# Problem Statement

Imagine you work for a **B2B company**, and you were tasked with building their new **eCommerce website**. Design a solution that allows Customers to **purchase Products** through this website while enforcing your company’s business rules.

1. Include at the minimum:

a. All **high-level concepts/abstractions/components**/etc. that interact with one another for a full working solution.

b. The design should account for all the **below requirements** as well as enough **use cases** to exercise those requirements

c. Any **diagrams or pseudocode** that convey your design, thought process, and decisions. Actual code can be written, but certainly not necessary

2. You have been given these requirements:

a. Products can be purchased on their site if they have been **activated** and have **not** yet been **discontinued**.

b. Each Product has both a **minimum and maximum quantity** allowed per Order.

c. **Customers** have to be **verified** before they can submit Orders.

d. Once verified, the Customer will be given a **Credit Limit**.

e. An Order’s **total cannot exceed the Customer’s Approved Credit Limit** minus the remaining balances of any **unpaid invoices**.

f. All **Orders** the Customer submits will have to be **reviewed** if the sum of all the **unpaid invoices** that have become **delinquent** is **over the allowable threshold**.

# Analysis: Axes of Volatility

Architectural analysis starts with an interview (in this case, using the problem statement given) and an assessment of the axes of volatility: two orthogonal aspects of predicted changes to the system over time. This determines where boundaries should be defined so that changes can be absorbed with minimal impact to the system (i.e. fewest possible ripple effects). Further, proper decomposition ensures no deadlocks will occur.

## Changes for the same customer over time

* System users
* Payment option
* Delivery option

## Changes at the same time across customers

* Product availability
* Authentication process
* Approval process
* Billing process
* Fulfillment process

# Architecture

Following iDesign’s volatility-based decomposition, I arrived at a model that defines the system components. Provided is the component/service model, from which several related models are normally derived. Additional models (not included) would typically be created to show call chains, assemblies, services, processes, identities, authentication/authorization boundaries, transactions, and logical threads.

*See Architecture Diagrams.vsdx, Tab “Architecture”.*

# Data Design

To ensure the database has a solid and extensible design, I provided an ERD detailing the desired model. This model can be used by developers to write DDL or EF Code First logic. Provided is a complete ERD of the entities. As needed (not included), explicit relationship multiplicities, non-clustered indexes, and constraints can be defined.

*See Architecture Diagrams.vsdx, Tab “ERD”.*

# Sequence Diagrams

Developers will often want more specifics than architectural call chains show. I have found sequence diagrams the best approach to address this gap. Using activations and method names and parameters, such a diagram quickly conveys what is expected when a client initiates a call and receives a result. Provided is a sequence diagram showing a user searching for products and adding selected product quantities to an order. More sequence diagrams can be written but they quickly become redundant and hard to maintain with perfect fidelity.

*See Architecture Diagrams.vsdx, Tab “Create Order”.*

Architecture and design took approximately 2.5 hours.

# Vertical Slice

It’s my belief that an architect should be a technical leader, and therefore, providing a functional vertical slice of the application is expected. This ensures a clean framework for engineers to work from, and many examples of how logic and transactions flow through the system. In lieu of pseudocode, I went ahead and sketched out such a slice, completely coding the “Create Order” use case as well as providing a working solution to the entire set of business requirements in the problem statement.

Note, the slice provided is not production-ready. While complete for the problem statement, I kept it simple here. Critically, authentication/authorization, transactions, and concurrency are not considered. There is no means to pay invoices yet. There are no unit tests, though the solution is readily covered at nearly 100% code coverage. Additionally, my preference is that all operations should return a Result<T> instead of void, scalars and POCOs, so that workflow and error handling/messaging can be performed consistently across the solution.

*See B2BCorp.sln.*

Coding took approximately 7 hours.