COP with Qi4j

Rickard Öberg, Jayway

Agenda

:: What is Qi4j?

:: Describe problems that Qi4j solves

:: Explain Composite Oriented Programming

:: Composites

:: Structures

:: Properties and Associations

What is Qi4j?

What is Qi4j?

:: Qi4j is an implementation of Composite Oriented Programming (COP) on the Java platform

What is Qi4j?

:: Qi4j is an implementation of Composite Oriented Programming (COP) on the Java platform

:: COP is a programming model that allows creation of rich domain models

What is Qi4j?

:: Qi4j is an implementation of Composite Oriented Programming (COP) on the Java platform

:: COP is a programming model that allows creation of rich domain models

:: A rich domain model requires objects to adapt to many different contexts

What is Qi4j?

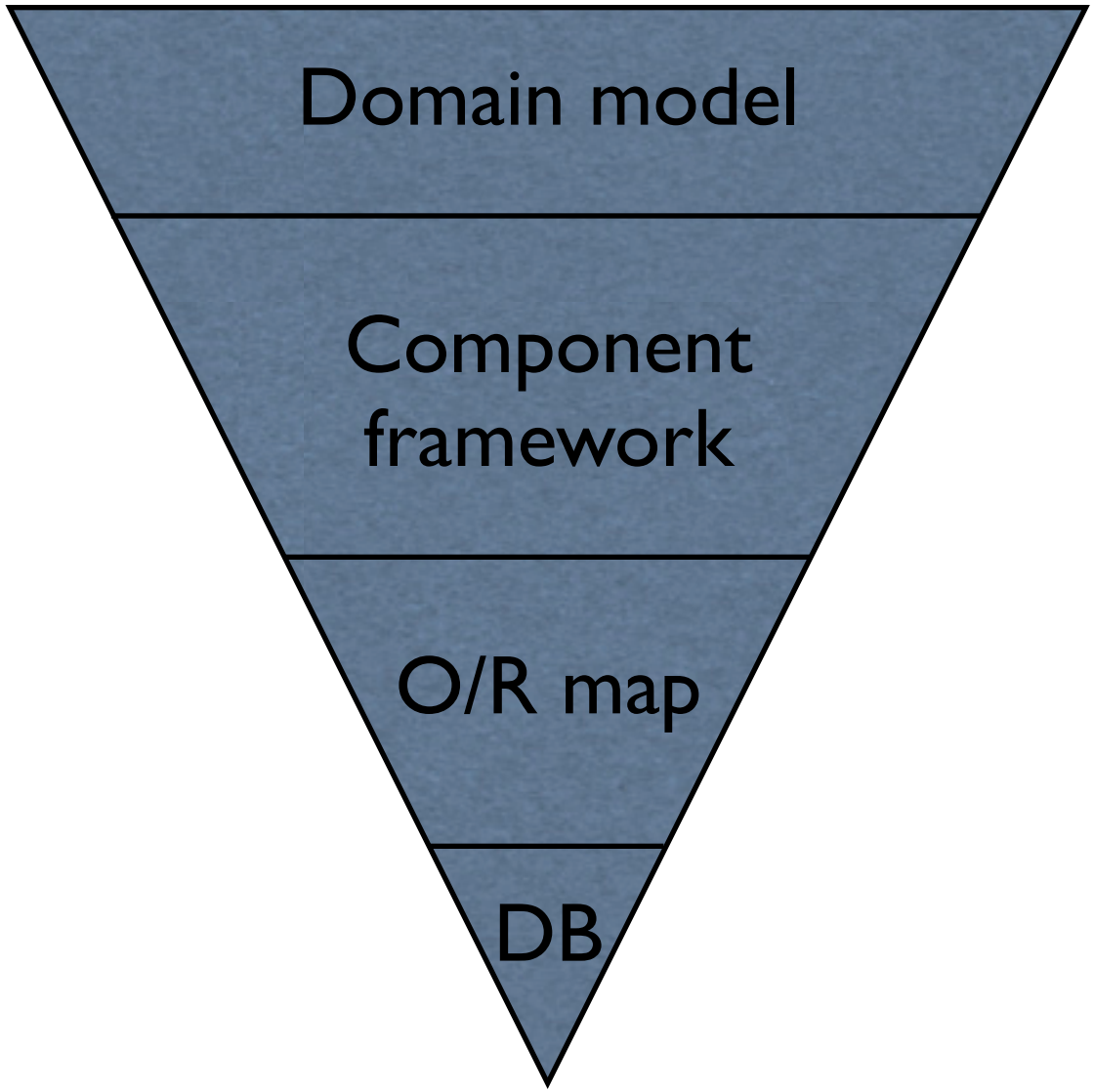
:: Qi4j is an implementation of Composite Oriented Programming (COP) on the Java platform

:: COP is a programming model that allows creation of rich domain models

:: A rich domain model requires objects to adapt to many different contexts

:: Qi4j is nothing new. It is an evolutionary next step based on existing patterns and ideas

Shaky foundation



Flipping the pyramid

Flipping the pyramid

:: Start with the business problem

Flipping the pyramid

:: Start with the business problem

:: Use the terminology from Domain Driven Design

Flipping the pyramid

:: Start with the business problem

:: Use the terminology from Domain Driven Design

:: Allow developer to implement model directly in code using that terminology

Flipping the pyramid

:: Start with the business problem

:: Use the terminology from Domain Driven Design

:: Allow developer to implement model directly in code using that terminology

:: Use infrastructure that can adapt to these needs

Context ignorance

Context ignorance

:: Objects are goooooood

:: In the world we can find and talk about objects

Context ignorance

:: Objects are goooooood

  :: In the world we can find and talk about objects

:: Classes are baaaaaaad

  :: Classification is context sensitive

Context awareness

Context awareness

:: Objects need different interfaces for each context

Context awareness

- Objects need different interfaces for each context
- Compose objects from parts implementing those interfaces

Context awareness

:: Objects need different interfaces for each context

:: Compose objects from parts implementing those interfaces

:: Each part helps the object interact with a specific context

Maintenance Hell

Maintenance Hell

:: “The only constant in the Universe is change”
- Albert Einstein

Maintenance Hell

:: "The only constant in the Universe is change"
- Albert Einstein

:: Inability to deal with change

Maintenance Hell

:: "The only constant in the Universe is change"
- Albert Einstein

:: Inability to deal with change

:: Refactoring limitations

Maintenance Hell

- "The only constant in the Universe is change"
  - Albert Einstein
- Inability to deal with change
  - Refactoring limitations
  - Data schema evolution problems

Maintenance Hell

- "The only constant in the Universe is change"
  - Albert Einstein
- Inability to deal with change
  - Refactoring limitations
  - Data schema evolution problems
  - Growing codebase complexity

Living with change

Living with change

:: Keep domain model definitions in refactorable artifacts (i.e. code)

Living with change

:: Keep domain model definitions in refactorable artifacts (i.e. code)

:: Express queries using domain model

Living with change

:: Keep domain model definitions in refactorable artifacts (i.e. code)

:: Express queries using domain model

:: Separation of storage and indexing

Living with change

:: Keep domain model definitions in refactorable artifacts (i.e. code)

:: Express queries using domain model

:: Separation of storage and indexing

:: Store object version and schema version with each object

Living with change

:: Keep domain model definitions in refactorable artifacts (i.e. code)

:: Express queries using domain model

:: Separation of storage and indexing

:: Store object version and schema version with each object

:: Encourage reuse

Living with change

:: Keep domain model definitions in refactorable artifacts (i.e. code)

:: Express queries using domain model

:: Separation of storage and indexing

:: Store object version and schema version with each object

:: Encourage reuse

:: Structural declaration and visualization

We need change

:: What we have now doesn't work

:: How can we make something new that reuses the good ideas and avoids the bad?

There are good ideas

Scripting

Dependency Injection

There are good ideas

Aspect Oriented Programming

Domain Driven Design

Scripting

Dependency Injection

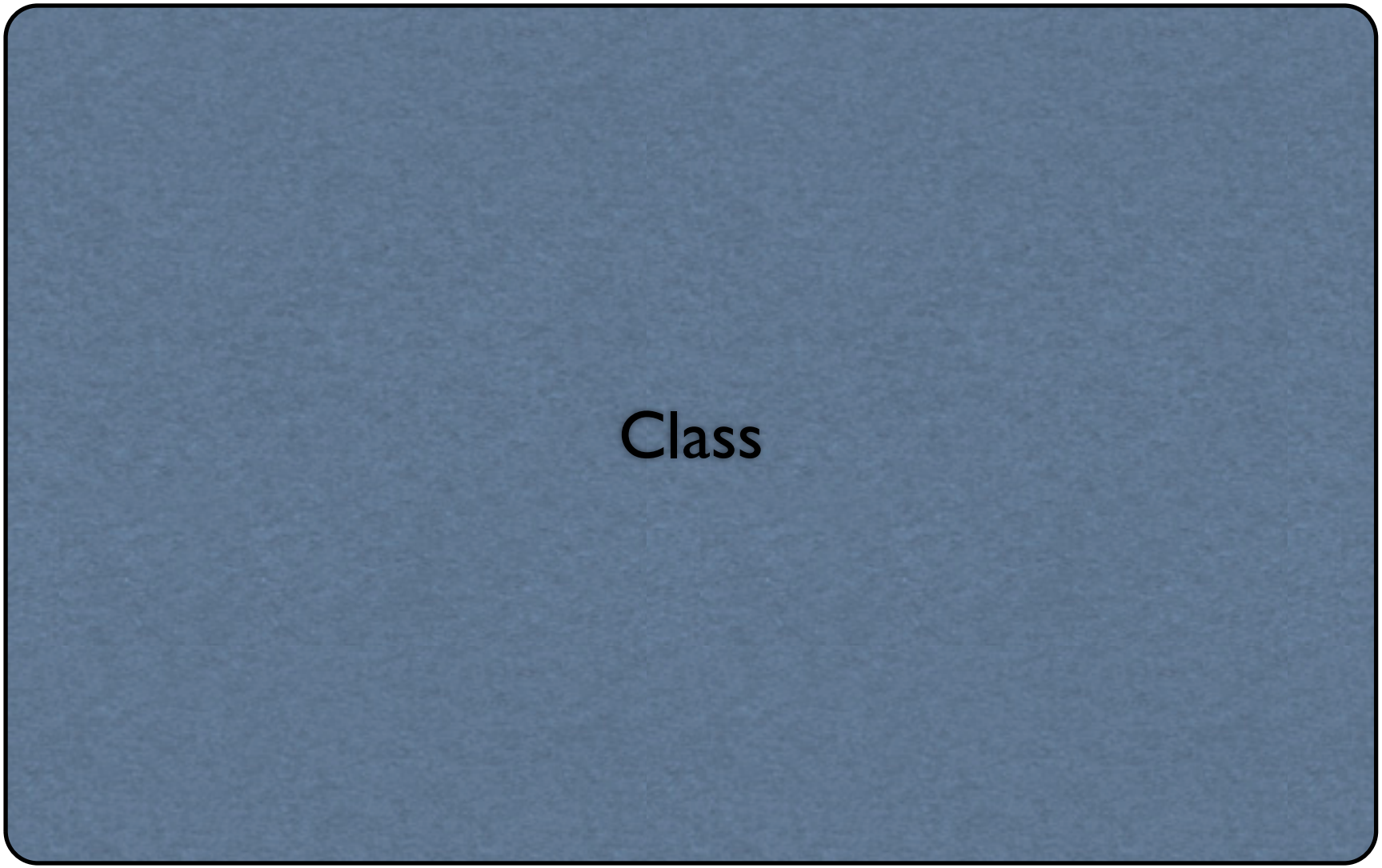
What if we put it all together?

Aspect Oriented Programming

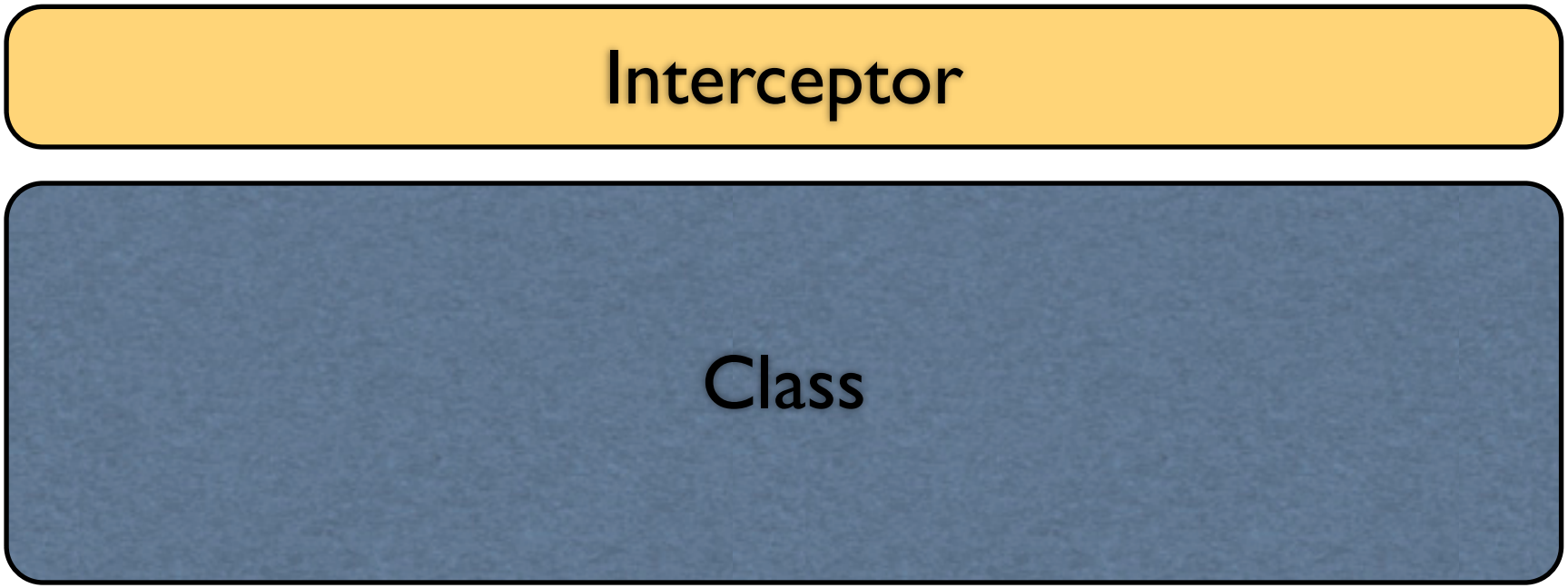
Domain Driven Design

Composite Oriented Programming

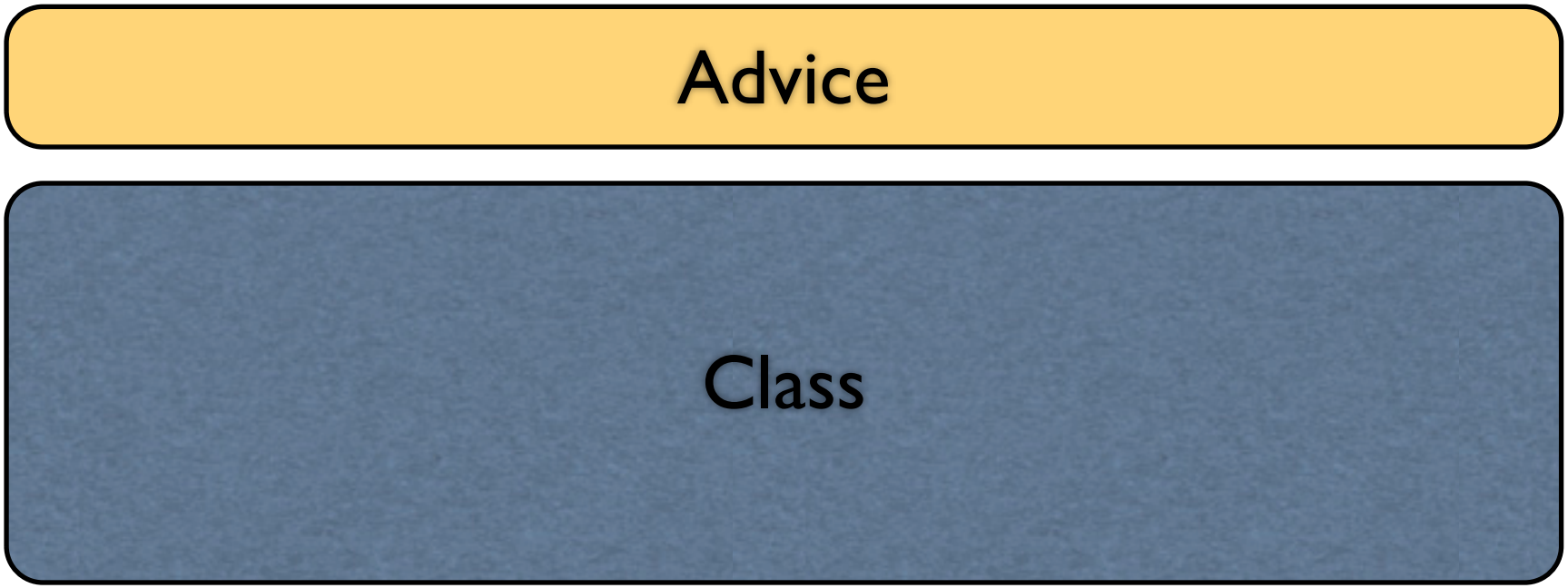
Terminology



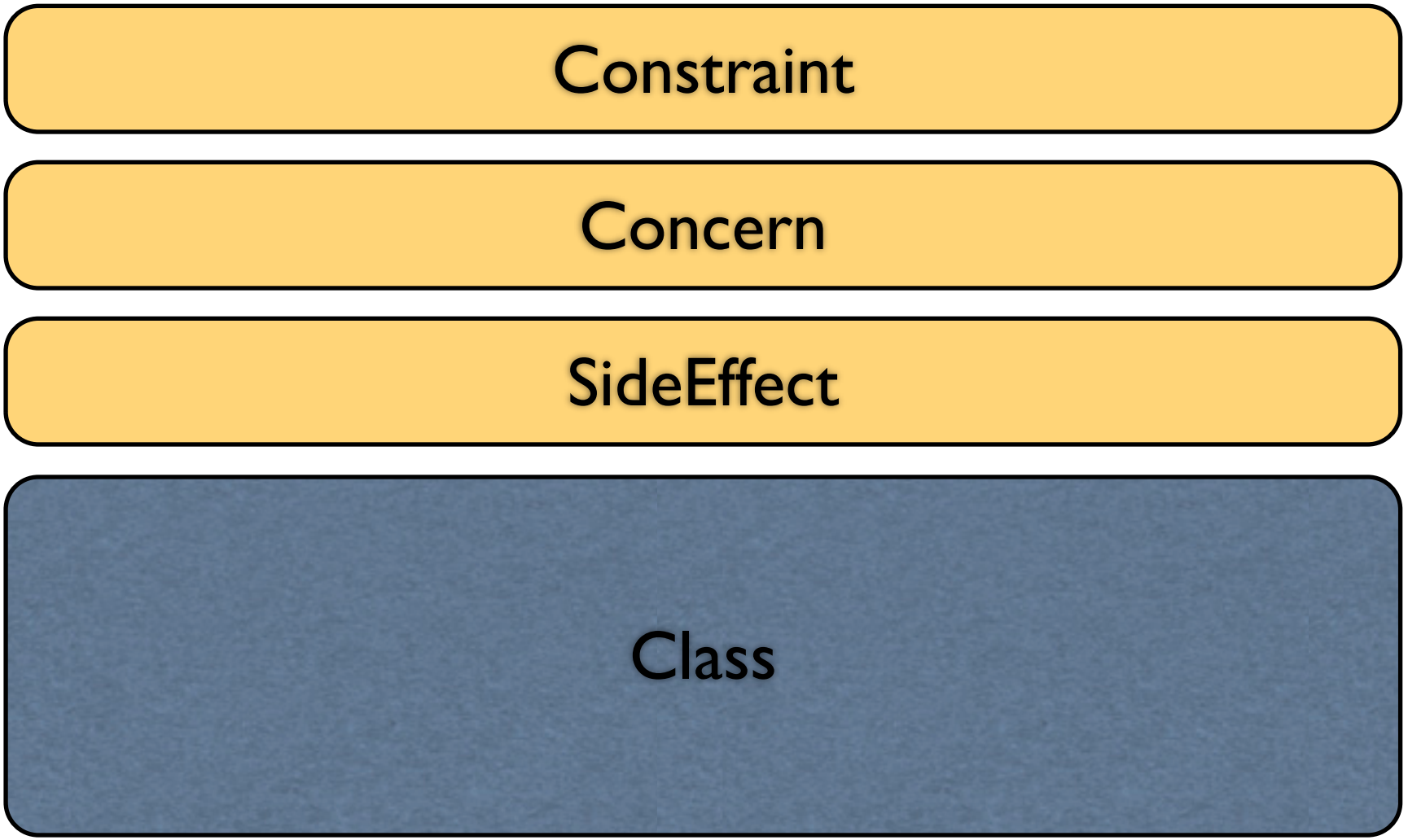
Terminology



Terminology



Terminology



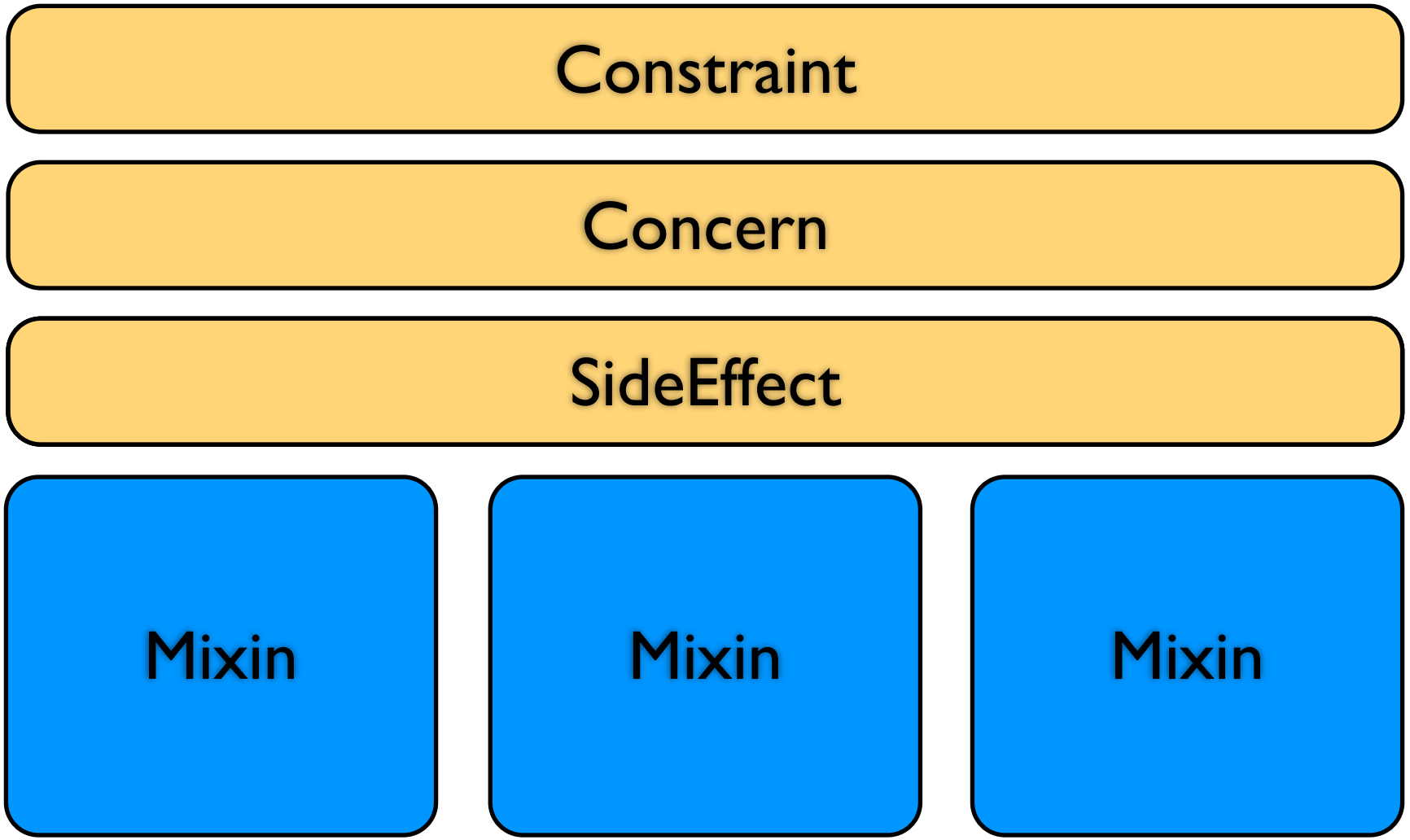
Terminology

Constraint

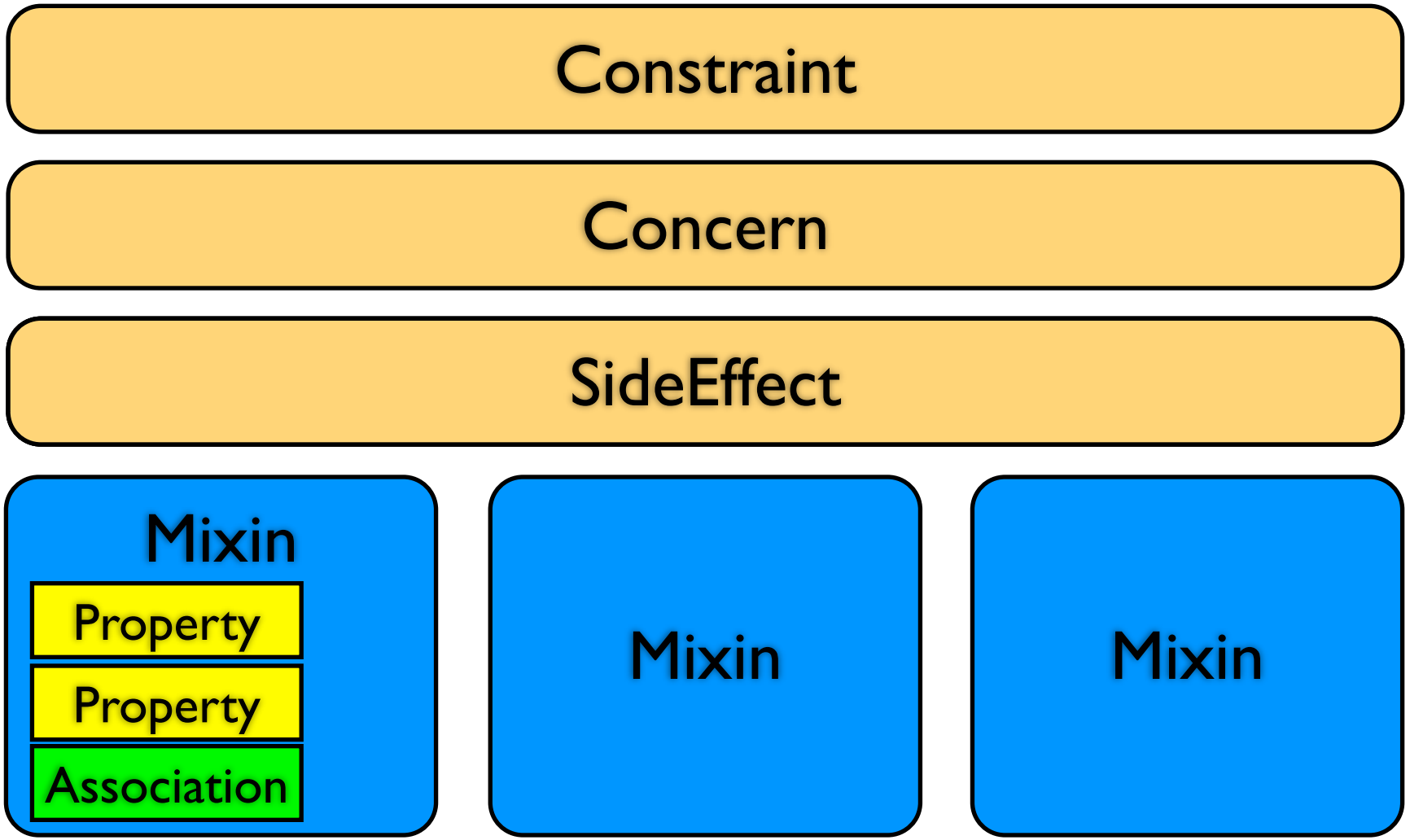
Concern

SideEffect

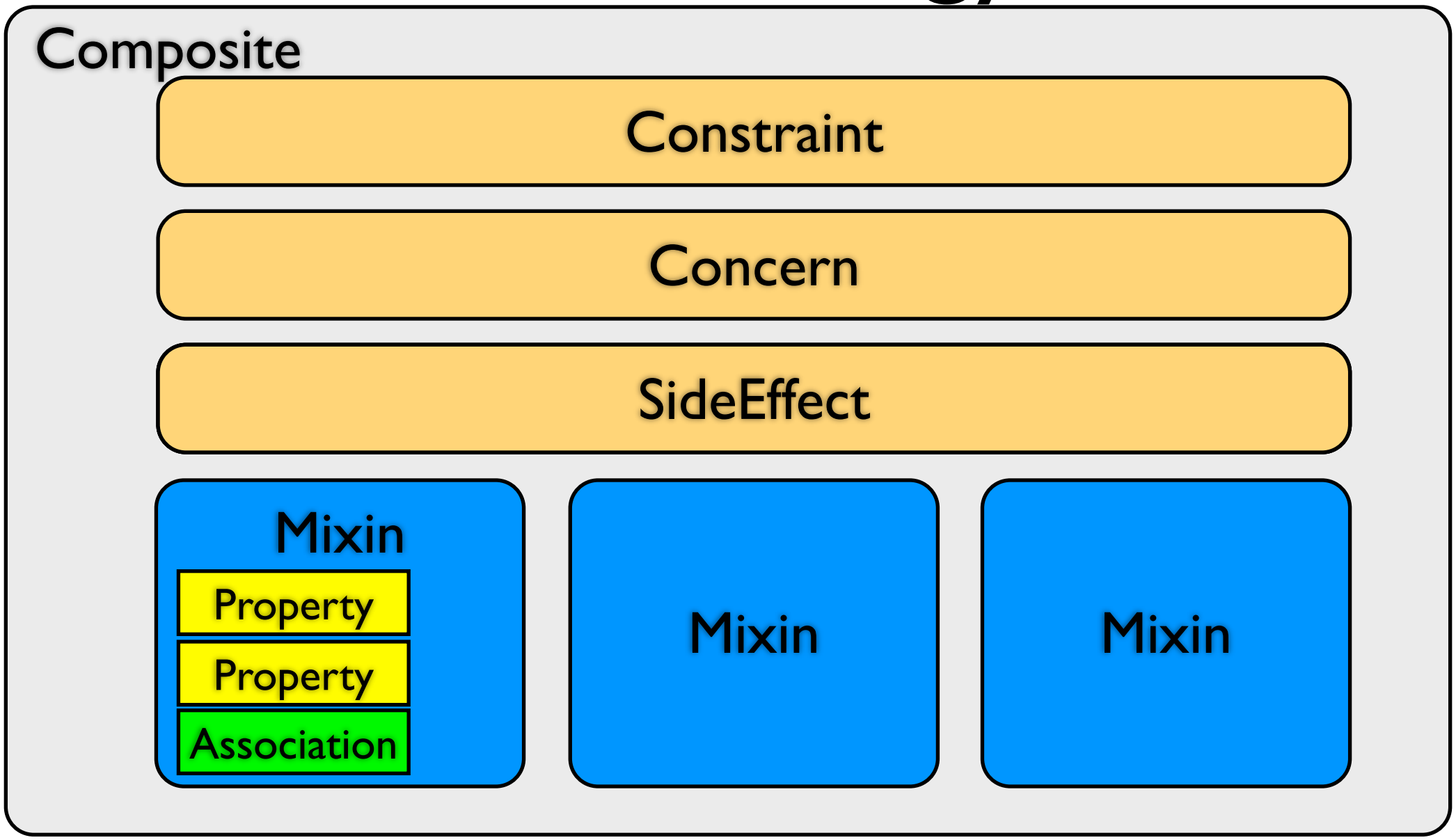
Terminology



Terminology



Terminology



The Small Picture

The Small Picture

:: The most basic element in Qi4j is the Composite

The Small Picture

:: The most basic element in Qi4j is the Composite
:: A Composite is created by composing a number of Fragments.

The Small Picture

:: The most basic element in Qi4j is the Composite
:: A Composite is created by composing a number of Fragments.
:: Mixins are Fragments that can handle method invocations

The Small Picture

:: The most basic element in Qi4j is the Composite
:: A Composite is created by composing a number of Fragments.
:: Mixins are Fragments that can handle method invocations
:: Modifiers are Fragments that modify method invocations (Decorator pattern)
  :: Constraints, Concern, SideEffects

The Big Picture

The Big Picture

:: Composites define the internals of objects

The Big Picture

:: Composites define the internals of objects

:: Composites resides in Modules

The Big Picture

:: Composites define the internals of objects

:: Composites resides in Modules

:: Modules can be grouped into Layers

The Big Picture

:: Composites define the internals of objects

:: Composites resides in Modules

:: Modules can be grouped into Layers

:: Layers form an Application

The Big Picture

:: Composites define the internals of objects

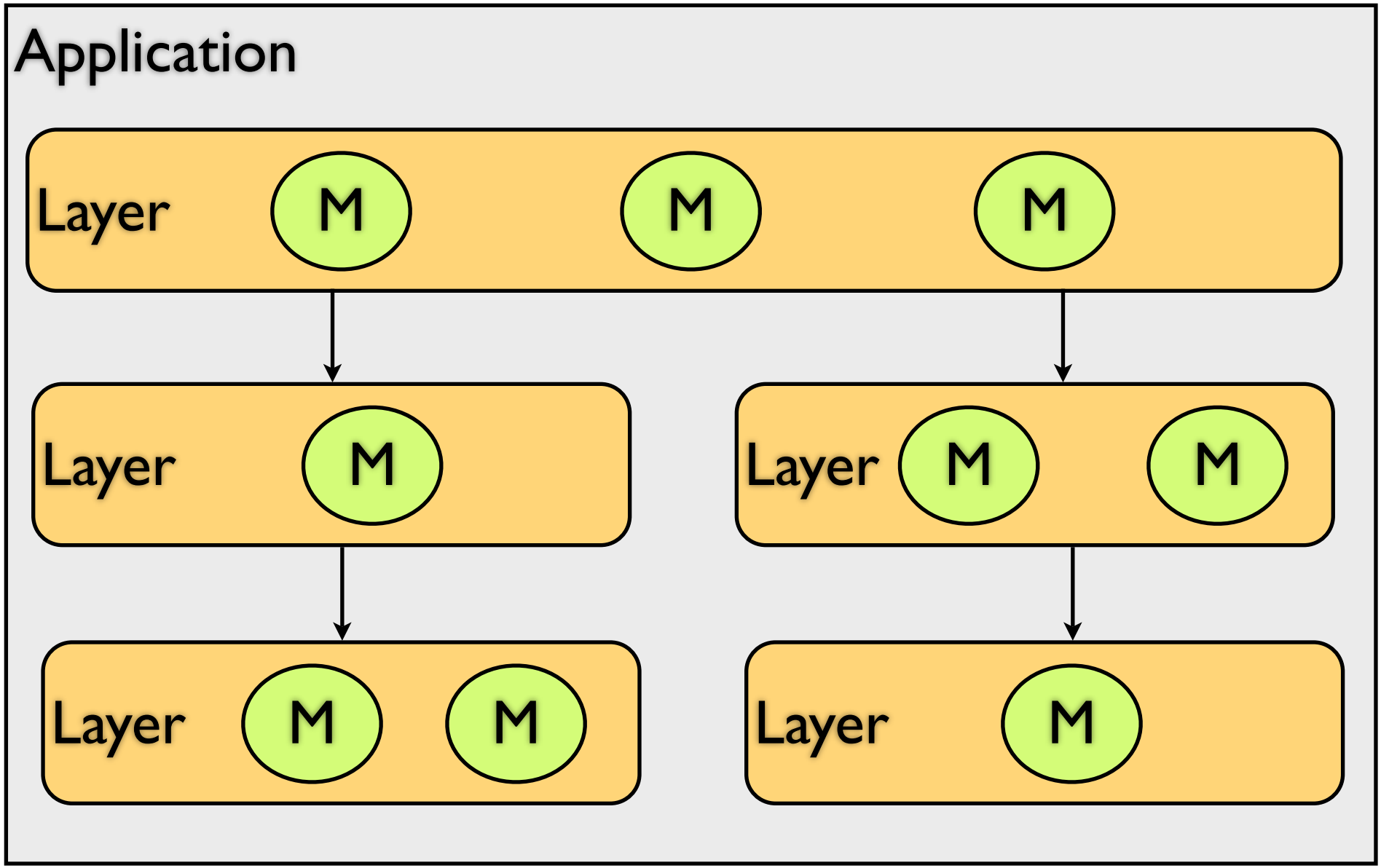
:: Composites resides in Modules

:: Modules can be grouped into Layers

:: Layers form an Application

:: Visibility of Composites between structures is controlled

Structure



```
@Mixins({PropertyMixin.class, AssociationMixin.class})
public interface CarComposite
  extends Composite, Car
{}

public interface Car
  extends Startable, HasWheels, HasEngine, HasOwner
{}

public interface HasOwner
{
  Association<Owner> owner();
}

public interface HasEngine
{
  Property<Engine> engine();
}
```

```
public interface PersonComposite
  extends Composite, Person, Owner
{}
```

```
public interface CompanyComposite
  extends Composite, Company, Owner
{}
```

```
public interface Owner
{
  ManyAssociation<HasOwner> owned();
}
```

@Concerns

- Concerns intercept method calls
  - "around advice" in AOP
- Allowed to modify arguments and return values
- Allowed to return without calling next in chain
- Allowed to throw exceptions

```
@Mixins({PropertyMixin.class, AssociationMixin.class})
@Concerns({CheckClutchConcern.class})
public interface CarComposite
  extends Composite, Car, Startable
{}
```

```
public abstract class CheckClutchConcern
  implements Startable
{
  @ConcernFor Startable next;
  @ThisCompositeAs ClutchStatus clutch;

  public boolean start()
  {
    if (!clutch.engaged().get())
      return false;

    return next.start();
  }
}
```

@Constraints

:: Constraints validates method arguments

:: Can have many Constraints per argument

:: Uses annotations to trigger

:: Cooperate with concern for failure actions

```
@Mixins({PropertyMixin.class, AssociationMixin.class})
@Constraints({FreshOilConstraint.class})
public interface CarComposite
  extends Composite, Car
{}
```

```
public class FreshOilConstraint
  implements Constraint<CheckOil,Oil>
{
  private static final long YEAR = 365*24*3600*1000;

  public boolean isValid(CheckOil annotation, Oil oil)
  {
    Date now = new Date();
    Date expiry = new Date(now.getTime()-YEAR*3);
    return oil.productionDate().get().after(expiry);
  }
}
```

```
public void refillOil(@CheckOil Oil oil);
```

@SideEffects

- Side-effects are called after a method call has finished
- Cannot change method arguments or return value
- Cannot throw exceptions
- Can inspect exceptions and return values
- May be asychronous

```
@Mixins({PropertyMixin.class, AssociationMixin.class})
@SideEffects({StartRadioSideEffect.class})
public interface CarComposite
  extends Composite, Car
{}
```

```
public abstract class StartRadioSideEffect
  implements Startable
{
  @SideEffectFor Startable next;
  @ThisCompositeAs HasRadio radio;

  public boolean start()
  {
    radio.radio().get().start();
    return null; // Ignored anyway
  }
}
```

@Mixins

:: Implements Composite interfaces

:: A Mixin may implement one interface, many interfaces, or only some methods

:: May contain Composite state, such as Property and Association instances

:: May be Composite private - not exposed in Composite interface

```
@Mixins({DistanceToEmptyMixin.class,
PropertyMixin.class, AssociationMixin.class})
public interface CarComposite
  extends Composite, Car
{}
```

```
public abstract class DistanceToEmptyMixin
  implements Car
{
  @ThisCompositeAs HasFuelTank tank;
  @ThisCompositeAs HasFuelConsumption fc;

  public long computeDistanceToEmpty()
  {
    FuelTank fuelTank = tank.fuelTank().get();
    long fuel = fuelTank.fuelLeft().get();
    long consumption = fc.get().current().get();
    return fuel / consumption;
  }
}
```

Summing up

Summing up

:: Business first ➙ Domain Driven Design

Summing up

:: Business first ➙ Domain Driven Design

:: Embrace change ➙ Refactoring friendly

Summing up

:: Business first ➙ Domain Driven Design

:: Embrace change ➙ Refactoring friendly

:: Reduce complexity ➙ Reuse by composition

Summing up

:: Business first ➙ Domain Driven Design

:: Embrace change ➙ Refactoring friendly

:: Reduce complexity ➙ Reuse by composition

:: Classes are dead ➙ Long live interfaces

Summing up

:: Business first ➙ Domain Driven Design

:: Embrace change ➙ Refactoring friendly

:: Reduce complexity ➙ Reuse by composition

:: Classes are dead ➙ Long live interfaces

:: All of the above ➙ Qi4j ☺

Community

:: www.qi4j.org

:: Only in Subversion, no releases (yet)

:: Open participation policy

:: Get involved!

Questions?