# Apex Enterprise Patterns

https://www.apexhours.com/apex-enterprise-patterns/

https://github.com/apex-enterprise-patterns

* Enterprise software needs to change over time.  It evolves.
* It’s one thing to write code to satisfy the immediate need.  It’s another thing to write code that will be easy to maintain and be able to be changed over time.
* Complex code will become unmaintainable if proper boundaries are not established

**Separation of Concerns SoC**  
Разделение ответсвенности

Принцип разделения ответственности (SoC) имеет решающее значение в разработке программного обеспечения, включая разработку Salesforce Apex. Он относится к практике организации кода таким образом, чтобы разделять различные аспекты приложения, такие как пользовательский интерфейс, бизнес-логика и уровни доступа к данным. Такой подход помогает управлять сложностью, повышать удобство обслуживания и улучшать масштабируемость. В контексте разработки Salesforce Apex SoC обеспечивает многочисленные **преимущества.**

**1. Enhanced Maintainability**  
Разделяя проблемы, разработчики могут изолировать части приложения, которые изменяются по разным причинам. Такая изоляция упрощает понимание и изменение кода.

**2. Improved Scalability**  
SoC позволяет различным частям приложения масштабироваться независимо. Например, если уровень доступа к данным находится под большой нагрузкой, его можно оптимизировать или масштабировать, не влияя на остальную часть приложения.   
**3. Easier Debugging and Testing**

Разделение проблем упрощает тестирование и отладку, поскольку каждая часть может быть протестирована независимо. Это приводит к более быстрому и эффективному выявлению и решению проблем.  
**4. Facilitates Reusability**

Разделяя проблемы, разработчики могут создавать повторно используемые компоненты или сервисы. Это уменьшает дублирование кода и способствует подходу DRY (Don't Repeat Yourself) к разработке.

**5. Promotes Collaboration**

Когда задачи разделены, нескольким разработчикам или группам проще работать над разными аспектами приложения одновременно, не вызывая конфликтов.

https://trailhead.salesforce.com/content/learn/modules/apex\_patterns\_sl/apex\_patterns\_sl\_soc

SOC Layers  
Presentation

Declarative: Layouts, Record Pages, Flow, Record Types, Formulas, Reports, Dashboards

Coding: Apex Controllers, Visualforce, Lightning Components

Business Logic Layer

Declarative: Formula, Flow, Validation Rules, Sharing Rules, Approval Processes

Coding: Apex Services, Apex Custom Actions, Asynchronous Apex

Data Access Layer

Declarative: Data Loaders, Salesforce Connect

Coding: SOQL, SOSL, Salesforce APIs

Database Layer

Declarative: Custom Objects, Fields, Relationships, Rollups

Coding: Apex Triggers

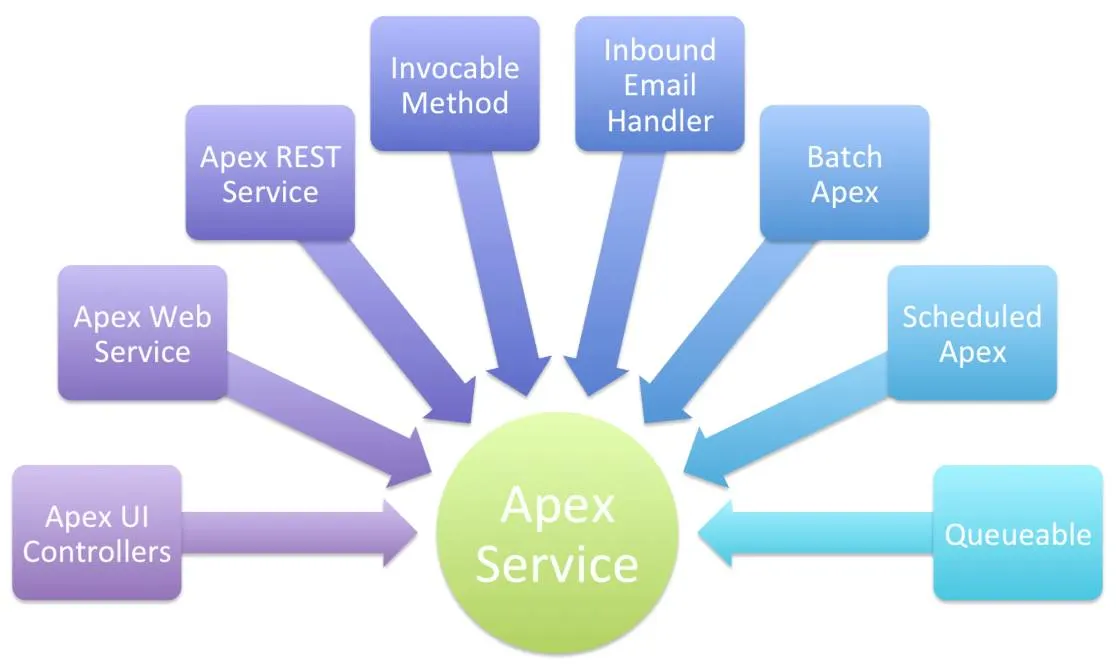
https://github.com/Coding-With-The-Force/Salesforce-Separation-Of-Concerns-And-The-Apex-Common-Library/tree/main

**Service Layer**  
***https://trailhead.salesforce.com/content/learn/modules/apex\_patterns\_sl/apex\_patterns\_sl\_learn\_sl\_principles***

Уровень сервиса : *«Определяет границу приложения с уровнем сервисов, который устанавливает набор доступных операций и координирует реакцию приложения на каждую операцию»*. **Мартин Фаулер**

Уровень Service помогает вам сформировать четкую и строгую инкапсуляцию кода, реализующего бизнес-задачи, вычисления и процессы. Важно убедиться, что уровень Service готов к использованию в различных контекстах, таких как мобильные приложения, формы пользовательского интерфейса, насыщенные веб-интерфейсы и многочисленные API. Он должен оставаться чистым и абстрактным, чтобы выдерживать меняющиеся времена и требования, которые его ожидают

The consumer of a service layer is called the "client."



**Note**

*Apex triggers are missing because the logic belongs to your application’s Domain layer, which is closely aligned with the objects and thus manipulation of the records in your application. Domain logic is called both directly and indirectly within the Service layer and, of course, via the platform UI and APIs.*

Design Considerations

**Naming conventions** The Service layer must be abstract enough to be meaningful to a number of clients. *(абстрактное название классов и методов что бы можно было переиспользовать другими клиентами— нет персонализации)*  
**Platform / Caller sympathy** - Design method signatures that support the platform’s best practices, especially bulkification. *Рассмотрите службы, которые можно вызывать со списками, а не с отдельными наборами параметров*   
**SOC considerations** - Service layer code encapsulates task or process logic typically utilizing multiple objects in your application. *Подумайте об этом как о оркестраторе. Напротив, код, относящийся конкретно к проверке, значениям полей или вычислениям, которые происходят во время вставки, обновления и удаления записей, является заботой связанного объекта. Domain для этого типа кода.*

****Security**** Service layer code and the code it calls should by default run with user security applied.   
*Чтобы гарантировать, что это так, используйте with sharingмодификатор в классах сервиса Apex (особенно важно, если вы раскрываете такой код через глобальный модификатор). Хорошим подходом является использование закрытого внутреннего класса Apex с without sharing применением модификатора.*

**Marshalling** Avoid prescribing how aspects of interacting with the service layer are handled because certain aspects are better left to the callers of your service, for example, semantics like error handling and messaging.*Например, Visualforce использует <apex:pagemessages>, а задания Schedule, скорее всего, будут использовать электронные письма, сообщения Chatter или журналы для сообщения об ошибках. Поэтому в этом случае обычно лучше всего использовать семантику обработки ошибок по умолчанию Apex, выдавая исключения.*

**Compound services** Although clients can execute multiple service calls one after another, doing so can be inefficient and cause database transactional issues. It’s better to create compound services that internally group multiple service calls together in one service call It is also important to ensure that the service layer is as optimized as possible in respect to SOQL and DML usage *Это не означает, что нельзя предоставить более детализированные службы; это просто означает, что вы должны предоставить вызывающим возможность использовать более конкретную отдельную службу при необходимости.*

**Transaction management and statelessness Clients of the service layer often have different requirements regarding the longevity of the process being undertaken and the information being managed. *Например, один запрос к серверу и несколько запросов, разделенных на separate scopes : состояние управления (например, Batch Apex) или сложный пользовательский интерфейс, который поддерживает свое собственное состояние страницы в нескольких запросах. Учитывая эти вариации управления состоянием,* ***лучше всего инкапсулировать операции базы данных и состояние сервиса в вызове метода к уровню сервиса.******Другими словами, сделайте сервис без состояния , чтобы предоставить вызывающим контекстам гибкость для использования собственных решений по управлению состоянием.*** *The scope of a transaction с базой данных также должна содержаться в каждом методе сервиса, чтобы вызывающему не приходилось учитывать это со своим собственным* ***SavePoints****, например.***

**Configuration** You might have common configuration or behavioral overrides in a service layer, such as providing control to allow the client to instruct the server layer not to commit changes or send emails. *Этот сценарий может быть полезен в случаях, когда клиент реализует функциональность предварительного просмотра или типа «что если». Обязательно продумайте, как вы реализуете это последовательно, возможно, как перегрузку метода, которая принимает общий параметр Options, аналогично методам DML в Apex.*

**Unit of Work principle**

<https://trailhead.salesforce.com/content/learn/modules/apex_patterns_sl/apex_patterns_sl_learn_uow_principles>

[Martin Fowler](http://martinfowler.com/): "Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems."

On the Salesforce platform this translates to the pattern handling the following use cases:

- Recording record updates, inserts, and deletes to implement a specific business requirement

- Recording record relationships to make inserting child or related records easier with less coding

- When asked to write (or commit) to the database, bulkifies all records captured

- Wrapping DML performed in SavePoint, freeing the developer from implementing this each time for every service method that is written

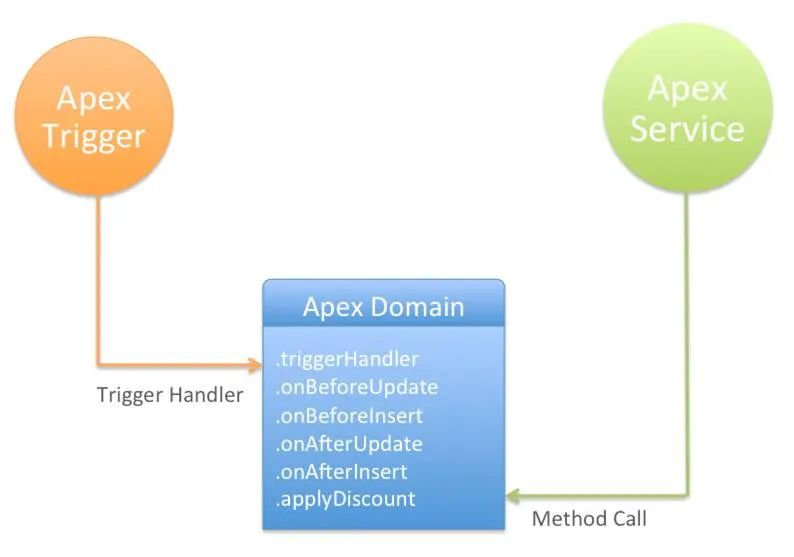
**Domain Layer**  
***https://trailhead.salesforce.com/content/learn/modules/apex\_patterns\_dsl/apex\_patterns\_dsl\_learn\_dl\_principles***

With Salesforce Platform Enterprise Patterns, the domain layer is defined through the custom objects you create (project, invoice, and so on), allowing you to rapidly build the data storage of your application.

**Domain Model** - “An object model of the domain that incorporates both behavior and data." “At its worst business logic can be very complex. Rules and logic describe many different cases and slants of behavior, and it's this complexity that objects were designed to work with.” —Fowler, Martin

used   
 **Apex triggers:** Create, read, update, and delete (CRUD) operations, including undelete, occur on your custom objects as users or tools interact via the standard Salesforce UIs or one of the platform’s APIs. The operations are routed to the appropriate Domain class code corresponding to that object and operation.

**Apex services:** Service layer code should be easy to identify and make it easy to reuse code relating to one or more of the objects that each of its operations interacts with via Domain classes. This approach keeps the code in the service layer focused on orchestrating whatever ***business process or task it is exposing***.



**Design Considerations**

**Bulkification** – ensure that domain class constructors, parametrs, methods and props deal with data in terms of list and no singular  
**Extension by containment** – dom logic encapsulate both data and behavior (мы не можем наследовать Sobject класс но можем его обернуть или использовать внутри класса который реализет методы)

**Naming conventions** – because of bulkification class name must reflect this … *public class Invoices*

**OOP** – use inheritation or interfaces to abstracr common behaviour aspects  
**Security** – use with sharing, without sharing and inherited sharing keyword on Domain Apex class  
**Separation of Concerns** – be consisten regarding where common types of domain logic, such as validation, defaulting, and calculations go. While it is possible to mix this in with other Domain logic solely related to database and trigger events, you have the option to split this logic across different methoda in the Domain class. Consider if it is useful to expose this code to make it accessible in contexts not driven by database operations, such as services that provide validation or defaulting-only behavior

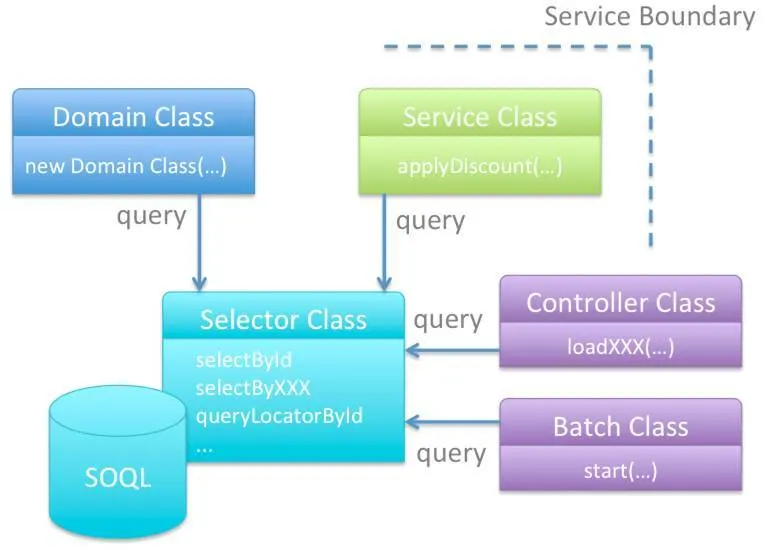
|  |  |  |
| --- | --- | --- |
| Type of Application Concern | Service or Domain | Example |
| Making sure that fields are validated and defaulted consistently as record data is manipulated. | Domain | Apply default discount policy to products as they are added. |
| Responding to a user or system action that involves pulling together multiple pieces of information or updating multiple objects. Mostly provides actions that can occur to a collection of records and coordinates everything that is needed to complete that action (potentially with other supporting services). | Service | Create and calculate invoices from work orders. Might pull in price book information. |
| Handling changes to records that occur in the application as part of other related records changing or through the execution of a user or system action. For example, defaulting values as required. If changing one field affects another, it’s also updated. | Domain | How the Account object reacts when an Opportunity is created or how the Discount is applied when the Opportunity discount process is run. Note: This sort of logic might start in the Service layer, but be better served in the Domain layer to manage the size and complexity of the service method size or improve reuse. |
| Handling of common behavior that applies across a number of different objects. | Domain | Calculate price on the opportunity product or work order product lines. Note: You could place this in a shared Domain base class, overriding the fflib\_SObjectDomain method to hook into the Apex trigger events, with concrete Domain classes extending this class in turn with their behaviors. |

**Selector Layer**

*A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself." - Fowler, Martin*

Selector Pattern, a layer of code that encapsulates logic responsible for querying information from standard objects and your custom objects

The selector layer feeds that data into your Domain layer and Service layer code. You can also reuse selector classes from other areas that require querying, such as Batch Apex and controllers.



Why need  
**Query inconsistencies:** The same queries being made from different places for the same information or criteria (or subtle variants) can lead to inconsistencies in your application.

**Query data inconsistencies:** When queried record data (essentially sObjects) starts to flow around your logic, up and down and side to side around the code in the org, calling methods receiving the records can become fragile.   
**Security inconsistencies:** Salesforce requires that all Apex code adheres to the object security of the running user.

In terms of Separation of Concerns, a Selector concerns itself with providing the following:   
 **Visibility, reusability, and maintainability** - The selector makes it easy to find and maintain database query logic.

**Predictability of queried data** - It must be clear what the selector is doing in the method name and what it’s returning

**Security**- Provide a means for the caller to opt in or opt out (for system level scenarios) of security checks that enforce sharing and permissions applied to the current user context

**Platform sympathy** - Make queries as optimal as possible, mainly expressing criteria via sets, thus encouraging the caller to be bulkified in its code when calling the selector methods.

**Design Considerations  
Naming –** group selector functionality by object type. For example OpportunitiesSelector. (one selector class per object)  
**Methods** – can be static or instance scope. The names of the methods (typically prefixed with select) ideally indicate the information and any related child information, such as selectById() and selectByIdWithProductLines(), for example. Selector methods that return a Database.QueryLocator typically have a method name prefixed with queryLocator; queryLocatorInvoicesToProcess() for example.  
**Method paprameters** - Always take sets of any primitive data typed criteria or lists of any Apex object-based criteria that the query uses. Per the rules for service and domain class logic, bulkification is key, so make this part of the contract of a selector class as well.  
**Returning sObject lists** – should return sObject lists.  
**Returning QueryLocators** – (for batch)

**Returning custom logical set** – use wrapper to return aggregation or custom logical query result

**Security** - Avoid using the with sharing or without sharing keywords on Selector Apex classes and instead use inherited sharing to ensure that the calling context establishes this context. A developer could also use the Security.stripInaccessible() method to ensure that records are not returned with fields that the current user does not have access to.

Project ApexCheckout  
Git https://github.com/apex-enterprise-patterns/fflib-apex-common

Apex  
leshii85@curious-badger-61i0ud.com  
xiezuB6j