Domain-Driven Design

Klaus Kusche, May 2013

Contents

DDD history & sources

- *"The book"*:

  Eric Evans (Addison-Wesley, **2003**):
  "Domain-Driven Design:
  Tackling Complexity in the Heart of Software"

- *Compact summary* of "the book":

  Abel Avram, Floyd Marinescu (InfoQ, 2006):
  "Domain-Driven Design Quickly"

  Online / freely downloadable PDF!

- `http://dddcommunity.org`

DDD is ...

“Domain-driven design is
not a technology or a methodology.

It is a way of thinking and a set of priorities,
aimed at accelerating *software projects*
that have to deal with *complicated domains*.”

http://dddcommunity.org/learning-ddd/what_is_ddd/

DDD includes …

- … principles to

  <u>design the code</u>

  (*<u>technically</u>*)

- … principles to

  <u>manage the development</u>

  (*<u>organizationally</u>*)

SW structure defined by DDD

Clearly separated layers:

- **Presentation layer** = User interface
- **Application layer**
  = Coordination, client session management, …

  *No business data or business logic! ==> **Thin!***
- **Domain layer** = Business data and logic

  *DDD is **for this layer only!***
- **Infrastructure layer**
  = Communication (Network), Persistence (DB), …

Application areas for DDD

- DDD is *best suited* for software projects with

  complex business logic or workflow

- DDD is *not suited*
  - … for **data-centric projects** with little logic
  - … for designing and describing **user interfaces**
- DDD *does not care*
  - … about data **persistency** (i.e. databases) and **I/O**
    (that's "hidden" in repository classes, see later)
  - … about **infrastructure** (e.g. networking, …)
  - … about the **user interface**

What does “***Domain***” mean?

“Domain” in DDD: *Not* its technical meaning!

“Domain”

=

Business */ Activity / Knowledge*
of the ***user*** */ customer*

(German: *“Anwendungsbereich”,*
“Fachgebiet”, “Geschäftsfeld”)

DDD goals (1)

Common <u>observation</u>:

If the problem isn't understood,
the solution won't make users happy.

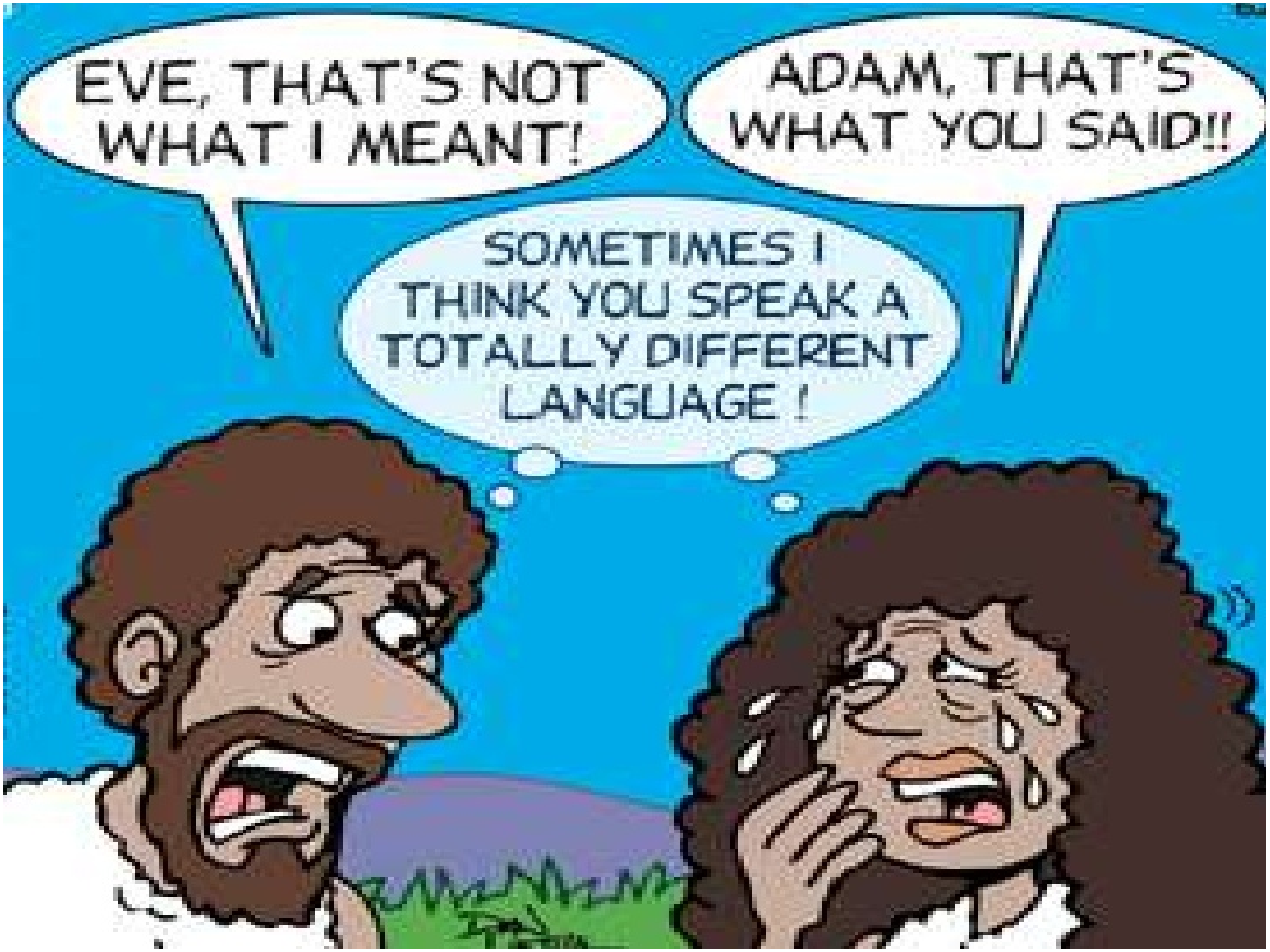
<u>Remedy</u>:

<u>Avoid the user / developer gap</u>
in understanding what
the software is expected to do!

==> Meet the *<u>customer's needs & expectations!</u>*

“I know you believe
that you understand
what you think I said,

but I am not sure
if you realize
that what you heard
is not what I meant …”


EVE, THAT'S NOT WHAT I MEANT!
ADAM, THAT'S WHAT YOU SAID!!
SOMETIMES I THINK YOU SPEAK A TOTALLY DIFFERENT LANGUAGE !

DDD's central concept

User's **Knowledge & Problem**

--> **Language** to express it

--> **Model** representing it

--> **Implementation**

The **main effort** in DDD goes into

understanding and ***modelling***
(by those who actually implement it!)
what the user is doing.

The language (1)

Define a common, "**ubiquitous language**"
understood by *both* sides
==> *Business (user) terms, not technical terms!*

Write it down ("**Glossary**" of terms used)
and *strictly adhere to it*:

- In all ***discussions*** and communications
- In all ***documents***
- In the ***code*** (class names, ...)

The language (2)

Double-check for each language term defined:

- **Domain expert:**
  - Do I *understand* it?
  - Does its definition *say what I think*?
  - Can I *clearly express* my problem with it?
- **Developer:**
  - Is it *unambiguous* / consistent / well-defined / complete / ... ?
  - Can I write code for it?

The model (1)

Reality =
Objects / Values
& their ***Actions / Behaviour / Operations***

Model =
Description / Abstraction of the reality

The model is for a specific **purpose**: *To solve a problem!*

==> *Drop* irrelevant or unimportant things

==> Describe *relevant things exactly* & in detail

The model (2)

Differing from “classic” approaches,

the model is not “internal” to the development, but created in collaboration with the customer!

==> The model must be

readable & understandable for the customer!

Avoid:

- *Technical terms & concepts*: Reality isn’t talking SQL!
- Anything related to the *user interface*:
  - Don’t describe data or actions based on their UI
  - DDD isn’t for defining UI’s

The model (3)

What does the model **represent**?

Roughly

- *Object-Relation-Diagrams* with *methods*
- similar to *UML*

But:

Use *any format* which is *easily understood*:

- Plain text, hypertext, ...
- Free-hand drawings, UML, other diagrams
- Even documented code (e.g. Javadoc) is ok!

DDD & SW project management

DDD by itself is

not a software project management methodology

but it *requires* some

***agile** software development process*

It *goes well* with
Scrum, Extreme Programming, …

It *won't work* with
Waterfall or spiral model, german “V-Modell”, …

The “*agile manifesto*”

“… we have come to value:

- *Individuals and interactions*
  over processes and tools
- *Working software*
  over comprehensive documentation
- *Customer collaboration*
  over contract negotiation
- *Responding to change*
  over following a plan”

==> Interact and iterate!

Agile principles in DDD (1)

Interaction:

- *Direct and frequent discussion*
- … during the *whole project lifetime*
- … between the ***"domain experts"***
  of the customer
  (not the managers / lawyers, not the average users)
- … and *all developers*!

==> If you have *no direct access* to the domain experts,

DDD is *not the way to go*!!!

Agile principles in DDD (2)

Iteration:

Language and model ***evolve** during implementation*:

- *Unclear*, things *missing*?
- Hard to implement, too *slow*, ...?
- Good ideas for *restructuring*?

==> *Immediately rediscuss* with the domain experts!

==> *Extend or adapt language, model & code!*

At any time, ***language, model & code***
must ***match each other exactly!***

==> Continuously update them synchonously!

Agile principles in DDD (3)

Continuous refinement & refactoring requires

continuous integration

of all developments:

- *Merge daily, build daily, test daily!*
- *Automatic unit tests*
  are highly recommended!

Agile principles in DDD (4)

<u>Others:</u>

- There are just “developers”,
  <u>no dedicated</u> “analysts”, “designers” or “architects”:

  The *<u>developer</u>* must understand the user’s needs!

- There are *<u>no phases</u>*
  (like specification, design, implementation, test, ...)

  ==> Implement and test *<u>early</u>*!

- There are *<u>no formal requirements</u>*,
  no required documents, no milestones, ...
  (no “Lastenheft” or “Pflichtenheft”, just the model)

DDD goals (2)

"Master the complexity"

- Make large (business) software projects with complex business logic *manageable*
- Produce *correct, understandable & maintainable code* within time & budget
- *Avoid* the ***"big ball of mud"!***

DDD is based on …

- **<u>Object-oriented principles</u>**

  ==> *<u>Language independent</u>*,

  but suitable *<u>only for O-O languages</u>*
  (Java, C#, some “web” languages)

  ==> “Plain” Java / C# / … suffices,

  <u>no special framework</u> required

- **<u>Some O-O patterns</u>** of the “Gang of Four”
  (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
  “Design Patterns. Elements of Reusable Object-Oriented Software”)

  ==> *<u>Experienced O-O programmers</u>* required!

Designing code with DDD

Two different, independent levels:

- Designing the *interaction between "Bounded Contexts"*
- Designing the *classes within one "Bounded Context"*

Each ***"Bounded Context"*** corresponds to

one subproject

==> One *separate team*,
separate discussions with domain experts

==> One *separate language & model*

(Example: Parcel distribution:
Logistics, billing / finance, customer care, statistics, ...)

The “big picture”

- The ***“Context map”*** describes
  the *“contact points”* between Bounded Contexts
  and their *interaction / relation*.
- DDD lists *six typical patterns* of interaction:
  Shared kernel, customer / supplier, separate ways,
  conformist, open host service, anticorruption layer
- In most cases, the *innards are hidden:*
  Bounded contexts do *not share objects* directly!

  ==> Each context is a *separate application or process*
  ==> They likely *communicate by messages*
  ==> This leads to a *service-oriented architecture*

Designing classes

Categorize and refactor the classes in the initial model:

- **Entities**: *Objects* with unique *identity*
- **Value Objects**: *Values* without identity
- **Aggregates**: *Combine* Entities and Value Objects
- **Factories**: *Generate* new complex Aggregates
- **Repositories**: *Store* Aggregates persistently
- **Services**: *Functionality* not belonging to objects
- **Modules**: *Structure* the model
- + some **GoF design patterns**: *Specification, strategy, ...*

Entities and Value Objects

Entities (e.g. person, parcel, truck, bank account, ...):

- Have a *unique and persistent identity*
- Have *state* and a well-defined *lifecycle*
- Have *behaviour* (methods)

Value Objects (e.g. color, postal address, ...):

- Only represent *values / properties*
- *Don't* have a unique *identity* nor state or lifecycle
- Are *immutable* (read-only)
  ==> Can be *copied* & destroyed at will

Aggregates

… ***combine Entities and Value Objects***

which belong together

Example:

Parcel + pack list + route + …

Quick check:

If a ***cascading delete*** is required,
the *objects affected* should perhaps
be combined into an aggregate!

The “*aggregate root*”

… is the “*topmost*” entity, *representing* the aggregate

… “*owns*” all other objects in the aggregate

… is the object giving the aggregate its *identity*

… is the *only object* whose reference (identity)
should be *visible & stored outside* the aggregate

… is the *only object* (“single point of access”)
whose methods can be *called directly from the outside*

==> Aggregates

… are *visibility / identity borders* for their subobjects

… *protect* their innards from direct access

Repositories = Object stores

- 1 Repository = Abstract *collection* of all objects of a certain *Aggregate class* (including subobjects)
- (Virtually) "*in memory*": The model assumes *infinite and persistent memory*
- "*Flat*" (no specific data organization or index), but with *powerful search functionalities*
- ***Repositories hide persistence*** *(permanent storage)* ***and search / access mechanisms*** (Database / SQL, Filesystem, ...): Modelled *only by functionality / interface!*
- Typical *operations*: **`add`**, **`remove`**, **`find`**, **`list`**

Services

DDD prefers *“fat” object classes*, not “anemic” ones

==> most “simple” operations should be defined
in Entity / Value Object / Aggregate classes!

Separate Service classes are intended only for

- Operations which *don’t fit well* elsewhere
- Operations which are *highly complex*
- Operations involving *multiple independent objects*

Examples: *“Calculate route”*, *“Move parcel”*

Service classes are *stateless* (have *no data* of their own)

“The end”

Questions ?