. Because all stats are constrained to unsigned integers, negatives and non-numeric input causes errors. As for string fields, they will automatically get truncated if they're too long. Also, using the file type restriction on the Windows save and load dialogs offers means the end user simply can't choose files whose extensions are not explicitly listed.

Problems encountered

Learning WPF and MVVM Light

One of the major hurdles for this project was learning to work with WPF. The module's classes only gave us a fairly rudimentary introduction to the framework and skipped over a lot of details that could have come in handy, essentially leaving the students to figure these things out for themselves. Naturally this meant that things moved rather slowly in the beginning, though they definitely sped up towards the end.

Time restraints

While not a technical problem, this module has not been prioritized as highly as others this term. As such, I admittedly have not poured as much time into this as I ought to have. This is probably the biggest reasons for the project's shortcomings.

Database setup issues

A big issue which also played a key role in the arguably most significant missing piece in this project was setting up the database project. Coupled with the lack of time mentioned previously, this meant that after spending a good while frantically troubleshooting the database project setup (it wouldn't compile as a fresh project–probably some innocuous misconfiguration somewhere), I had to pull the trigger and decided my time would be better spent working on the level editor project.

Shortcomings

Due to the problems mentioned, the project does have some shortcomings. The biggest one is definitely the lack of a database. With no database integration up and running, a couple of the functional requirements are not covered: storing, deleting, and browsing cards in the database is not possible. Furthermore, while moves can be saved and loaded from JSON files (just like Pokémon themselves), they can not be loaded in as options in the 'Pokémon' tab.

Elements you wanted to include but did not get time for

Apart from the obvious database integration, there are a number of elements I would have liked to include.

A more thorough integration of game rules

Due to the lack of time, certain features had to be cut and some game rules had to be simplified. Most notably the 'Energy cost' for moves was cut, along with 'Retreat cost'. These two stats offered some interesting challenges as they both consist of an arbitrary collection of energy cards, so the idea was to implement them as a `Dictionary<Type, uint>` where the total value of the dictionaries values could be no more than a set amount (4 and 3, respectively). The basic idea isn't particularly complicated, but when it came to creating a sensible interface for it in WPF (which would include automatically showing and hiding controls), the scope started to grow a bit too large. After working with the `StackPanel` control in the level editor, though, I can see how it could be solved well.

Another thing relating to moves is damage multipliers and additions. Pokémon TCG moves often deal damage that isn't just a set number, but can be '20x' or '30+' where the amount of damage they do depend on outside factors such as status conditions and coin flips. It would be easy enough to just be able to tag on a 'x' or '+' after the damage, but when using moves like this, it would be useful to have other damage caps than with normal moves. Where having a move that deals 150 damage is incredibly strong, having one that deals 150 damage multiplied by a number of coin flips is definitely in the 'broken' realm.

A more robust type system

This is less interesting from the application point of view, but very interesting when thinking about the language and domain modeling and what guarantees you can get out of the application.

In the current version of the application, Pokémon types are modeled as an `enum`. This is adequate for the current implementation, but doesn't provide any strong guarantees. In earlier iterations I played around with using interfaces to create 'phantom types' (a form of marker traits–see the next subsection for a quick explanation) which would allow for some really interesting interplay to be encoded into the type system, but this isn't idiomatic C# and it became too much of a hassle for the payoff, so I had to let it go in the end. Given more time (and perhaps more freedom in language choice), that would be a very interesting avenue to explore.

On phantom types

> A phantom type is a parameterized type whose parameters do not all appear on the right-hand side of its definition [...] – The Haskell wiki on phantom types

So what does this mean? In practical terms it allows us to take a type `T` parameterized by some other type `U` and then restrict what instances of `T` can be used in functions.

To give a more concrete example: Imagine a class `Energy<I>`, representing Pokémon energy cards, where `I` is an interface and will be used to decide what Pokémon type it is. In this case, we could declare a function:

```
int ThunderTackle(Energy<IElectric> e, Energy<IColorless> c)
```

This function can then only be called with the correctly typed versions of the Energy class, even if the two objects themselves are the exact same (i.e. not inherited, but of the same class).

In the end, however, trying to force this way of coding onto C# is probably going to create more work than it's worth, but it would be a very interesting experiment.

Further UI refinement

The user interface is still very bare and doesn't offer a lot of help with anything. I'd like to change this to at least make it inform you of limits of stats / string lengths and the like.

Highlights

While the project certainly has its shortcomings, there are also some things I am very pleased with:

`IResult<T>` and `IIOResult<T>`

The `IResult<T>` (from the `Result` project) type is a fairly rudimentary implementation of Rust's `Result` type or Haskell's `Either` type. It is intended to be used for operations that can fail, but where it's not appropriate to throw an exception.

While the 'correct' way to use exceptions and exception handling can be debated for ages, I am of the opinion that exceptions should only be used for truly exceptional circumstances, such as if the application suddenly runs out of

memory or something else happens that is beyond the developer's control and that cannot be recovered from. This is why the IO operations (the `IO` project) return `IResult<T>` or derivatives: if something goes wrong when saving or loading data, this is something that should be handled by the developer and should not cause a crash.

So we have the basic `IResult<T>` interface; why do we need `IIOResult<T>`? If we try and think about all the outcomes of a save/load dialog we get 'success: everything went as expected' and 'error: something went wrong and we could not complete the operation', but what if the user cancels the operation? That shouldn't be an 'error' as nothing went wrong–it's a valid action after all–but we also could not complete the operation. So we introduce a third variable, a `bool Completed`. This allows us to get a more complete picture of what happened during an operation and lets us make a more informed decision as to how we want to deal with it.

Overall, I find this approach to be more elegant and more ergonomic than trying to catch exceptions everywhere.

`enum` control generation

With the release of C# 7.3 came the ability to use `T : System.Enum` for generics in functions and classes. Using this newfangled ability to have a class be generic over `enum` and after getting more familiar with WPF and user controls, I created classes which would take an `enum` value and generate all labels for a combo box based on the enum and its type (`ComboBoxViewModel`). Being able to use this for all enum combo box components provided a nice abstraction and sped up the enum process considerably.

Architecture

And finally, without repeating everything that was mentioned in the Software Architecture section, I am quite happy with how the architecture turned out and the use of interfaces. This was my first chance to program something in C# after I realized just how useful interfaces could be, and it turns out that the power it gives you really is something.