Enterprise Software Architectures

Hexagonal & Onion Architecture
Event Sourcing & CQRS
DDD – Domain Driven Design
Functional Reactive Programming

Araf Karsh Hamid

> “It is NOT the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change.”
> - Charles Darwin

Agenda

Architecture Styles

- Hexagonal Architecture
- Onion Architecture

Design Styles

- Functional Reactive Programming
- Event Sourcing & CQRS
- Domain Driven Design

Scalability & Performance

- In-Memory Computing
- On Performance and Scalability
- Low Latency and Java
- Scalability Best Practices – eBay
- CAP Theorem

Thread Safety

- Threat Safety
- Dependency Injection
- Service Scope
- Generics & Type Safety

Design Patterns

- Design Patterns
- Java EE Patterns
- Patterns of Enterprise Application Architecture
- Domain Driven Design
- Enterprise Integration Patterns
- Service Design Patterns

ARCHITECTURE STYLES

- Hexagonal Architecture
- Onion Architecture

Hexagonal Architecture

Ports & Adapters

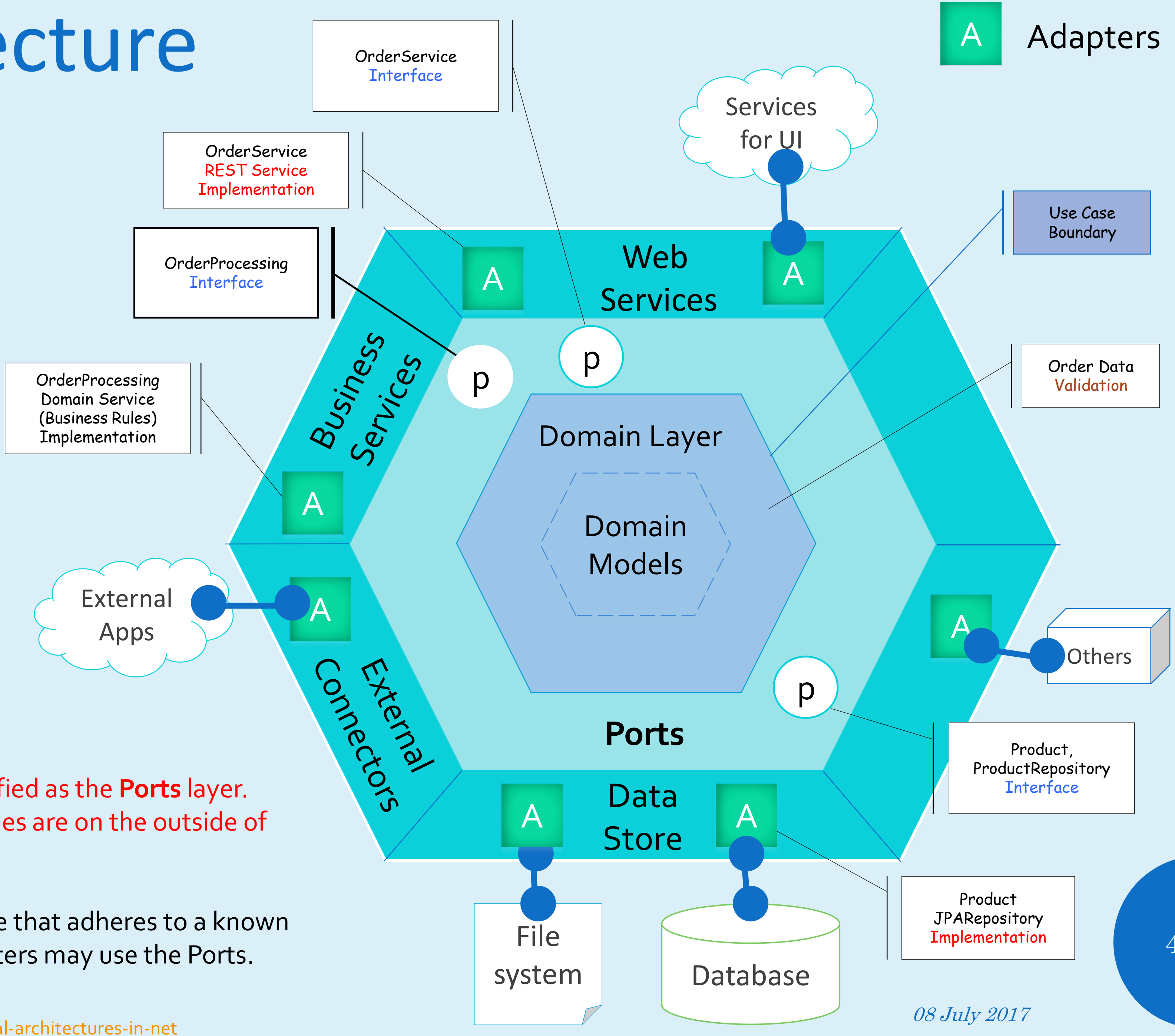
Ports and **Adapters** architecture style (Hexagonal Architecture) is a variation of layered architecture style which makes clear the separation between the **Domain Model** which contains the rules of the Application and the **adapters**, which abstract the inputs to the system and our outputs.

The advantage is the Application is decoupled from the nature of input / output device and any framework used to implement them.

For Eg., When a client makes a REST API request the **Adapter** receives the HTTP Request and transforms that into a call into the Domain Layer and marshals the response back out to the client over HTTP. Similarly if the app needs to retrieve persisted Entity to initialize the domain it calls out to an Adapter that wraps the access to the DB.

The layer between the **Adapter** and the **Domain** is identified as the **Ports** layer. The Domain is inside the port, adapters for external entities are on the outside of the port.

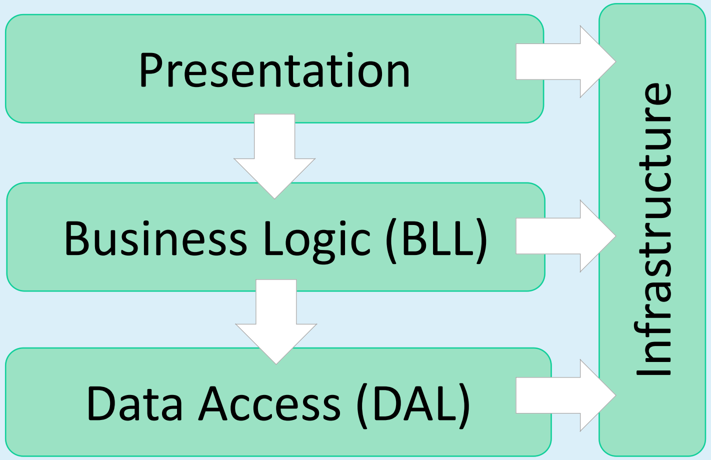
The notion of a "port" invokes the OS idea that any device that adheres to a known protocol can be plugged into a port. Similarly many adapters may use the Ports.



08 July 2017

Source : https://skillsmatter.com/skillscasts/5744-decoupling-from-asp-net-hexagonal-architectures-in-net

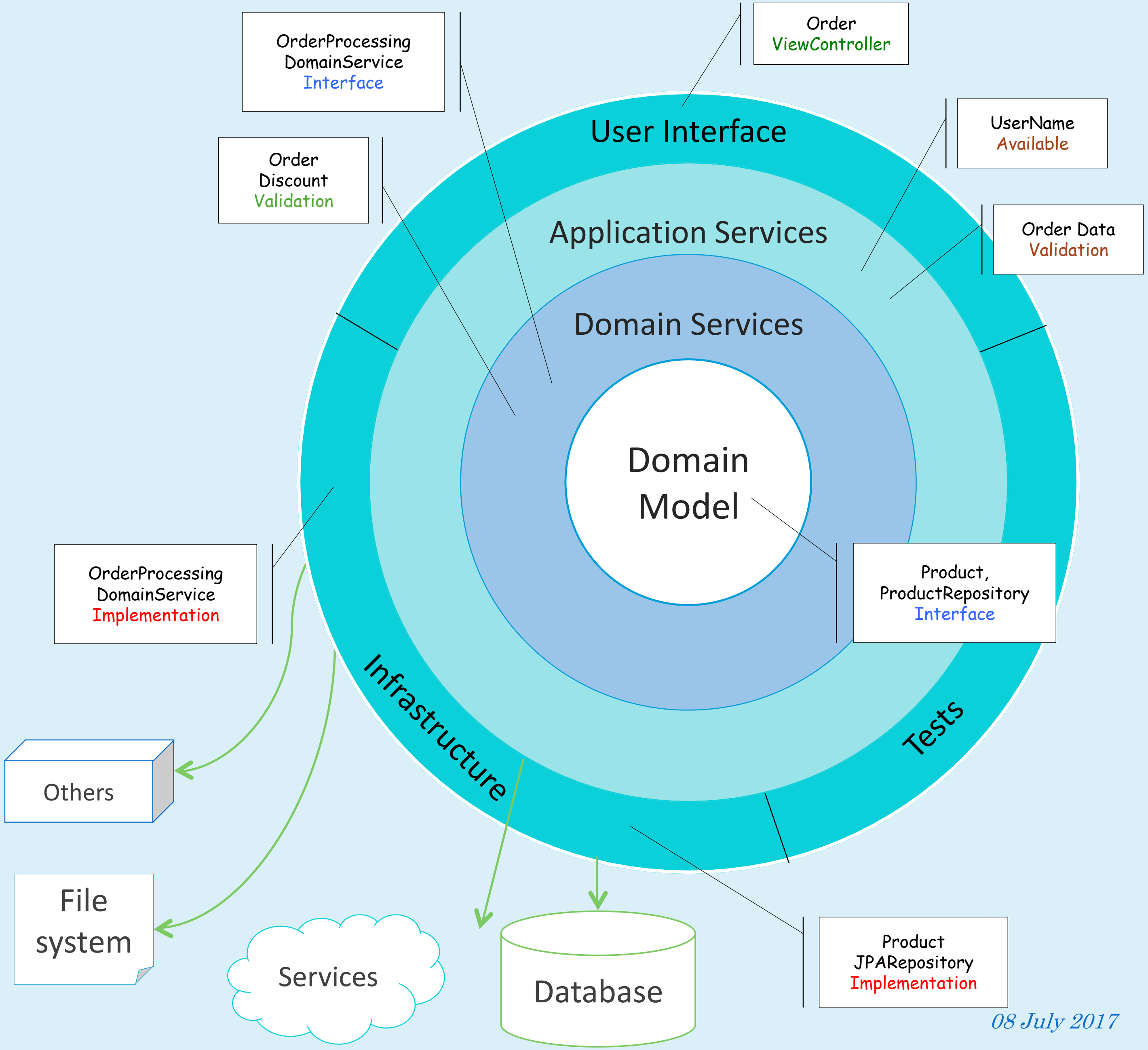
Traditional Architecture Vs. Onion Architecture + DDD



- Interface Driven
- Dependency Injection
- Auto wiring

- **Adaptable to future changes / enhancements**

Source: Onion Architecture
By Jeffrey Palermo, Ex Microsoft


OrderProcessing DomainService Interface
Order ViewController
User Interface
UserName Available
Order Discount Validation
Application Services
Order Data Validation
Domain Services
Domain Model
OrderProcessing DomainService Implementation
Product, ProductRepository Interface
Infrastructure
Tests
Others
File system
Services
Database
Product JPARepository Implementation

08 July 2017

APP DESIGN GUIDELINES

Better design better code base

Design Styles

Scalability

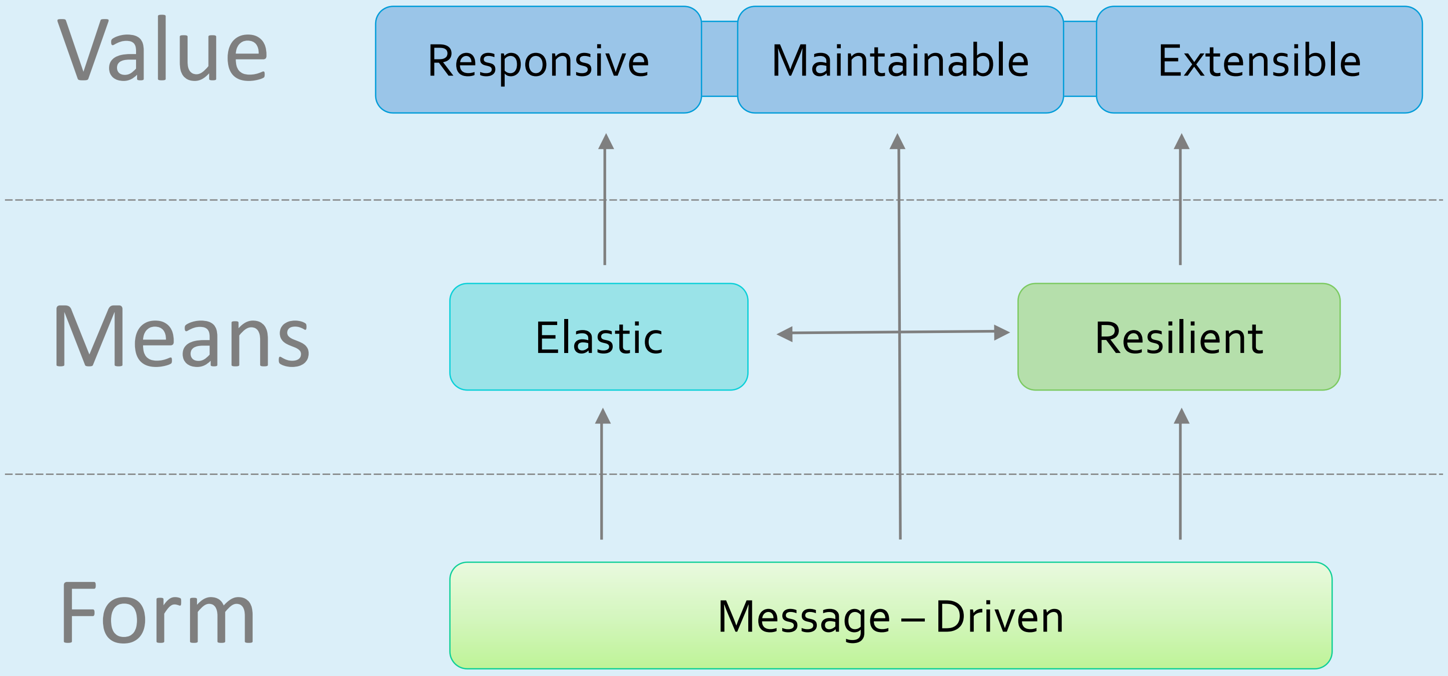
Thread Safety

Design Patterns

Functional Reactive Programming

Reactive (Principles)	Micro Services (Strategy)
Responsive, Resilient,	Bounded Context (DDD)
Elastic,	Event Sourcing, CQRS
Message-Driven	Eventual Consistency

Functional Reactive Programming



Principles		What it means?
Responsive	thus	React to users demand
Resilient	thus	React to errors and failures
Elastic	thus	React to load
Message-Driven	thus	React to events and messages

Source: http://reactivex.io/

Reactive Extensions (Rx)

1. A ***responsive, maintainable & Extensible*** application is the goal.
2. A *responsive* application is both ***scalable (Elastic)*** and ***resilient***.
3. Responsiveness is impossible to achieve without both scalability and resilience.
4. A ***Message-Driven*** architecture is the foundation of scalable, resilient, and ultimately responsive systems.

Functional Reactive Programming : Result = Decoupling

1. Containment of
   1. Failures
   2. Implementation Details
   3. Responsibility
2. Shared Nothing Architecture, Clear Boundaries
3. Micro Services: Single Responsibility Principle

Functional Reactive Programming : Design Patterns

Single Component Pattern

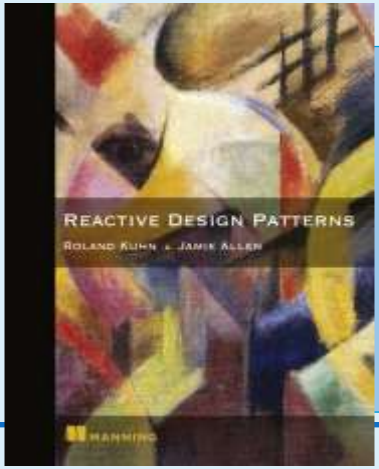
A Component shall do ONLY one thing,
But do it in FULL.

Single Responsibility Principle By DeMarco : Structured Analysis & System Specification (Yourdon, New York, 1979)

Let-It-Crash Pattern

Prefer a FULL component restart to complex internal failure handling.

Candea & Fox: Crash-Only Software (USENIX HotOS IX, 2003)
Popularized by Netflix Chaos Monkey. Erlang Philosophy



Saga Pattern

Divide long-lived distributed transactions into quick local ones with compensating actions for recovery.

Pet Helland: Life Beyond Distributed Transactions CIDR 2007

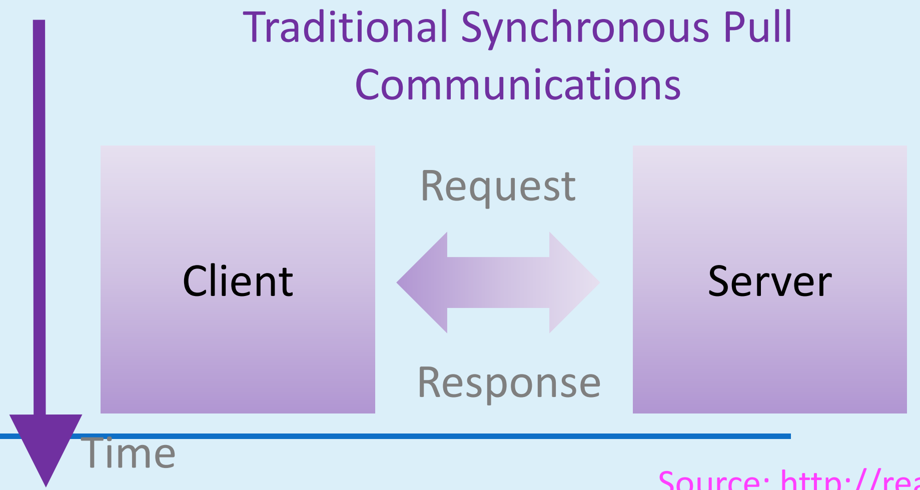
4 Building Blocks of RxJava

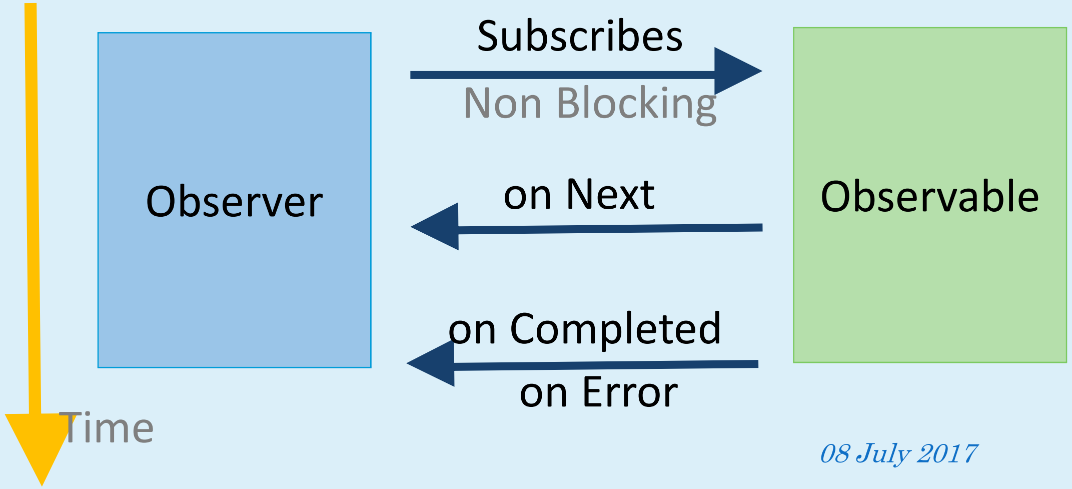
Reactive Extensions (Rx)

1 **Observable** Source of Data Stream [Sender]

2 **Observer** Listens for emitted values [Receiver]

1. The Observer subscribes (listens) to the Observable
2. Observer react to what ever item or sequence of items the Observable emits.
3. Many Observers can subscribe to the same Observable





Source: http://reactivex.io/

08 July 2017

4 Building Blocks of RxJava

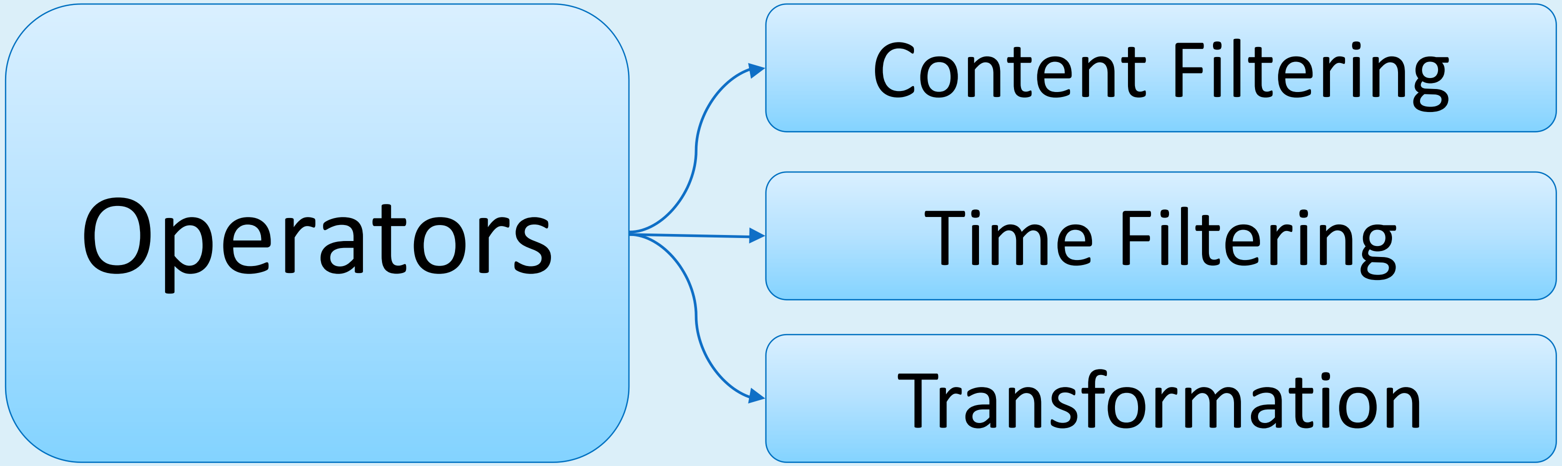
Reactive Extensions (Rx)

3 Schedulers

Schedulers are used to manage and control concurrency.

1. observeOn: Thread Observable is executed
2. subscribeOn: Thread subscribe is executed

4



Operators that let you Transform, Combine, Manipulate, and work with the sequence of items emitted by Observables

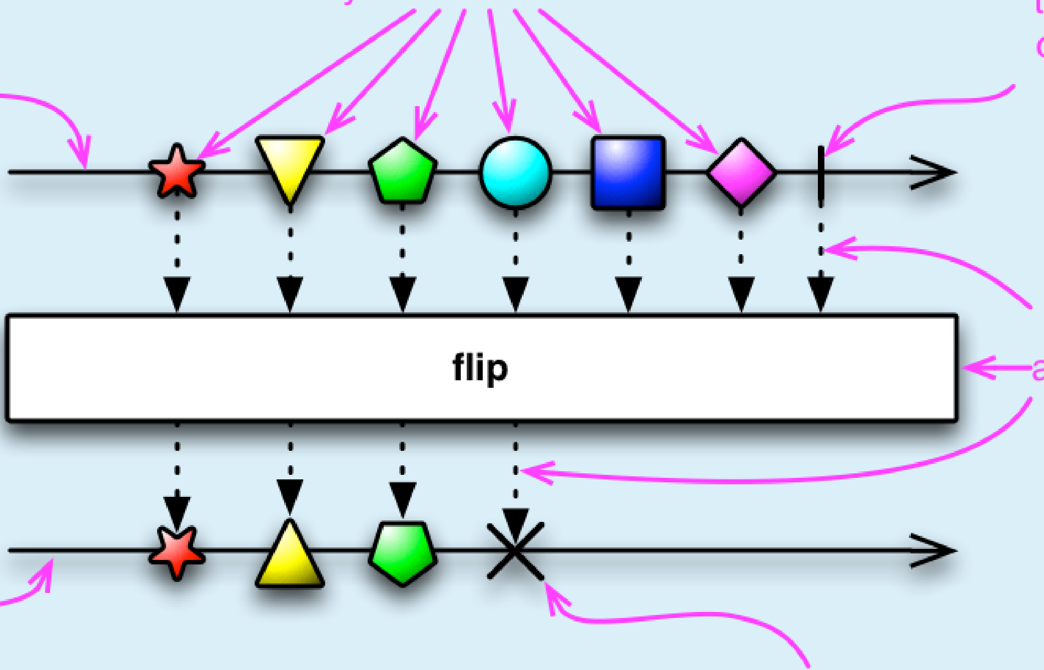
Source: http://reactivex.io/

Observable Design Pattern (Marble Diagram)

Building Block 1

- An *Observer subscribes* to an *Observable*.
- Then that observer reacts to whatever item or sequence of items the Observable *emits*.

Reactive Extensions (Rx)



This is the timeline of the Observable. Time flows from left to right.

These are items emitted by the Observable.

This vertical line indicates that the Observable has completed successfully.

These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.

This Observable is the result of the transformation.

If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html | http://rxmarbles.com

Observable / Observer Design Pattern

Building Block

1 & 2

- Allows for Concurrent Operations: the observer does not need to block while waiting for the observable to emit values
- Observer waits to receive values when the observable is ready to emit them
- Based on push rather than pull

Reactive Extensions (Rx)

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html

08 July 2017

What's missing from GOF Observer Pattern

Building Block 1

1. The ability for the producer to signal to the consumer that there is no more data available (a foreach loop on an Iterable completes and returns normally in such a case; an Observable calls its observer's onCompleted method)
2. The ability for the producer to signal to the consumer that an error has occurred (an Iterable throws an exception if an error takes place during iteration; an Observable calls its observer's onError method)
3. Multiple Thread Implementations and hiding those details.
4. Dozens of Operators to handle data.

Source:http://reactivex.io/intro.html

Compare Iterable Vs. Observable

Building Block 1

Observable is the asynchronous / push dual to the synchronous pull Iterable

Reactive Extensions (Rx)

Observables are:

- Composable: Easily chained together or combined
- Flexible: Can be used to emit:
  - A scalar value (network result)
  - Sequence (items in a list)
  - Infinite streams (weather sensor)
- Free from callback hell: Easy to transform one asynchronous stream into another

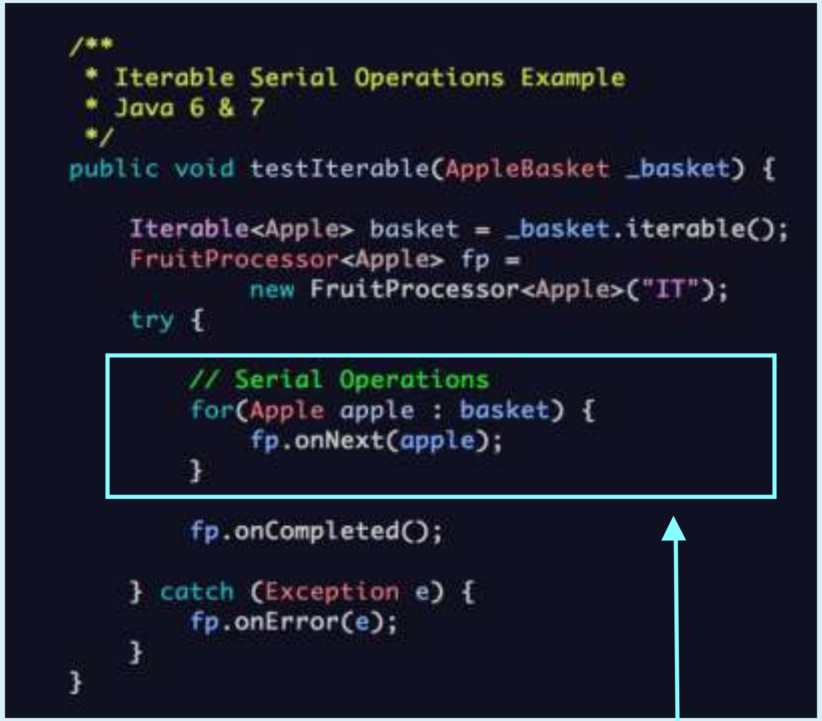
Event	Iterable (Pull)	Observable (Push)
Retrieve Data	T next()	onNext(T)
Discover Error	throws Exception	onError (Exception)
Complete	!hasNext()	onComplete()

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html

Comparison : Iterable / Streams / Observable

Building Block 1

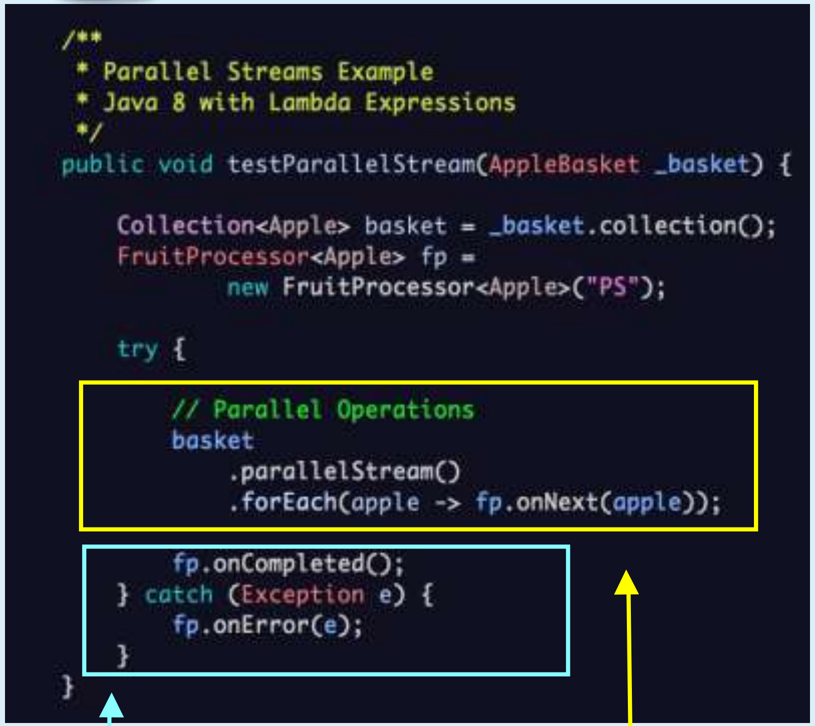
Java 6 – Blocking Call



```
/**
 * Iterable Serial Operations Example
 * Java 6 & 7
 */
public void testIterable(AppleBasket _basket) {

    Iterable<Apple> basket = _basket.iterable();
    FruitProcessor<Apple> fp =
            new FruitProcessor<Apple>("IT");
    try {

        // Serial Operations
        for(Apple apple : basket) {
            fp.onNext(apple);
        }

        fp.onCompleted();

    } catch (Exception e) {
        fp.onError(e);
    }
}
```

First Class Visitor (Consumer)
Serial Operations

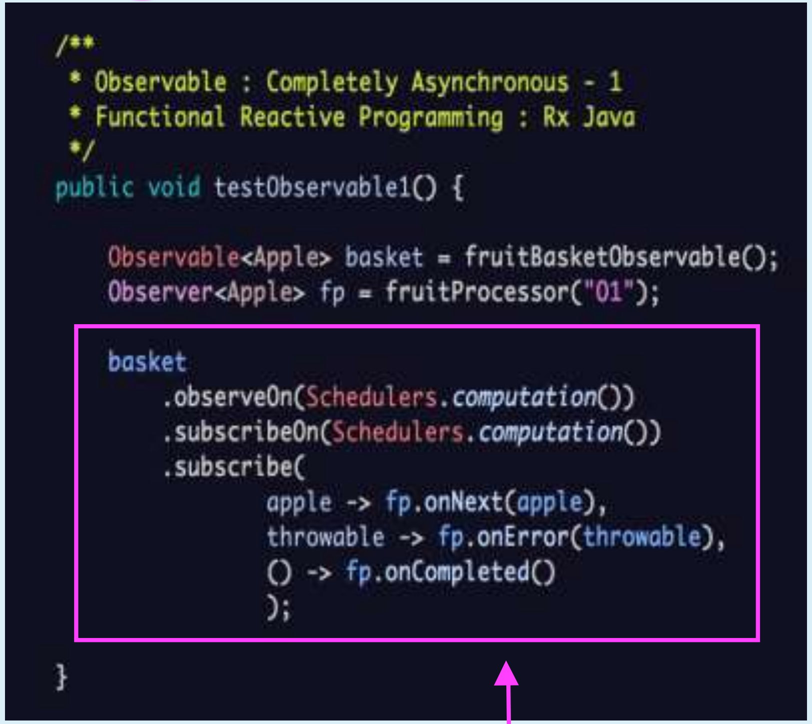
Java 8 – Blocking Call



```
/**
 * Parallel Streams Example
 * Java 8 with Lambda Expressions
 */
public void testParallelStream(AppleBasket _basket) {

    Collection<Apple> basket = _basket.collection();
    FruitProcessor<Apple> fp =
            new FruitProcessor<Apple>("PS");

    try {

        // Parallel Operations
        basket
            .parallelStream()
            .forEach(apple -> fp.onNext(apple));

        fp.onCompleted();
    } catch (Exception e) {
        fp.onError(e);
    }
}
```

Parallel Streams (10x Speed)
Still On Next, On Complete and
On Error are Serial Operations

Rx Java - Freedom



```
/**
 * Observable : Completely Asynchronous - 1
 * Functional Reactive Programming : Rx Java
 */
public void testObservable1() {

    Observable<Apple> basket = fruitBasketObservable();
    Observer<Apple> fp = fruitProcessor("01");

    basket
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
            apple -> fp.onNext(apple),
            throwable -> fp.onError(throwable),
            () -> fp.onCompleted()
        );

}
```

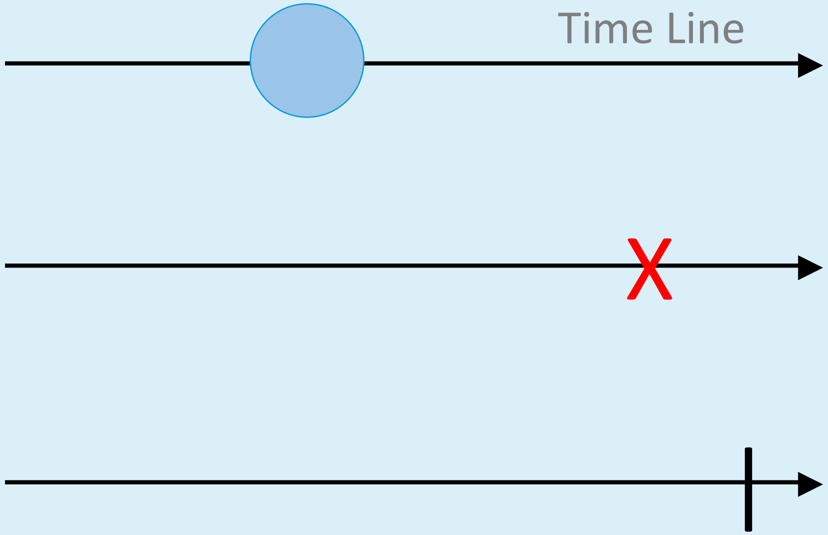
Completely Asynchronous
Operations

Observer<T> Contract

Building Block 2

Methods:

- onNext(T)
- onError(Throwable T)
- onCompleted()



onError / onCompleted called exactly once

Reactive Extensions (Rx)

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html

RxJava Scheduler Details

Building Block 3

- If you want to introduce multithreading into your cascade of Observable operators, you can do so by instructing those operators (or particular Observables) to operate on particular *Schedulers*.
- By default, an Observable and the chain of operators that you apply to it will do its work, and will notify its observers, on the same thread on which its Subscribe method is called.
- The SubscribeOn operator changes this behavior by specifying a different Scheduler on which the Observable should operate. TheObserveOn operator specifies a different Scheduler that the Observable will use to send notifications to its observers.

Reactive Extensions (Rx)

Source: http://reactivex.io/documentation/scheduler.html

RxJava Scheduler Threading Details

Building Block 3

	Scheduler	Purpose
1	Schedulers.computation ( )	Meant for computational work such as event-loops and callback processing; do not use this scheduler for I/O. (useSchedulers.io( ) instead) The number of threads, by default, is equal to the number of processors
2	Schedulers.from(executor)	Uses the specified Executor as a Scheduler
3	Schedulers.immediate ( )	Schedules work to begin immediately in the current thread
4	Schedulers.io ( )	Meant for I/O-bound work such as asynchronous performance of blocking I/O, this scheduler is backed by a thread-pool that will grow as needed; for ordinary computational work, switch to Schedulers.computation( ); Schedulers.io( ) by default is a CachedThreadScheduler, which is something like a new thread scheduler with thread caching
5	Schedulers.newThread ( )	Creates a new thread for each unit of work
6	Schedulers.trampoline ( )	Queues work to begin on the current thread after any already-queued work

Source: http://reactivex.io/documentation/scheduler.html

RxJava Operator : Frequently used

Building Block 4

	Content Filtering
1	Observable<T> filter (Func1<T, Boolean> Predicate)
2	Observable<T> skip (int num)
3	Observable<T> take (int num)
4	Observable<T> takeLast (int num)
5	Observable<T> elementAt (int index)
6	Observable<T> distinct ()
7	Observable<T> distinctUntilChanged ()

	Time Filtering
1	Observable<T> throttleFirst (long duration, TimeUnit unit)
2	Observable<T> throttleLast (long duration, TimeUnit unit)
3	Observable<T> timeout (long duration, TimeUnit unit)

Reactive Extensions (Rx)

	Transformation	Use
1	Observable<T> map (Func1<T, R> func)	Transforms Objects
2	Observable<T> flatMap (Func1<T, Observable<R>> func)	Transforms an Observable to another Observable
3	Observable<T> cast (Class<R> klass)	
4	Observable<GroupedObservable<K, T>> groupBy (Func1<T, K> keySelector)	

Source: http://reactivex.io/

The Problem? How Rx Helps!

Reactive Extensions (Rx)

Building Block 2

Apple Basket

Fruit Processor

Scenario

1. A Basket Full of Apples
2. Fruit Processor capacity (limit) is 3 Apples at a time.

The Problem

1. I don’t have time to wait till the Fruit Processor finishes it’s job.
2. I don’t mind having multiple Fruit Processors too. But that still doesn’t solve the problem

What I need

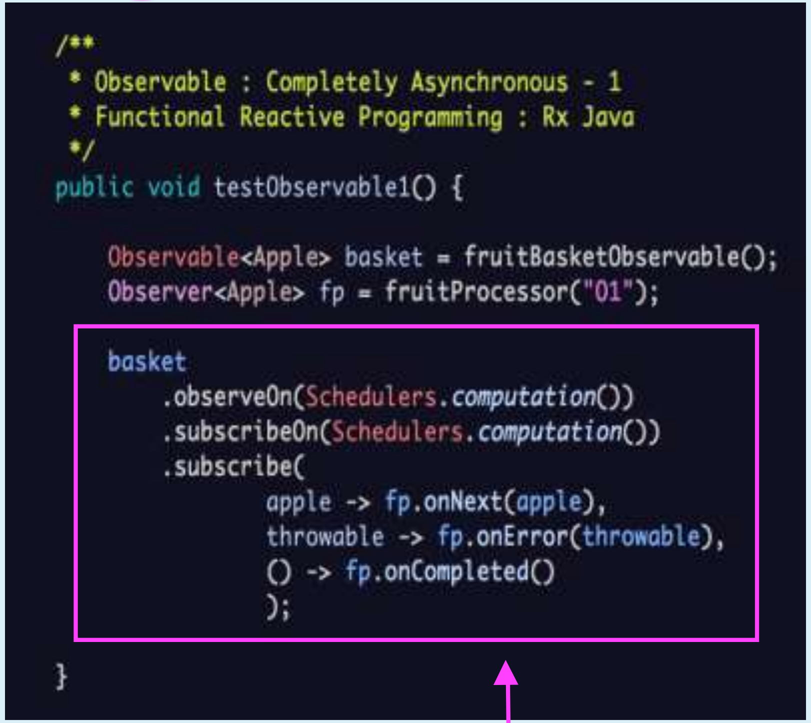
1. I want to start the Process and leave.
2. Once the processing is done, then the system should inform me.
3. If something goes wrong, then the system should inform me.

Me = the calling program

RxJava : Examples – Handling Fruit Processor

Building Block 1

Rx Java - Freedom



```
/**
 * Observable : Completely Asynchronous - 1
 * Functional Reactive Programming : Rx Java
 */
public void testObservable1() {

    Observable<Apple> basket = fruitBasketObservable();
    Observer<Apple> fp = fruitProcessor("01");

    basket
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
                apple -> fp.onNext(apple),
                throwable -> fp.onError(throwable),
                () -> fp.onCompleted()
                );

}
```

Completely Asynchronous Operations

Entities

1. Abstract Fruit
2. Fruit (Interface)
3. Apple
4. Orange
5. Grapes
6. Mixed Fruit (Aggregate Root)

Business Layer

1. Fruit Processor (Domain Service) - Observer
2. Fruit Basket Repository
3. Fruit Basket Observable Factory

Next Section focuses more on Operators and how to handle business logic in Asynchronous world!

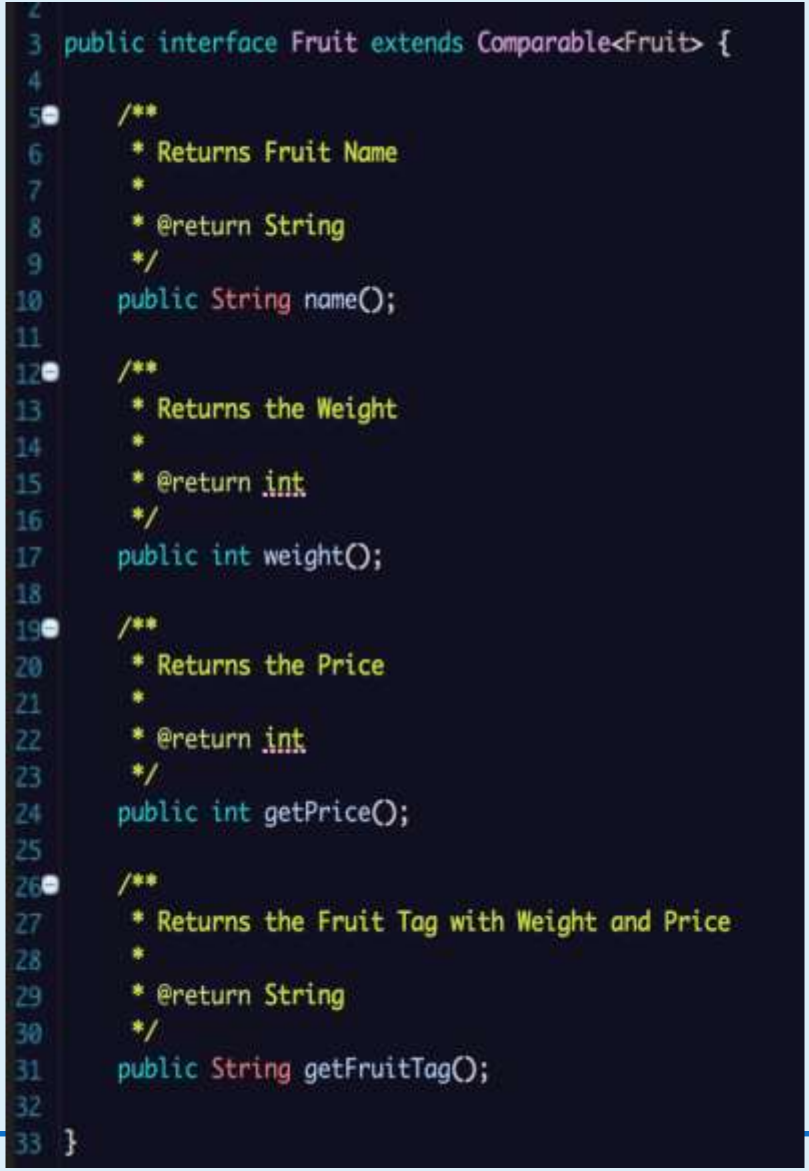
1. Rx Java Example : Observable / Observer / Scheduler Implementation
2. Rx Java Example 2 : Merge / Filters (on Weight) / Sort (on Price)

Source Code GIT:

RxJava : Entities & Repositories

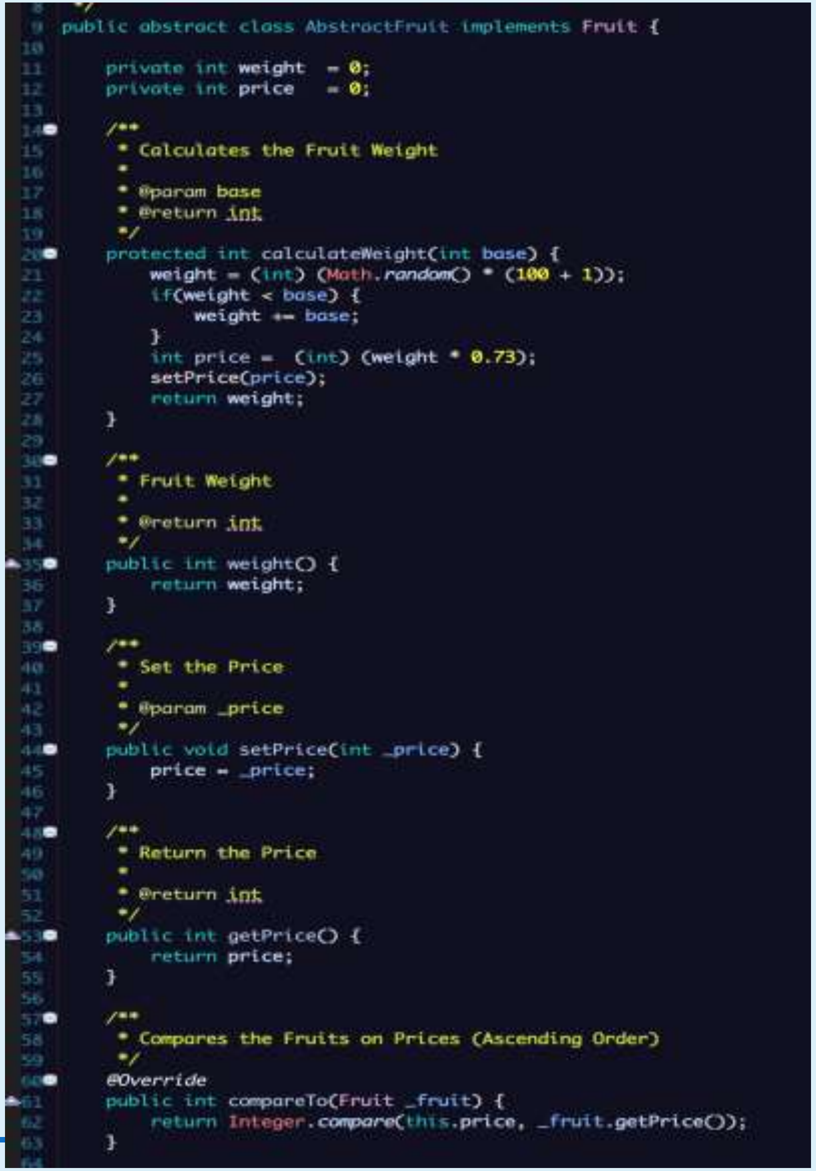
Building Block 1

Fruit Interface



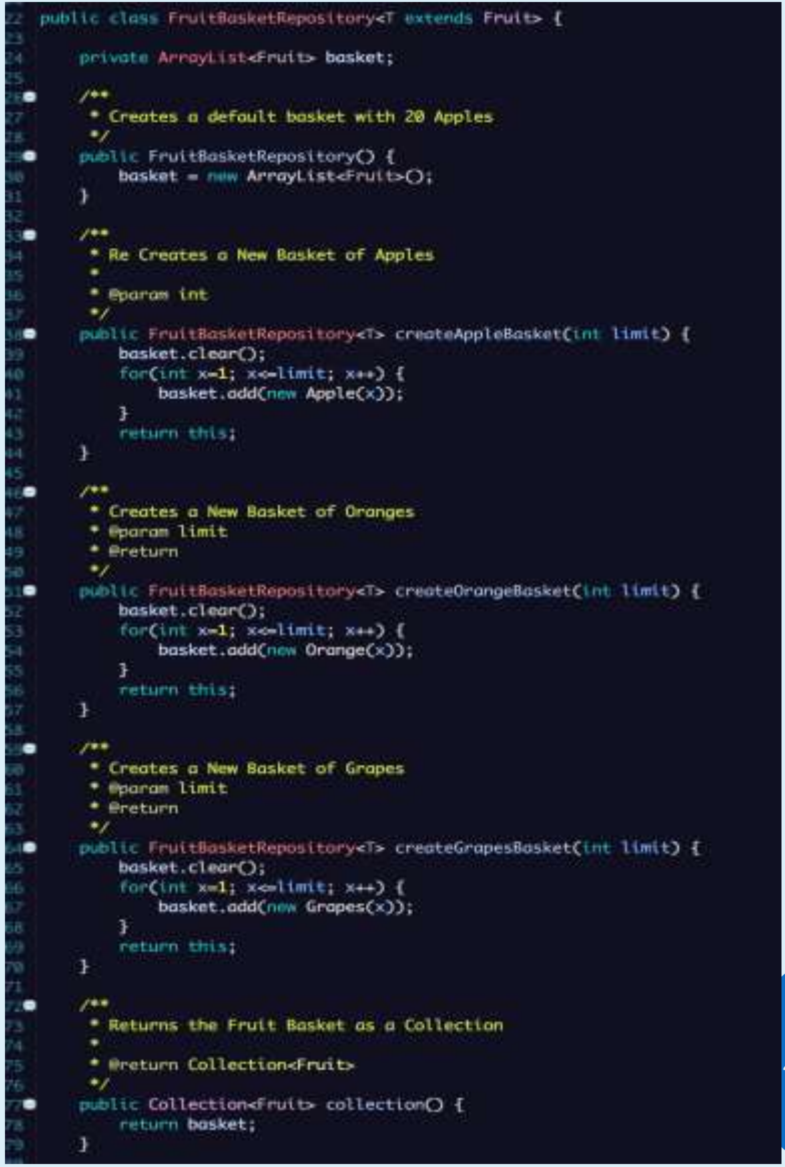
```java
public interface Fruit extends Comparable<Fruit> {

    /**
     * Returns Fruit Name
     *
     * @return String
     */
    public String name();

    /**
     * Returns the Weight
     *
     * @return int
     */
    public int weight();

    /**
     * Returns the Price
     *
     * @return int
     */
    public int getPrice();

    /**
     * Returns the Fruit Tag with Weight and Price
     *
     * @return String
     */
    public String getFruitTag();

}
```

Abstract Fruit



```java
 */
public abstract class AbstractFruit implements Fruit {

    private int weight  = 0;
    private int price   = 0;

    /**
     * Calculates the Fruit Weight
     *
     * @param base
     * @return int
     */
    protected int calculateWeight(int base) {
        weight = (int) (Math.random() * (100 + 1));
        if(weight < base) {
            weight += base;
        }
        int price =  (int) (weight * 0.73);
        setPrice(price);
        return weight;
    }

    /**
     * Fruit Weight
     *
     * @return int
     */
    public int weight() {
        return weight;
    }

    /**
     * Set the Price
     *
     * @param _price
     */
    public void setPrice(int _price) {
        price = _price;
    }

    /**
     * Return the Price
     *
     * @return int
     */
    public int getPrice() {
        return price;
    }

    /**
     * Compares the Fruits on Prices (Ascending Order)
     */
    @Override
    public int compareTo(Fruit _fruit) {
        return Integer.compare(this.price, _fruit.getPrice());
    }
```

Fruit Repository



```java
public class FruitBasketRepository<T extends Fruit> {

    private ArrayList<Fruit> basket;

    /**
     * Creates a default basket with 20 Apples
     */
    public FruitBasketRepository() {
        basket = new ArrayList<Fruit>();
    }

    /**
     * Re Creates a New Basket of Apples
     *
     * @param int
     */
    public FruitBasketRepository<T> createAppleBasket(int limit) {
        basket.clear();
        for(int x=1; x<=limit; x++) {
            basket.add(new Apple(x));
        }
        return this;
    }

    /**
     * Creates a New Basket of Oranges
     * @param limit
     * @return
     */
    public FruitBasketRepository<T> createOrangeBasket(int limit) {
        basket.clear();
        for(int x=1; x<=limit; x++) {
            basket.add(new Orange(x));
        }
        return this;
    }

    /**
     * Creates a New Basket of Grapes
     * @param limit
     * @return
     */
    public FruitBasketRepository<T> createGrapesBasket(int limit) {
        basket.clear();
        for(int x=1; x<=limit; x++) {
            basket.add(new Grapes(x));
        }
        return this;
    }

    /**
     * Returns the Fruit Basket as a Collection
     *
     * @return Collection<Fruit>
     */
    public Collection<Fruit> collection() {
        return basket;
    }
```

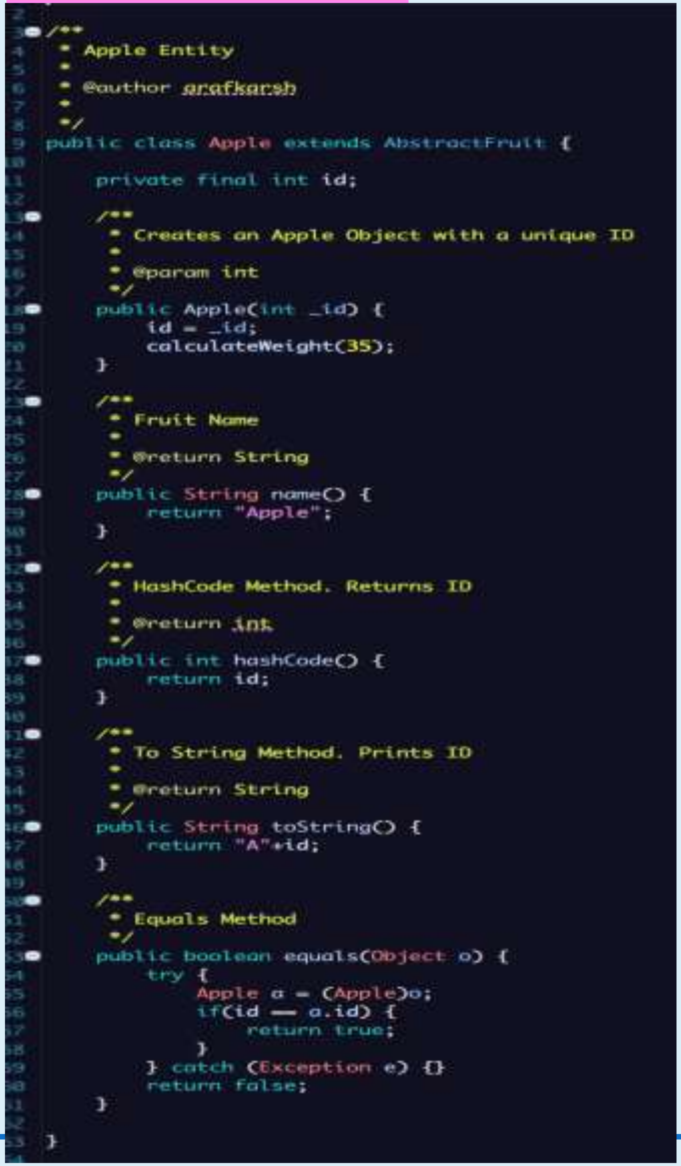
Source Code GIT:

08 July 2017

RxJava : Entities

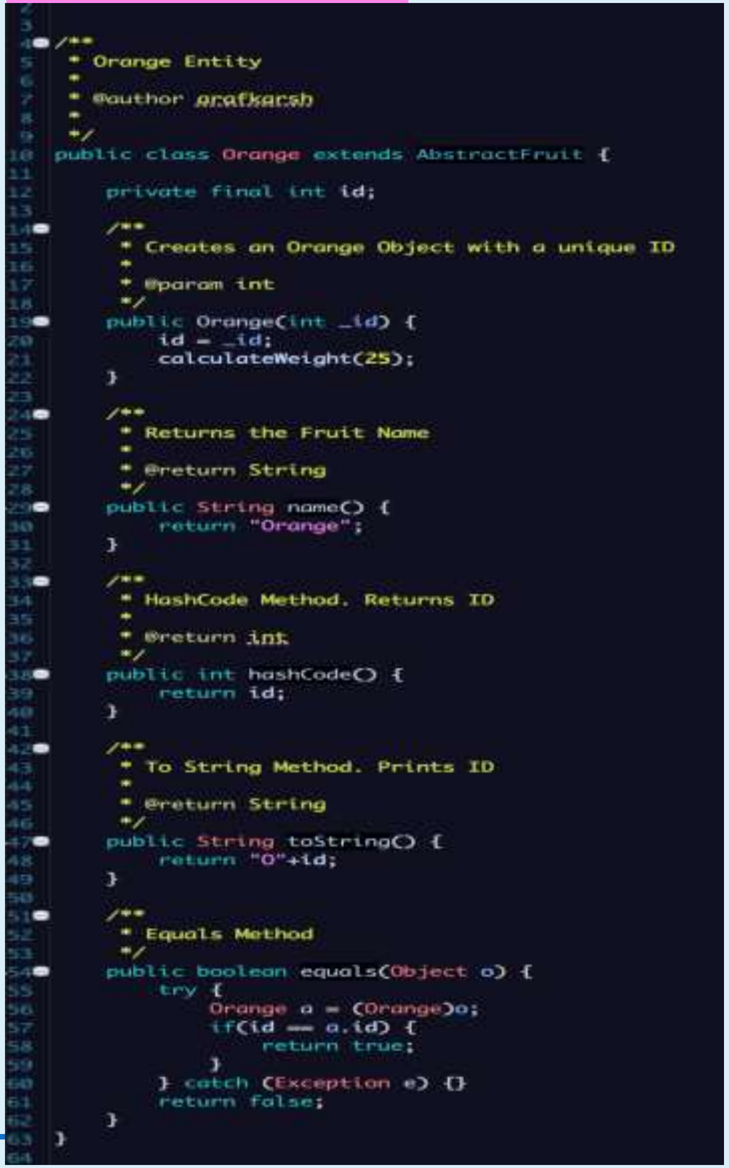
Building Block 1

Apple



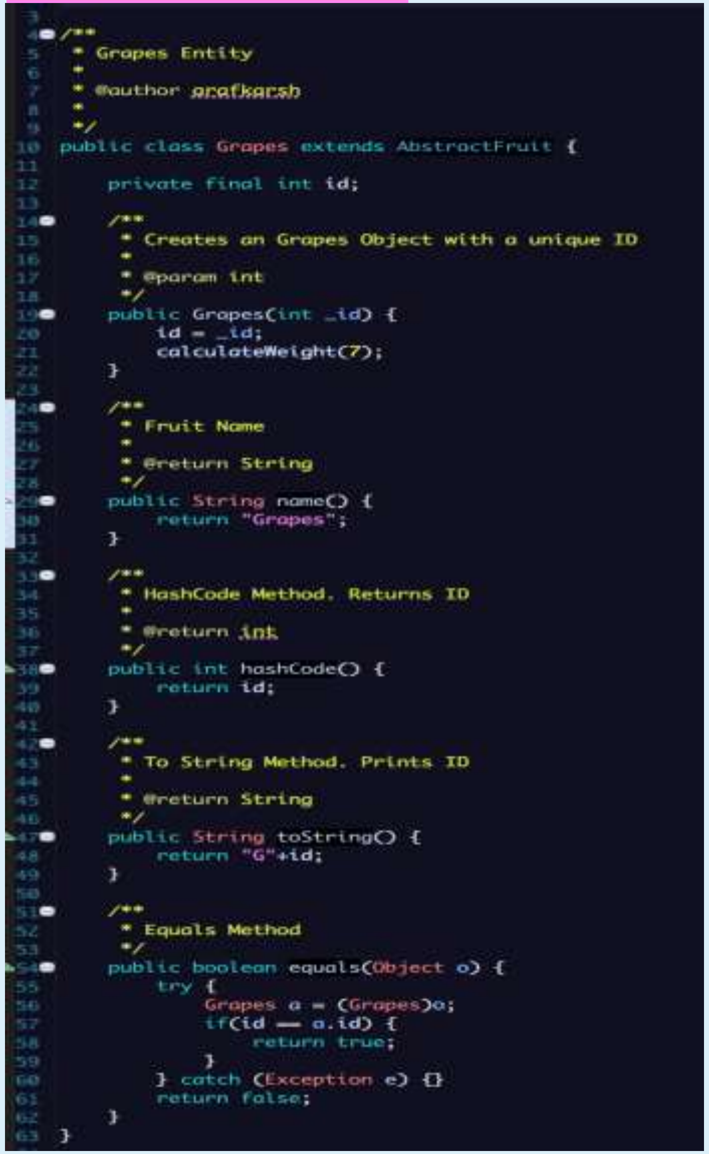
```
/**
 * Apple Entity
 *
 * @author arafkarsh
 *
 */
public class Apple extends AbstractFruit {

    private final int id;

    /**
     * Creates an Apple Object with a unique ID
     *
     * @param int
     */
    public Apple(int _id) {
        id = _id;
        calculateWeight(35);
    }

    /**
     * Fruit Name
     *
     * @return String
     */
    public String name() {
        return "Apple";
    }

    /**
     * HashCode Method. Returns ID
     *
     * @return int
     */
    public int hashCode() {
        return id;
    }

    /**
     * To String Method. Prints ID
     *
     * @return String
     */
    public String toString() {
        return "A"+id;
    }

    /**
     * Equals Method
     */
    public boolean equals(Object o) {
        try {
            Apple a = (Apple)o;
            if(id == a.id) {
                return true;
            }
        } catch (Exception e) {}
        return false;
    }

}
```

Orange



```
/**
 * Orange Entity
 *
 * @author arafkarsh
 *
 */
public class Orange extends AbstractFruit {

    private final int id;

    /**
     * Creates an Orange Object with a unique ID
     *
     * @param int
     */
    public Orange(int _id) {
        id = _id;
        calculateWeight(25);
    }

    /**
     * Returns the Fruit Name
     *
     * @return String
     */
    public String name() {
        return "Orange";
    }

    /**
     * HashCode Method. Returns ID
     *
     * @return int
     */
    public int hashCode() {
        return id;
    }

    /**
     * To String Method. Prints ID
     *
     * @return String
     */
    public String toString() {
        return "O"+id;
    }

    /**
     * Equals Method
     */
    public boolean equals(Object o) {
        try {
            Orange a = (Orange)o;
            if(id == a.id) {
                return true;
            }
        } catch (Exception e) {}
        return false;
    }
}
```

Grapes



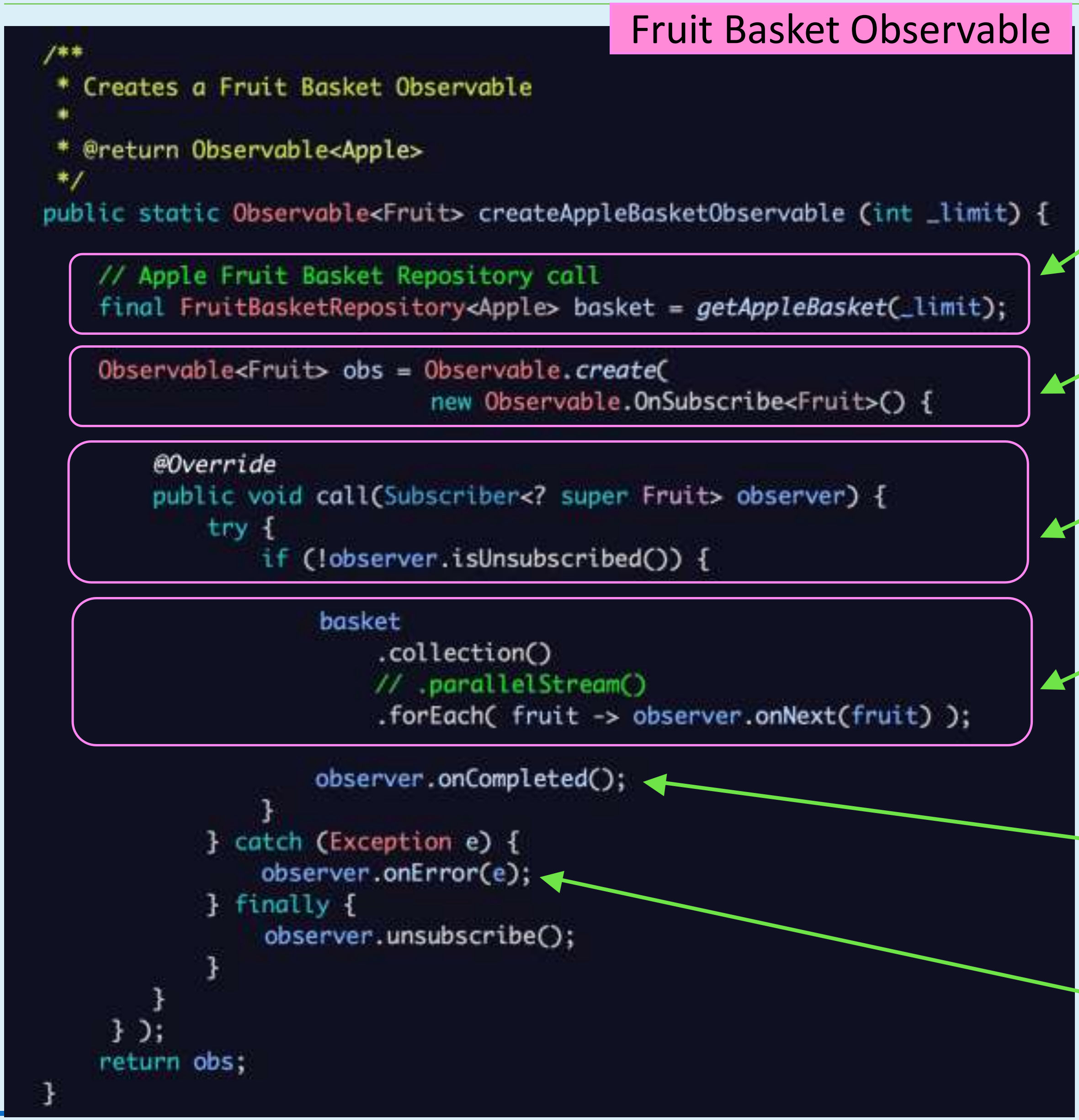
```
/**
 * Grapes Entity
 *
 * @author arafkarsh
 *
 */
public class Grapes extends AbstractFruit {

    private final int id;

    /**
     * Creates an Grapes Object with a unique ID
     *
     * @param int
     */
    public Grapes(int _id) {
        id = _id;
        calculateWeight(7);
    }

    /**
     * Fruit Name
     *
     * @return String
     */
    public String name() {
        return "Grapes";
    }

    /**
     * HashCode Method. Returns ID
     *
     * @return int
     */
    public int hashCode() {
        return id;
    }

    /**
     * To String Method. Prints ID
     *
     * @return String
     */
    public String toString() {
        return "G"+id;
    }

    /**
     * Equals Method
     */
    public boolean equals(Object o) {
        try {
            Grapes a = (Grapes)o;
            if(id == a.id) {
                return true;
            }
        } catch (Exception e) {}
        return false;
    }
}
```

Source Code GIT:

RxJava : Apple Fruit Basket : Observable

Building Block 1

Fruit Basket Observable



```
/**
 * Creates a Fruit Basket Observable
 *
 * @return Observable<Apple>
 */
public static Observable<Fruit> createAppleBasketObservable (int _limit) {

    // Apple Fruit Basket Repository call
    final FruitBasketRepository<Apple> basket = getAppleBasket(_limit);

    Observable<Fruit> obs = Observable.create(
                            new Observable.OnSubscribe<Fruit>() {

        @Override
        public void call(Subscriber<? super Fruit> observer) {
            try {
                if (!observer.isUnsubscribed()) {

                    basket
                        .collection()
                        // .parallelStream()
                        .forEach( fruit -> observer.onNext(fruit) );

                    observer.onCompleted();
                }
            } catch (Exception e) {
                observer.onError(e);
            } finally {
                observer.unsubscribe();
            }
        }
    } );
    return obs;
}
```

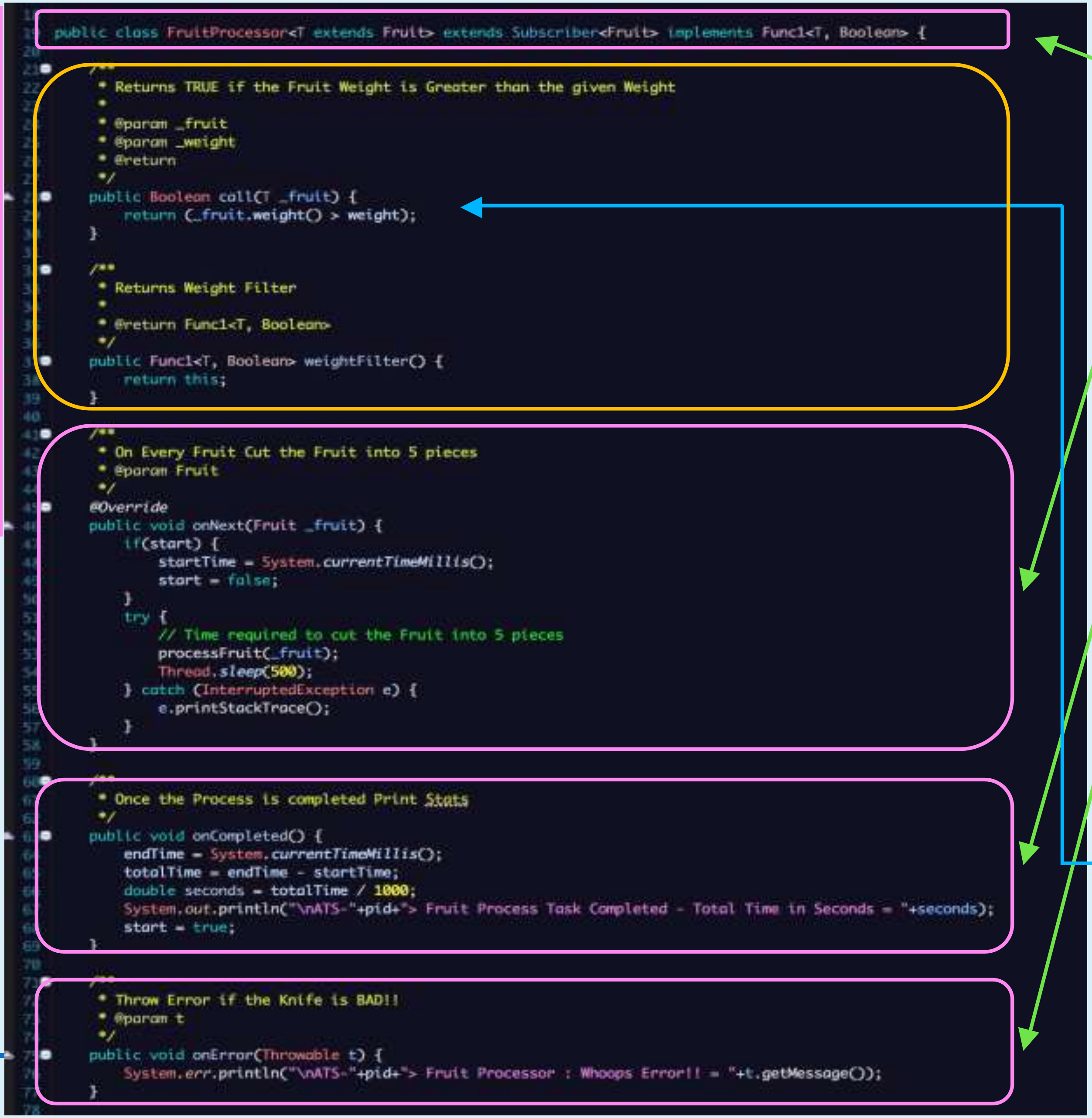
1. This function calls the Apple Basket Repository function to load the data.
2. Observable is created using the collection from the data returned by the Repository
3. Call method checks if the Observer is subscribed
4. If Yes for Each Apple it calls the Observer to do the action = observer.onNext (fruit)
5. Once the process is completed
6. Observable will call the on Completed method in Observer.
7. If an error occurs then Observable will call the on Error method in Observer

Source Code GIT:

RxJava : Fruit Processor : Observer

Building Block 1

Fruit Processor Observer



```
public class FruitProcessor<T extends Fruit> extends Subscriber<Fruit> implements Func1<T, Boolean> {

    /**
     * Returns TRUE if the Fruit Weight is Greater than the given Weight
     *
     * @param _fruit
     * @param _weight
     * @return
     */
    public Boolean call(T _fruit) {
        return (_fruit.weight() > weight);
    }

    /**
     * Returns Weight Filter
     *
     * @return Func1<T, Boolean>
     */
    public Func1<T, Boolean> weightFilter() {
        return this;
    }

    /**
     * On Every Fruit Cut the Fruit into 5 pieces
     * @param Fruit
     */
    @Override
    public void onNext(Fruit _fruit) {
        if(start) {
            startTime = System.currentTimeMillis();
            start = false;
        }
        try {
            // Time required to cut the Fruit into 5 pieces
            processFruit(_fruit);
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    /**
     * Once the Process is completed Print Stats
     */
    public void onCompleted() {
        endTime = System.currentTimeMillis();
        totalTime = endTime - startTime;
        double seconds = totalTime / 1000;
        System.out.println("\nATS-"+pid+"> Fruit Process Task Completed - Total Time in Seconds = "+seconds);
        start = true;
    }

    /**
     * Throw Error if the Knife is BAD!!
     * @param t
     */
    public void onError(Throwable t) {
        System.err.println("\nATS-"+pid+"> Fruit Processor : Whoops Error!! = "+t.getMessage());
    }
```

1. Fruit Processor implements Subscriber (Observer) interface
2. On Next Method Implementation. Main Business logic is done over here. Observable will call on Next and pass the Fruit Object for processing.
3. On Complete Method. Observable will call this method once the processing is done.
4. If Any error occurred then Observable will call on Error method and pass on the Exception.
5. call() implements the Func1 for filtering. Weight is passed as a parameter in Fruit Processor Constructor.

Fruit Processor Handles the Business Logic

Source Code GIT:

08 July 2017

Bringing all the pieces together

Observable / Observer / Schedulers / Operators

Apple Basket

Fruit Processor

RxJava Operator : Merge 3 Streams

Reactive Extensions (Rx)

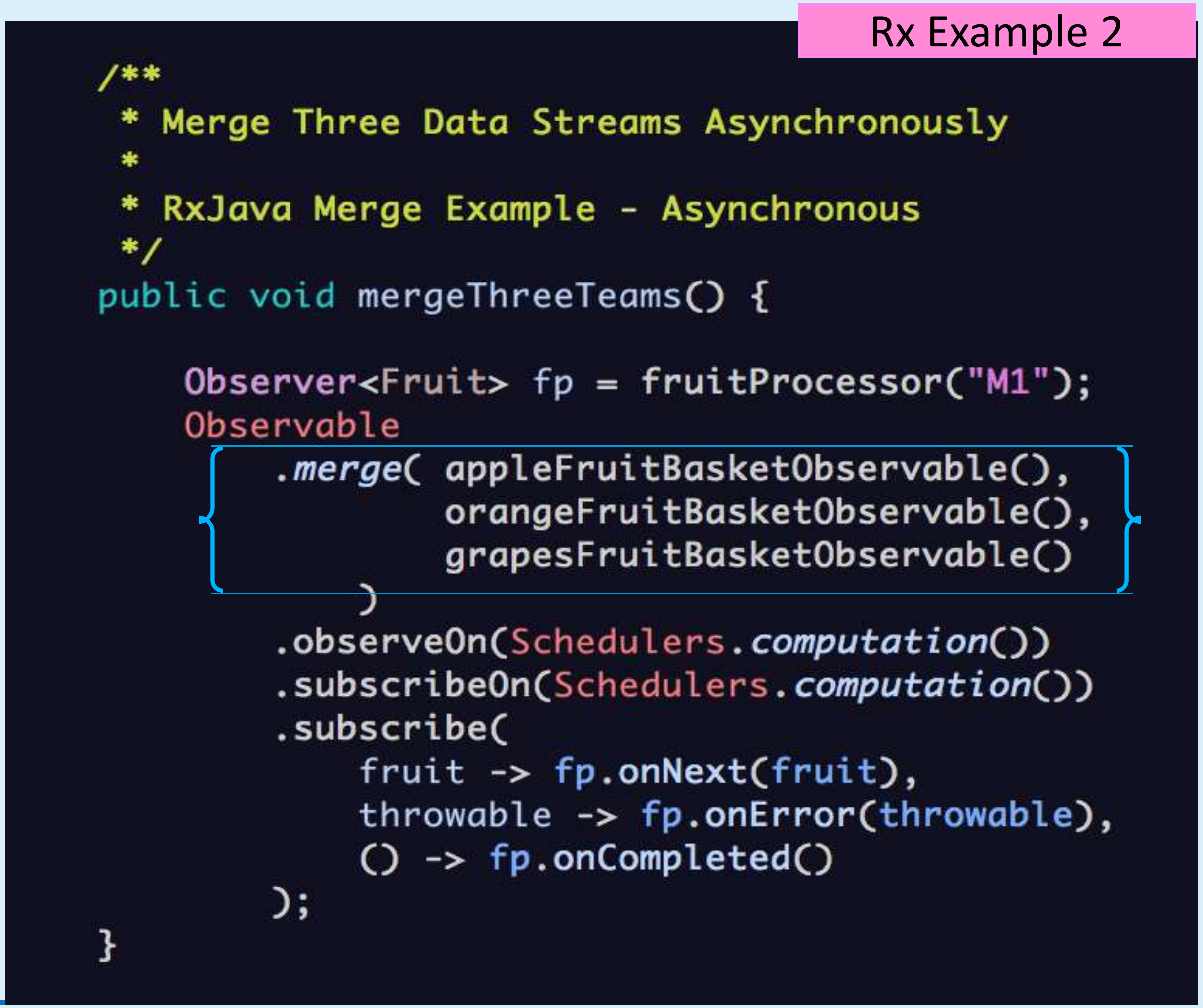
Building Block 4

08 July 2017

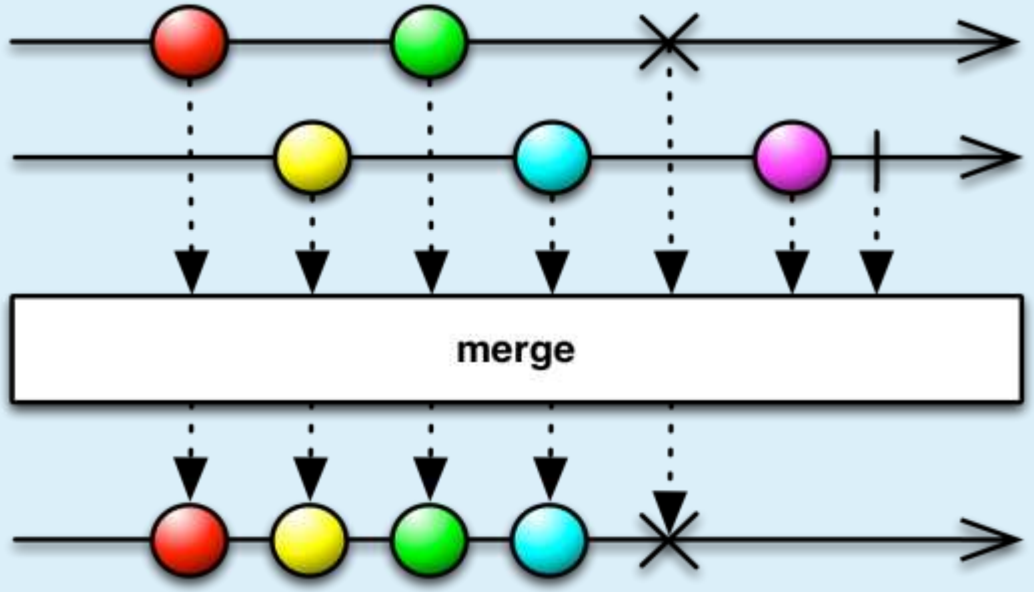
Objective:

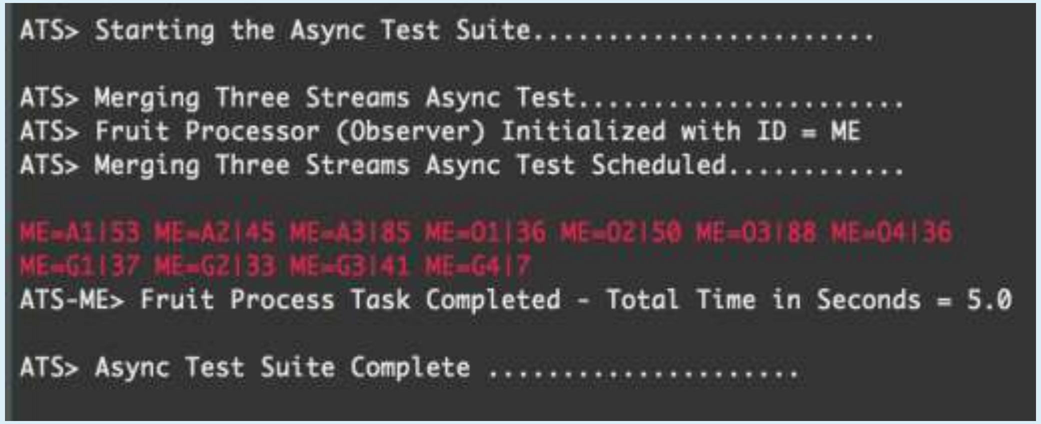
Merge Three Data Streams (Apple Fruit Basket, Orange Fruit Basket & Grapes Fruit Basket) into a Single Stream and do the Processing on that new Stream Asynchronously

Rx Example 2



```
/**
 * Merge Three Data Streams Asynchronously
 *
 * RxJava Merge Example - Asynchronous
 */
public void mergeThreeTeams() {

    Observer<Fruit> fp = fruitProcessor("M1");
    Observable
        .merge( appleFruitBasketObservable(),
                orangeFruitBasketObservable(),
                grapesFruitBasketObservable()
        )
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
            fruit -> fp.onNext(fruit),
            throwable -> fp.onError(throwable),
            () -> fp.onCompleted()
        );
}
```





```
ATS> Starting the Async Test Suite......................

ATS> Merging Three Streams Async Test.....................
ATS> Fruit Processor (Observer) Initialized with ID = ME
ATS> Merging Three Streams Async Test Scheduled...........

ME=A1|53 ME=A2|45 ME=A3|85 ME=O1|36 ME=O2|50 ME=O3|88 ME=O4|36
ME=G1|37 ME=G2|33 ME=G3|41 ME=G4|7
ATS-ME> Fruit Process Task Completed - Total Time in Seconds = 5.0

ATS> Async Test Suite Complete ....................
```

A1 is the Apple Series, O1 is the Orange Series & G1 is the Grapes Series

RxJava Operator : Merge / Filter Streams

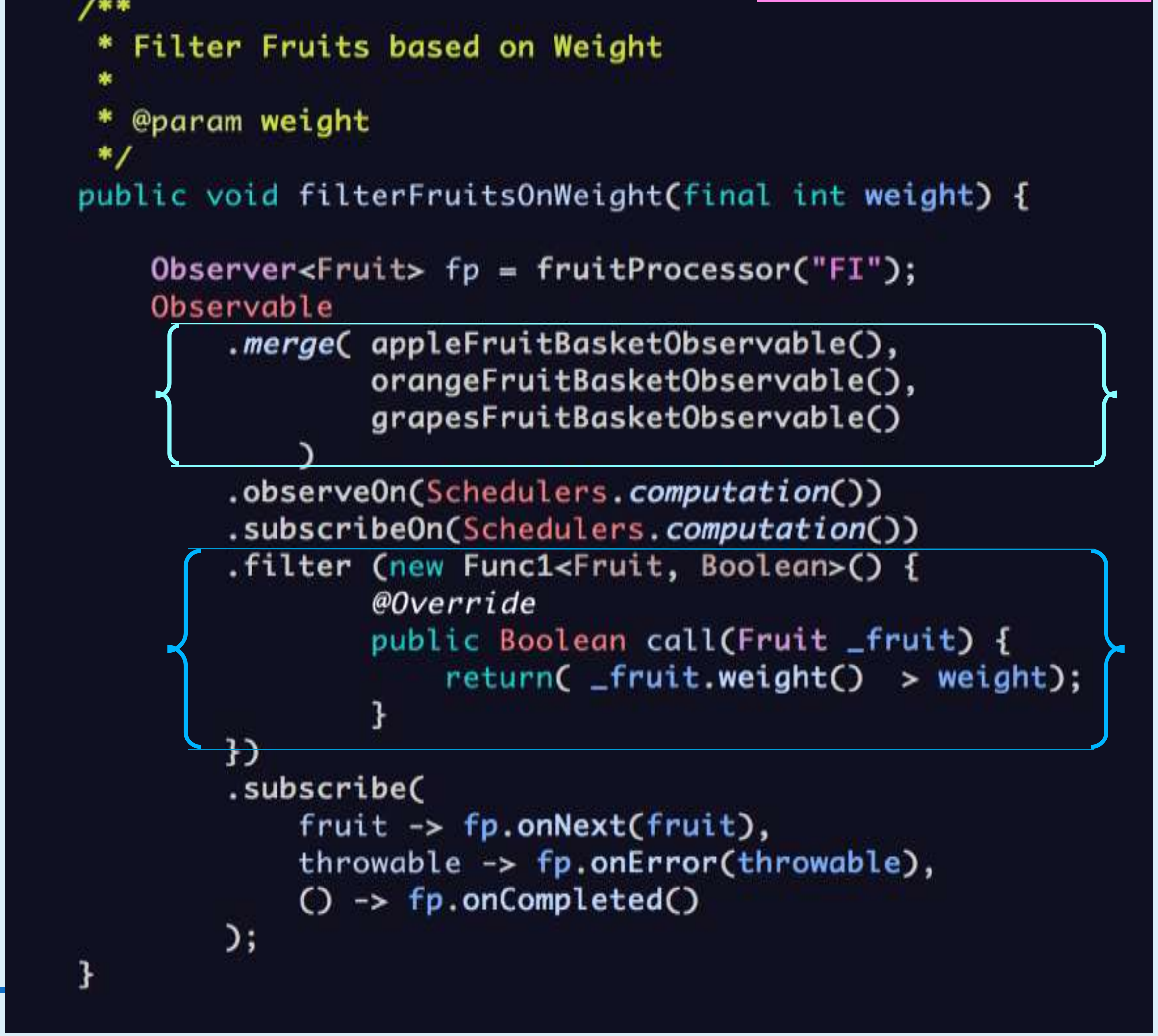
Reactive Extensions (Rx)

Building Block 4

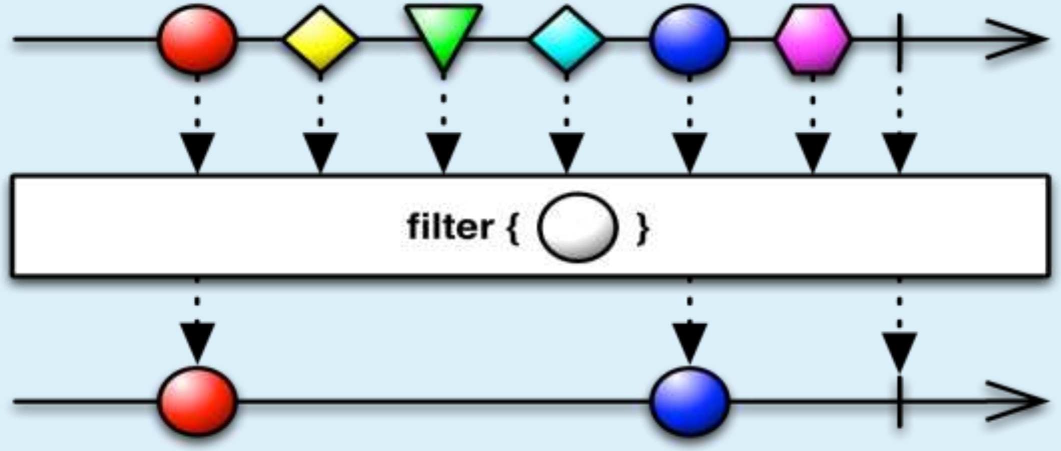
Objective:

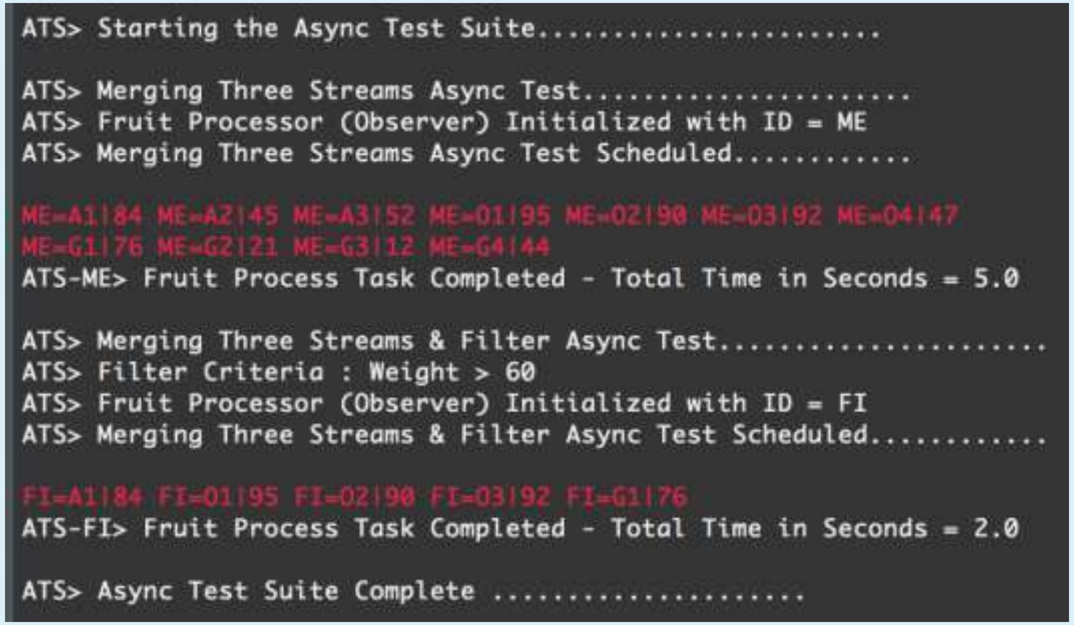
From the previous three Merged streams Filter Fruits which weighs more than 50 grams.

Rx Example 2



```
/**
 * Filter Fruits based on Weight
 *
 * @param weight
 */
public void filterFruitsOnWeight(final int weight) {

    Observer<Fruit> fp = fruitProcessor("FI");
    Observable
        .merge( appleFruitBasketObservable(),
                orangeFruitBasketObservable(),
                grapesFruitBasketObservable()
        )
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .filter (new Func1<Fruit, Boolean>() {
                @Override
                public Boolean call(Fruit _fruit) {
                    return( _fruit.weight()  > weight);
                }
        })
        .subscribe(
            fruit -> fp.onNext(fruit),
            throwable -> fp.onError(throwable),
            () -> fp.onCompleted()
        );
}
```





```
ATS> Starting the Async Test Suite.......................

ATS> Merging Three Streams Async Test......................
ATS> Fruit Processor (Observer) Initialized with ID = ME
ATS> Merging Three Streams Async Test Scheduled...........

ME=A1|84 ME=A2|45 ME=A3|52 ME=O1|95 ME=O2|90 ME=O3|92 ME=O4|47
ME=G1|76 ME=G2|21 ME=G3|12 ME=G4|44
ATS-ME> Fruit Process Task Completed - Total Time in Seconds = 5.0

ATS> Merging Three Streams & Filter Async Test.....................
ATS> Filter Criteria : Weight > 60
ATS> Fruit Processor (Observer) Initialized with ID = FI
ATS> Merging Three Streams & Filter Async Test Scheduled...........

FI=A1|84 FI=O1|95 FI=O2|90 FI=O3|92 FI=G1|76
ATS-FI> Fruit Process Task Completed - Total Time in Seconds = 2.0

ATS> Async Test Suite Complete ....................
```

08 July 2017

RxJava Operator : Merge / Filter / Sort

Reactive Extensions (Rx)

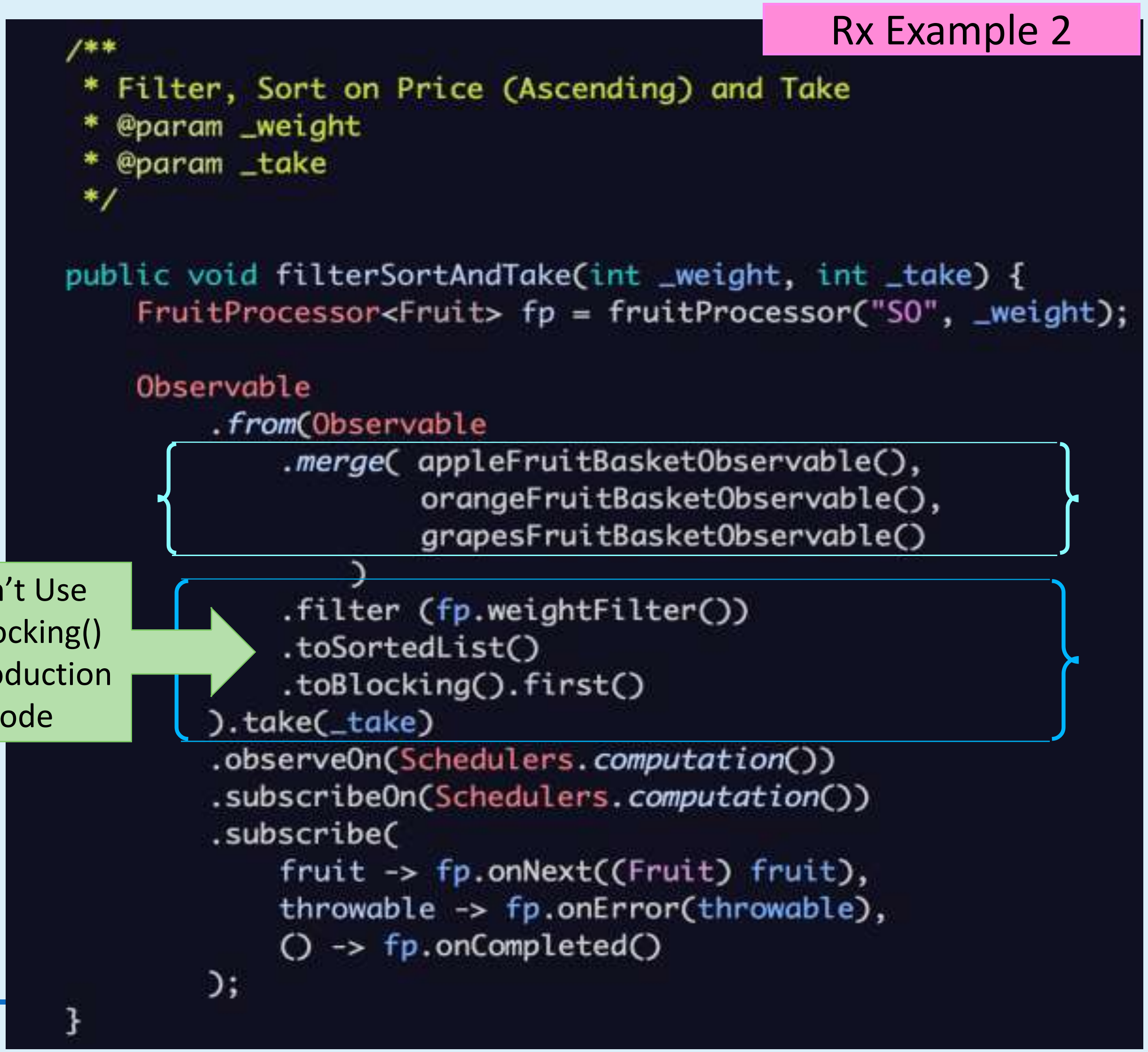
Building Block 4

Objective:

From the previous three Merged streams Filter Fruits which weighs more than 50 grams and sort on Price (Asc).

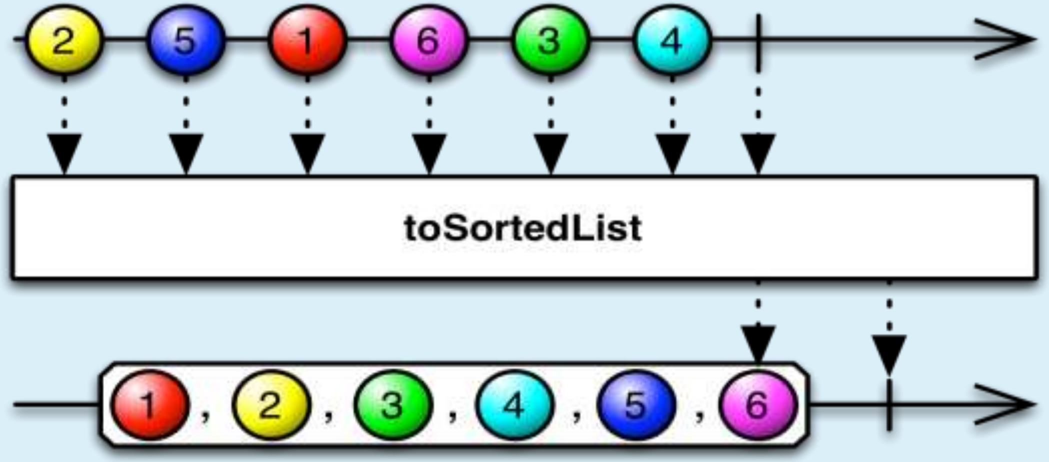
08 July 2017

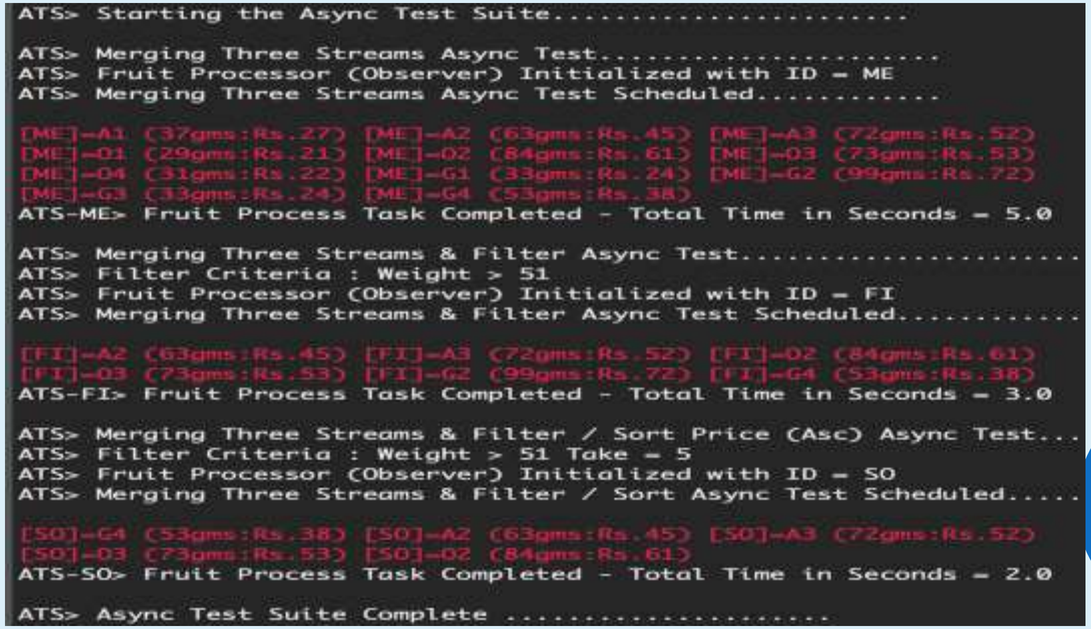
Rx Example 2



```
/**
 * Filter, Sort on Price (Ascending) and Take
 * @param _weight
 * @param _take
 */

public void filterSortAndTake(int _weight, int _take) {
    FruitProcessor<Fruit> fp = fruitProcessor("SO", _weight);

    Observable
        .from(Observable
            .merge( appleFruitBasketObservable(),
                    orangeFruitBasketObservable(),
                    grapesFruitBasketObservable()
            )
            .filter (fp.weightFilter())
            .toSortedList()
            .toBlocking().first()
        ).take(_take)
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
            fruit -> fp.onNext((Fruit) fruit),
            throwable -> fp.onError(throwable),
            () -> fp.onCompleted()
        );
}
```

Don't Use toBlocking() in Production Code





```
ATS> Starting the Async Test Suite.......................

ATS> Merging Three Streams Async Test.......................
ATS> Fruit Processor (Observer) Initialized with ID - ME
ATS> Merging Three Streams Async Test Scheduled...........

[ME]-A1 (37gms:Rs.27) [ME]-A2 (63gms:Rs.45) [ME]-A3 (72gms:Rs.52)
[ME]-O1 (29gms:Rs.21) [ME]-O2 (84gms:Rs.61) [ME]-O3 (73gms:Rs.53)
[ME]-O4 (31gms:Rs.22) [ME]-G1 (33gms:Rs.24) [ME]-G2 (99gms:Rs.72)
[ME]-G3 (33gms:Rs.24) [ME]-G4 (53gms:Rs.38)
ATS-ME> Fruit Process Task Completed - Total Time in Seconds - 5.0

ATS> Merging Three Streams & Filter Async Test.......................
ATS> Filter Criteria : Weight > 51
ATS> Fruit Processor (Observer) Initialized with ID - FI
ATS> Merging Three Streams & Filter Async Test Scheduled...........

[FI]-A2 (63gms:Rs.45) [FI]-A3 (72gms:Rs.52) [FI]-O2 (84gms:Rs.61)
[FI]-O3 (73gms:Rs.53) [FI]-G2 (99gms:Rs.72) [FI]-G4 (53gms:Rs.38)
ATS-FI> Fruit Process Task Completed - Total Time in Seconds - 3.0

ATS> Merging Three Streams & Filter / Sort Price (Asc) Async Test...
ATS> Filter Criteria : Weight > 51 Take - 5
ATS> Fruit Processor (Observer) Initialized with ID - SO
ATS> Merging Three Streams & Filter / Sort Async Test Scheduled....

[SO]-G4 (53gms:Rs.38) [SO]-A2 (63gms:Rs.45) [SO]-A3 (72gms:Rs.52)
[SO]-O3 (73gms:Rs.53) [SO]-O2 (84gms:Rs.61)
ATS-SO> Fruit Process Task Completed - Total Time in Seconds - 2.0

ATS> Async Test Suite Complete .......................
```

RxJava Operator : Filter / Sort / FlatMap

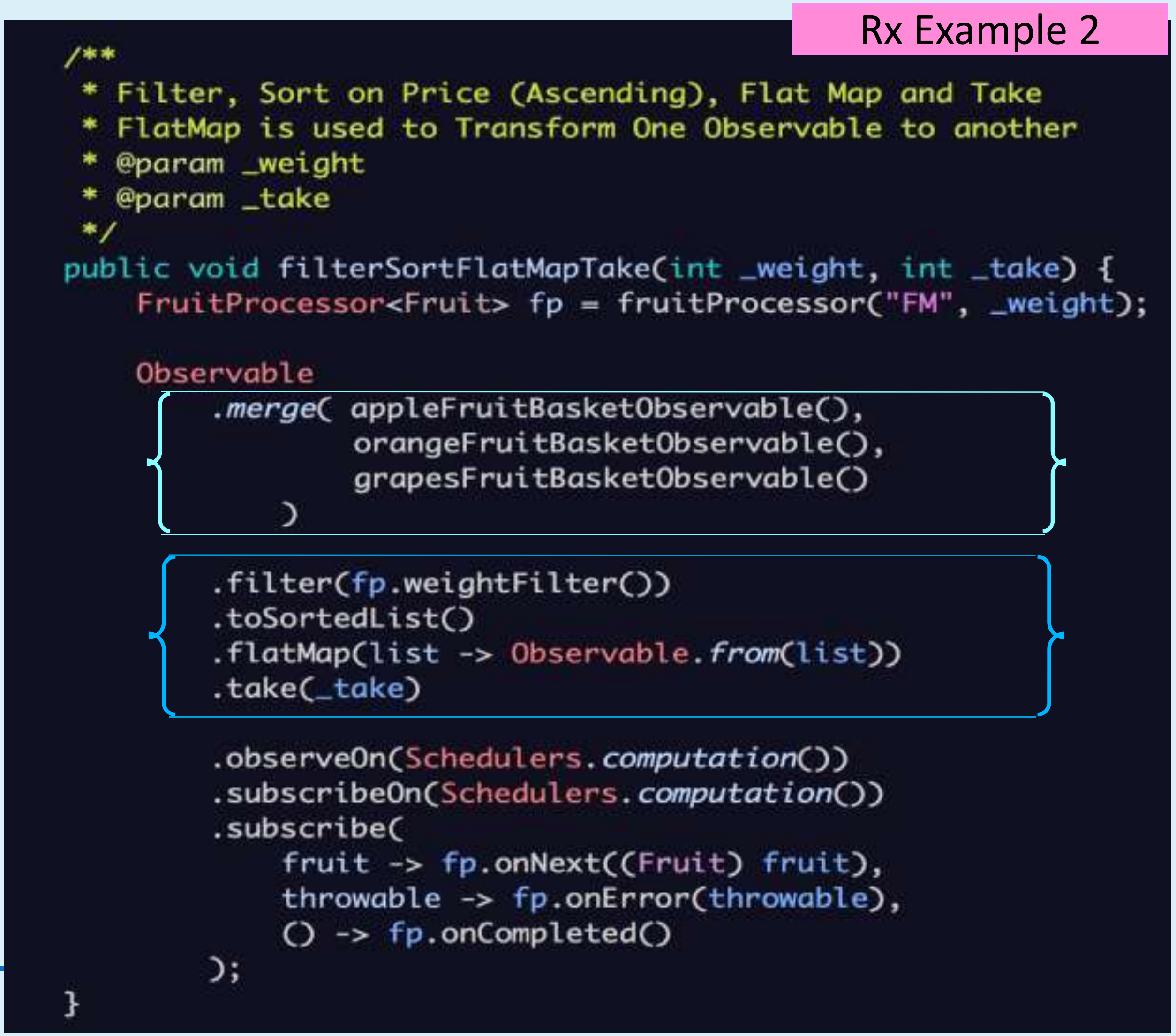
Reactive Extensions (Rx)

Building Block 4

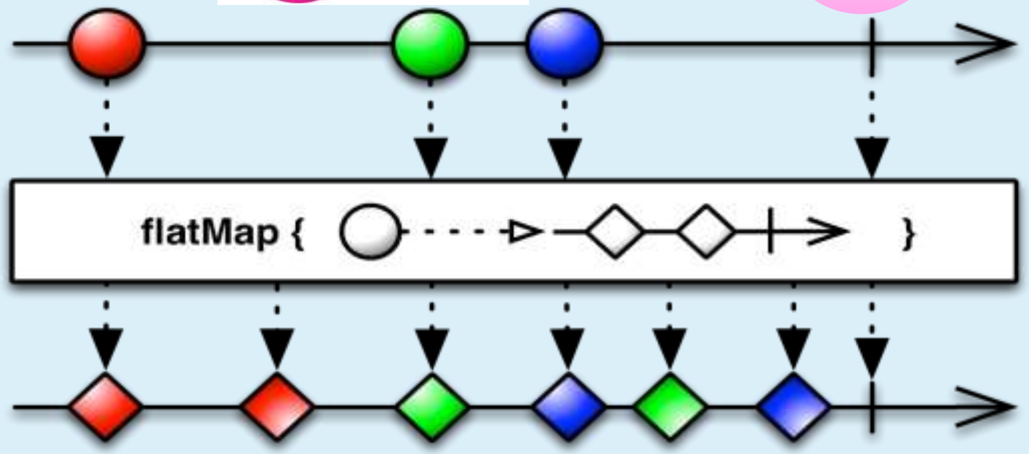
Objective:

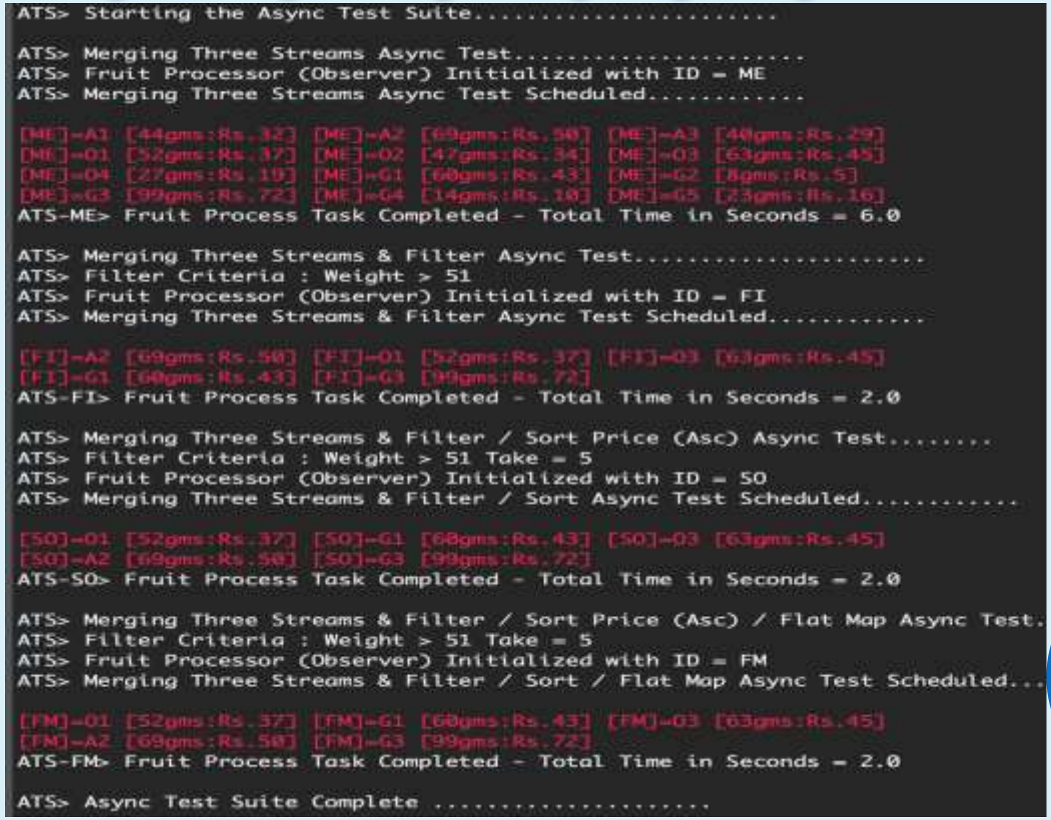
toSortedList() returns an Observable with a single List containing Fruits.
Using FlatMap to Transform Observable <List> to Observable <Fruit>

Rx Example 2



```
/**
 * Filter, Sort on Price (Ascending), Flat Map and Take
 * FlatMap is used to Transform One Observable to another
 * @param _weight
 * @param _take
 */
public void filterSortFlatMapTake(int _weight, int _take) {
    FruitProcessor<Fruit> fp = fruitProcessor("FM", _weight);

    Observable
        .merge( appleFruitBasketObservable(),
                orangeFruitBasketObservable(),
                grapesFruitBasketObservable()
          )

        .filter(fp.weightFilter())
        .toSortedList()
        .flatMap(list -> Observable.from(list))
        .take(_take)

        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
            fruit -> fp.onNext((Fruit) fruit),
            throwable -> fp.onError(throwable),
            () -> fp.onCompleted()
        );
}
```





```
ATS> Starting the Async Test Suite.......................

ATS> Merging Three Streams Async Test......................
ATS> Fruit Processor (Observer) Initialized with ID = ME
ATS> Merging Three Streams Async Test Scheduled............

[ME]-A1 [44gms:Rs.32] [ME]-A2 [69gms:Rs.50] [ME]-A3 [40gms:Rs.29]
[ME]-O1 [52gms:Rs.37] [ME]-O2 [47gms:Rs.34] [ME]-O3 [63gms:Rs.45]
[ME]-O4 [27gms:Rs.19] [ME]-G1 [60gms:Rs.43] [ME]-G2 [8gms:Rs.5]
[ME]-G3 [99gms:Rs.72] [ME]-G4 [14gms:Rs.10] [ME]-G5 [23gms:Rs.16]
ATS-ME> Fruit Process Task Completed - Total Time in Seconds = 6.0

ATS> Merging Three Streams & Filter Async Test......................
ATS> Filter Criteria : Weight > 51
ATS> Fruit Processor (Observer) Initialized with ID = FI
ATS> Merging Three Streams & Filter Async Test Scheduled............

[FI]-A2 [69gms:Rs.50] [FI]-O1 [52gms:Rs.37] [FI]-O3 [63gms:Rs.45]
[FI]-G1 [60gms:Rs.43] [FI]-G3 [99gms:Rs.72]
ATS-FI> Fruit Process Task Completed - Total Time in Seconds = 2.0

ATS> Merging Three Streams & Filter / Sort Price (Asc) Async Test........
ATS> Filter Criteria : Weight > 51 Take = 5
ATS> Fruit Processor (Observer) Initialized with ID = SO
ATS> Merging Three Streams & Filter / Sort Async Test Scheduled............

[SO]-O1 [52gms:Rs.37] [SO]-G1 [60gms:Rs.43] [SO]-O3 [63gms:Rs.45]
[SO]-A2 [69gms:Rs.50] [SO]-G3 [99gms:Rs.72]
ATS-SO> Fruit Process Task Completed - Total Time in Seconds = 2.0

ATS> Merging Three Streams & Filter / Sort Price (Asc) / Flat Map Async Test.
ATS> Filter Criteria : Weight > 51 Take = 5
ATS> Fruit Processor (Observer) Initialized with ID = FM
ATS> Merging Three Streams & Filter / Sort / Flat Map Async Test Scheduled...

[FM]-O1 [52gms:Rs.37] [FM]-G1 [60gms:Rs.43] [FM]-O3 [63gms:Rs.45]
[FM]-A2 [69gms:Rs.50] [FM]-G3 [99gms:Rs.72]
ATS-FM> Fruit Process Task Completed - Total Time in Seconds = 2.0

ATS> Async Test Suite Complete .......................
```

08 July 2017

The Problem? How Rx Helps!

Scenario – Three Different Data streams

1. Movie database, Movie Reviews, Movie Trivia
2. Based on the Recommendation Engine combine all the three above and send the final result to end-user / customer.

The Problem / Rule

1. No Synchronous Calls allowed

What I need

1. Movie Suggestion based on recommendation Engine
2. Filter the suggested movies based on the User filter (Movie Ratings)
3. Once the movies are identified, combine all the other related info and Zip (Movies, Reviews & Trivia) it into a Single Entity (Movie Titles) and send the collection back to the caller.

RxJava Operator : Filter / Sort / FlatMap

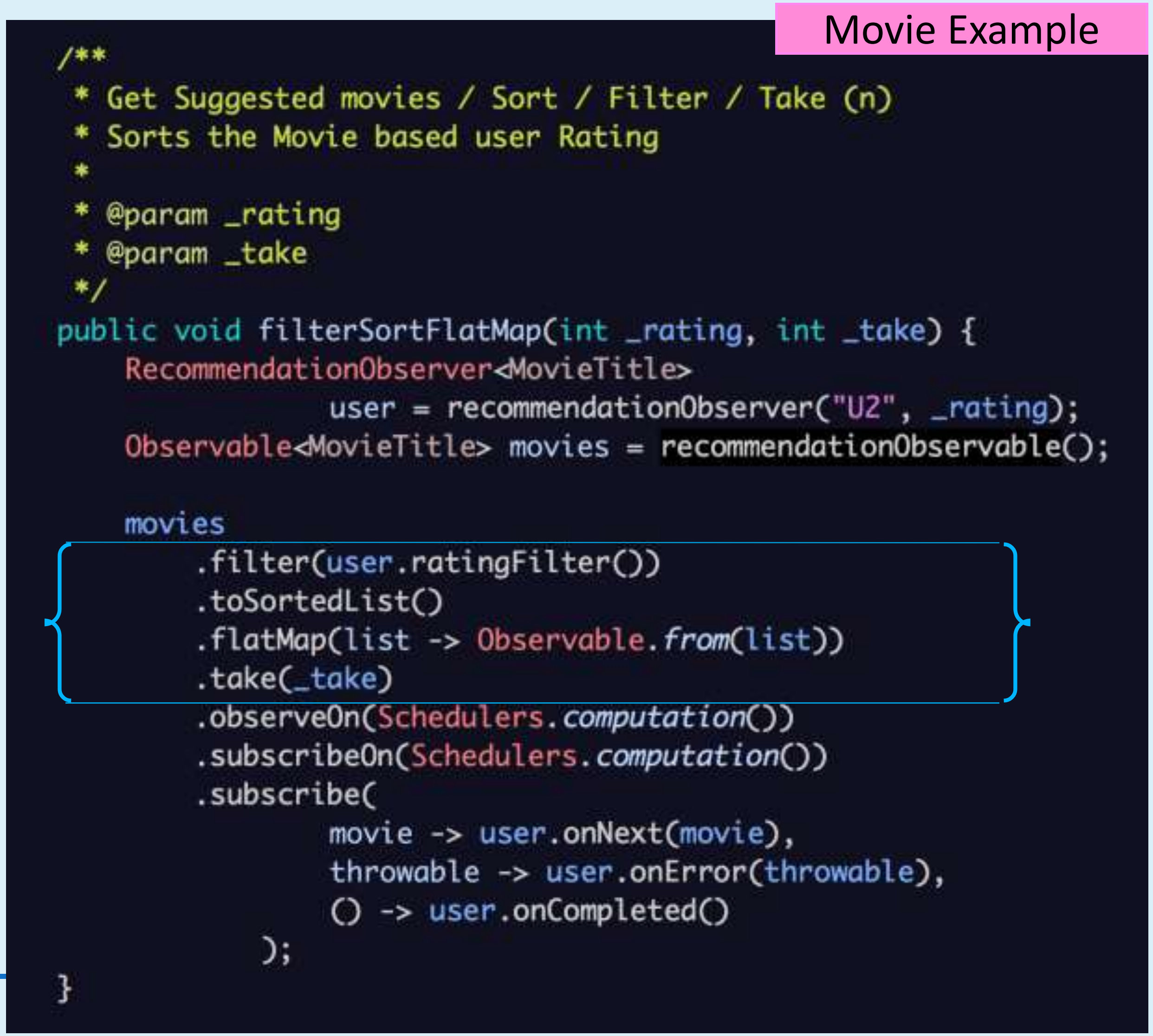
Reactive Extensions (Rx)

Building Block 4

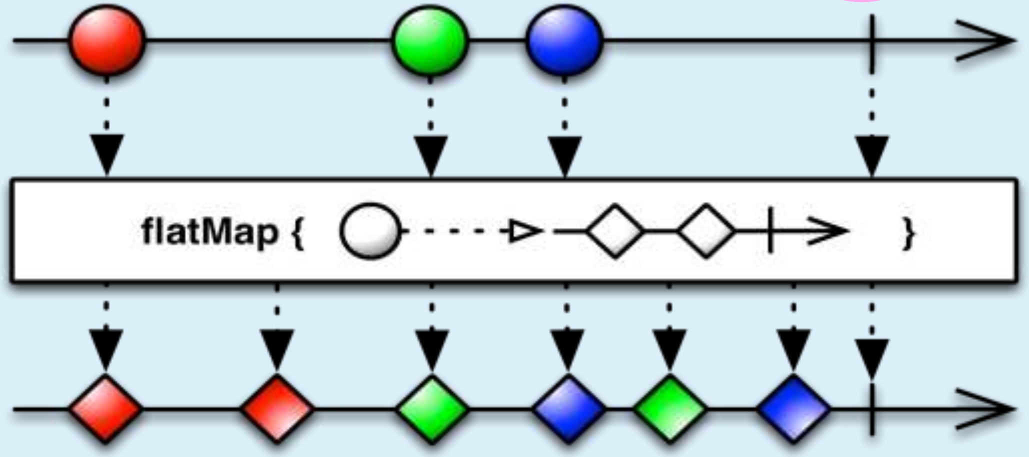
Objective:

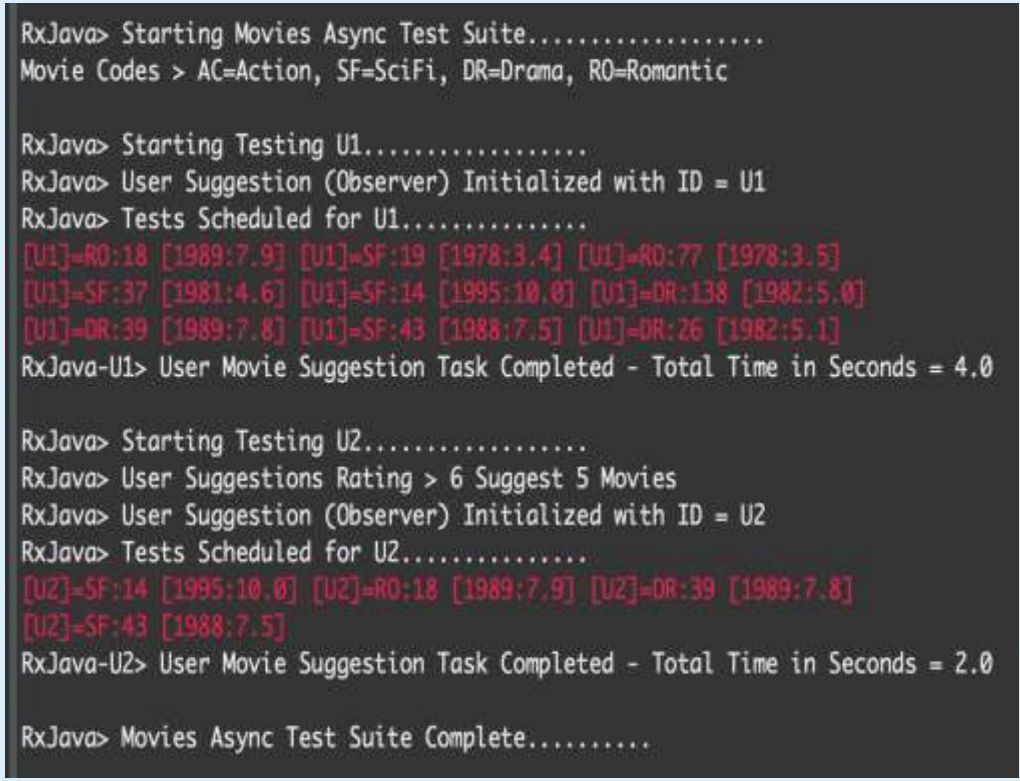
toSortedList() returns an Observable with a single List containing Movies. Using FlatMap to Transform Observable <List> to Observable <MovieTitle>

Movie Example



```
/**
 * Get Suggested movies / Sort / Filter / Take (n)
 * Sorts the Movie based user Rating
 *
 * @param _rating
 * @param _take
 */
public void filterSortFlatMap(int _rating, int _take) {
    RecommendationObserver<MovieTitle>
            user = recommendationObserver("U2", _rating);
    Observable<MovieTitle> movies = recommendationObservable();

    movies
        .filter(user.ratingFilter())
        .toSortedList()
        .flatMap(list -> Observable.from(list))
        .take(_take)
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
            movie -> user.onNext(movie),
            throwable -> user.onError(throwable),
            () -> user.onCompleted()
        );
}
```





```
RxJava> Starting Movies Async Test Suite..................
Movie Codes > AC=Action, SF=SciFi, DR=Drama, RO=Romantic

RxJava> Starting Testing U1.................
RxJava> User Suggestion (Observer) Initialized with ID = U1
RxJava> Tests Scheduled for U1..............
[U1]=RO:18 [1989:7.9] [U1]=SF:19 [1978:3.4] [U1]=RO:77 [1978:3.5]
[U1]=SF:37 [1981:4.6] [U1]=SF:14 [1995:10.0] [U1]=DR:138 [1982:5.0]
[U1]=DR:39 [1989:7.8] [U1]=SF:43 [1988:7.5] [U1]=DR:26 [1982:5.1]
RxJava-U1> User Movie Suggestion Task Completed - Total Time in Seconds = 4.0

RxJava> Starting Testing U2.................
RxJava> User Suggestions Rating > 6 Suggest 5 Movies
RxJava> User Suggestion (Observer) Initialized with ID = U2
RxJava> Tests Scheduled for U2..............
[U2]=SF:14 [1995:10.0] [U2]=RO:18 [1989:7.9] [U2]=DR:39 [1989:7.8]
[U2]=SF:43 [1988:7.5]
RxJava-U2> User Movie Suggestion Task Completed - Total Time in Seconds = 2.0

RxJava> Movies Async Test Suite Complete.........
```

08 July 2017

RxJava Operator Details

Building Block 4

Creating Observables

Operators that originate new Observables.

- Create — create an Observable from scratch by calling observer methods programmatically
- Defer — do not create the Observable until the observer subscribes, and create a fresh Observable for each observer
- Empty/Never/Throw — create Observables that have very precise and limited behavior
- From — convert some other object or data structure into an Observable
- Interval — create an Observable that emits a sequence of integers spaced by a particular time interval
- Just — convert an object or a set of objects into an Observable that emits that or those objects
- Range — create an Observable that emits a range of sequential integers
- Repeat — create an Observable that emits a particular item or sequence of items repeatedly
- Start — create an Observable that emits the return value of a function
- Timer — create an Observable that emits a single item after a given delay

Reactive Extensions (Rx)

Source: http://reactivex.io/

RxJava Operator Details

Building Block 4

Transforming Observables

Operators that transform items that are emitted by an Observable.

- **Buffer** — periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time
- **FlatMap** — transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable
- **GroupBy** — divide an Observable into a set of Observables that each emit a different group of items from the original Observable, organized by key
- **Map** — transform the items emitted by an Observable by applying a function to each item
- **Scan** — apply a function to each item emitted by an Observable, sequentially, and emit each successive value
- **Window** — periodically subdivide items from an Observable into Observable windows and emit these windows rather than emitting the items one at a time

Reactive Extensions (Rx)

Source: http://reactivex.io/

RxJava Operator Details

Building Block 4

Filtering Observables

Operators that selectively emit items from a source Observable.

- **Debounce** — only emit an item from an Observable if a particular timespan has passed without it emitting another item
- **Distinct** — suppress duplicate items emitted by an Observable
- **ElementAt** — emit only item *n* emitted by an Observable
- **Filter** — emit only those items from an Observable that pass a predicate test
- **First** — emit only the first item, or the first item that meets a condition, from an Observable
- **IgnoreElements** — do not emit any items from an Observable but mirror its termination notification
- **Last** — emit only the last item emitted by an Observable
- **Sample** — emit the most recent item emitted by an Observable within periodic time intervals
- **Skip** — suppress the first *n* items emitted by an Observable
- **SkipLast** — suppress the last *n* items emitted by an Observable
- **Take** — emit only the first *n* items emitted by an Observable
- **TakeLast** — emit only the last *n* items emitted by an Observable

Source: http://reactivex.io/

RxJava Operator Details

Building Block 4

Combining Observables

Operators that work with multiple source Observables to create a single Observable

- **And/Then/When** — combine sets of items emitted by two or more Observables by means of Pattern and Plan intermediaries
- **CombineLatest** — when an item is emitted by either of two Observables, combine the latest item emitted by each Observable via a specified function and emit items based on the results of this function
- **Join** — combine items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable
- **Merge** — combine multiple Observables into one by merging their emissions
- **StartWith** — emit a specified sequence of items before beginning to emit the items from the source Observable
- **Switch** — convert an Observable that emits Observables into a single Observable that emits the items emitted by the most-recently-emitted of those Observables
- **Zip** — combine the emissions of multiple Observables together via a specified function and emit single items for each combination based on the results of this function

Reactive Extensions (Rx)

Source: http://reactivex.io/

RxJava Operator Details

Building Block 4

Observable Utility Operators

A toolbox of useful Operators for working with Observables

- **Delay** — shift the emissions from an Observable forward in time by a particular amount
- **Do** — register an action to take upon a variety of Observable lifecycle events
- **Materialize/Dematerialize** — represent both the items emitted and the notifications sent as emitted items, or reverse this process
- **ObserveOn** — specify the scheduler on which an observer will observe this Observable
- **Serialize** — force an Observable to make serialized calls and to be well-behaved
- **Subscribe** — operate upon the emissions and notifications from an Observable
- **SubscribeOn** — specify the scheduler an Observable should use when it is subscribed to
- **TimeInterval** — convert an Observable that emits items into one that emits indications of the amount of time elapsed between those emissions
- **Timeout** — mirror the source Observable, but issue an error notification if a particular period of time elapses without any emitted items
- **Timestamp** — attach a timestamp to each item emitted by an Observable
- **Using** — create a disposable resource that has the same lifespan as the Observable

Source: http://reactivex.io/

RxJava Operator Details

Building Block 4

Conditional and Boolean Operators

Operators that evaluate one or more Observables or items emitted by Observables

- All — determine whether all items emitted by an Observable meet some criteria
- Amb — given two or more source Observables, emit all of the items from only the first of these Observables to emit an item
- Contains — determine whether an Observable emits a particular item or not
- DefaultIfEmpty — emit items from the source Observable, or a default item if the source Observable emits nothing
- SequenceEqual — determine whether two Observables emit the same sequence of items
- SkipUntil — discard items emitted by an Observable until a second Observable emits an item
- SkipWhile — discard items emitted by an Observable until a specified condition becomes false
- TakeUntil — discard items emitted by an Observable after a second Observable emits an item or terminates
- TakeWhile — discard items emitted by an Observable after a specified condition becomes false

Source: http://reactivex.io/

RxJava Operator Details

Building Block 4

Mathematical and Aggregate Operators

Operators that operate on the entire sequence of items emitted by an Observable

- **Average** — calculates the average of numbers emitted by an Observable and emits this average
- **Concat** — emit the emissions from two or more Observables without interleaving them
- **Count** — count the number of items emitted by the source Observable and emit only this value
- **Max** — determine, and emit, the maximum-valued item emitted by an Observable
- **Min** — determine, and emit, the minimum-valued item emitted by an Observable
- **Reduce** — apply a function to each item emitted by an Observable, sequentially, and emit the final value
- **Sum** — calculate the sum of numbers emitted by an Observable and emit this sum

Backpressure Operators

- **backpressure operators** — strategies for coping with Observables that produce items more rapidly than their observers consume them

Reactive Extensions (Rx)

Source: http://reactivex.io/

RxJava Operator Details

Building Block 4

Connectable Observable Operators

Specialty Observables that have more precisely-controlled subscription dynamics

- Connect — instruct a connectable Observable to begin emitting items to its subscribers
- Publish — convert an ordinary Observable into a connectable Observable
- RefCount — make a Connectable Observable behave like an ordinary Observable
- Replay — ensure that all observers see the same sequence of emitted items, even if they subscribe after the Observable has begun emitting items

Operators to Convert Observables

- To — convert an Observable into another object or data structure

Error Handling Operators

Operators that help to recover from error notifications from an Observable

- Catch — recover from an onError notification by continuing the sequence without error
- Retry — if a source Observable sends an onError notification, re-subscribe to it in the hopes that it will complete without error

Reactive Extensions (Rx)

Source: http://reactivex.io/

EVENT SOURCING & CQRS

Event Sourcing & CQRS

Command and Query Responsibility Segregation

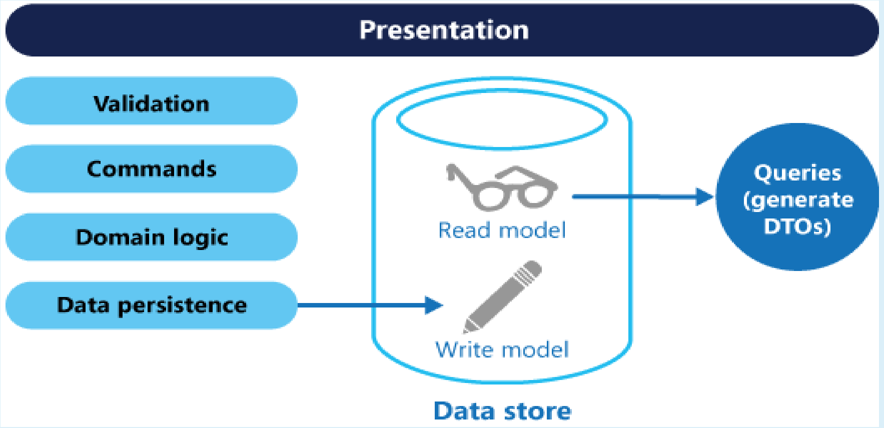
Axon
Framework
For Java

- In traditional data management systems, both commands (updates to the data) and queries (requests for data) are executed against the same set of entities in a single data repository.

- CQRS is a pattern that segregates the operations that read data (Queries) from the operations that update data (Commands) by using separate interfaces.

- CQRS should only be used on specific portions of a system in Bounded Context (in DDD).

- CQRS should be used along with Event Sourcing.



Presentation

Validation

Commands

Domain logic

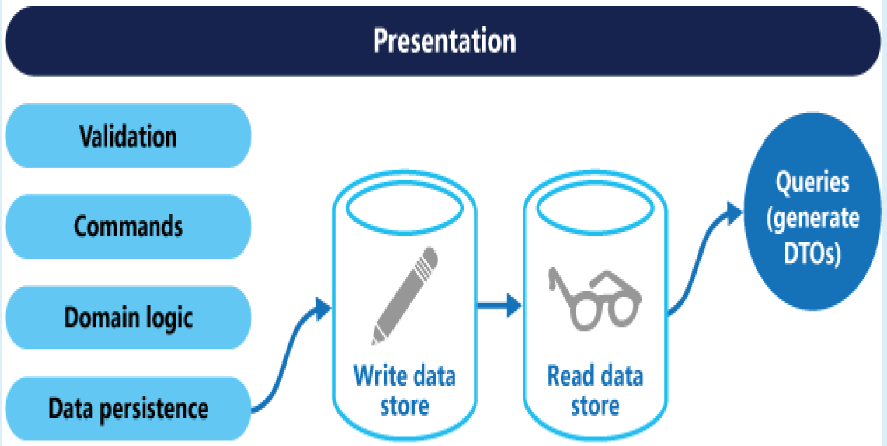
Data persistence

Read model

Write model

Data store

Queries (generate DTOs)



Presentation

Validation

Commands

Domain logic

Data persistence

Write data store

Read data store

Queries (generate DTOs)

CQS :
Bertrand Meyer

Greg
Young

Java Axon Framework Resource : http://www.axonframework.org

MSDN – Microsoft https://msdn.microsoft.com/en-us/library/dn568103.aspx | Martin Fowler : CQRS – http://martinfowler.com/bliki/CQRS.html

08 July 2017

Event Sourcing and CQRS Design Example

Domain

The example focus on a concept of a Café which tracks the visit of an individual or group to the café. When people arrive at the café and take a table, a tab is opened. They may then order drinks and food. Drinks are served immediately by the table staff, however food must be cooked by a chef. Once the chef prepared the food it can then be served.

Events

- TabOpened
- DrinksOrdered
- FoodOrdered
- DrinksCancelled
- FoodCancelled
- DrinksServed
- FoodPrepared
- FoodServed
- TabClosed

An Event Stream which is an **immutable** collection of events up until a specific version of an **aggregate**.

The purpose of the version is to implement optimistic locking:

Commands

- OpenTab
- PlaceOrder
- AmendOrder
- MarkDrinksServed
- MarkFoodPrepared
- MarkFoodServed
- CloseTab

Commands are things that indicate **requests** to our domain. While an event states that something certainly happened, a command may be **accepted** or **rejected**.

An accepted command leads to zero or more events being emitted to incorporate new facts into the system. A rejected command leads to some kind of exception.

Aggregates

- A Single Object, which doesn't reference any others.
- An isolated Graph of objects, with One object designated as the Root of the Aggregate.

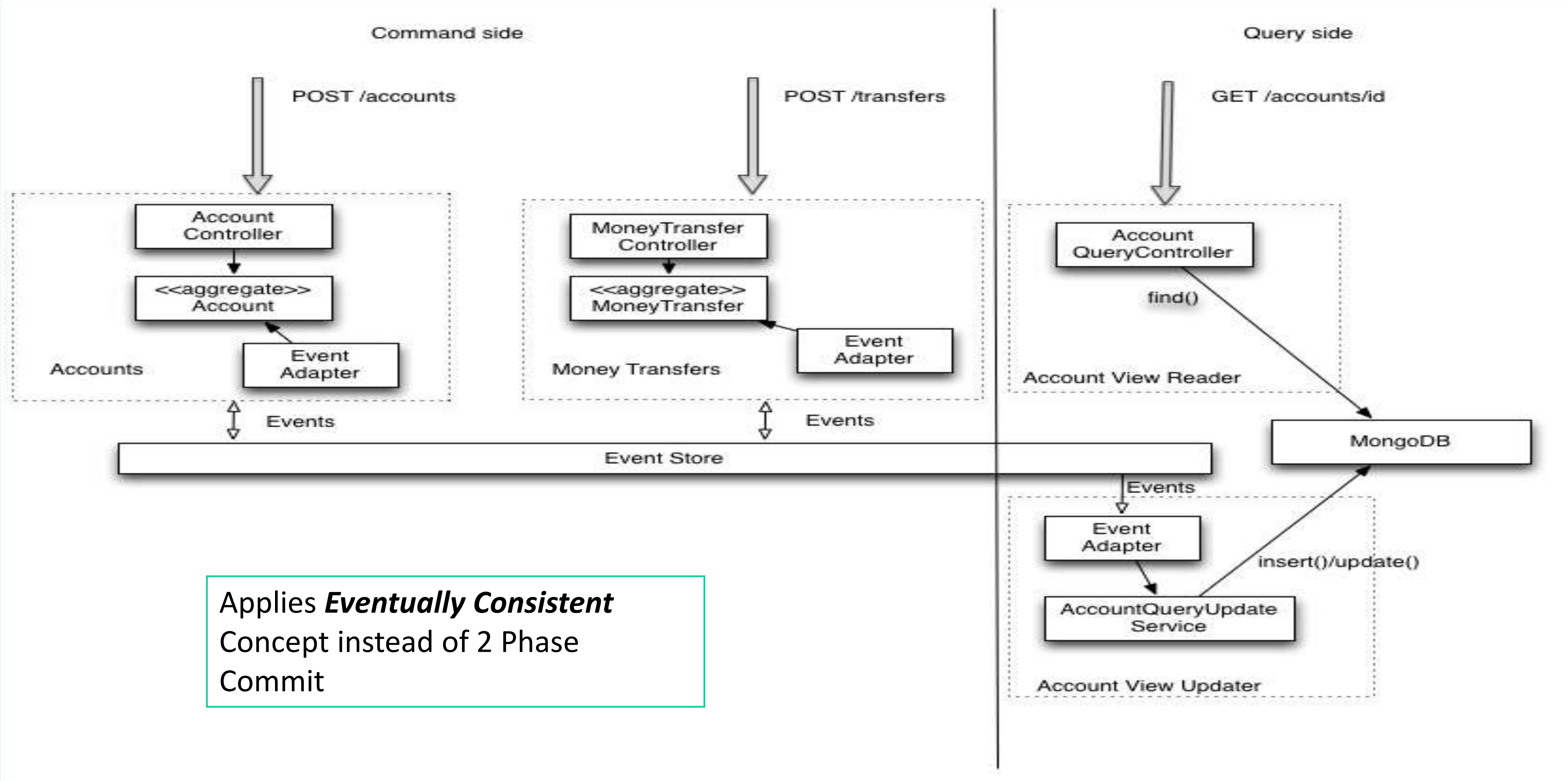
Exception

- CannotCancelServedItem
- TabHasUnservedItem
- MustPayEnough

An important part of the modeling process is thinking about the things that can cause a command to be refused.

Source: http://cqrs.nu/tutorial/cs/01-design

Event Sourcing & CQRS : Banking App Example

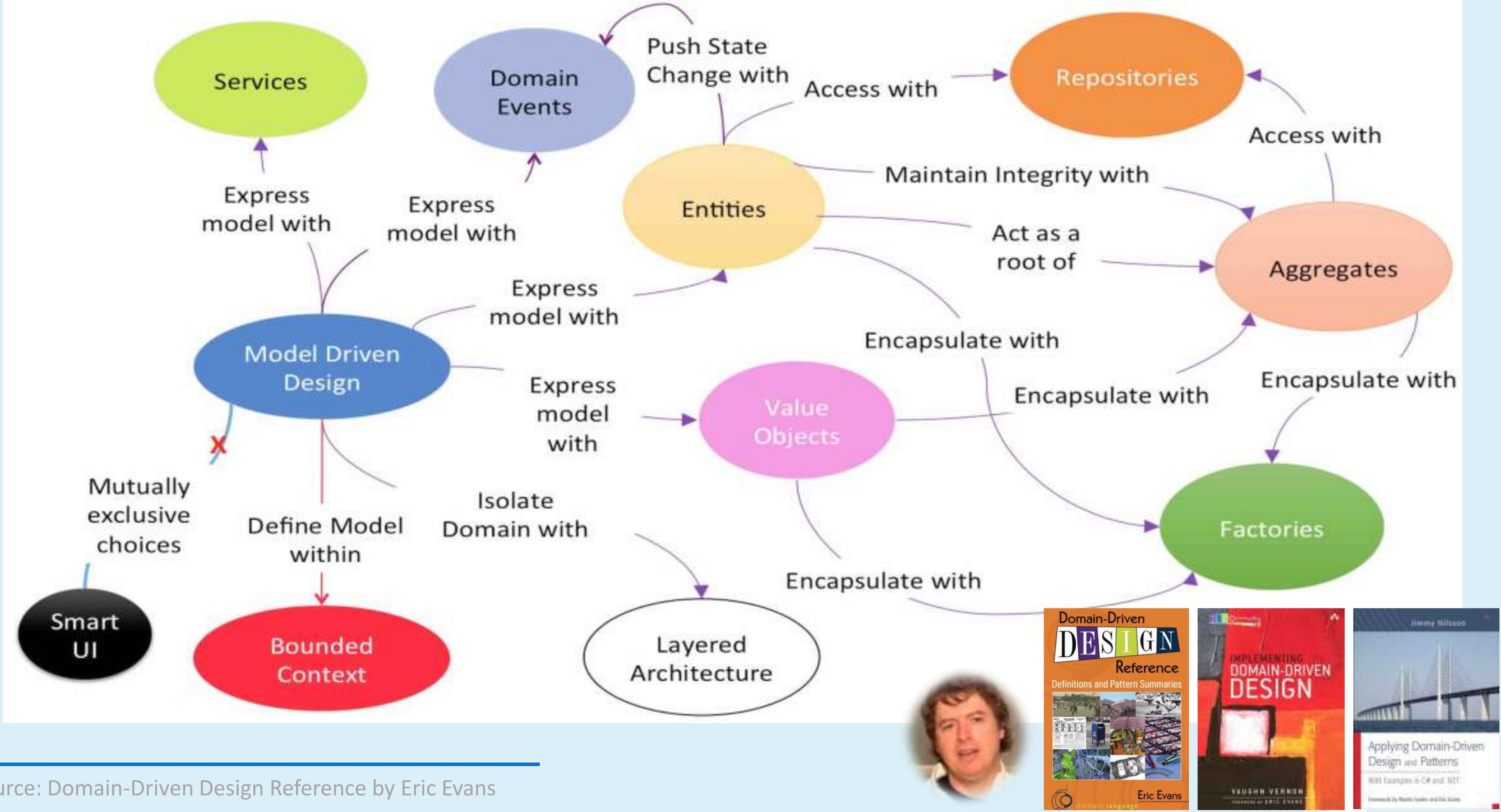


Applies ***Eventually Consistent*** Concept instead of 2 Phase Commit

Source: http://plainoldobjects.com

DOMAIN DRIVEN DESIGN

Domain Driven Design



Source: Domain-Driven Design Reference by Eric Evans

What is Domain-Driven Design?

Domain-Driven Design is an approach to the development of complex software in which we:

- Focus on the Core Domain
- Explore models in a creative collaboration of domain specialists and software practitioners
- Speak a ubiquitous language within an explicitly bounded context

Source: Domain-Driven Design Reference by Eric Evans

What is Domain-Driven Design?

Domain

A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software. Domain encapsulates a Domain Model.

Domain Model

A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.

Context

The setting in which a word or statement appears that determines its meaning. *Statements about a model can only be understood in a context.*

Bounded Context

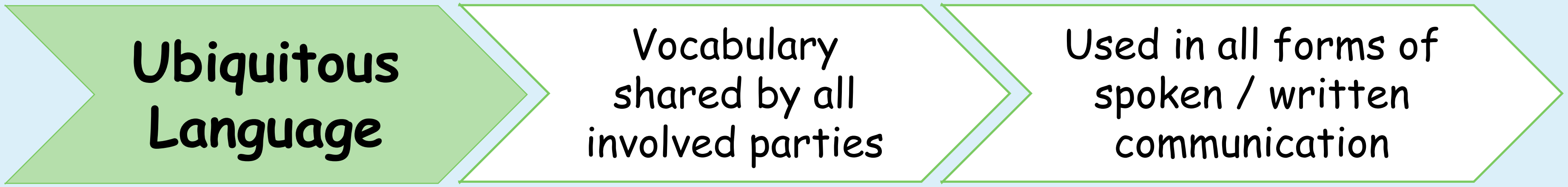
A description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable.

Source: Domain-Driven Design Reference by Eric Evans

Requirements

1. Requirement Analysis using DDD
2. Understanding Bounded Context
3. Context Map
4. Understanding Context Map
5. Context Map Terms

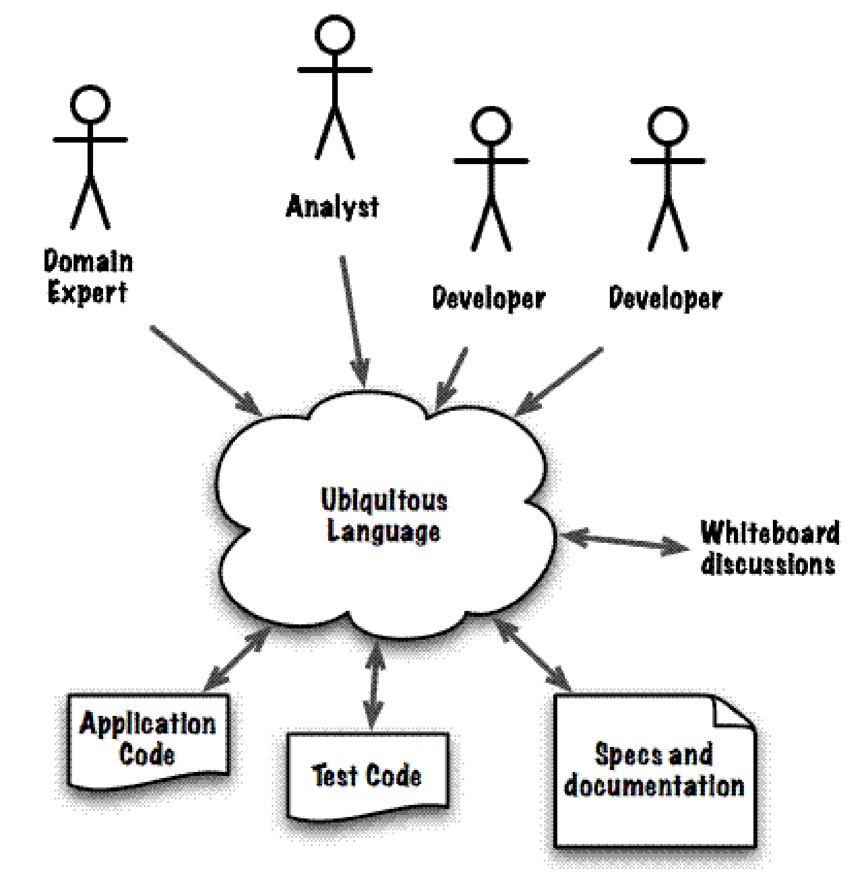
Understanding Requirement Analysis using DDD



Ubiquitous Language

Vocabulary shared by all involved parties

Used in all forms of spoken / written communication



Ubiquitous Language with BDD

As an insurance Broker
I want to know who my Gold Customers are
So that I sell more

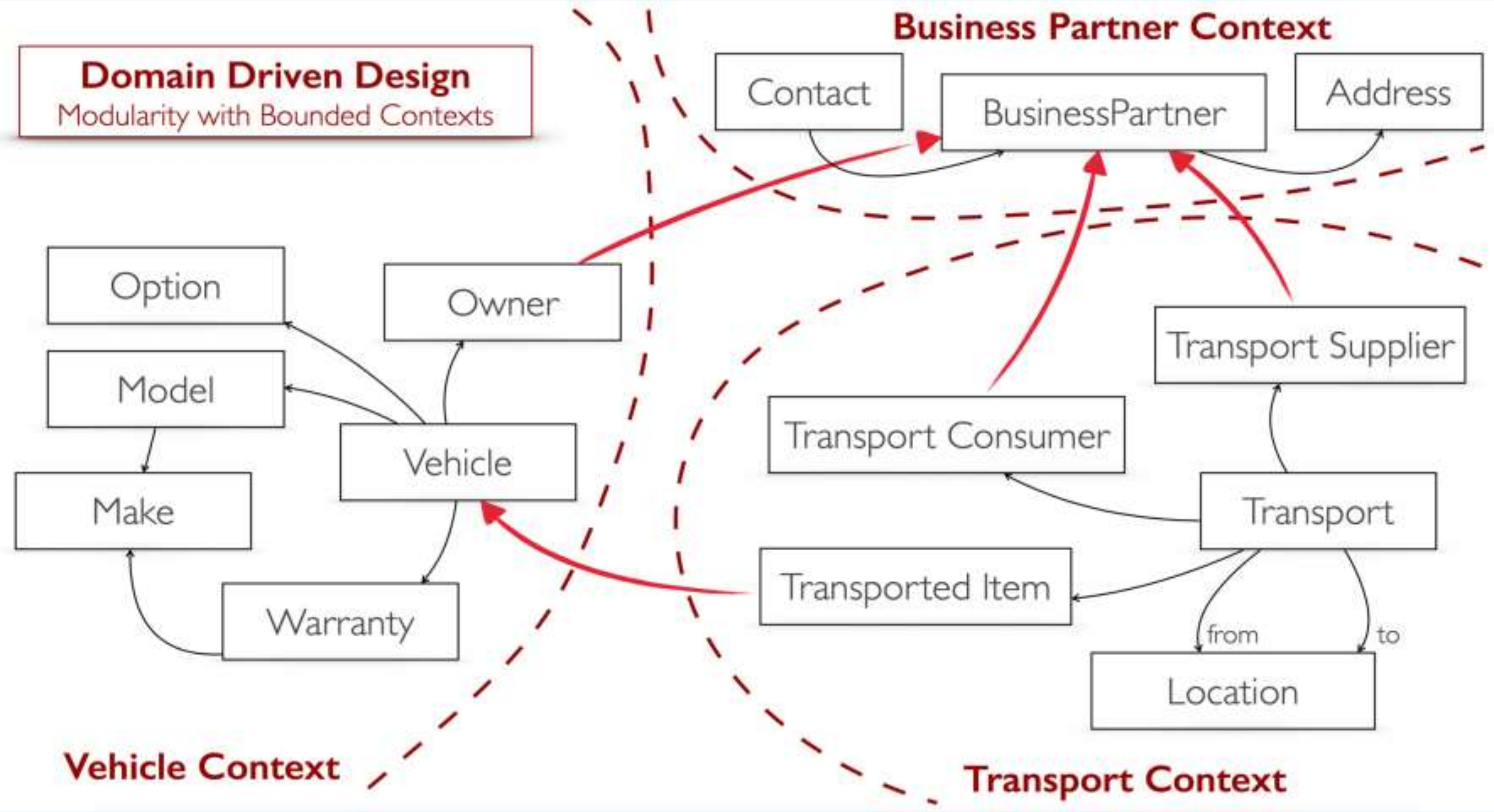
Given	Customer John Doe exists
When	he buys insurance ABC for $1000 USD
Then	He becomes a Gold Customer

Understanding Requirement Analysis using DDD

Bounded Context

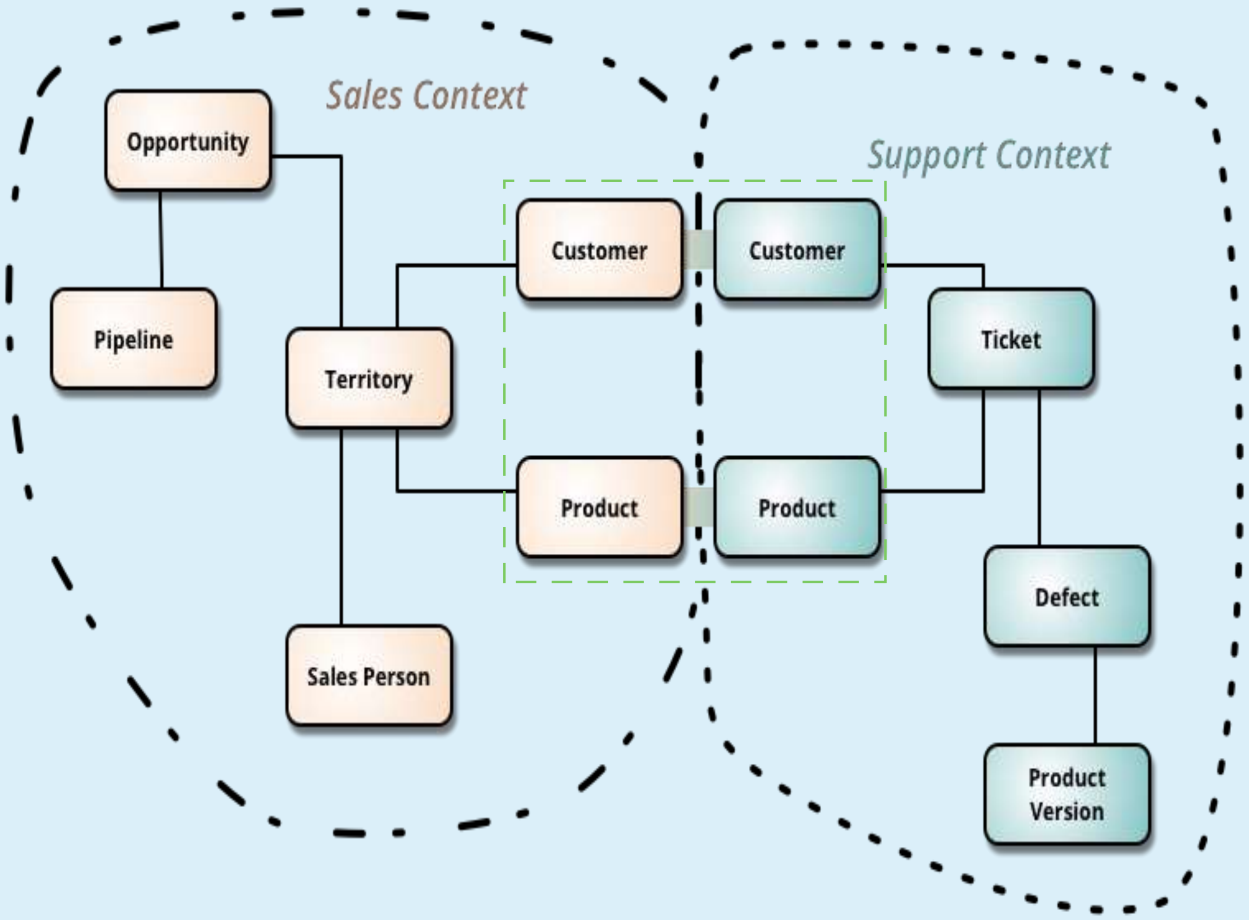
Areas of the domain treated independently

Discovered as you assess requirements and build language



DDD : Understanding Bounded Context

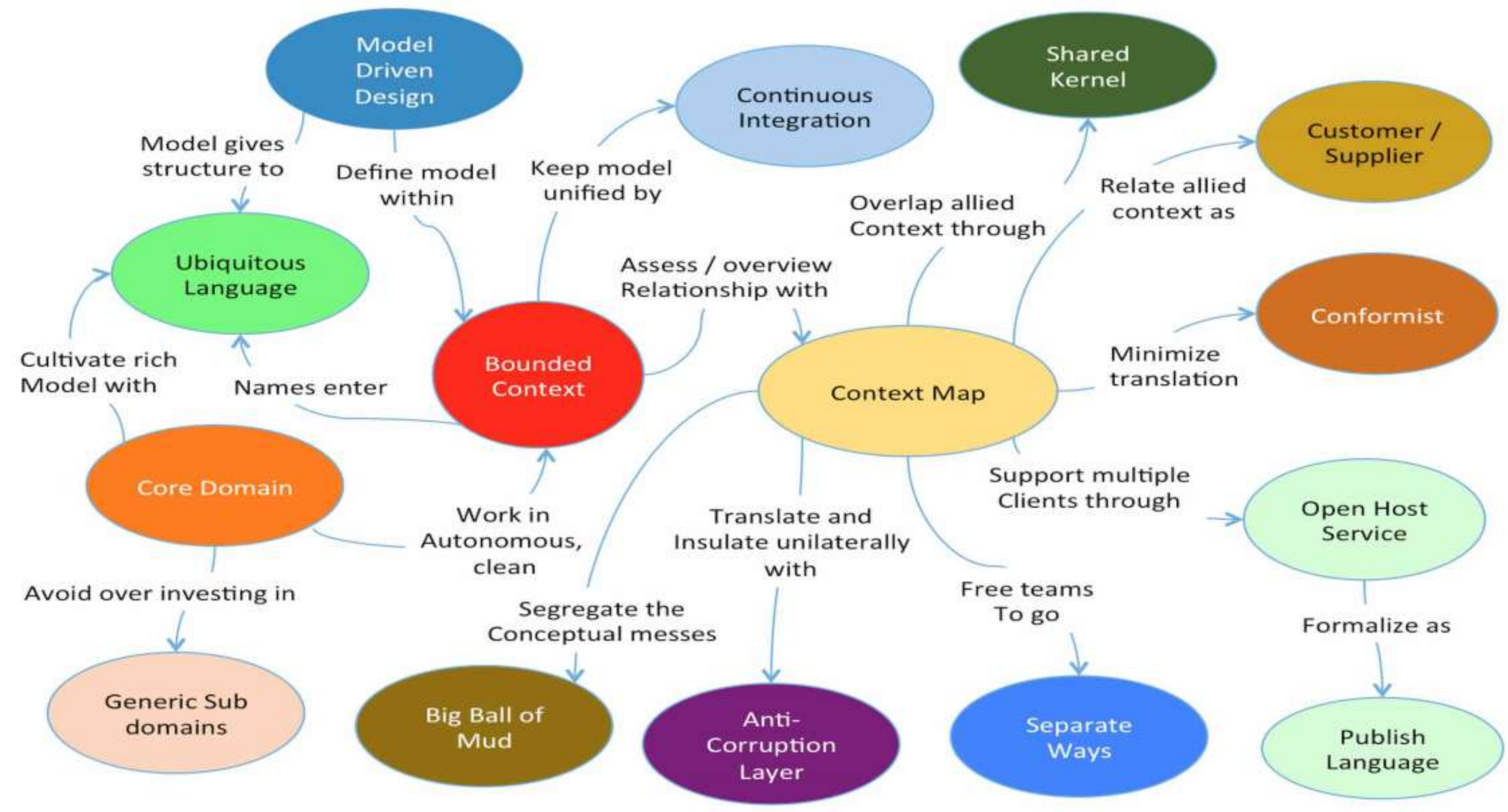
- DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.
- Bounded Contexts have both unrelated concepts
  - Such as a support ticket only existing in a customer support context
  - But also share concepts such as products and customers.
- Different contexts may have completely different models of common concepts with mechanisms to map between these polysemic concepts for integration.



Source: BoundedContext By Martin Fowler : http://martinfowler.com/bliki/BoundedContext.html

08 July 2017

DDD : Context Map



Source: Domain-Driven Design Reference by Eric Evans

08 July 2017

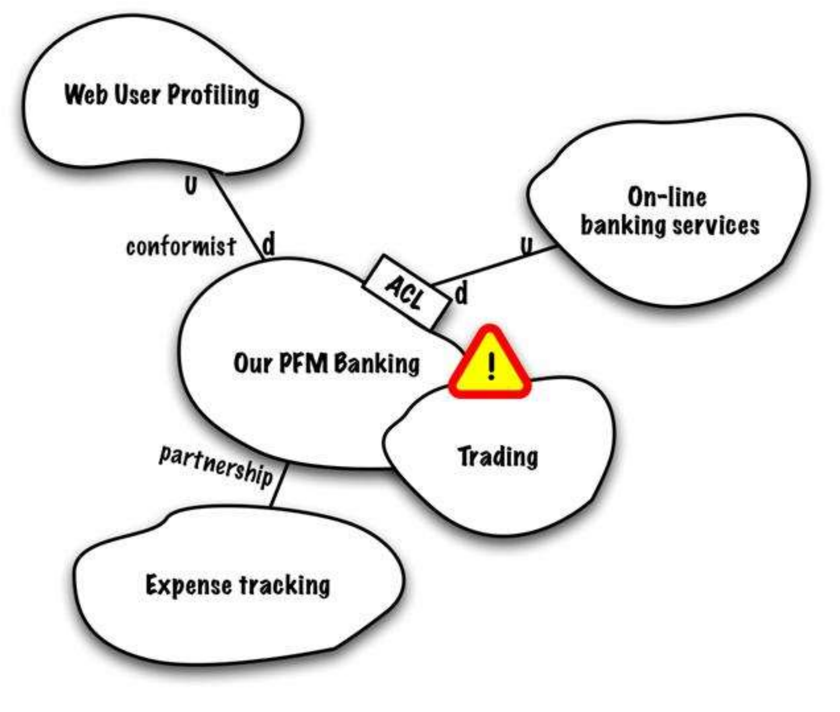
DDD : Understanding Context Map

1. A context map provides Global View of the system that UML or architecture diagrams completely miss, helping us to focus on choices that are really viable in your scenario without wasting money.
2. Each Bounded Context fits within the Context Map to show how they should communicate amongst each other and how data should be shared.

Up Stream (u) – Down Stream (d)
The upstream team may succeed independently of the fate of the downstream team.

Mutually Dependent
Success depends on the success of both the teams.

Free
In which success or failure of the development work in other contexts has little affect on delivery.



Context Map Terms

Term	Definition	Action
Partnership	When teams in two context will succeed or fail together, a cooperative relationship often emerges.	Forge Partnerships
Shared Kernel	Sharing a part of the Mode and associated code is very intimate interdependency, which can leverage design work or undermine it.	Keep the Kernel Small.
Customer / Supplier	When two teams are in upstream – downstream relationship, where the upstream team may succeed independently of the fate of the downstream team, the needs of the downstream come to be addressed in a variety of ways with wide range of consequences.	Downstream priorities factor into upstream planning.
Conformist	Upstream has no motivation in this partnership to keep the promise. Altruism may motivate Upstream developers to give promises they cant keep.	Share just enough info with upstream to keep their motivation.
Anti Corruption Layer	When the translation between two bounded context becomes more complex, then the translation layer takes on a more defensive tone.	(d) creates a layer in sync own model and matching (u) functionality.

DDD : Context Map Terms

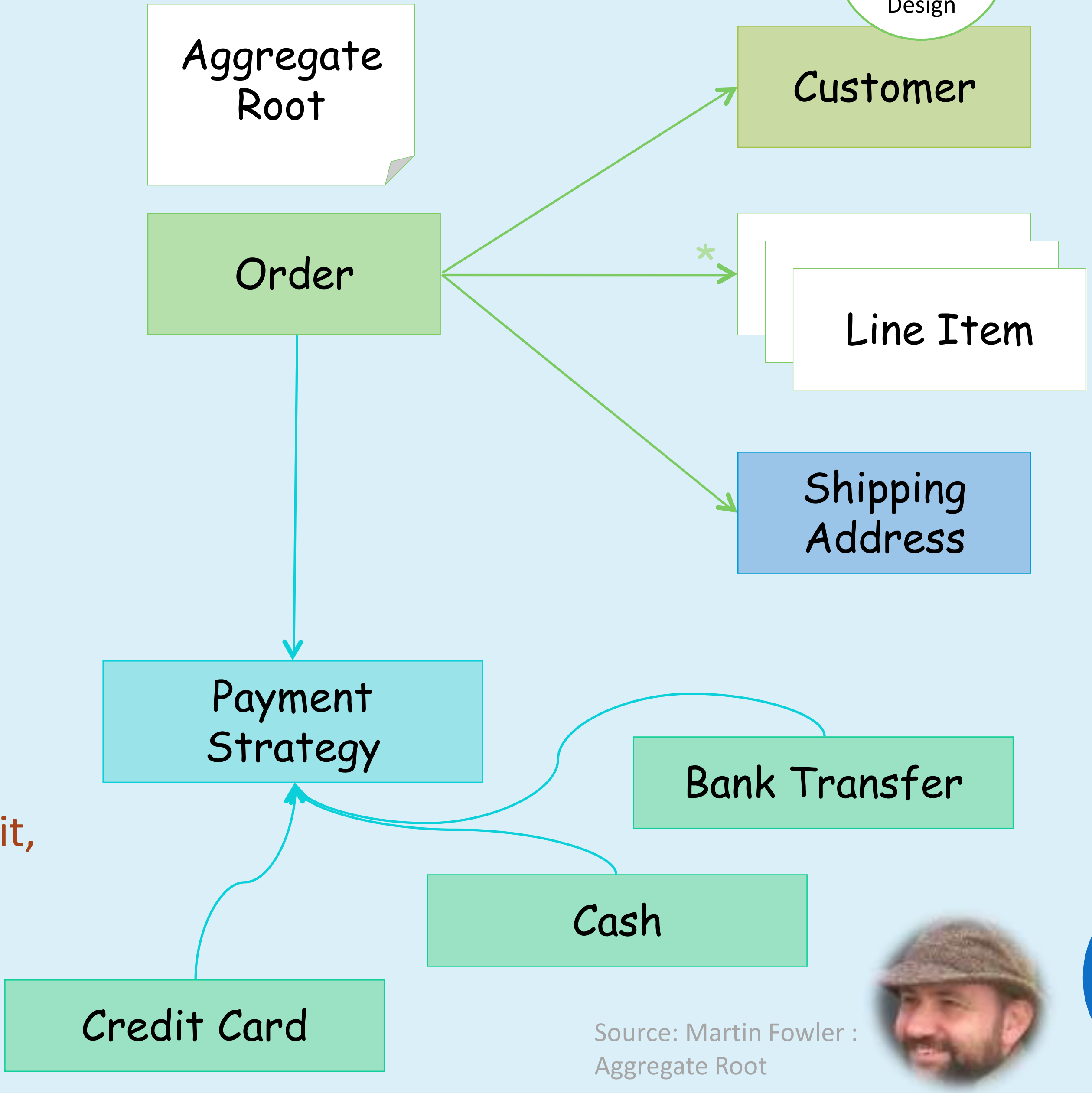
Term	Definition	Action
Open Host Service	When a subsystem has to be integrated with many others, customizing a translator for each can bog down the team. There is more and more to maintain, and more and more to worry about when changes are made.	Use a one of translator to augment the Protocol to share info (REST)
Published Language	Translation between the models of two bounded contexts requires a common language. Published Language is often combined with Open Host Service.	Use a well documented shared language (JSON)
Separate Ways	If two sets of functionality have no significant relationship, they can be completely cut loose from each other. Integration is always expensive and sometimes the benefit is small.	Bounded context with no connection to others.
Big Ball of Mud	As we survey existing systems, we find that, in fact, there are parts of systems, often large ones, where models are mixed and boundaries are inconsistent.	Designate the mess as a Big Ball of Mud.

Key Design Concepts

1. Aggregate Root
2. Value Object
3. Anti Corruption Layer
4. Repository Pattern
5. Procedural Coding Vs. Domain Driven

Understanding Aggregate Root

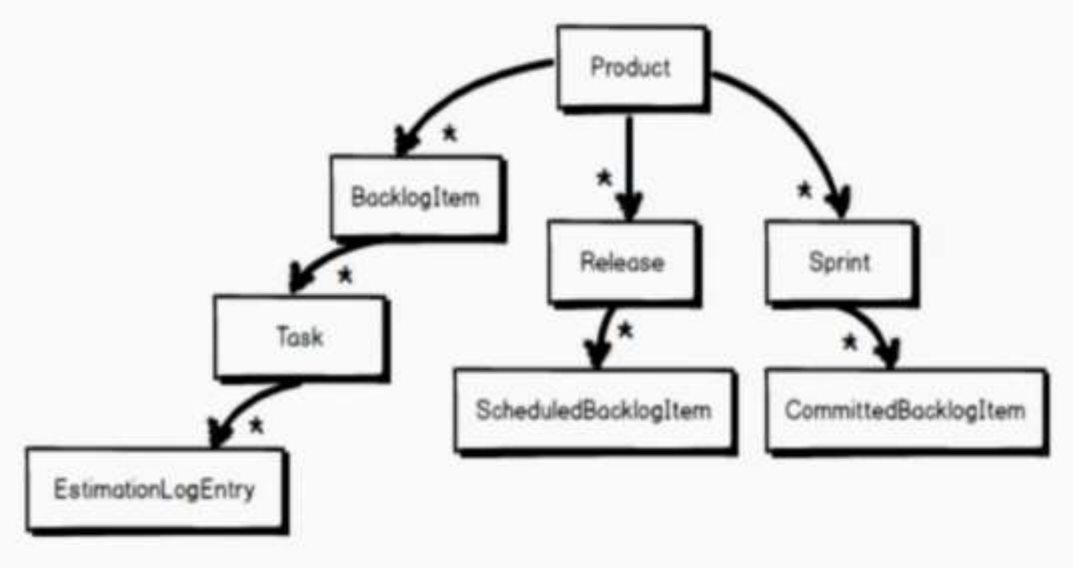
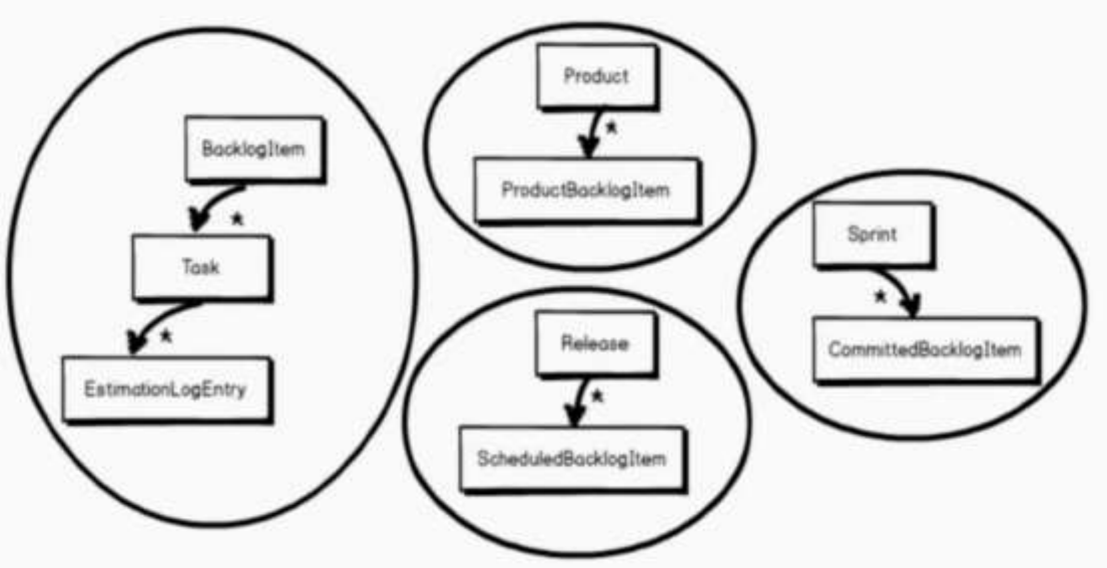
- An aggregate will have one of its component objects be the aggregate root. Any references from outside the aggregate should only go to the aggregate root. The root can thus ensure the integrity of the aggregate as a whole.
- Aggregates are the basic element of transfer of data storage - you request to load or save whole aggregates. Transactions should not cross aggregate boundaries.
- Aggregates are sometimes confused with collection classes (lists, maps, etc.).
- Aggregates are domain concepts (order, clinic visit, playlist), while collections are generic. An aggregate will often contain multiple collections, together with simple fields.



Source: Martin Fowler : Aggregate Root

08 July 2017

Designing and Fine Tuning Aggregate Root

Aggregate Root - #1	Aggregate Root - #2
	
Super Dense Single Aggregate Root Results in Transaction concurrency issues.	Super Dense Aggregate Root is split into 4 different smaller Aggregate Root in the 2nd Iteration.

Working on different design models helps the developers to come up with best possible design.

Source : Effective Aggregate Design Part 1/2/3 : Vaughn Vernon
http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf

Rules for Building Aggregate Roots

1. Protect True Invariants in Consistency Boundaries. This rule has the added implication that you should modify just one Aggregate instance in a single transaction. In other words, when you are designing an Aggregate composition, plan on that representing a transaction boundary.
2. Design Small Aggregates. The smallest Aggregate you can design is one with a single Entity, which will serve as the Aggregate Root.
3. Reference Other Aggregates Only By Identity.
4. Use Eventual Consistency Outside the Consistency Boundary. This means that ONLY ONE Aggregate instance will be required to be updated in a single transaction. All other Aggregate instances that must be updated as a result of any one Aggregate instance update can be updated within some time frame (using a Domain Event). The business should determine the allowable time delay.
5. Build Unidirectional Relationship from the Aggregate Root.

Source : Vaughn Vernon : Effective Aggregate Design : https://vaughnvernon.co/?p=838

Data Transfer Object vs. Value Object

A small simple object, like money or a date range, whose equality isn't based on identity.

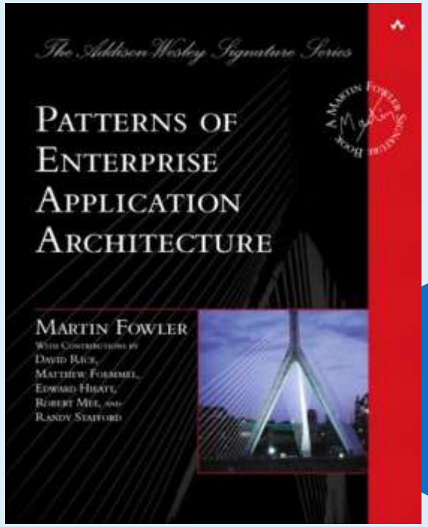
Data Transfer Object	Value Object
A DTO is just a data container which is used to transport data between layers and tiers.	A Value Object represents itself a fix set of data and is similar to a Java enum.
It mainly contains of attributes and it's a serializable object.	A Value Object doesn't have any identity, it is entirely identified by its value and is immutable.
DTOs are anemic in general and do not contain any business logic.	A real world example would be Color.RED, Color.BLUE, Currency.USD

486
P of EAA

Java EE 7 Retired the DTO

In Java EE the RS spec became the de-facto standard for remoting, so the implementation of serializable interface is no more required. To transfer data between tiers in Java EE 7 you get the following for FREE!

1. JAXB : Offer JSON / XML serialization for Free.
2. Java API for JSON Processing – Directly serialize part of the Objects into JSON


The Addison Wesley Signature Series
Patterns of Enterprise Application Architecture
Martin Fowler
With Contributions by David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford

Patterns of Enterprise Application Architecture : Martin Fowler
http://martinfowler.com/books/eaa.html

DTO – Data Transfer Object

An object that carries data between processes in order to reduce the number of method calls.

401
P of EAA

Problem: How do you preserve the simple semantics of a procedure call interface without being subject to the latency issues inherent in remote communication?

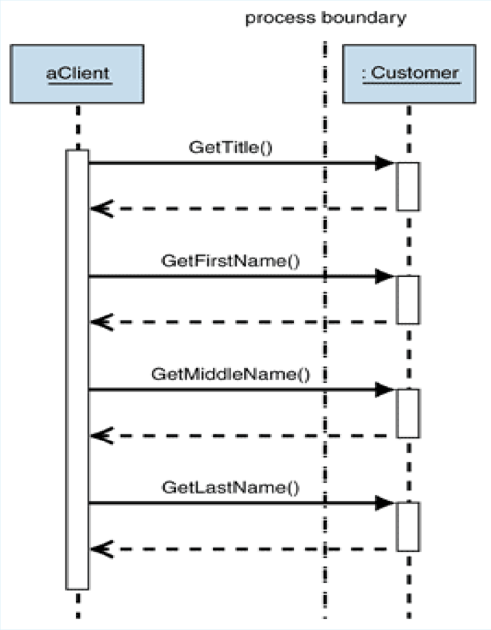
Benefits

1. Reduced Number of Calls
2. Improved Performance
3. Hidden Internals
4. Discovery of Business objects

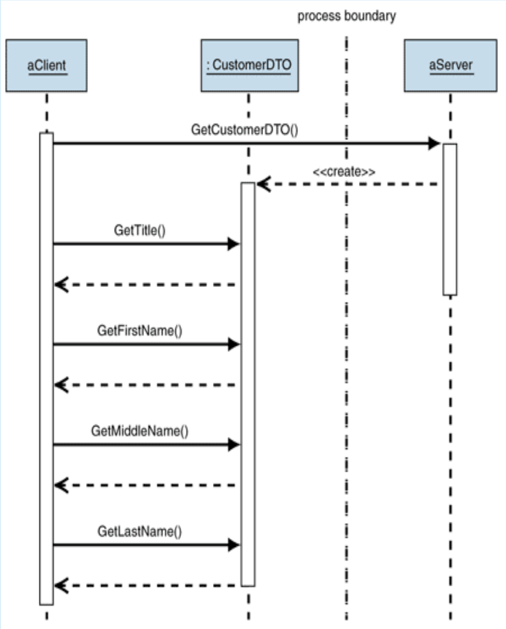
Liabilities

1. Class Explosion
2. Additional Computation
3. Additional Coding Effort

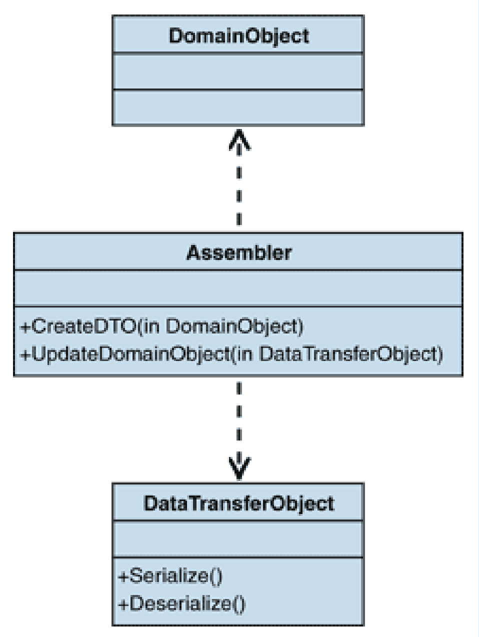
The Problem



The Solution



Assembler Pattern


DomainObject
Assembler
+CreateDTO(in DomainObject)
+UpdateDomainObject(in DataTransferObject)
DataTransferObject
+Serialize()
+Deserialize()

- Security Considerations
- Data obtained from untrusted sources, such as user input from a Web page, should be cleansed and validated before being placed into a DTO. Doing so enables you to consider the data in the DTO relatively safe, which simplifies future interactions with the DTO.

https://msdn.microsoft.com/en-us/library/ms978717.aspx

DTO – Data Transfer Object

An object that carries data between processes in order to reduce the number of method calls.

401
P of EAA

Don't underestimate the cost of [using DTOs].... It's significant, and it's painful - perhaps second only to the cost and pain of object-relational mapping.

Another argument I've heard is using them in case you want to distribute later. This kind of speculative distribution boundary is what I rail against. Adding remote boundaries adds complexity.

One case where it is useful to use something like a DTO is when you have a significant mismatch between the model in your presentation layer and the underlying domain model.

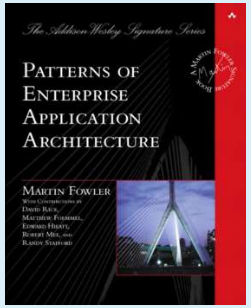
In this case it makes sense to make presentation specific facade/gateway that maps from the domain model and presents an interface that's convenient for the presentation.

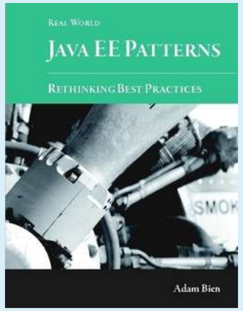
The most misused pattern in the Java Enterprise community is the DTO.

DTO was clearly defined as a solution for a distribution problem.

DTO was meant to be a coarse-grained data container which efficiently transports data between processes (tiers).

On the other hand considering a dedicated DTO layer as an investment, rarely pays off and often lead to over engineered bloated architecture.

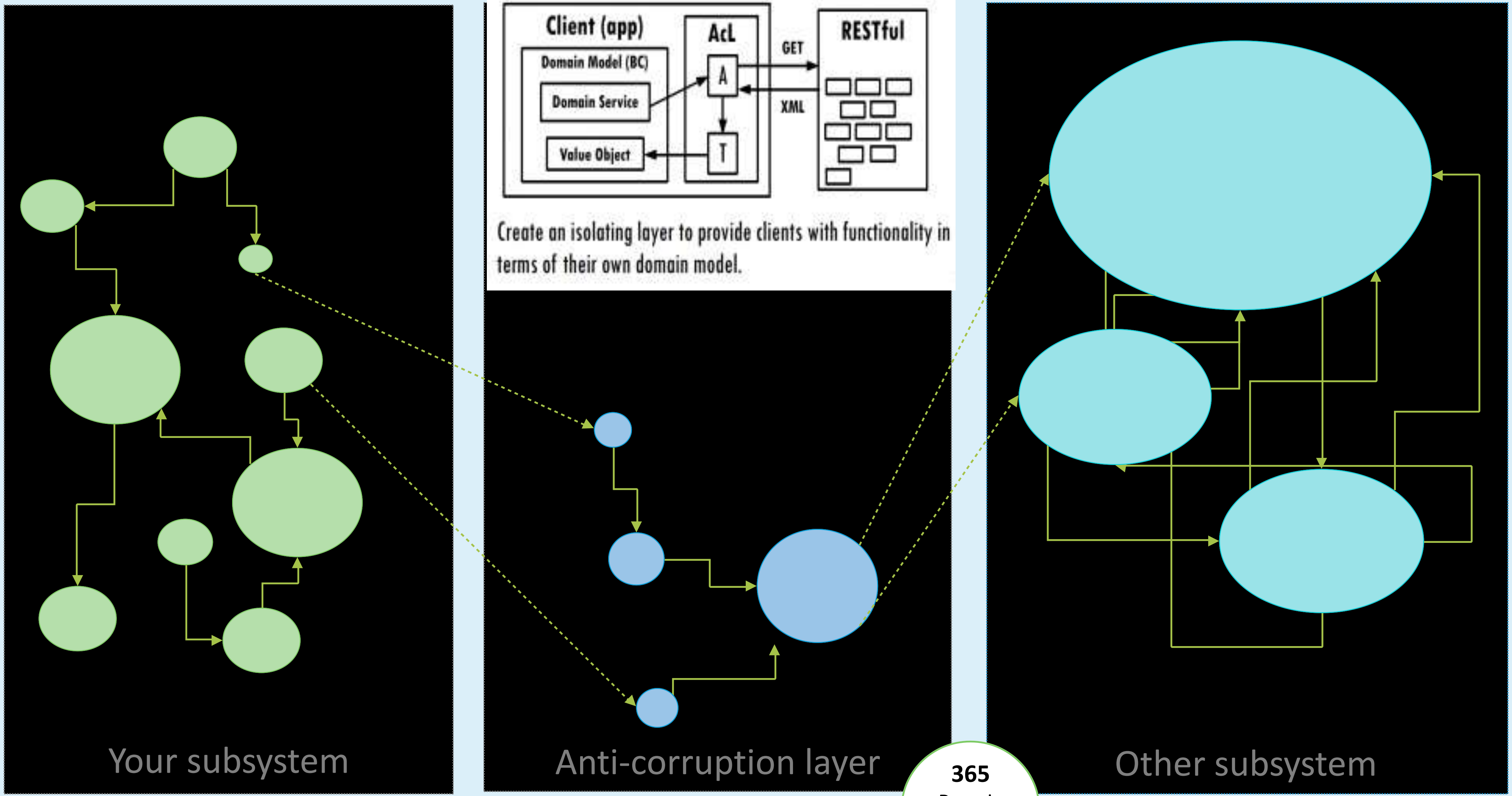

The Addison-Wesley Signature Series
Patterns of Enterprise Application Architecture
Martin Fowler



Patterns of Enterprise Application Architecture : Martin Fowler
http://martinfowler.com/books/eaa.html

Real World Java
EE Patterns
Adam Bien
http://realworldpatterns.com

Anti Corruption Layer – ACL



Create an isolating layer to provide clients with functionality in terms of their own domain model.

365
Domain
Driven
Design

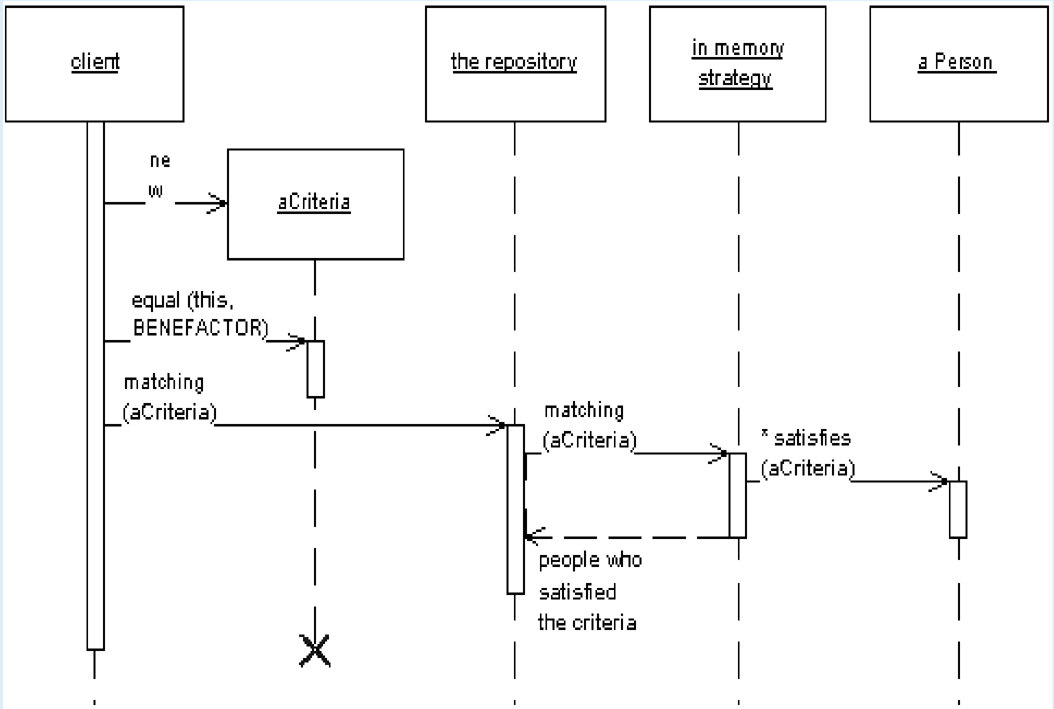
Repository Pattern

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.

322
P of EAA

08 July 2017

- Objectives
- Use the Repository pattern to achieve one or more of the following objectives:
- You want to maximize the amount of code that can be tested with automation and to isolate the data layer to support unit testing.
- You access the data source from many locations and want to apply centrally managed, consistent access rules and logic.
- You want to implement and centralize a caching strategy for the data source.
- You want to improve the code's maintainability and readability by separating business logic from data or service access logic.
- You want to use business entities that are strongly typed so that you can identify problems at compile time instead of at run time.
- You want to associate a behavior with the related data. For example, you want to calculate fields or enforce complex relationships or business rules between the data elements within an entity.
- You want to apply a domain model to simplify complex business logic.

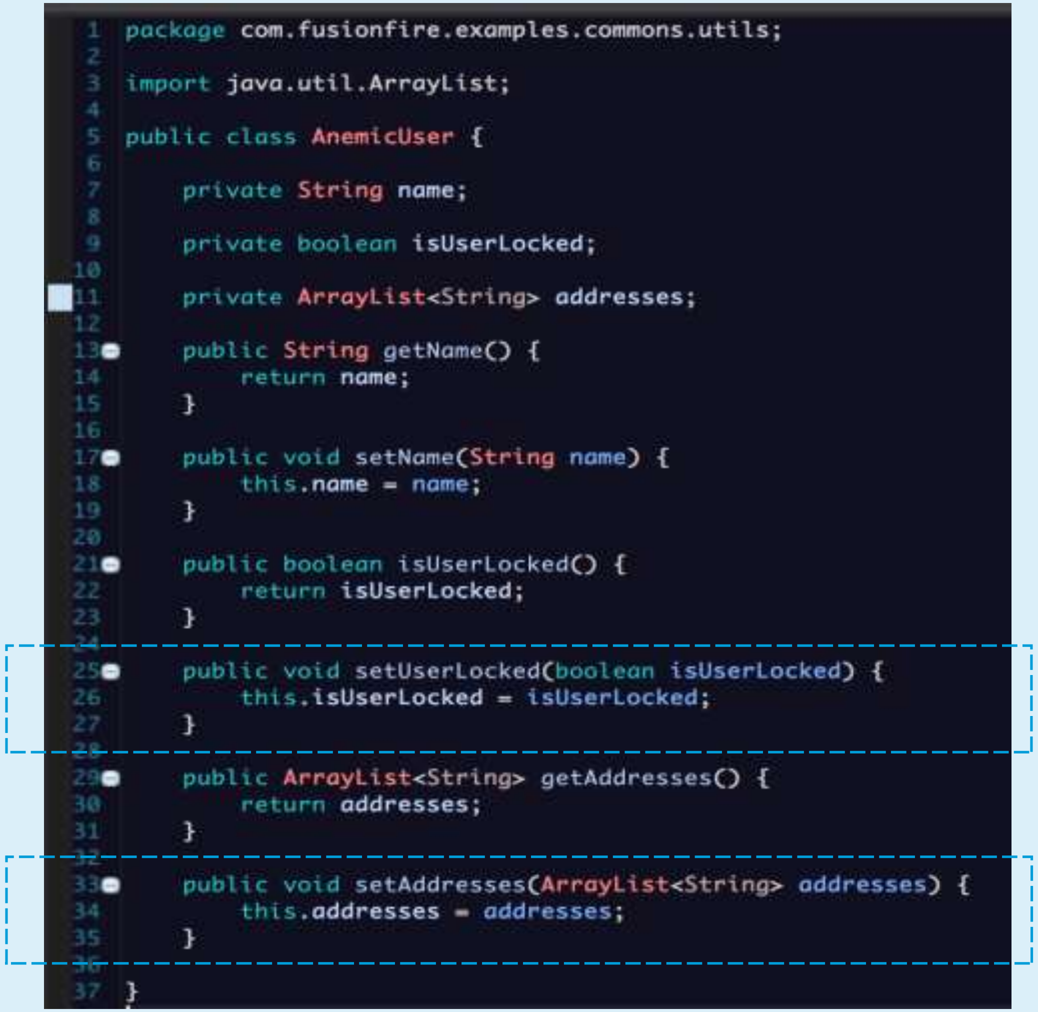


Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

Repository Pattern Source:
Martin Fowler : http://martinfowler.com/eaaCatalog/repository.html | Microsoft : https://msdn.microsoft.com/en-us/library/ff649690.aspx

Anemic Domain Model : Anti Pattern

- There are objects, many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have.
- The catch comes when you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters.
- The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design; which is to combine data and process together.
- The anemic domain model is really just a procedural style design, exactly the kind of thing that object bigots like me (and Eric) have been fighting since our early days in Smalltalk.

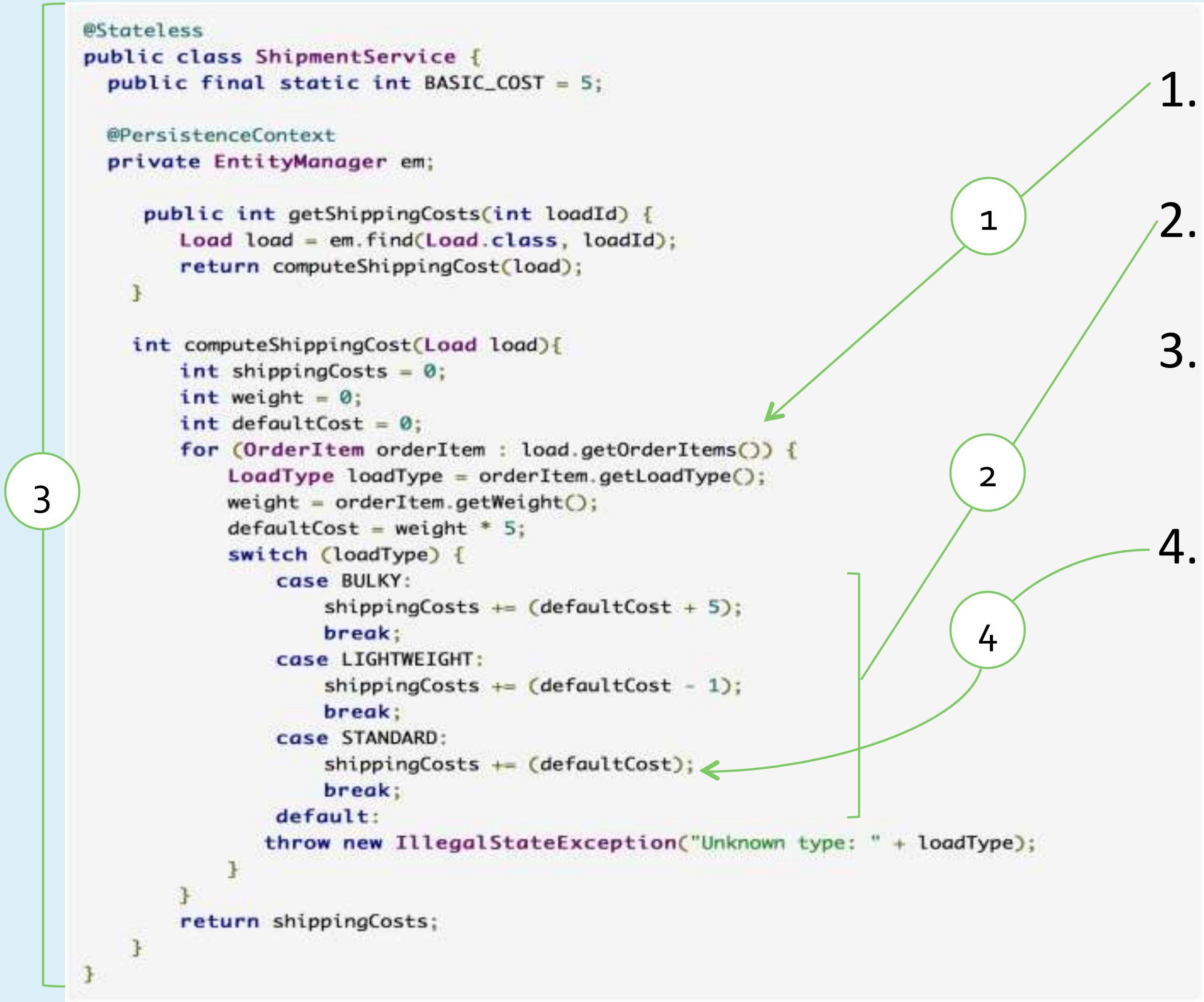


```java
package com.fusionfire.examples.commons.utils;

import java.util.ArrayList;

public class AnemicUser {

    private String name;

    private boolean isUserLocked;

    private ArrayList<String> addresses;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isUserLocked() {
        return isUserLocked;
    }

    public void setUserLocked(boolean isUserLocked) {
        this.isUserLocked = isUserLocked;
    }

    public ArrayList<String> getAddresses() {
        return addresses;
    }

    public void setAddresses(ArrayList<String> addresses) {
        this.addresses = addresses;
    }

}
```

- lockUser()
- unlockUser()
- addAddress(String address)
- removeAddress(String address)

Source: Anemic Domain Model By Martin Fowler :
http://martinfowler.com/bliki/AnemicDomainModel.html

Procedural Design Vs. Domain Driven Design



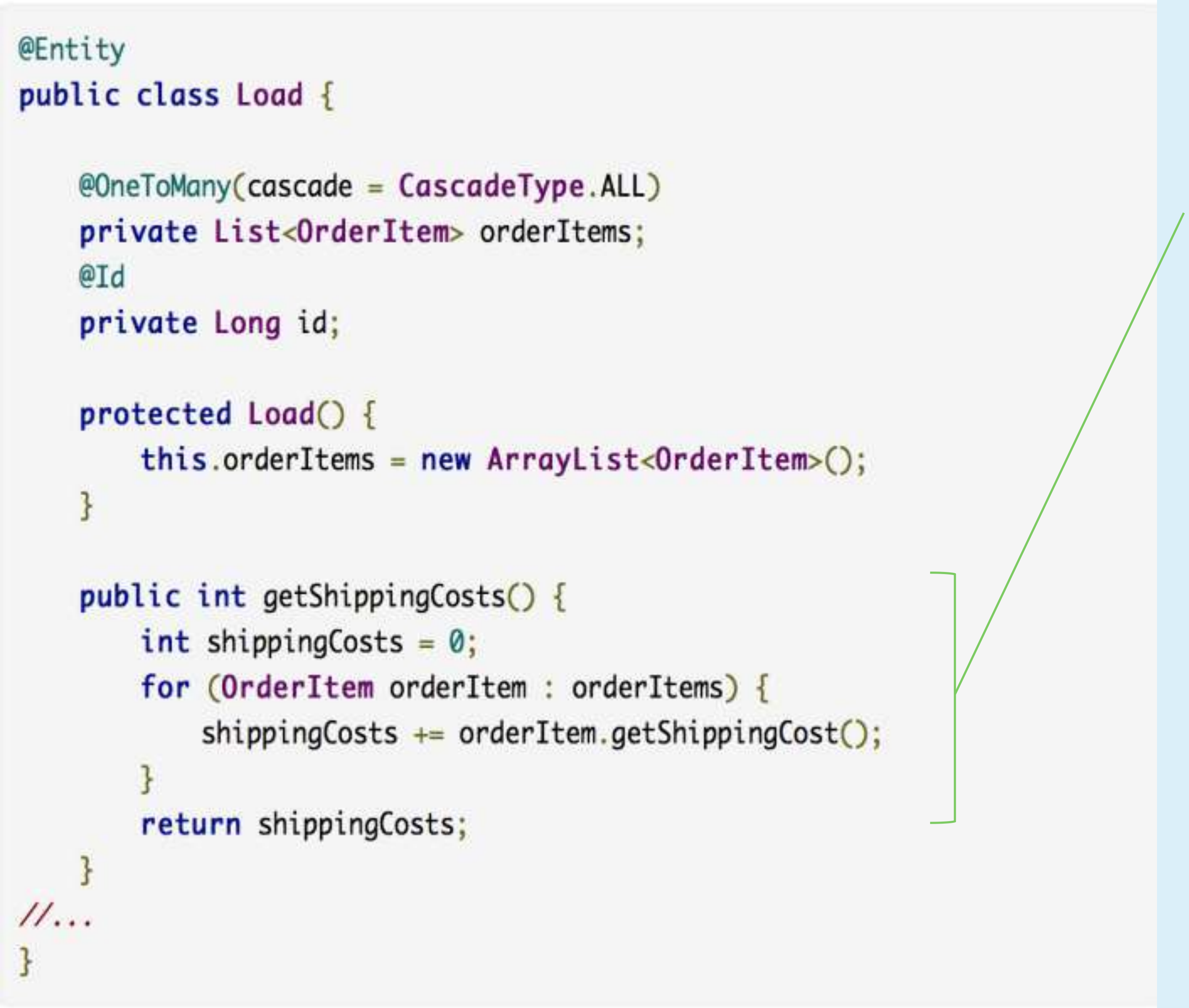
```java
@Stateless
public class ShipmentService {
  public final static int BASIC_COST = 5;

  @PersistenceContext
  private EntityManager em;

     public int getShippingCosts(int loadId) {
        Load load = em.find(Load.class, loadId);
        return computeShippingCost(load);
    }

    int computeShippingCost(Load load){
        int shippingCosts = 0;
        int weight = 0;
        int defaultCost = 0;
        for (OrderItem orderItem : load.getOrderItems()) {
            LoadType loadType = orderItem.getLoadType();
            weight = orderItem.getWeight();
            defaultCost = weight * 5;
            switch (loadType) {
                case BULKY:
                    shippingCosts += (defaultCost + 5);
                    break;
                case LIGHTWEIGHT:
                    shippingCosts += (defaultCost - 1);
                    break;
                case STANDARD:
                    shippingCosts += (defaultCost);
                    break;
                default:
               throw new IllegalStateException("Unknown type: " + loadType);
            }
        }
        return shippingCosts;
    }
}
```

1. Anemic Entity Structure
2. Massive IF Statements
3. Entire Logic resides in Service Layer
4. Type Dependent calculations are done based on conditional checks in Service Layer

Domain Driven Design with Java EE 6
By Adam Bien | Javaworld

Source: http://www.javaworld.com/article/2078042/java-app-dev/domain-driven-design-with-java-ee-6.html

Polymorphic Business Logic inside a Domain object



```java
@Entity
public class Load {

    @OneToMany(cascade = CascadeType.ALL)
    private List<OrderItem> orderItems;
    @Id
    private Long id;

    protected Load() {
        this.orderItems = new ArrayList<OrderItem>();
    }

    public int getShippingCosts() {
        int shippingCosts = 0;
        for (OrderItem orderItem : orderItems) {
            shippingCosts += orderItem.getShippingCost();
        }
        return shippingCosts;
    }
//...
}
```

Computation of the total cost realized inside a rich Persistent Domain Object (PDO) and not inside a service.

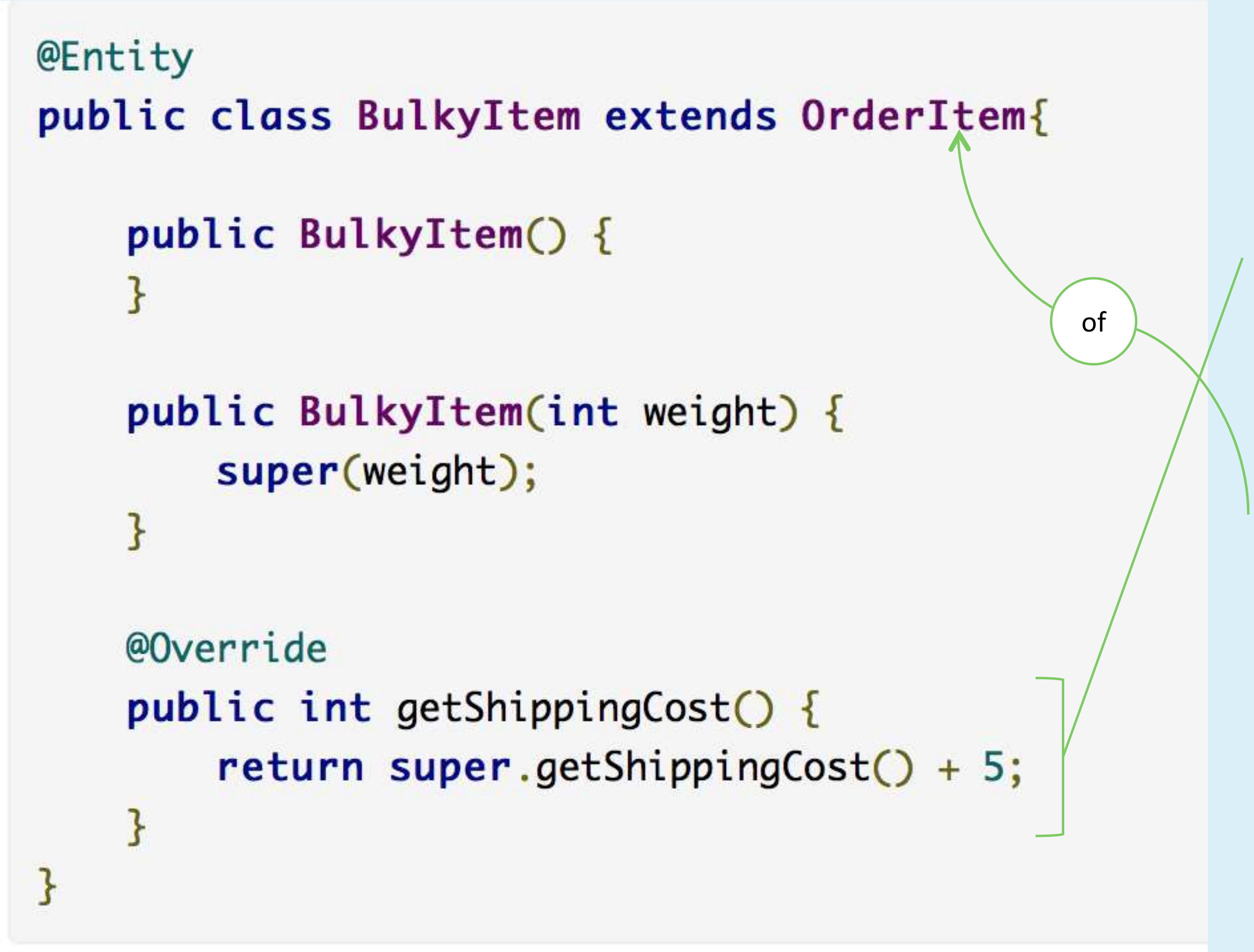
This simplifies creating very complex business rules.

Domain Driven Design with Java EE 6
By Adam Bien | Javaworld

Source: http://www.javaworld.com/article/2078042/java-app-dev/domain-driven-design-with-java-ee-6.html

08 July 2017

Type Specific Computation in a Sub Class



```
@Entity
public class BulkyItem extends OrderItem{

    public BulkyItem() {
    }

    public BulkyItem(int weight) {
        super(weight);
    }

    @Override
    public int getShippingCost() {
        return super.getShippingCost() + 5;
    }
}
```

We can change the computation of the shipping cost of a Bulky Item without touching the remaining classes.

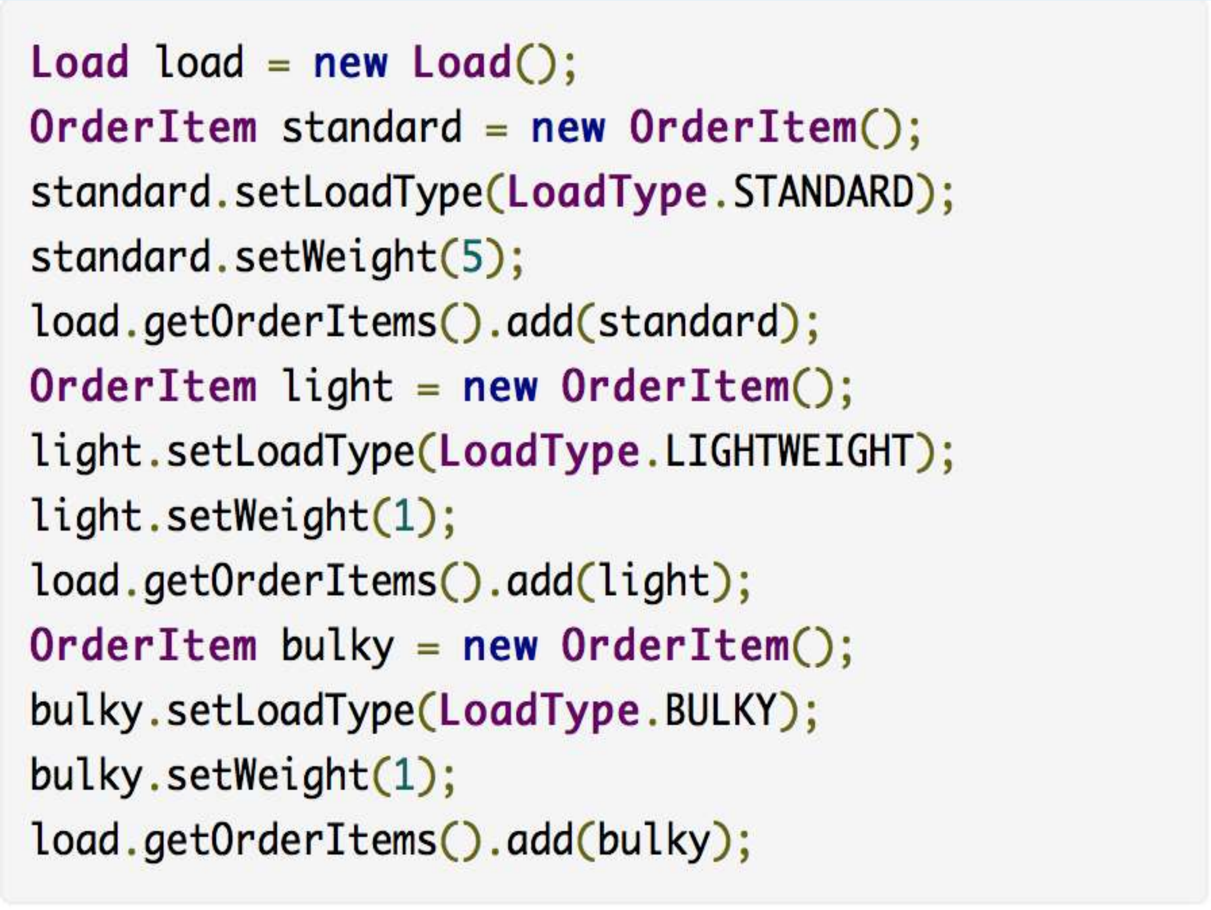
Its easy to introduce a new Sub Class without affecting the computation of the total cost in the Load Class.

Domain Driven Design with Java EE 6
By Adam Bien | Javaworld

Source: http://www.javaworld.com/article/2078042/java-app-dev/domain-driven-design-with-java-ee-6.html

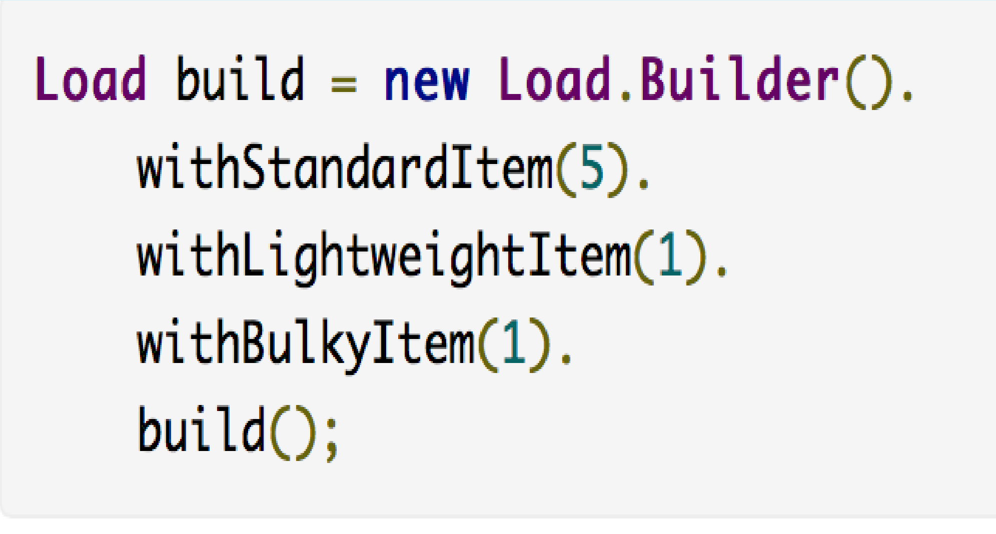
Object Construction : Procedural Way Vs. Builder Pattern

Procedural Way



```
Load load = new Load();
OrderItem standard = new OrderItem();
standard.setLoadType(LoadType.STANDARD);
standard.setWeight(5);
load.getOrderItems().add(standard);
OrderItem light = new OrderItem();
light.setLoadType(LoadType.LIGHTWEIGHT);
light.setWeight(1);
load.getOrderItems().add(light);
OrderItem bulky = new OrderItem();
bulky.setLoadType(LoadType.BULKY);
bulky.setWeight(1);
load.getOrderItems().add(bulky);
```

Builder Pattern



```
Load build = new Load.Builder().
    withStandardItem(5).
    withLightweightItem(1).
    withBulkyItem(1).
    build();
```

Domain Driven Design with Java EE 6
By Adam Bien | Javaworld

Source: http://www.javaworld.com/article/2078042/java-app-dev/domain-driven-design-with-java-ee-6.html

Scalability & Performance

1. In-Memory Computing
2. On Performance and Scalability
3. Low Latency and Java
4. Scalability Best Practices – eBay
5. CAP Theorem

Operational In-Memory Computing

Java Cache API : JSR 107 [Distributed Caching / Distributed Computing / Distributed Messaging]

Caching Technologies

- Hazelcast
- Oracle Coherence
- Infinispan
- Ehcache

- Apache Ignite
- GridGain
- GemFire

Cache Topology

Standalone

This setup consists of a single node containing all the cached data. It’s equivalent to a single-node cluster and does not collaborate with other running instances.

Distributed

Data is spread across multiple nodes in a cache such that only a single node is responsible for fetching a particular entry. This is possible by distributing/partitioning the cluster
in a balanced manner (i.e., all the nodes have the same number of entries and are hence load balanced). Failover is handled via configurable backups on each node.

Replicated

Data is spread across multiple nodes in a cache such that each node consists of the complete cache data, since each cluster node contains **all** the data; failover is not a concern.

Caching Mode

Embedded

When the cache and the application co-exist within the same JVM, the cache can be said to be operating in embedded mode. The cache lives and dies with the application JVM. This strategy should be used when:

- Tight coupling between your application and the cache is not a concern
- The application host has enough capacity (memory) to accommodate the demands of the cache

Client / Server

In this setup, the application acts as the client to a standalone (remote) caching layer. This should be leveraged when:

- The caching infrastructure and application need to evolve independently
- Multiple applications use a unified caching layer which can be scaled up without affecting client applications.

Caching Strategies

Read Through

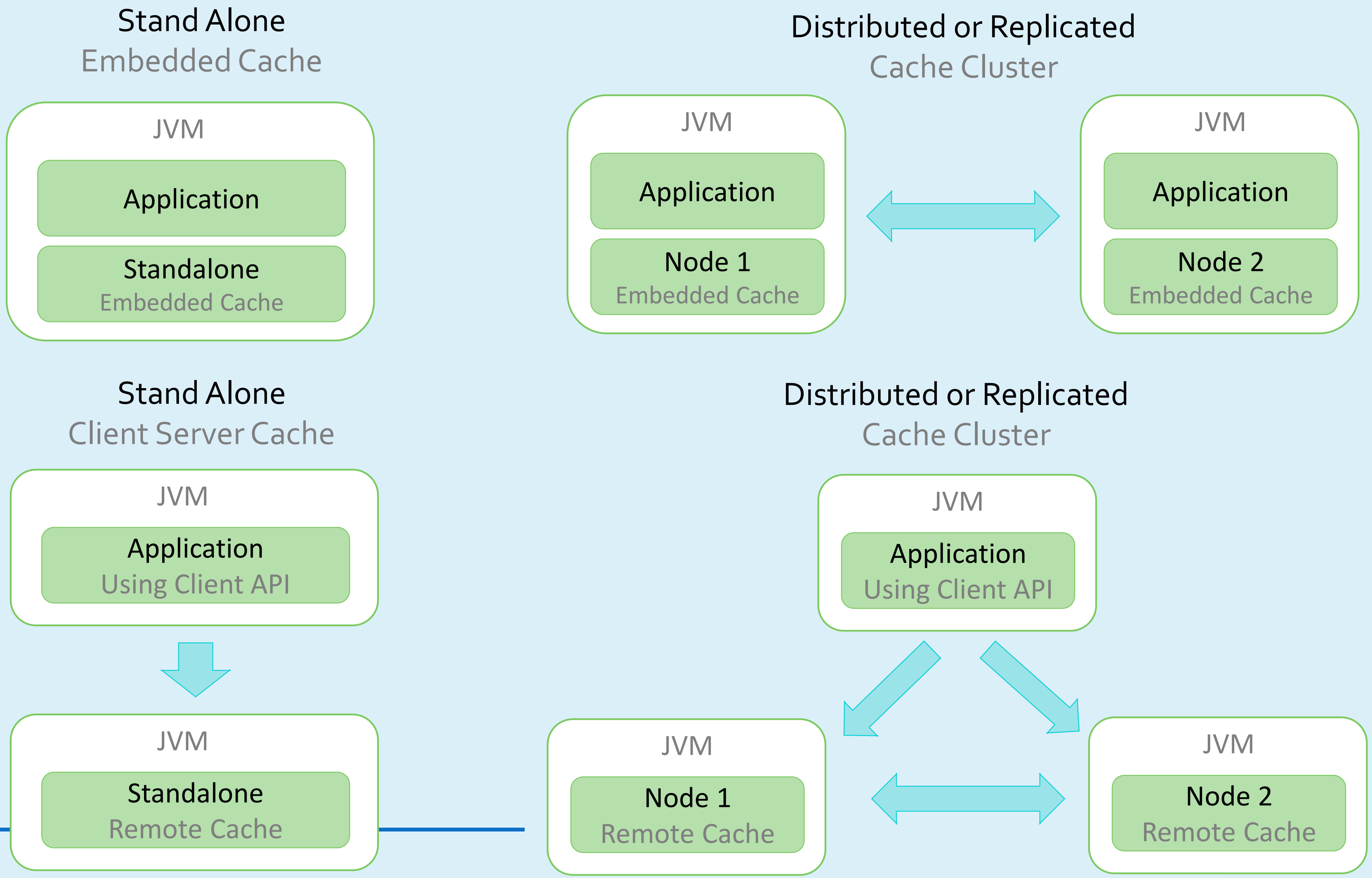
A process by which a missing cache entry is fetched from the integrated backend store.

Write Through

A process by which changes to a cache entry (create, update, delete) are pushed into the backend data store.
It is important to note that the business logic for *Read-Through* and *Write-Through* operations for a specific cache are confined within the caching layer itself. Hence, your application remains insulated from the specifics of the cache and its backing system-of-record.

Operational In-Memory Computing

Java Cache API : JSR 107 [Distributed Caching / Distributed Computing / Distributed Messaging]



Stand Alone
Embedded Cache

Distributed or Replicated
Cache Cluster

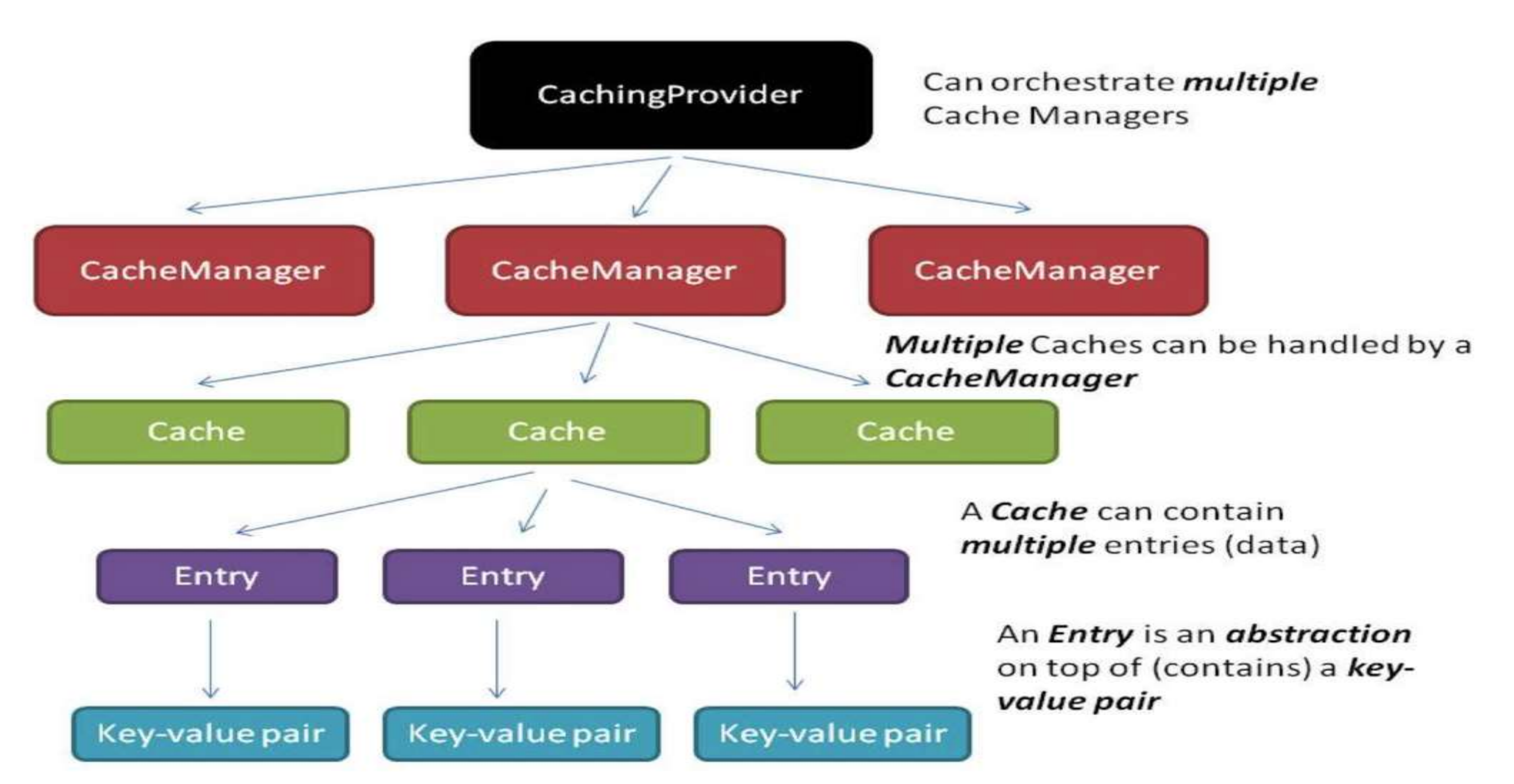
Stand Alone
Client Server Cache

Distributed or Replicated
Cache Cluster

08 July 2017

Operational In-Memory Computing

Java Cache API : JSR 107 [Distributed Caching / Distributed Computing / Distributed Messaging]



Can orchestrate ***multiple*** Cache Managers

Multiple Caches can be handled by a ***CacheManager***

A ***Cache*** can contain ***multiple*** entries (data)

An ***Entry*** is an ***abstraction*** on top of (contains) a ***key-value pair***

On Performance & Scalability

Stateless Architecture & Cloud Computing

Application Server

1. Thread Pools
2. JDBC Connection Pool Size
3. Compression for Data Transfer
4. Optimizing the CSS & JS files
5. Load Balancing for Web Server and App Server

Web Application

1. On Demand Loading (Modularized JS files)
2. Shared Nothing Architecture for Java
3. Stateless Architecture – No Session info is stored in App Server (Tomcat).
4. Session info is stored in the Database layer.
5. Optimize the Query based on Database Explain plans.
6. Aggregate Roots based on DDD
7. Command and Query Responsibility Segregation for better Read and Write operations.

Database – (Oracle)

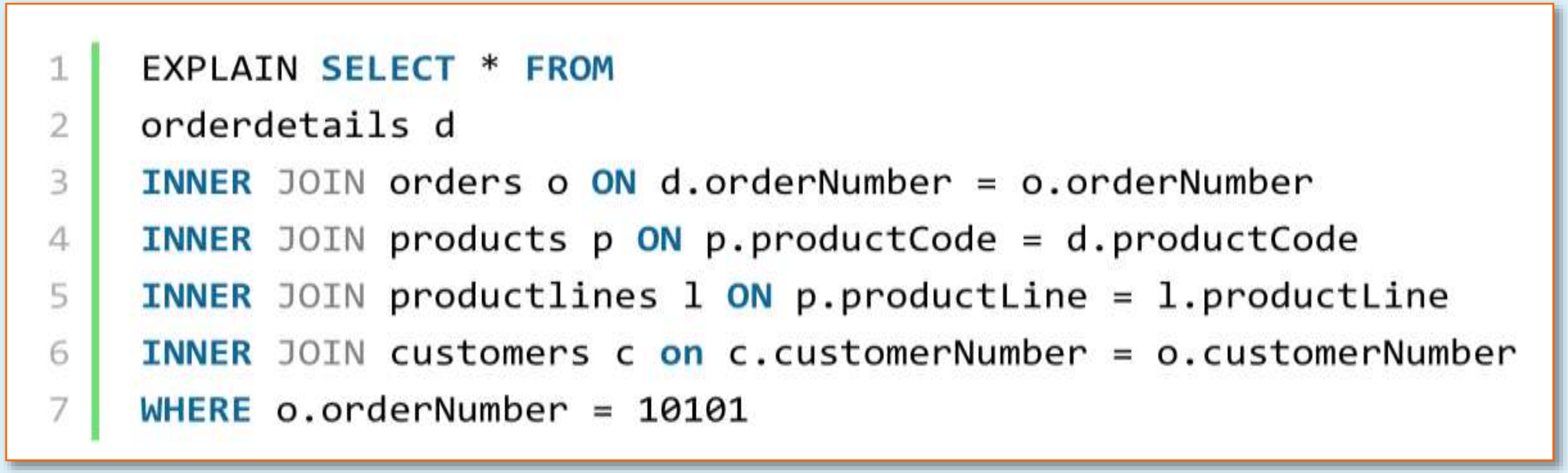
1. DB Block Size (8K for Transaction Processing and higher – 16K or 32K for Read intensive Operations)
2. SGA – System Global Area
3. In – Memory Column Store (Oracle 12c) in SGA
    1. Tables
    2. Materialized Views
    3. Partitions (All or Selective)
    4. Tablespace (@ this level, all tables and materialized views will be in Memory)

       ALTER SYSTEM SET INMEMORY_SIZE = 100G;
4. Memory Compression for Column Store
5. Query Optimization based on Explain Plan
    1. Cardinality (Estimated number of rows returned in each operations)
    2. Access Method (Table Scan or Index Scan)
    3. Join Method (Hash, Sort, Merger)
    4. Join Order (Order in which tables are joined)
    5. Partitioning (Access only required partitions)
    6. Parallel Execution
    - Examples -> Next

08 July 2017

SQL Query Performance : Explain Plan MySQL

Table	Rows
Order	326
Order Details	2996
Products	110
Product Lines	7
Customers	122



```sql
EXPLAIN SELECT * FROM
orderdetails d
INNER JOIN orders o ON d.orderNumber = o.orderNumber
INNER JOIN products p ON p.productCode = d.productCode
INNER JOIN productlines l ON p.productLine = l.productLine
INNER JOIN customers c on c.customerNumber = o.customerNumber
WHERE o.orderNumber = 10101
```

If You don't have Primary Indexes, MySQL Explain Plan shows the following……. ☺

What database does?	7 × 110 × 122 × 326 × 2996	91,750,822,240 Records

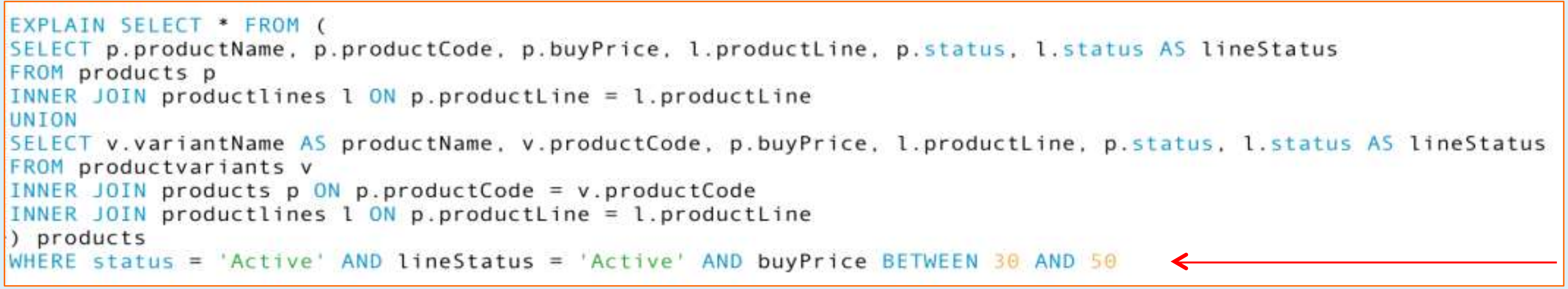
Table	Rows
Order	1
Order Details	4
Products	1
Product Lines	1
Customers	1

After Adding Primary Key Index

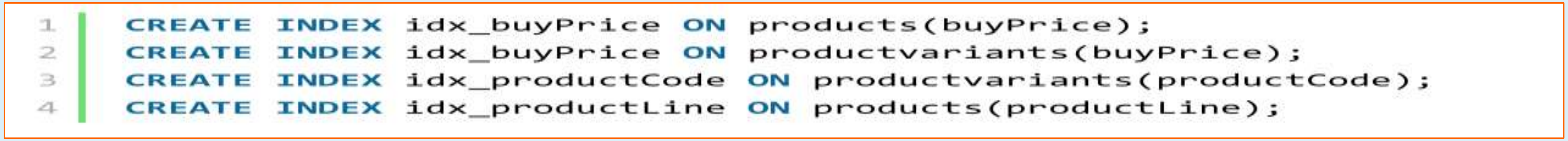
1 x 1 x 4 x 1 x 1	4 Records

Source: http://www.sitepoint.com/using-explain-to-write-better-mysql-queries/

Just adding Indexes is not enough!



```
EXPLAIN SELECT * FROM (
SELECT p.productName, p.productCode, p.buyPrice, l.productLine, p.status, l.status AS lineStatus
FROM products p
INNER JOIN productlines l ON p.productLine = l.productLine
UNION
SELECT v.variantName AS productName, v.productCode, p.buyPrice, l.productLine, p.status, l.status AS lineStatus
FROM productvariants v
INNER JOIN products p ON p.productCode = v.productCode
INNER JOIN productlines l ON p.productLine = l.productLine
) products
WHERE status = 'Active' AND lineStatus = 'Active' AND buyPrice BETWEEN 30 AND 50
```



```
CREATE INDEX idx_buyPrice ON products(buyPrice);
CREATE INDEX idx_buyPrice ON productvariants(buyPrice);
CREATE INDEX idx_productCode ON productvariants(productCode);
CREATE INDEX idx_productLine ON products(productLine);
```

Explain Plan Result

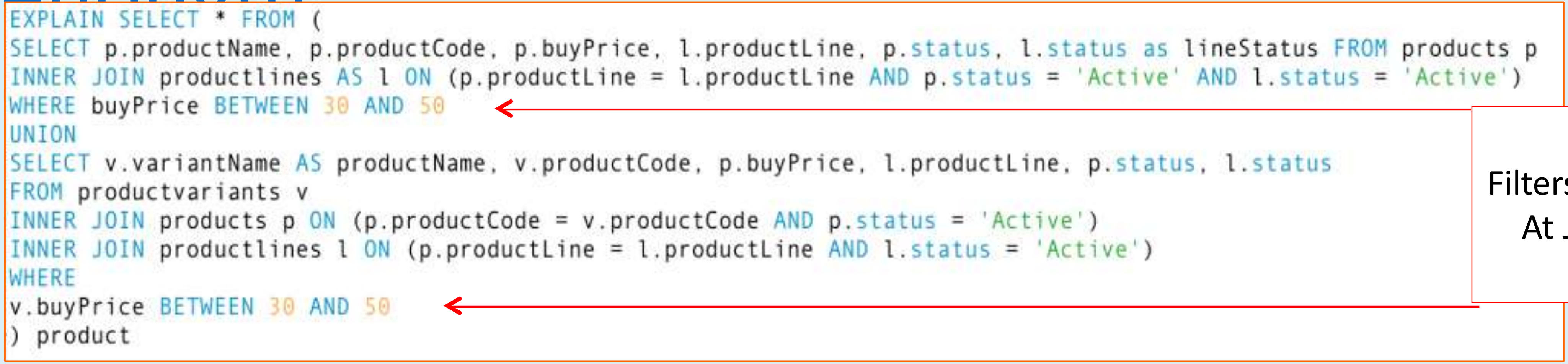
#	Select Type	Table	Possible Keys	Reference	Rows
1	Primary	Derived			219
2	Derived	Products			110
3	Derived	Product Lines	Primary	Classicmodels.p.productline	1
4	Union	Product Variant			109
5	Union	Products	Primary	Classicmodels.v.productCode	1
6	Union	Product Line	Primary	Classicmodels.p.productline	1
7	Union Result	Union 2 and 3			

Number of records scanned	219 x 110 x 1 x 109 x 1 x 1	2,625, 810 records

Source: http://www.sitepoint.com/using-explain-to-write-better-mysql-queries/

08 July 2017

Just adding Indexes is not enough!



```
EXPLAIN SELECT * FROM (
SELECT p.productName, p.productCode, p.buyPrice, l.productLine, p.status, l.status as lineStatus FROM products p
INNER JOIN productlines AS l ON (p.productLine = l.productLine AND p.status = 'Active' AND l.status = 'Active')
WHERE buyPrice BETWEEN 30 AND 50
UNION
SELECT v.variantName AS productName, v.productCode, p.buyPrice, l.productLine, p.status, l.status
FROM productvariants v
INNER JOIN products p ON (p.productCode = v.productCode AND p.status = 'Active')
INNER JOIN productlines l ON (p.productLine = l.productLine AND l.status = 'Active')
WHERE
v.buyPrice BETWEEN 30 AND 50
) product
```

Filters Data At Join

08 July 2017

Explain Plan Result

#	Select Type	Table	Possible Keys	Reference	Rows
1	Primary	Derived2			12
2	Derived	Products	idx_buyPrice, idx_productLine		23
3	Derived	Product Lines	Primary	Classicmodels.p.productline	1
4	Union	Product Variant	idx_buyPrice, idx_productCode		1
5	Union	Products	Primary, idx_productLine	Classicmodels.v.productCode	1
6	Union	Product Lines	Primary	Classicmodels.p.productline	1
7	Union Result	Union 2, 3			

Number of records scanned	12 x 23	276 Records

Source: http://www.sitepoint.com/using-explain-to-write-better-mysql-queries/

Low Latency and Java : 100 Nano Seconds to 100 Milliseconds

1. Immutable Objects
2. Share Nothing Architecture
3. Stateless Architecture
4. Lock Free and Wait Free Patterns

Zing Java Virtual Machine

- Pause less Garbage Collection
- C4 Garbage Collection Algorithm
- Elastic Java Heap Memory
- Tuned for Low Latency Apps

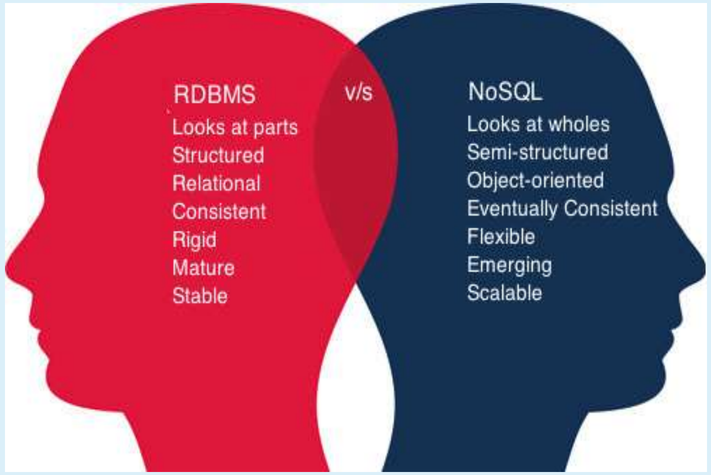
JVM	Collector Name	Young Gen	Old Gen
Oracle JVM	ParallelGC	Monolithic, stop the world, copying	Monolithic, stop-the-world, Mark/Sweep/Compact
	CMS (Concurrent / Mark / Sweep)	Monolithic, stop the world, copying	Mostly concurrent marker, concurrent, non-compacting sweeper, fall back to monolithic stop-the-world compaction
	G1 (Garbage First)	Monolithic, stop the world, copying	Mostly concurrent marker, mostly incremental compaction, fall back to monolithic stop-the-world
Oracle Jrockit	Dynamic Garbage Collector	Monolithic, stop the world, copying	Mark/Sweep - can choose mostly concurrent or parallel, incremental compaction, fall back to monolithic stop-the-world
IBM J9	Balanced	Monolithic, stop the world, copying	Mostly concurrent marker, mostly incremental compaction, fall back to monolithic stop-the-world
	Optthruput	Monolithic, stop the world, copying	Parallel Mark/Sweep, stop-the-world compaction
Zing	C4	Concurrent, Compacting	C4 – Continuously Concurrent Compacting Collector

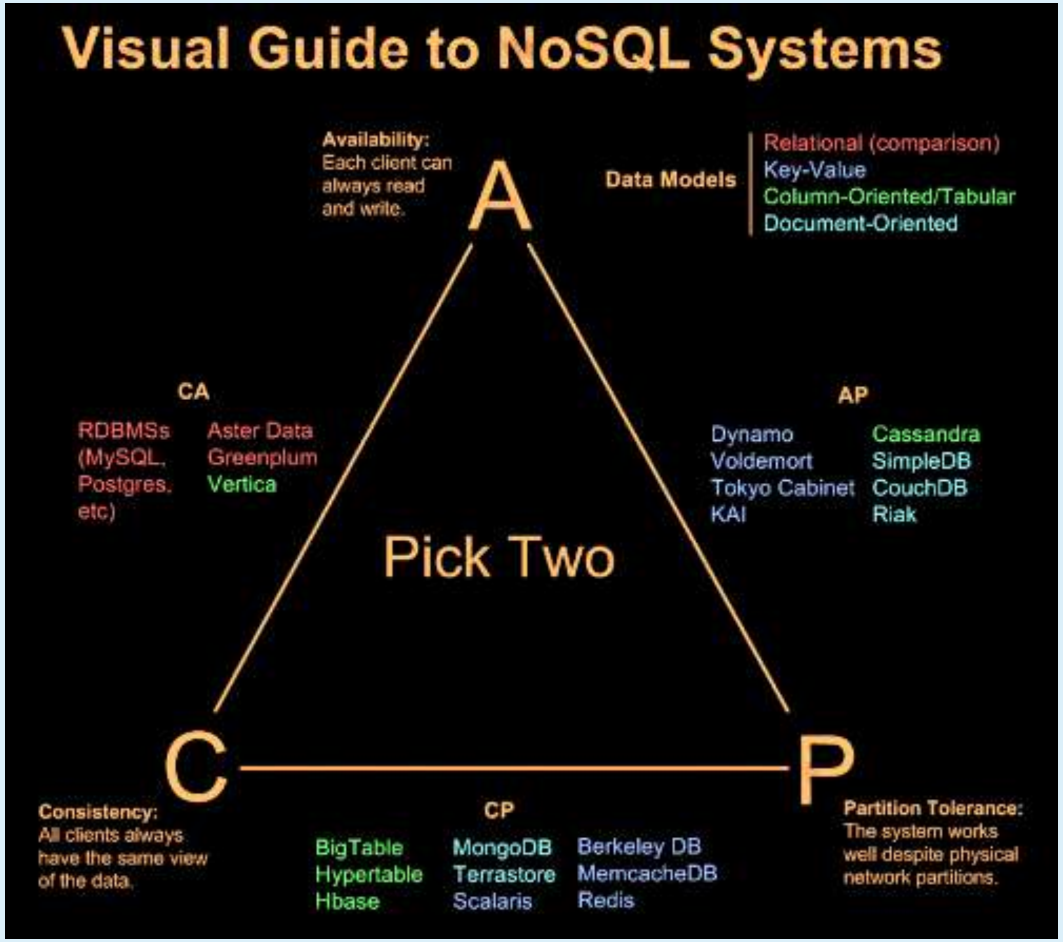
CAP Theorem

by Eric Allen Brewer

Pick Any 2!!

Say NO to 2 Phase Commit ☺





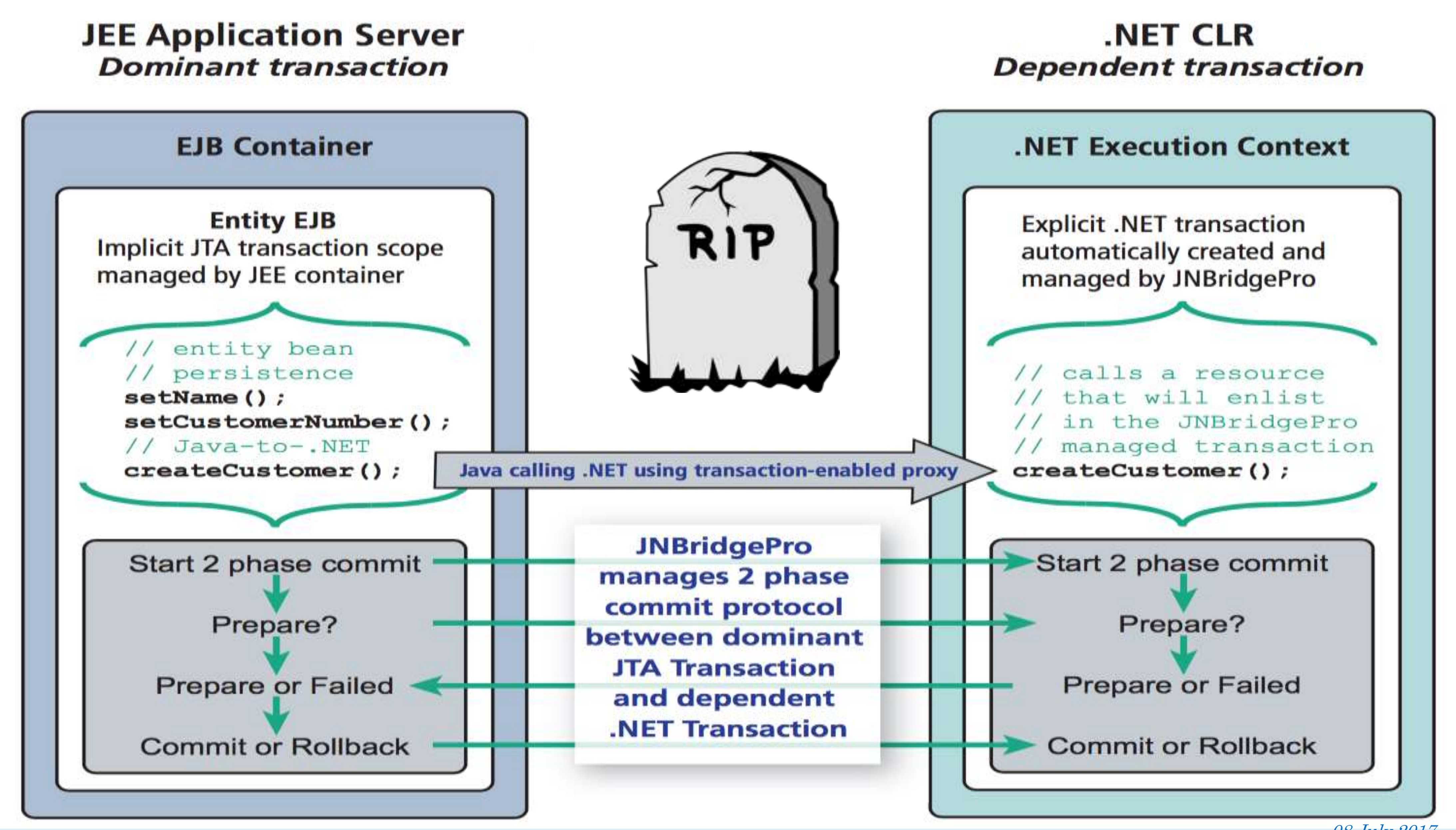
Source: http://en.wikipedia.org/wiki/Eric_Brewer_(scientist)

08 July 2017

“In a network subject to communication failures, it is impossible for ant web service to implement an atomic read / write shared memory that guarantees a response to every request.”

CAP 12 years later: How the “Rules have changed” : http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

Java & .NET 2 PC



JEE Application Server
Dominant transaction

.NET CLR
Dependent transaction

EJB Container

Entity EJB
Implicit JTA transaction scope managed by JEE container

```
// entity bean
// persistence
setName();
setCustomerNumber();
// Java-to-.NET
createCustomer();
```

.NET Execution Context

Explicit .NET transaction automatically created and managed by JNBridgePro

```
// calls a resource
// that will enlist
// in the JNBridgePro
// managed transaction
createCustomer();
```

Java calling .NET using transaction-enabled proxy

JNBridgePro manages 2 phase commit protocol between dominant JTA Transaction and dependent .NET Transaction

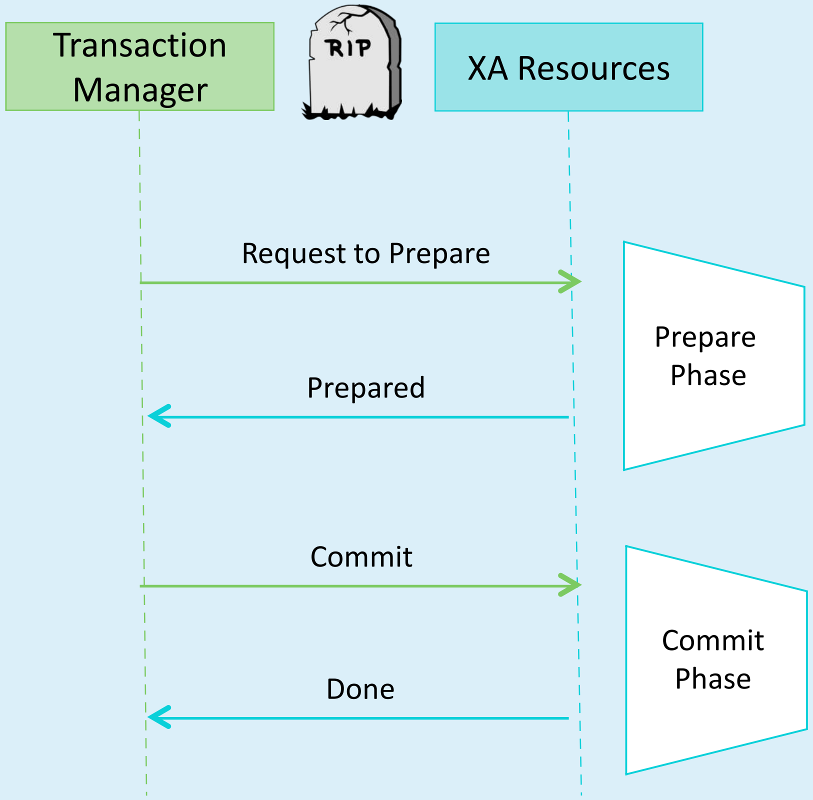
08 July 2017

Distributed Transactions : 2 Phase Commit

2 PC or not 2 PC, Wherefore Art Thou XA?

How does 2PC impact scalability?

- Transactions are committed in two phases.
- This involves communicating with every database (XA Resources) involved to determine if the transaction will commit in the first phase.
- During the second phase each database is asked to complete the commit.
- While all of this coordination is going on, locks in all of the data sources are being held.
- ***The longer duration locks create the risk of higher contention.***
- *Additionally, **the two phases require more database processing time than a single phase commit.***
- **The result is lower overall TPS in the system.**



Solution : Resilient System

- Event Based
- Design for failure
- Asynchronous Recovery
- Make all operations idempotent.
- Each DB operation is a 1 PC

Source : Pat Helland (Amazon) : Life Beyond Distributed Transactions Distributed Computing : http://dancres.github.io/Pages/

Stored Procedures :

Procedural Languages

Database	Languages
DB2	SQL PL or Java
MS SQL Server	T-SQL or .Net
MySQL	SQL/PSM
Oracle	PL/SQL or Java
PostgreSQL	PL/pgSQL
Sybase	T-SQL

1. Vendor Specific Code base, Portability Issues.
2. No Abstraction of Business Logic from Persistence Layer.
3. Based on 25 years OLD client server architecture.
4. Scalability Issues in Cloud Computing era.
5. You are still in the Procedural world!

Pros for Stored Procedure		In Reality
Performance	→	Why Big Data? (Like No SQL, Mongo DB……)
Security	→	Remember Target credit card breach in 2014!
		Scalability is a big issue

Stored Procedures are BAD! Period

If you are still in the world of stored procedures, you are creating legacy Apps NOW!!!!

Scalability Best Practices : Lessons from eBay

08 July 2017

	Best Practices	Highlights
#1	Partition By Function	• Decouple the Unrelated Functionalities. • Selling functionality is served by one set of applications, bidding by another, search by yet another. • 16,000 App Servers in 220 different pools • 1000 logical databases, 400 physical hosts
#2	Split Horizontally	• Break the workload into manageable units. • eBay's interactions are stateless by design • All App Servers are treated equal and none retains any transactional state • Data Partitioning based on specific requirements
#3	Avoid Distributed Transactions	• 2 Phase Commit is a pessimistic approach comes with a big COST • **CAP Theorem** (Consistency, Availability, Partition Tolerance). Apply any two at any point in time. • @ eBay No Distributed Transactions of any kind and NO 2 Phase Commit.
#4	Decouple Functions Asynchronously	• If Component A calls component B synchronously, then they are tightly coupled. For such systems to scale A you need to scale B also. • If Asynchronous A can move forward irrespective of the state of B • SEDA (Staged Event Driven Architecture)
#5	Move Processing to Asynchronous Flow	• Move as much processing towards Asynchronous side • Anything that can wait should wait
#6	Virtualize at All Levels	• Virtualize everything. eBay created their on O/R layer for abstraction
#7	Cache Appropriately	• Cache Slow changing, read-mostly data, meta data, configuration and static data.

Source: http://www.infoq.com/articles/ebay-scalability-best-practices

Thread Safety

1. Threat Safety
2. Dependency Injection
3. Service Scope
4. Generics & Type Safety

Thread Safety – Shared Nothing

Re-Entrancy

Execution by a Thread	**Result**
• Partial Execution • Re-Executed by the Same Thread • Simultaneously executed by another thread	Correctly completes the original execution. Thread Safe
This requires the saving of info in variables local to each execution. No Static or Global or other Non-Local States.	**Variables on Stack ONLY** Method only Variables

Thread-Local Storage

Execution by a Thread	**Result**
• Variables are localized • Each Thread has its own private copy • Variables retains values across sub routines and other code boundaries. • Allows simultaneously executed by another thread.	Correctly completes the original execution. Thread Safe

Thread Safety – Locks and Synchronization

Mutual Exclusion

Execution by a Thread	Improper Usage will result in
Access to shared data is serialized using mechanisms that ensure only one thread reads or writes to the shared data at any time.	Deadlocks Resource Starvation

Atomic Operations

Execution by a Thread	Examples in Java
• Shared data access cannot be interrupted by threads. • Since the operation is atomic shared data is always kept in valid state. • Atomic operations are basis for many thread locking mechanisms and used to implement MutEx. • Usually requires machine language instructions	Atomic Boolean Atomic Integer Atomic Long Atomic Reference …

Immutable Objects

Object State	Result
• Object State cannot be changed after construction • Read-Only data is shared • Mutations are allowed using Builder Pattern creating a new Object. • Entities can be created as Immutable objects.	Thread Safe

Dependency Injection and Service Scope

REST Service, Domain Service (Business Rules), Repository

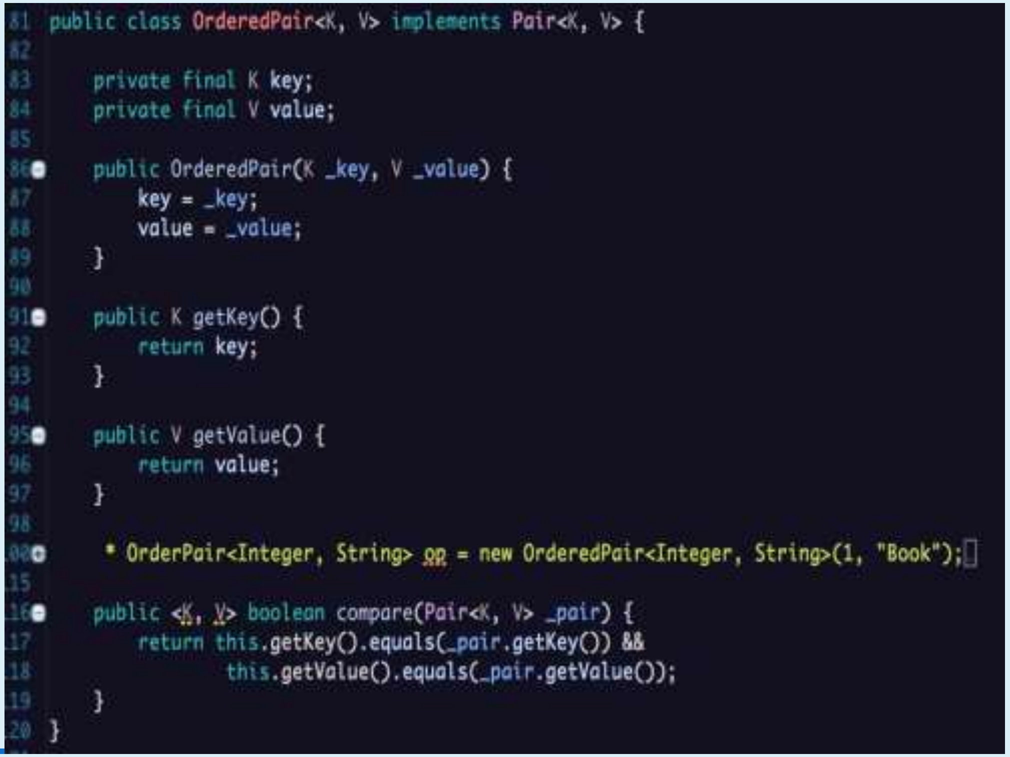
	Bean Scope Definition	Thread Safety
Request	The bean is created every time when a new request comes in.	Thread Safe

	Bean Scope Definition	Thread Safety
Session	• The bean is created for the user’s session. • Asynchronous access from a Web 2.0 app can corrupt the data (multiple tabs). • Developers need to ensure that shared state is NOT corrupt.	Not Thread Safe

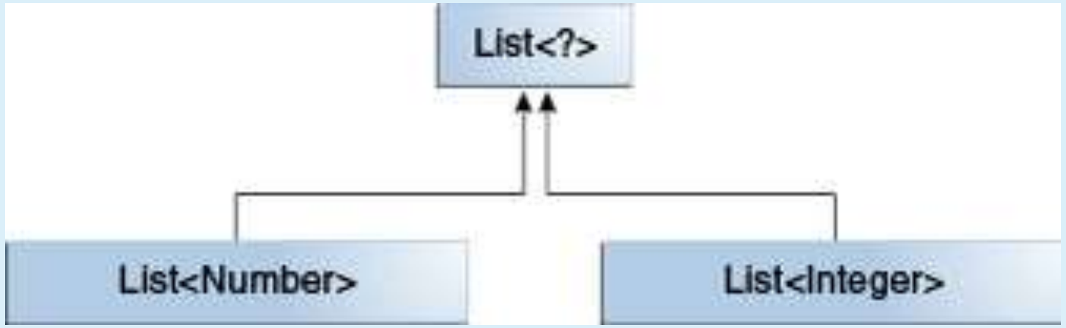
	Bean Scope Definition	Thread Safety
Singleton	• Only ONE instance of the bean is created. • Any Bean without a STATE can be singleton. • Asynchronous access from a Web 2.0 app can corrupt the data (multiple tabs). • Developers need to ensure that shared state is NOT corrupt.	Not Thread Safe

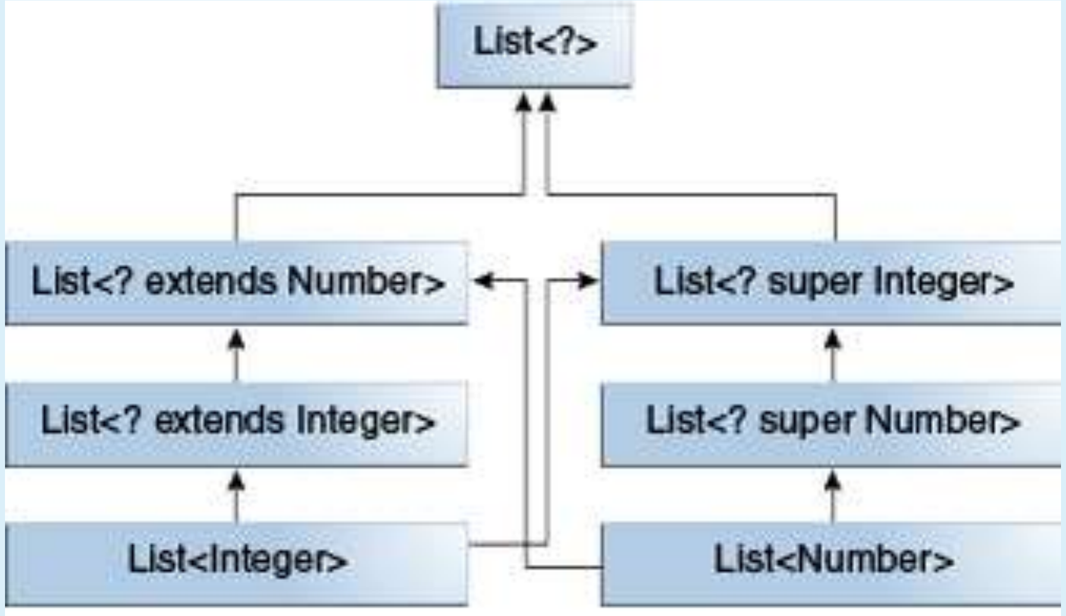
Generics and Type Safety

- Platform automatically takes care of the Type Safety for all the layers in the Architecture
  1. App Service Layer (REST)
  2. Domain Service (Business Rules)
  3. Repository (Data Layer)



```java
public class OrderedPair<K, V> implements Pair<K, V> {

    private final K key;
    private final V value;

    public OrderedPair(K _key, V _value) {
        key = _key;
        value = _value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }

     * OrderPair<Integer, String> op = new OrderedPair<Integer, String>(1, "Book");

    public <K, V> boolean compare(Pair<K, V> _pair) {
        return this.getKey().equals(_pair.getKey()) &&
               this.getValue().equals(_pair.getValue());
    }
}
```





Source: https://docs.oracle.com/javase/tutorial/java/generics/subtyping.html

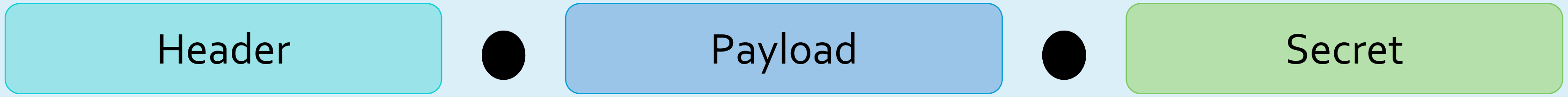
- Type Inference

- Upper bounded Wildcards
- Lower bounded Wildcards
- Unbounded Wildcards

JSON WEB TOKENS

Secure RESTful API's – Stateless

JSON Web Tokens : http://jwt.io



- JSON Web Tokens (JWT)
- Pronounced "jot", are a standard since the information they carry is transmitted via JSON. We can read more about the draft, but that explanation isn't the most pretty to look at.
- JSON Web Tokens work across different programming languages:
- JWTs work in .NET, Python, Node.JS, Java, PHP, Ruby, Go, JavaScript, and Haskell. So you can see that these can be used in many different scenarios.
- JWTs are self-contained:
- They will carry all the information necessary within itself. This means that a JWT will be able to transmit basic information about itself, a payload (usually user information), and a signature.
- JWTs can be passed around easily:
- Since JWTs are self-contained, they are perfectly used inside an HTTP header when authenticating an API. You can also pass it through the URL.

JSON Web Signature
JWS: RFC 7515

JSON Web Encryption
JWE : RFC 7516

JSON Web Key
JWK : RFC 7517

JSON Web Algorithms
JWA : RFC 7518

JSON Web Token
JWT: RFC 7519

Authentication Traditional Way

The Problems with Server Based Authentication. A few major problems arose with this method of authentication.

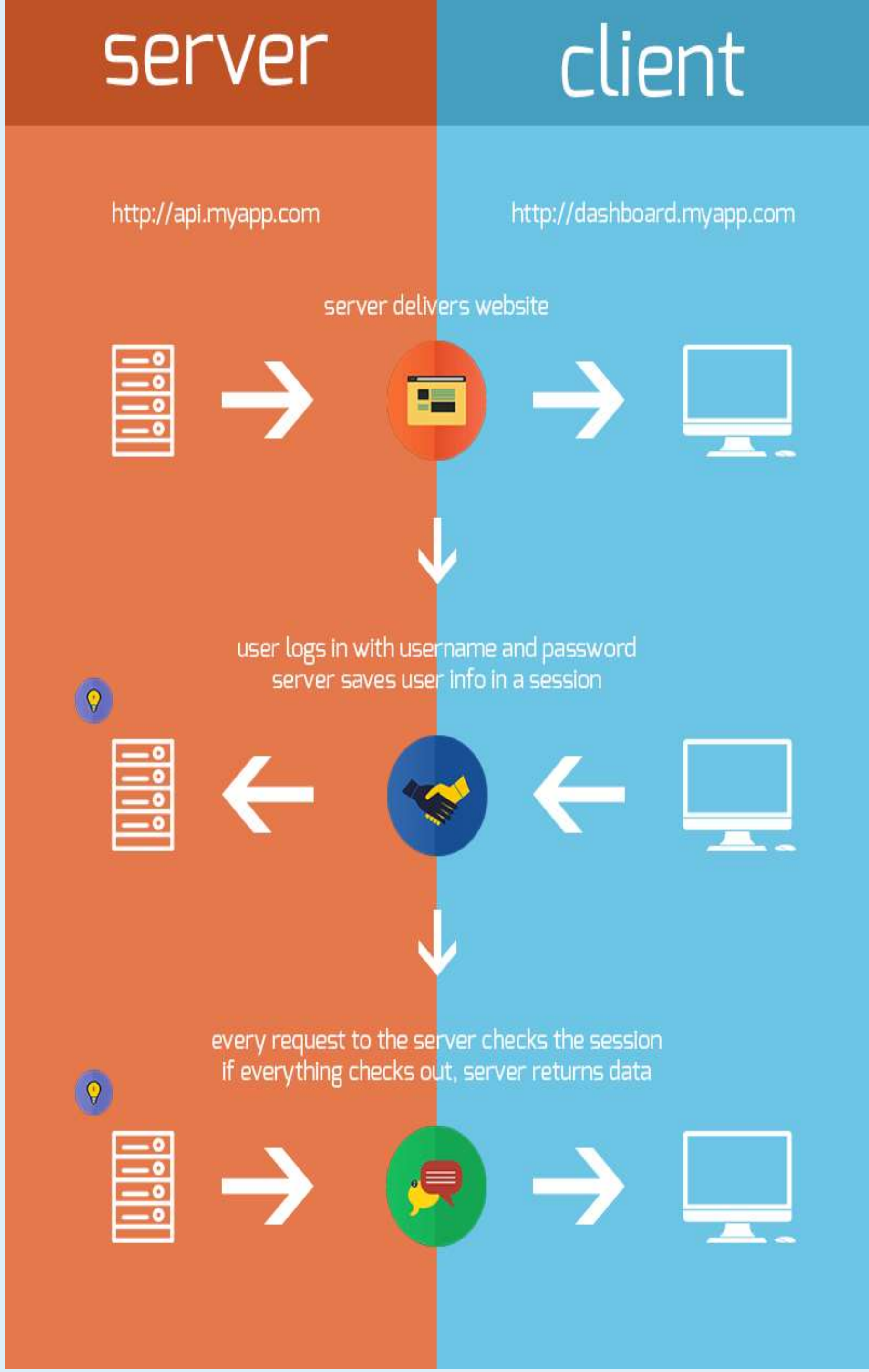
Sessions: Every time a user is authenticated, the server will need to create a record somewhere on our server. This is usually done in memory and when there are many users authenticating, the overhead on your server increases.

Scalability: Since sessions are stored in memory, this provides problems with scalability. As our cloud providers start replicating servers to handle application load, having vital information in session memory will limit our ability to scale.

CORS: As we want to expand our application to let our data be used across multiple mobile devices, we have to worry about cross-origin resource sharing (CORS). When using AJAX calls to grab resources from another domain (mobile to our API server), we could run into problems with forbidden requests.

CSRF: We will also have protection against cross-site request forgery (CSRF). Users are susceptible to CSRF attacks since they can already be authenticated with say a banking site and this could be taken advantage of when visiting other sites.

With these problems, scalability being the main one, it made sense to try a different approach.


server
client
http://api.myapp.com
http://dashboard.myapp.com
server delivers website
user logs in with username and password
server saves user info in a session
every request to the server checks the session
if everything checks out, server returns data

Source: https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication

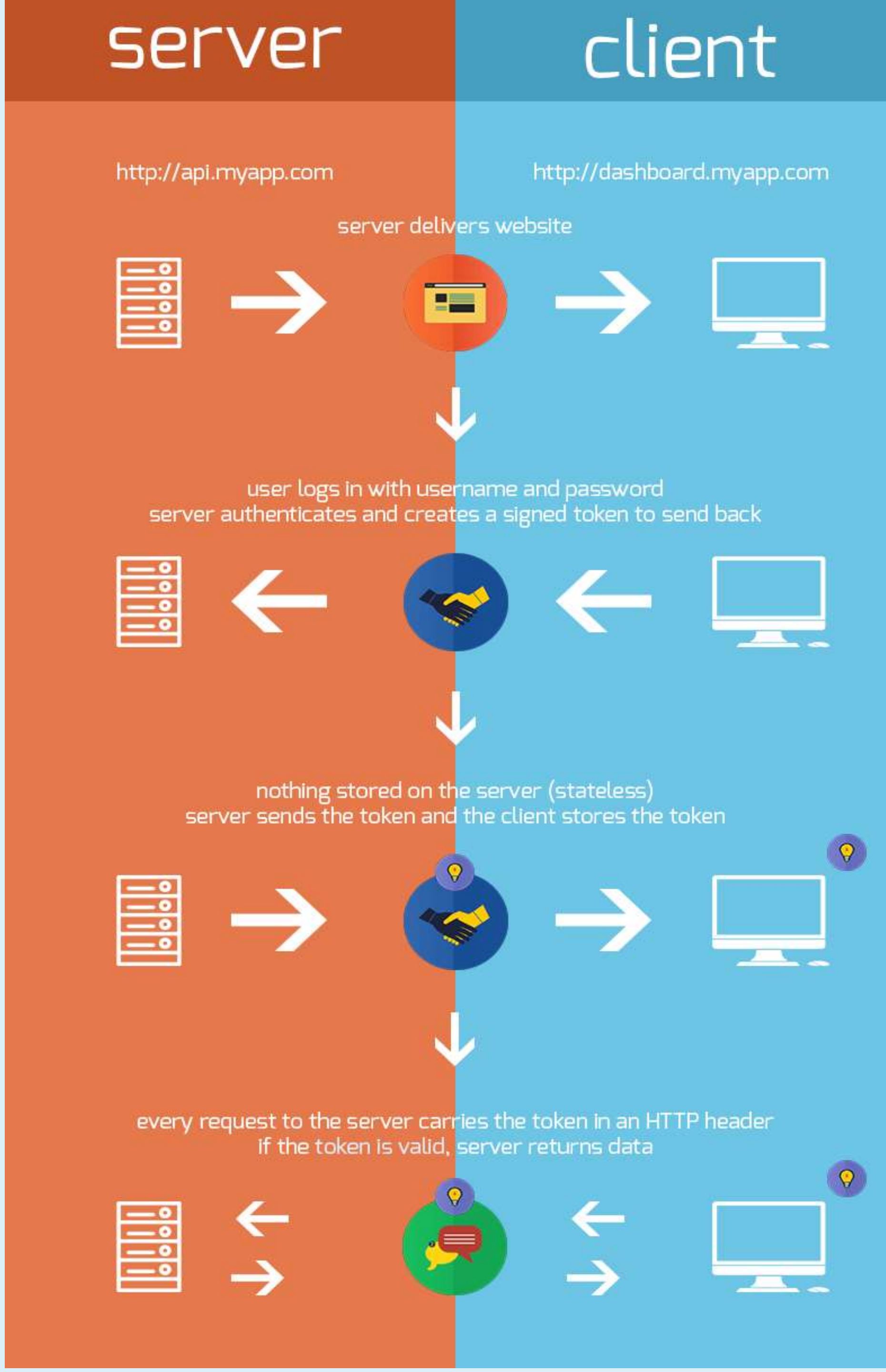
JSON Web Token: Stateless Architecture

How it Works

Token based authentication is stateless. We are not storing any information about our user on the server or in a session.

1. User Requests Access with Username / Password
2. Application validates credentials
3. Application provides a signed token to the client
4. Client stores that token and sends it along with every request
5. Server verifies token and responds with data

Every single request will require the token. This token should be sent in the HTTP header so that we keep with the idea of stateless HTTP requests. We will also need to set our server to accept requests from all domains using Access-Control-Allow-Origin: *. What's interesting about designating * in the ACAO header is that it does not allow requests to supply credentials like HTTP authentication, client-side SSL certificates, or cookies.


server
client
http://api.myapp.com
http://dashboard.myapp.com
server delivers website
user logs in with username and password
server authenticates and creates a signed token to send back
nothing stored on the server (stateless)
server sends the token and the client stores the token
every request to the server carries the token in an HTTP header
if the token is valid, server returns data

Source: https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication

JSON Web Token : Benefits

Stateless and Scalable

Tokens stored on client side. Completely stateless, and ready to be scaled. Our load balancers are able to pass a user along to any of our servers since there is no state or session information anywhere.

Security

The token, not a cookie, is sent on every request and since there is no cookie being sent, this helps to prevent CSRF attacks. Even if your specific implementation stores the token within a cookie on the client side, the cookie is merely a storage mechanism instead of an authentication one. There is no session based information to manipulate since we don't have a session!

Extensibility (Friend of A Friend and Permissions)

Tokens will allow us to build applications that share permissions with another. For example, we have linked random social accounts to our major ones like Facebook or Twitter.

Multiple Platforms and Domains

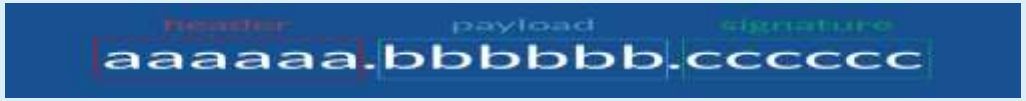
When the application and service expands, we will need to be providing access to all sorts of devices and applications (since the app will most definitely become popular!).

Having the App API just serve data, we can also make the design choice to serve assets from a CDN. This eliminates the issues that CORS brings up after we set a quick header configuration for our application.

Standards Based

Source: https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication

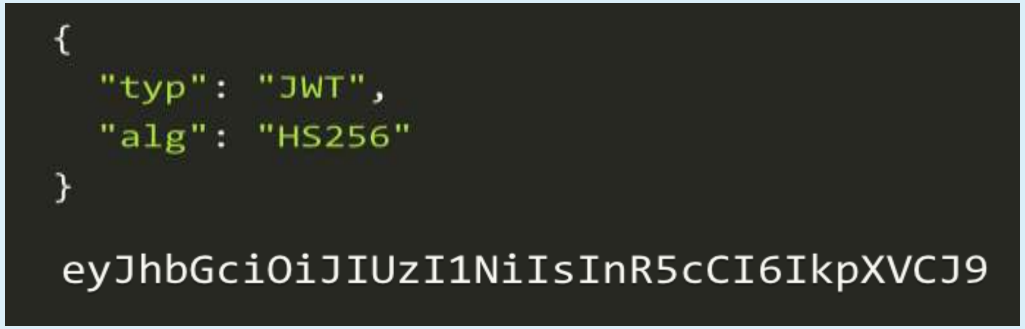
JSON Web Token : Anatomy



Header

The header carries 2 parts:

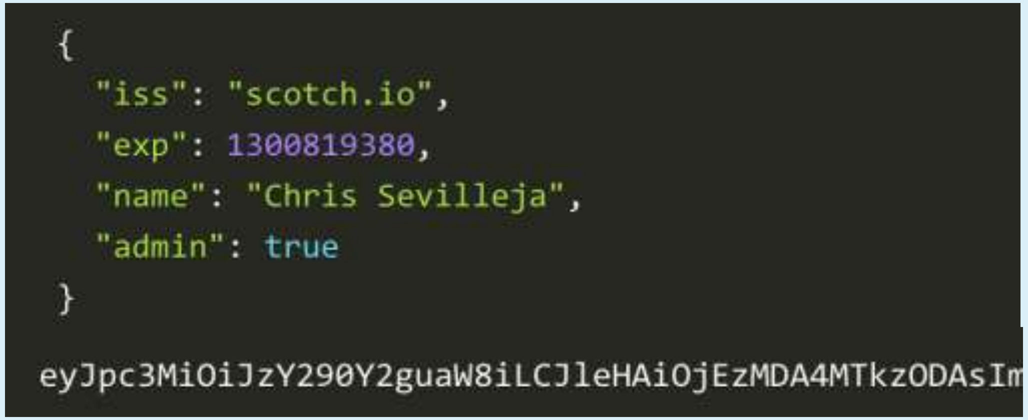
1. declaring the type, which is JWT
2. the hashing algorithm to use (HMAC SHA256 in this case)



```
{
  "typ": "JWT",
  "alg": "HS256"
}

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Payload

The payload will carry the bulk of our JWT, also called the JWT Claims. This is where we will put the information that we want to transmit and other information about our token.



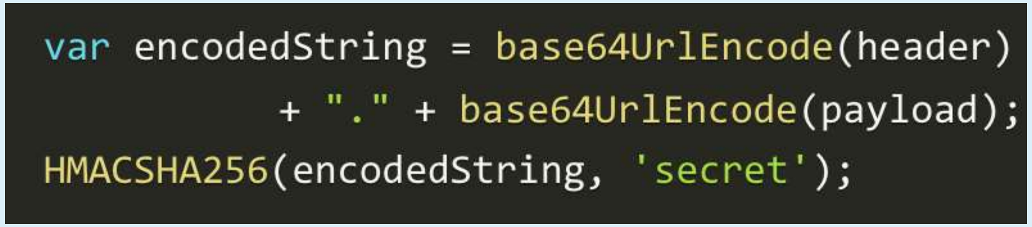
```
{
  "iss": "scotch.io",
  "exp": 1300819380,
  "name": "Chris Sevilleja",
  "admin": true
}

eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOjEzMDA4MTkzODAsIm
```

Secret

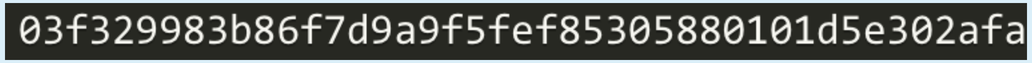
This signature is made up of a hash of the following components:

1. The header
2. The payload
3. Secret

The secret is the signature held by the server. This is the way that our server will be able to verify existing tokens and sign new ones.
This gives us the final part of our JWT.



```
var encodedString = base64UrlEncode(header)
           + "." + base64UrlEncode(payload);
HMACSHA256(encodedString, 'secret');
```



```
03f329983b86f7d9a9f5fef85305880101d5e302afa
```

eyJhbGcioiJI---.eyJpc3Mioi---.03f32998---

Header ● Payload ● Secret

JSON Web Token : Anatomy – The Claims

Registered Claims

Claims that are not mandatory whose names are reserved for us. These include:

- iss: The issuer of the token
- sub: The subject of the token
- aud: The audience of the token
- exp: This will probably be the registered claim most often used. This will define the expiration in Numeric Date value. The expiration MUST be after the current date/time.
- nbf: Defines the time before which the JWT MUST NOT be accepted for processing
- iat: The time the JWT was issued. Can be used to determine the age of the JWT
- jti: Unique identifier for the JWT. Can be used to prevent the JWT from being replayed. This is helpful for a one time use token.

Public Claims

These are the claims that we create ourselves like user name, information, and other important information.

Private Claims

A producer and consumer may agree to use claim names that are private. These are subject to collision, so use them with caution.

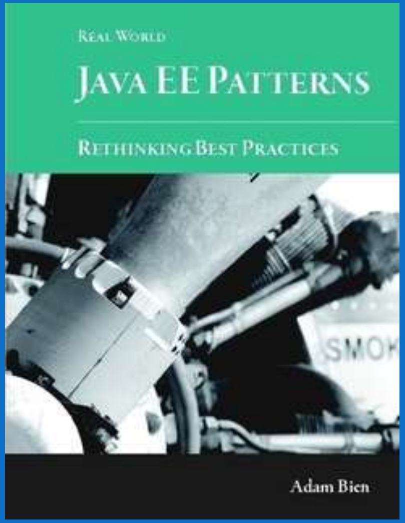
Source: https://scotch.io/tutorials/the-ins-and-outs-of-token-based-authentication

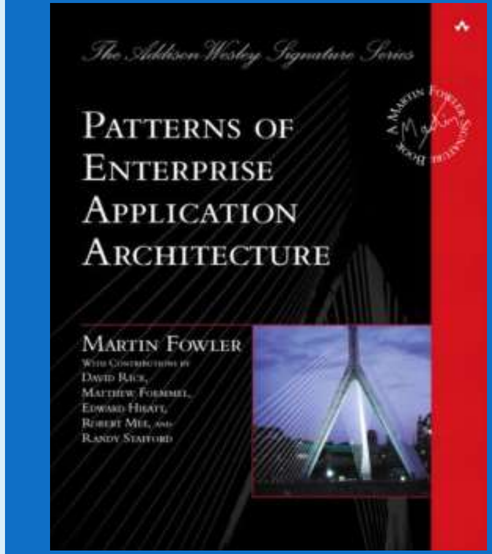
Design Patterns

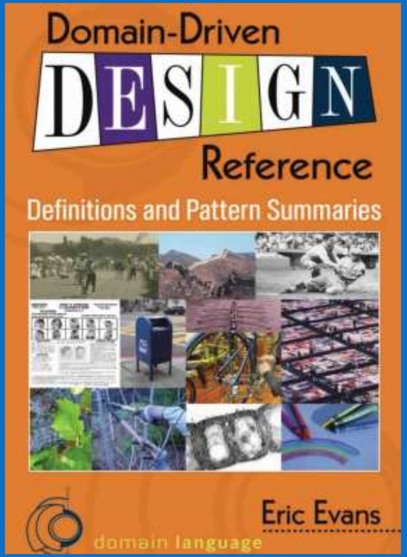
Are solutions to general problems that software developers faced during software development.

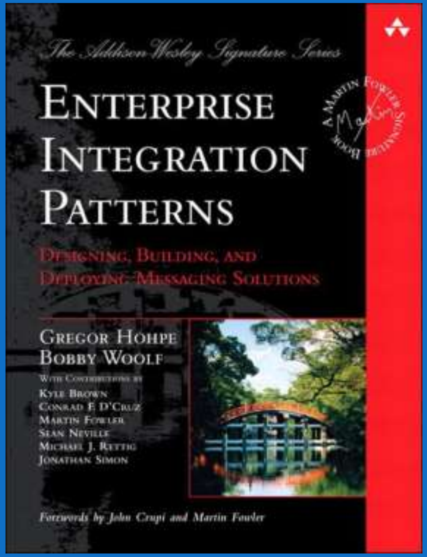
These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

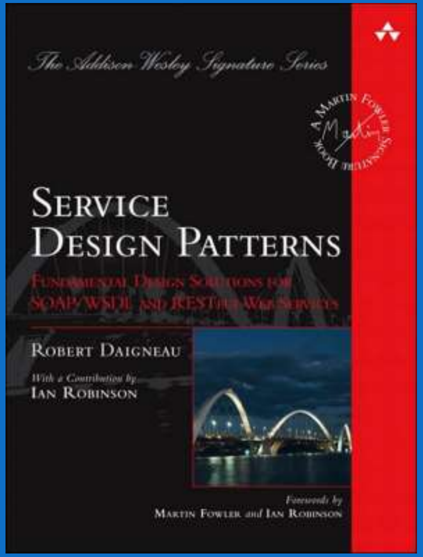
Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
Addison-Wesley Professional Computing Series




The Addison Wesley Signature Series
Patterns of Enterprise Application Architecture
Martin Fowler
With Contributions by David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford
A Martin Fowler Signature Book

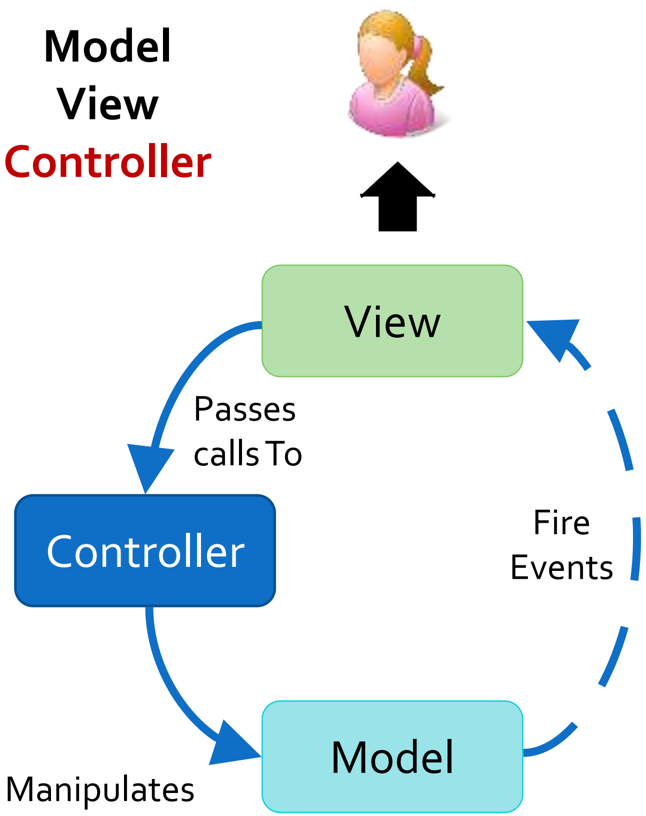



The Addison Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
Gregor Hohpe
Bobby Woolf
With Contributions by Kyle Brown, Conrad F. D'Cruz, Martin Fowler, Sean Neville, Michael J. Rettig, Jonathan Simon
Forewords by John Crupi and Martin Fowler
A Martin Fowler Signature Book


The Addison Wesley Signature Series
Service Design Patterns
Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services
Robert Daigneau
With a Contribution by Ian Robinson
Forewords by Martin Fowler and Ian Robinson
A Martin Fowler Signature Book

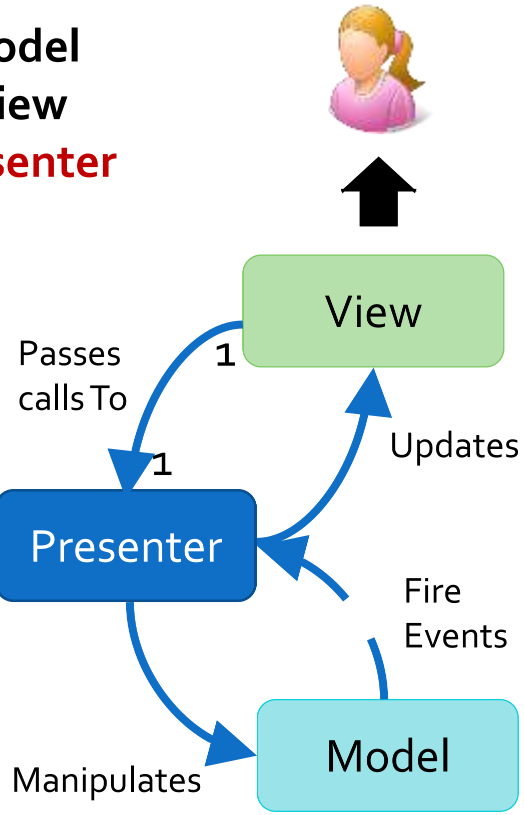
UI Design Patterns MVC / MVP / MVVM

Model View Controller



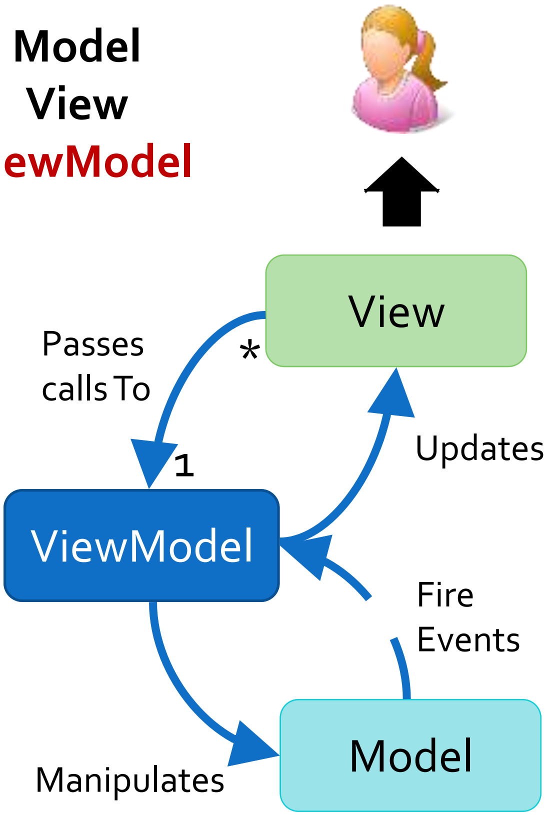
- The **Controller** is responsible to process incoming requests. It receives input from users via the View, then process the user's data with the help of Model and passing the results back to the View.
- Typically, it acts as the coordinator between the View and the Model.

Model View Presenter



- The **Presenter** is responsible for handling all UI events on behalf of the view. This receive input from users via the View, then process the user's data with the help of Model and passing the results back to the View.
- Unlike view and controller, **view and presenter are completely decoupled from each other's and communicate to each other's by an interface**. Also, presenter does not manage the incoming request traffic as controller.
- **Supports two-way data binding between View and ViewModel.**

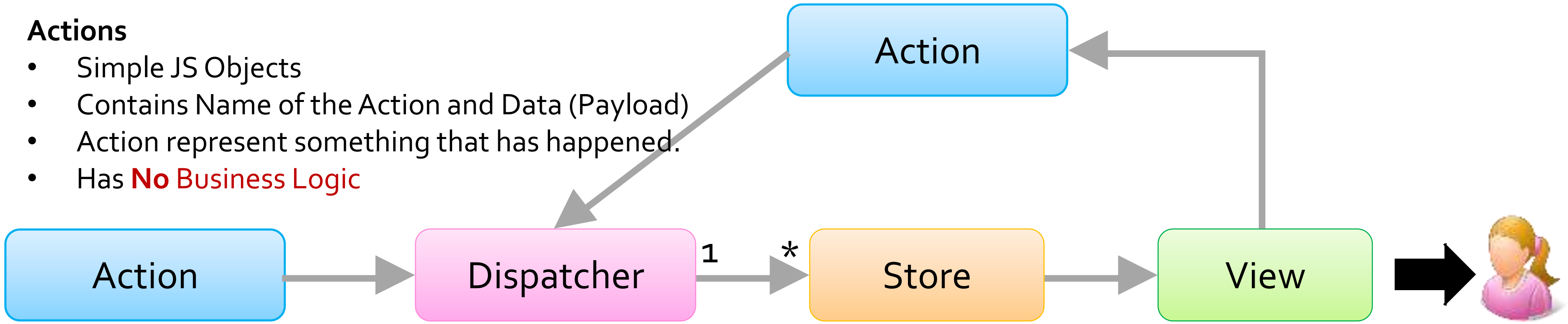
Model View ViewModel



- The View Model is responsible for exposing methods, commands, and other properties that helps to maintain the state of the view, manipulate the model as the result of actions on the view, and trigger events in the view itself.
- There is **many-to-one** relationship between View and ViewModel means many View can be mapped to one ViewModel.
- **Supports two-way data binding between View and ViewModel.**

Actions

- Simple JS Objects
- Contains Name of the Action and Data (Payload)
- Action represent something that has happened.
- Has **No** Business Logic



Every action is sent to all Stores via callbacks the stores register with the Dispatcher

Controller-Views

- Listens to Store changes
- Emit Actions to Dispatcher

Dispatcher

- Single Dispatcher per Application
- Manages the Data Flow View to Model
- Receives Actions and dispatch them to Stores

Stores

- Contains state for a Domain (Vs. Specific Component)
- **In Charge** of modifying the Data
- Inform the views when the Data is changed by emitting the Changed Event.

Flux Core Concepts

1. One way Data Flow
2. No Event Chaining
3. Entire App State is resolved in store before Views Update
4. Data Manipulation ONLY happen in one place (Store).

UI Design Patterns
Flux / Redux

Redux Core Concepts

1. One way Data Flow
2. No Dispatcher compared to Flux
3. Immutable Store

Available for React & Angular

Store

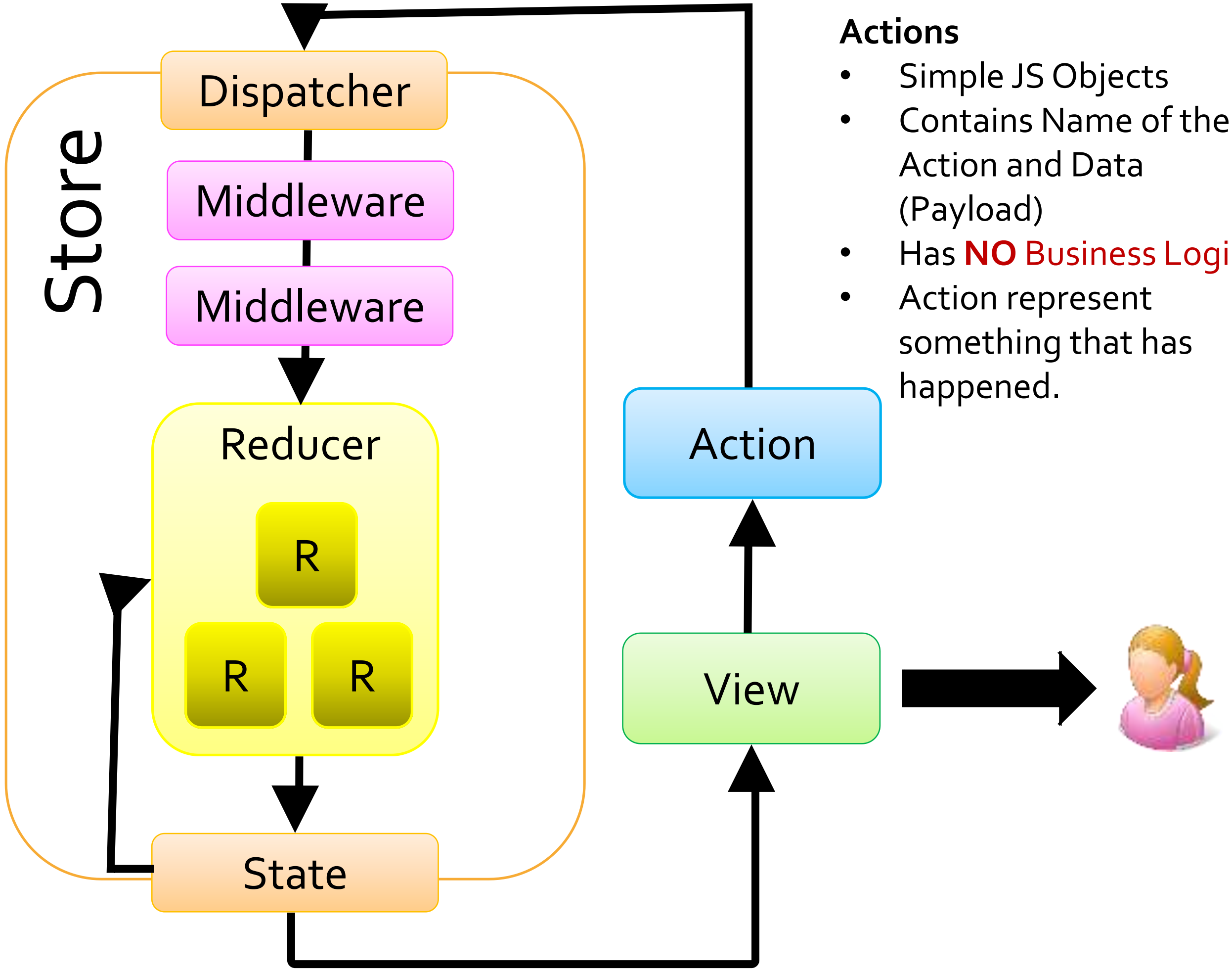
- Multiple View layers can **Subscribe**
- View layer to **Dispatch** actions
- Single Store for the Entire Application
- Data manipulation logic moves out of store to Reducers

Middleware

- Handles External calls
- Multiple Middleware's can be chained.

Reducer

- **Pure** JS Functions
- No External calls
- Can combine multiple reducers
- A function that specifies how the state changes in response to an Action.
- Reducer does **NOT modify** the state. It returns the **NEW State**.



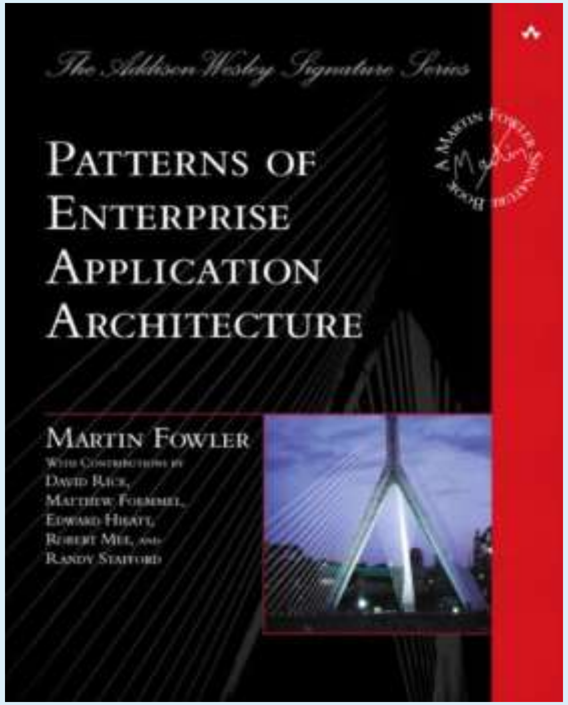
Actions

- Simple JS Objects
- Contains Name of the Action and Data (Payload)
- Has **NO** Business Logic
- Action represent something that has happened.

UI Design Patterns Redux

Patterns of Enterprise Application Architecture

	Pattern	Description	Page
1	Domain Model	An Object Model of the domain that incorporates both behavior and data	116
2	Service Layer	Defines an application’s boundary with a layer of services that establishes a set of available operations and coordinates the applications response in each operation	133
3	Repository	Mediates between the Domain and data mapping layers using a collection like interface for accessing Domain Objects.	322
4	Data Transfer Object	An Object that carries data between processes in order to reduce the number of method calls.	401
5	Value Object	A Small Object, like Money, or a date range, whose equality isn’t based on identity.	486
6	Single Table Inheritance	Represents an inheritance hierarchy of classes as a Single table that has columns for all the fields of the various classes.	278
7	Class Table Inheritance	Represents an inheritance hierarchy of classes with one table for each class.	285
8	Concrete Table Inheritance	Represents an inheritance hierarchy of classes with one table per concrete class in the hierarchy	293
9	Remote Facade	Provides a coarse-grained façade on fine grained objects to improve efficiency over a network.	388
10	Optimistic Offline Lock	Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back transactions.	416


The Addison Wesley Signature Series

PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE

MARTIN FOWLER

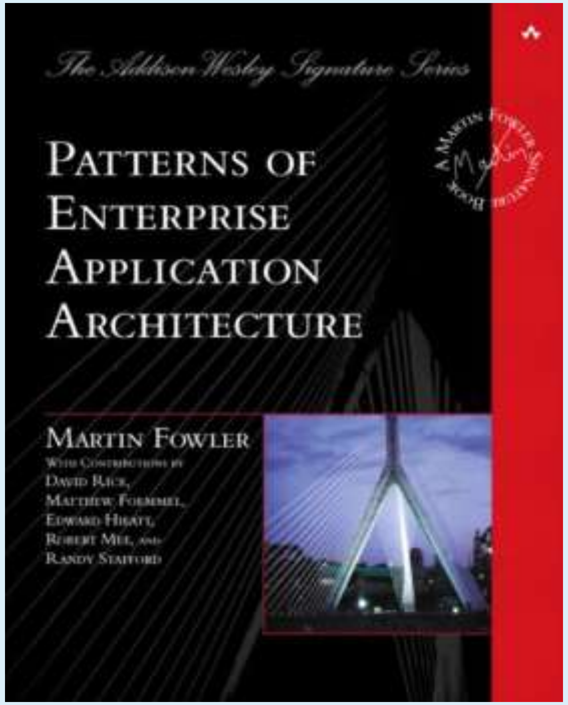
WITH CONTRIBUTIONS BY DAVID RICE, MATTHEW FOEMMEL, EDWARD HIEATT, ROBERT MEE, AND RANDY STAFFORD

A MARTIN FOWLER SIGNATURE BOOK

Page Number from P of EAA

Patterns of Enterprise Application Architecture

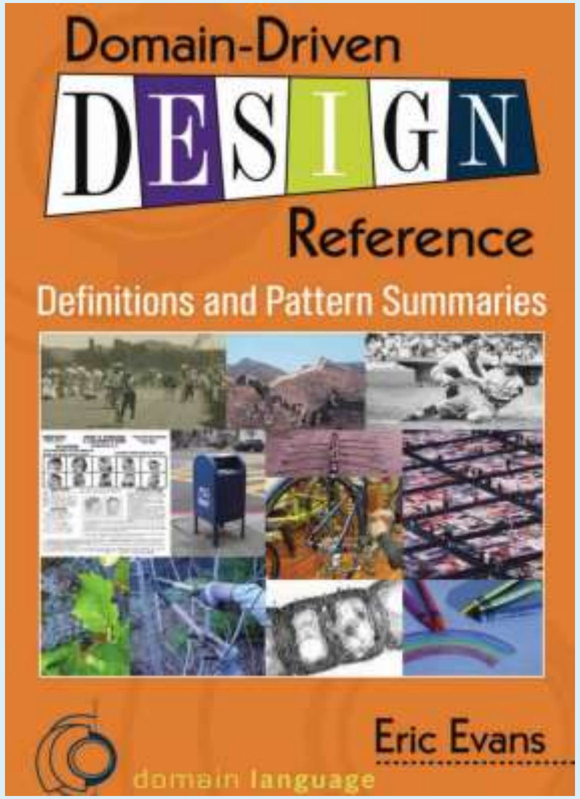
	Pattern	Description	Page
11	Pessimistic Offline Lock	Prevents conflicts between concurrent business transactions by allowing only one business transaction at a time to access data.	426
12	Client Session State	Stores session state on the client.	456
13	Server Session State	Keeps the session state on a server system in a serialized form	458
14	Separate Interface	Defines an interface in a separate package from its implementation	476
15	Service Stubs	Removes dependence upon problematic services during testing. (External Integration)	504
16	Record Set	An in-memory representation of tabular data.	509
17	Serialized LOB	Saves a Graph of Objects by serializing them into a Single Large Object (LOB), which it stores in a database field.	272


The Addison-Wesley Signature Series
Patterns of Enterprise Application Architecture
A Martin Fowler Signature Book
Martin Fowler
With Contributions by
David Rice,
Matthew Foemmel,
Edward Hieatt,
Robert Mee, and
Randy Stafford

Page Number from P of EAA

Design Patterns – Domain Driven Design : Strategic

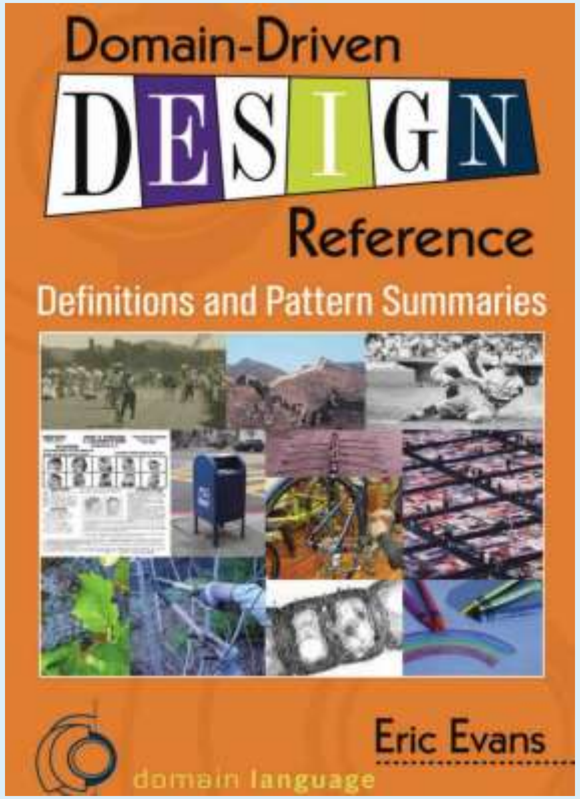
	Pattern	Description	Page
1	Bounded Context They are NOT Modules	A Bounded Context delimits the applicability of a particular model so that the team members have a clear and shared understanding of what has to be consistent and how it relates to other Contexts. Contexts can be created from (but not limited to) the following: • how teams are organized • the structure and layout of the code base • usage within a specific part of the domain	335
2	Context Map	Context mapping is a design process where the contact points and translations between bounded contexts are explicitly mapped out. Focus on mapping the existing landscape, and deal with the actual transformations later. 1. Shared Kernel 2. Customer / Supplier 3. Conformist 4. Anti Corruption Layer 5. Separate Ways	
3	Specification Pattern	Use the specification pattern when there is a need to model rules, validation and selection criteria. The specification implementations test whether an object satisfies all the rules of the specification.	
4	Strategy Pattern	The strategy pattern, also known as the Policy Pattern is used to make algorithms interchangeable. In this pattern, the varying 'part' is factored out.	
5	Composite Pattern	This is a direct application of the GoF pattern within the domain being modeled. The important point to remember is that the client code should only deal with the abstract type representing the composite element.	



Page Number from Domain Driven Design – Published in 2015

Design Patterns – Domain Driven Design

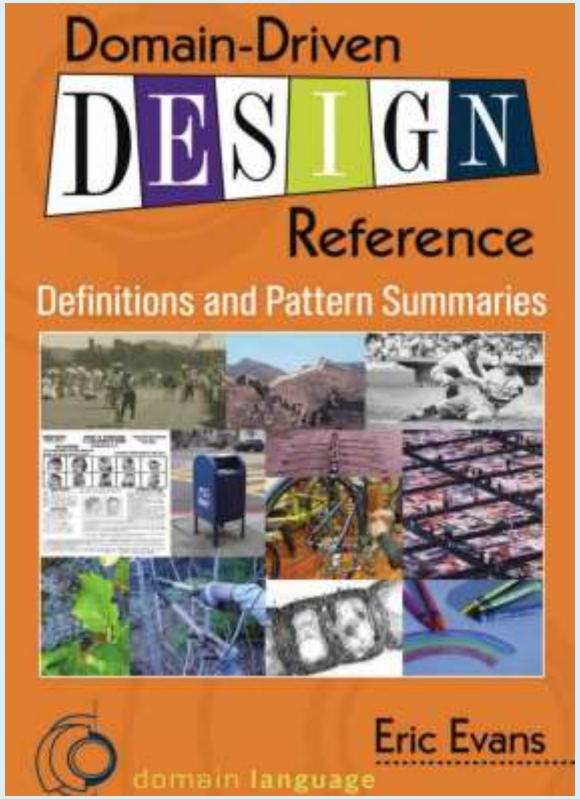
	Pattern	Description	Page
6	Entity	An object defined Primarily by its identity is called an Entity	91
-	Value Object (Already referred in P of EAA)	Many Objects have no conceptual Identity. These objects describe the characteristic of a thing.	97
7	Aggregate	Aggregate is a cluster of domain objects that can be treated as a Single Unit. Example Order and Order Item.	125
	Aggregate Root	An Aggregate will have one of its component object be the Aggregate Root.	127
-	Repositories (Already referred in P of EAA)	A Repository represents all objects of a certain type as a conceptual set. It acts like a collection, except with more elaborate querying capabilities. Objects of appropriate type are added and removed, and the machinery behind the Repository inserts them or deletes them from the database. This definition gathers a cohesive set of responsibilities for providing access to the roots of Aggregates from early life cycle through the end.	147
8	Factory / Builder Pattern	When creation of an Object, or an entire Aggregate, becomes complicated or reveals too much of the internal structure, Factories provides encapsulation.	136



Page Number from Domain Driven Design – Published in 2015

Design Patterns – Domain Driven Design

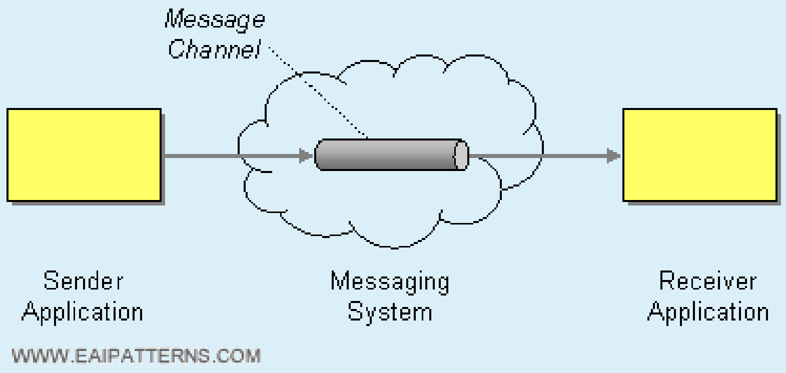
	Pattern	Description	Page
9	Factory / Builder Pattern	When creation of an Object, or an entire Aggregate, becomes complicated or reveals too much of the internal structure, Factories provides encapsulation.	136
10	Domain Service	A Service tends to be named of an Activity rather than an Entity. 1. The Operation relates to a domain concept that is not a natural part of an Entity. 2. The interface is defined in terms of other elements of the Domain Model 3. The operation is stateless	104
11	Anti – Corruption Layer (External Integration)	Creating an isolating layer to provide clients with functionality in terms of their own Domain Model. The layer talks to the other system through its existing interface, requiring little or no modification to the other system. Internally the Layer translates in both directions as necessary between the two models.	365
12	Domain Events	A Domain Event is a full-fledged part of the Domain Model, a representation of of something that happened in the Domain. Explicit events that the domain experts wants to track and notified of or which are associated with the state changes in other Domain Models.	

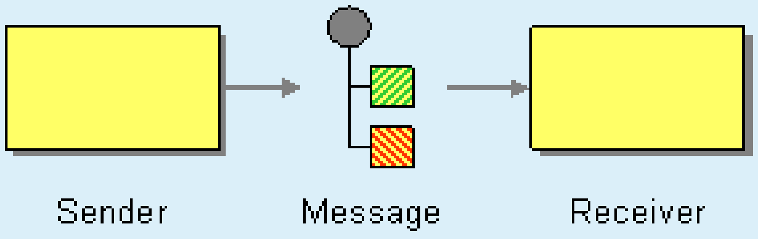


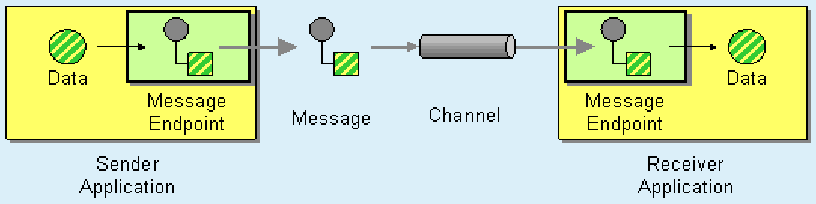
Page Number from Domain Driven Design – Published in 2015

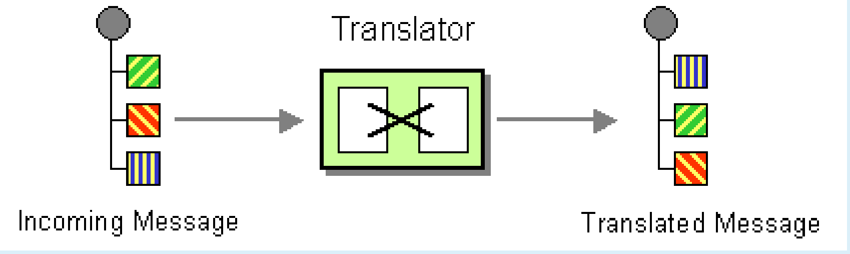
Enterprise Integration Patterns – Messaging System

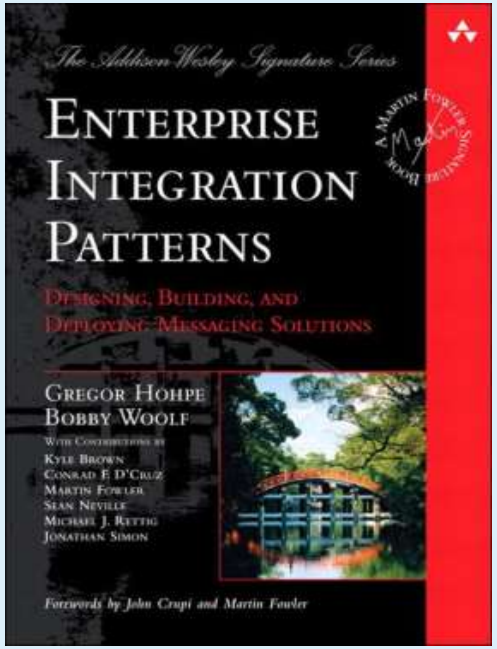
	Pattern	Description	Page
1	Message Channel	The Message Channel is an internal implementation detail of the Endpoint interface and all interactions with the Message Channel are via the Endpoint Interfaces.	60
2	Message	To support various message exchange patterns like one way Event Message and Request Reply messages Camel uses an Exchange interface which has a pattern property which can be set to In-Only for an Event Message which has a single inbound Message, or In-Out for a Request Reply where there is an inbound and outbound message.	53
3	Message Endpoint	Supports 4 endpoints • Web Services: REST & SOAP • Database : JDBC • File : Reads File	95
4	Message Translator	We have 3 built-in Processors 1. SQL Processor (Take DTO and pass value to query) 2. SOAP Processor (Generate SOAP Message) 3. REST Processor(Take DTO convert into respected format	85





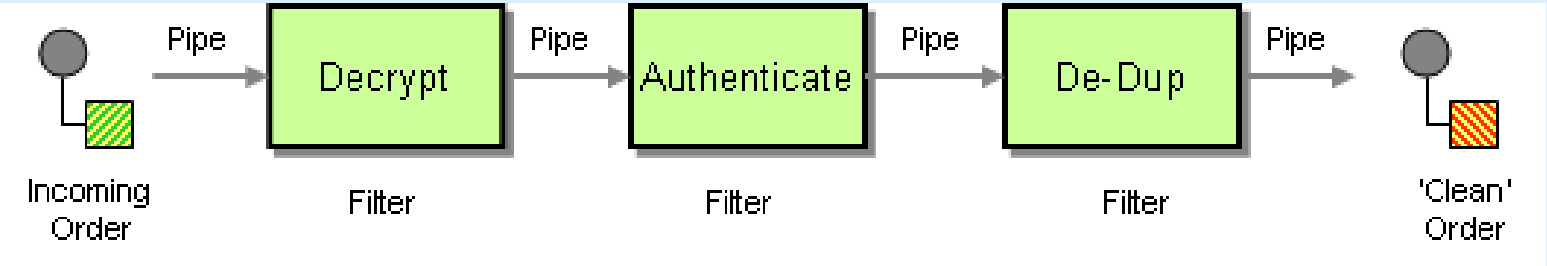
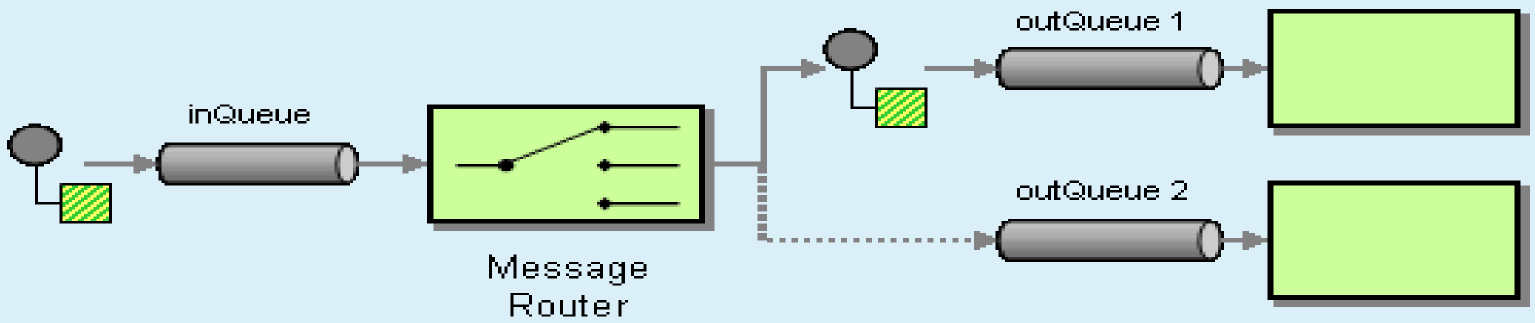


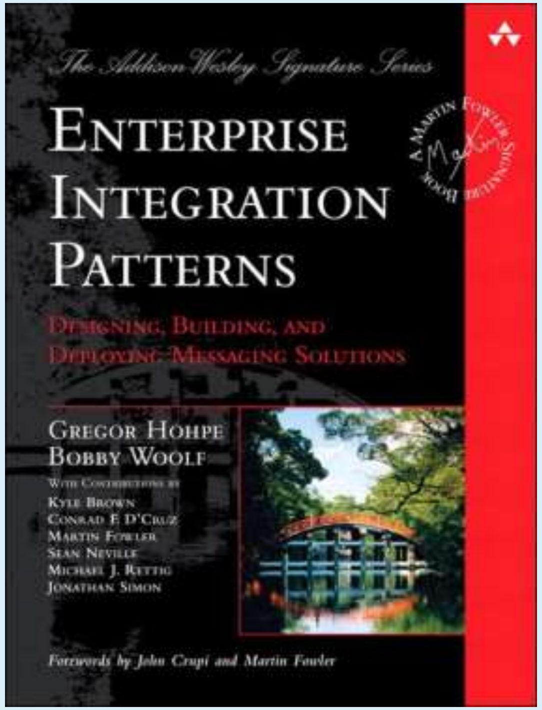



The Addison-Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
Gregor Hohpe
Bobby Woolf
With Contributions by
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon
Forewords by John Crupi and Martin Fowler
A Martin Fowler Signature Book

Page Number from Enterprise Integration Patterns

Enterprise Integration Patterns – Messaging Systems

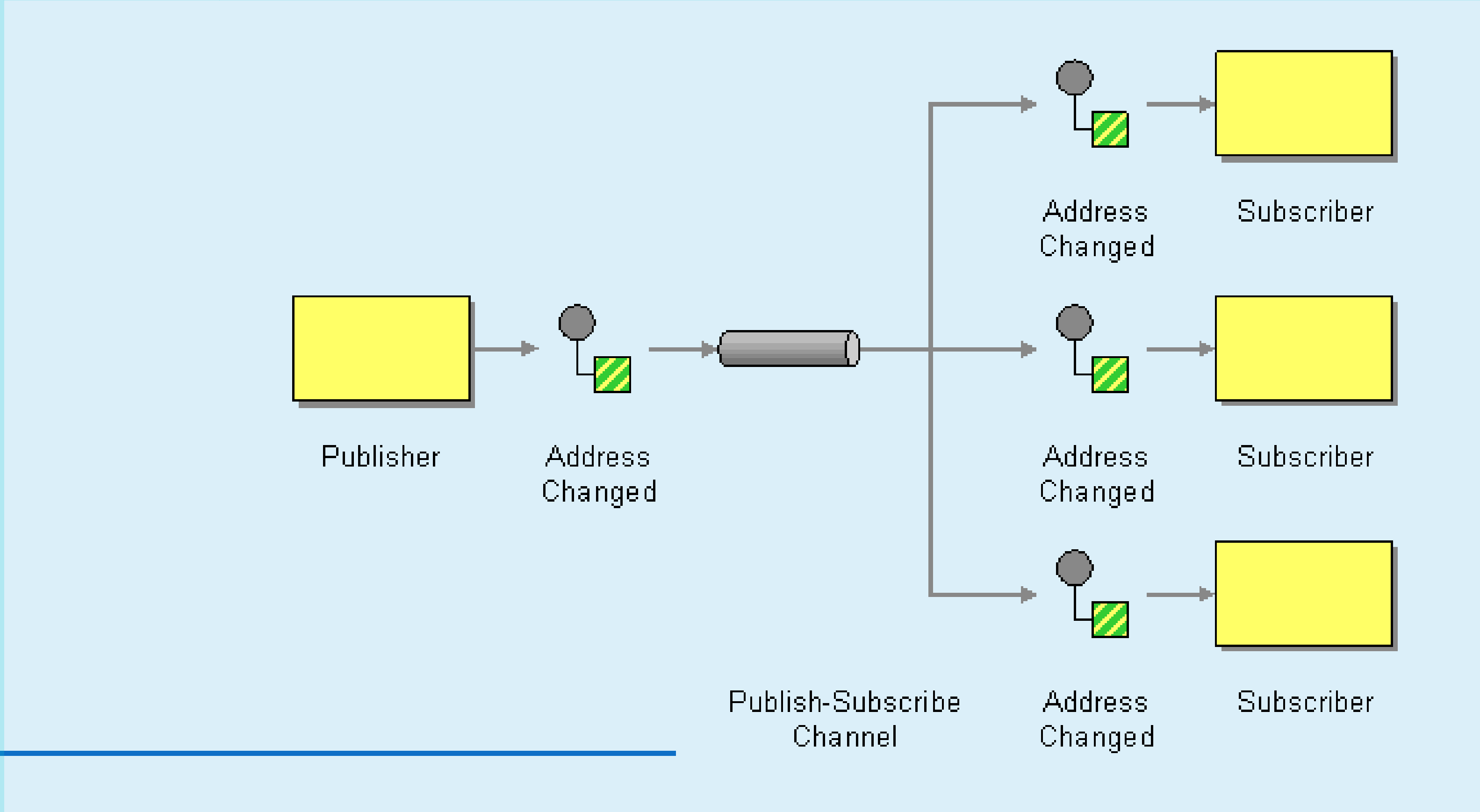
	Pattern	Description	Page
5	Pipes and Filters	Supports the Pipes and Filters from the EIP patterns in various ways. You can split your processing across multiple independent Endpoint instances which can then be chained together. 	70
6	Message Router	The Message Router from the EIP patterns allows you to consume from an input destination, evaluate some predicate then choose the right output destination. 	78

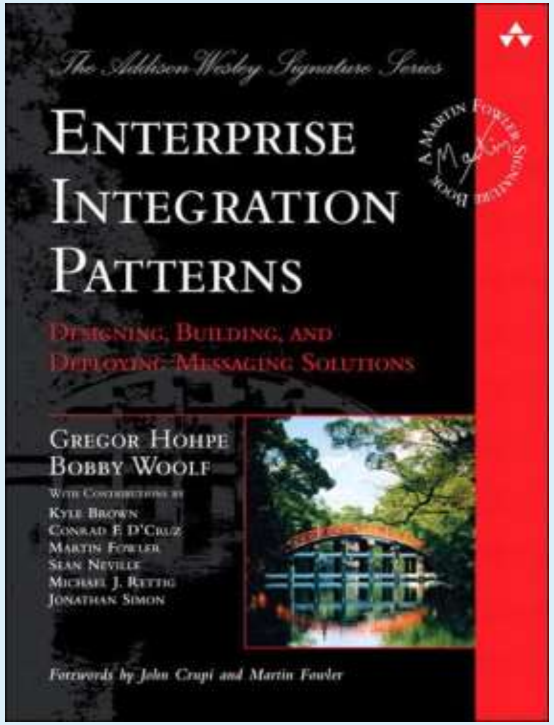


The Addison-Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
A Martin Fowler Signature Book
Gregor Hohpe
Bobby Woolf
With Contributions by
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon
Forewords by John Crupi and Martin Fowler

Page Number from Enterprise Integration Patterns

Enterprise Integration Patterns – Messaging Channels

	Pattern	Description	Page
7	Publish Subscribe System	Supports the Publish Subscribe Channel from the EIP patterns using for example the following components: • JMS for working with JMS Topics for high performance, clustering and load balancing • XMPP when using rooms for group communication • SEDA for working with SEDA in the same Camel Context which can work in pub-sub, but allowing multiple consumers. • VM as SEDA but for intra-JVM. 	106

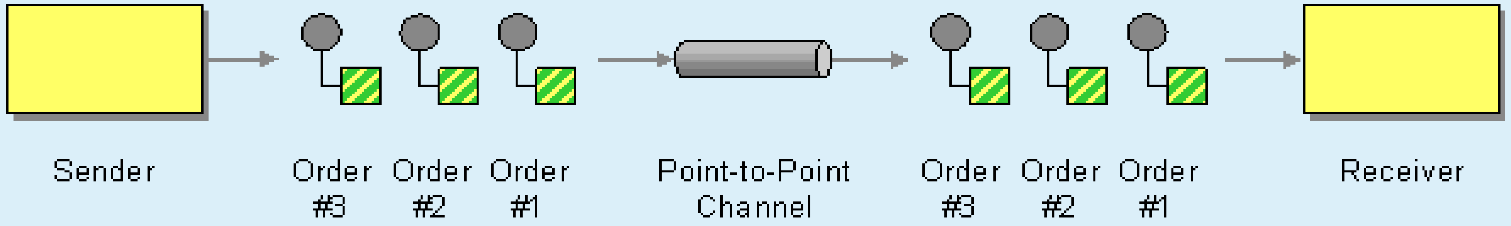


The Addison Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
A Martin Fowler Signature Book
Gregor Hohpe
Bobby Woolf
With Contributions by
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon
Forewords by John Crupi and Martin Fowler

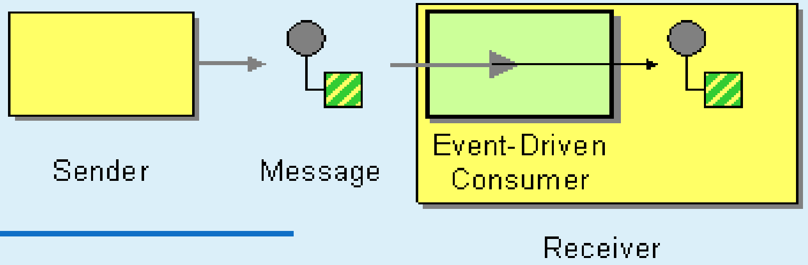
Page Number from Enterprise Integration Patterns

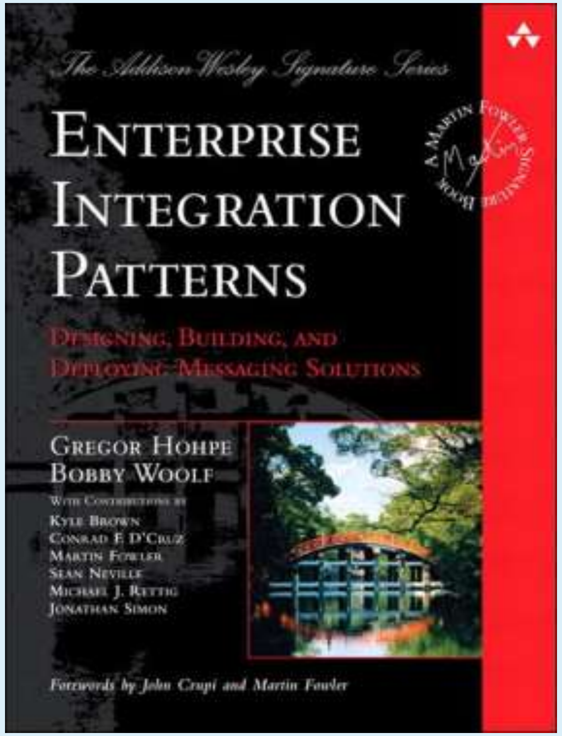
Enterprise Integration Patterns – Messaging Channels

	Pattern	Description	Page
8	Point to Point Channel	Supports the Point to Point Channel from the EIP patterns using the following components 1. SEDA for in-VM seda based messaging 2. JMS for working with JMS Queues for high performance, clustering and load balancing 3. JPA for using a database as a simple message queue 4. XMPP for point-to-point communication over XMPP (Jabber) 5. and others	103



	Pattern	Description	Page
9	Event Driven Consumer	Supports the Event Driven Consumer from the EIP patterns. The default consumer model is event based (i.e. asynchronous) as this means that the Camel container can then manage pooling, threading and concurrency for you in a declarative manner. The Event Driven Consumer is implemented by consumers implementing the Processor interface which is invoked by the Message Endpoint when a Message is available for processing.	498





The Addison Wesley Signature Series

Enterprise Integration Patterns

Designing, Building, and Deploying Messaging Solutions

A Martin Fowler Signature Book

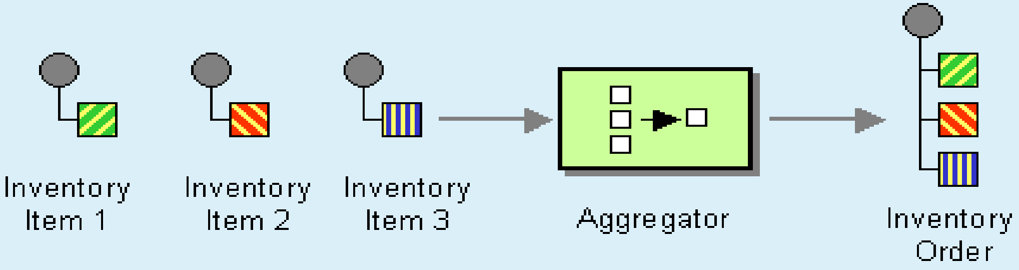
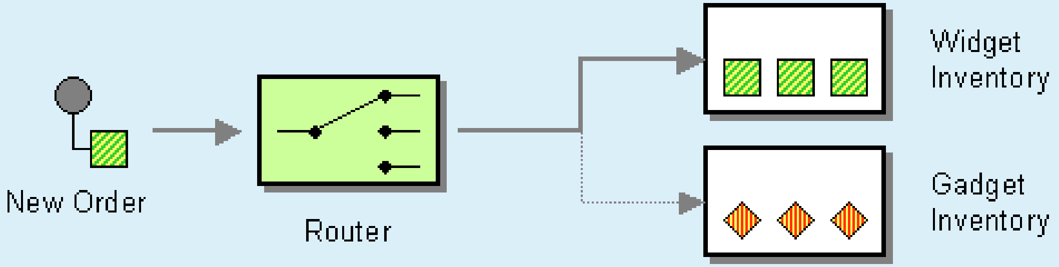
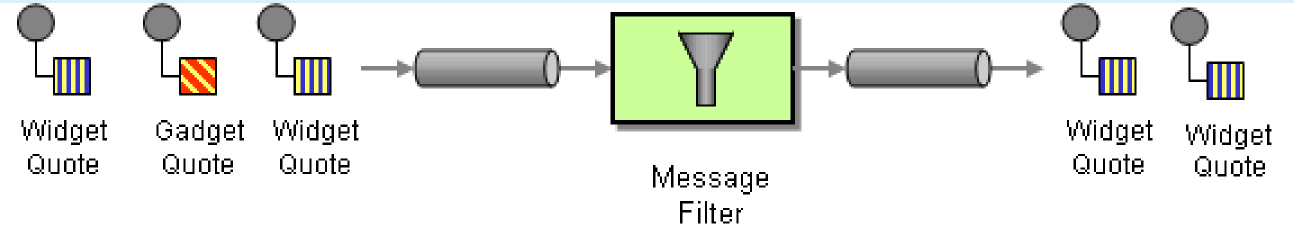
Gregor Hohpe
Bobby Woolf

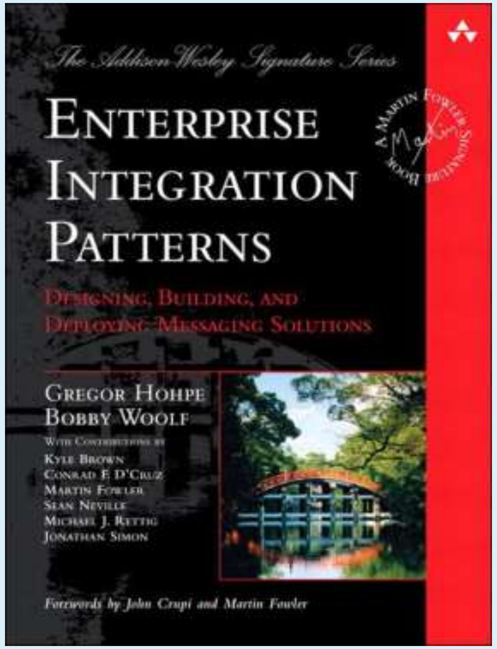
With Contributions by
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon

Forewords by John Crupi and Martin Fowler

Page Number from Enterprise Integration Patterns

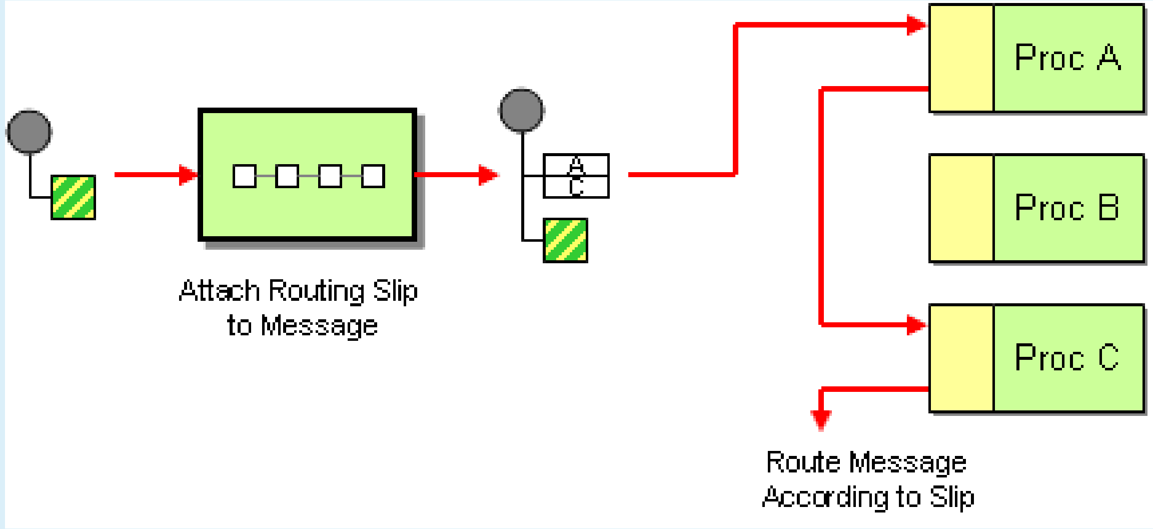
Enterprise Integration Patterns – Message Routing

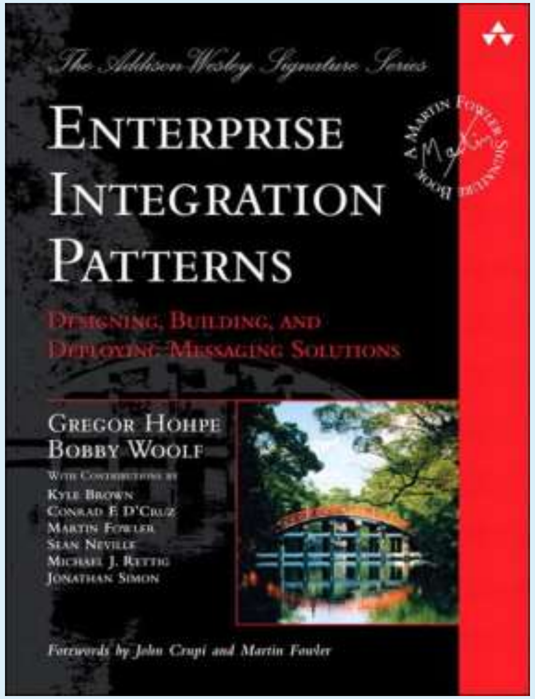
	Pattern	Description	Page
10	Aggregator	The Aggregator from the EIP patterns allows you to combine a number of messages together into a single message.	268
		 Inventory Item 1 Inventory Item 2 Inventory Item 3 Aggregator Inventory Order	
11	Content Based Router	The Content Based Router from the EIP patterns allows you to route messages to the correct destination based on the contents of the message exchanges.	230
			
12	Message Filter	The Message Filter from the EIP patterns allows you to filter messages.	237
		 Widget Quote Gadget Quote Widget Quote Message Filter Widget Quote Widget Quote	


The Addison-Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
Gregor Hohpe
Bobby Woolf
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon
Forewords by John Crupi and Martin Fowler

Page Number from Enterprise Integration Patterns

Enterprise Integration Patterns – Message Routing

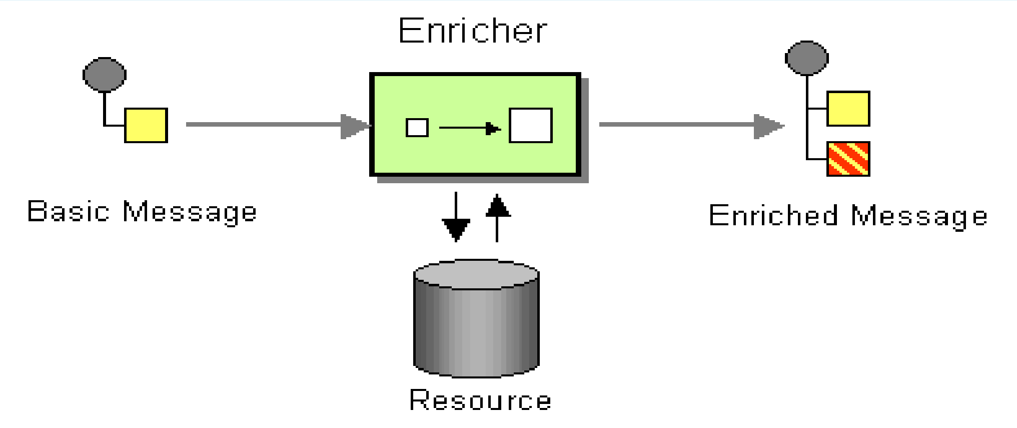
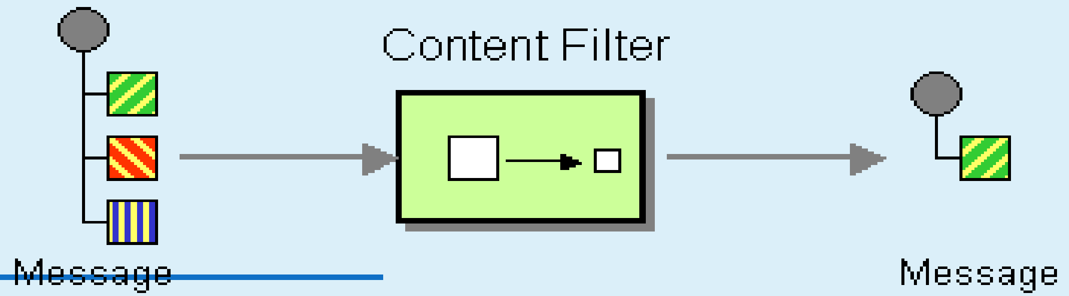
	Pattern	Description	Page
13	Routing Slip	The Routing Slip from the EIP patterns allows you to route a message consecutively through a series of processing steps where the sequence of steps is not known at design time and can vary for each message.  Attach Routing Slip to Message A C Proc A Proc B Proc C Route Message According to Slip	301
14	Load Balancer	The Load Balancer Pattern allows you to delegate to one of a number of endpoints using a variety of different load balancing policies.	
15	Multicast	The Multicast allows to route the same message to a number of endpoints and process them in a different way. The main difference between the Multicast and Splitter is that Splitter will split the message into several pieces but the Multicast will not modify the request message.	

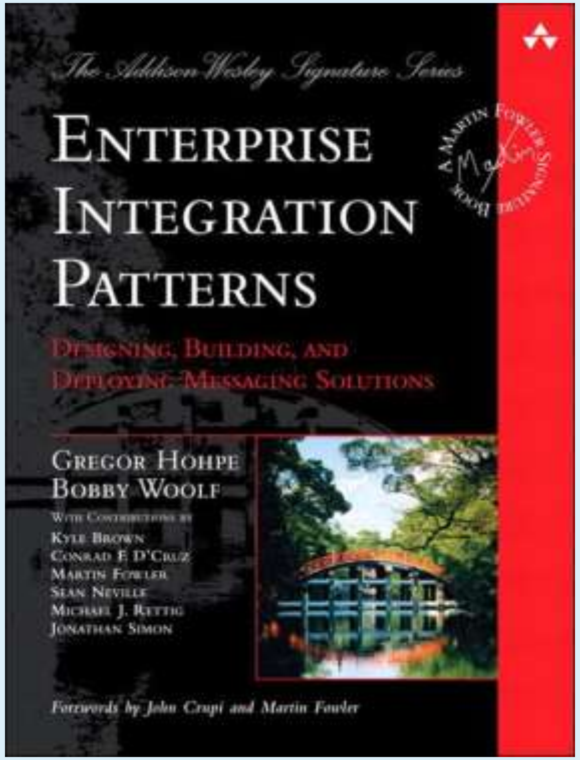


The Addison-Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
Gregor Hohpe
Bobby Woolf
With Contributions by
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon
Forewords by John Crupi and Martin Fowler
A Martin Fowler Signature Book

Page Number from Enterprise Integration Patterns

Enterprise Integration Patterns – Message Transformation

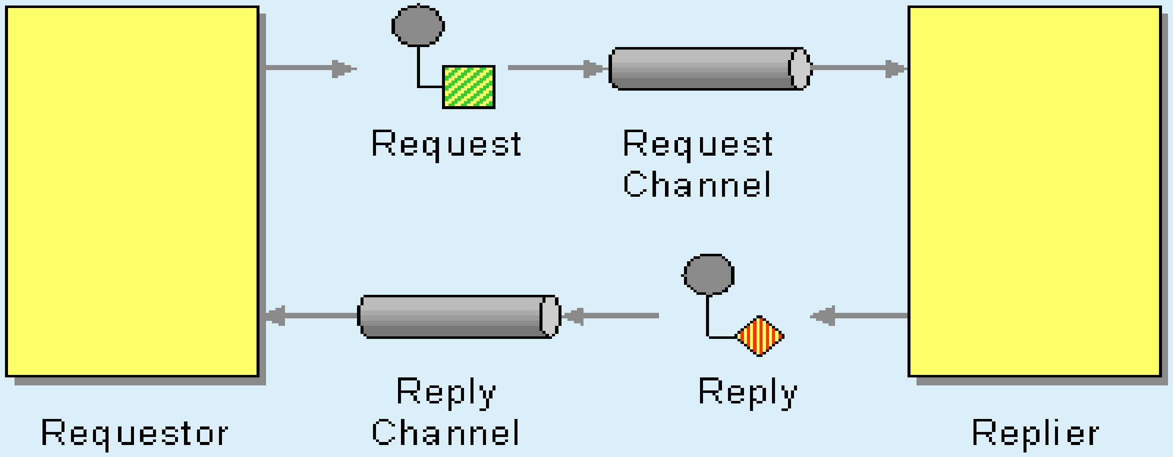
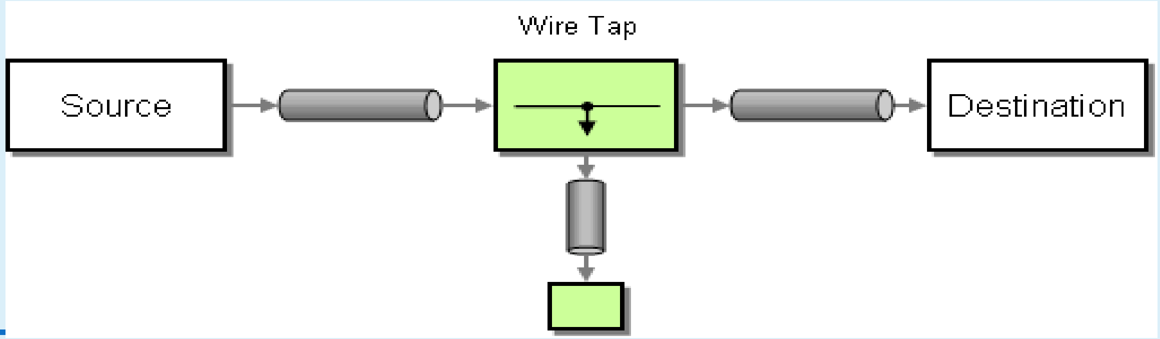
	Pattern	Description	Page
16	Content Enricher	Supports the Content Enricher from the EIP patterns using a Message Translator, an arbitrary Processor in the routing logic, or using the enrich DSL element to enrich the message.	336
		 	
17	Content Filter	Supports the Content Filter from the EIP patterns using one of the following mechanisms in the routing logic to transform content from the inbound message. • Message Translator • invoking a Java bean • Processor object	
		 	342

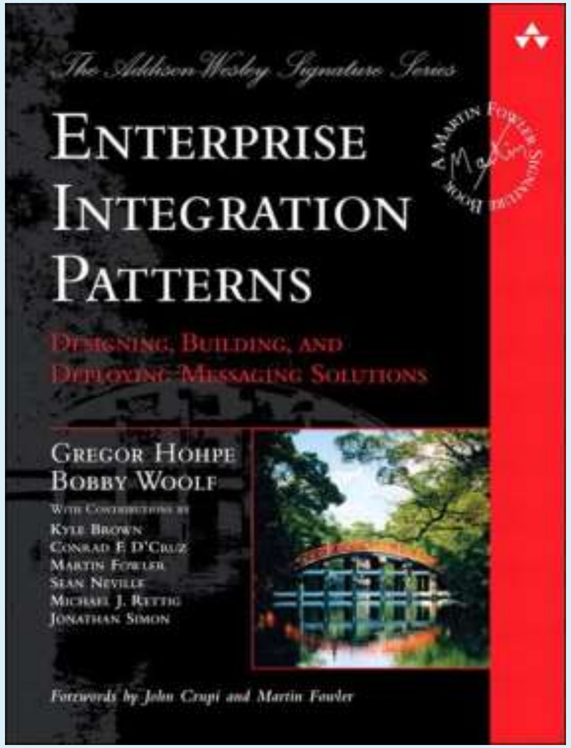


The Addison-Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
A Martin Fowler Signature Book
Gregor Hohpe
Bobby Woolf
With Contributions by
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon
Forewords by John Crupi and Martin Fowler

Page Number from Enterprise Integration Patterns

Enterprise Integration Patterns – Others

	Pattern	Description	Page
18	Request Reply	Supports the Request Reply from the EIP patterns by supporting the Exchange Pattern on a Message which can be set to In Out to indicate a request/reply. Camel Components then implement this pattern using the underlying transport or protocols.  Request Request Channel Reply Channel Reply Requestor Replier	154
19	Wire tap	Wire Tap (from the EIP patterns) allows you to route messages to a separate location while they are being forwarded to the ultimate destination.  	547

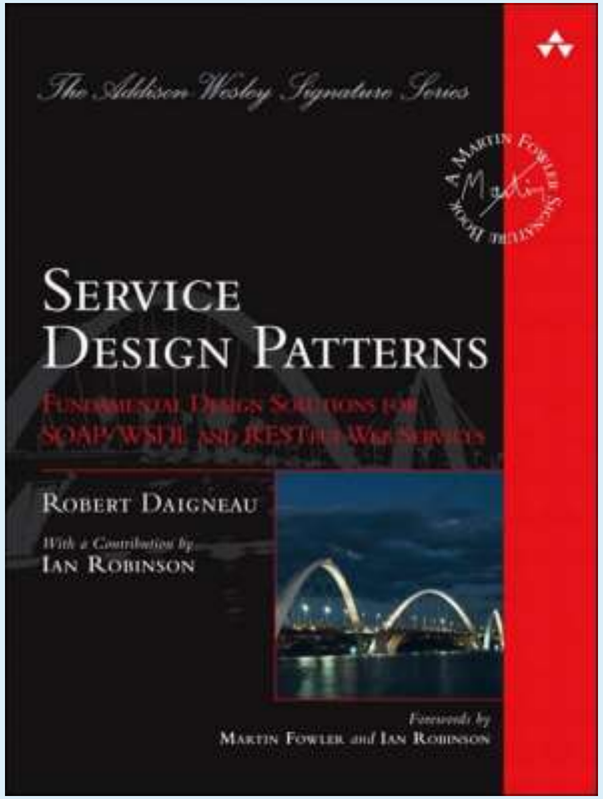


The Addison-Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
Gregor Hohpe
Bobby Woolf
A Martin Fowler Signature Book

Page Number from Enterprise Integration Patterns

Design Patterns – Service Design Pattern

	Patterns	Description	Page
1	Request Mapper	How can a service process data from requests that are structurally different yet semantically equivalent?	109
2	Response Mapper	How can the logic required to construct a response be re-used by multiple services?	122
3	Data Source Adapter	How can web service provide access to internal resources like database tables, domain objects, or files with a minimum amount of custom code?	137
4	Asynchronous Response Handler	How can a client avoid blocking when sending a request?	184
5	Service Connector	How can clients avoid duplicating the code required to use a specific service and also be insulated from the intricacies of communication logic?	168


The Addison Wesley Signature Series
A Martin Fowler Signature Book
Service Design Patterns
Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services
Robert Daigneau
With a Contribution by
Ian Robinson
Forewords by
Martin Fowler and Ian Robinson

Page Number from service Design Patterns

References

1. Hexagonal Architecture :
    - https://skillsmatter.com/skillscasts/5744-decoupling-from-asp-net-hexagonal-architectures-in-net
2. Onion Architecture :

Thank You

email : araf.karsh@ozazo.com | Phone : +91.999.545.8627
Social : arafkarsh | Skype, Facebook, LinkedIn, Twitter, G+