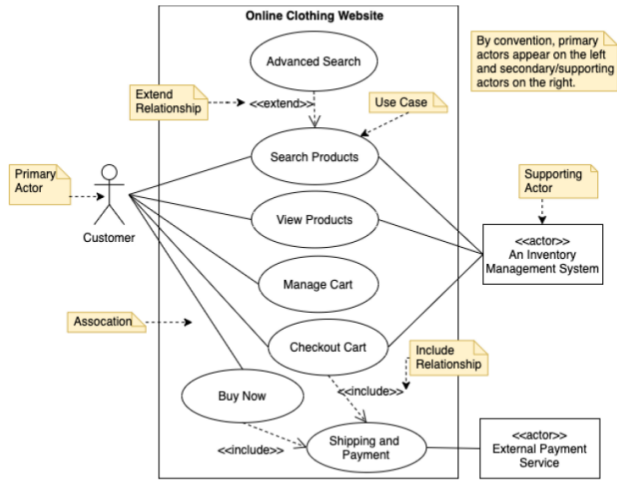


Use Cases

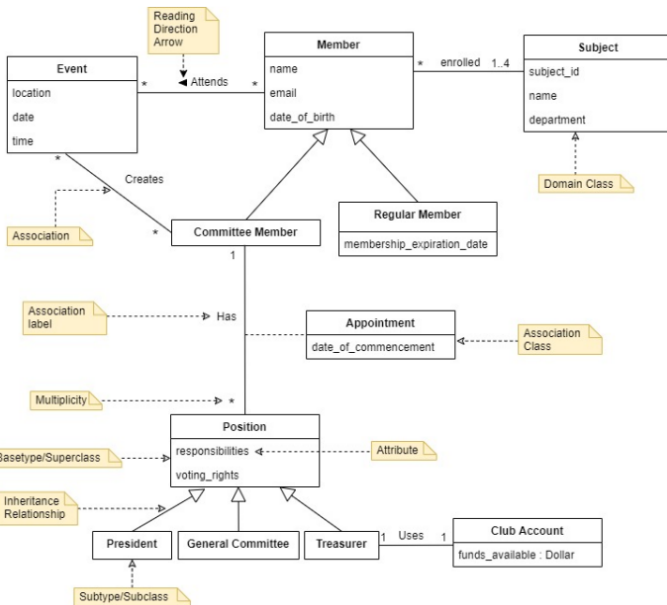
Tests

- **Boss Test:** told boss what I was doing, would they be happy? yes=use case
- **Size Test:** use case with too few steps should be part of another use case
- **Elementary Business Process Test (EBP):** task performed by one person in one place at one time, in response to a business event. Adds measurable business value and leaves data in consistent state

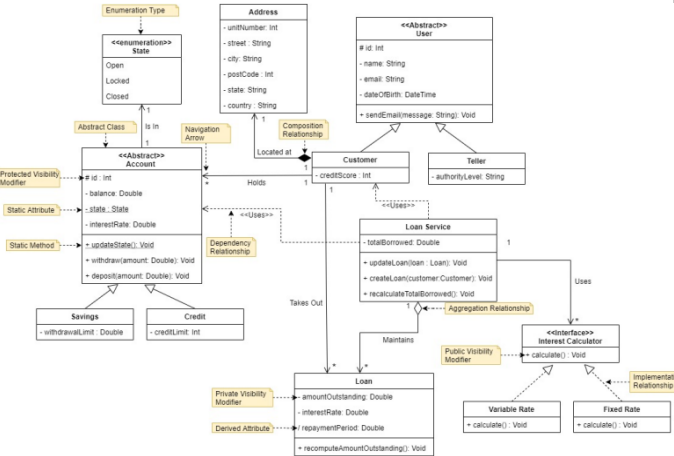


Diagrams

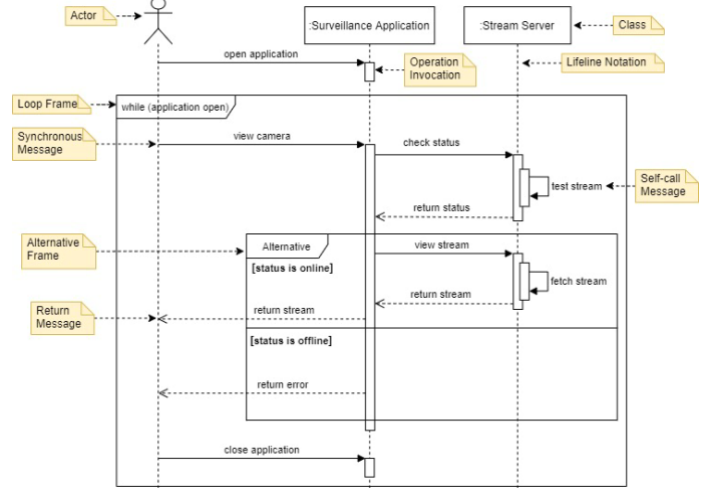
Domain Class Diagram



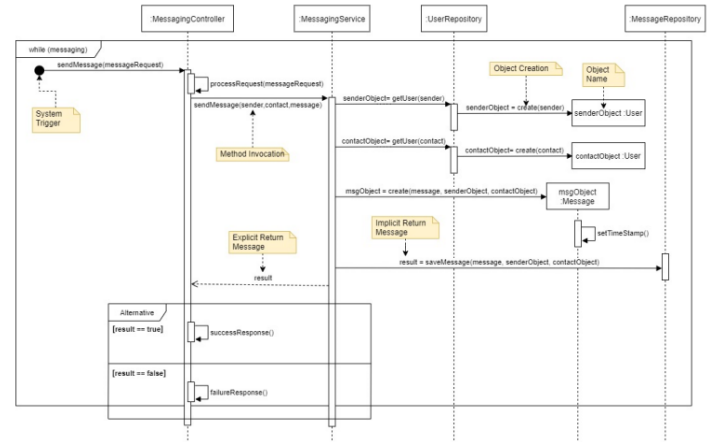
Design Class Diagram



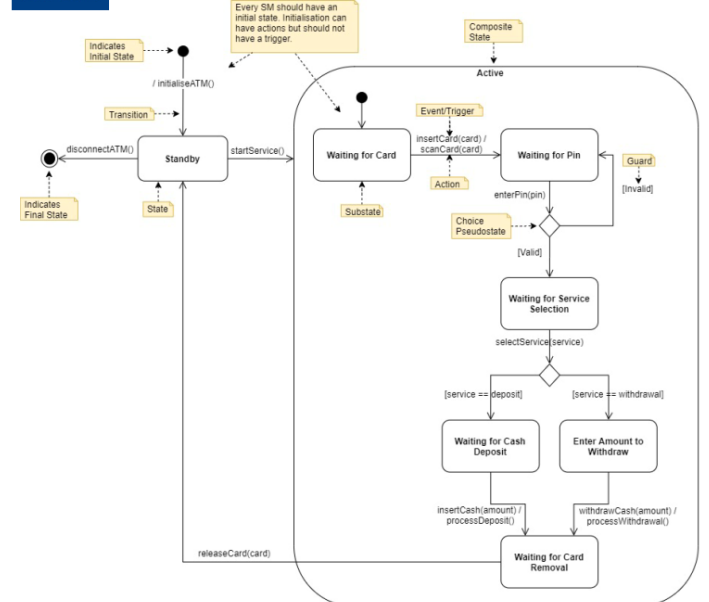
System Sequence Diagram





Design Sequence Diagram

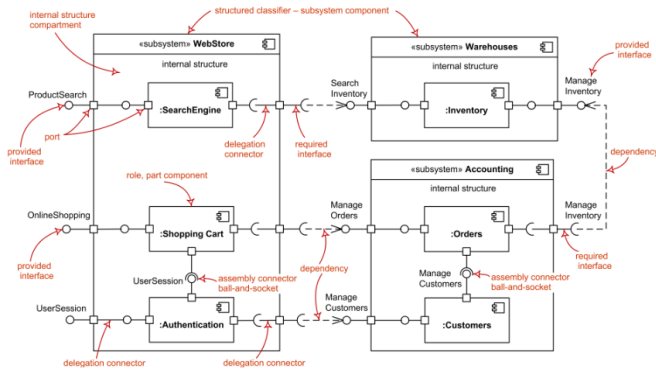


State Machine Diagram



Composition	Aggregation
has-a relationship	
strong ownership	weak ownership
- child cannot exist w/o parent	- children are independent
Car  Engine	Uni  Student

Component Diagram



GoF Design Patterns

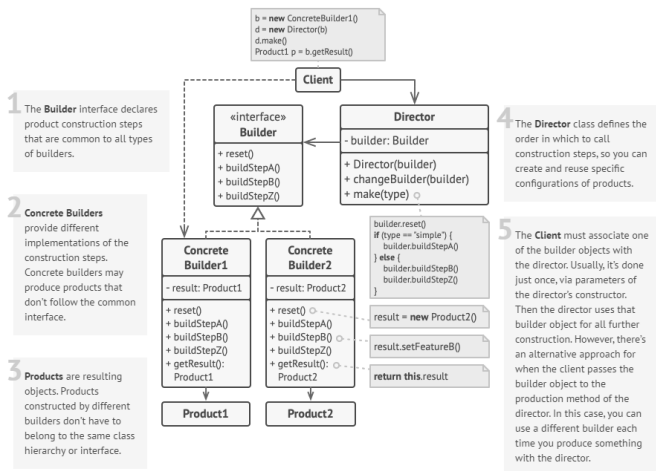
- improve code maintainability, reusability and readability; reuse established patterns; use common terminology
- Creational, Structural, Behavioural

[C] Builder

- construct complex objects step-by-step. Allows you to produce different types and representations of object using same construction code
- WHEN: get rid of "telescoping constructor"

```
class Pizza {
    Pizza(int size) { ... }
    Pizza(int size, boolean cheese) { ... }
    Pizza(int size, boolean cheese, boolean pepperoni) { ... }
    // ...
}
```

- WHEN: want code to create different representations of same product
1. Make sure that you can clearly define the common construction steps for building all available product representations. Otherwise, you won't be able to proceed with implementing the pattern.
 2. Declare these steps in the base builder interface.
 3. Create a concrete builder class for each of the product representations and implement their construction steps.
Don't forget about implementing a method for fetching the result of the construction. The reason why this method can't be declared inside the builder interface is that various builders may construct products that don't have a common interface. Therefore, you don't know what would be the return type for such a method. However, if you're dealing with products from a single hierarchy, the fetching method can be safely added to the base interface.
 4. Think about creating a director class. It may encapsulate various ways to construct a product using the same builder object.
 5. The client code creates both the builder and the director objects. Before construction starts, the client must pass a builder object to the director. Usually, the client does this only once, via parameters of the director's class constructor. The director uses the builder object in all further construction. There's an alternative approach, where the builder is passed to a specific product construction method of the director.
 6. The construction result can be obtained directly from the director only if all products follow the same interface. Otherwise, the client should fetch the result from the builder.



[C] Factory

- interface for creating objects in superclass; allows subclasses to alter type of objects created
- makes object creation cohesive; hides complexity
- WHEN: don't know beforehand the exact types and dependencies of the objects your code should work with
- WHEN: want to provide users of library/framework a way to extend its internal components
- WHEN: want to save system resources by reusing existing objects instead of rebuilding each time

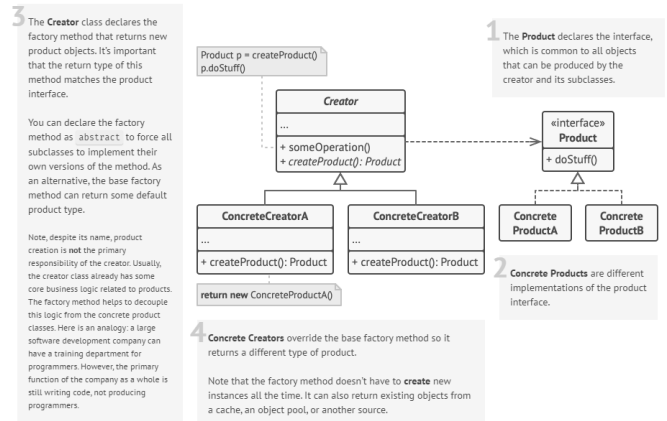
1. Make all products follow the same interface. This interface should declare methods that make sense in every product.
2. Add an empty factory method inside the creator class. The return type of the method should match the common product interface.
3. In the creator's code find all references to product constructors. One by one, replace them with calls to the factory method, while extracting the product creation code into the factory method.
You might need to add a temporary parameter to the factory method to control the type of returned product.

At this point, the code of the factory method may look pretty ugly. It may have a large switch statement that picks which product class to instantiate. But don't worry, we'll fix it soon enough.

4. Now, create a set of creator subclasses for each type of product listed in the factory method. Override the factory method in the subclasses and extract the appropriate bits of construction code from the base method.
5. If there are too many product types and it doesn't make sense to create subclasses for all of them, you can reuse the control parameter from the base class in subclasses.

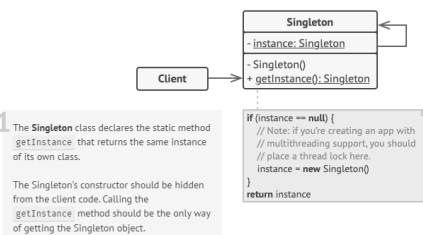
For instance, imagine that you have the following hierarchy of classes: the base Mail class with a couple of subclasses: AirMail and GroundMail; the Transport classes are Plane, Truck and Train. While the AirMail class only uses Plane objects, GroundMail may work with both Truck and Train objects. You can create a new subclass (say TrainMail) to handle both cases, but there's another option. The client code can pass an argument to the factory method of the GroundMail class to control which product it wants to receive.

6. If, after all of the extractions, the base factory method has become empty, you can make it abstract. If there's something left, you can make it a default behaviour of the method.
- specialisation of Template Method; Factory Method can serve as a step in Template Method



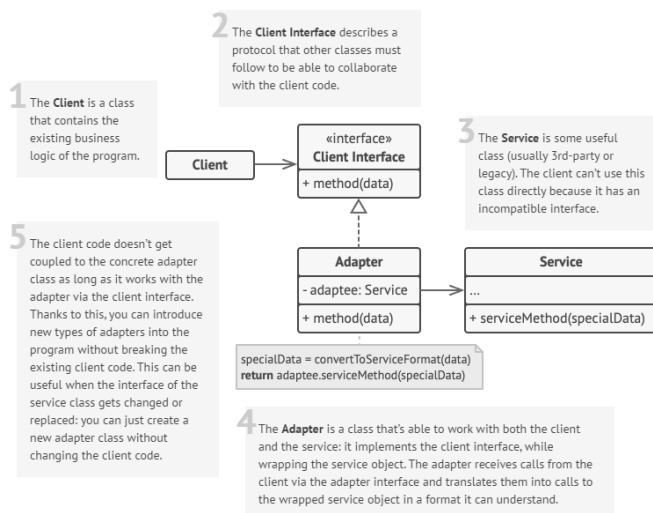
[C] Singleton

- ensure a class only has one instance; provide global point of access to it



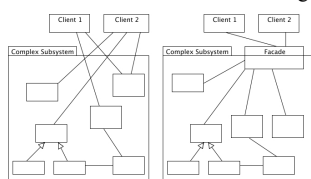
[S] Adapter

- allows objects with incompatible interfaces to collaborate
 - provide stable interface to similar (but different) components
 - WHEN: want to use some existing class/subclass, but its interface isn't compatible with rest of your code/cannot add to superclass
1. Make sure that at least 2 classes with incompatible interfaces
 2. Declare client interface and describe how clients communicate with the service
 3. Create adapter class - follow client interface; make methods empty first
 4. Add field to adapter class to store reference to service object. Initialise field via constructor or pass to adapter when calling methods
 5. Implement all methods of client interface in adapter class. Adapter should delegate most of the real work to the service object, handling only the interface or data format conversion
 6. Client should use adapter via client interface - allows change/extend adapters without affecting client code
 - Decorator enhances object without changing interface; adapter changes existing class' interface. Adapter does not have recursive composition.



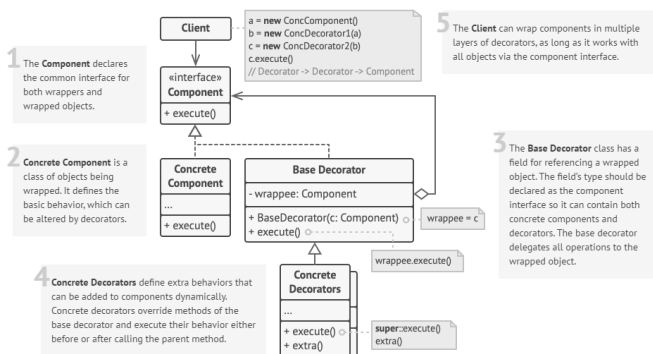
[S] Facade (Simplified Interface)

- provide higher-level interface; encapsulate/hide subsys.
- Implementation: Define new class(es) that has the required interface; have new class use the existing system



[S] Decorator

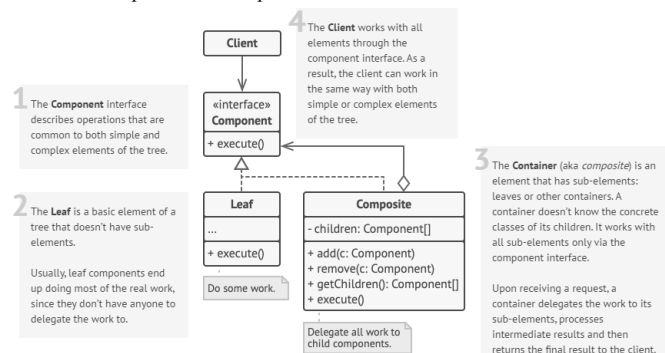
- attach new behaviours to object by placing objects inside special wrapper objects that contain the behaviours
 - recursive composition (Russian doll)
 - WHEN: need to be able to assign extra behaviours to object at runtime without breaking code that uses object
 - WHEN: awkward/not possible to extend object's behaviour using interface
1. Make sure your business domain can be represented as a primary component with multiple optional layers over it.
 2. Figure out what methods are common to both the primary component and the optional layers. Create a component interface and declare those methods there.
 3. Create a concrete component class and define the base behaviour in it.
 4. Create a base decorator class. It should have a field for storing a reference to a wrapped object. The field should be declared with the component interface type to allow linking to concrete components as well as decorators. The base decorator must delegate all work to the wrapped object.
 5. Make sure all classes implement the component interface.
 6. Create concrete decorators by extending them from the base decorator. A concrete decorator must execute its behaviour before or after the call to the parent method (which always delegates to the wrapped object).
 7. The client code must be responsible for creating decorators and composing them in the way the client needs.



[S] Composite

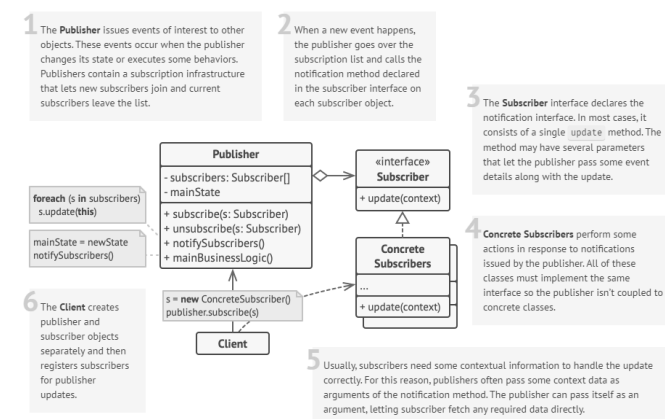
- compose objects into tree structures then work with structures as if they were individual objects
 - WHEN: tree-like object structure
 - WHEN: client code treat both simple and complex elements uniformly
1. Make sure that the core model of your app can be represented as a tree structure. Try to break it down into simple elements and containers. Remember that containers must be able to contain both simple elements and other containers.

2. Declare the component interface with a list of methods that make sense for both simple and complex components.
3. Create a leaf class to represent simple elements. A program may have multiple different leaf classes.
4. Create a container class to represent complex elements. In this class, provide an array field for storing references to sub-elements. The array must be able to store both leaves and containers, so make sure it's declared with the component interface type. While implementing the methods of the component interface, remember that a container is supposed to be delegating most of the work to sub-elements.
5. Finally, define the methods for adding and removal of child elements in the container.
 - Builder can be used to create complex Composite trees (recursive creation)
 - like Decorator, recursive composition to organise open-ended number of objects. Composite only has 1 child component.
 - Decorator adds additional responsibilities to wrapped object
 - Composite "sums up" children's results



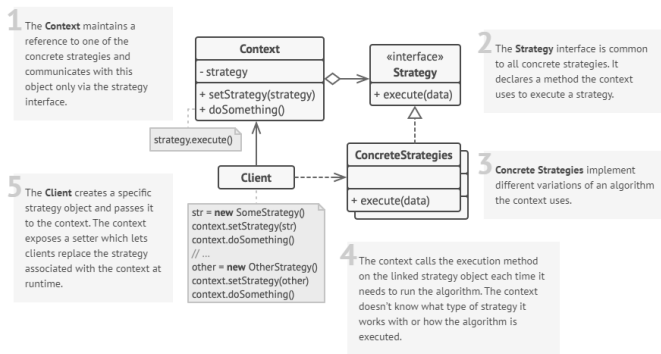
[B] Observer (Pub-Sub)

- define



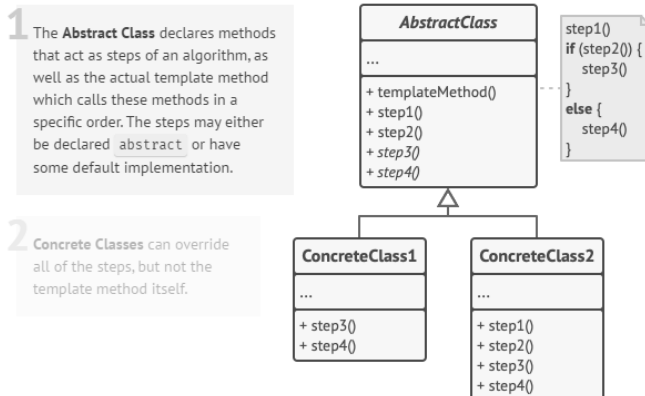
[B] Strategy

- define a family of algorithms, put each of them into a separate class, make their objects interchangeable
 - WHEN: want to use different variants of an algorithm within an object and be able to switch algorithms during runtime
 - WHEN: want to isolate business logic of class from implementation details of algorithms
1. In context class, identify algorithm that is prone to frequent changes.
 2. Declare strategy interface common to all variants of algorithm
 3. Extract all algorithms into own classes. All implement strategy interface
 4. In context class, add field storing reference to strategy object. Provide setter. Context should work with strategy object only via strategy interface. Context may define interface which lets strategy access data
 5. Clients of context must associate it with a suitable strategy that matches the way they expect the context to perform its primary job
- Decorator changes skin of object; Strategy changes the guts



[B] Template

- skeleton of algorithm in superclass but lets subclasses override specific steps without changing its structure
 - WHEN: you want to let clients extend only particular steps of algorithm, but not the whole algorithm or structure
 - WHEN: several classes contain almost identical algorithms with some minor differences
1. Try to break algorithm to steps – common vs unique steps
 2. Create abstract base class; set template method; set abstract methods representing the steps. Outline algorithm's structure in template method by executing corresponding steps. Template method should be `final` to prevent overriding.
 3. Steps can be abstract. Hooks: step with default implementation/empty that are optional (extension points).
 4. New concrete subclasses must implement all abstract steps; may override optional steps.
- Template VS Strategy
 - *Template* is based on **inheritance**: lets you alter parts of algorithm by extending those parts in subclasses; Class-level (static)
 - *Strategy* is based on **composition**: alter parts by supplying it with different strategies that correspond to that behaviour; works on Object-level (can switch behaviours at runtime)



GRASP Principles

set of patterns/principles of **assigning responsibilities** into an object; supports Responsibility-Driven Design (RDD)

Creator	Factory
B should create instances of A if: <ul style="list-style-type: none"> - B “contains” / compositely aggregates A - B records/closely uses A - B has initialising data for A 	
- Complex creation: conditional creation, reusing objects	
Information Expert	
assign responsibility to object that has information to fulfil role	
Controller	Controller
UI Layer to receive and coordinate controls for operations <ul style="list-style-type: none"> - overall system root object/major subsystem - a use case scenario deals with specific event - use case or session controller 	
- can become bloated and too complex → add more controllers/delegate using InfoExp and HighCohesion	
Polymorphism	Adapter
different classes treated as same type <ul style="list-style-type: none"> - flexible, reusable - Inheritance, Interfaces 	
Indirection	Adapter
objects should communicate through intermediary rather than directly	
reduces coupling; more flexible; easier maintain	
Pure Fabrication	

create objects (fabrications) to hold/process information/perform **specific task**

- simplify system; reduce coupling
- Utility Classes, Factory, Mock Objects

Protected Variations

Adapter

parts that are **likely to change** should be **isolated** from others

- minimise impact of change; more flexible
- abstract implementation details - Interfaces

Low Coupling

Coupling: how strongly one element relies on other elements

Low: each element is not dependent on too many other elements

High: hard to understand in isolation, hard to reuse, high impact changes (one change impacts many classes)

Assign responsibilities to avoid unnecessary coupling

High Cohesion

Cohesion: how strongly related and **focused** the **responsibilities** of an element are

Assign responsibilities; keep objects focused, understandable and manageable

Architectural Requirements

Functional	Non-functional
WHAT system does/n't do	Quality of system
Affects inputs/outputs	Doesn't affect basic functionality
Interfacing with external sys	
Basic system functionality	Product properties
User requirements	User expectations

Reliability—Recoverability of POS

Factor	Recovery from remote service (e.g., Tax Calculator) failure
Measures and quality scenarios	When remote service fails, re-establish connectivity with it within 1 min. of its detected re-availability, under normal store load in a production environment.
Variability (current flexibility and future evolution)	current flexibility - our SME says local client-side simplified services are acceptable (and desirable) until reconnection is possible. evolution - within 2 years, some retailers may be willing to pay for full local replication of remote services (such as the tax calculator). Probability? High.
Impact of factor (and its variability) on stakeholders, architecture and other factors	High impact on large-scale design. Retailers really dislike it when remote services fail, as it prevents them from using a POS to make sales.
Priority for Success	High
Difficulty or Risk	Medium

Technical Memo: Issue: Reliability—Recovery from Remote Service Failure

Factors

Robust recovery from remote service failure, e.g., tax calculator, inventory

Solution

To satisfy the quality scenarios of reconnection with the remote services ASAP, use smart Proxy objects for the services, that on each service call test for remote service reactivation, and redirect to them when possible.

Where possible, offer local implementations of remote services. For example, implementing a small cache to store data (e.g., tax rates)

Achieve protected variation with respect to location of services using an Adapter created in a ServicesFactory.

Motivation

Retailers really don't want to stop making sales! Therefore, if the NextGen POS offers this level of reliability and recovery, it will be a very attractive product, as none of our competitors provide this capability. The small product cache is motivated by very limited client-side resources. The real third-party tax calculator is not replicated on the client primarily because of the higher licensing costs, and configuration efforts (as each calculator installation requires almost weekly adjustments). This design (Adapter and ServiceFactory) also supports the evolution point of future customers willing and able to permanently replicate services such as the tax calculator to each client terminal.

Unresolved Issues

none

Alternatives Considered

A “gold level” quality of service agreement with remote credit authorization services to improve reliability. It was available, but much too expensive.