Domain design principle

For easy and performance-optimized usage

Myself

- Pertti Erkkilä
- Senior Software Engineer
- pertti.erkkila@sysart.fi

- At Sysart 4 years
- Specialized to performance issues
  - Design
  - Tuning
  - Optimization

Introduction

- Easy to use
  - Different skill-levels
  - Rapid learning curve
- Strong type-safety
- Queries are defined from domain
  - No query-language
  - Not bound with implementation
  - Reduces significantly required code under the hood
- Efficient
  - automatically optimized queries

Overall description 1/2

- » DomainContext
  - » Entry point to use domain
  - » Creating and saving domain entities
  - » Creating and executing queries
- » Entity
  - » Base interface for application domain interfaces
- » EntityProperty
  - » Interface for property-based data handling
- » Query
  - » Base interface for application domain queries
- » QueryProperty
  - » Interface for property-based query building

Overall description 2/2

- DomainService
  - Server-side interface
  - Offers same operations to entities
  - Queries are converted to instructions
- Projection
- Restriction

- Why to split DomainContext and DomainService?
  - Different client types
  - Different server implementations
  - Caching

Example domain

- Customer
  - Name (String)
  - Age (Integer)
  - 1-1 Address
  - 1-n Order
- Address
  - Street (String)
  - Zip Code (Integer)
  - 1-1 Customer
- Order
  - Date (Date)
  - n-1 Customer

Creating new domain entity 1/2

```
Customer customer = domainContext.create(Customer.class);
customer.name().set(“Happy Customer");
customer.age().set(55);

domainContext.save(customer);
```

Creating new domain entity 2/2

```
Customer customer = domainContext.create(Customer.class);
customer.name().set(“Happy Customer");
customer.age().set(55);

Address address = domainContext.create(Address.class);
address.street().set(“Street 123");
address.zipCode().set(888);
address.customer().set(customer);
// or: customer.address().set(address);

domainContext.save(customer);
```

Fetching domain data

```
CustomerQuery query = domainContext.create(CustomerQuery.class);
query.id().eq(customer.id());
query.name().fetch();
query.age().fetch();
query.address().street().fetch();
query.address().zipCode().fetch();
Set<Customer> result = domainContext.execute(query);
```

Filtering domain data

```
CustomerQuery query = domainContext.create(CustomerQuery.class);
query.name().like(“Happy %").fetch();
query.address().zipCode().greaterOrEqual(888);
query.orders().date().eq(new Date());
Set<Customer> result = domainContext.execute(query);
```

Reading domain data

```
String name = customer.name().get();

Address address = customer.address().get();
Integer zipCode = customer.address().get().zipCode().get();

Iterator<Order> iterator = customer().orders().iterator();
```

Hibernate versus our domain

Customer report with name, zip code and order count

» Hibernate

```
List<Customer> result = session.createCriteria(Customer.class).list();
```

» Our domain

```
CustomerQuery query = domainContext.create(CustomerQuery.class);
query.name().fetch();
query.address().zipCode().fetch();
query.orders().id().fetch();
Set<Customer> result = domainContext.execute(query);
```

Extra bits

- EntityKey
  - Type-safe: EntityKey<Customer>
  - Allows storing version information internally
  - Allows reference to exact version: EntityInstance
- Separates What? and How?
  - Even beginners can produce efficient queries
  - Allows further optimizations
- Server implementation
  - Different implementations without changes to application code
  - Memory, file, SQL, Google BigTable, Amazon SimpleDB, …

And the end…

- Questions?
- Comments?

Thank you!