
SKY
The art of Software Integration
Rafael Priego Porcuna
Online Engineering Offsite, September 2014
Moone Boy
sky 1 HD

The art of software integration

- We all have to integrate systems in our day to day basis
- Software integration is an “art” that requires knowledge, experience, intuition and creativeness

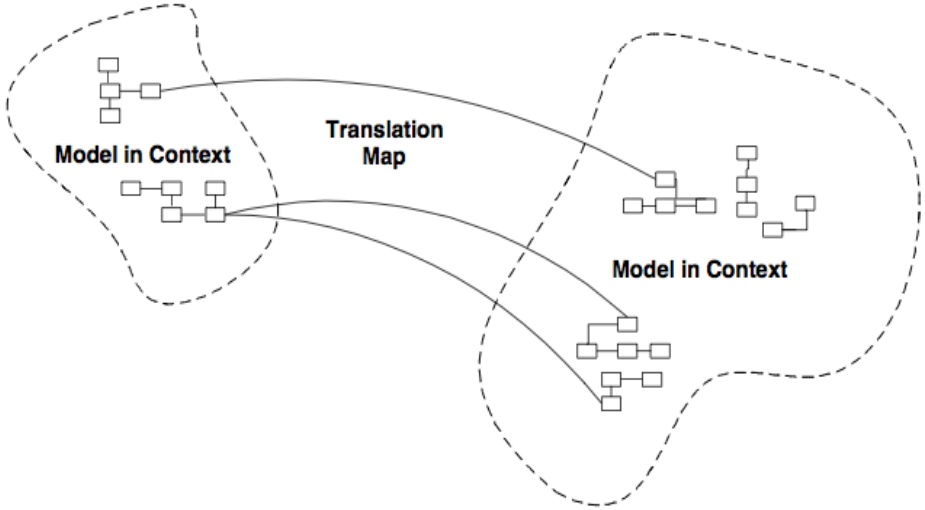
Contents

- Strategies for integration
- Integration techniques
- Implementation
- References, Q&A

Strategies for integration

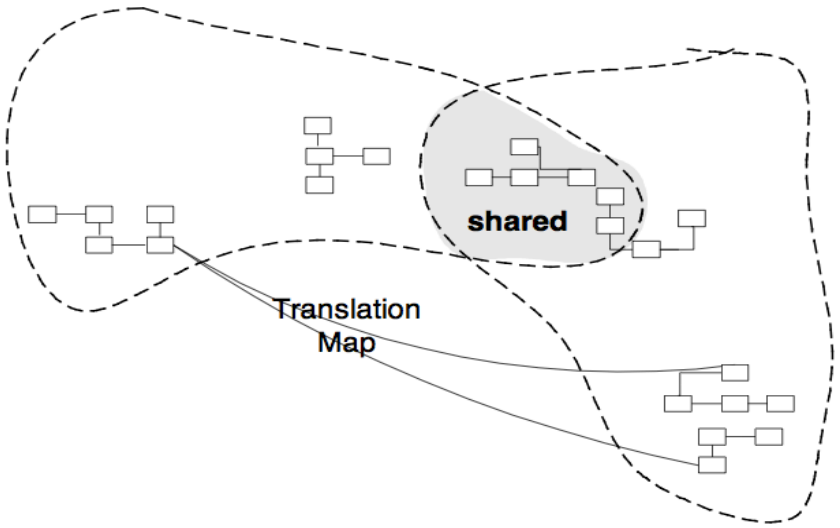
Choosing your strategy

- Cooperation between teams
- Overlap between models
- Allow independent development
- Cost of the integration



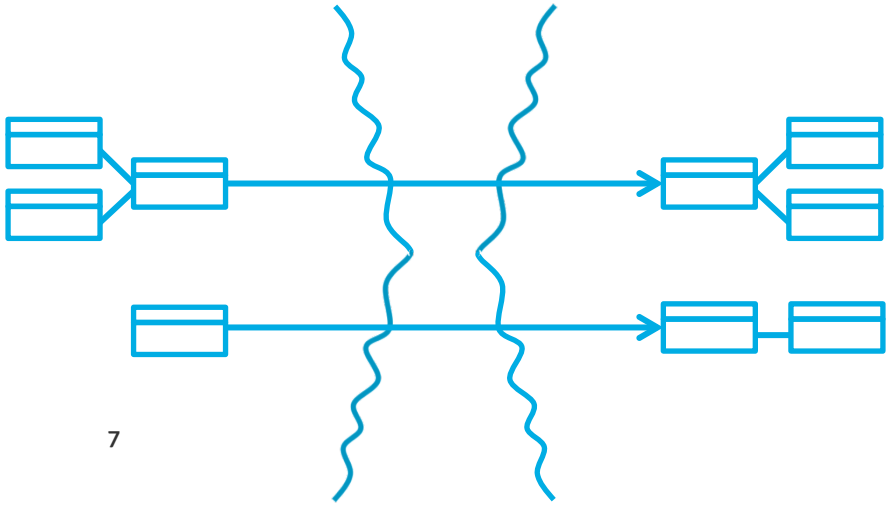
Shared Kernel

- Intent/Problem: Split one team in multiple teams, creation of different domains in the same application
- Applicability: Unidirectional and bidirectional communications, both teams report to the same management and ideally are located close to each other
- Known use: Feature teams
- Pros: Reduces overhead of Continuous Integration in big teams, shared model and domain language
- Cons: Extra coordination required



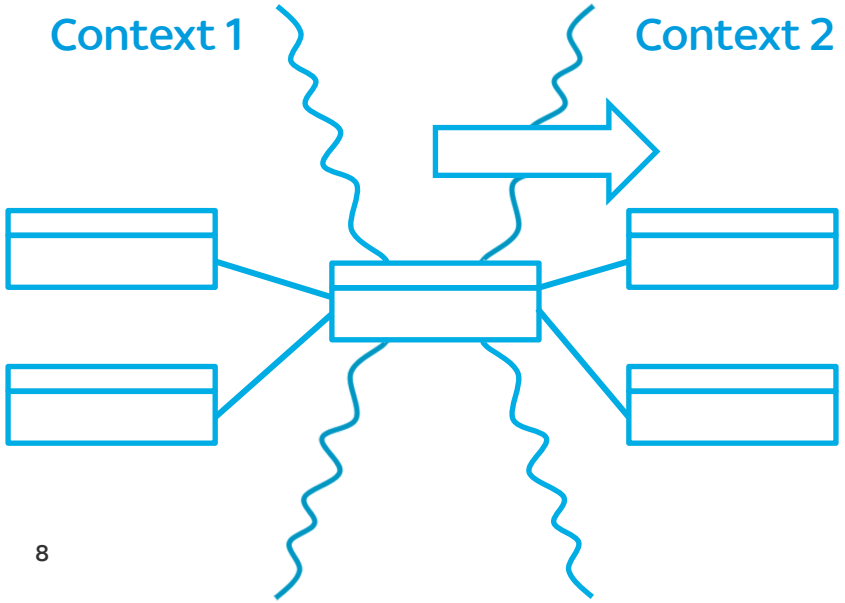
Customer/Supplier

- Intent/Problem: Handle unidirectional dependencies, when the downstream system needs data from upstream
- Applicability: Unidirectional dependencies and both systems share the same goals
- Known use: analytics systems
- Related strategies: Conformist, Anti-corruption layer, Open host
- Pros: Translation easier due to be unidirectional
- Cons: Downstream system can be limited by the upstream system, difficult to take into practice when the supplier serves several customers



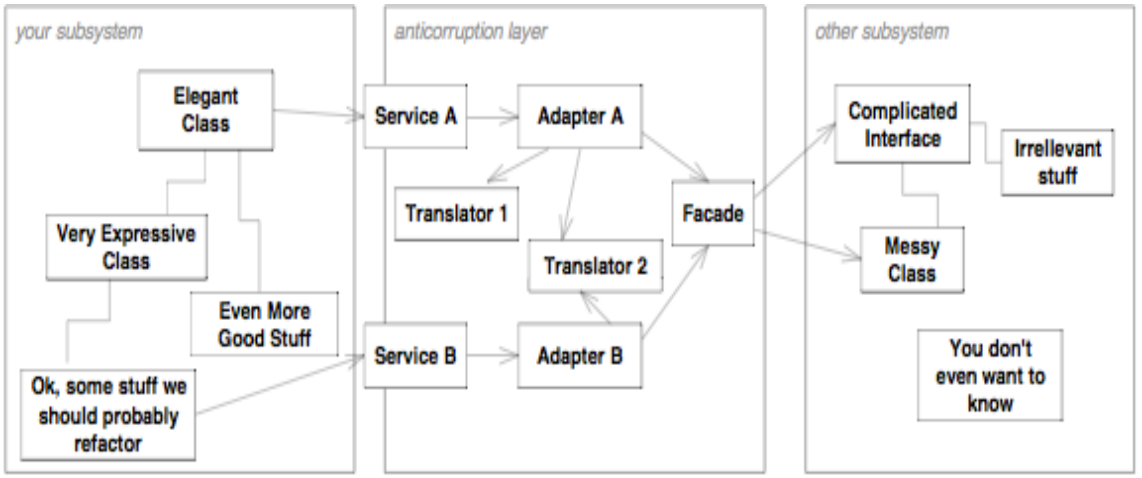
Conformist

- Intent/Problem: Difficulty to follow customer/supplier approach when they follow different directions
- Applicability: Dependencies are unidirectional and the models are not very distant
- Related strategies: Customer/Supplier, Anti-corruption layer
- Pros: Shared domain language between the teams, allow development independently
- Cons: Limits the creativity of downstream system



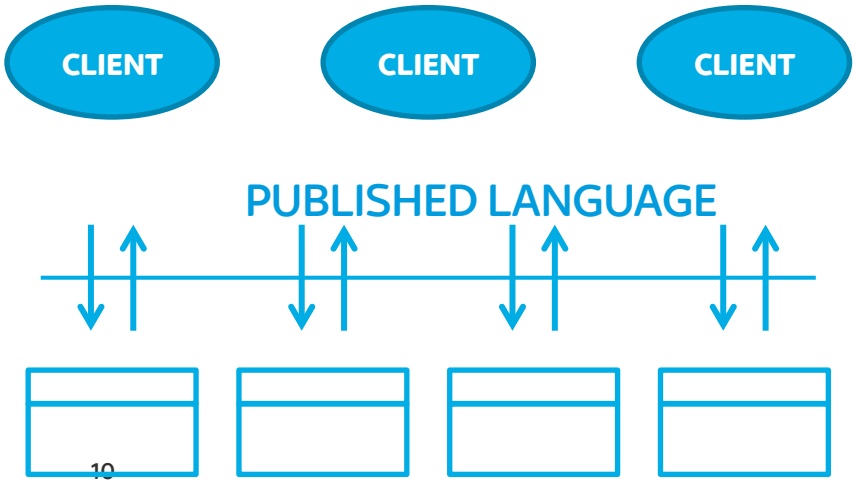
Anti-corruption layer

- Intent/Problem: Difficulty to follow customer/supplier approach when they follow different directions
- Applicability: Unidirectional and bidirectional communication. Interfaces are not really big.
- Known use: Replacing legacy systems. Protection against constant interface changes
- Related strategies: Customer/Supplier, Conformist
- Pros: Allow development independently of the other system
- Cons: Investment in translation required, changes in one model can lead to high cost in translation to the other



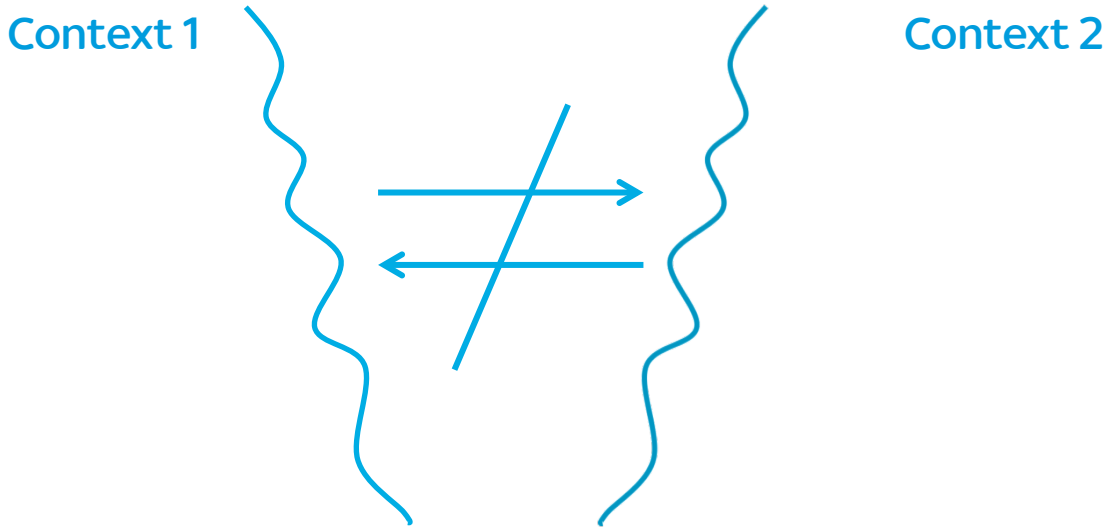
Open host services

- Intent/Problem: System that will talk to multiple clients
- Applicability: Unidirectional and bidirectional communications, multiple clients interact with a system
- Known use: Banking
- Related strategies: Customer/Supplier
- Pros: Enables a published language, interaction with multiple clients without model alterations
- Cons: Hard to design a protocol to be understood by multiple systems



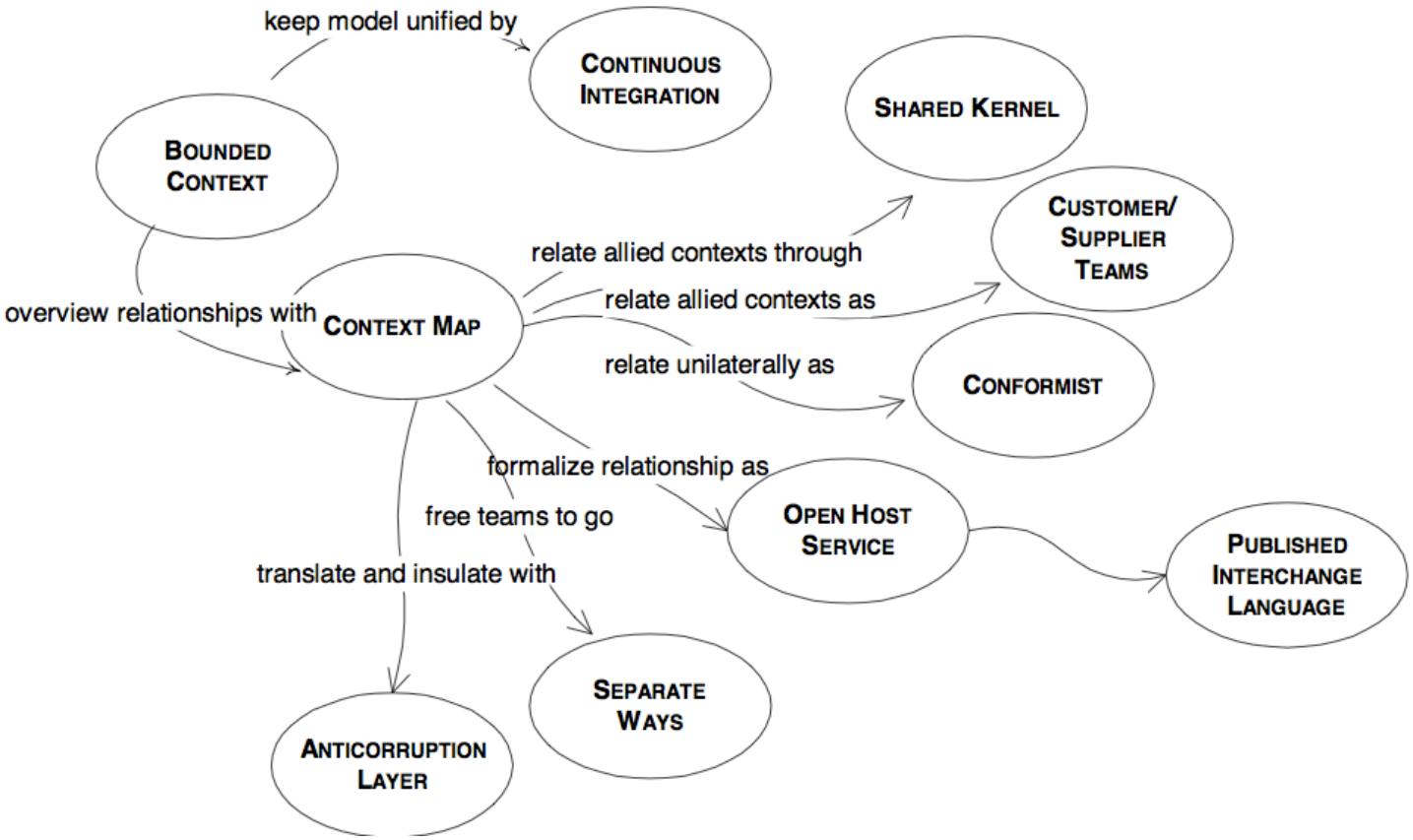
Separate ways

- Intent/Problem: Integration cost is really high and doesn't offer much value
- Applicability: Integration is not needed or there is a work around
- Known use: Links
- Pros: Allow independent development and saves the cost of integration
- Cons: If integration is needed, translation layers will be more complex



	Collaboration	Bidirectional	Shared language	Integration cost
Shared Kernel	Yes	Yes	Yes	Low
Customer / Supplier	Yes	No	Yes	Low
Conformist	No	No	Yes	Medium
Anti-corruption layer	No	Yes	No	High
Open host	No	Yes	Yes	High
Separate ways	No	No	No	None

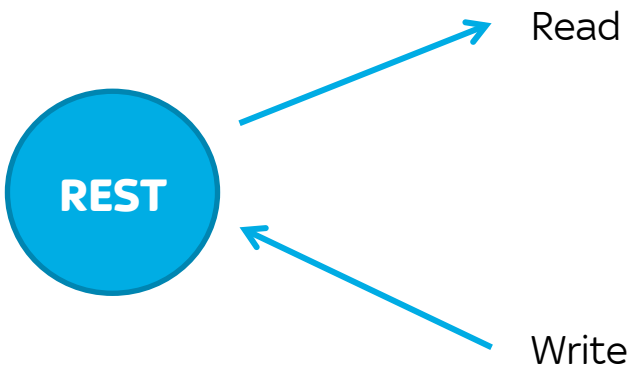
Strategies for integration



Integration techniques

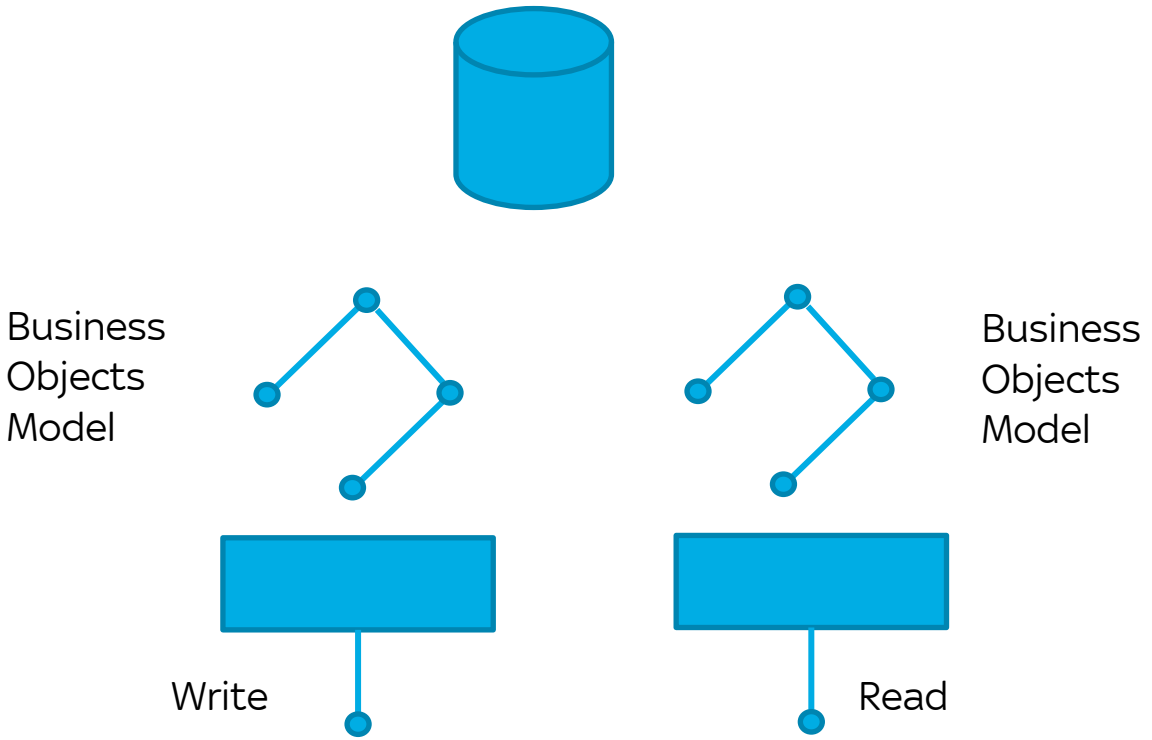
REST endpoint

- Read and write as separate services



Client-server

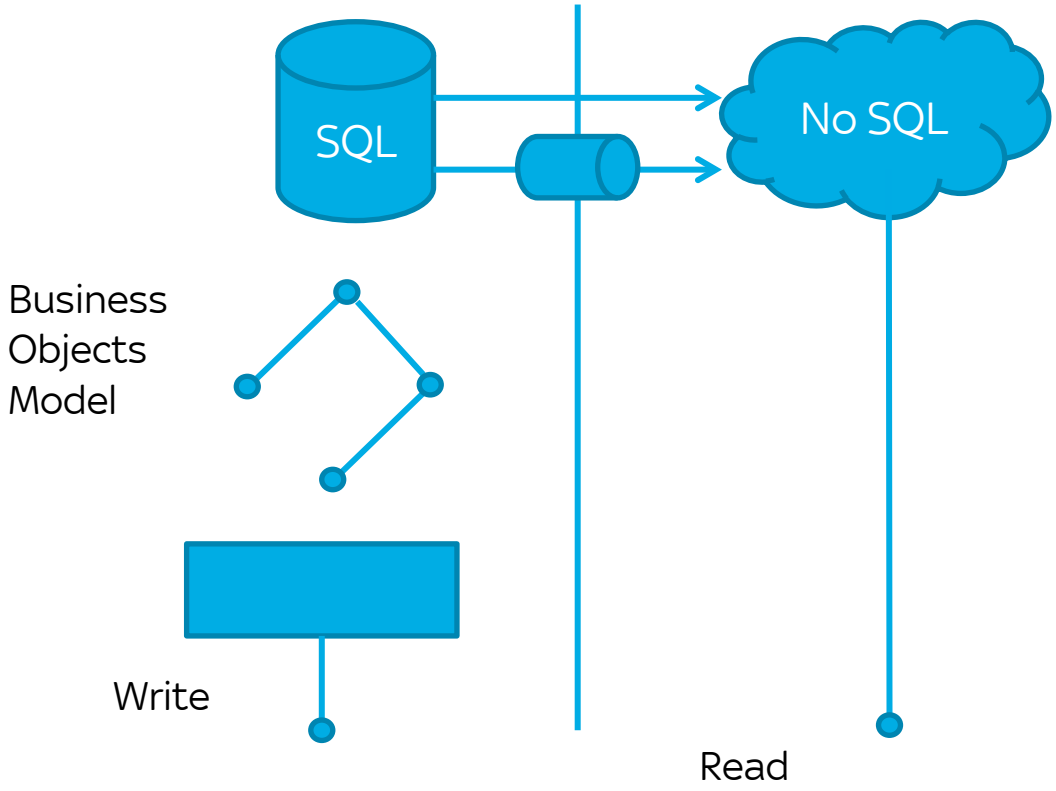
- Same protocol for read and write
- Looks simple
- Complex scalability (concurrency)
- Read performance will decrease
- Security



Segregating requests and commands

- Different repository access to read and write
- Domain objects will apply the business logic and will apply constraints
- Command Query Responsibility Segregation: message driven integration, eventual consistency
- Notification based integration: asynchronous
- Feed based integration: asynchronous and batch

Segregating requests and commands



	Client-Server	CQRS	Notification based integration	Feed based integration
Synchronicity	Synchronous	Synchronous	Asynchronous	Asynchronous
Read/Write	R/W	R/W	RO	RO
Scalability	Coupled read and write	Decoupled	Decoupled of read not from write	Decoupled
Resilience	No	Read	Integrated	Integrated
Eventual consistency	No	Yes	Yes	Yes

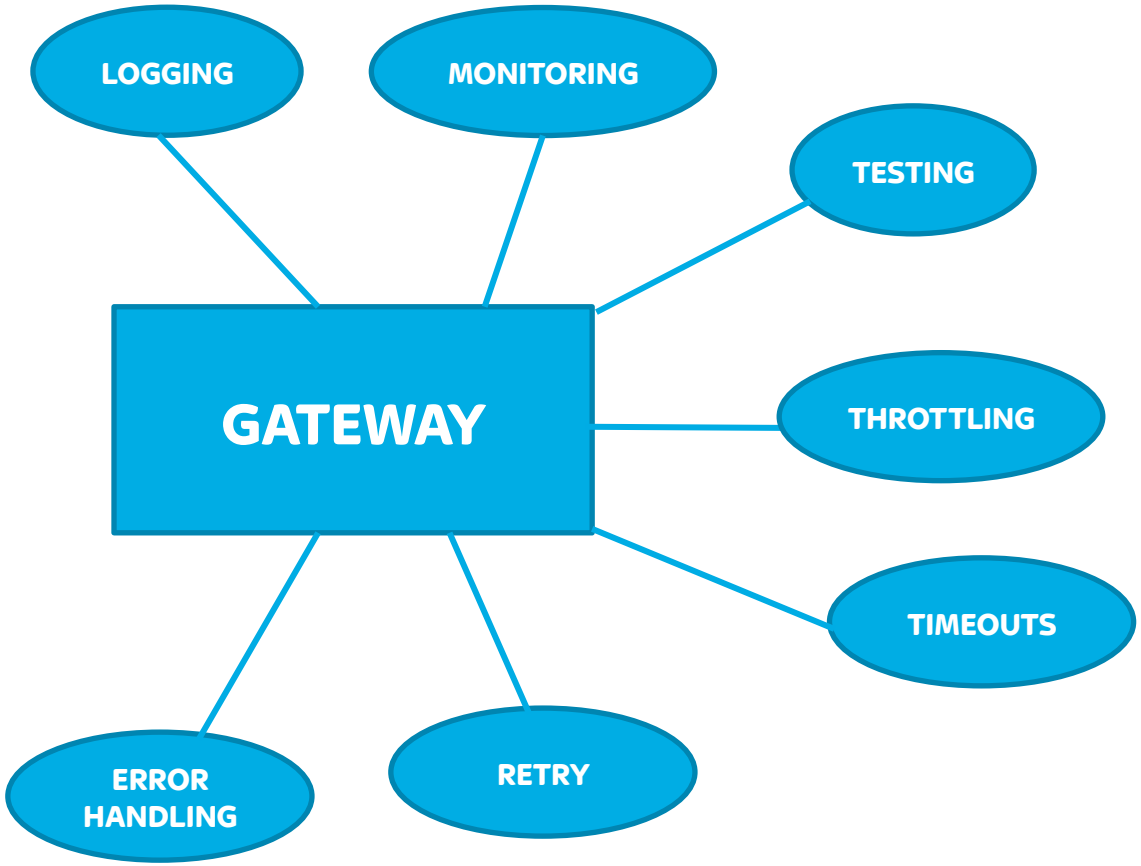
Implementation

How much effort does it take you to implement the happy path of an interface with other system in comparison with the unhappy path?

A. 50%-50%
B. 80%-20%
C. 20%-80%

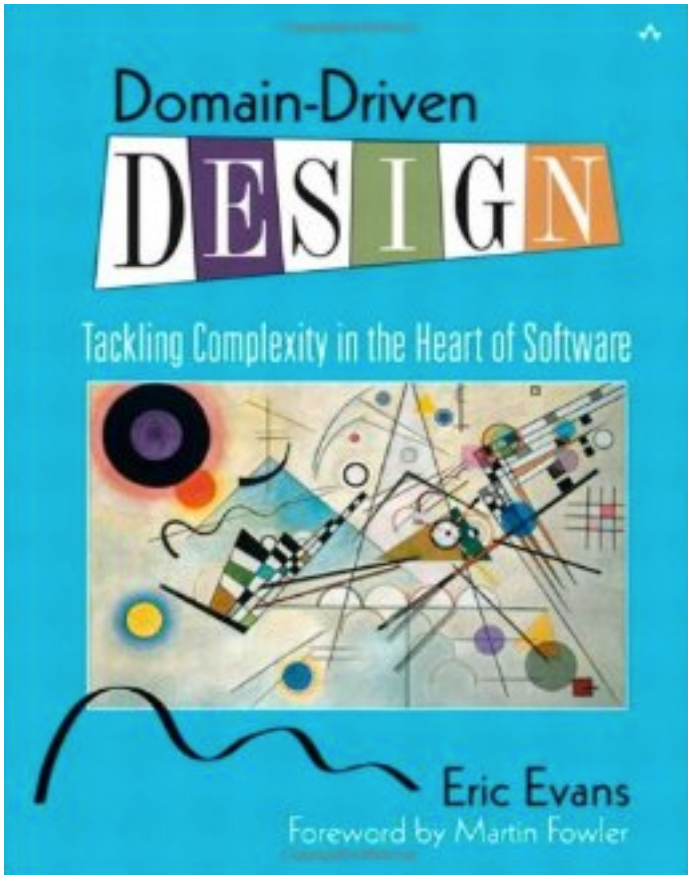
Gateway pattern

- Retry?
- Logging
- Monitoring
- Error handling
- Throttling
- Timeouts
- Testing: mocking, Chaos Monkey



References

References


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

Any questions?