Testing

Spock

https://github.com/jakubnabrdalik/spock-examples-wjugflashtalks

Test Driven Development

https://docs.google.com/presentation/d/1JluKIftNzKnf3EWFQSDWPaONm95MFtRT4OZBFpt9uA0

What Spring has to do with it?

Dependency Injection for Integration testing

IoC container caching

Transaction management

Some base support

Outside-in (MVC) testing

Context caching

By default, once loaded, the configured ApplicationContext is reused for each test. Thus the setup cost is incurred only once per test suite, and subsequent test execution is much faster.

In this context, the term *test suite* means all tests run in the same JVM — for example, all tests run from an Ant, Maven, or Gradle build for a given project or module.

Load the context

```
@ContextConfiguration(classes=[Application.class])
@ActiveProfiles(profiles = ['starter.test'])
abstract class IntegrationSpec extends Specification {
    @Autowired
    private AuthenticationManager authenticationManager
```

@Autowired and @Inject but not on constructors

The TestContext framework does not instrument the manner in which a test instance is instantiated. Thus the use of @Autowired or @Inject for constructors has no effect for test classes.

Load web resources
default: file:src/main/webapp

```
@ContextConfiguration(classes=[Application.class])
@ActiveProfiles(profiles = ['starter.test'])
@WebAppConfiguration
abstract class IntegrationSpec extends Specification {
    @Autowired
    private AuthenticationManager authenticationManager
```

Load via ApplicationContextInitializer

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(initializers = MyAppInitializer.class)
public class MyTest {
    // class body...
}
```

Spring Boot shortcut

```
@SpringApplicationConfiguration(classes = SpringExercisesApplication.class)
```

Similar to the standard ContextConfiguration but uses Spring Boot's SpringApplicationContextLoader.

Can be used to test non-web features (like a repository layer) or start an fully-configured embedded servlet container.

Use @WebIntegrationTest to indicate that you want to use a real servlet container or @WebAppConfiguration alone to use a MockServletContext.

For spock you should rather use

```
@ContextConfiguration(classes = SpringExercisesApplication,
                      initializers = ConfigFileApplicationContextInitializer)
```

but hey...

Do not get dirty

```
@DirtiesContext
public class ContextDirtyingTests {


@DirtiesContext(classMode = ClassMode.AFTER_EACH_TEST_METHOD)
public class ContextDirtyingTests {
    // some tests that result in the Spring container being dirtied
}
```

Transactions

```
@TransactionConfiguration(transactionManager = "txMgr",
defaultRollback = false)
public class CustomConfiguredTransactionalTests {
    // class body...
}
```

```
@Transactional
@TransactionConfiguration(defaultRollback = true)
abstract class IntegrationSpec extends Specification {
```

Transactions

```
@BeforeTransaction
public void beforeTransaction() {
    // logic to be executed before a transaction is started
}

@AfterTransaction
public void afterTransaction() {
    // logic to be executed after a transaction has ended
}
```

Properties in tests

requires: SmartContextLoader

```
@ContextConfiguration
@TestPropertySource("classpath:test.properties")
public class MyIntegrationTests {...}
```

```
@ContextConfiguration
//relative to the package in which the test class is defined
@TestPropertySource("/test.properties")
public class MyIntegrationTests {...}
```

Properties in tests

also can be inlined

```
@ContextConfiguration
@TestPropertySource(properties={"timezone = GMT", "port: 4242"})
public class MyIntegrationTests {...}
```

we can inline them in many ways:

"key=value"

"key:value"

"key value"

Properties in tests: the default

If @TestPropertySource is declared as an empty annotation (i.e., without explicit values for the locations or properties attributes), an attempt will be made to detect a default properties file relative to the class that declared the annotation.

```
@ContextConfiguration
@TestPropertySource
public class MyTest {...}
```

"classpath:com/example/MyTest.properties"

Properties in tests: precedence

Test property sources have higher precedence than those loaded from the operating system's environment or Java system properties as well as property sources added by the application declaratively via @PropertySource or programmatically.

Inlined properties override other.

All together: it's good to have a base

```
@Transactional
@TransactionConfiguration(defaultRollback = true)
@ContextConfiguration(classes=[Application.class])
@ActiveProfiles(profiles = ['starter.test'])
@WebAppConfiguration
abstract class IntegrationSpec extends Specification {
...
```

```
class SomeControllerIntegrationSpec extends IntegrationSpec {
    @Autowired SomeController someController
    @Autowired SomeEntityRepository someEntityRepository
...
```

Or make your own annotation

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@ContextConfiguration(classes=[Application.class])
@ActiveProfiles("dev")
@Transactional
public @interface TransactionalDevTest { }
```

Support for JUnit

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes=[Application.class])
@Transactional
@TransactionConfiguration(defaultRollback = true)
abstract class MvcIntegrationTest {
```

Advanced features for JUnit

```
@IfProfileValue(name="test-groups", values={"unit-tests", "integration-tests"})
@Test
public void testProcessWhichRunsForUnitOrIntegrationTestGroups() {
    // some logic that should run only for unit and integration test groups
}


@Timed(millis=1000)
public void testProcessWithOneSecondTimeout() {
    // some logic that should not take longer than 1 second to execute
}


@Repeat(10)
@Test
public void testProcessRepeatedly() {
    // ...
}
```

What WebApplicationContext provides

```
@WebAppConfiguration
@ContextConfiguration
public class WacTests {

    @Autowired WebApplicationContext wac; // cached

    @Autowired MockServletContext servletContext; // cached

    @Autowired MockHttpSession session;

    @Autowired MockHttpServletRequest request;

    @Autowired MockHttpServletResponse response;

    @Autowired ServletWebRequest webRequest;

}
```

Session/Request scoped in tests

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@WebAppConfiguration
public class RequestScopedBeanTests {

    @Autowired UserService userService;
    @Autowired MockHttpServletRequest request;

    @Test
    public void requestScope() {
        //given
        request.setParameter("user", "enigma");
        request.setParameter("pswd", "$pr!ng");

        //when
        LoginResults results = userService.loginUser();
```

Spring Data

Support?

Spring Data JPA

Spring Data MongoDB

Spring Data NEO4J

Spring Data Redis

Spring Data Hadoop

Spring Data Gemfire

Spring Data Rest

Spring Data Solr

and more...

Spring Data JPA

because everyone has Oracle
...or MySQL …or PostgreSQL ...or MSSQL

Database

spring-boot-starter-data-jpa + HSQL/H2/Derby on classpath
= embedded in-memory database

For real projects add to application.properties

```
spring.datasource.url=jdbc:mysql://localhost/test
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.jpa.hibernate.ddl-auto=create-drop
```

Or just register your own DataSource as a bean

Don’t forget to have the driver on the classpath!

Database

To pass params to hibernate entity manager, just start it with spring.jpa.properties.hibernate:

application.properties

spring.jpa.properties.hibernate.globally_quoted_identifiers=true

hibernate sees:

hibernate.globally_quoted_identifiers=true

The common CRUD needs

Save an entity

Return the entity identified by the given id

Return all entities

Return the number of entities

Delete the given entity

Indicate whether an entity with the given id exists

Repository

```
public interface Repository<T, ID extends Serializable> {}

public interface CrudRepository<T, ID extends Serializable>
                                                    extends Repository<T, ID> {
    <S extends T> S save(S entity);
    <S extends T> Iterable<S> save(Iterable<S> entities);
    T findOne(ID id);
    boolean exists(ID id);
    Iterable<T> findAll();
    Iterable<T> findAll(Iterable<ID> ids);
    long count();
    void delete(ID id);
    void delete(T entity);
    void delete(Iterable<? extends T> entities);
    void deleteAll();
}
```

Pageable Repository

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
                                        extends CrudRepository<T, ID> {
    Iterable<T> findAll(Sort sort);
    Page<T> findAll(Pageable pageable);
}
```

```
public interface Pageable {
    int getPageNumber();
    int getPageSize();
    int getOffset();
    Sort getSort();
    Pageable next();
    Pageable previousOrFirst();
    Pageable first();
    boolean hasPrevious();
}
```

```
public class PageRequest
                    extends AbstractPageRequest {
  public PageRequest(int page, int size) {
     ...
  }
…
}

public abstract class AbstractPageRequest
                    implements Pageable, Serializable ...
```

```
Page<User> users = repository.findAll(new PageRequest(1, 20));
```

And finally - JPA Repository

```
public interface JpaRepository<T, ID extends Serializable> extends
                                                PagingAndSortingRepository<T, ID> {
    List<T> findAll();
    List<T> findAll(Sort sort);
    List<T> findAll(Iterable<ID> ids);
    <S extends T> List<S> save(Iterable<S> entities);
    void flush();
    T saveAndFlush(T entity);
    void deleteInBatch(Iterable<T> entities);
    void deleteAllInBatch();
    T getOne(ID id);
}
```

How to set it up?

1. Create your interface

```
interface SomeEntityRepository extends JpaRepository<SomeEntity, Long> {
}
```

2. Register your repositories in Spring

assuming you have transactionManager and entityManagerFactory

```
@EnableJpaRepositories("eu.solidcraft.starter.domain")
```

3. Inject and start using your new repository

```
List<SomeEntity> entities = someEntityRepository.findAll();
```

But all I have is an interface...

Spring will create the implementation for you.

Because it’s boring.

Which repository to extend?

You can extend any repository, but if you need CRUD and Paging, use JpaRepository.

But do not always extend JpaRepository blindly.

If all you need are your own, very specific data access methods...

Not extending an interface?

Instead of extending Repository, you can annotate your interface with **@RepositoryDefinition**

Or you can annotate your interface, that extends Repository, with **@NoRepositoryBean.** This way you can create abstracts (base for your repos).

How to fine-tune your repository

```
@Entity
class SomeEntity {
   @NotNull private String username;

   …
}


interface SomeEntityRepository extends JpaRepository<SomeEntity, Long> {
   List<SomeEntity> findByUsername(String username)
}
```

Query method examples

```
List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

// Enables the distinct flag for the query
List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

// Enabling ignoring case for an individual property
List<Person> findByLastnameIgnoreCase(String lastname);

// Enabling ignoring case for all suitable properties
List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

// Enabling static ORDER BY for a query
List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
```

What it understands?

property traversals

AND | OR

operators (Between, LessThan, Like, In)

IgnoreCase (for each property)

OrderBy (Asc | Desc)

FirstX, TopY

more...

Nested properties

findByAddressZipCode(ZipCode zipCode);

//check if property exists
person.addressZipCode

//split camel case and find
person.addressZip.Code
person.address.zipCode
person.address.zip.code

Nested properties

//what if we have both?

person.address.zipCode

person.address.zip.code

//tell it directly

findByAddress_ZipCode(ZipCode zipCode);

Query creation anatomy

find…By, read…By, query…By, count…By, and get…By

first “By” marks the moment of parsing

Custom JPA queries

```
public interface UserRepository extends JpaRepository<User, Long> {
  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);

  @Query("select u from User u where u.firstname like %?1")
  List<User> findByFirstnameEndsWith(String firstname);

  @Query(
  "select u from User u where u.firstname = :firstname or
   u.lastname = :lastname")
  User findByLastnameOrFirstname(
                        @Param("lastname") String lastname,
                        @Param("firstname") String firstname);
}
```

Custom JPA queries

You can load data directly into DTOs (no Entity required):

```
  @Query("SELECT NEW com.company.PublisherInfo(pub.id,
pub.revenue, mag.price) FROM Publisher pub JOIN pub.magazines mag
WHERE mag.price > ?1")
  PublisherInfo findByEmailAddress(BigDecimal price);
```

Custom Native queries

```
public interface UserRepository extends JpaRepository<User, Long> {
  @Query(
      value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1",
      nativeQuery = true)
  User findByEmailAddress(String emailAddress);
}
```

Special parameters

Page<User> findByLastname(String lastname, Pageable pageable);

Slice<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);

Slice

```
public interface Slice<T> extends Iterable<T> {
  int getNumber();
  int getSize();
  int getNumberOfElements();
  List<T> getContent();
  boolean hasContent();
  Sort getSort();
  boolean isFirst();
  boolean isLast();
  boolean hasNext();
  boolean hasPrevious();
  Pageable nextPageable();
  Pageable previousPageable();
}
```

Page vs Slice

A Page knows about the total number of elements and pages available. It does so by the infrastructure triggering a **count query** to calculate the overall number.

As this might be expensive depending on the store used, Slice can be used as return instead. A Slice only knows about **whether there's a next Slice** available which might be just sufficient when walking though a larger result set.

Sort

Build in Pageable & Slice

You can have only sorting, if you wish.

```
public Sort(Order... orders) {...
public Sort(Direction direction, String... properties) {...
public Sort and(Sort sort) {...
```

Custom logic in repository

//step 1: define your custom method in an interface

```
interface UserRepositoryCustom {
  public void someCustomMethod(User user);
}
```

//step 2: implement it

```
class UserRepositoryCustomImpl implements UserRepositoryCustom {
  public void someCustomMethod(User user) {
    // Your custom implementation
  }
}
```

Custom logic in repository

//step 3: declare your interface extending both your custom, and Spring Data repository

```
public interface UserRepository
                extends CrudRepository<User, Long>, UserRepositoryCustom {
  // Declare query methods here
}
```

If you use namespace configuration, the repository infrastructure tries to autodetect custom implementations by **scanning for classes below the package we found a repository in**. These classes need to follow the naming convention of appending the namespace element's attribute repository-impl-postfix to the found repository interface name. This postfix defaults to **Impl**.

But you can change it:

```
<repositories base-package="com.acme.repository" repository-impl-postfix="Customization" />
```

Custom logic in repository

If your custom implementation bean needs special wiring, you simply declare the bean and name it after the conventions just described.

The infrastructure will then refer to the manually defined bean definition by name instead of creating one itself.

What about Java 8 default methods?

Only if all you need is access to other repository methods.

Because you have no access to any state of the instance, you just have an interface.

```
default Owner getSafeCopy(Long id) {
   Owner owner = findById(id);
   return new Owner(owner).withAccountNumber(“stripped”);
}
```

Prepopulating the database

```
@Entity
class SomeEntity {
   @Id
   @SequenceGenerator(name = "SomeSequence",
    sequenceName = "SEQ_SOME_PK", initialValue=10000)
   @GeneratedValue(generator = "SomeSequence")
   private Long id;

   @NotNull
   private String username;

   @NotNull
   private BigDecimal someAmount;

   @NotNull
   private Date someDate;
…
}
```

Prepopulating the database

```
[
  {
    "_class" : "eu.solidcraft.starter.domain.some.SomeEntity",
    "id" : 10000,
    "username" : "test",
    "someAmount": 100,
    "someDate": "2009-04-12T20:44:55"
  },
  {
    "_class" : "eu.solidcraft.starter.domain.some.SomeEntity",
    "id" : 10001,
    "username" : "test",
    "someAmount": 50,
    "someDate": "2009-04-12T20:44:55"
  },
  {
    "_class" : "eu.solidcraft.starter.domain.some.SomeEntity",
    "id" : 10002,
    "username" : "test",
    "someAmount": 30,
    "someDate": "2009-04-12T20:44:55"
  }
]
```

Prepopulating the database

```
<beans profile="starter.development">
   <repository:jackson-populator locations="classpath:predefinedData.json" />
</beans>
```

Or:

```
@Configuration
class ApplicationConfig {

  @Bean
  public JacksonRepositoryPopulatorFactoryBean repositoryPopulator() {
    Resource sourceData = new ClassPathResource("test-data.json");

    JacksonRepositoryPopulatorFactoryBean factory = new JacksonRepositoryPopulatorFactoryBean();
    factory.setObjectMapper(…);      //custom ObjectMapper if needed
    factory.setResources(new Resource[] { sourceData });
    return factory;
  }
}
```

Auditing

@CreatedBy, @LastModifiedBy, @CreatedDate, @LastModifiedDate

```
class Customer {

  @CreatedBy
  private User user;

  @CreatedDate
  private DateTime createdDate;
}
```

or implement **Auditable**

or extend **AbstractAuditable**

Auditing

```
class SpringSecurityAuditorAware implements AuditorAware<User> {

  public User getCurrentAuditor() {

    Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

    if (authentication == null || !authentication.isAuthenticated()) {
      return null;
    }

    return ((MyUserDetails) authentication.getPrincipal()).getUser();
  }
}
```

JPA 2.1 support

@Modyfing - the query will change DB state

@QueryHints

@EntityGraph & @NamedEntityGraph

@NamedStoredProcedureQuery & @Procedure

Criteria… errr… Specification

Criteria - Hibernate, JPA 2

Specification - Spring

```
public interface CustomerRepository extends CrudRepository<Customer, Long>,
                                            JpaSpecificationExecutor { … }
```

and you get `List<T> findAll(Specification<T> spec)` in your repo

```
public interface Specification<T> {
  Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,CriteriaBuilder builder);
}
```

Pays back when you have to combine them

```
List<Customer> customers = customerRepository.findAll(
  where(isLongTermCustomer()).or(hasSalesOfMoreThan(amount)));


public class Specifications<T> implements Specification<T> {
  private final Specification<T> spec;
  private Specifications(Specification<T> spec) {...}
  public static <T> Specifications<T> where(Specification<T> spec) {...}
  public Specifications<T> and(final Specification<T> other) {... }
  public Specifications<T> or(final Specification<T> other) {...}
  public static <T> Specifications<T> not(final Specification<T> spec) {...}
  public Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder builder) {...}
}
```

Transactions

because the spice must flow

Declarative transaction management

Setup assuming you have transactionManager:

@EnableTransactionManagement(proxyTargetClass=”true”)

Declarative transaction management

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        // do something
    }

    // these settings have precedence for this method
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // do something
    }
}
```

Transaction attributes

Isolation - what the transaction can see

ISOLATION_DEFAULT
ISOLATION_READ_UNCOMMITTED
ISOLATION_READ_COMMITTED
ISOLATION_REPEATABLE_READ
ISOLATION_SERIALIZABLE

Transaction attributes

Propagation - what if transaction already exists

REQUIRED *Support a current transaction, create a new one if none exists.*
SUPPORTS *Support a current transaction, execute non-transactionally if none exists.*
MANDATORY *Support a current transaction, throw an exception if none exists.*
REQUIRES_NEW *Create a new transaction, suspend the current transaction if one exists.*
NOT_SUPPORTED *Execute non-transactionally, suspend the current transaction if one exists.*
NEVER *Execute non-transactionally, throw an exception if a transaction exists.*
NESTED *Execute within a nested transaction if a current transaction exists,*

Transaction attributes

Timeout

Read-only status

rollbackFor + rollbackForClassName

noRollbackFor + noRollbackForClassName

Default @Transactional attributes

Propagation = PROPAGATION_REQUIRED.
Isolation = ISOLATION_DEFAULT.
Transaction = read/write.

Transaction timeout = default timeout of the underlying transaction system, or to none if timeouts are not supported.

Any RuntimeException triggers rollback, and any checked Exception does not.

Transactions

All repository methods are transactional by default (queries with readOnly flag).

Where to start transactions

Open session in view - Transaction per Request

OpenSessionInViewInterceptor and OpenSessionInViewFilter

Transaction on Controllers

Transaction on Services

Transaction on Repositories

Open Session in View

Advantages:

you can do lazy loading in view layer

you can be lazy

Disadvantages:

lazy loading is very bad for you (performance)

coupling view with sesion

if you get an exception, you are out of luck

many more, depending who you ask

Transaction on Controllers

Advantages:

popular in Anemic model

makes you eager load more

Disadvantages:

anemic model is often bad for you (complexity, maintainability)

why is your presentation layer handling database?

Transaction on Services

Advantages:

popular in both Rich and Anemic model

makes you eager load even more

looks like it is the logical thing to do

Disadvantages:

you need services

you can end up with services just for transactions

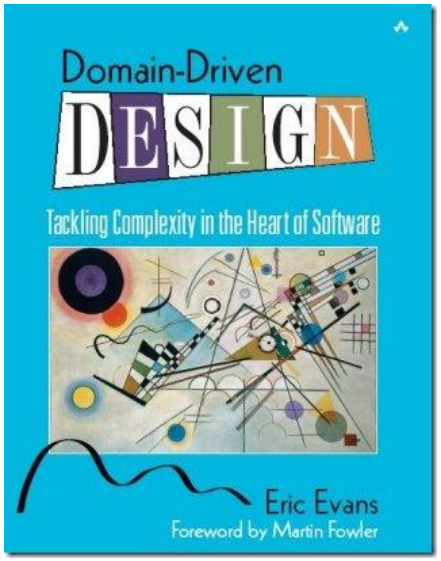
Transaction on Repositories

Errrr….

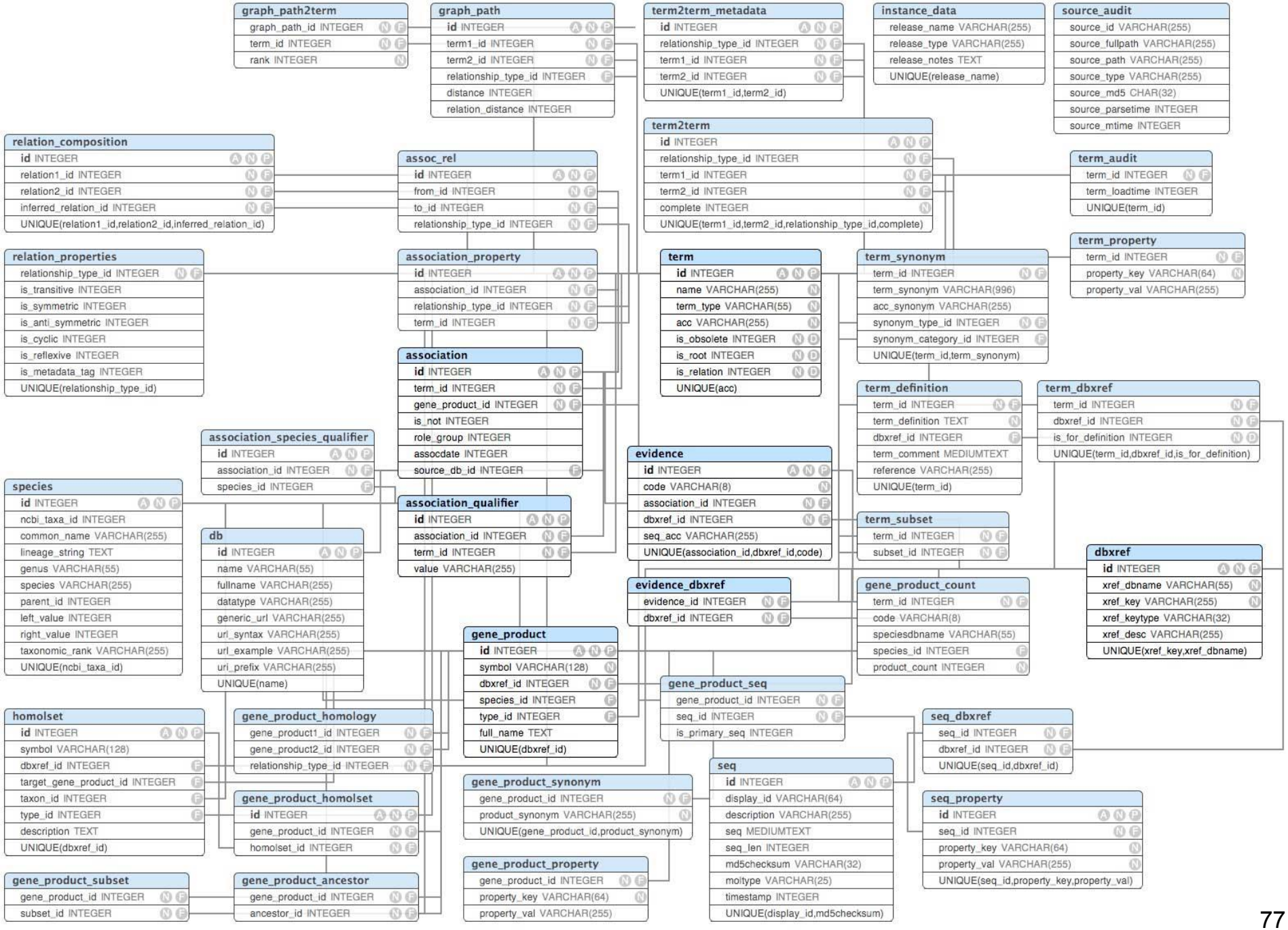
Word of advice

Why do you have to make a single choice for the whole project?

Bounded contexts and aggregates



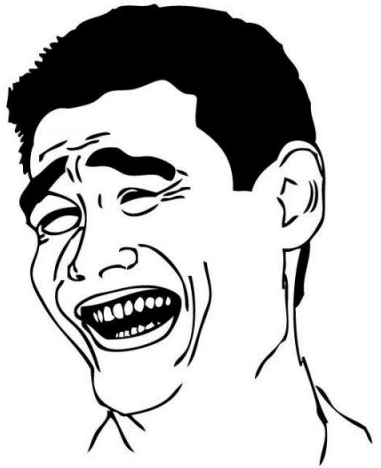
because we have to handle that complexity


graph_path2term
graph_path_id INTEGER
term_id INTEGER
rank INTEGER
graph_path
id INTEGER
term1_id INTEGER
term2_id INTEGER
relationship_type_id INTEGER
distance INTEGER
relation_distance INTEGER
term2term_metadata
id INTEGER
relationship_type_id INTEGER
term1_id INTEGER
term2_id INTEGER
UNIQUE(term1_id,term2_id)
instance_data
release_name VARCHAR(255)
release_type VARCHAR(255)
release_notes TEXT
UNIQUE(release_name)
source_audit
source_id VARCHAR(255)
source_fullpath VARCHAR(255)
source_path VARCHAR(255)
source_type VARCHAR(255)
source_md5 CHAR(32)
source_parsetime INTEGER
source_mtime INTEGER
relation_composition
id INTEGER
relation1_id INTEGER
relation2_id INTEGER
inferred_relation_id INTEGER
UNIQUE(relation1_id,relation2_id,inferred_relation_id)
assoc_rel
id INTEGER
from_id INTEGER
to_id INTEGER
relationship_type_id INTEGER
term2term
id INTEGER
relationship_type_id INTEGER
term1_id INTEGER
term2_id INTEGER
complete INTEGER
UNIQUE(term1_id,term2_id,relationship_type_id,complete)
term_audit
term_id INTEGER
term_loadtime INTEGER
UNIQUE(term_id)
relation_properties
relationship_type_id INTEGER
is_transitive INTEGER
is_symmetric INTEGER
is_anti_symmetric INTEGER
is_cyclic INTEGER
is_reflexive INTEGER
is_metadata_tag INTEGER
UNIQUE(relationship_type_id)
association_property
id INTEGER
association_id INTEGER
relationship_type_id INTEGER
term_id INTEGER
term
id INTEGER
name VARCHAR(255)
term_type VARCHAR(55)
acc VARCHAR(255)
is_obsolete INTEGER
is_root INTEGER
is_relation INTEGER
UNIQUE(acc)
term_synonym
term_id INTEGER
term_synonym VARCHAR(996)
acc_synonym VARCHAR(255)
synonym_type_id INTEGER
synonym_category_id INTEGER
UNIQUE(term_id,term_synonym)
term_property
term_id INTEGER
property_key VARCHAR(64)
property_val VARCHAR(255)
association
id INTEGER
term_id INTEGER
gene_product_id INTEGER
is_not INTEGER
role_group INTEGER
assocdate INTEGER
source_db_id INTEGER
term_definition
term_id INTEGER
term_definition TEXT
dbxref_id INTEGER
term_comment MEDIUMTEXT
reference VARCHAR(255)
UNIQUE(term_id)
term_dbxref
term_id INTEGER
dbxref_id INTEGER
is_for_definition INTEGER
UNIQUE(term_id,dbxref_id,is_for_definition)
association_species_qualifier
id INTEGER
association_id INTEGER
species_id INTEGER
evidence
id INTEGER
code VARCHAR(8)
association_id INTEGER
dbxref_id INTEGER
seq_acc VARCHAR(255)
UNIQUE(association_id,dbxref_id,code)
species
id INTEGER
ncbi_taxa_id INTEGER
common_name VARCHAR(255)
lineage_string TEXT
genus VARCHAR(55)
species VARCHAR(255)
parent_id INTEGER
left_value INTEGER
right_value INTEGER
taxonomic_rank VARCHAR(255)
UNIQUE(ncbi_taxa_id)
association_qualifier
id INTEGER
association_id INTEGER
term_id INTEGER
value VARCHAR(255)
term_subset
term_id INTEGER
subset_id INTEGER
db
id INTEGER
name VARCHAR(55)
fullname VARCHAR(255)
datatype VARCHAR(255)
generic_url VARCHAR(255)
url_syntax VARCHAR(255)
url_example VARCHAR(255)
uri_prefix VARCHAR(255)
UNIQUE(name)
dbxref
id INTEGER
xref_dbname VARCHAR(55)
xref_key VARCHAR(255)
xref_keytype VARCHAR(32)
xref_desc VARCHAR(255)
UNIQUE(xref_key,xref_dbname)
evidence_dbxref
evidence_id INTEGER
dbxref_id INTEGER
gene_product_count
term_id INTEGER
code VARCHAR(8)
speciesdbname VARCHAR(55)
species_id INTEGER
product_count INTEGER
gene_product
id INTEGER
symbol VARCHAR(128)
dbxref_id INTEGER
species_id INTEGER
type_id INTEGER
full_name TEXT
UNIQUE(dbxref_id)
gene_product_seq
gene_product_id INTEGER
seq_id INTEGER
is_primary_seq INTEGER
homolset
id INTEGER
symbol VARCHAR(128)
dbxref_id INTEGER
target_gene_product_id INTEGER
taxon_id INTEGER
type_id INTEGER
description TEXT
UNIQUE(dbxref_id)
gene_product_homology
gene_product1_id INTEGER
gene_product2_id INTEGER
relationship_type_id INTEGER
seq_dbxref
seq_id INTEGER
dbxref_id INTEGER
UNIQUE(seq_id,dbxref_id)
gene_product_synonym
gene_product_id INTEGER
product_synonym VARCHAR(255)
UNIQUE(gene_product_id,product_synonym)
seq
id INTEGER
display_id VARCHAR(64)
description VARCHAR(255)
seq MEDIUMTEXT
seq_len INTEGER
md5checksum VARCHAR(32)
moltype VARCHAR(25)
timestamp INTEGER
UNIQUE(display_id,md5checksum)
gene_product_homolset
id INTEGER
gene_product_id INTEGER
homolset_id INTEGER
seq_property
id INTEGER
seq_id INTEGER
property_key VARCHAR(64)
property_val VARCHAR(255)
UNIQUE(seq_id,property_key,property_val)
gene_product_property
gene_product_id INTEGER
property_key VARCHAR(64)
property_val VARCHAR(255)
gene_product_subset
gene_product_id INTEGER
subset_id INTEGER
gene_product_ancestor
gene_product_id INTEGER
ancestor_id INTEGER

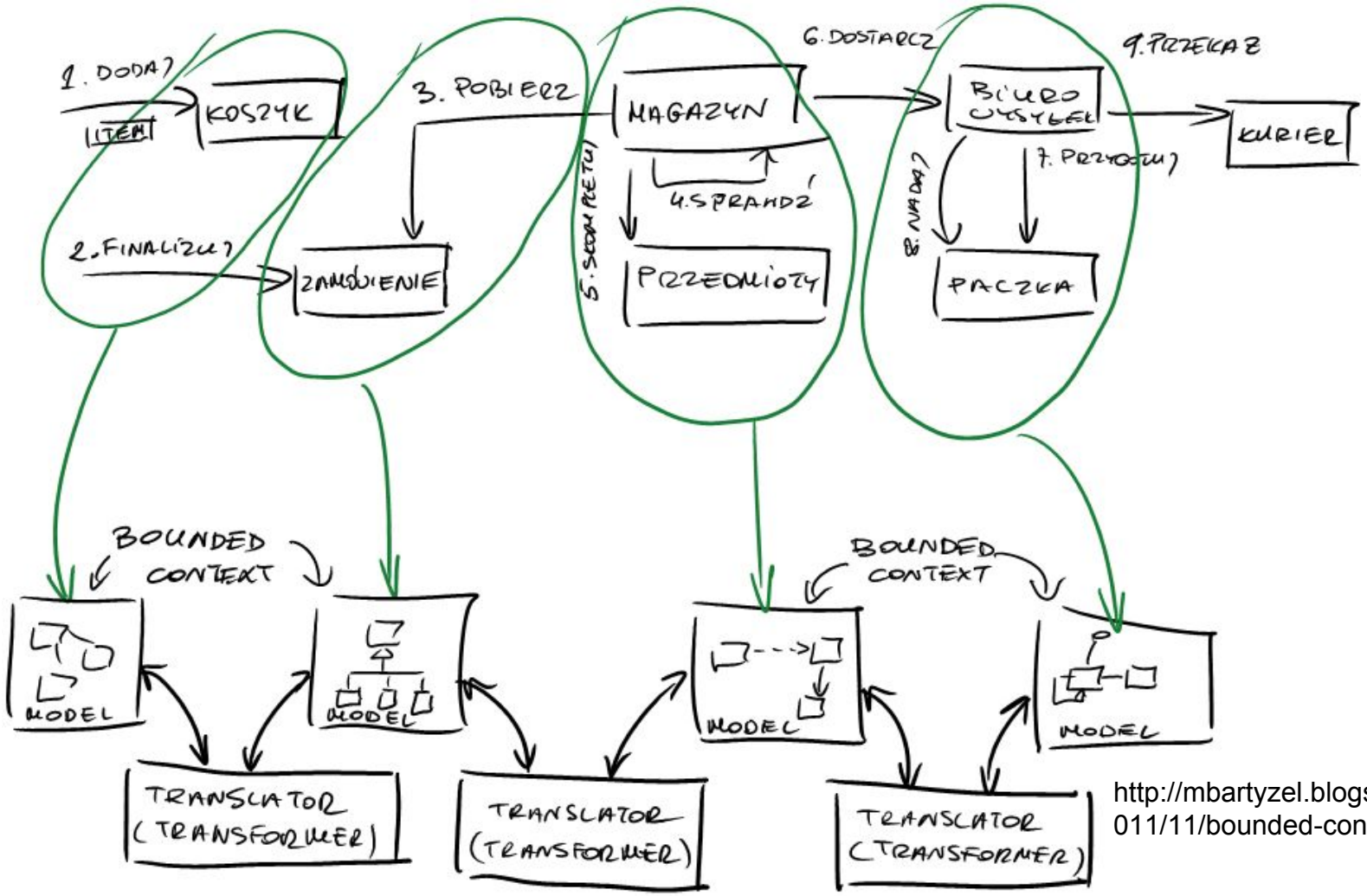
How do you put that into your head?

"I worked at a place that had several hundred tables (near 1k) and no one really knew what was going on in the system, company was growing and hiring a lot. "

http://stackoverflow.com/questions/3092668/printing-an-er-diagram-for-mysql-database-800-tables

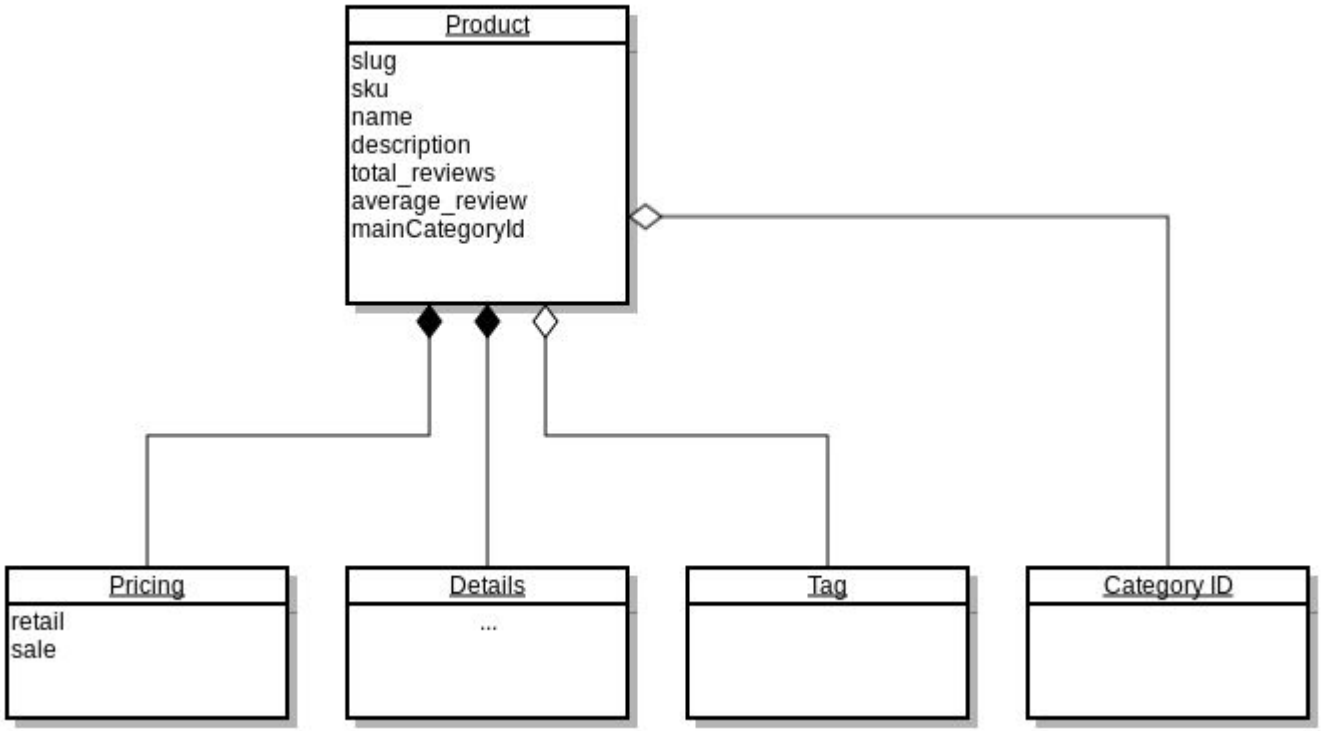


Split into smaller pieces



http://mbartyzel.blogspot.com/2011/11/bounded-context.html

Build aggregates

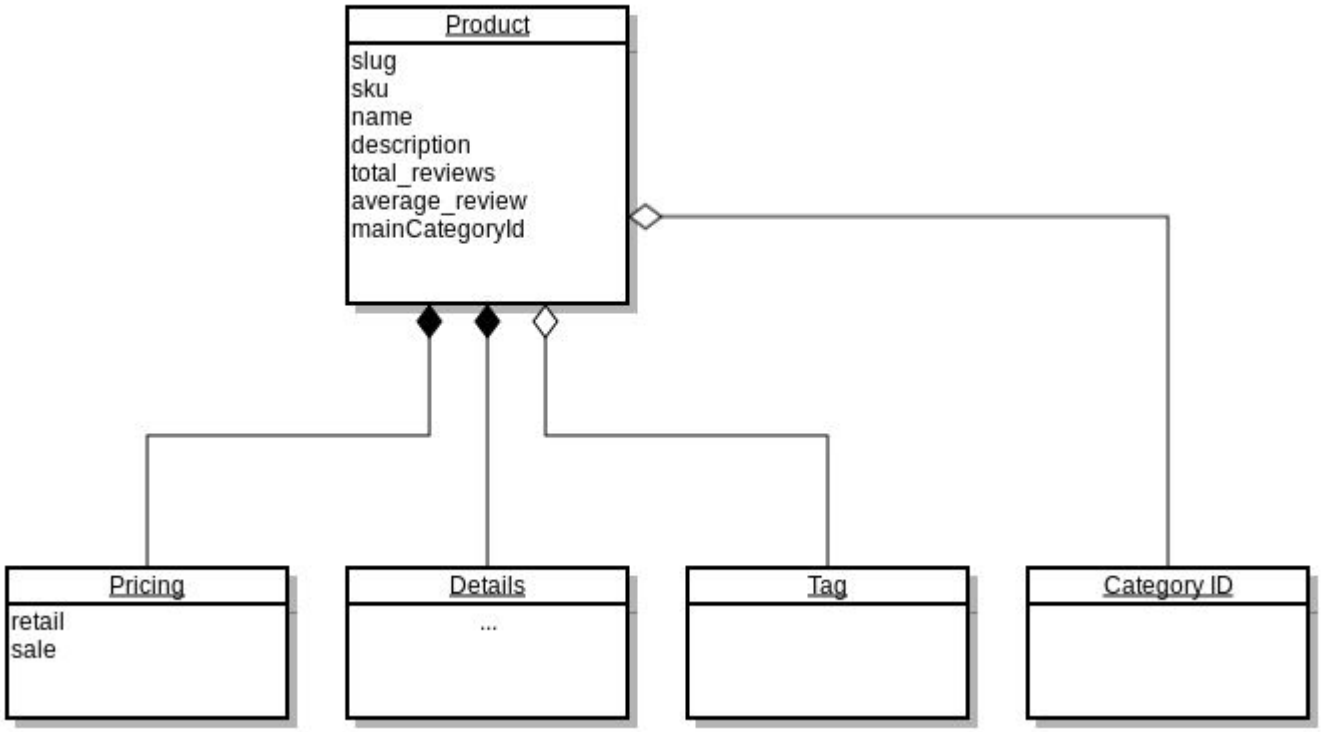


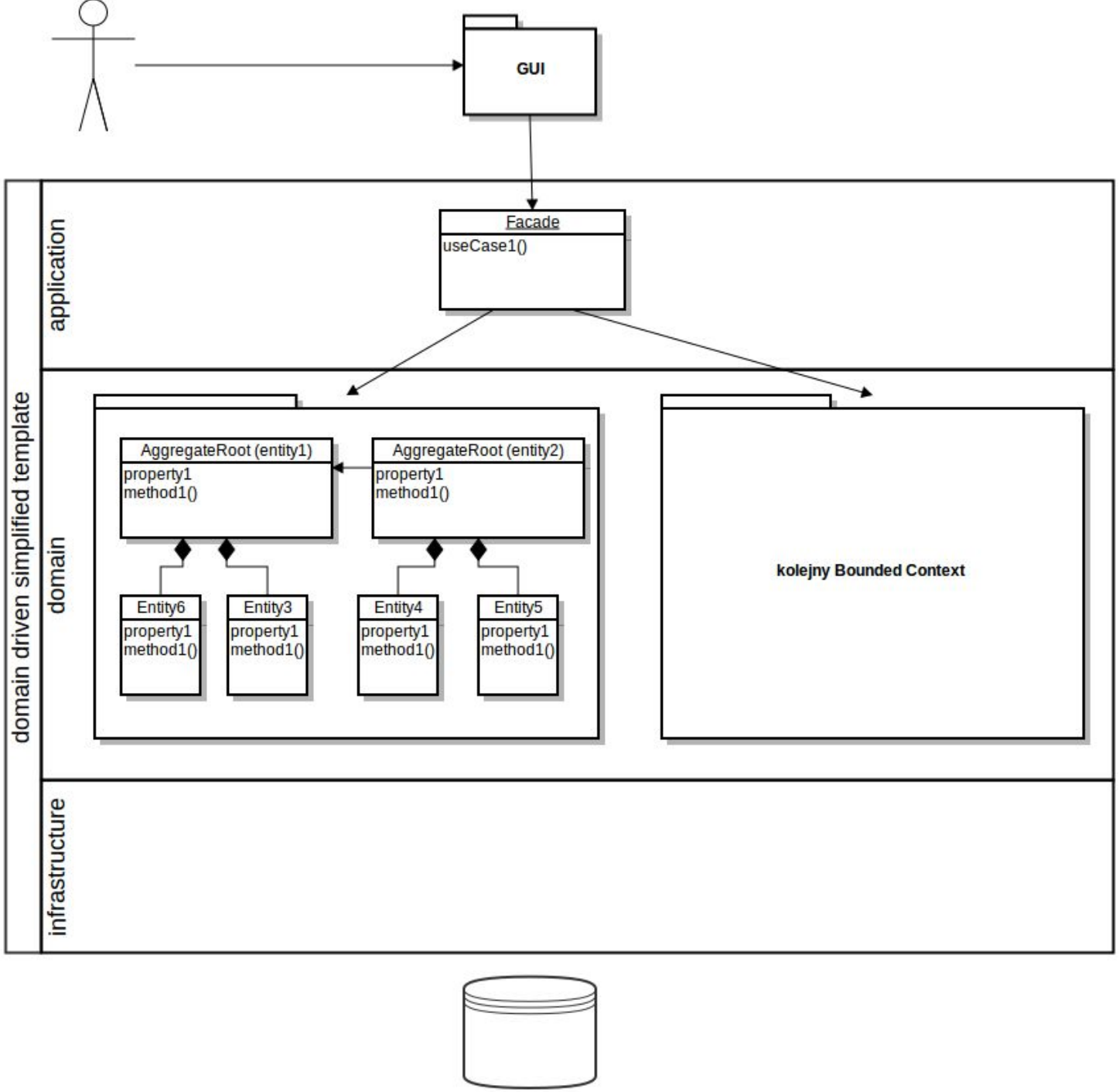
Navigate downwards only

```
@Entity
@Table(name="PRICINGS")
class Pricing {
   @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER)
   @JoinColumn(name="PRICING_ID", referencedColumnName="ID")
   private Set<PricingColumn> columns
}
```

Do not define the relation from the other side.

Create repository for aggregate root only




GUI
Facade
useCase1()
application
domain driven simplified template
domain
AggregateRoot (entity1)
property1
method1()
AggregateRoot (entity2)
property1
method1()
Entity6
property1
method1()
Entity3
property1
method1()
Entity4
property1
method1()
Entity5
property1
method1()
kolejny Bounded Context
infrastructure

Bounded Context

Explicitly define the context within which a model applies.

Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas.

Keep the model strictly consistent within these bounds, but don’t be distracted or confused by issues outside.

Bounded Context

Explicitly define the context within which a model applies.

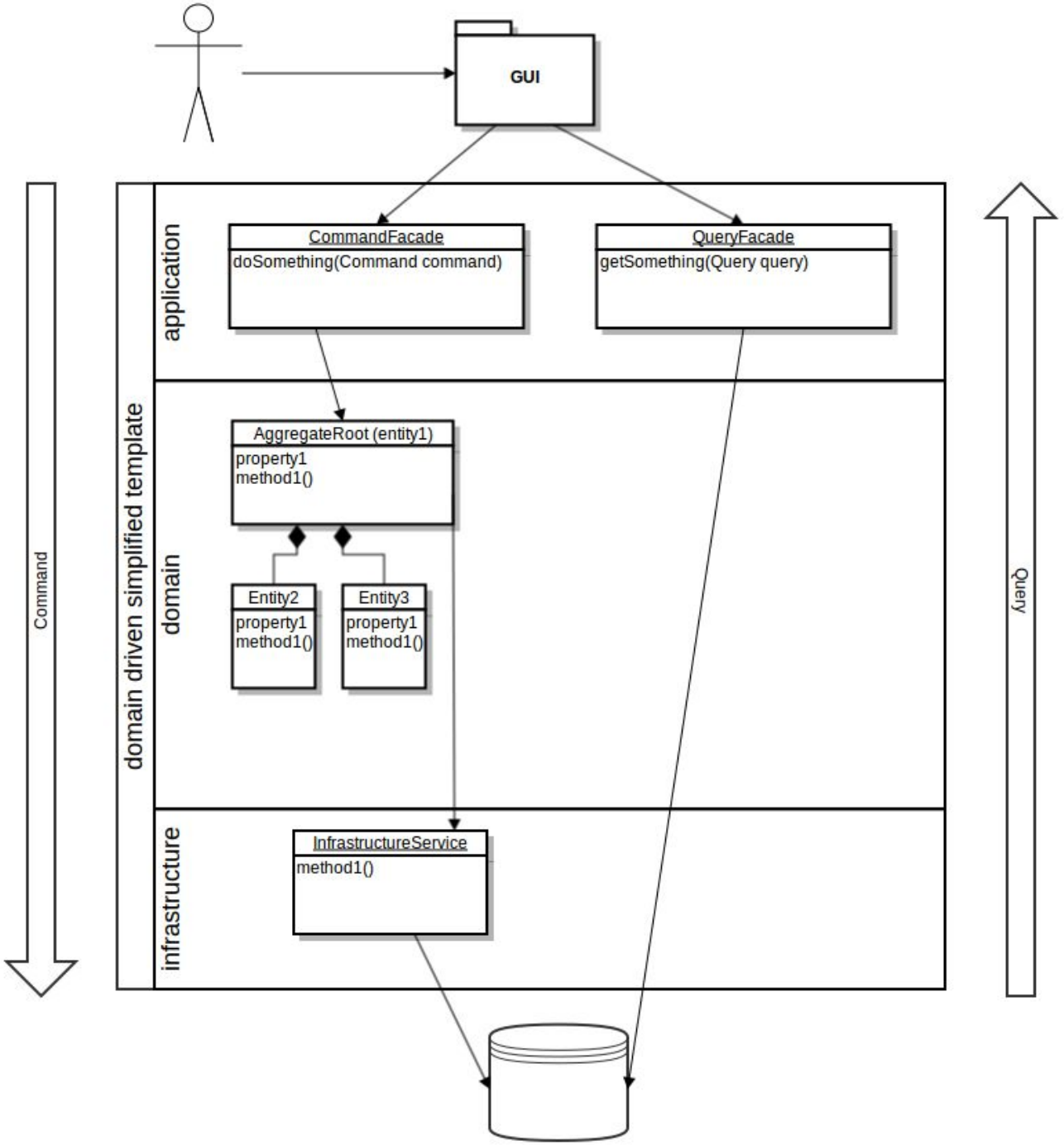
Explicitly set boundaries in terms of team organization, usage within specific parts of the application, and physical manifestations such as code bases and database schemas.

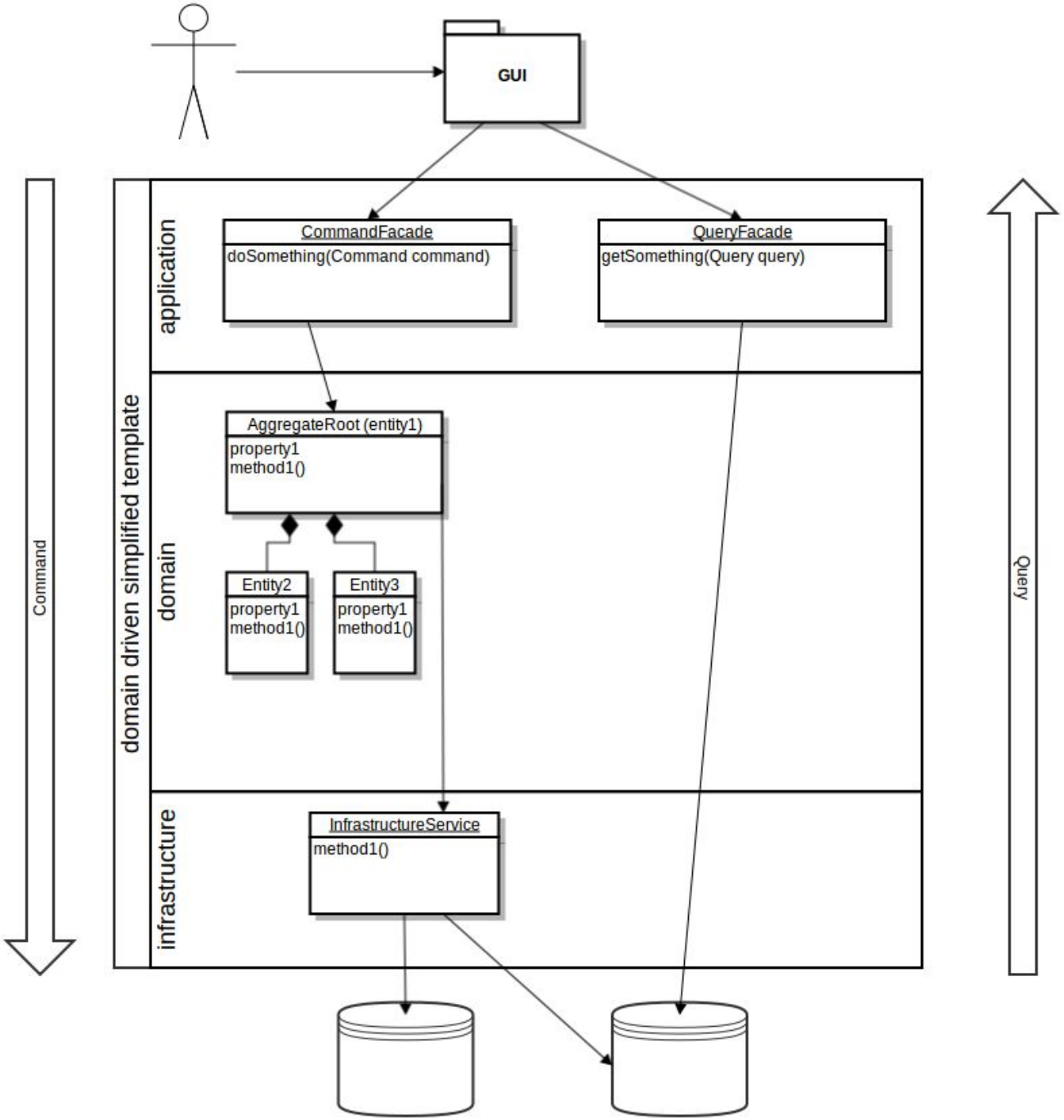
Keep the model strictly consistent within these bounds, but don't be distracted or confused by issues outside.

DOES SCALE

Command Query Responsibility Segregation

because it’s easy to read a book and hard to write it


GUI
domain driven simplified template
application
CommandFacade
doSomething(Command command)
QueryFacade
getSomething(Query query)
domain
AggregateRoot (entity1)
property1
method1()
Entity2
property1
method1()
Entity3
property1
method1()
infrastructure
InfrastructureService
method1()
Command
Query


GUI
Command
domain driven simplified template
application
CommandFacade
doSomething(Command command)
QueryFacade
getSomething(Query query)
domain
AggregateRoot (entity1)
property1
method1()
Entity2
property1
method1()
Entity3
property1
method1()
infrastructure
InfrastructureService
method1()
Query