Apex – What makes a good design for code customization?

Overview

Sometimes there is a need to customize Salesforce to meet one's business objectives when our first guiding principle***, Clicks, Not Code***, cannot be satisfied. This can be a dauting task if you have not gone down this path before.

This document provides some helpful hints and design qualities to help guide you down this path. If not done well, your technical debt may overwhelm. These concepts and qualities are not new but re-surfaced for edification around Salesforce Apex.

What are Common Qualities of a Good Design?

Common qualities may seem obvious; however, we often loose track when we do not plan or are faced with a time crunch.

Trade-Offs

Trade-Offs are expected as you can never have a perfect design. If you don't think you need, or have, trade-offs, chances are you did not consider, or were aware of, other factors. An adage, *bend but don't break*, states, compromise without breaking good design principles.

Borrow from the Past

Chances are what you are looking to do has either already been done, or fragments exists. The *DRY rule, Don't Repeat Yourself*, is very true in writing code, as it is in developing a design. Borrow the good aspects from those designs. Build on the shoulders of other mistakes and goodness.

Supports Evolution

A good design will evolve just as business requirements will evolve. Volatility is inevitable and understanding how to address this volatility will allow your design to evolve. Be careful, you don't have endless time and resources, and, must be mindful of **Trades-Offs**!

Works with Others

A design that does not work with other aspects (current environment, internal and external), isolated, will have ignore integration and adoption concerns.

Consistency

Providing consistency, even with a poor design, helps users and developers with adaptabilities and productivity.

Where to Start?

A design must satisfy a single objective. Understanding the overall objective, we can develop scenarios. Technologists can cull from the business, scenarios and find underlying patterns. These patterns should exhibit a **common theme**. These patterns are expressed in User Stories, or Use Cases, with both Happy and non-Happy paths.

If you find more than one common theme, it would be best to separate out. This will allow better management, design and focus.

Consider using C4 model. A C4 model allows you to move from a high-level to low-level. Understanding the **C**ontext (persona-base) , then moving into **C**ontainers , to **C**omponents and finally **C**ode (Apex or Flow).

User Story

A poorly written user story, or Use Case, will not meet your consumer's expectation and subject to a brittle design. A good User Story will have SMART or INVEST qualities. In addition, there needs to be two other qualities present, Definition of Ready [**DoR**] and Definition of Done [**DoD**]. Miss any of these aspects and your overall design will suffer.

Please note, this document does not go into detail of an Agile Process or any other process methodology. The reason these aspects come into play is because it provides a level of architectural documentation of why, what and how.

Tangible Qualities

Above we outlined common qualities of a good design; however, these are intangible qualities of software. This section covers the tangible qualities one can reference and measure.

A good design has the following characteristics:

1. **Simplicity** – easy to use, change and understand (single responsibility),
2. **Abstraction** – Behavior is defined but the implementation will vary. This concept may bleed into *Information Hiding*.
3. **Coupling** – the degree of interdependence between modules/section; low coupling is ideal.
4. **Cohesion** – Degree of functionally related elements. High Cohesion is ideal
5. **Information Hiding** – Users and Classes see ONLY what they need.
6. **Performance** – Provides know performance metrics; supports Service Level Agreements (SLAs)
7. **Security** – Only authorized, (perhaps, *authenticated*) users can run the service/functions.

Simplicity

Simplicity can be subjective but one can make it objective by expressing the design with a single responsibility (adapted from SOLID principles). When more than one responsibility exists it no longer becomes simple to understand; it is complicated. A common theme, or vision, allows one to focus on the problem. This does not mean a problem does not exhibit the need to include other features, it just means, it will only focus on one-and-only-one behavior.

When designing, specify, and expose, the interface/abstraction of your responsibility. This allows one *what* your responsibility looks like without defining **how**. In addition, it will allow you to satisfy your responsibility without constraining your design.

Abstraction

Abstraction is should friend. Abstraction allows you to express your understanding without committing coupling to the underlying details. Once one exposes the details, your dependencies become firm and rigid, making it difficult to extend.

For example, if a design needs to log information, my abstraction is ***log***; nothing more. Here you have not defined the details not committed to how that is done. Logging may depend on other aspects, such as, the user, environment, location, external systems, etc. This information becomes an extension of the behavior not a concrete part. You can vary behavior you cannot vary concreteness.

Coupling

Coupling refers to how related, or dependent, components are toward each other. When a major change occurs, low coupling, will have little to no impact on other components. The astute reader will recognize that when a component depends on an abstraction (the behavior), the underlying implementation should not cause an impact.

Measuring coupling can be found in the book by Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices. The metrics of *efferent* and *afferent* coupling will provide one with how likely a defect in one component will ripple across your design.

Cohesion

Cohesion refers to how well your component is focused, i.e., Single Responsibility. High cohesion represents related items where things that live together, stay together, and change together.

Information Hiding

Information Hiding refers to what is, and is not, made available to a consumer. The more information exposed can result in a design that is rigid and fragile. The more exposed information, the more likely change will be difficult.

If you must, or need to, expose information consider using primitive types or creating a pure Data Transfer Object (DTO) or Business Object; which does nothing but holds data (no defined behavior). The latter provides a level of encapsulation that can adapt to change. For example, instead of exposing an Account object to a consumer, if all that is needed is a Mailing Address, consider an Address DTO object, which can then support the Concept of Customer, Employer, Company Address. Which also supports the use in a view-model.

In addition, consider exposing CRUD Methods via Repository or Selector Pattern.

Performance

Performance of your design is critical but often neglected. Let's go back to the logging example. Performance is important and should factor in when implementations are constructed. There may be different metrics to information that are written to a custom object as oppose to published as an event. The performance metrics will weigh heavy with Bulkification and contribution to regular use.

In the logging design, you should be able to have metrics for all implementation, including No-Op. No-Op supports the logging capabilities but does not log. This would allow you measure the cost of using versus, the other logging capabilities.

Security

Security is another aspect left out; or becomes an afterthought. In Salesforce, Apex should be adorned with the proper sharing modes *(inherited sharing, with sharing or without sharing*). If Apex is not adorned with the proper share mode, there may be unintended consequences.

A preferred annotation, is inherited sharing, Why?

*Using inherited sharing enables you to pass AppExchange Security Review and ensure that your privileged Apex code is not used in unexpected or insecure ways. An Apex class with inherited sharing runs as **with sharing** when used as:*

- *An Aura component controller*
- *A Visualforce controller*
- *An Apex REST service*
- *Any other entry point to an Apex transaction*

Because using *inherited sharing* allows the class to adapt to the caller's intent, you can use the class without having to change the class or using a wrapper class.

When designing your software consider the use of a CRUD Matrix. CRUD Matrix will help illustrate your CRUD access on the Object as well as the User/Role CRUD operations on said Object.

Tips for a Good Design

Below is a list of Tips for a good Apex Design.

Tips

Tip	Comment
Avoid the use of new**, except for primitives.**	Using the **new** operator **binds** you to that Implementation. This makes it difficult to change behavior. Instead, use Factor or Builder Design Patterns. See **OCP**. **If this provides challenges, consider encapsulating the functionality inside a class, i.e., *newInstance*();**
Use Abstraction to handle changes; or what is considered volatile.	Abstraction defines **what** you are doing, not **how**. Many of the SOLID principles outline the benefit of Abstraction.
Avoid the use of static **methods**.	Static methods cannot vary behavior. Static methods are not bad, but when used to define behavior, because you cannot vary or extend, developers then revert to modifying the component. The side effects are: • a loss of Single Responsibility, • unnecessary bloat, • fragility where a change ripple throughout, or • rigidity which breaks dependent components.
Do use Data Transfer Object (DTO) or Business Object to separate state from behavior.	Providing DTO between reflects the lowest level of coupling between components without coupling to an entire data structure; this can be synonymous with View Model; or surrogate data.
Designing a Framework, consider Separation of Concerns (SOC). Simple layering provides focused responsibility.	Separation of Concerns (SOC) supports High Cohesion. For example, one may have log configuration data that lives in Custom Setting or Custom Metadata. These two aspects of configuration have no place explicitly in a

Tip	Comment
	logger component, i.e. *Logger*. Instead, these two configurations should be encapsulated into two separate classes that inherit from an abstract or interface. Why? One can then inject the user configuration, i.e. *IUserConfiguration*, into the corresponding Logger (via constructor or method).
Low-Level methods/functions that have a simple/single responsibility you can consider the use of static method.	As mentioned, **static** methods are not bad. Encapsulating a string manipulation function, i.e. remove ALL spaces, in a static method, ***removeAllSpaces***(String), is fine. Consider encapsulating these String functionalities inside a class (StringUtility).
Use Interface Segregation (IS) for Bloated APIs	Brownfield projects may have unneeded bloat. This bloat exposes your design to a fragile environment. Use **IS** to define the functionality you need, hiding unneeded information.
Design with Testability in mind.	Classes that are either to test; generally, are design well as they are easily adaptable to Mocks. If an item is difficult to test, it is probably difficult to use.
Public/Global Methods with arguments, perform data validation (pre-check or post-check).	Public/Global methods with arguments will validate incoming data, or consider a composition, i.e. Validator/Guard, to perform such a task. These methods then delegate the work to protected, private methods or composite, to perform the operation. Consider public/global methods the gatekeeper in providing a stable state.
Test Driven-Design (TDD)	Sometimes we struggle on a design. Starting with a Test-Driven approach may help clear the roadblock. This iterative process helps one see how to use and consume. Remember, if it is had to use, it will be hard to test!
Design with extension in mind.	This follows along the OCP Principle, o*pen for extension, closed for modification.* This may be one of the most important concepts, and least understood principle. When you design with extension in mind, consider the benefits: • You test ONLY the code you extend, • Behavior is the same, only the implementation changed, • You did not modify already tested and running code! • It is easy to inject as it is to remove.

References

Software Engineering | Coupling and Cohesion
SOLID
Object Orientation Tips
Design Principles and Design Patterns
Separation of Concerns
C4 Model