Dutch Devops Engineers

Refactoring a Monolith to Microservices

Bastiaan Bakker, Marco van der Linden, Jan Vermeir

Programme

- Welcome
- Microservices, Bounded Context and ABC
- Identify service domains through visualization
- Other techniques
- Exercise 1: Break up the Shop
- Integration patterns
- Exercise 2: Increase robustness
- Exercise 3: From REST towards Events
- Close

Introduction

Jan Vermeir

Bastiaan Bakker

Marco van der Linden

Microservices, Bounded Context and ABC

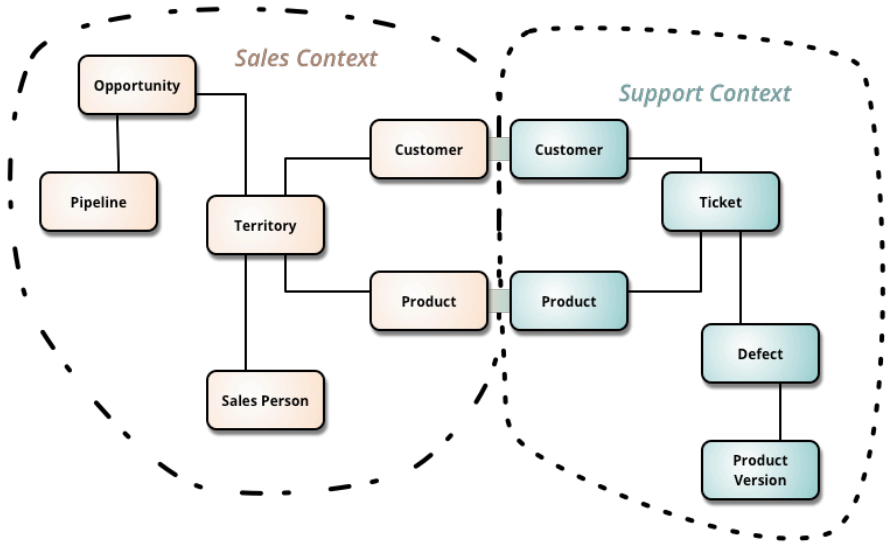
Microservices

Architectural style for software systems based on **services**:

- Each service provides a well-defined business capability
- Communication with uniform, lightweight, interfaces
- Allows independent development and maintenance
- Allows independent installation and operation
- Allows implementation in different technologies

Bounded Context

Use of DDD concepts such as Domain and Bounded Context allow clean separation and identification of boundaries (interfaces)



*"Bounded contexts are autonomous components, with their own domain models and their own ubiquitous language. They should not have any dependencies on each other at run time and should be capable of running in isolation. However they are a part of the same overall system and do need to exchange data with one another."**

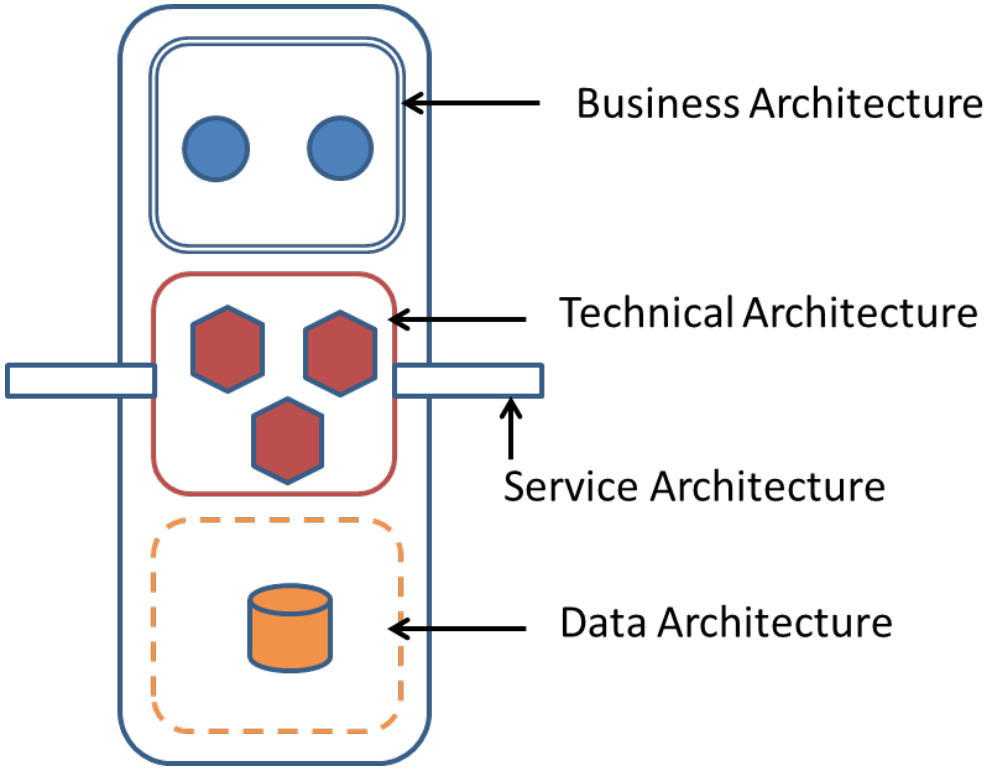
* https://msdn.microsoft.com/en-us/library/jj591572.aspx

Autonomous Business Capability

Terminology from Roger Sessions' **Snowman Architecture**. This approach can be used to simplify complex architectures.

Basic tenet: group together related functionality into autonomous components with explicit boundaries.

- #Rule 1: Deep partitioning
- #Rule 2: Data is private, information is shared
- #Rule 3: Connect with async messages
- #Rule 4: Transactions never cross boundaries
- #Rule 5: Boundaries must be maintained



See http://simplearchitectures.blogspot.nl/2012/09/snowman-architecture-part-one-overview.html

Divide and conquer... and synchronize

- Although autonomous, services will still need to exchange information
- Some synchronization patterns (and their drawbacks):
  - Direct (synchronous) calls → requires simultaneous availability
  - Distributed transactions → slow, brittle
  - Asynchronous calls → delayed consistency
  - Local cache of (remote) data → redundancy
  - Event publication → eventual consistency
- There is no ‘best’ approach, it depends …

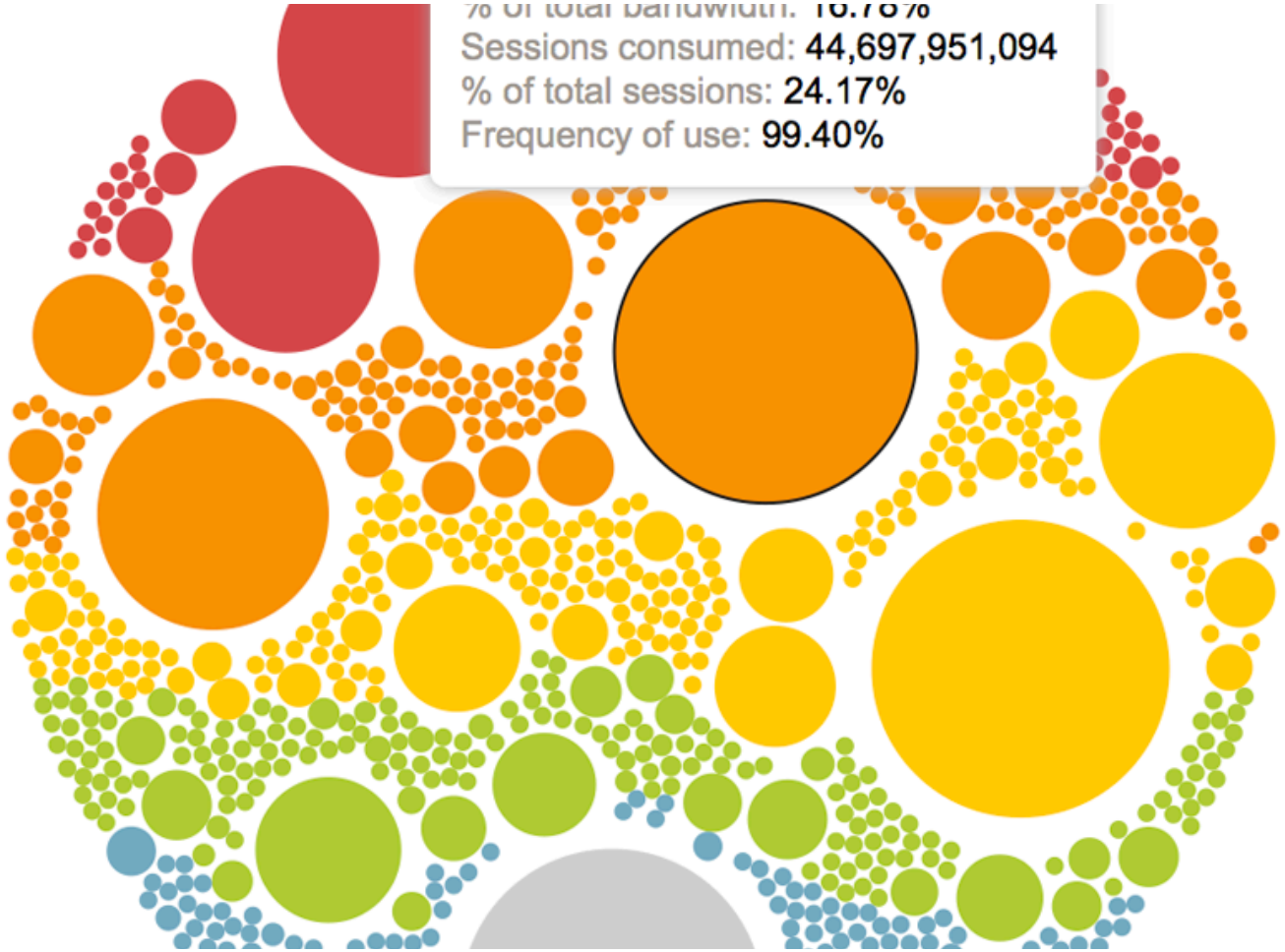
Five steps to break up a monolith

Five steps to break up a monolith

1. Determine ABC (exercise). Result is a list of services
2. Map ABC to IT-landscape
3. Isolate ABC
   - define service boundaries
4. Implement service
   - look at existing packages and if needed refactor package structure
   - move code and data to new service
5. Integrate service
   - integrate new service with old monolith and other services

Visualization

Visualization



Size of circle indicates
total bandwidth occupied

Visualization of relationships

Xebia

Low Coupling

Measurement of dependency between two components

- Expressed as sensitivity towards propagation of changes and errors

Levels

- No Coupling
  - Components are completely insensitive to changes and errors in other components.
- Low Coupling
  - Error and changes seldom propagate errors and changes to dependent components
- High Coupling
  - Errors and changes in one component almost always propagate to other components

High Cohesion

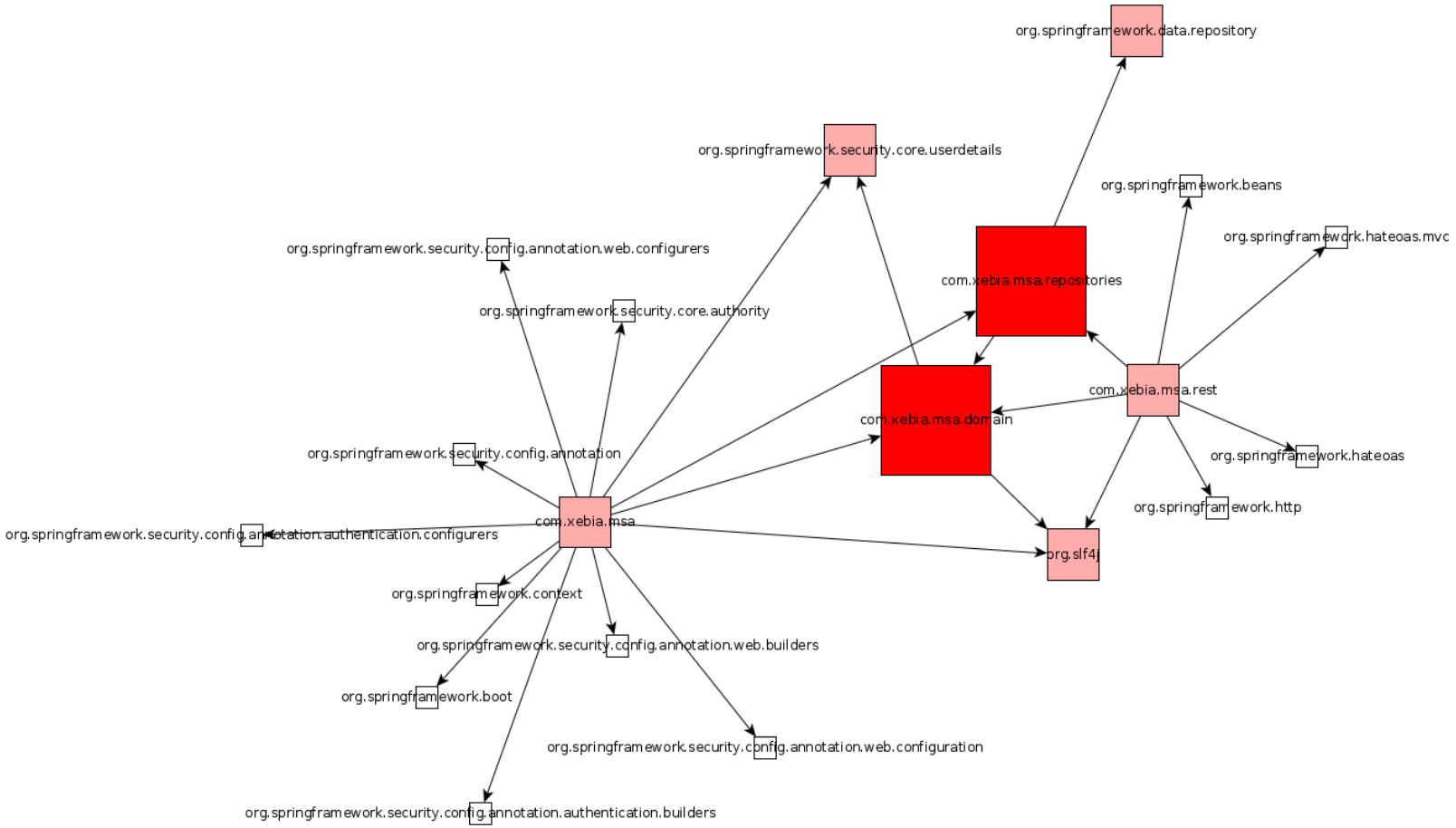
Degree to which parts of a module belong together

Levels

- Coincidental cohesion (**worst**)
  - random
- Logical cohesion
  - same in nature
- Temporal cohesion
  - executed at a particular time
- Procedural cohesion
  - part of fixed execution sequence
- Informational cohesion
  - operating on the same data
- Sequential cohesion
  - Output of one is used as input by another
- Functional cohesion (**best**)
  - contributing to a single task

http://en.wikipedia.org/wiki/Cohesion_%28computer_science%29

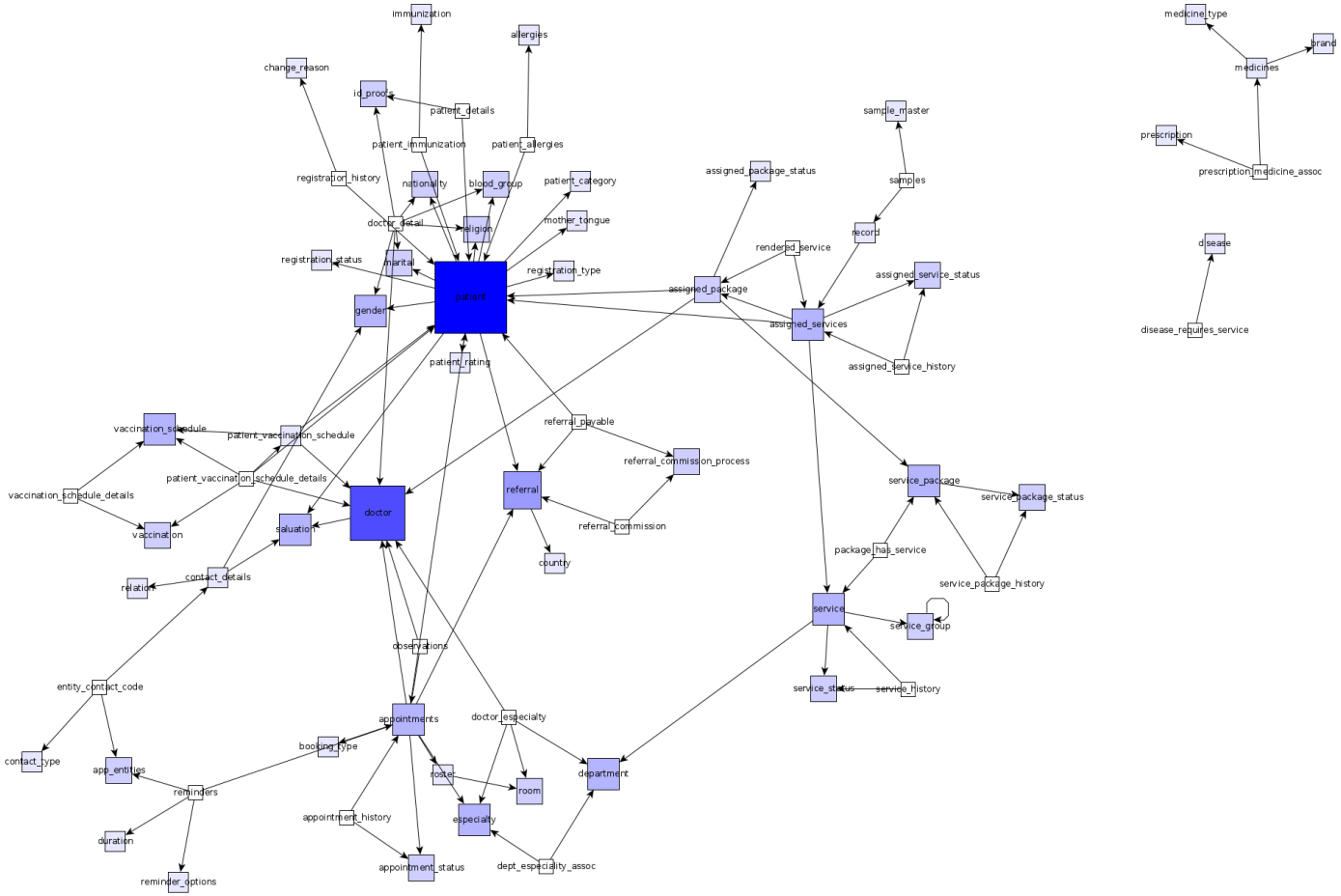
Package Dependencies



Foreign Key Constraints (1)

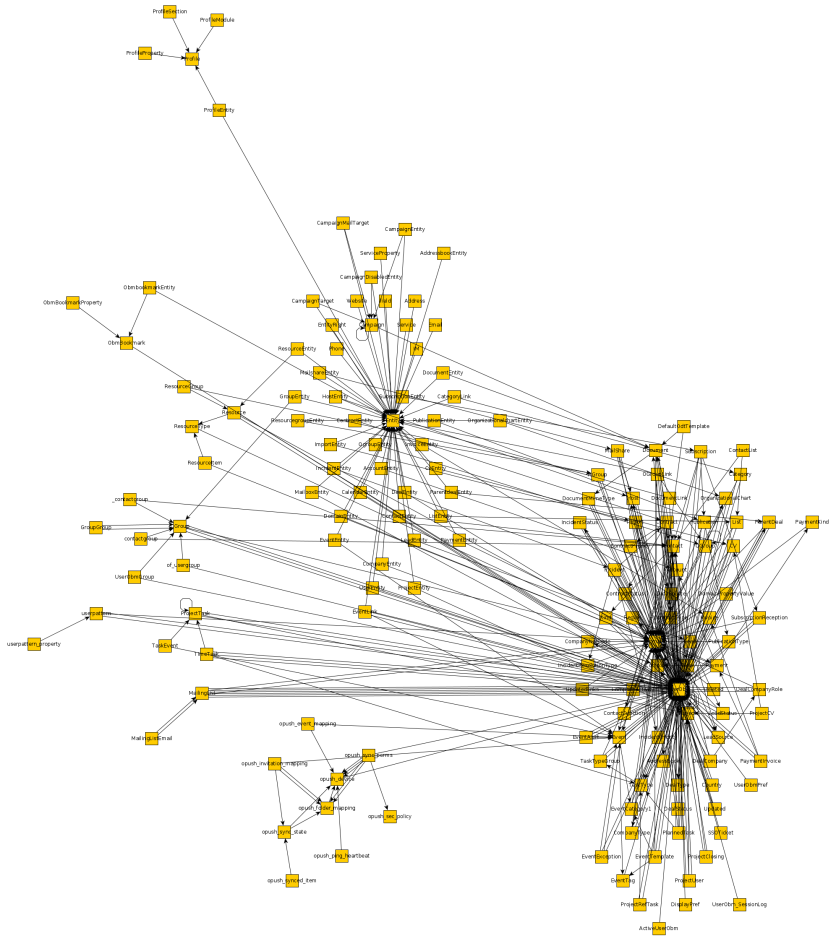
```
CREATE TABLE `patient`
…

ALTER TABLE `patient_allergies`
  ADD CONSTRAINT `patient_allergies_ibfk_1` FOREIGN
KEY (`PATIENT_ID`) REFERENCES `patient`
(`PATIENT_ID`),
  ADD CONSTRAINT `patient_allergies_ibfk_2` FOREIGN
KEY (`ALLERGY_CODE`) REFERENCES `allergies`
(`ALLERGIES_CODE`);
```

Foreign Key Constraints (2)



https://github.com/walkingtree/hcis/blob/master/Sql/hcis_core_schema.sql

Foreign Key Constraints (3)



https://github.com/mbaechler/OBM/blob/master/ui/scripts/2.4/create_obmdb_2.4.mysql.sql

On dependencies between services

- Different kinds of dependencies:
  - At **develop-time**: designing and implementing functionality
  - At **compile-time**: building executable services
  - At **run-time**: delivering business functionality in customer journeys
- Microservices architecture primarily addresses the first two
  - Independent development of business capabilities
  - Isolated deployment of business services
- But on customer journey-level, dependencies might remain
  - Customer journeys integrate multiple business capabilities

Other analysis techniques

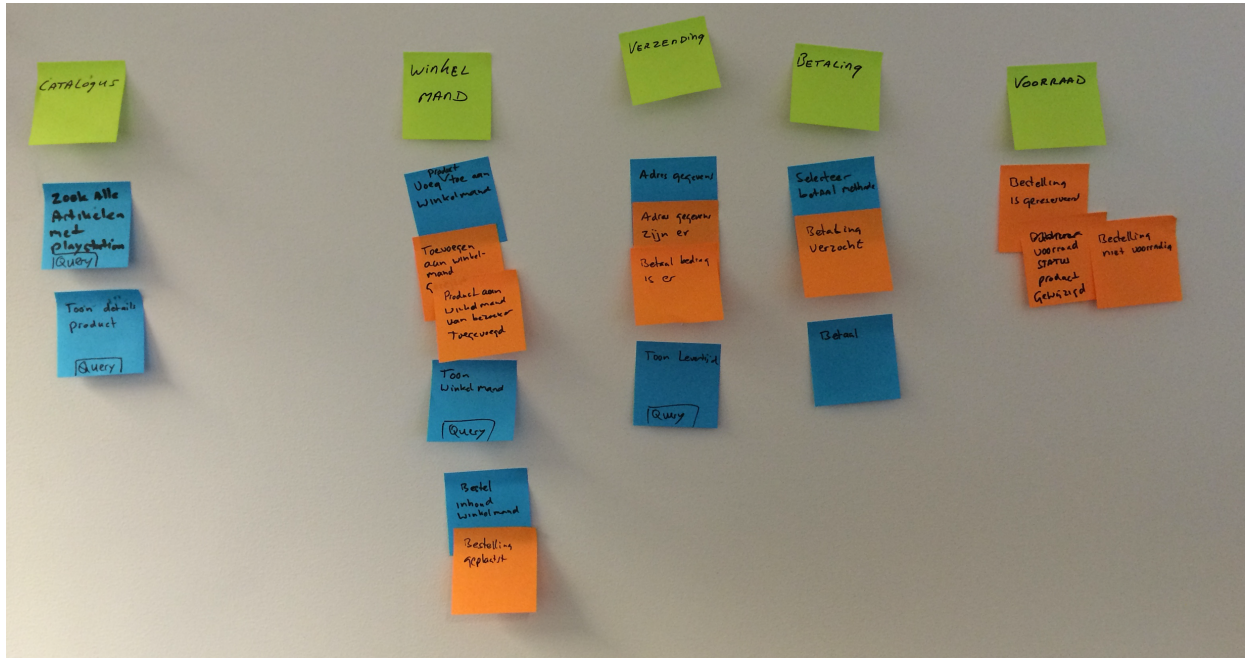
Event storming

workshop format for quickly exploring complex business domains

- Identify domain events (orange)
- Identify commands (blue)
- Look for aggregates

Optional

- Subdomains and contexts
- User personas
- Key acceptance tests
- …



http://ziobrando.blogspot.nl/2013/11/introducing-event-storming.html

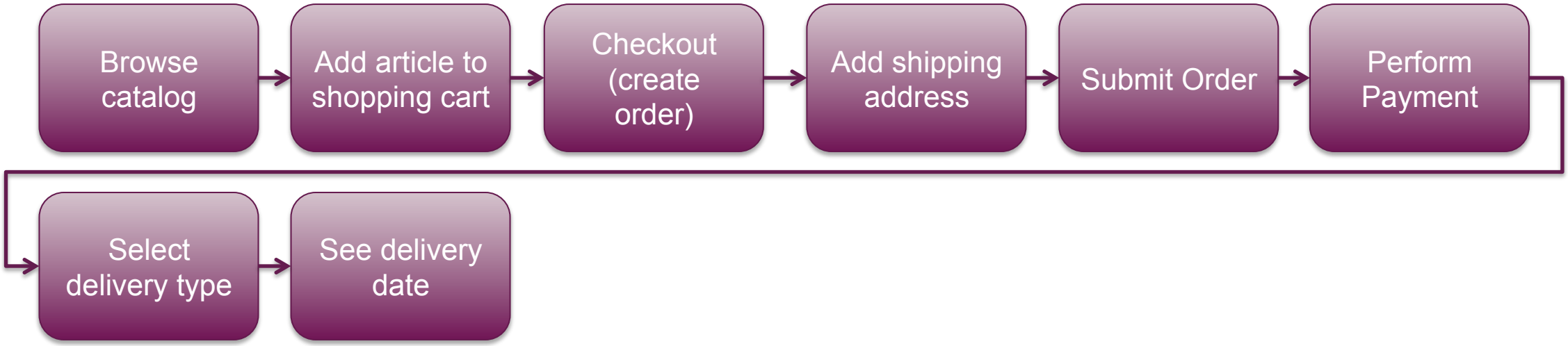
Simple Iterative Process

- Take 2 functions. If changes in the implementation of A are very likely to cause changes to the implementation of B, or visa versa, then group these functions together
- Do this for all the functions

A partitioning will emerge which can be used to identify bounded contexts

Exercise 1: The “Shop”

Order products



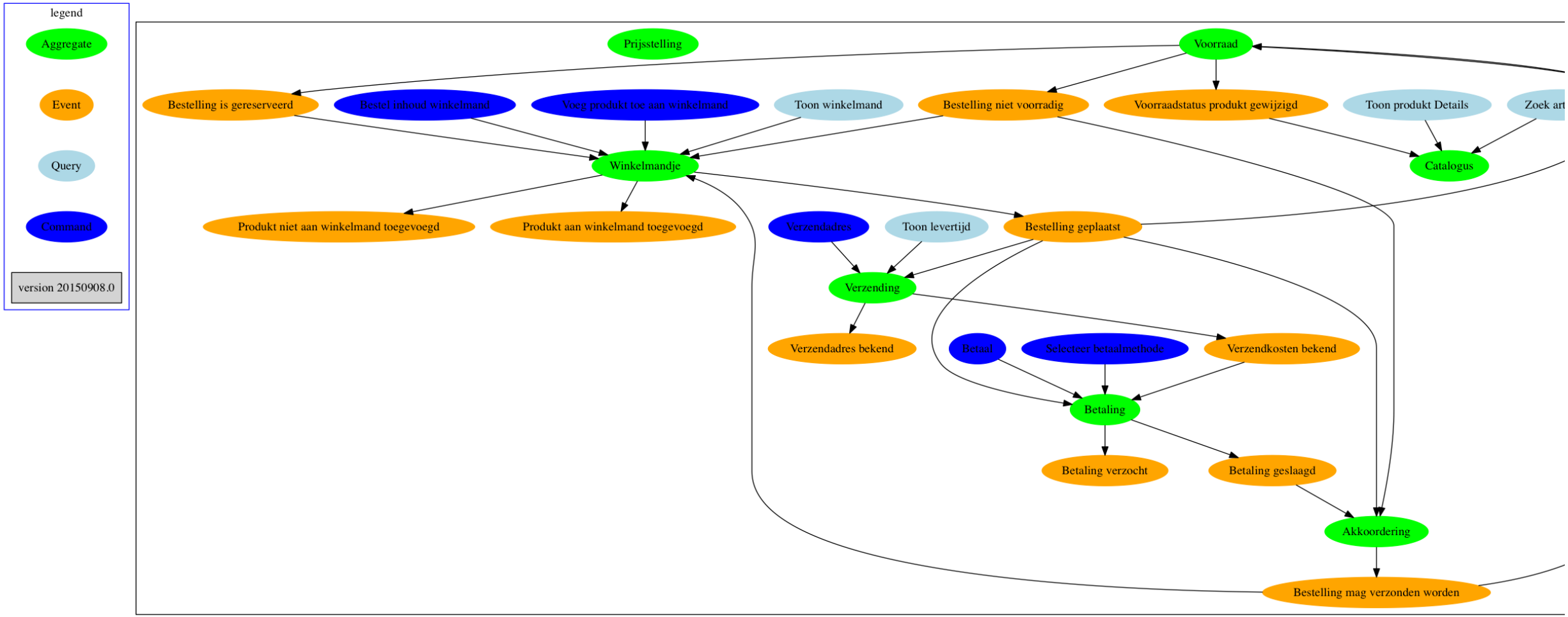
Exercise – break up the monolith

Goal of this exercise is to find out how a monolith can be broken up.

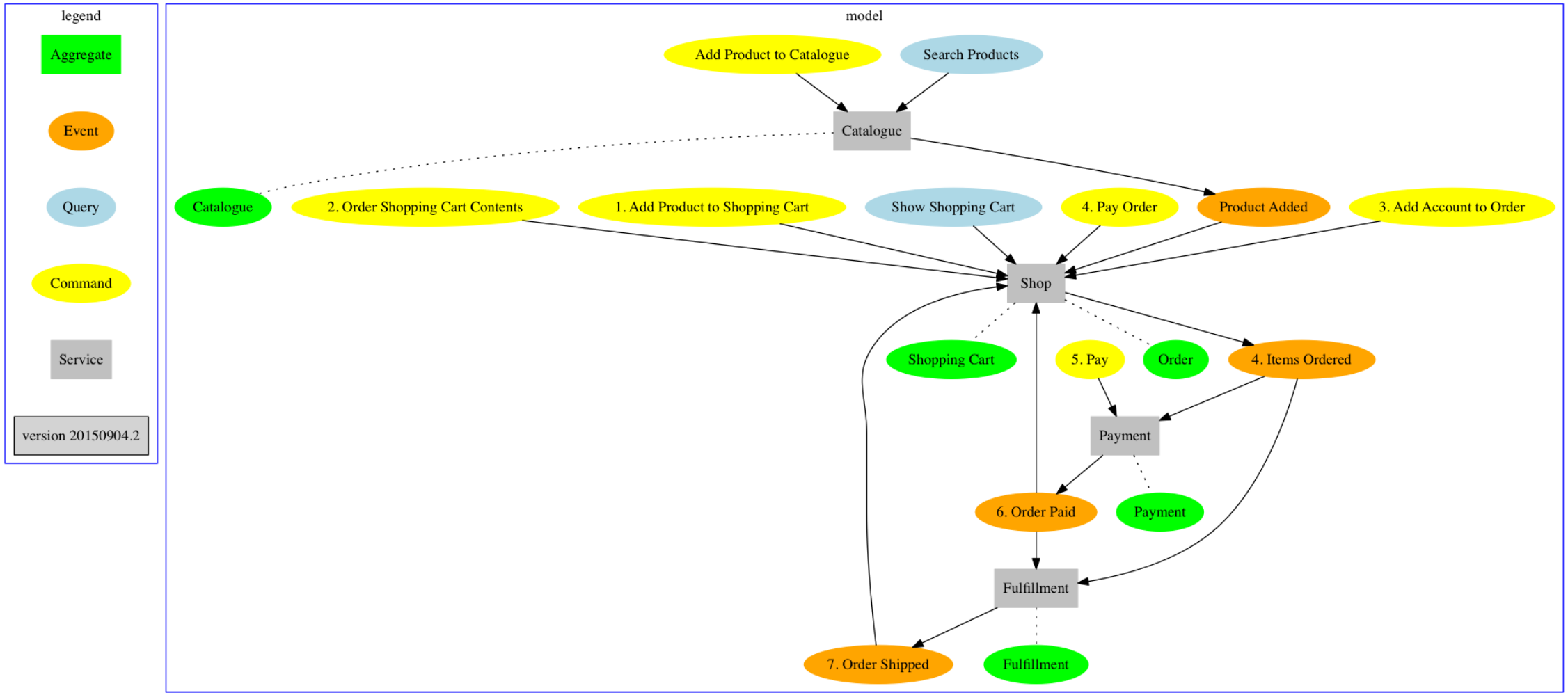
- Form groups
- Take 15 minutes and group the functions to identify services and their boundaries
- What additional business functions can you think of ?

Business Functions

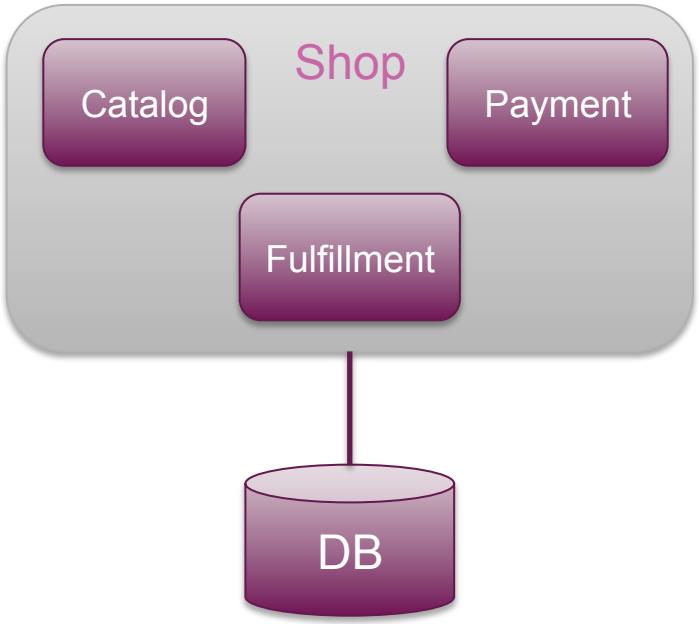
Event Storming results



Shop: design with 4 services



However, we started with this



Refactoring steps

1. For each service defined its (domain) model, and how to break up the monolithic database model
2. Extracted code from Shop to new Spring Boot applications: Payment, Fulfilments (Shipping) and Catalogue
3. No real need to refactor package structure, but a lot of entities now appear in multiple services (for instance Order), but with different meaning and different fields
4. Most REST interfaces remain, but based on the design we add new interfaces for communication between services
5. Automated tests were added to allow testing of the customer journey across the services

There is now a lot more code, dependencies that were hidden in the monolith are now explicit. However, we now need to handle process to process communication.

In our first approach all services communicate via REST calls

RESTful integration

Shopping Cart with Shipping

```
try {
    RestTemplate restTemplate = new RestTemplate();
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);
    HttpEntity<String> requestEntity = new HttpEntity<~>(objectMapper.writeValueAsString(orderr), headers);
    restTemplate.exchange(FF_ENDPOINT + "/orders", HttpMethod.POST, requestEntity, String.class);
} catch (RestClientException e) {
```

- Brittle, what happens if fulfilment service is slow or down?
- How can we improve this?

http://blog.xebia.com/2015/03/18/microservices-coupling-vs-autonomy/

Exercise 2

Increase robustness 

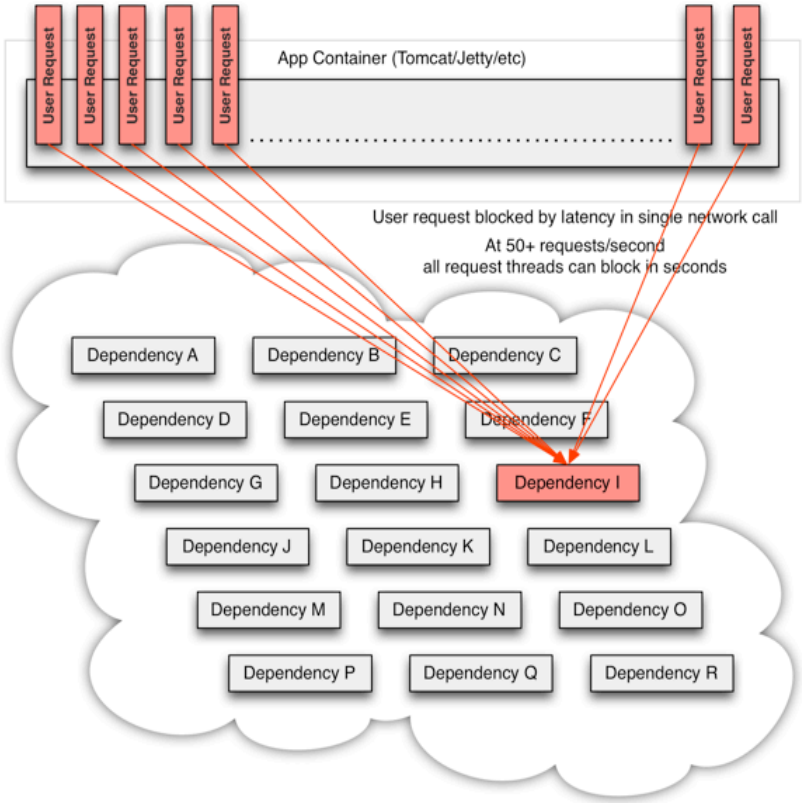
Circuit breaker pattern

"Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable"

Hystrix is a library that implements the circuit breaker pattern and provides:

- Timeouts
- Fallbacks
- Monitoring of failures and successful calls.

Allows graceful degradation, Retries, Fail Over or Fail Fast



https://github.com/Netflix/Hystrix

http://martinfowler.com/bliki/CircuitBreaker.html

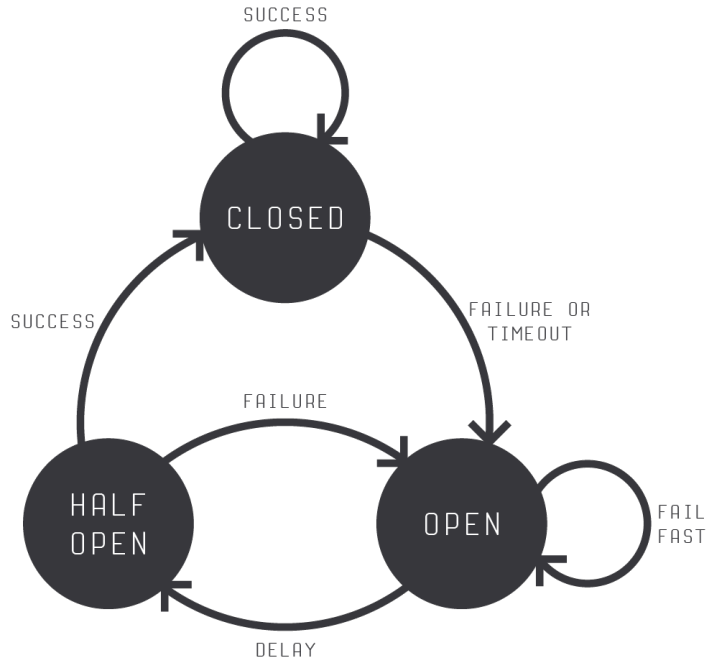
Increase robustness

In order to be able to do this exercise you must clone the following GitHub repository:

```
git clone https://github.com/xebia/microservices-breaking-up-a-monolith.git
```

- The exercise is described in git repo/exercise-circuitbreaker/readme.md
- The goal is to implement a circuit breaker
- Time limit for this exercise is 30 minutes

- Use https://github.com/Netflix/Hystrix to get familiar with circuit breakers.



http://bartoszsypytkowski.com/design-patterns-circuit-breaker/

Exercise 3

From REST towards Events 

From REST towards events

In order to be able to do this exercise you must clone the following GitHub repository and follow the instructions described in the `readme.md` file.

```
git clone https://github.com/xebia/microservices-breaking-up-a-monolith.git
```

- The exercise is described in git repo – src/exercise-queues/readme.md
- The goal is to implement integration using messages
- Time limit for this exercise is 30 minutes

Things we will explore next

We found that, already with only a couple of services, overview (who calls whom) becomes an issue. We also found that integrated testing of the services needs a better approach.

So for the next iteration we will:

- Add a testing framework
- Add a monitoring solution that uses the metrics and logs
- Add Consul or similar
- Look at a standardized format for the events
- Run load tests
- Add security
- Run on a platform (Docker?)

Evaluation

- What was cool..
- What can be improved..

Thank you for your participation!

Things to take away