Introduction

1. Questions

2. Animated Intro

3. Terminology

Qi4j Tutorial

4. Abstract Composites

5. Example

6. DDD and Qi4j
Some Questions
Where is a class in real life?
All we have are objects
Is it always the same thing?
It depends
It depends
- on the context!
Different views on the same object.
Different data, different functions
How to model this?
How to model this?
- With classes?
Roles, Responsibilities, Relationships
= Patterns
(in every area of life)
Modeling successful (business) patterns ?
Animated Intro
Object Oriented Programming
Domain model
But where is it?

Value Objects
XML descriptors
SQL queries
Data Transfer Objects
Session Facades

It’s gotta be in here somewhere...
Let's abstract it!

XML Schemas

Code generation

Domain Specific Languages

Pointcuts

Weaving

Containers

Byte code manipulation

Aspects

Model Driven Architecture

But where's the domain model?
Typeless

Specifications

Code conventions

Testing

Contracts?

Reuse?

Ruby

Ruby on Rails

GRails

JavaScript

Python

Refactoring?

I want my domain model goddamnit...
What do we need?

Constraints, Assertions, and Side Effects

Refactoring

IDE support  Strong typing  Server  Client  Security

Code control  Reuse

Mixins  Contexts

Multidomain  Persistence

Ubiquitous Language  Design Patterns

Domain Driven Design  OOP
All the pieces are there

Scripting  Dependency Injection

Aspect Oriented Programming  Domain Driven Design
What if we put it all together?
Composite Oriented Programming
Terminology
Terminology

Composite

Constraint

Assertion

SideEffect

Mixin

Mixin

Mixin
Agenda

:: Composite interfaces
:: Mixins
:: Assemblies
:: Composite factories
:: Modifiers
Composite interfaces

- Composites in Qi4j are defined using Java interfaces
- Extends org.qi4j.api.composite.Composite
- Uses annotations to declare fragment implementations that can be used
Composite factories

- Composites are instantiated by using CompositeBuilderFactory

- Acquired either through SingletonAssembly or by using dependency injection annotation
  
  - @Structure CompositeBuilderFactory cbf;

- Each factory is specific for the module it was acquired from
Modifiers

- Sometimes you want to modify the behaviour of methods
- Usually cross-cutting concerns
- Or just code that you don’t want in the mixin code
  - E.g. validation rules that may change
Modifiers

:: COP describes three types of Modifiers
  :: Constraints
  :: Concerns
  :: SideEffects
Constraints

- Constraints implement parameter validation
  - E.g. value range checks, string expressions
- Implemented by implementing interface Constraint<annotation,type>
- Applied by adding annotation to method parameter
Concerns

- Concerns can be used to add arbitrary behaviour before and after method execution
  - Same as “around advice” in AOP
- Concerns can be stacked
- Concerns can change method arguments, return values, throw exception, abort the call chain
SideEffects

- SideEffects are executed after the original method call has completed
  - The return value or exception cannot be changed!
  - Method arguments cannot be changed
- Used to modify external state or invoke external services
  - E.g. “send mail after method has completed”
Agenda

- Multiple mixins
- Generic concerns
- @AppliesTo
- Generic mixins
- Abstract composites
- CompositeBuilder
- Private mixins
Multiple mixins

- If your Composite extends many interface you may want one mixin per interface
- If your interfaces extends other interfaces you may want to implement only the superinterface
  - E.g. “Person extends FirstName, LastName” but PersonMixin implements both FirstName and LastName interfaces
- Mixins may implement a subset of an interface
Generic concerns

- Many Concerns can be applied to any method
- Generic concerns implement InvocationHandler from Dynamic Proxy API in the JDK
- Only implement a generic “invoke” method
- Typically used for non-domain Concerns
  - E.g. transactions, security, logging, etc.
@AppliesTo

- For generic Modifiers that should only trigger on methods with specific annotations or fulfilling some expression

- Add @AppliesTo annotation to Modifier class which points to either triggering annotation, or to AppliesToFilter implementation

- AppliesToFilters can do any checks it wants
Generic mixins

- Sometimes methods can be handled by generic mixin implementations
  - E.g. interface extends Remote and method throws IOException -> delegate to RMI service
- Generic mixins implement InvocationHandler
- One Mixin instance per Composite instance
- Useful for coding by convention patterns
Abstract composites

- If you are repeating some Composite declarations often, put them in an Abstract Composite

- Abstract Composites are interfaces with declarations that are then extended by Composite interfaces

- This allows Composites to easily reuse the declarations

- The Abstract Composite provides a central point where you can easily add more declarations
@This

:: If a Fragment needs a reference to the Composite instance it is a part of, it can use the @This dependency injection annotation

:: The Composite should implement the type of the injected field
Composite

MixinA

->

MixinB
Private mixins

Sometimes you may want to have mixins which are only available internally through @This injections

- State with property methods
- Helper mixins
- Simply don’t add the interfaces to the Composite interface
Sometimes you may want to use the Builder pattern instead of Factory for Composite instantiation.

From a CompositeBuilderFactory you can get a CompositeBuilder.

It allows you to set things that should apply to all instances created from that CompositeBuilder.

CompositeBuilder implements Iterable.
Agenda

- What is an Abstract Composite?
- Abstract Composite structure
- Abstract Composite examples
  - Default Constraints
  - Generic Mixins
- Caching
- Aggregation
- Validation
What is an Abstract Composite?

- Qi4j contains reusable Fragments
  - Constraints
  - Assertions
  - SideEffects
  - Mixins
- Can be grouped together in extendable Abstract Composite interfaces for easy reuse
  - Examples: Caching, Default constraints, Aggregation
Abstract Composite structure

- Abstract Composites are regular Composite interfaces which declare a number of Fragments
- Useless on their own
- Extend it to use
- A Composite can extend many Abstract Composites at the same time
- Check what annotations are used by the Fragments in the Abstract Composite
Composite extends MyAbstractComposite

Constraint

Assertion

Reusable Assertion

SideEffect

Mixin

Mixin

ReusableMixin
Some Abstract Composites provide library functionality.

It can be useful for a project to have a “StandardAbstractComposite” base interface that all domain Composites extend from, and which extends other Abstract Composites.

Makes it easy to add/remove standard functionality later on.
Abstract Composites in Qi4j

- A number of Abstract Composites are available in Qi4j
- Implements a number of common use cases
Default Constraints

- Defines and declares a number of commonly useful annotations and Constraints
- Extend DefaultConstraintsAbstractComposite to use
Generic Mixins

- Qi4j contains a number of Generic Mixins
- Provides implementations based on conventions
  - PropertiesMixin - get/set/add/remove
  - FinderMixin - findByNameOrEmail
- Provides implementations for scripting
  - JavaScriptMixin
- Provides implementations for infrastructure delegation
  - RMIMixin
- NoopMixin - can handle anything!
Invocation Caching

- If a method call is expensive, use result caching to avoid unnecessary computation.
- Can be used in distributed setups to mask connection errors.
- InvocationCacheAbstractComposite provides a number of Mixins, Assertions and SideEffects that implement caching for you.
- Trigger using @Cached annotation on methods, in domain interface or Mixin
Invocation Caching

- Trigger using `@Cached` annotation on methods
- Method results are cached with method name+arguments as key as a `SideEffect`
- Cache-aware `Assertions` return cached values always or only on exception
- Invalidate cache explicitly using `InvocationCache` interface or implicitly using `SideEffects` on setters
Aggregation

:: When working with persistent objects (Entities) it is often useful to express Aggregates
  :: Common lifecycle
  :: Common validation rules
:: Aggregated objects implements the interface Aggregated
  :: AggregatedMixin contains reference to “owning” object
:: Owning objects may be Aggregated themselves
Aggregation Tree

- Root
  - Aggregated Object
    - Aggregated Object
    - Aggregated Object
  - Aggregated Object
Validation

- Often there are business rules that need to be checked to verify that an object is in a valid state.
- Check either statically (isValid) or dynamically (Assertion).
- Many validation problems may need to be collected.
  - Web form
Example
DDD and Qi4j
What is Domain Driven Design?

:: Principles

:: Software should be focused on the domain and domain logic

:: Domain design should be based on a model

:: It’s a way of thinking

:: Ubiquitous language

:: Refactoring due to new insights
What is Qi4j?

:: Implementation of Composite Oriented Programming

:: Objects are composed of Fragments
   :: “No” classes

:: Runtime + libraries and extensions

:: Focus on using DDD language in code
DDD sample app

:: Cargo application from the book
:: Implemented by Citerus AB
:: Spring + Hibernate + Spring MVC
:: http://dddsample.sourceforge.net/
:: Ported to Qi4j
  :: Not yet fully adapted to Qi4j “way”
Application structure
Structure in application

:: Ability to declare structure
   :: Modules, Layers, Application
   :: Visibility for dependency injection
   :: Metadata
:: Refactoring tolerant
:: Visualization friendly
Structure in original

:: No explicit structure at all

:: No dependency injection visibility rules

```xml
<bean id="bookingService" class="se.citerus.dddsample.domain.service.impl.BookingServiceImpl">
  <property name="cargoRepository" ref="cargoRepository"/>
  <property name="locationRepository" ref="locationRepository"/>
  <property name="routingService" ref="routingService"/>
</bean>

<bean id="bookingServiceFacade"
  class="se.citerus.dddsample.application.remoting.BookingServiceFacadeImpl">
  <property name="bookingService" ref="bookingService"/>
  <property name="cargoRepository" ref="cargoRepository"/>
  <property name="locationRepository" ref="locationRepository"/>
  <property name="carrierMovementRepository" ref="carrierMovementRepository"/>
</bean>
```
Structure in Qi4j

:: Structure explicitly specified

```java
applicationAssembly.setName( "Cargo sample DDD application" );

// Infrastructure
LayerAssembly infrastructureLayer = createInfrastructureLayer( applicationAssembly );

// Domain layer
LayerAssembly domainLayer = createDomainLayer( applicationAssembly );
domainLayer.uses( infrastructureLayer );

// Application layer
LayerAssembly applicationLayer = createApplicationLayer( applicationAssembly );
applicationLayer.uses( domainLayer );
applicationLayer.uses( infrastructureLayer );
```
private LayerAssembly createDomainLayer( ApplicationAssembly applicationAssembly )
    throws AssemblyException
{
    LayerAssembly modelLayer = applicationAssembly.newLayerAssembly( LAYER_DOMAIN );
    ModuleAssembly cargoModule = modelLayer.newModuleAssembly( MODULE_CARGO );
    cargoModule.addAssembler( new CargoModelAssembler( LOAD_SAMPLE_DATA ) );
    ... more modules ...
}
Structure in Qi4j, module definition

:: Cargo module

```java
module.addComposites(
    RouteSpecificationComposite.class,
    DeliveryHistoryComposite.class
);

module.addEntities(
    CargoEntity.class,
    LegEntity.class,
    ItineraryEntity.class
).visibleIn( layer );

module.addServices(
    ExternalRoutingService.class,
    CargoRepositoryService.class,
    BookingService.class
).visibleIn( application ).instantiateOnStartup();
```
Services
:: Booking service interface

```java
public interface BookingService {
    TrackingId bookNewCargo( UnLocode origin, UnLocode destination );
    List<Itinerary> requestPossibleRoutesForCargo( TrackingId trackingId );
    void assignCargoToRoute( TrackingId trackingId, Itinerary itinerary );
}
```
public interface Booking
{
    TrackingId bookNewCargo( UnLocode origin, UnLocode destination );
    Query<Itinerary> requestPossibleRoutesForCargo( TrackingId trackingId );
    void assignCargoToRoute( TrackingId trackingId, Itinerary itinerary );
}
public interface Query<T>
    extends Iterable<T>, Serializable
{
    Query<T> orderBy( OrderBy... segments );
    Query<T> firstResult( int firstResult );
    Query<T> maxResults( int maxResults );
    T find()
        throws QueryExecutionException;
    void setVariable( String name, Object value );
    <V> V getVariable( String name );
    Class<T> resultType();
    long count()
        throws QueryExecutionException;
}

::Parameters are mandatory by default
::Use @Optional
private RoutingService routingService;
public TrackingId bookNewCargo(final UnLocode originUnLocode, final UnLocode destinationUnLocode) {
    Validate.notNull(originUnLocode);
    Validate.notNull(destinationUnLocode);
    final TrackingId trackingId = cargoRepository.nextTrackingId();
    final Location origin = locationRepository.find(originUnLocode);
    final Location destination = locationRepository.find(destinationUnLocode);
    Cargo cargo = new Cargo(trackingId, origin, destination);
    cargoRepository.save(cargo);
    logger.info("Registered new cargo with tracking id " +
    cargo.trackingId().idString());
    return cargo.trackingId();
}
public void setRoutingService(RoutingService routingService) {
    this.routingService = routingService;
}
Service implementation - Qi4j

```java
@Structure private UnitOfWorkFactory uowf;
@Service private Routing routing;
@Service private LocationRepository locationRepository;
public final TrackingId bookNewCargo(UnLocode originUnLocode, UnLocode destinationUnLocode) {
    Location origin = locationRepository.find(originUnLocode);
    Location destination = locationRepository.find(destinationUnLocode);
   UnitOfWork uow = uowf.currentUnitOfWork();
    EntityBuilder<Cargo> builder = uow.newEntityBuilder(Cargo.class);
    CargoState cargoState = builder.stateFor(CargoState.class);
    cargoState.origin().set(origin);
    cargoState.destination().set(destination);
    Cargo cargo = builder.newInstance();
    return cargo.trackingId();
}
```
Entities
public final class Cargo implements Entity<Cargo> {
    private TrackingId trackingId;
    private Location origin;
    private Location destination;
    private Itinerary itinerary;
    private DeliveryHistory deliveryHistory;
    public Cargo(final TrackingId trackingId, final Location origin, final Location destination) {
        Validate.notNull(trackingId);
        Validate.notNull(origin);
        Validate.notNull(destination);
        this.trackingId = trackingId;
        this.origin = origin;
        this.destination = destination;
    }
    ...
    public boolean equals(final Object object) {
        if (!(object instanceof Cargo)) {
            return false;
        }
        final Cargo other = (Cargo) object;
        return sameIdentityAs(other);
    }
    @Override
    public int hashCode() {
        return trackingId.hashCode();
    }
}
public interface Cargo extends Entity<Cargo>
{
    TrackingId trackingId();
    Location origin();
    void changeDestination( Location newDestination );
    Location destination();
    Itinerary itinerary();
...
}
Entity in Qi4j
DeliveryHistory - alternatives

Cargo with Value Object

Cargo as Entity with Mixin

CargoEntity
CargoMixin  DeliveryHistoryMixin

Cargo
DeliveryHistory
Entities in Qi4j

Cargo composite / mixin

@Mixins( CargoEntity.CargoMixin.class )
interface CargoEntity extends Cargo, DeliveryHistory, EntityComposite
{

class CargoMixin
    implements Cargo
{

    @This DeliveryHistory history;
    @Structure private Module module;
    private final CargoState state;
    private final TrackingId trackingId;

    public CargoMixin(
        @This Identity identity,
        @This CargoState cargoState)
    {
        state = cargoState;
        String trackingIdString = identity.identity().get();
        trackingId = new TrackingId( trackingIdString );
    }

    ...
}
Entities in Qi4j
State

:: Cargo state

interface CargoState
{
    @Immutable Association<Location> origin();
    Association<Location> destination();
    Association<Itinerary> itinerary();
}

:: Identity state

interface Identity
{
    @Immutable Property<String> identity();
}
Lifecycle concerns

final class ItineraryLifecycleConcern  
extends ConcernOf<Lifecycle>  
implements Lifecycle
{
    @Structure UnitOfWorkFactory uowf;
    @This CargoState cargo;
    private static final String EMPTY_ITINERARY_ID = Itinerary.class.getName() + "Empty";
    public void create() throws LifecycleException
    {
        next.create();
        Itinerary emptyItinerary;
        try {
            emptyItinerary = uowf.currentUnitOfWork().find( EMPTY_ITINERARY_ID, Itinerary.class );
        }
        catch( EntityCompositeNotFoundException e ) {
            emptyItinerary = uowf.currentUnitOfWork().newEntity( EMPTY_ITINERARY_ID, Itinerary.class );
        }
        cargo.itinerary().set( emptyItinerary );
    }
    public void remove() throws LifecycleException
    {
        next.remove();
        if (!((Identity)cargo.itinerary().get()).identity().get().equals(EMPTY_ITINERARY_ID))
            ((Lifecycle)cargo.itinerary()).remove();
    }
}
Constraints
public interface BookingServiceFacade extends Remote {
    String registerNewCargo(String origin, String destination) throws RemoteException;
    ...

    public class BookingServiceFacadeImpl implements BookingServiceFacade {
        public String registerNewCargo(String origin, String destination) {
            TrackingId trackingId = bookingService.bookNewCargo(new UnLocode(origin), new UnLocode(destination));
            return trackingId.idString();
        }
        ...

        public UnLocode(final String countryAndLocation) {
            Validate.notNull(countryAndLocation, "Country and location may not be null");
            Validate.isTrue(VALID_PATTERN.matcher(countryAndLocation).matches(),
                    countryAndLocation + " is not a valid UN/LOCODE (does not match pattern)");
            this.unlocode = countryAndLocation.toUpperCase();
        }
    }
}
Constraints in Qi4j

:: Annotations in Booking service facade

```java
public interface BookingFacade extends Remote {
    String registerNewCargo( @Name("origin")
        @Matches("[a-zA-Z]{2}[a-zA-Z2-9]{3}")
    String origin,
        @Name("destination") @UnLocodePattern
    String destination )
    throws RemoteException;
...

@ConstraintDeclaration
@Retention( RetentionPolicy.RUNTIME )
@Matches("[a-zA-Z]{2}[a-zA-Z2-9]{3}")
public @interface UnLocodePattern
{
}
```

:: Will generate one exception with all errors

:: Error messages use i18n through
Unit of Work
Tracking changes

Transaction

Unit of Work

complete()

Usecase execution
Benefits of Unit of Work - pattern

- Less contention on database
- Changes are committed in bulk
- If UoW.complete() fails it can be retried
  - Transaction.commit() failure does not allow retry of commit()
- UoW can be long-running
  - As long as complete() is “short”!
- UoW is easily nestable
Where are we?

:: www.qi4j.org
:: Mailing list
:: Community Plan
   :: Open Source Community vs. business
:: Get involved!
Code in git!

:: git clone git://dscm.ops4j.org/<repository>.git
:: <repository> = qi4j-xxxx
:: -core, -libraries, -extensions
:: -examples, -tutorials, -tests
:: -tools, -ide, -sandbox
:: Get your hands dirty!