Concept/Topic: *Tactic DDD*

Context

Architectural patterns are software patterns that offer well-established solutions to architectural problems in software engineering. They express the fundamental structural organization schema for a software system and define subsystems and their responsibilities and interrelations. Compared to design patterns, architectural patterns are larger in scale.

The PoEAA book (Fowler (2002)) only has one pattern called Domain Model to represent an object-oriented BLL; the *Domain-Driven Design (DDD)* (Evans (2003)) book steps in and completes the coverage of this topic. DDD comes in two parts, *tactic DDD* (this fact sheet) and *strategic DDD* (separate fact sheet).

DDD decomposes the Domain Model Pattern from PoEAA book. It classifies and stereotypes certain types of object/class roles, and constrains their properties with the objective to improve runtime qualities such as scalability and build time qualities such as preserving data integrity across versions and releases and subsystems (e.g., by placing runtime transaction boundaries and security checks only at certain well defined points in the code). DDD emphasizes need for modeling and a common vocabulary/glossary (called Ubiquitous Language). It also adds a strategic design and integration dimension to function-level partitioning (notion of contexts with boundaries).

Definition(s)

The following patterns were introduced in the lecture:

- *Ubiquitous Language*: "A language structured around the domain model and used by all team members to connect all the activities of the team with the software." (source: DDD glossary[1])
- *Entity*: see figure "Core DDD Patterns (Part 1)"
- *Value Object*: see figure "Core DDD Patterns (Part 1)"
- *Service*: see figure "Core DDD Patterns (Part 1)"
- *Aggregate*: see figure "Core DDD Patterns (Part 2)"
- *Repository*: see figure "Core DDD Patterns (Part 3)"
- *Factory*: see figure "Core DDD Patterns (Part 3)"

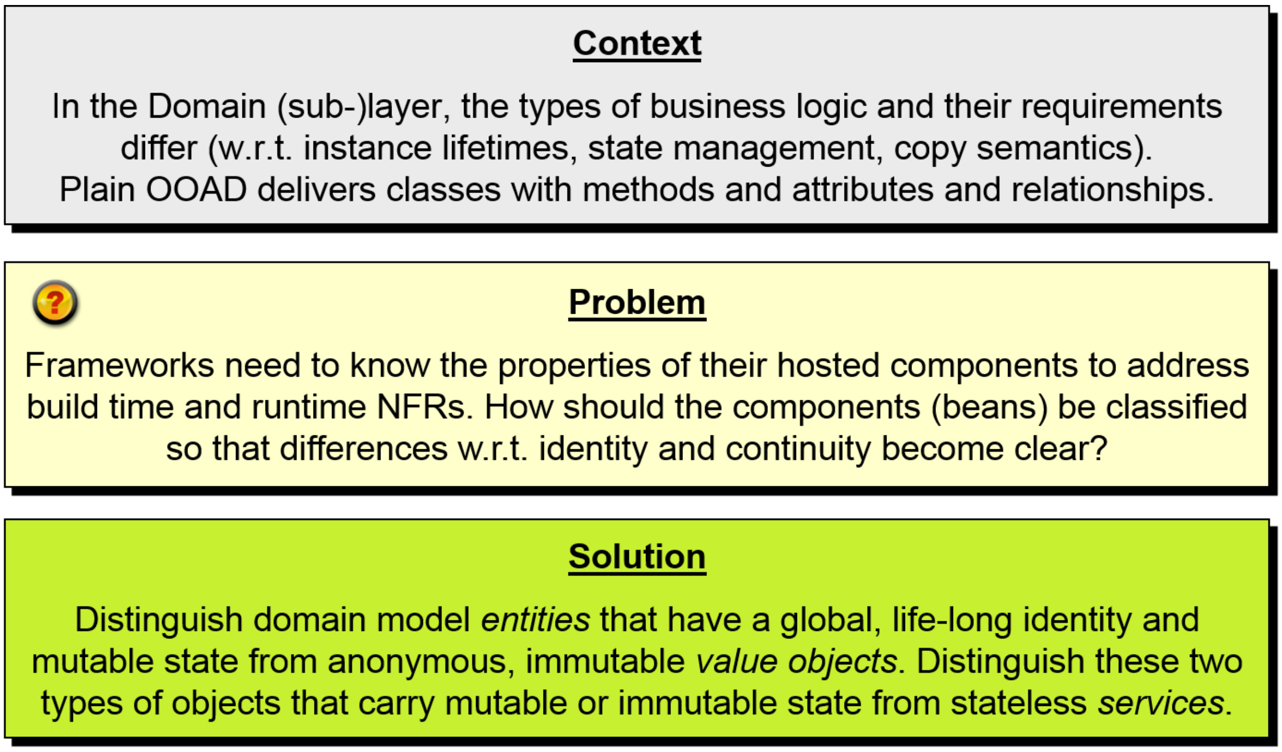
Additional patterns were briefly mentioned only: Domain Event, Module, Layered Architecture, and Intention Revealing Interface.

Aggregate pattern

This pattern is hard to grasp; in the sample application it is hardly visible as packages are used to represent them. Its defining key property is that is enforces invariants and is responsible for conceptual integrity of all contained entities by establishing a *consistency boundary*. Is has one defined entry point called *root entity* (or *aggregate root*).

The name Aggregate is rather generic. It gets the clustering/grouping aspect of the pattern across, but fails to communicate the equally important aspect of invariant preservation and information hiding (a best practice for aggregate design is that no object references but only entity ids and value objects can be sent to the outside). A

[1]http://dddcommunity.org/resources/ddd_terms/



Context

In the Domain (sub-)layer, the types of business logic and their requirements differ (w.r.t. instance lifetimes, state management, copy semantics).
Plain OOAD delivers classes with methods and attributes and relationships.

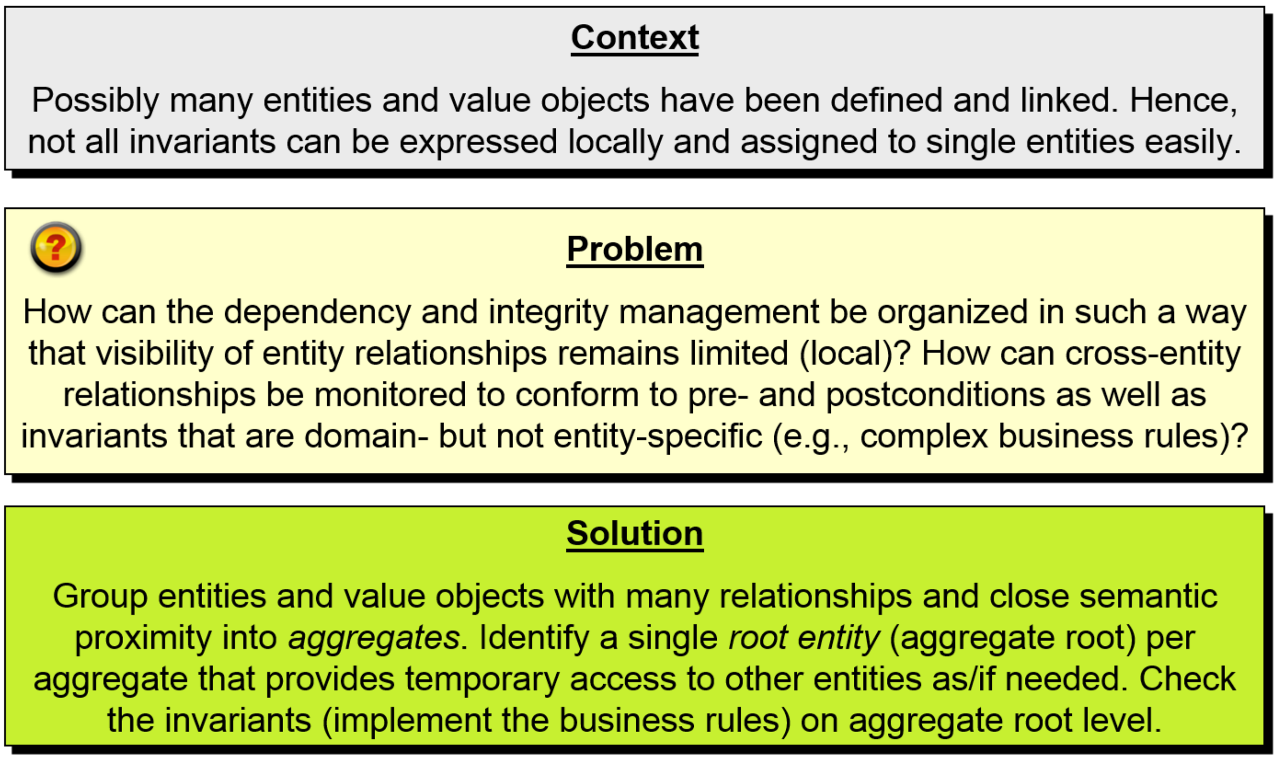
Problem

Frameworks need to know the properties of their hosted components to address build time and runtime NFRs. How should the components (beans) be classified so that differences w.r.t. identity and continuity become clear?

Solution

Distinguish domain model *entities* that have a global, life-long identity and mutable state from anonymous, immutable *value objects*. Distinguish these two types of objects that carry mutable or immutable state from stateless *services*.

Figure 1: *Core DDD Patterns (Part 1)*



Context

Possibly many entities and value objects have been defined and linked. Hence, not all invariants can be expressed locally and assigned to single entities easily.

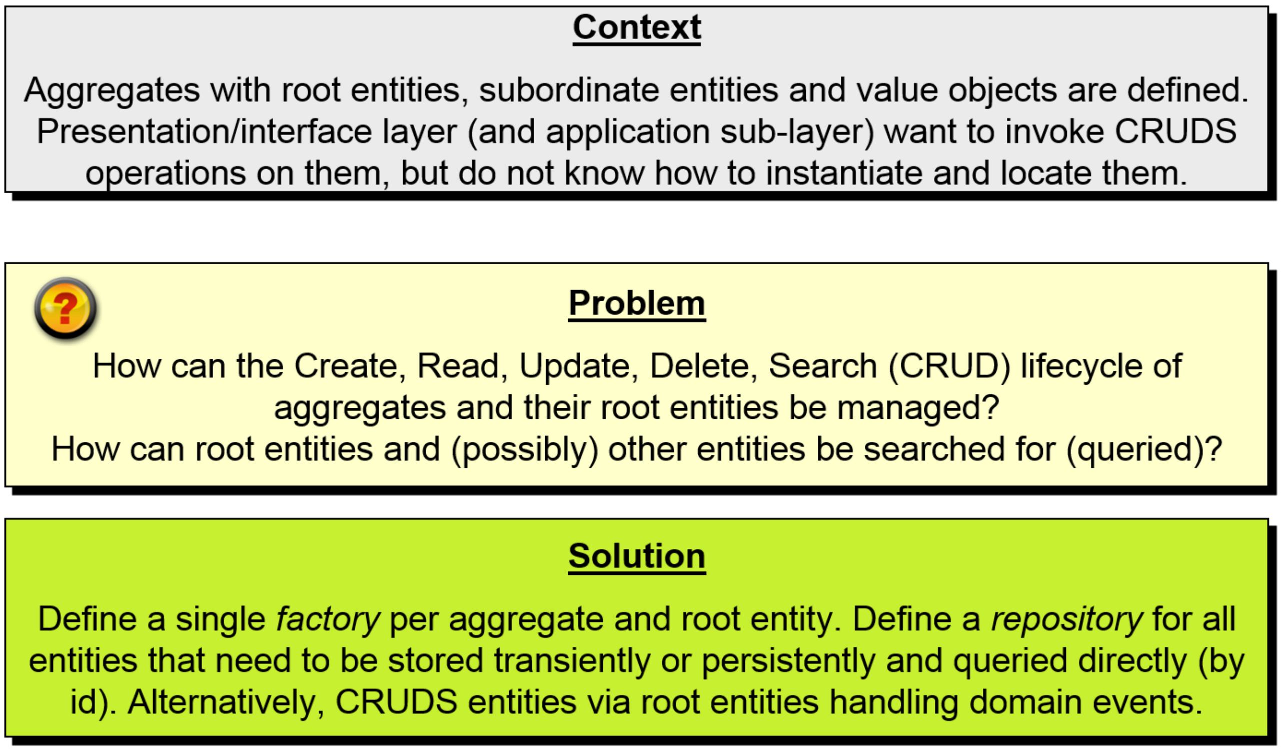
Problem

How can the dependency and integrity management be organized in such a way that visibility of entity relationships remains limited (local)? How can cross-entity relationships be monitored to conform to pre- and postconditions as well as invariants that are domain- but not entity-specific (e.g., complex business rules)?

Solution

Group entities and value objects with many relationships and close semantic proximity into *aggregates*. Identify a single *root entity* (aggregate root) per aggregate that provides temporary access to other entities as/if needed. Check the invariants (implement the business rules) on aggregate root level.

Figure 2: *Core DDD Patterns (Part 2)*



Context

Aggregates with root entities, subordinate entities and value objects are defined. Presentation/interface layer (and application sub-layer) want to invoke CRUDS operations on them, but do not know how to instantiate and locate them.

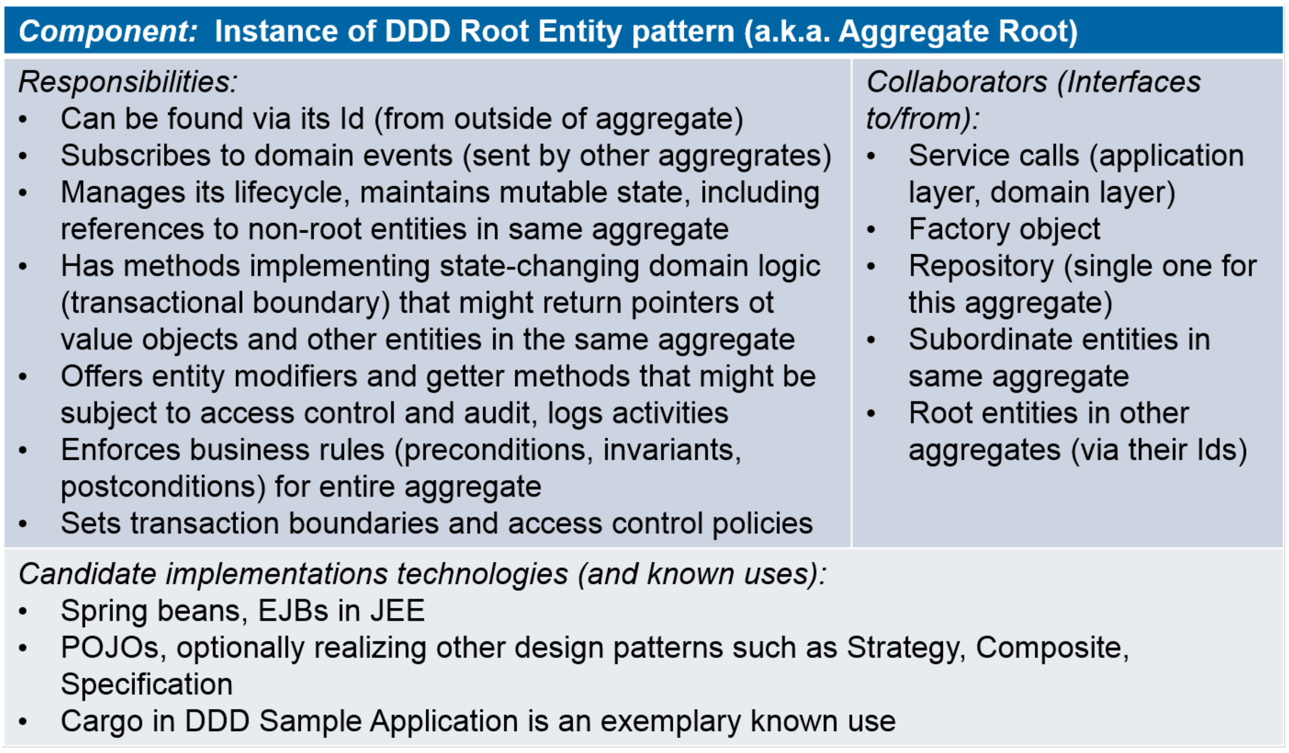
Problem

How can the Create, Read, Update, Delete, Search (CRUD) lifecycle of aggregates and their root entities be managed?
How can root entities and (possibly) other entities be searched for (queried)?

Solution

Define a single *factory* per aggregate and root entity. Define a *repository* for all entities that need to be stored transiently or persistently and queried directly (by id). Alternatively, CRUDS entities via root entities handling domain events.

Figure 3: *Core DDD Patterns (Part 3)*



***Component:* Instance of DDD Root Entity pattern (a.k.a. Aggregate Root)**

Responsibilities:	*Collaborators (Interfaces to/from):*
• Can be found via its Id (from outside of aggregate) • Subscribes to domain events (sent by other aggregrates) • Manages its lifecycle, maintains mutable state, including references to non-root entities in same aggregate • Has methods implementing state-changing domain logic (transactional boundary) that might return pointers ot value objects and other entities in the same aggregate • Offers entity modifiers and getter methods that might be subject to access control and audit, logs activities • Enforces business rules (preconditions, invariants, postconditions) for entire aggregate • Sets transaction boundaries and access control policies	• Service calls (application layer, domain layer) • Factory object • Repository (single one for this aggregate) • Subordinate entities in same aggregate • Root entities in other aggregates (via their Ids)

Candidate implementations technologies (and known uses):
- Spring beans, EJBs in JEE
- POJOs, optionally realizing other design patterns such as Strategy, Composite, Specification
- Cargo in DDD Sample Application is an exemplary known use

Figure 4: *CRC Card for Root Entity a.k.a. Aggregate Root*

longer name such as “ConsistencyPreservingAggregate” or “InvariantsEnforcingEntityCluster” would be more intention revealing (but also take longer to type and comprehend).

In “Implementing DDD”, V. Vaughn establishes the following rules for Aggregate design (Vaughn (2013)):

- Model true invariants in consistency boundaries.
- Design small aggregates.
- Reference other aggregates by identity.
- Use eventual consistency outside the boundary.

These nuggets of advice can also be found online in an article series at domaindrivendesign.org[2].

A common convention is to have one subpackage in the domain model package per aggregate. V. Vaughn also recommends a separate IDE project per bounded context (one of the strategic DDD pattern we will cover later); this goes beyond the advice in Evans (2003) and the corresponding Cargo sample application.

DDD and Layers

DDD suggests its own layering scheme (with four layers), and has its own pattern Layered Architecture:

- *Interface Layer* (or User Interface Layer), which corresponds to the Presentation Layer in our previous basic layering scheme
- *Application Layer*, which can be seen as a refinement or sub-layer of our BLL
- *Domain Layer*, a second refinement or sub-layer of the BLL
- *Infrastructure Layer*, which includes but is not limited to Data Access Layer responsibilities

The patterns mostly “live” in the domain model layer (with a few exceptions, for instance services may also appear in the application layer and in the infrastructure layer). As DDD repositories reside in the Domain Layer per default, they can also serve as main memory lookup interfaces only (but also use services provided by the infrastructure layer to store entities persistently). The data loading/unloading discussion is an important, but separate one. The DDD repository pattern does not assume or imply any particular prefetching or lazy loading strategy; you are free (and have) to choose one as one of your architectural decisions on any project.

The figure “Core DDD Patterns and their Relationships” summarizes the key properties of the patterns and their relationships.

Other DDD Patterns (Selection, Brief Coverage)

Subdomains represent a logical partition in/of a complex model. The Core Domain can be distinguished from Supporting Subdomains and Generic Subdomains. The Core Domain has high priority (as it is a differentiator), Generic Subdomains are less important.

Modules (a.k.a. packages) simply group related concepts/abstractions. They should be named expressively and should promote loose coupling.

A Domain Event represents “something that happened in the domain”: discrete, full-fledged domain objects that users want to keep track of/be notified of” (e.g., state change) according to Evans (2003) and Vaughn (2013). Domain Events are not featured in the original book (Evans (2003)) under this name), but in later ones such as Vaughn (2013). They can support an overarching architectural pattern Command Query Responsibility Segregation (not covered in the lecture yet).

[2]http://dddcommunity.org/library/vernon_2011/

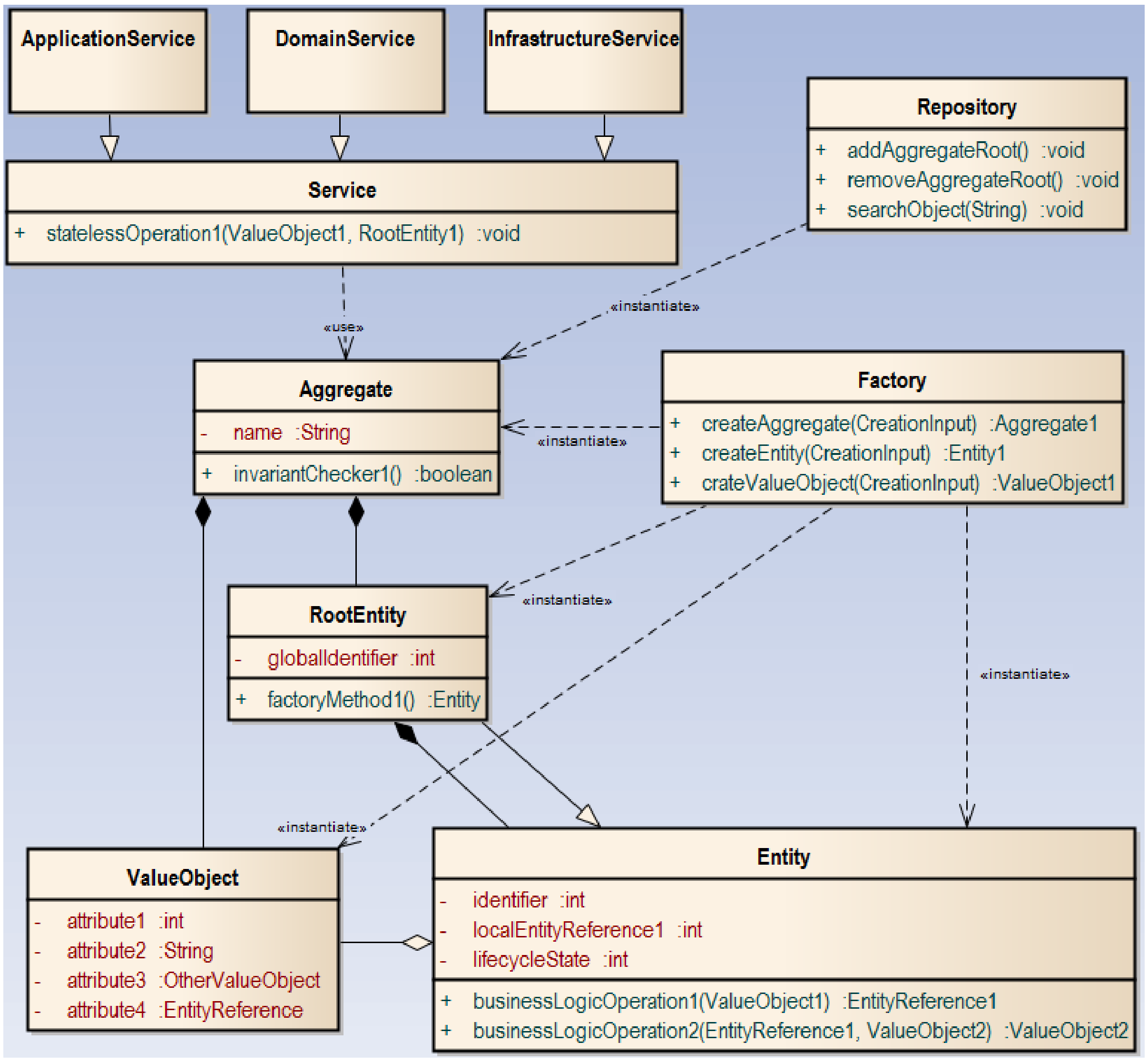

ApplicationService
DomainService
InfrastructureService
Repository
+ addAggregateRoot() :void
+ removeAggregateRoot() :void
+ searchObject(String) :void
Service
+ statelessOperation1(ValueObject1, RootEntity1) :void
«use»
«instantiate»
Aggregate
- name :String
+ invariantChecker1() :boolean
Factory
+ createAggregate(CreationInput) :Aggregate1
+ createEntity(CreationInput) :Entity1
+ crateValueObject(CreationInput) :ValueObject1
«instantiate»
RootEntity
- globalIdentifier :int
+ factoryMethod1() :Entity
«instantiate»
«instantiate»
«instantiate»
ValueObject
- attribute1 :int
- attribute2 :String
- attribute3 :OtherValueObject
- attribute4 :EntityReference
Entity
- identifier :int
- localEntityReference1 :int
- lifecycleState :int
+ businessLogicOperation1(ValueObject1) :EntityReference1
+ businessLogicOperation2(EntityReference1, ValueObject2) :ValueObject2

Figure 5: *Core DDD Patterns and their Relationships*

Example(s)

A DDD Sample Application[3] is available on GitHub, dealing with cargo booking, routing and tracking. It implements the running example in Evans (2003).

The agile project management sample[4] accompanying Vaughn (2013) can be found on GitHub as well.

More Information

Related Topics and Concepts

Strategic DDD (lesson 8)

Miscellaneous Links:

We refer the reader to the following information sources (no need to copy this material here): InfoQ eBook (free, but registration required)[5].

There is an IFS website on DDD[6] as well that has additional pointers.

References

Evans, Eric. 2003. *Domain-Driven Design: Tacking Complexity in the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Fowler, Martin. 2002. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Vaughn, Vernon. 2013. *Implementing Domain-Driven Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

[3] https://github.com/citerus/dddsample-core
[4] https://github.com/VaughnVernon/IDDD_Samples
[5] https://www.infoq.com/minibooks/domain-driven-design-quickly
[6] https://www.ifs.hsr.ch/index.php?id=15666&L=4