Creating a Software Solution

An Overview of Software Patterns

Our Story so Far

- **Found A Problem To Solve**
- **Built Requirements and Use Cases**
- **Created A Clickable Demo**
- **Created a User Experience**
- **Designed a Database**
- **General Objects Design**

Next Step

We have started looking at the code structure. Now we want to go deeper into patterns of software design to speed our implementation

What Are Software Patterns?

- **A Pattern is a best-practice or well-defined approach to solving a problem**
- **New Designers can use patterns to quickly add a number of solutions to their bag of tricks**
- **The Original Patterns book is called "Design Patterns: Elements of Reusable Object-Oriented Software" and often referred to as the GOF (Gang Of Four) Patterns books**

Common Patterns

- **Strategy**
- **MVC (MVVM, etc.)**
- **Factory**
- **Iterator**
- **FlyWeight**
- **Adapter**
- **Builder**
- **Singleton**
- **Command**
- **Interpreter**

Strategy Pattern

Strategy is a way to do something, for example a strategy to get from a hotel to the airport (car, bus, walk, etc)

- `Identify an algorithm that will be used (travelToAirport)`
- `Specify the signature (number of travelers, method, returns a time and cost)`
- `Bury the alternative implementation details in derived classes (airportByCar, airportByFoot, etc)`
- `Clients couple themselves to the interface`

MVC Pattern

A complex strategy for building a solution that stands for Model-View-Controller

- `Identify the data you will be working with (model)`
- `Identify the manipulations you will need to do to the data`
- `Identify the signatures for the manipulations (controller)`
- `Identify the ways to display the results and gather input (view)`

MVC Pattern

A complex strategy for building a solution that stands for Model-View-Controller

- **`Identify the data you will be working with (model)`**
- **`Identify the manipulations you will need to do to the data`**
- **`Identify the signatures for the manipulations (controller)`**
- **`Identify the ways to display the results and gather input (view)`**

Factory Pattern

A Factory provides a way to build a family of objects (Auto Factory could build car, bus, truck, etc.)

- **Map out the “platforms” and “products”**
- **Describe a factory interface that has a method for each product**
- **Define a factory derived class that encapsulates all new references**
- **Remove ability to call new and leave only the factory as an option**

Iterator Pattern

Common pattern for traversing a collection

- `Provide an interface to create/retrieve and iterator for the collection`
- `Design an iterator class that encapsulates traversal`
- `Add common iterator methods (first, last, next, is_done, cur_item, etc)`

FlyWeight Pattern

For large systems that can reduce memory requirements

- **Ensure object overhead needs attention and the client class can handle the restructure**
- **Divide the target class into shareable and non-shareable states**
- **Create a factory that can cache and reuse class instances**
- **The client or a third party must lookup or compute the non shareable data**
- **Add ability to add the non shareable state and supply it to the shareable methods**

Adapter Pattern

Converts one interface to another. This is a software version of physical adapters you have used

- `Identify the players to be accommodated`
- `Identify the interface that is required`
- `Design a wrapper class to match the interface`
- `The wrapper has an instance of the adaptee`
- `The wrapper maps the client interface to the adaptee`
- `The client uses (coupled to) the new interface`

Singleton Pattern

Insure a class has only one instance at a time and is globally accessible

- Define a private static attribute in the singleton
- Define a public static accessor
- Do lazy initialization in the accessor
- Define all constructors to be protected or private
- Clients may only use the accessor

Builder Pattern

Rather than build an instance in one step a builder does it a step at a time

- Decide if a common input and many outputs is the problem at hand
- Encapsulate the parsing of the input in a reader
- Define a builder for each of the steps
- Client creates a reader and a builder and registers the builder with the reader
- Client asks the reader to construct
- Client asks the builder to return the result

Command Pattern

Encapsulate a request as an object to provide enhanced features like logging, rollback, queue, etc

- **`Define a command interface with a method like execute()`**
- **`Create one or more subclasses to encapsulate a receiver, method to invoke, and arguments`**
- **`Instantiate a command for each request`**
- **`Pass the command object from the sender to the receiver`**
- **`Receiver decides when to execute`**

Interpreter Pattern

Given a language, define a representation for its language

- **Decide if a “little language” is worth the effort**
- **Define a grammar for the language**
- **Map each production in the grammar to a class**
- **Organize the classes into a composite**
- **Define an interpret(context) method in the hierarchy**
- **The context encapsulates the current state as input is parsed and output is accumulated**

There Are More

- **These are some common patterns, but many more have been defined**
- **A search for "software design patterns" can turn up several sites with varying details about the patterns they cover**
- **Note that all of these are language agnostic so you might focus on designs within a specific language and how to implement them.**

Thanks!

Send any questions, comments, or requests for assistance to info@develpreneur.com or contact us on the site. We are available to help you build your solution at any point in the process.