Kari korpinen Ohjelmistotuotanto k 2012

Tehtävä 9 referenssi

Domain-Driven Design in an Evolving Architecture

Posted by **Mat Wall and Nik Silver** on Jul 22, 2008

Domain driven design can be most readily applied to stable domains in which the key activity is for developers to capture and model what is in users' heads.This article examines how we used DDD in the context of a two-year programme of work to rethink and rebuild guardian.co.uk.

1. Background to the programme

Guardian.co.uk has a long history of news, comment and features and currently receives over 18 million unique users and 180 million page impressions a month.February 2006 work began on a major programme of work to move it onto a more modern platform, of which the earliest phase debuted in November 2006 when the new-look Travel site launched, continued with a new front page in May 2007, and more followed. Although just a handful of people started on the work in that February the team later peaked at 104.

In essence, the way we thought about our work had progressed beyond what our software could handle. This is why DDD was so valuable to us.

We will look briefly problems for our internal users, and then with problems for our developers.

1.1. Problems for our internal users

Journalism is an old profession with an established training, qualification and career structure, yet it was impossible for new, journalistically-trained editorial staff members to join us and work effectively with our web tools, even within a few months of arrival.

1.2. Problems for developers

For example, one of the CMS concepts was an "artifact", which was sufficiently core that all developers worked with them every day. One of the team once admitted that it was a full nine months before he realised that these "artifacts" were in fact just web pages. Cryptic language and the software that had grow up around it had obscured the real nature of his work.

It had become clear to us that a divergence between how people thought about their work (launches, web pages, RSS feeds) and how it was implemented (caching workflows, "artifacts", fuzzy logic) was having a tangible and costly impact on our effectiveness.

2. Starting out with DDD

2.1. Choosing DDD

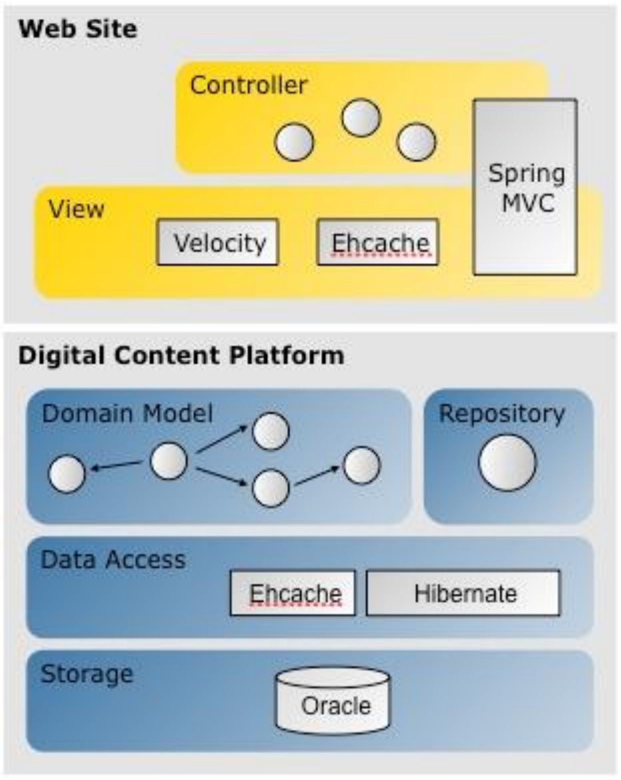
The first aspect of DDD that appealed was that of the ubiquitous language, and embedding users' own concepts directly in the code. This clearly addressed our issue of conceptual mismatches described above.

What took it further was the technical language and concepts that DDD brought with it: entities, value objects, services, repositories, and so on.

2.2. Embedding the domain model into the system

This section shows DDD's place in the overall system's architecture.

Our system is built up of three main components: our user facing website rendering application; our tools application which faces the editors and is used to create and manage content; and our feeds application which routes data in and out of the system. Each of these applications is a Java web application constructed using Spring and Hibernate, with Velocity as our templating language.



The Hibernate layer provides data access.The Model layer contains the domain objects and repositories.Above this we have our Velocity template layer. Finally, the topmost layer contains the controllers that act as entry points into the application.

..as we are using domain driven design we require a ubiquitous language, not only for us to use as people when talking about the domain but also to be used everywhere in the application. The model layer exists not only to isolate business logic from rendering logic but also to provide the vocabulary for other layers to use.

Also, the model layer can be built as a standalone unit of code and exported as a JAR into many applications that are dependent on it. This is not true of any of the other layers. This has an implication for building and releasing applications: Changing code in the model layer in our infrastructure must be a global change across all of our applications. We can change a Velocity template in our front-end website and only have to deploy the front-end application, not the admin application or the feeds. If we change the logic of an object in our domain model we must roll out all of our applications that are dependent on the model as we only have (and want) one view of the domain.

Working in an Agile environment we look to perform full production rollouts of all our applications every two weeks anyway. However, we are constantly looking at the cost of change of the code in this layer. If it begins to rise to an unacceptable level we will probably have to look at splitting the single model into multiple smaller models and providing adaption layers between each sub-model.

2.3. Early domain modelling

Very early on in the project, long before anyone reached for a computer keyboard and started working on code, we had decided that we would co-locate our developers, QAs, BAs and business people in the same room for the duration of the project. At this stage we had a small team of business and technical people, and we required only a modest first release. This ensured our model and our process were both fairly simple.

Our first objective was to get as clear an understanding of what our editors (a key constituent of our business representatives) were expecting for the first few iterations of the project

Our editors decided that their highest priority at this early point in the project was for the system to be able to produce web pages which could display articles and a system of categorising articles.

Their initial requirements can be summarised as follows.

- We should only be able to associate one article with any given URL.
- We would need to be able to change the way that the resulting page was rendered by selecting a different template.
- We need to group our content into broad sections for management, i.e. news, sport, travel
- The system must be able to display a page containing links to all articles in a given categorisation.

Our editors required a very flexible approach to categorisation of articles. The approach they designed was based around the use of keywords. Each article could be associated with many keywords, as an article could be about many topics.

From the language used by the editors it seemed that we were introducing a few key entities into our domain:

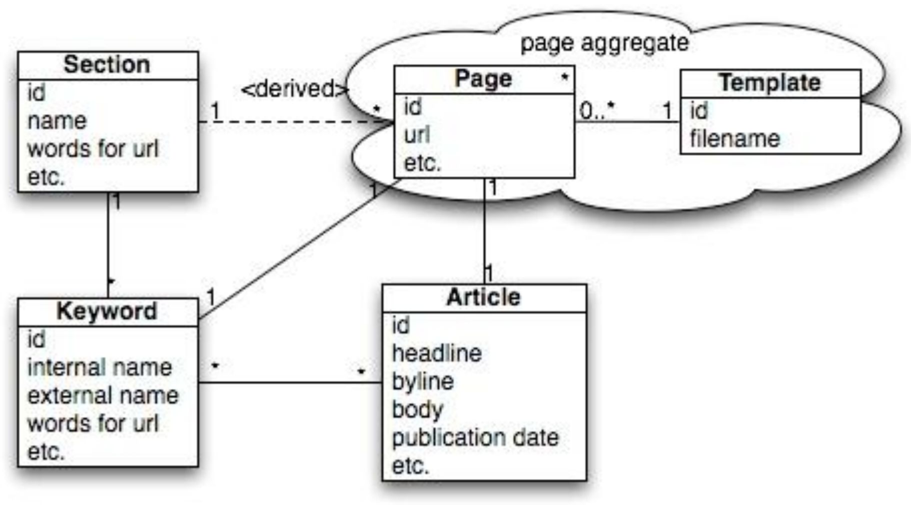
- ***Page*** The owner of the URL. Responsible for selecting a template to render the content.
- ***Template*** A layout for a page, which might be changed at any time. The technical people implemented a template as a Velocity file on the disc.
- ***Section*** A very broad categorisation of pages. Sections each have an editor and a common look and feel for pages within them. Examples of sections are News, Travel, and Business.

- ***Keyword*** A way of describing a subject that exists within a section. Used to categorise articles, keywords drives automatic navigation. They will be associated with a page so that and automatic page of all articles about a given subject can be produced.
- ***Article*** A piece of textual content that we can deliver to a user.

After we had extracted this information we began modelling the domain. A decision that was made early on in the project was that the editors owned the domain model and were responsible, with help from the technical team, for designing it.We found that by having workshops featuring the editors, a few developers and the technical architects we were able to sketch out and evolve the domain model using simple, low-tech approaches. An area of the model would be discussed and candidate solutions drawn, using felt pens, file cards and Blu-Tak.

It was also interesting to watch the domain language evolve. Sometimes the object that was going to become the *Article* was spoken about as a "story". Obviously we cannot have a ubiquitous language that features multiple names for the same entity so this was a problem.

The resulting model that our editors initially designed looked like this:



After a few iterations the system began to take shape and we had build tools to create and manage *Keywords*, *Articles* and *Pages*. The editors were able to use them as they were built and to advise on changes. It was generally felt that this simple core model was working and could go on to form the basis of the first release of the site.

3. Processes for DDD in a growing programme

After the initial release our project team grew with both technical people and business representatives, and we intended the domain model to evolve. It was clear that we needed an organised way to introduce newcomers to domain model and to manage the system's evolution.

3.1. Inductions for new staff

Our induction process includes a DDD session for each of these two audiences, and though the detail varies, the high-level agenda covers the same two areas: what DDD is and why it's important.

Important things we stress when describing DDD are:

- The domain model is very much owned by the business representatives. This is about extracting concepts from the heads of the business representatives and embedding them into the software, not taking ideas from the software and trying to train the business representatives.
- The technical team is a key stakeholder. We will still argue hard around specific details.

We do three things here:

- We draw the concepts and relationships on a whiteboard, and so we can provide a tangible demonstration of how the system-so-far works.
- We make sure an editor is on hand to explain a lot of the domain model, to emphasise the fact that it's not owned by the technical team.
- We explain some of the historical changes we've made to get to this point, so inductees can understand (a) this is not set in stone but is changable, and (b) what kind of role they can play in forthcoming planning conversations to develop the model further.

3.2. DDD in planning

Induction is essential, but that knowledge is only exercised practically when we start to plan each iteration.

3.2.1. Using the ubiquitous language

The common language DDD forces allows business people, technical staff, and designers to meet round the same table to plan and prioritise specific tasks. This means more meetings are relevant for the business people, they get closer to the technical people and understand the technical process more.

There are two important principles we use when using DDD in the planning phase:

1. The domain model is owned by the business; and
2. There needs to an authoritative business source for the domain model.

Business ownership of the domain model is explained during inductions, but only comes into play here. This means that the technical team's key role is to listen and understand, not to explain what is and isn't possible.

3.2.2. Problems planning with DDD

Unfortunately, we have found particular challenges with applying DDD in the planning process, and particularly in an Agile environment where planning is continual. These problems are:

1. The nature of writing software to a new and uncertain business model;
2. Being tied to an old model;
3. Business people going native.

However the principles of Agile development did:

- Build the simplest thing. Though we couldn't settle on all details early on we usually understood enough to build the next piece of useful functionality.
- Release frequently. By releasing this functionality we were able to see how it worked in the field. Further tweaks and evolutionary steps became most apparent from this (and inevitably they were not always what we expected).
- Minimise the cost of change. With these tweaks and evolutionary steps inevitable it was essential to reduce the cost of change. For us this included automated build process, automated testing, and so on.
- Refactor often. After several steps of evolution we would see technical debt accumulate and this needed to be addressed.

4. Evolving the domain model

4.1. Evolutionary step 1: Beyond articles

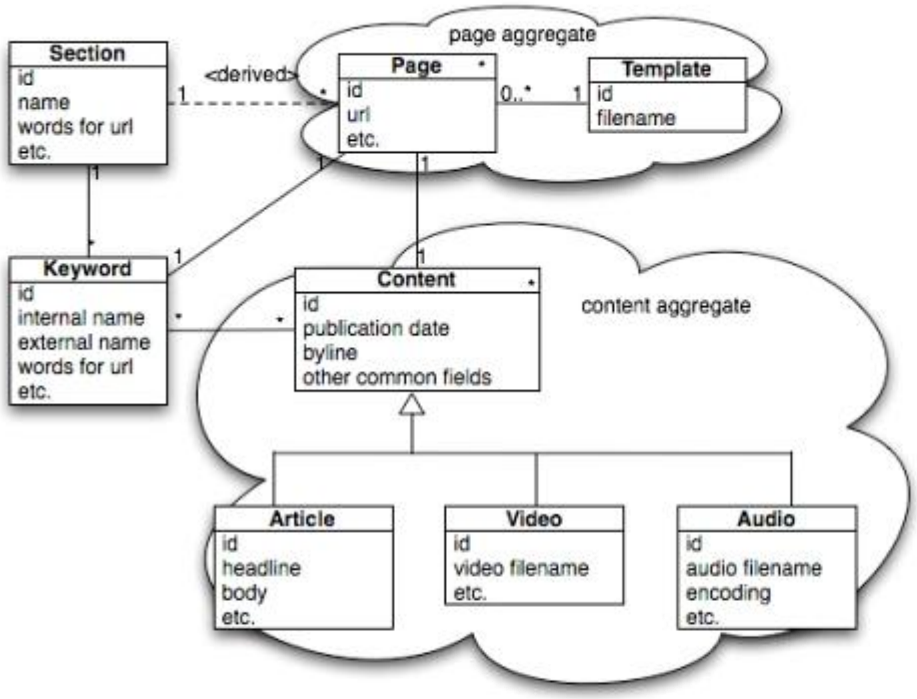
Shortly after the initial release the editors required the system to be capable of dealing with more than one type of content, beyond an *Article*.

This is a key point: Rather than try to architect the whole system up front we focussed on the whole team gaining a good understanding of the model and the modelling process in small manageable chunks.

The new types of content that our editors required in the next iterations were *Audio* and *Video*. Our technical team sat with our editors and went through the domain modelling process again. From talking to our editors it was clear that *Audio* and *Video* were similar to *Articles*: it should be possible to place a *Video* or *Audio* on a *Page*. Only one piece of content was allowed per page. *Video* and *Audio* could be categorised by *Keywords*. *Keywords* could belong in *Sections*

It was clear that *Audio, Video* and *Article* all had something in common: they all were types of *Content*. Our editors were not familiar with the concept of inheritance so the technical team were able to teach the editors about inheritance so that they could correctly express the model as they saw it.

Here is the new model that our editors designed with the new content types added.

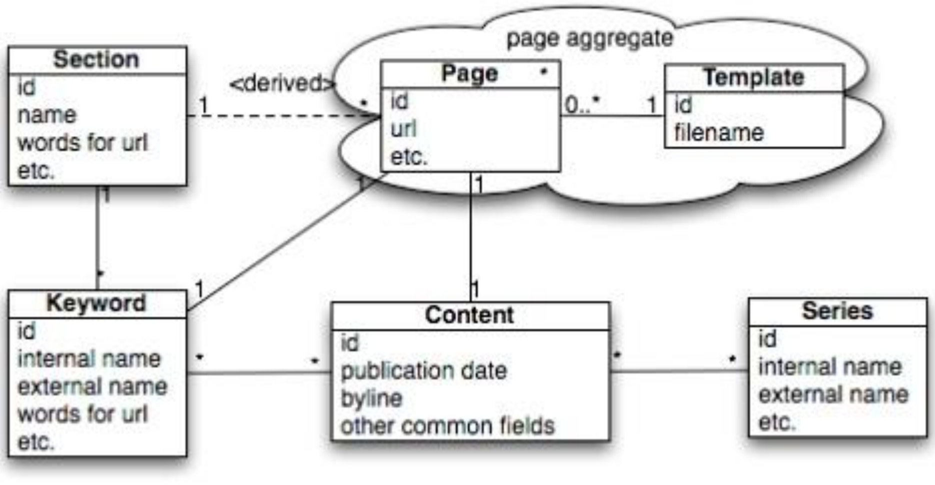


This single evolution to the model is the result of a number of smaller evolutions to our ubiquitous language. We now have three extra words: *Audio*, *Video* and *Content*; our editors have learned about inheritance and can use it in future iterations of the model; and we have a future expansion strategy for adding new content types and have made this simple for our editors.

4.2. Evolutionary step 2:

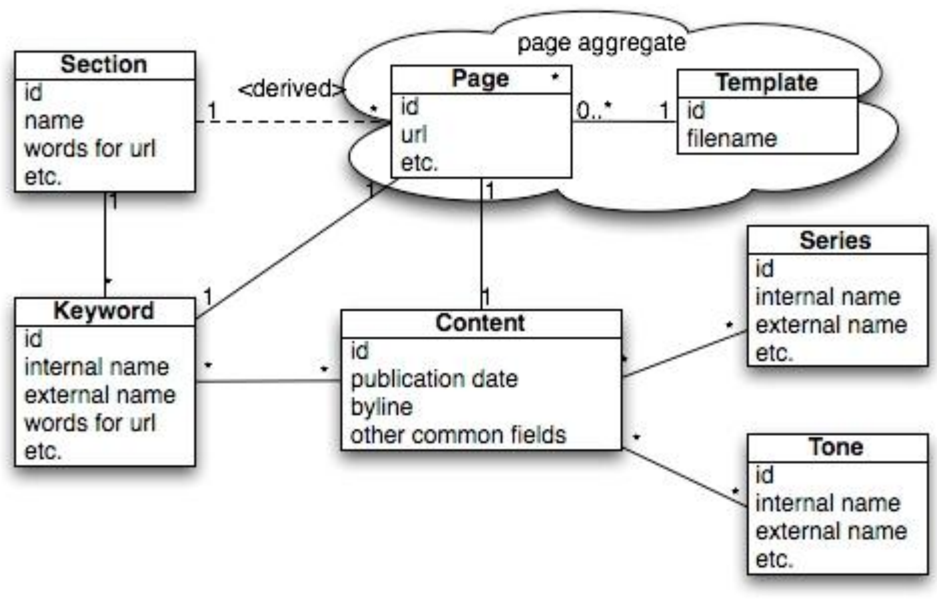
As our model was extended to include more types of *Content* it needed to be categorised more flexibly. We began adding extra types of metadata to our domain model, but it was not exactly clear where the editors' final intentions lay.

Here is the model for *series* that our editors designed:



The editors also required that the system dealt with content differently depending on the *Tone* of the *Content*. Examples of different *Tones* are reviews, obituaries, reader offers, and letters.

Here is the model for *Tone* that our editors designed:



4.3. Evolutionary step 3: Refactoring the metadata

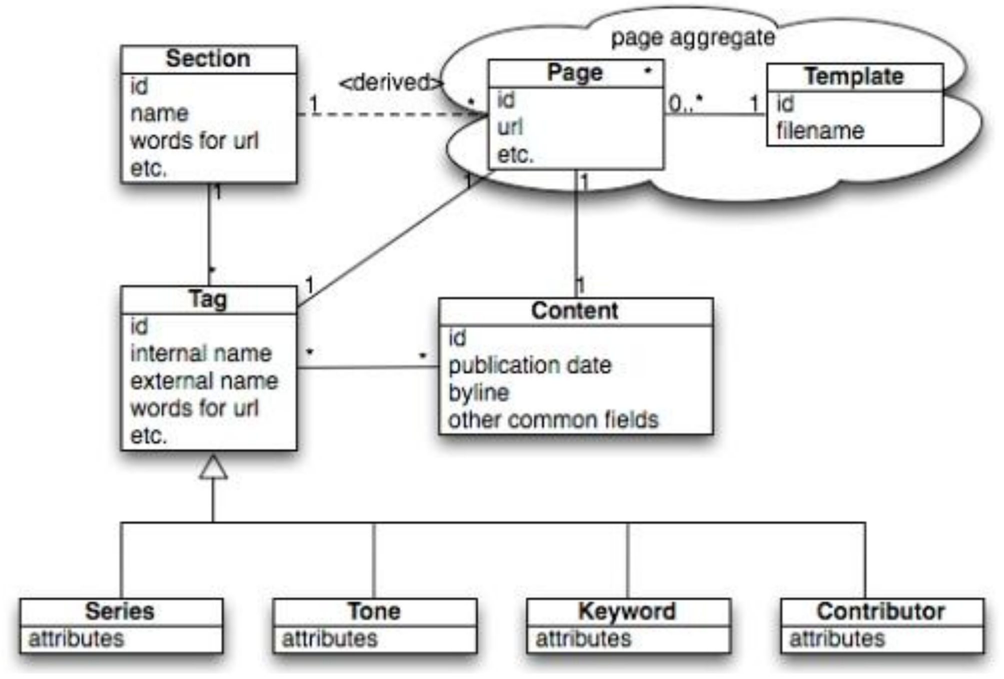
Our editors wanted to add the concept of *Content* having a *Contributor*. A *Contributor* is someone who creates content, and might be a writer for articles or a producer for videos.

The editors also saw another problem.They had to ask for a tool to be built to create a *Series* and another tool built to create a *Tone*. They had to specify how these objects related to *Content* and *Page*. Each time they found that they were specifying very similar development tasks for both types of domain objects; it was time-consuming and repetitive.

This was clearly become a problem. It seemed that our editors had spotted something wrong with our model that the developers had not. Why was it so expensive to add new metadata objects?We held another domain modelling session with the editors to try and identify the problem.

In the meeting our editors suggested that all of the existing types of metadata could in fact be derived from the same base idea.We refactored the model to introduce a new superclass called *Tag* and subclassed the other metadata.

Our revised model now looked like this:

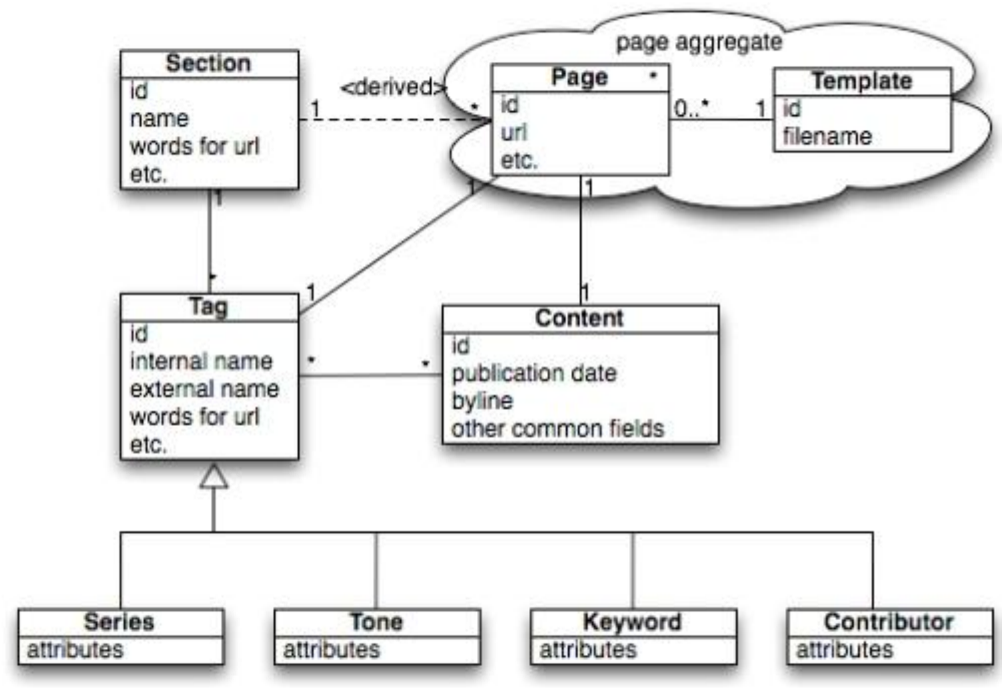


5. Evolution at the code level

DDD has an impact on the code level, too, and evolving business demands meant there were changes there, too.

5.1. Structuring the model

When structuring a domain model the first thing to identify is the aggregates that occur within the domain. An aggregate can be thought of as a collection of related objects that have references between each other.



Looking at the examples of our model as defined above we can start to see flavours of objects forming. We have the *Page* and *Template* objects which act together to give our web pages a URL and a look and feel.

We can also see another aggregate forming. This is the collection of metadata objects: *Tag, Series, Tone,* etc.

The Java programming language provides an ideal way to model these aggregates. We can use Java packages to model each aggregate, and standard POJOs to model each domain object.

The package structure for the above model looks like this ("r2" is the name of our suite of applications):

```
com.gu.r2.model.page
com.gu.r2.model.tag
com.gu.r2.model.content
com.gu.r2.model.content.article
com.gu.r2.model.content.video
com.gu.r2.model.content.audio
```

However, we have come to realise that the above package structure could cause us problems later and intend to change it. The problem can be illustrated by looking at a sample of the package structure from our front-end application to see how we are structuring our controllers:

```
com.gu.r2.frontend.controller.page
com.gu.r2.frontend.controller.articl
```

Here we see that our codebase is starting to fragment. We have extracted all of our aggregates into packages but we do not have a single package that contains every object that is related to that aggregate. This means that we could have difficulties resolving dependencies if we wished to break up the application in the future due to the domain becoming too large to manage as a single unit. An improved structure would be:

```
com.gu.r2.page.model   (domain objects in the page aggregate)
com.gu.r2.page.controller (controllers providing access to aggregate)
com.gu.r2.content.article.model
com.gu.r2.content.article.controller
...
etc
```

5.2. Evolution of the core DDD concepts

An application built following the principles of domain driven design will feature four broad types of objects: entities, value objects, repositories and services.

5.2.1. Entities

Entities are objects that exist within an aggregate and have identity. Not all entities are aggregate roots but only entities can be aggregate roots.

The confusion seemed to be related in part to our use of Hibernate to persist our entities. As we are using Hibernate we generally model our entities as simple POJOs. Each entity has properties that can be accessed with setter and getter methods. Each property is mapped in an XML file defining how it should be persisted in the database.

An example of this type of mistake can be seen in this (simplified) code snippet. We have a simple entity object to represent a football match:

```
public class FootballMatch extends IdBasedDomainObject
{
    private final FootballTeam homeTeam;
    private final FootballTeam awayTeam;
    private int homeTeamGoalsScored;
    private int awayTeamGoalsScored;

    FootballMatch(FootballTeam homeTeam, FootballTeam awayTeam) {
        this.homeTeam = homeTeam;
        this.awayTeam = awayTeam;
    }

    public FootballTeam getHomeTeam() {
        return homeTeam;
    }

    public FootballTeam getAwayTeam() {
        return awayTeam;
    }
    public int getHomeTeamScore() {
        return homeTeamScore;
    }

    public void setHomeTeamScore(int score) {
        this.homeTeamScore = score;
    }

    public void setAwayTeamScore(int score) {
        this.awayTeamScore = score;
    }
}
```

This entity object uses `FootballTeam` entities to model the teams, and looks like the type of object that any Java developer using Hibernate will be familiar with. Each property of this entity is persisted in the database, and although that detail is not really important from the perspective of domain driven design, our developers were elevating persisted properties to a higher status than they deserved.

```
public class FootballMatchSummary {

    public FootballTeam getWinningTeam(FootballMatch footballMatch) {
        if(footballMatch.getHomeTeamScore() >
footballMatch.getAwayTeamScore()) {
            return footballMatch.getHomeTeam();
        }
        return footballMatch.getAwayTeam();
    }
}
```

A moment's thought should suggest that something has gone wrong. We have created a new class called a `FootballMatchSummary` which exists in our domain model but does not mean anything to the business.Our developers were thinking of the entity as an entity in a traditional ORM sense rather than as a business-owned and business-defined domain object.

We also have a strict rule that developers cannot create new object types in the model that do not mean anything to the business.

5.2.2. Value objects

Value objects are properties of entities that do not have a natural identity that means anything within the domain, but which do express a concept that has meaning within the domain. These objects are important as they add clarity to the ubiquitous language.

An example of the clarifying abilities of value objects can be seen by looking at our *Page* class in more detail. Any *Page* on our system has two possible URLs.

A simplistic view of these two possible URLs would be to model them both as string objects on the *Page* class:

```
 public String getUrl();
 public String getCmsUrl();
```

It is difficult to know exactly what these methods will return by looking at their signatures other than the fact that they return strings.We may have a method signature that looks like:

```
public Page loadPage(String url);
```

Which URL is required here? The public facing one or the CMS url?

We may have differing validation rules for internal and external URLs, and wish to execute different operations on them. How can we encapsulate this logic correctly if we do not have anywhere to put it?

The evolution suggested by domain driven design is that we model these value objects explicitly. We should create simple wrapper classes that represent the value objects to type them. If we do so our signature on *Page* now looks like this:

```
 public Url getUrl();
 public CmsPath getCmsPath();
```

We can now pass *CmsPath* or *Url* objects around in the application safely, and have a conversation with our business representatives about this code in a language that they will understand.

5.2.3. Repositories

Repositories are objects that exist within an aggregate to provide access to instances of that aggregate’s root object while abstracting away any persistence mechanisms. These objects are asked business questions and respond with domain objects.

But repositories are domain objects: they answer business questions.

Here we can see this in action by taking a small sample of code from our page repository:

```
 private final PageDAO<Page> pageDAO;
 private final PagesRelatedBySectionDAO pagesRelatedBySectionDAO;

 public PageRepository(PageDAO<Page> pageDAO,
         EditorialPagesInThisSectionDAO pagesInThisSectionDAO,
         PagesRelatedBySectionDAO pagesRelatedBySectionDAO) {
     this.pageDAO = pageDAO;
     this.pagesRelatedBySectionDAO = pagesRelatedBySectionDAO;
 }

 public List<Page> getAudioPagesForPodcastSeriesOrderedByPublicationDate(Series
series, int maxNumberOfPages) {
     return
pageDAO.getAudioPagesForPodcastSeriesOrderedByPublicationDate(series,
maxNumberOfPages);
 }

 public List<Page> getLatestPagesForSection(Section section, int maxResults) {
     return pagesRelatedBySectionDAO.getLatestPagesForSection(section,
maxResults);
 }
```

Our repository contains business questions: Get *Pages* for a specific *Series* of *Podcasts* ordered by *PublicationDate*. Get the latest *Pages* for a specific *Section.* We can see the business domain language in use here.

It took us a while to realise that regarding repositories as domain objects could help us overcome technical problems with the implementation of our domain model.

5.2.4. Services

Services are objects that manage the execution of business problems by orchestrating the interaction of domain objects.

The primary problem is that it is quite easy for developers to create services that really should not exist; they either end up containing domain logic that should exist in domain objects or they actually represent missing domain objects that have not been created as part of the model.

Early on in the project we started to find services cropping up with names like `ArticleService`. What is this? We have a domain object called *Article*; what is the purpose of an article service? On inspection of the code we found that this class seemed to be following a similar pattern to the `FootballMatchSummary` object discussed above, containing domain logic that really belonged on the core domain object.

In order to address this behaviour we performed a code review of all services in the application and executed refactorings to move the logic into appropriate domain objects. We also instigated a new rule: Any service object must have a verb in its name. This simple rule stops developers from creating a class like an `ArticleService`. We can instead create a classes like an `ArticlePublishingService` and `ArticleDeletionService`.

6. Some final lessons of DDD in an evolving architecture

Despite the challenges we have found significant advantages in using DDD in an evolving and Agile environment, and we learnt these lessons among others:

- You don't have to understand the whole of the domain to add business value. You don’t even need a full knowledge of domain driven design. All members of the team can reach a shared understanding of the model as much as they need at any time.
- It *is* possible (even essential) to evolve the model *and* the process over time and to correct previous mistakes as our shared understanding improves.

The full domain model for our system is very much larger than the simplified version described here, and is constantly evolving as our business expands.Indeed, our business representatives often wish to experiment with new ideas and approaches. Some will bear fruit and others will not be successful.