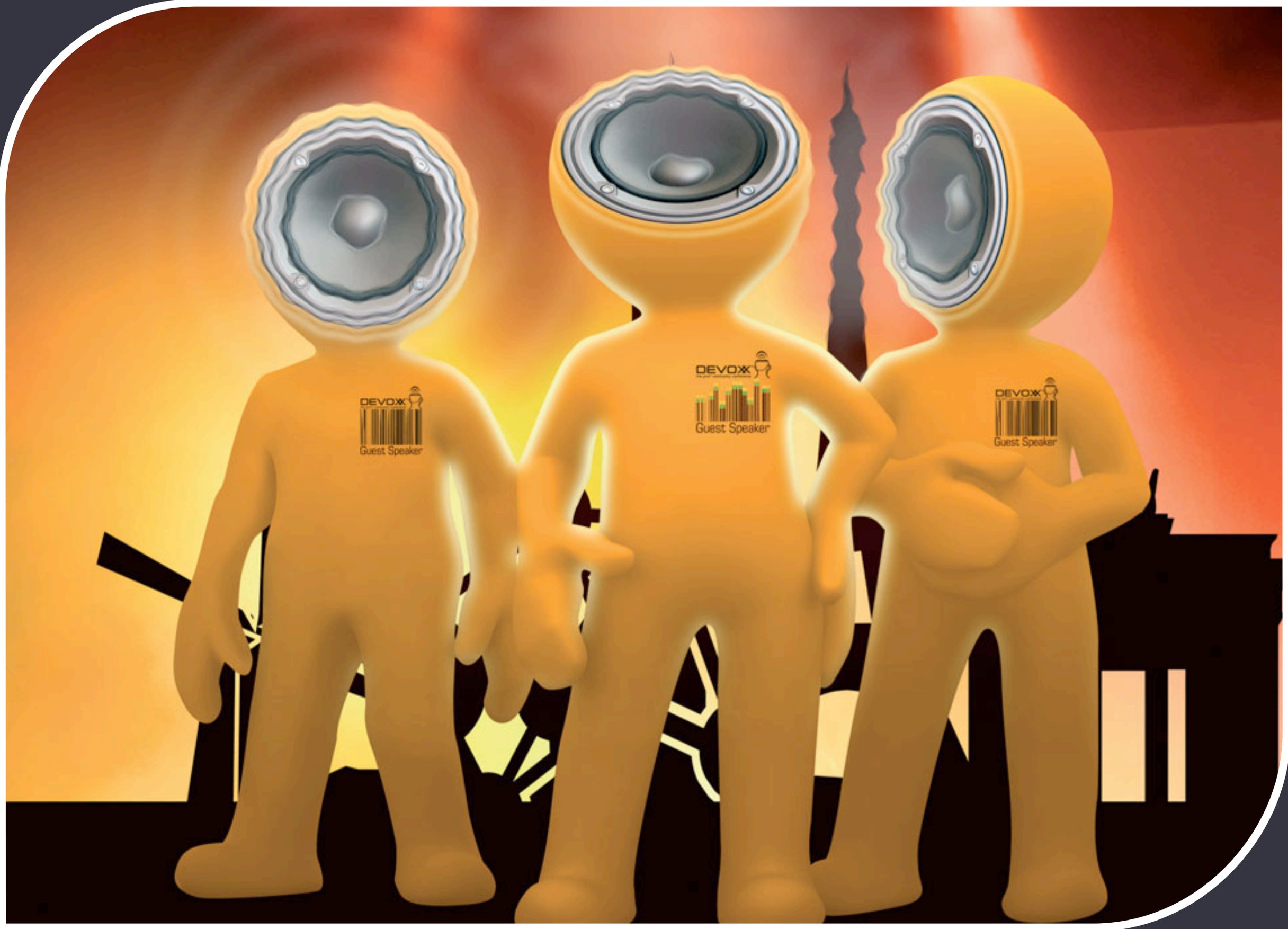

DEVOXX
Guest Speaker
DEVOXX
Guest Speaker
DEVOXX
Guest Speaker

Introducing Bastion, a DDD Framework

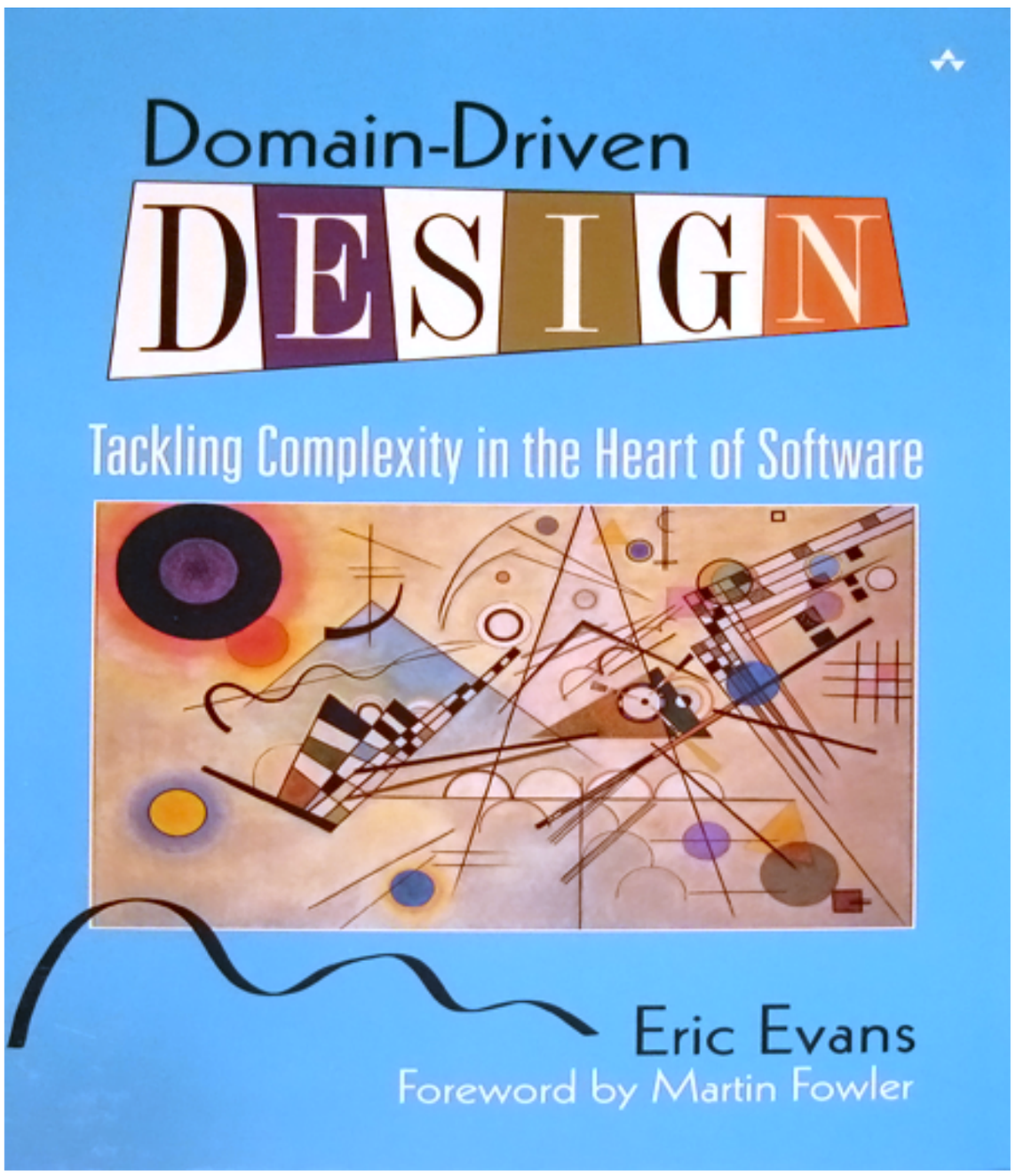
Danny Lagrouw
Profict b.v.
www.profict.nl

Hi, my name is Danny Lagrouw, I work for a Dutch Java company called Profict. I'm here today to talk about Domain-Driven Design, or DDD.

Domain-Driven Design

First of all, how many of you are familiar with DDD? Maybe you've used it in a project, or you've just read about it?
Okay, then you will probably also know about this book...

Evans



Domain-Driven Design, Tackling Complexity in the Heart of Software, written by Eric Evans. It is *the* standard book to read if you want to know about DDD; so for those of you *un*familiar with DDD, go and buy this book (after this session of course).
Obviously 15 minutes is not enough to go into even the basics of DDD if you don't know anything about it yet. What I can tell you is what I think are the three main principles for DDD that Evans describes.

One language

First of all, Evans says, Domain-Driven Design means that you need one ubiquitous language, a language to be spoken by all people on the project. That way, you'll have less chance of confusion, less will be lost in translation, and the application will better match the expectations your client has of it.

One domain model

Throughout the project, everyone should use the same domain model. The same model you discuss with your users, the domain experts, should be the one you implement. And from that follows that the implemented domain should contain nothing but business classes--classes your users will recognize. There shouldn’t be anything in there *but* domain logic--nothing technical, nothing that isn’t part of the domain model.

One layer for the domain

(and nothing but the domain)

Which is the third principle: the domain layer in your application should contain nothing but the domain logic. Everything else should be in separate layers. Most notably user interface and infrastructural code, like database access.
So for me, these are the three main principles from Evans' book.
But there's more to DDD then what Evans tells us.

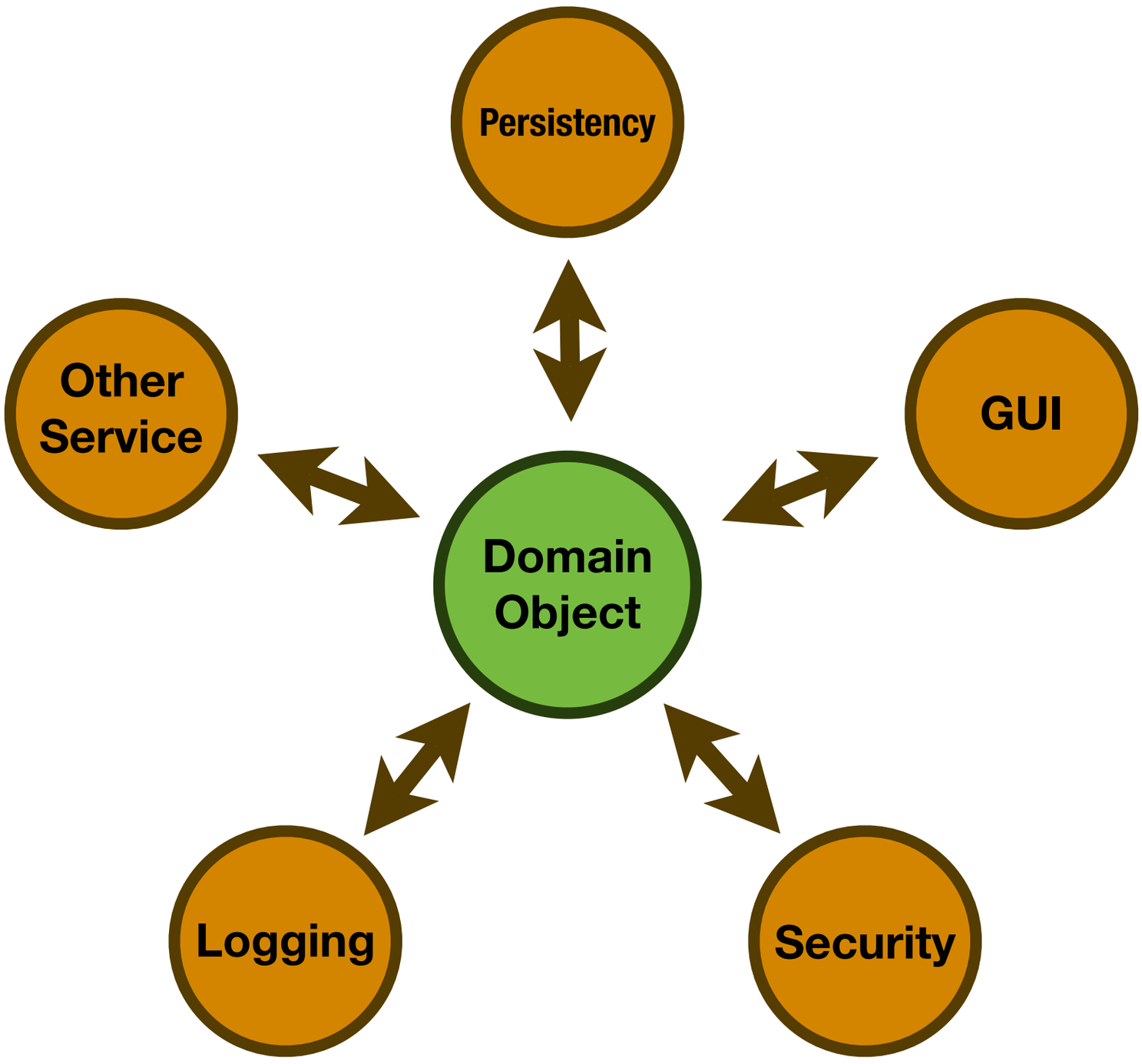
Generic, reusable services

Ruud van Vliet

Rob Vens

Back in 2000, my colleague Ruud van Vliet wrote an article about his idea of generic, reusable services, a way to isolate business logic from the rest of the application. Dutch DDD guru Rob Vens uses that same concept for his so-called satellite model of the domain.

Rob Vens' Satellite Model



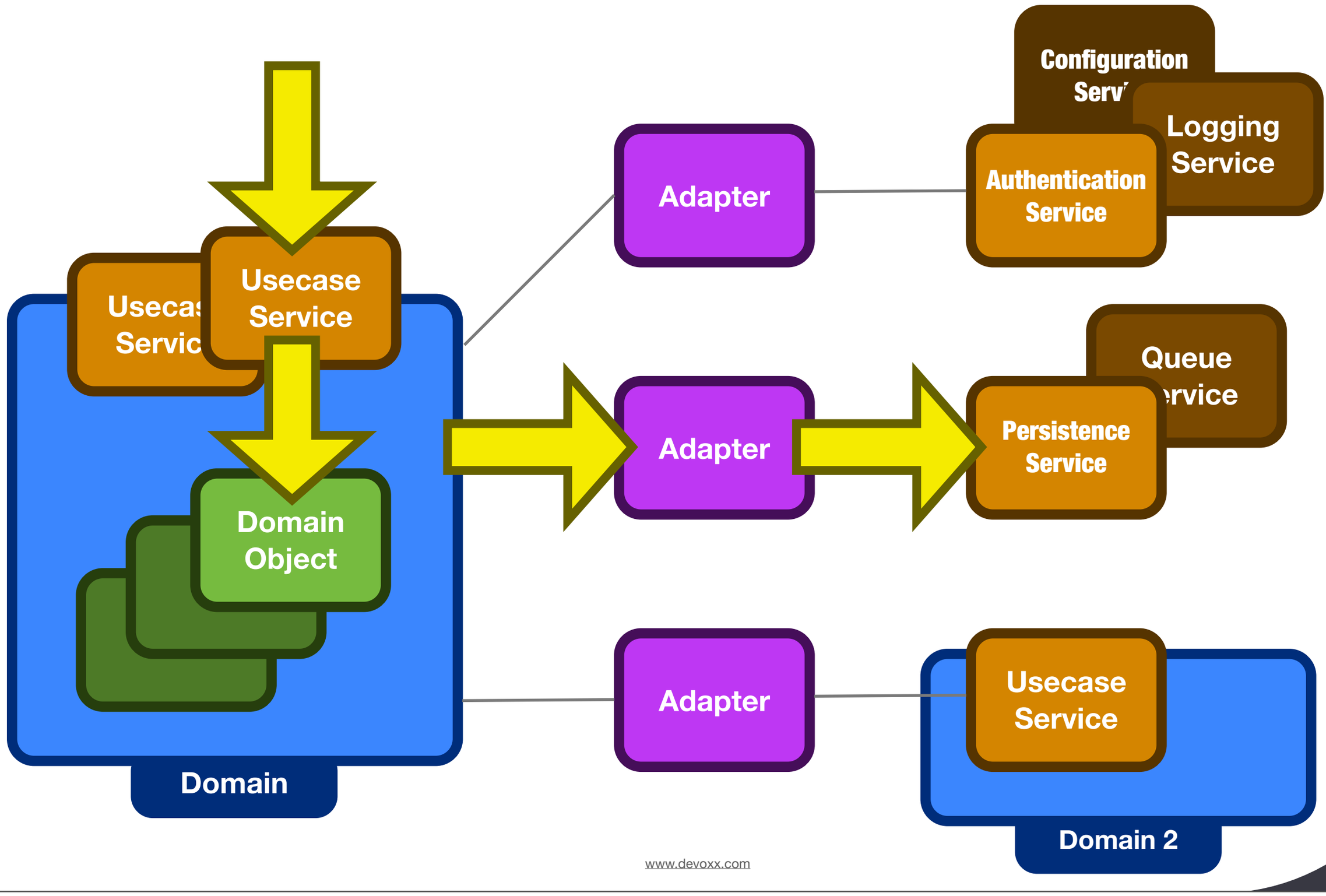
Vens, like Evans, says that the domain should be the **center** of the application. His view of a domain is that it should always be live, always running, ever expanding, almost unlimited in size. Placed around the domain are **adapter** classes that listen to events within the domain and respond by calling external services. Likewise, events from outside the domain can be caught by adapters and fed into the domain. Because domain and services are **decoupled,** the services can be made generic and are therefore reusable across domains. That way, you will end up with a **component library** of reusable services.

That’s the theory, now let’s build something

The **problem** with both Evans and Vens is, that neither offers any real, pragmatic guidelines for bringing their theory into practice. Evans focusses on his method of designing the domain model. Vens’ ideas about a live domain, ever-growing, might make sense in his preferred environment, Smalltalk.
But for the kind of application that I typically work on--Java based applications whose main interface is a web app--there’s no ready-to-use solution, no sample applications, no framework to get me up and running. And that is exactly the reason why I started working on...

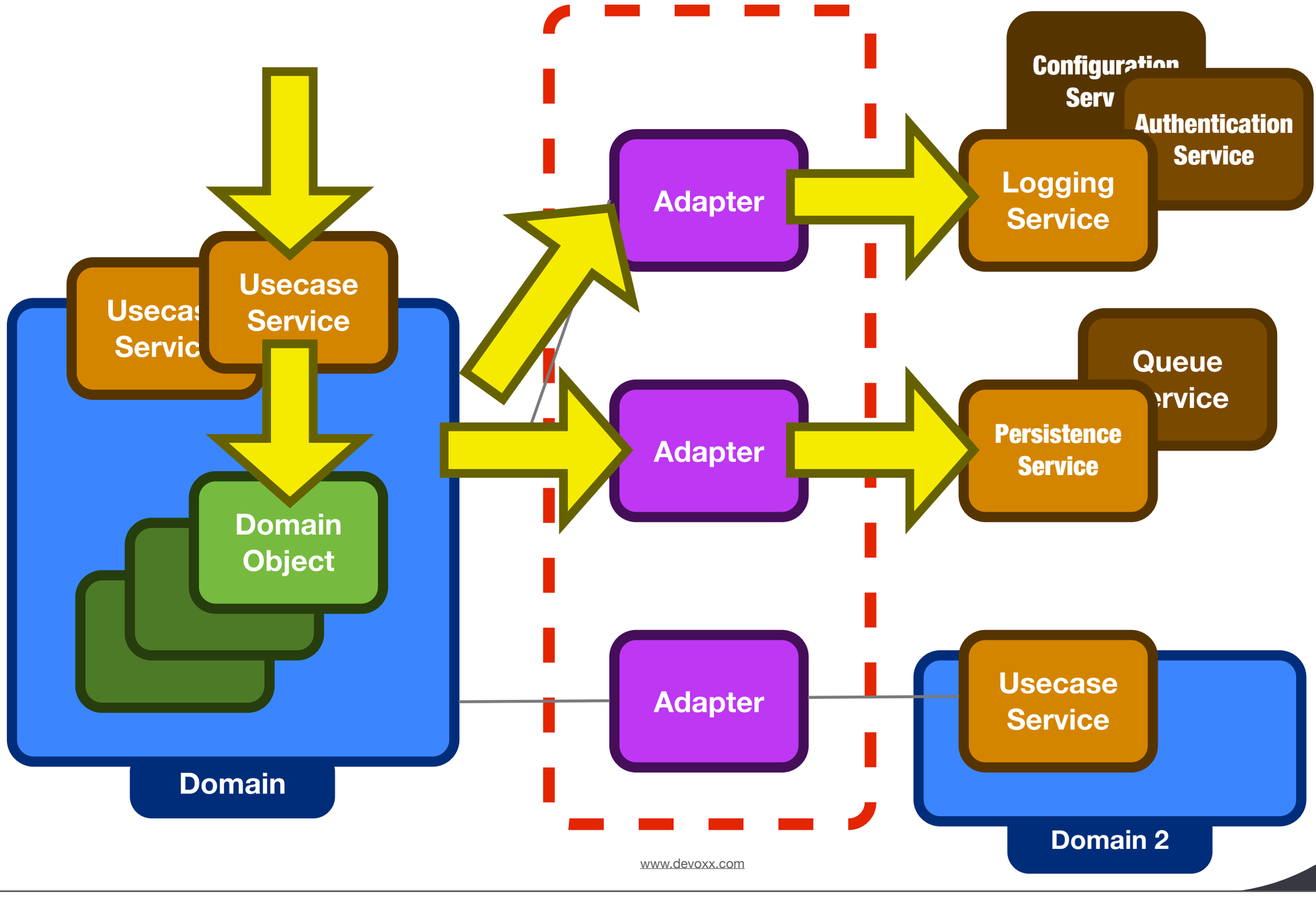
Bastion, a light-weight framework for developing DDD applications in the spirit of Evans and Vens. The name Bastion symbolizes the wall that separates the domain from the rest of the system.

The Bastion Model



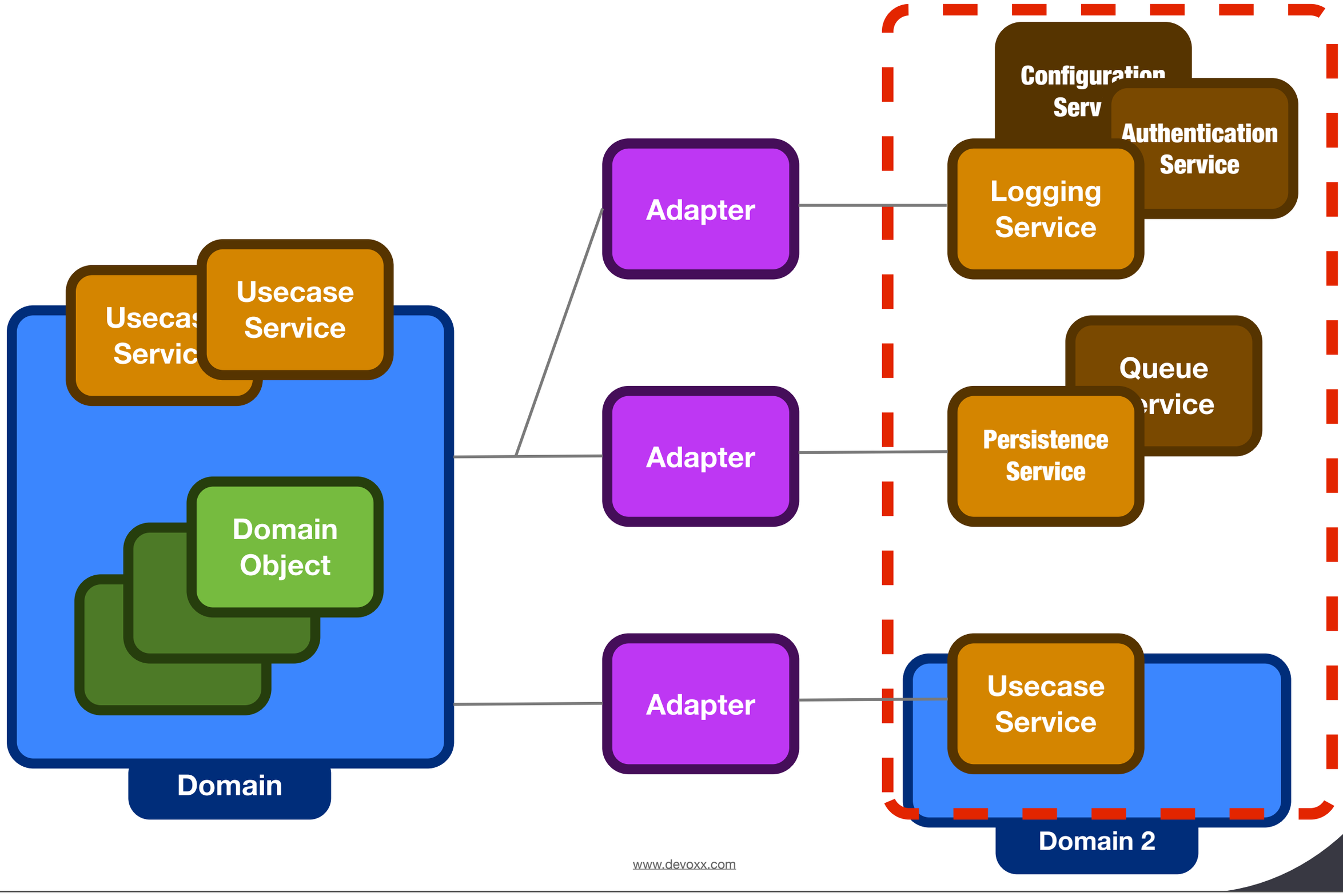
So what does Bastion's model look like? Like Vens' satellite model, Bastion separates domain logic, business logic, from everything else, using an adapter layer.
All domain logic is implemented in the domain according to the domain model. The domain's interface to the outside world consists of **usecase services,** that would mirror the actions in usecase scenario's. The usecase services delegate to **domain objects,** where the real domain logic is.
Outside the domain are these **adapters,** that listen to messages sent by the domain. For instance, the domain could 'broadcast' a message that a new domain object has been created. An appropriate adapter receives this message and calls a method on a persistence **service.** This service knows how to store the object in the database.

Bastion - Adapters



The adapters do nothing but translate domain messages into service calls. The adapters are the configuration of your domain. Through them you control how specific domain messages **will be handled.** You could, for instance, link the same message type to different adapters and services. When a domain object has been created, you could, besides store it in the database, trigger a logging service to log the event.

Bastion - Services



The services are completely independent of the domain and the adapters. The services know nothing about the domain, just like the domain knows nothing about the services. That's what the adapter layer is for. As a result, you can make the services **generic**. And that makes them **reusable** with other domains and other applications. When a **new version** arrives for the technology your service uses, you can update and test that service--not your domain. If you want to use a **different technological solution** for one of the services, you re-link the adapter to another service. Your domain classes will never know the difference.

Bastion Code Sample

Domain Object

```
public Client(String name) {
   this.name = name;
   Domain.notify(new RegisterMessage(this));
}
```

Domain Configuration

```
JmsService jmsService = new JmsService(connectionFactory, queue);
QueueAdapter adapter = new QueueAdapter(jmsService);
domain.registerAdapter(RegisterMessage.class, adapter);
```

Bastion Code Sample

Adapter

```
public void handle(RegisterMessage message) {
    service.send("Created: " + message.getDomainObject().toString());
}
```

Service

```
public void send(String messageText) throws Exception {
    Connection connection = null;
    connection = connectionFactory.createConnection();
    Session session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
    MessageProducer producer = session.createProducer(destination);
    TextMessage message = session.createTextMessage(messageText);
    producer.send(message);
}
```


So, that's how Bastion works. Your domain objects will only execute domain logic. If something happens in the domain that the outside world should know about (like object creation) the domain will trigger the corresponding adapter. If a domain object needs anything from the outside world (webservice, data from a database), it uses the same mechanism to trigger an adapter to get the information.

Bastion - Service Examples

- Store data and query data from a database
- Call a web service
- Send message to JMS queue
- Retrieve configuration parameters
- Query an LDAP server
- Get generated test data
- Communicate with other applications

Here's some more examples of services that Bastion provides or could provide.

Bastion - Roadmap

- Version 0.7
  - (Better) test coverage
  - Getting started manual
  - More services
- Your participation?

Bastion has recently seen its first public release, version 0.6. At the moment, the main goals for the next release are more test coverage, a getting started manual, and adding more services. At the same time, we're doing a pilot project at Profict using Bastion, and obviously my ultimate goal is to get it used in production systems. So if you're interested in looking at Bastion, using it in your own application, or even helping out with further development, I'd very much like to hear.

Further information

http://bastionframework.org

danny@bastionframework.org

Q&A

Thanks for your attention!

http://bastionframework.org

danny@bastionframework.org