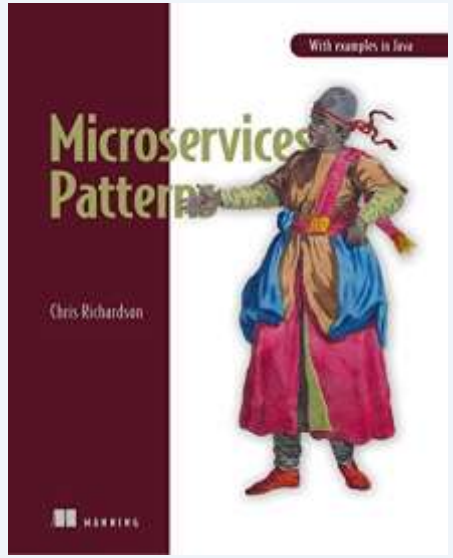
Refactoring Monolithic Applications

Objectives

- Refactoring Monolithic Applications
- The Strangler Pattern
- Refactoring Strategies
- The Anti Corruption Layer Pattern



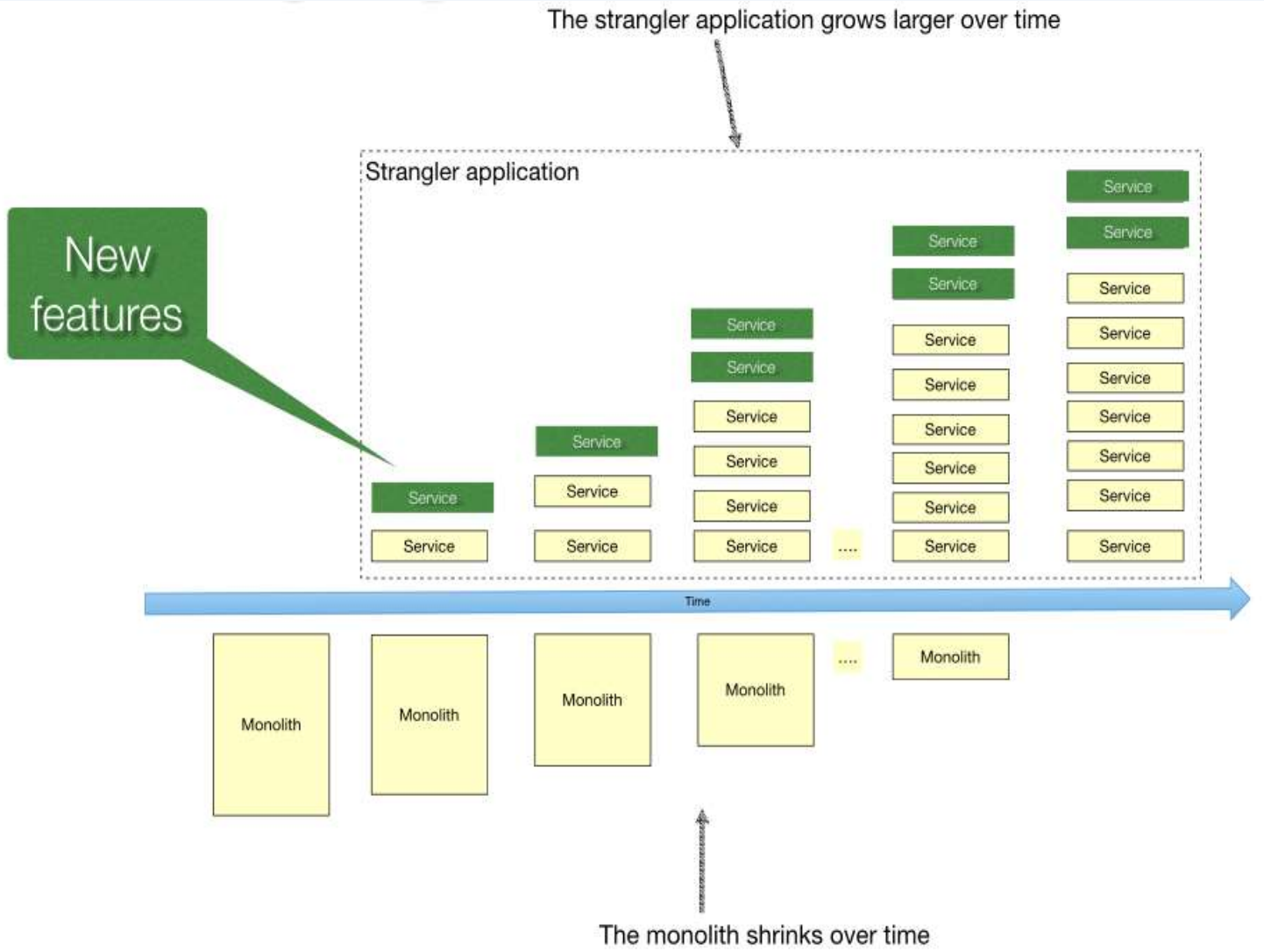
Refactoring Monolithic Applications

- You have an existing monolithic application that is under constant use and is showing the signs of strain
  - Slow delivery
  - Buggy software releases
  - Poor scalability

- Then you have two choices
  - Rebuild from the ground up
    - Normally not a great idea – long lead time, need to maintain your existing app at the same time
  - Refactor into a microservices application
    - A better idea, but how?

Strangling the Monolith

- The idea here is to incrementally refactor your monolith.
- Microservices are run in conjunction with your monolith.
- Over time, the amount of functionality implemented in the monolith shrinks until it either disappears or becomes another microservice.
- A bit like servicing your car while driving on the Autobahn
  - But less risky than a complete rewrite!



Owerya RESOURCING

Key points about refactoring

- Don’t be in a hurry
  - Could take months or years to complete
  - Amazon.com apparently took a couple of years to refactor
- Demonstrate value early and often
  - For your team’s morale and management interactions
  - Introduce more appropriate DevOps technologies as you introduce new microservices
  - Implement key new features and demonstrate how quick and robust this is.
- Minimize changes to the monolith
  - It is stable now, do everything you can to avoid, widespread changes that are time-consuming, costly and risky.
  - Introduce as few interface changes as you can!

Key points about refactoring

- You don’t need the whole technical deployment infrastructure immediately
  - Let your deployment infrastructure grow and adapt with your microservices
  - What you **do** need from the start is a deployment pipeline that does automated testing (E.g. Jenkins)
  - Once you have a few services running you can decide what sort of configuration architecture, microservices chassis etc you need.

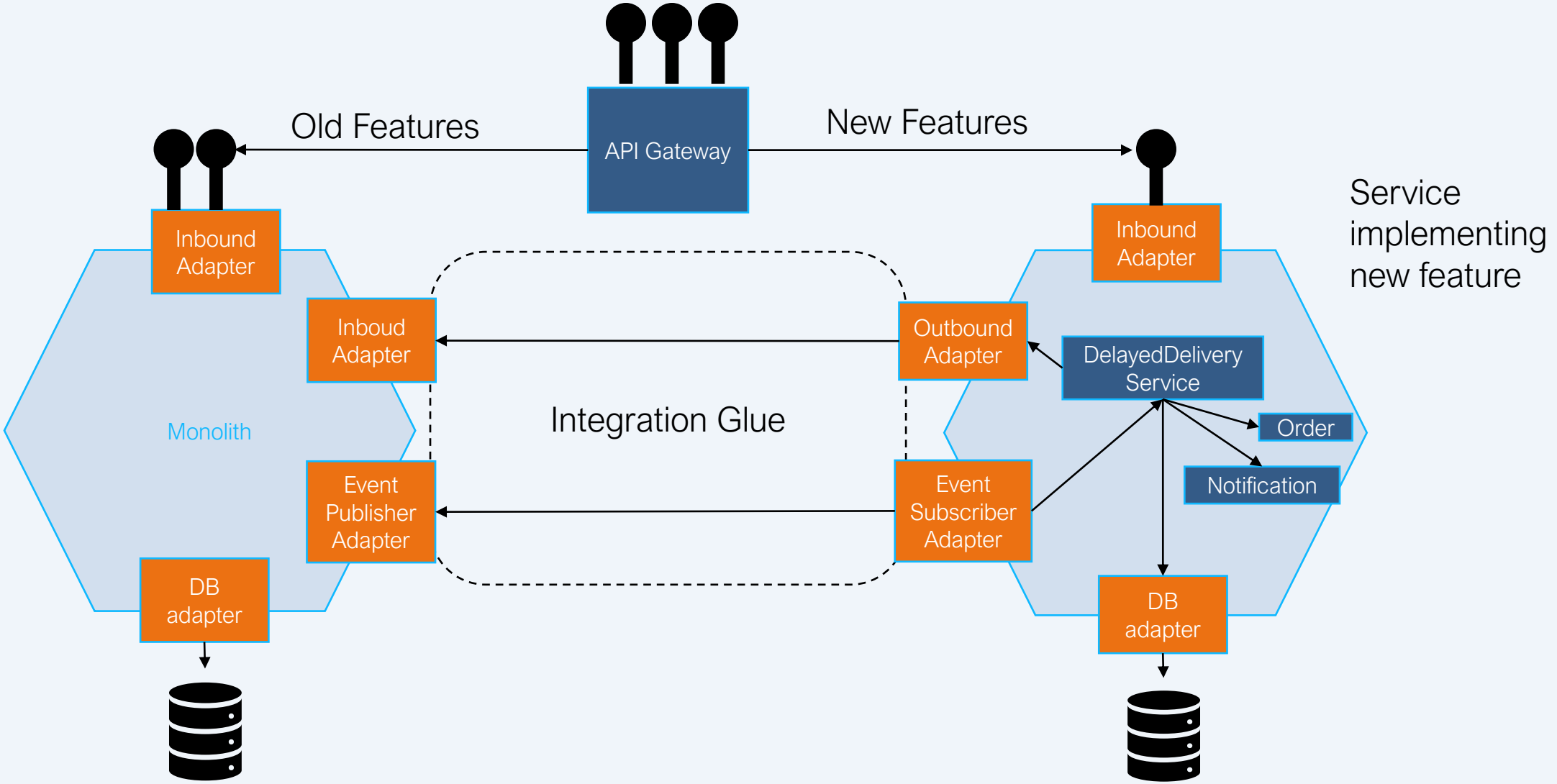
Refactoring Strategies 1: Implement new features as services

- You are refactoring because your monolith is unmanageable, don't add to the codebase!

- There are two pieces of extra functionality you will need:
  - An API Gateway – this can route requests for new functionality to the new service, and other requests to the old codebase.
  - Integration glue code – integrates the new service to the monolith.
    - Not a standalone component, but adapters in the monolith and the service that use interprocess communication of some sort.

- You can't implement every new feature as a service
  - It might be too small to make sense
  - It might be too tightly coupled to existing functionality to disentangle.

Implementing a new feature as a service



Strategy 2: Separate Presentation Tier from Backend

- Split the presentation layer from the business logic and data access layers.
  - *Presentation logic* handles HTTP requests, generates HTML pages that implement the UI.
  - *Business logic* implements the business rules
  - *Data access logic* access infrastructure services such as databases and message brokers.

- In most code bases there is already a fairly clean separation, so this is a good place to start.

Strategy 3: Extract Business Capabilities into Services

- Ultimately you have to start doing the core work of breaking apart the monolith.
- The Strangler Pattern encourages this to be done by incrementally migrating business capabilities to services.
- You really want to extract a vertical slice from the monolith that encapsulates a coherent business capability or aggregate
  - Inbound adapters that implement API endpoints
  - Domain logic
  - Outbound adapters such as database access logic
  - The monolith's database schema

Splitting the Domain Model

- This is where it really starts happening.
- You need to disentangle the new service from method calls and object references in the monolith.
- One way to do this is to replace references with object IDs/primary keys.
  - No object references crossing over service boundaries.
  - The *Domain Aggregate* pattern from DDD does this automatically.
- It may also require decomposing “God classes” that touch everything.

Refactoring the Database

- Many classes in the monolithic domain model may be persistent
  - This means the database will also need to undergo some changes.
- In some cases there is very little overlap, but in many cases a single database table could encapsulate multiple entities.
  - When you split the entity, you need to define a new table at the same time.
- One approach to this is to replicate data to avoid widespread changes.
  - For the transition period preserve the original schema and use triggers to synchronize the original and new schemas.

How to decide what and when

- All this does leave the difficult decision about exactly what is extracted when.

- One approach is to force yourself to go through it be freezing *all* development on the monolith,
  - Instead of implementing bug fixes or new features you extract the necessary services and change those.
  - Benefit is that you will get your refactoring done.
  - Drawback is that you may not refactor in the best way because you are driven by short-term.

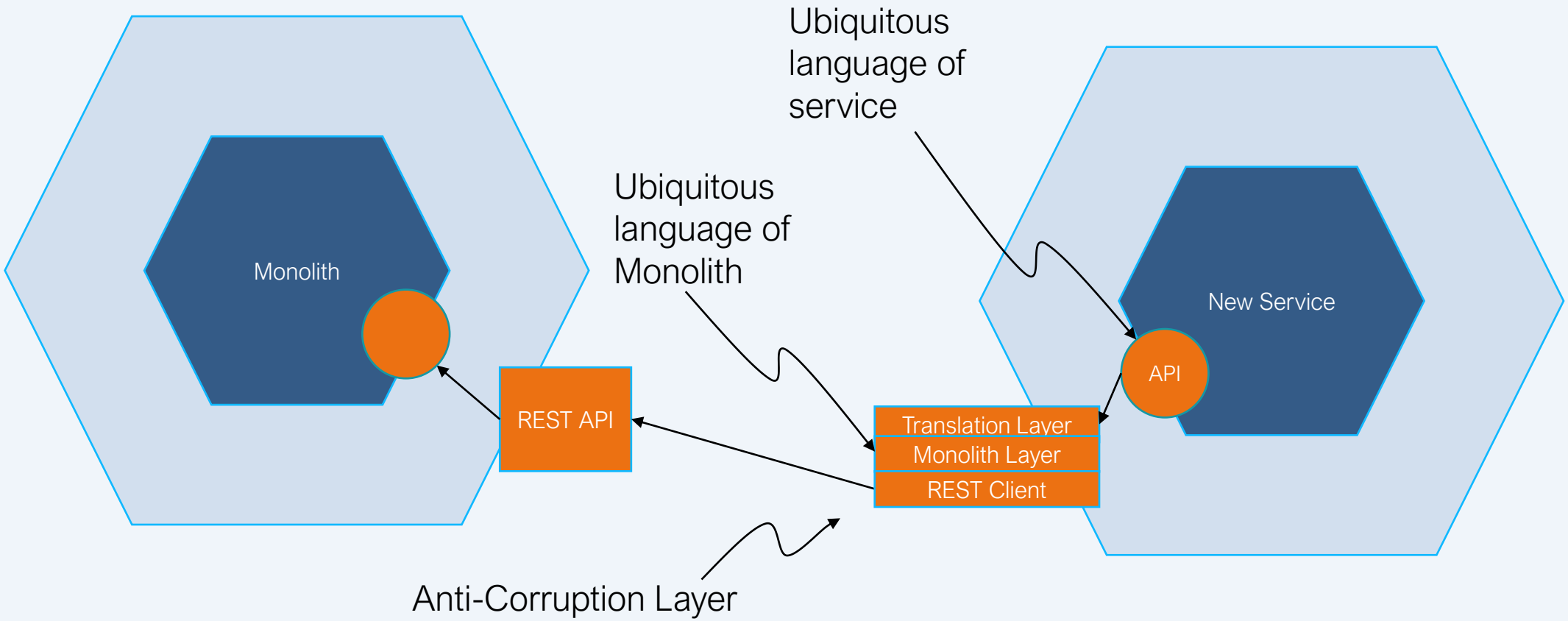
- Another approach – Rank all your services/modules by the benefit they offer and extract them one by one.

Anti-Corruption Layer Pattern

- When you break apart a monolith, you may wind up refactoring an extracted service so it behaves quite differently to the corresponding module in the monolith.
- This results in a mismatch in the domain models of the monolith and the new service.
- This poses the problem of how do you avoid the monolith domain model polluting your new service?

- Solution: Design an anti-corruption layer that translates messages between the monolith and the new service.
  - This maybe able to be removed later on depending on further refactoring.

Anti-corruption layer



Summary

- Refactoring Monolithic Applications
- The Strangler Pattern
- Refactoring Strategies
- The Anti Corruption Layer Pattern

Questions or Comments?

Appendix: Automatic Refactoring Tools

- There are a growing number of tools on the market to automatically refactor Java monoliths to microservices.
- These generally use an AI agent to analyse the application at runtime, as well as static code analysis to propose a decomposition.
- At the very least you may get a good starting point for your microservices.
- Examples:
  - Transform monolithic Java applications into microservices with the power of AI - IBM Developer
  - Cloud Application Modernization Platform | vFunction