Creating a Software Solution

An Object Oriented Approach

Our Story so Far

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

Next Step

We are starting the implementation phase.

That means we need to consider our approaching to code and its structure

An Object Oriented Approach

- **Most modern languages support some level of object oriented design**
- **We will embrace this to ease maintenance and provide for scalability**
- **This approach will make it easier to build out the solution in sections or modules**

Getting Started

We have a few steps to get our design started

- **Define our core data objects: This is done just as we created the core tables in our DB**
- **In a similar vein create "buckets" for core functionality**
- **Consider data that will always be required (auth token, user id, etc.)**

Our Example

- **Goals Class - Goal related data and functions**
- **Users Class - Authentication and account data**
- **Work Class - for handling work done on a goal**

Grouping Functions

Once you have the core data model, move to the process and functions

This is a great time to review the use cases and requirements

Look for steps that will be performed multiple times and that are linked to core data

Common Methods

- **CRUD (Create, retrieve, update, delete) for core data**
- **Authentication/DB Connect**
- **Aggregate/Transform data**
- **Events/Rules/Process Flows**

Non Core Methods

- **Organize data for views**
- **Notifications/Messages/Email**
- **Multi-Object Processes and Controllers including relational rules**
- **Exception handling/Errors and logging**

Our Example

- **Saving/Manipulating Data in the DB (CRUD)**
- **Listing/Reporting Data**
- **Cascading updates/Side effects**
- **Security, Logging, and Notifications/Errors**
- **Data Validation**

Legos, Not Sand

- **Functions/Methods should not be too large (>100's LOC)**
- **Avoid side effects and try to keep to single function other than transactions**
- **The goal is clean interfaces and not just small chunks of code**
- **Stack multiple interfaces where applicable to maintain a single primary function**

Inheritance Guidelines

- **Should be natural, do not force it just to support inheritance, use in moderation**
- **Consider global actions for a high level root object**
- **Parent objects should provide a single code source and not be regularly over-written**
- **Think about interfaces vs. hierarchy**

Plugins and Shared Code

- **Pay attention to recurring code chunks. These are good candidates for abstraction**
- **Centralize process flows to ease sweeping changes**
- **Keep configuration outside of code for easy deployments**
- **Limit assumptions for easier integration**
- **Make steady use of comments and documentation**

Modern Software

- **Avoid Reinventing the wheel, look for prebuilt functions and methods**
- **Batch and off loading processing may be best**
- **Allow for multiple instances unless it would somehow be prohibitive**
- **Provide a hook or API to avoid coding yourself into a corner**

Best Practices

- **Avoid complex logic and magic numbers**
- **Use meaningful names**
- **Keep loops tight**
- **Provide for exceptions and add in negative testing**
- **Validate user entered data**

Bottom Line

- **Start From Major Objects/Actors**
- **Add Details as needed (properties, configuration, etc.)**
- **Add basic functions: security, logging, error handling, etc.**
- **Follow Best Practices**

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.