_

Adaptive Software Life Cycle

Design, Develop, Operate, Manage and Govern Software in Agility

_

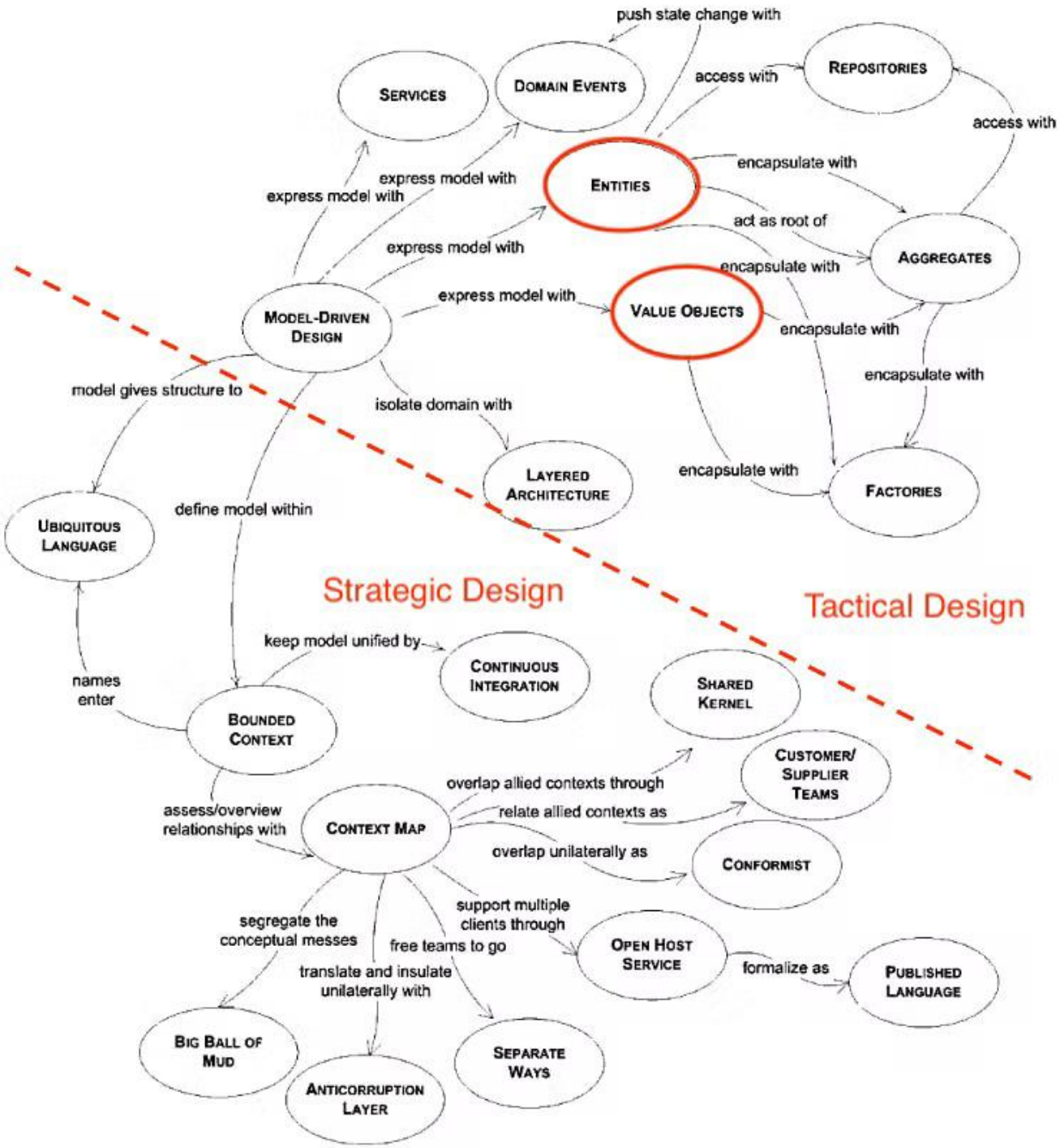
_

Adaptive Code,

Domain Driven Design (DDD) and

Microservice

Aligned with Clean Code and Clean Architecture Patterns & Principles


push state change with
SERVICES
DOMAIN EVENTS
access with
REPOSITORIES
access with
express model with
express model with
ENTITIES
encapsulate with
act as root of
express model with
AGGREGATES
encapsulate with
express model with
MODEL-DRIVEN DESIGN
VALUE OBJECTS
encapsulate with
encapsulate with
model gives structure to
isolate domain with
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
define model within
UBIQUITOUS LANGUAGE
Strategic Design
Tactical Design
keep model unified by
CONTINUOUS INTEGRATION
SHARED KERNEL
names enter
BOUNDED CONTEXT
CUSTOMER/ SUPPLIER TEAMS
overlap allied contexts through
assess/overview relationships with
CONTEXT MAP
relate allied contexts as
overlap unilaterally as
CONFORMIST
support multiple clients through
segregate the conceptual messes
free teams to go
OPEN HOST SERVICE
formalize as
PUBLISHED LANGUAGE
translate and insulate unilaterally with
BIG BALL OF MUD
SEPARATE WAYS
ANTICORRUPTION LAYER

Why Domain-Driven Design Matters.

Software is not just about code. If you think about it, code is rarely the end goal of our profession. Code is just the medium to solve business problems. So why does it have to talk a different language? Domain-Driven Design emphasizes making sure businesses and software speak the same language. Once broken the barrier, there is no need for translations or tedious syncing, information doesn't get lost. Everyone contributes to discovering the Business Domain, not just coders. The resulting software is the only truth for the common language.

Domain-Driven Design it also provides a framework for strategic and tactical design – strategic to pinpoint the most important areas to develop based on business value, and tactical to build a working Domain Model of battle-tested building blocks and patterns.

WRITING SOFTWARE IS EASY– at least if it’s greenfield software. When it comes to modifying code written by other developers or code you wrote six months ago, it can be a bit of a bore at best and a nightmare at worst. The software works, but you aren’t sure exactly how. It contains all the right frameworks and patterns, and has been created using an agile approach, but introducing new features into the codebase is harder than it should be. Even business experts aren’t helpful because the code bears no resemblance to the language they use. Working on such systems becomes a chore, leaving developers frustrated and devoid of any coding pleasure.

Domain-Driven Design (DDD) is a process that aligns your code with the reality of your problem domain. As your product evolves, adding new features becomes as easy as it was in the good old days of greenfield development. Although DDD understands the need for software patterns, principles, methodologies, and frameworks, it values developers and domain experts working together to understand domain concepts, policies, and logic equally. With a greater knowledge of the problem domain and a synergy with the business, developers are more likely to build software that is more readable and easier to adapt for future enhancement.

Following the DDD philosophy will give developers the knowledge and skills they need to tackle large or complex business systems effectively. Future enhancement requests won’t be met with an air of dread, and developers will no longer have stigma attached to the legacy application. In fact, the term legacy will be recategorized in a developer’s mind as meaning this: a system that continues to give value for the business.

```
Considering Domain-Driven Design.

Domain-Driven Design is not a silver bullet; as with everything in software, it
depends on the context. As a rule of thumb, use it to simplify your Domain, but
never to add more complexity.

If your application is data-centric and your use cases mainly manipulate rows in a
database and perform CRUD operations — that is, Create, Read, Update, and Delete —
you don't need Domain-Driven Design. Instead, the only thing your company needs is
a fancy face in front of your database.

If your application has less than 30 use cases, it might be simpler to use a
framework like Symfony or Laravel to handle your business logic.

However, if your application has more than 30 use cases, your system may be moving
toward the dreaded Big Ball of Mud. If you know for sure your system will grow in
complexity, you should consider using Domain-Driven Design to fight that
complexity.

If you know your application is going to grow and is likely to change often,
Domain-Driven Design will definitely help in managing the complexity and
refactoring your model over time.

If you don't understand the Domain you're working on because it's new and nobody
has invested in a solution before, this might mean it's complex enough for you to
start applying Domain-Driven Design. In this case, you'll need to work closely
with Domain Experts to get the models right.
```

> More importantly, Domain-Driven Design is not about technology. Instead, it's about developing knowledge around business and using technology to provide value. Only once you're capable of understanding the business your company works within will you be able to participate in the software model discovery process to produce a Ubiquitous Language.

_

From object basics through design pattern principles

_

```
- Advocacy and Agnosticism: The Object Debate.

    Many excellent programmers have produced excellent code for years without
using objects. One core difference between object-oriented and procedural code can
be found in the way that responsibility is distributed. Procedural code Takes the
form of a sequential series of commands and method calls. The controlling code
tends to take responsibility for handling differing conditions. This top-down
control can result in the development of duplications and dependencies across a
project. Object-oriented code Tries to minimize these dependencies by moving
responsibility for handling tasks away from client code and toward the objects in
the system.
```

```
- Patterns Promote Design and A Common Vocabulary.

    Although I’m an inveterate reinventor of wheels, the thrust of my argument is
not that we should all throw away our frameworks and build MVC applications from
scratch (at least not always). It is rather that, as developers, we should
understand the problems that frameworks solve, and the strategies they use to
solve them. We should be able to evaluate frameworks not only functionally but in
terms of the design decisions their creators have made, and to judge the quality
of their implementations. And yes, when the conditions are right, we should go
ahead and build our own spare and focused applications, and, over time, compile
our own libraries of reusable code.
```

.

_

Object Basics

Inheritance

The Inheritance Problem

```
// listing 03.30

class ShopProduct
{
    public $numPages;
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;

    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float $price,
        int $numPages = 0,
        int $playLength = 0
    ) {
        $this->title = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName = $mainName;
        $this->price = $price;
        $this->numPages = $numPages;
        $this->play
    }

    public function getNumberOfPages()
    {
        return $this->numPages;
    }

    public function getPlayLength()
    {
        return $this->playLength;
    }

    public function getProducer()
    {
        return $this->producerFirstName . " "
```

```
            . $this->producerMainName;
        }
    }
```

So forcing fields that don't belong together into a single class leads to bloated objects with redundant properties and methods.

Consider a method that summarizes a product

```
// listing 03.31

public function getSummaryLine()
{
    $base = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    if ($this->type == 'book') {
        $base .= ": page count - {$this->numPages}";
    } elseif ($this->type == 'cd') {
        $base .= ": playing time - {$this->playLength}";
    }
    return $base;
}
```

As ShopProduct is beginning to feel like two classes in one, I could accept this and create two types rather than one.

Consider with the ShopProductWriter class? Its write() method is designed to work with a single type: ShopProduct

```
// listing 03.34

class ShopProductWriter
{
    public function write($shopProduct)
    {
        if (
            ! ($shopProduct instanceof CdProduct) &&
            ! ($shopProduct instanceof BookProduct)
        ) {
            die("wrong type supplied");
        }
        $str = "{$shopProduct->title}: "
            . $shopProduct->getProducer()
            . " ({$shopProduct->price})\n";
        print $str;
    }
}
```

.

_

Objects And Design

Defining Code Design

One sense of code design concerns the definition of a system: the determination of a system's requirements, scope, and objectives. What does the system need to do? For whom does it need to do it? What are the outputs of the system? Do they meet the stated need? On a lower level, design can be taken to mean the process by which you define the participants of a system and organize their relationships. This chapter is concerned with the second sense: the definition and disposition of classes and objects.

As part of the design process, you must decide when an operation should belong to a type and when it should belong to another class used by the type. Everywhere you turn, you are presented with choices, decisions that might lead to clarity and elegance or might mire you in compromise.

_

Object-Oriented and Procedural Programming

Procedural

```
// listing 06.01

function readParams(string $source): array
{
    $params = [];
    // read text parameters from $source
    return $params;
}

function writeParams(array $params, string $source)
{
    // write text parameters to $source
}
```

```
// listing 06.02

$file = __DIR__ . "/params.txt";
$params = [
    "key1" => "val1",
    "key2" => "val2",
    "key3" => "val3",
];
writeParams($params, $file);
```

```
$output = readParams($file);
print_r($output);
```

```
// listing 06.03

function readParams(string $source): array
{
    $params = [];

    if (preg_match("/\.xml$/i", $source)) {
        // read XML parameters from $source
    } else {
        // read text parameters from $source
    }

    return $params;
}

function writeParams(array $params, string $source)
{
    if (preg_match("/\.xml$/i", $source)) {
        // write XML parameters to $source
    } else {
        // write text parameters to $source
    }
}
```

Object-Oriented

```
// listing 06.04

abstract class ParamHandler
{
    protected $source;
    protected $params = [];

    public function __construct(string $source)
    {
        $this->source = $source;
    }

    public function addParam(string $key, string $val)
    {
        $this->params[$key] = $val;
    }

    public function getAllParams(): array
```

```
    {
        return $this->params;
    }

    public static function getInstance(string $filename): ParamHandler
    {
        if (preg_match("/\.xml$/i", $filename)) {
            return new XmlParamHandler($filename);
        }
        return new TextParamHandler($filename);
    }

    abstract public function write(): bool;
    abstract public function read(): bool;
}
```

```
// listing 06.05

class XmlParamHandler extends ParamHandler
{
    public function write(): bool
    {
        // write XML
        // using $this->params
    }

    public function read(): bool
    {
        // read XML
        // and populate $this->params
    }
}
```

```
// listing 06.06

class TextParamHandler extends ParamHandler
{
    public function write(): bool
    {
        // write text
        // using $this->params
    }

    public function read(): bool
    {
        // read text
        // and populate $this->params
```

```
    }
}
```

```
// listing 06.07

$test = ParamHandler::getInstance(__DIR__ . "/params.xml");
$test->addParam("key1", "val1");
$test->addParam("key2", "val2");
$test->addParam("key3", "val3");
$test->write(); // writing in XML format
```

```
// listing 06.08
$test = ParamHandler::getInstance(__DIR__ . "/params.txt");
$test->read(); // reading in text format
$params = $test->getAllParams();
print_r($params);
```

So, what can we learn from these two approaches?

- Responsibility

  The controlling code in the procedural example takes responsibility for deciding about format—not once, but twice. The conditional code is tidied away into functions, certainly, but this merely disguises the fact of a single flow, making decisions as it goes. Calls to readParams() and to writeParams() take place in different contexts, so we are forced to repeat the file extension test in each function (or to perform variations on this test).

  In the object-oriented version, this choice about file format is made in the static getInstance() method, which tests the file extension only once, serving up the correct subclass. The client code takes no responsibility for implementation. It uses the provided object with no knowledge of, or interest in, the particular subclass it belongs to. It knows only that it is working with a ParamHandler object, and that it will support write() and read(). While the procedural code busies itself about details, the object-oriented code works only with an interface, unconcerned about the details of implementation. Because responsibility for implementation lies with the objects and not with the client code, it would be easy to switch in support for new formats transparently.

- Cohesion

  Cohesion is the extent to which proximate procedures are related to one another. Ideally, you should create components that share a clear responsibility. If your code spreads related routines widely, you will find them harder to maintain as you have to hunt around to make changes.

  Our ParamHandler classes collect related procedures into a common context. The methods for working with XML share a context in which they can share data and where changes to one method can easily be

reflected in another if necessary (e.g., if you needed to change an XML element name). The ParamHandler classes can therefore be said to have high cohesion.

The procedural example, on the other hand, separates related procedures. The code for working with XML is spread across functions.

- Coupling

  Tight coupling occurs when discrete parts of a system's code are tightly bound up with one another so that a change in one part necessitates changes in the others. Tight coupling is by no means unique to procedural code, though the sequential nature of such code makes it prone to the problem.

  You can see this kind of coupling in the procedural example. The writeParams() and readParams() functions run the same test on a file extension to determine how they should work with data. Any change in logic you make to one will have to be implemented in the other. If you were to add a new format, for example, you would have to bring the functions into line with one another, so that they both implement a new file extension test in the same way. This problem can only get worse as you add new parameter-related functions.

  The object-oriented example decouples the individual subclasses from one another and from the client code. If you were required to add a new parameter format, you could simply create a new subclass, amending a single test in the static getInstance() method.

- Orthogonality

  The killer combination of components with tightly defined responsibilities that are also independent from the wider system is sometimes referred to as orthogonality.

  Orthogonality, it is argued, promotes reuse in that components can be plugged into new systems without needing any special configuration. Such components will have clear inputs and outputs, independent of any wider context. Orthogonal code makes change easier because the impact of altering an implementation will be localized to the component being altered. Finally, orthogonal code is safer. The effects of bugs should be limited in scope. An error in highly interdependent code can easily cause knock-on effects in the wider system.

  There is nothing automatic about loose coupling and high cohesion in a class context. We could, after all, embed our entire procedural example into one misguided class. So how can we achieve this balance in our code? I usually start by considering the classes that should live in my system.

_

Choosing Your Classes

It can be surprisingly difficult to define the boundaries of your classes, especially as they will evolve with any system that you build.

It can seem straightforward when you are modeling the real world. Object-oriented systems often feature software representations of real things—Person, Invoice, and Shop classes abound. This would seem to suggest that defining a class is a matter of finding the things in your system and then giving them agency through methods. This is not a bad starting point, but it does have its dangers. If you see a class as a noun, a

subject for any number of verbs, then you may find it bloating as ongoing development and requirement changes call for it to do more and more things.

How should you think about defining classes? The best approach is to think of a class as having a primary responsibility and to make that responsibility as singular and focused as possible. Put the responsibility into words. It has been said that you should be able to describe a class's responsibility in 25 words or less, rarely using the words "and" or "or." If your sentence gets too long or mired in clauses, it is probably time to consider defining new classes along the lines of some of the responsibilities you have described.

_

Polymorphism

Polymorphism, or class switching.

Polymorphism is the maintenance of multiple implementations behind a common interface. This sounds complicated, but in fact, it should be very familiar to you by now. The need for polymorphism is often signaled by the presence of extensive conditional statements in your code.

It is important to note that polymorphism doesn't banish conditionals. Methods such as ParamHandler::getInstance() will often determine which objects to return based on switch or if statements. These tend to centralize the conditional code into one place, though.

_

Encapsulation

The hiding of data and functionality from a client.

Encapsulation is, in some ways, the key to object-oriented programming. Your objective should be to make each part as independent as possible from its peers. Classes and methods should receive as much information as is necessary to perform their allotted tasks, which should be limited in scope and clearly identified.

Encapsulation is a technique that should be observed equally by classes and their clients.

Polymorphism illustrates another kind of encapsulation. By placing different implementations behind a common interface, you hide these underlying strategies from the client. This means that any changes that are made behind this interface are transparent to the wider system. You can add new classes or change the code in a class without causing errors. The interface is what matters, not the mechanisms working beneath it. The more independent these mechanisms are kept, the less chance that changes or repairs will have a knock-on effect in your projects.

_

Forget How to Do It

Program to an interface not an implementation.

_

Four Signposts

Very few people get it absolutely right at the design stage. Most of us amend our code as requirements change or as we gain a deeper understanding of the nature of the problem we are addressing.

As you amend your code, it can easily drift beyond your control. A method is added here and a new class there, and gradually your system begins to decay. As you have seen already, your code can point the way to its own improvement. These pointers in code are sometimes referred to as code smells—that is, features in code that may suggest particular fixes or at least call you to look again at your design.

- Code Duplication
- The Class Who Knew Too Much
- The Jack of All Trades
- Conditional Statements

_

The UML

- Class Diagrams

  Aggregation and composition are similar to association. All describe a situation in which a class holds a permanent reference to one or more instances of another. With aggregation and composition, though, the referenced instances form an intrinsic part of the referring object.

  In the case of aggregation, the contained objects are a core part of the container, but they can also be contained by other objects at the same time. The aggregation relationship is illustrated by a line that begins with an unfilled diamond.

  Composition represents an even stronger relationship than this. In composition, the contained object can be referenced by its container only. It should be deleted when the container is deleted. Composition relationships are depicted in the same way as aggregation relationships, except that the diamond should be filled.

- Sequence Diagrams

.

_

Patterns

Why Use Design Patterns?

- A Design Pattern Defines a Problem
- A Design Pattern Defines a Solution
- Design Patterns Are Language Independent
- Patterns Define a Vocabulary
- Patterns Are Tried and Tested
- Patterns Are Designed for Collaboration
- Design Patterns Promote Good Design
- Design Patterns are Used By Popular Frameworks

.

_

Some Pattern Principles

The Gang of Four boiled this down into a principle: “favor composition over inheritance.” The patterns described ways in which objects could be combined at runtime to achieve a level of flexibility relationship impossible in an inheritance tree alone.

_

Composition and Inheritance

Inheritance is a powerful way of designing for changing circumstances or contexts. It can limit flexibility, however, especially when classes take on multiple responsibilities.

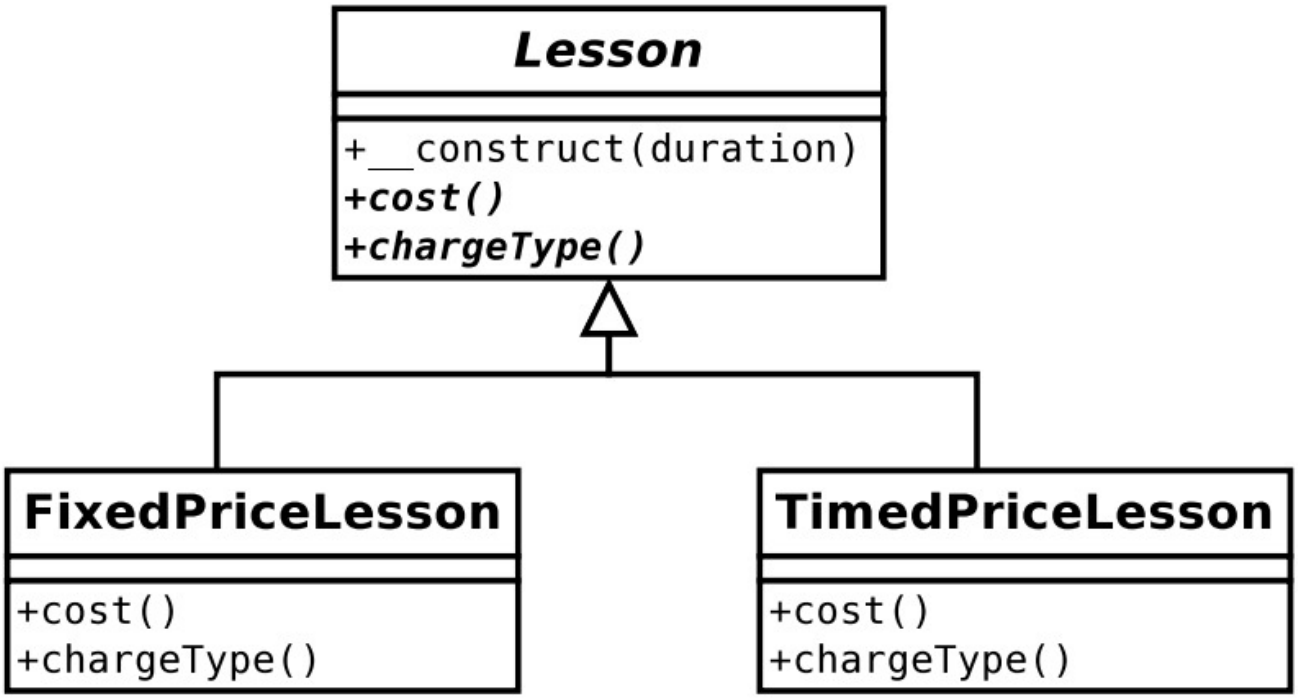


Figure 8-1. *A parent class and two child classes*

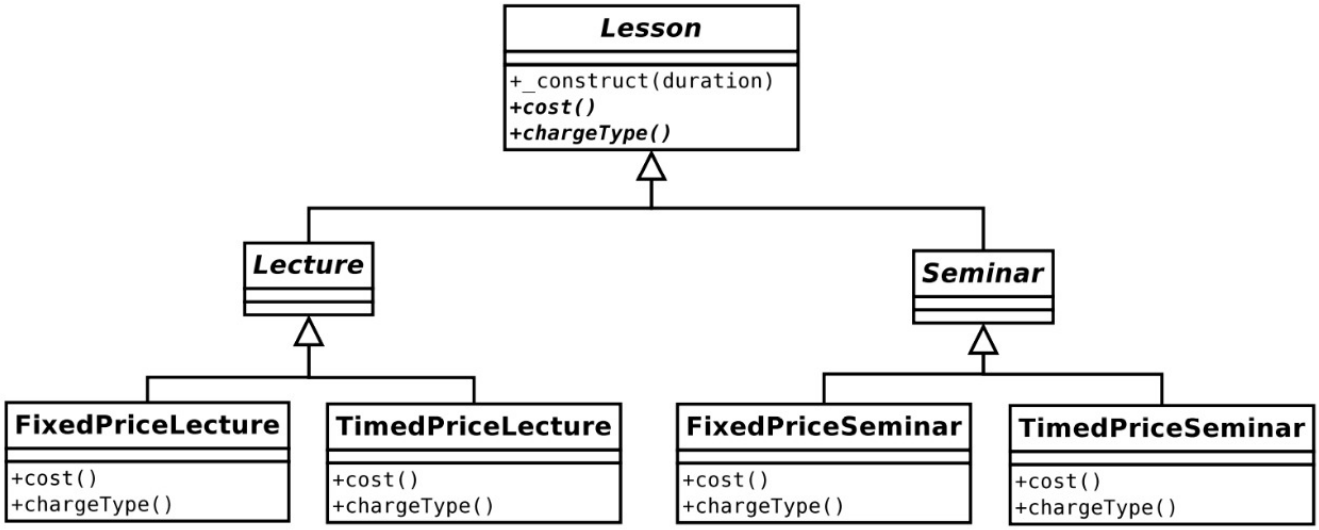


Figure 8-2. *A poor inheritance structure*

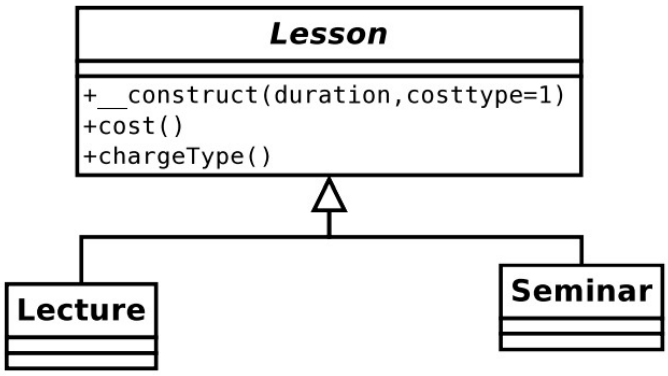


Figure 8-3. *Inheritance hierarchy improved by removing cost calculations from subclasses*

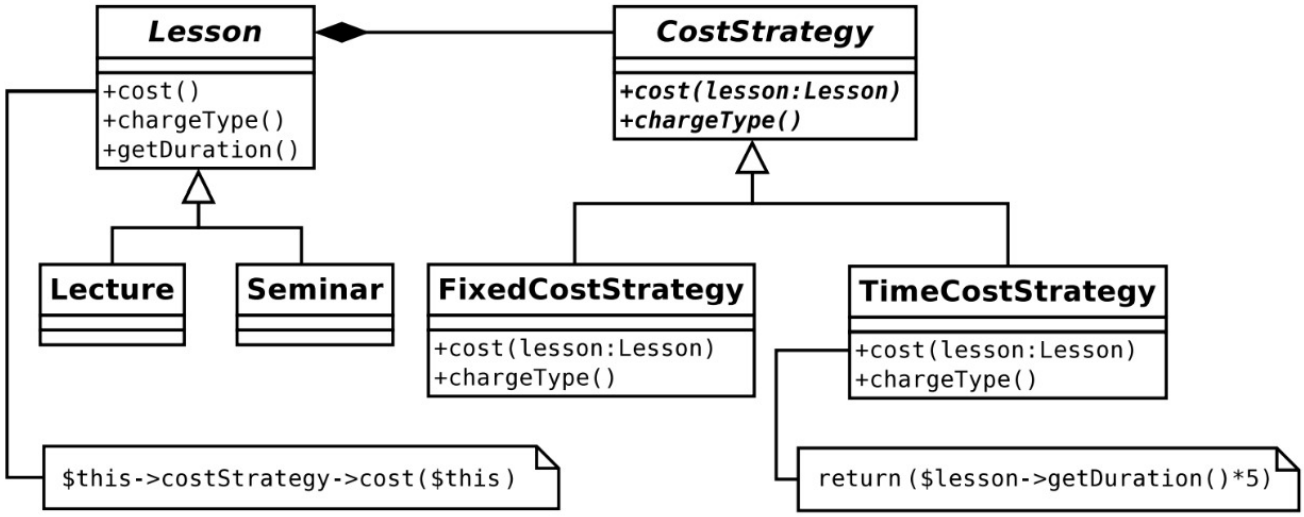


Figure 8-4. *Moving algorithms into a separate type*

_

Decoupling

That it makes sense to build independent components. A system with highly interdependent classes can be hard to maintain. A change in one location can require a cascade of related changes across the system.

Reusability is one of the key objectives of object-oriented design, and tight coupling is its enemy. You can diagnose tight coupling when you see that a change to one component of a system necessitates many changes elsewhere. You should aspire to create independent components, so that you can make changes without a domino effect of unintended consequences. When you alter a component, the extent to which it is independent is related to the likelihood that your changes will cause other parts of your system to fail.

The problem comes when such code is scattered throughout a project. For example: talking to databases is not the primary responsibility of most classes in a system, so the best strategy is to extract such code and group it together behind a common interface. In this way, you promote the independence of your classes. At the same time, by concentrating your gateway code in one place, you make it much easier to switch to a new platform without disturbing your wider system. This process, the hiding of implementation behind a clean interface, is known as encapsulation.

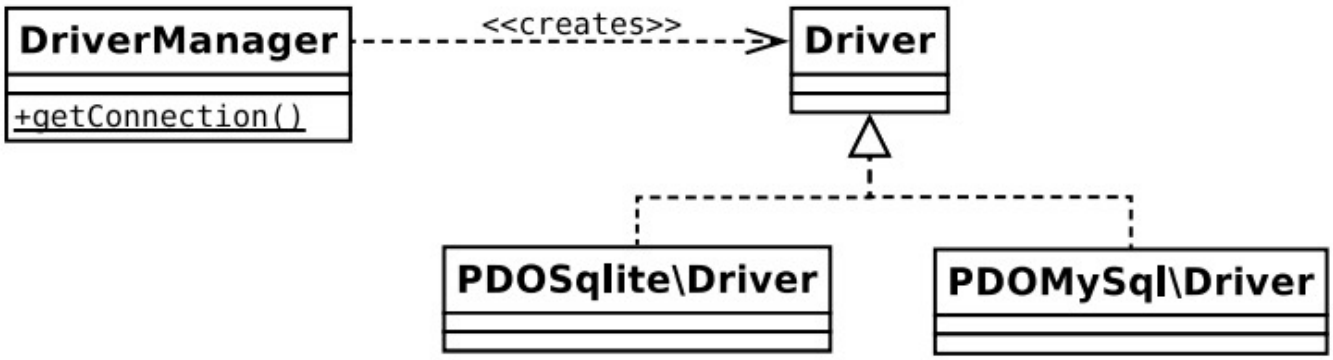


Figure 8-5. *The DBAL package decouples client code from database objects*

_

Loosening Your Coupling

Imagine, for example, that the Lesson system must incorporate a registration component to add new lessons to the system. As part of the registration procedure, an administrator should be notified when a lesson is added. The system's users can't agree whether this notification should be sent by mail or by text message. In fact, they're so argumentative that you suspect they might want to switch to a new mode of communication in the future. What's more, they want to be notified of all sorts of things, so that a change to the notification mode in one place will mean a similar alteration in many other places.

If you've hard-coded calls to a Mailer class or a Texter class, then your system is tightly coupled to a particular notification mode, just as it would be tightly coupled to a database platform by the use of a specialized database API.

```
// listing 08.12

class RegistrationMgr
{
    public function register(Lesson $lesson)
    {
        // do something with this Lesson
        // now tell someone
        $notifier = Notifier::getNotifier();
        $notifier->inform("new lesson: cost ({$lesson->cost()})");
    }
}
```

```
// listing 08.13

abstract class Notifier
{
    public static function getNotifier(): Notifier
    {
        // acquire concrete class according to
        // configuration or other logic
        if (rand(1, 2) === 1) {
            return new MailNotifier();
        } else {
            return new TextNotifier();
        }
    }

    abstract public function inform($message);
}
```

```
// listing 08.14

class MailNotifier extends Notifier
{
    public function inform($message)
    {
        print "MAIL notification: {$message}\n";
    }
}
```

```
// listing 08.15
```

```
class TextNotifier extends Notifier
{
    public function inform($message)
    {
        print "TEXT notification: {$message}\n";
    }
}
```

```
// listing 08.16

$lessons1 = new Seminar(4, new TimedCostStrategy());
$lessons2 = new Lecture(4, new FixedCostStrategy());
$mgr = new RegistrationMgr();
$mgr->register($lessons1);
$mgr->register($lessons2);
```

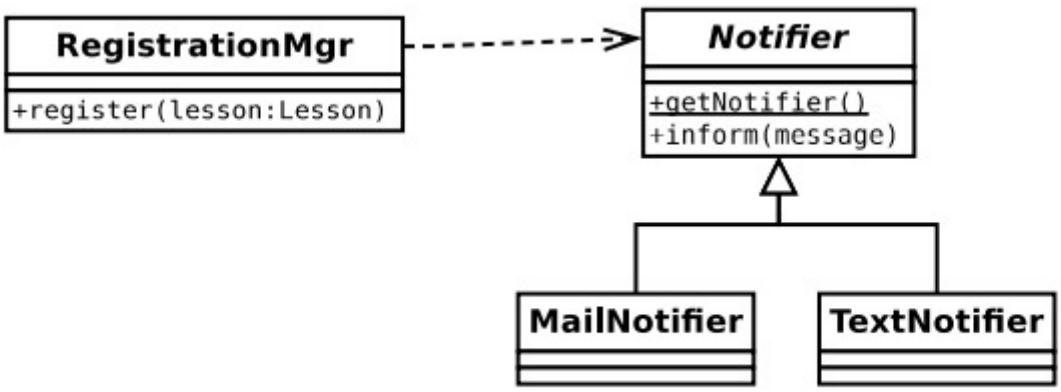


Figure 8-6. *The Notifier class separates client code from Notifier implementations*

_

Code to an Interface, Not to an Implementation

Client code can then require an object of the superclass's type rather than that of an implementing class, unconcerned by the specific implementation it is actually getting.

Parallel conditional statements, like the ones I rooted out from Lesson::cost() and Lesson::chargeType(), are a common sign that polymorphism is needed. They make code hard to maintain because a change in one conditional expression necessitates a change in its siblings. Conditional statements are occasionally said to implement a "simulated inheritance."

By placing the cost algorithms in separate classes that implement CostStrategy, I remove duplication. I also make it much easier should I need to add new cost strategies in the future.

From the perspective of client code, it is often a good idea to require abstract or general types in your methods' parameters. By requiring more specific types, you could limit the flexibility of your code at runtime.

Having said that, of course, the level of generality you choose in your argument hints is a matter of judgment. Make your choice too general, and your method may become less safe. If you require the specific functionality of a subtype, then accepting a differently equipped sibling into a method could be risky.

Still, make your choice of argument hint too restricted, and you lose the benefits of polymorphism.

_

The Concept that Varies

It's easy to interpret a design decision once it has been made, but how do you decide where to start?

The Gang of Four recommend that you actively seek varying elements in your classes and assess their suitability for encapsulation in a new type. Each alternative in a suspect conditional may be extracted to form a class that extends a common abstract parent. This new type can then be used by the class or classes from which it was extracted. This has the following effects:

- Focusing responsibility
- Promoting flexibility through composition
- Making inheritance hierarchies more compact and focused
- Reducing duplication

So how do you spot variation? One sign is the misuse of inheritance. This might include inheritance deployed according to multiple forces at one time (e.g., lecture/seminar and fixed/timed cost). It might also include subclassing on an algorithm where the algorithm is incidental to the core responsibility of the type. The other sign of variation suitable for encapsulation is, as you have seen, a conditional expression.

If you find that you are drowning in subclasses, it may be that you should be extracting the reason for all this subclassing into its own type. This is particularly the case if the reason is to achieve an end that is incidental to your type's main purpose.

Given a type UpdatableThing, for example, you may find yourself creating FtpUpdatableThing, HttpUpdatableThing, and FileSystemUpdatableThing subtypes. The responsibility of your type, though, is to be a thing that is updatable—the mechanism for storage and retrieval is incidental to this purpose. Ftp, Http, and FileSystem are the things that vary here, and they belong in their own type—let's call it UpdateMechanism. UpdateMechanism will have subclasses for the different implementations. You can then add as many update mechanisms as you want without disturbing the UpdatableThing type, which remains focused on its core responsibility.

Notice also that I have replaced a static compile-time structure with a dynamic runtime arrangement here, bringing us (as if by accident) back to our first principle: "Favor composition over inheritance."

> Encapsulate the Concept that Varies

_

Patternitis

One problem for which there is no pattern is the unnecessary or inappropriate use of patterns. This has earned patterns a bad name in some quarters. Because pattern solutions are neat, it is tempting to apply them wherever you see a fit, whether they truly fulfill a need or not.

The eXtreme Programming (XP) methodology offers a couple of principles that might apply here. The first is, “You aren’t going to need it” (often abbreviated to YAGNI). This is generally applied to application features, but it also makes sense for patterns.

> Do the simplest thing that works.

_

The Patterns

```
Patterns for Generating Objects
Patterns for Organizing Objects and Classes
Task-Oriented Patterns
Enterprise Patterns
Database Patterns
```

> I examined some of the principles that underpin many design patterns. I looked at the use of composition to enable object combination and recombination at runtime, resulting in more flexible structures than would be available using inheritance alone. I also introduced you to decoupling, the practice of extracting software components from their context to make them more generally applicable. Finally, I reviewed the importance of interface as a means of decoupling clients from the details of implementation.

.

_

Domain-Driven Design

Before delving into the details, it's good to take a bird's-eye view of the philosophy so you can get a sense of what DDD is really all about.

The Problem Space

Before you can develop a solution, you must understand the problem. DDD emphasizes the need to focus on the business problem domain: its terminology, the core reasons behind why the software is being developed, and what success means to the business. The need for the development team to value domain knowledge just as much as technical expertise is vital to gain a deeper insight into the problem domain and to decompose large domains into smaller subdomains.

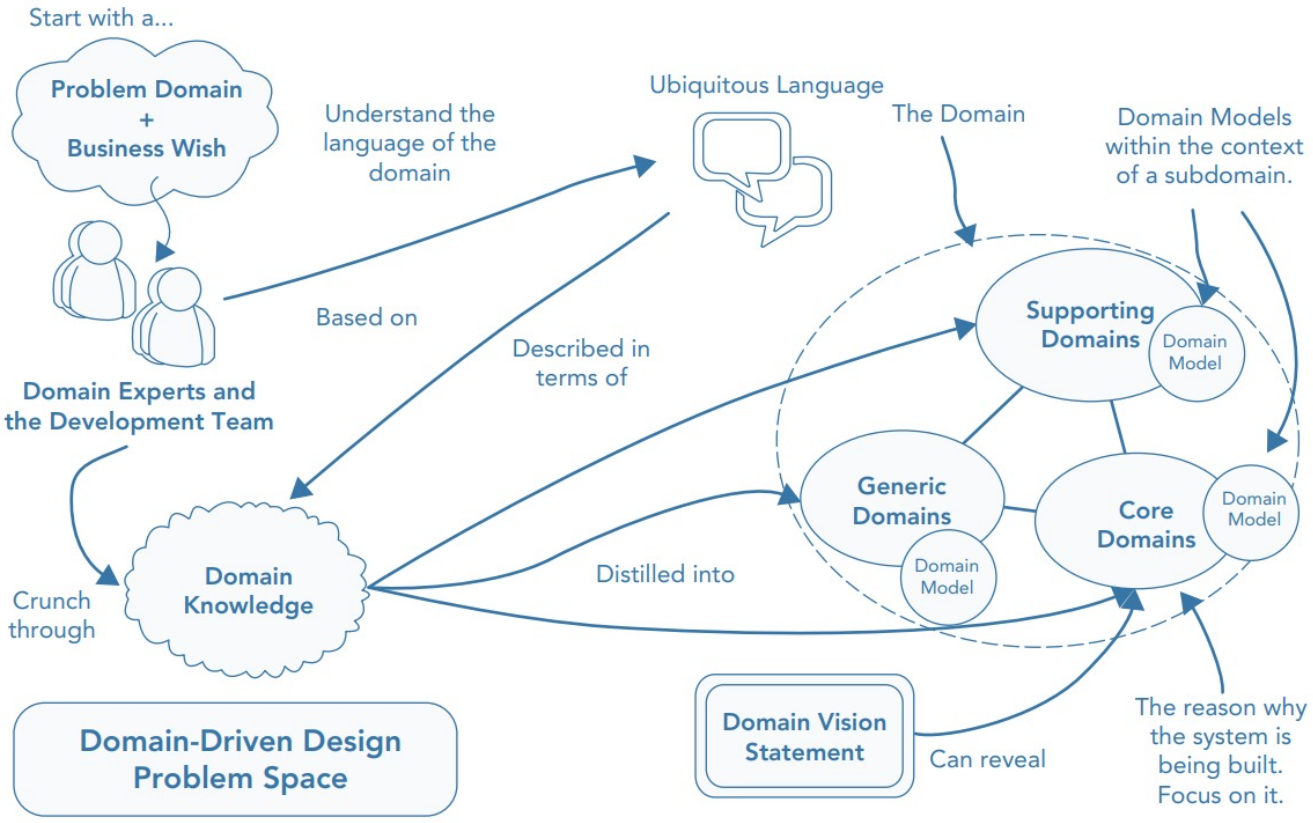


FIGURE I-1: A blueprint of the problem space of DDD.

The Solution Space

When you have a sound understanding of the problem domain, strategic patterns of DDD can help you implement a technical solution in synergy with the problem space. Patterns enable core parts of your system that are crucial to the success of the product to be protected from the generic areas. Isolating integral components allows them to be modified without having a rippling effect throughout the system.

Core parts of your product that are sufficiently complex or will frequently change should be based on a model. The tactical patterns of DDD along with Model-Driven Design will help you create a useful model of your domain in code. A model is the home to all of the domain logic that enables your application to fulfill business use cases. A model is kept separate from technical complexities to enable business rules and policies to evolve. A model that is in synergy with the problem domain will enable your software to be adaptable and understood by other developers and business experts.

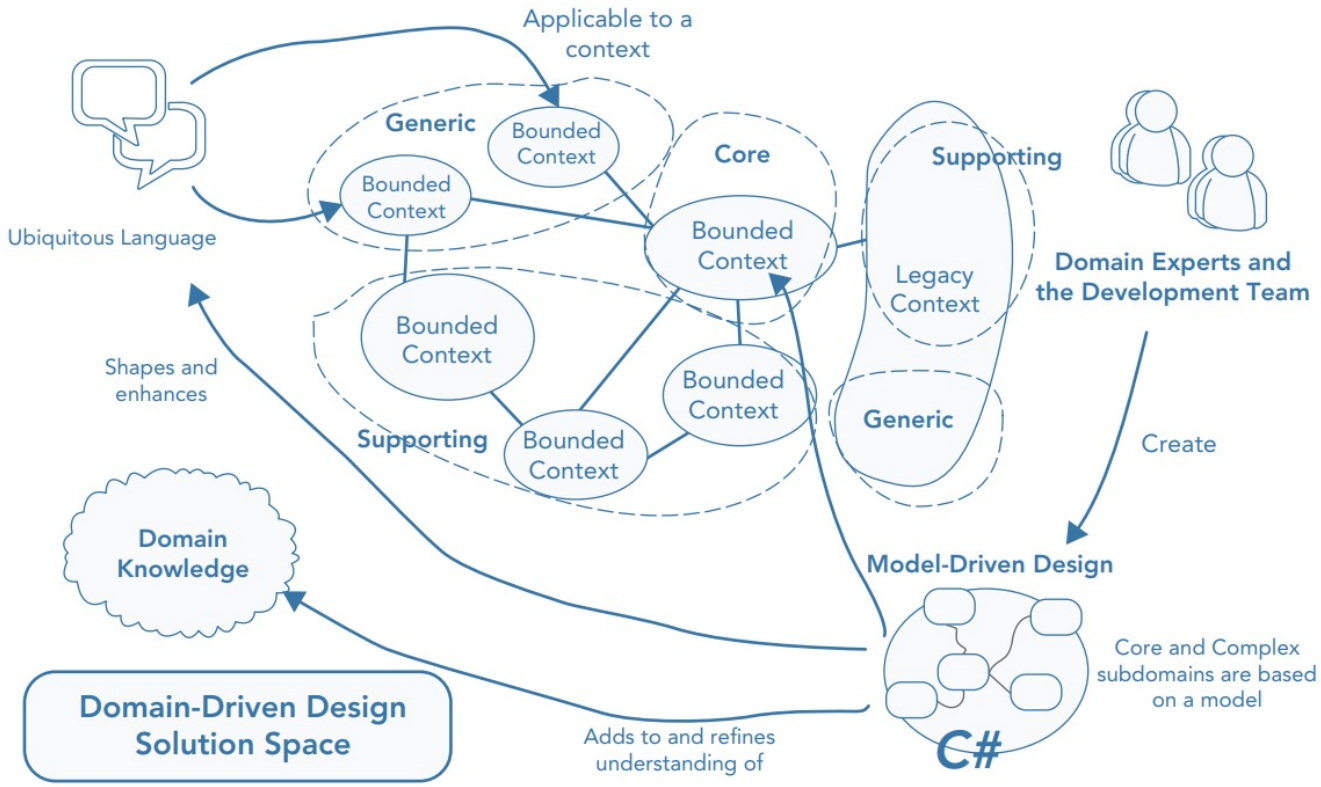


FIGURE I-2: A blueprint of the solution space of Domain-Driven Design.

_

Complexity in Software

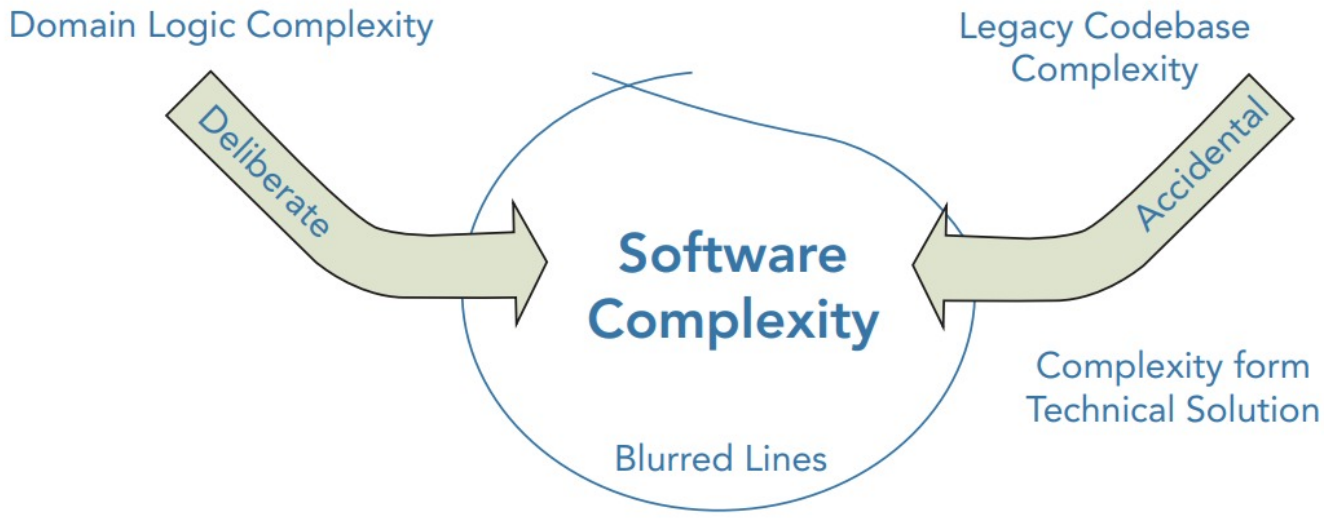


FIGURE 1-1: Complexity in software.

_

Applying The Strategic Patterns of Domain-Driven Design

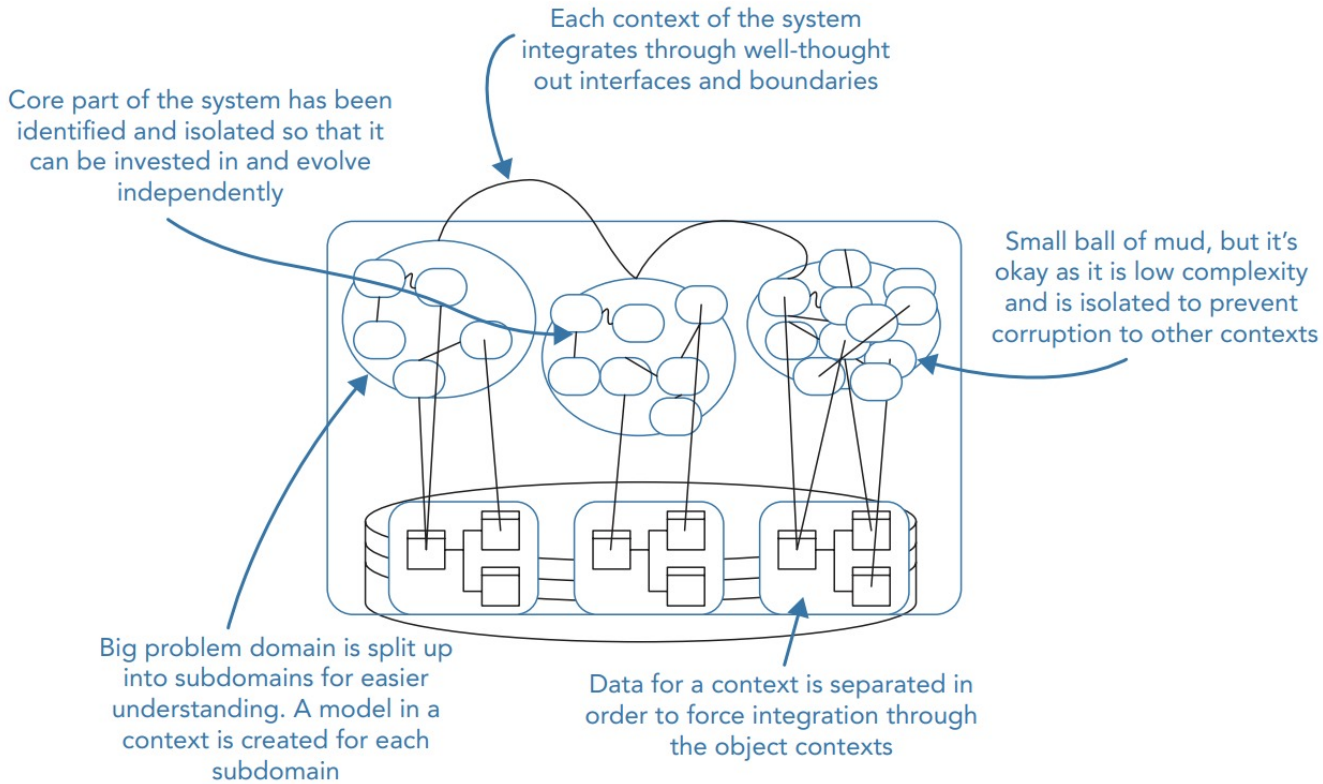


FIGURE 1-3: Applying the strategic patterns of Domain-Driven Design.

_

Knowledge Crunching

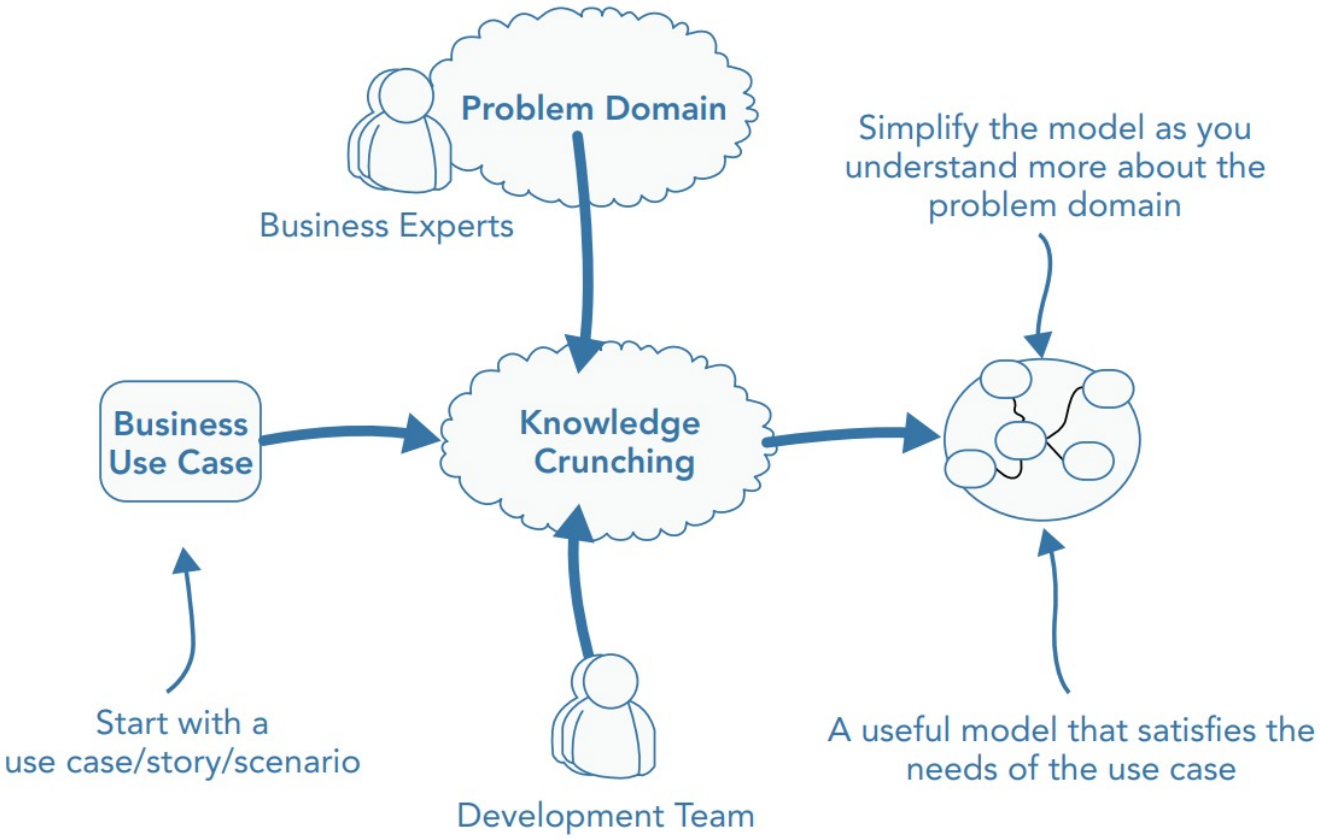


FIGURE 2-1: Knowledge crunching.

_

The Role of Domain Model

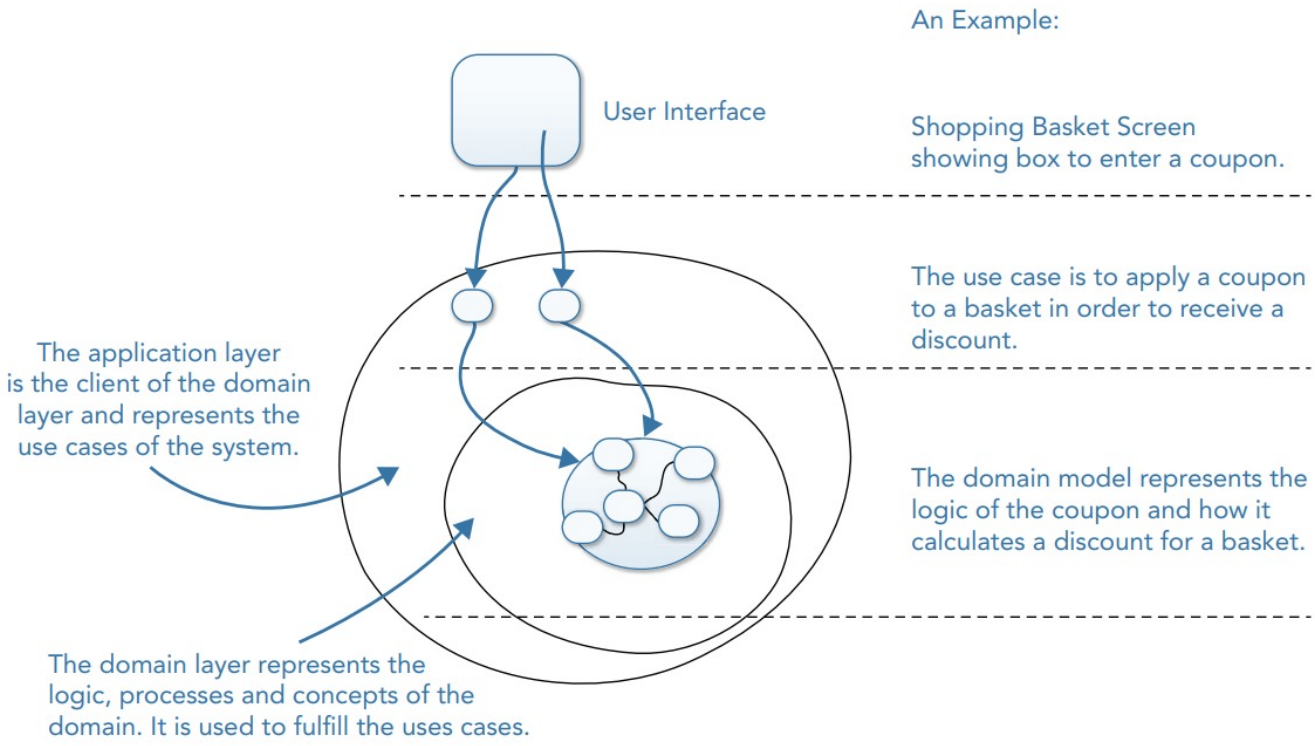


FIGURE 4-1: The role of a domain model.

_

The Domain Versus The Domain Model

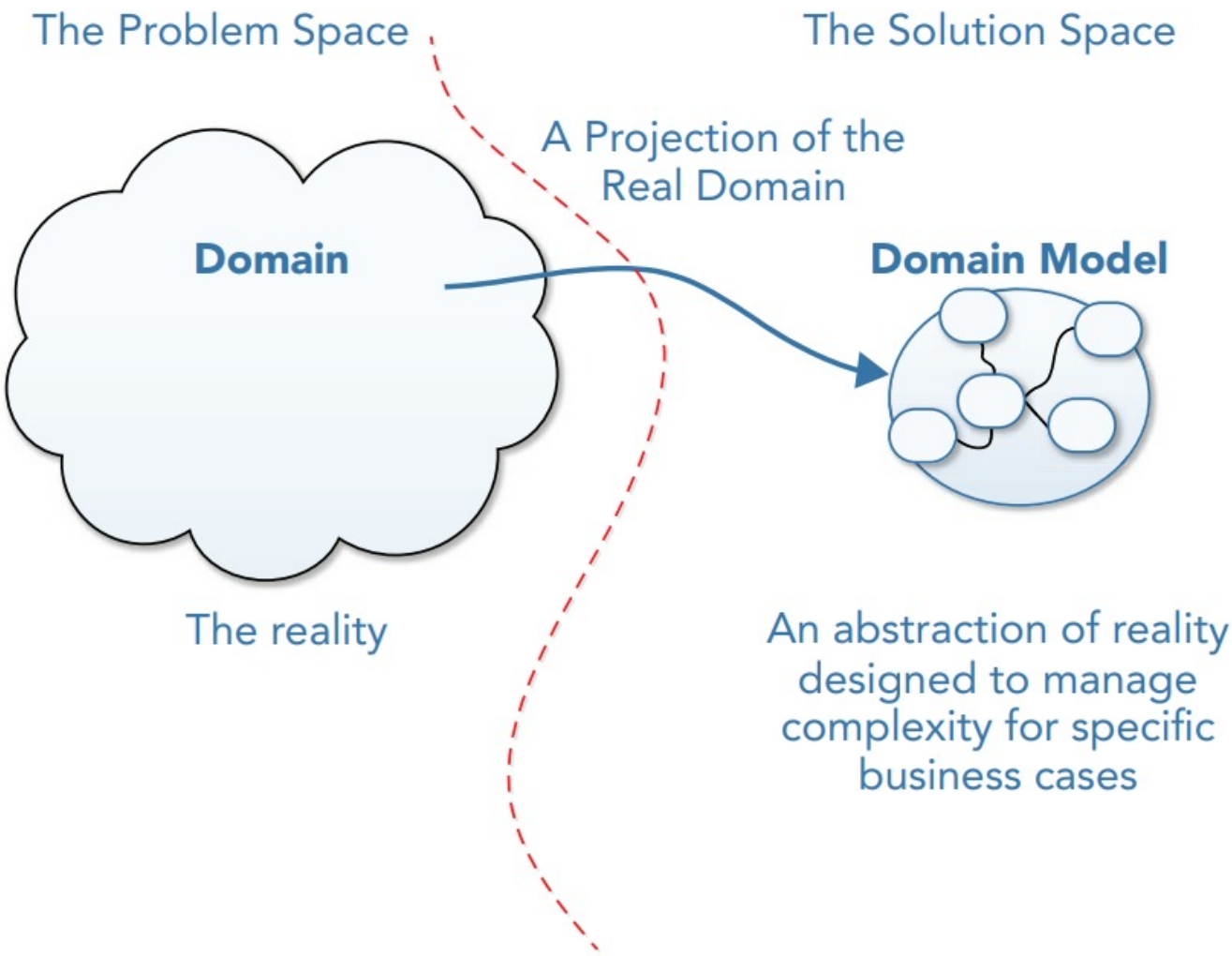


FIGURE 4-2: The domain versus the domain model.

_

The Binding between The Code and Analysis Model

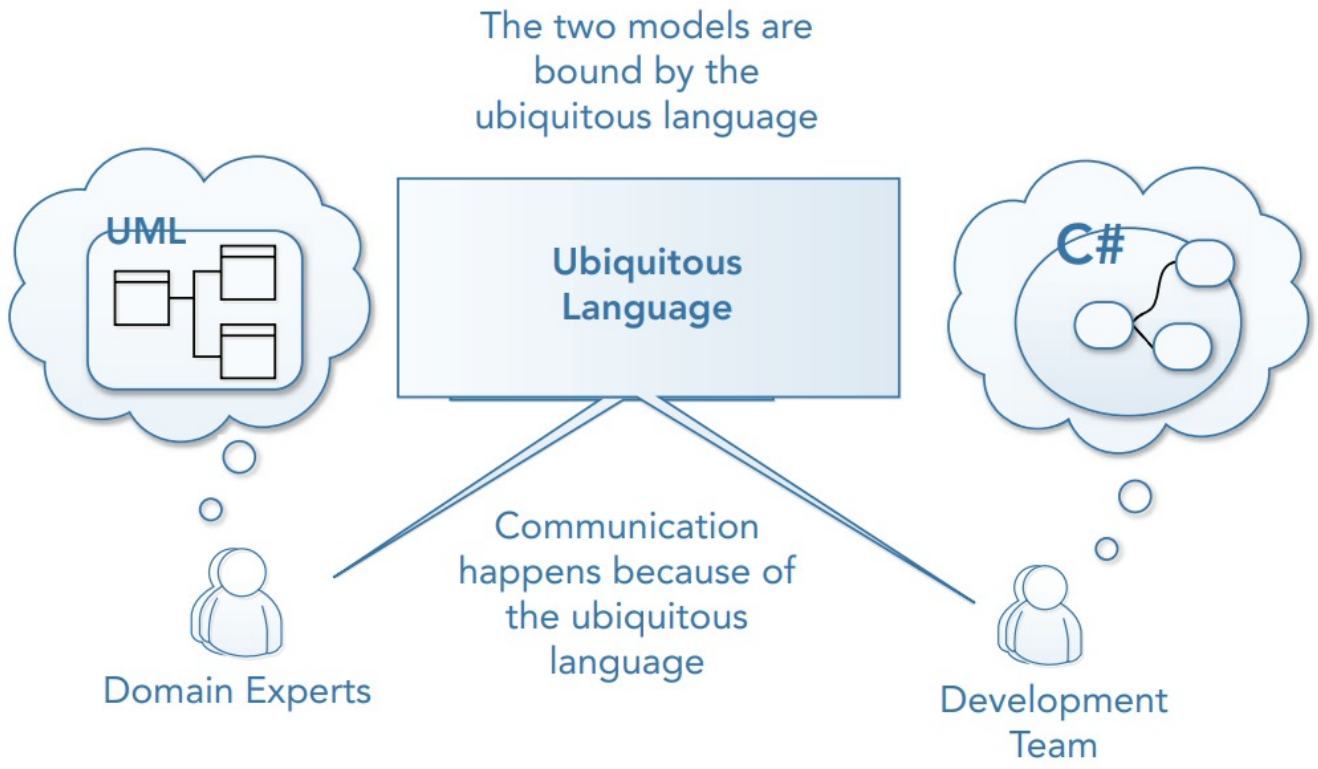


FIGURE 4-3: The binding between the code and analysis model.

_

A Model will Grow in Complexity

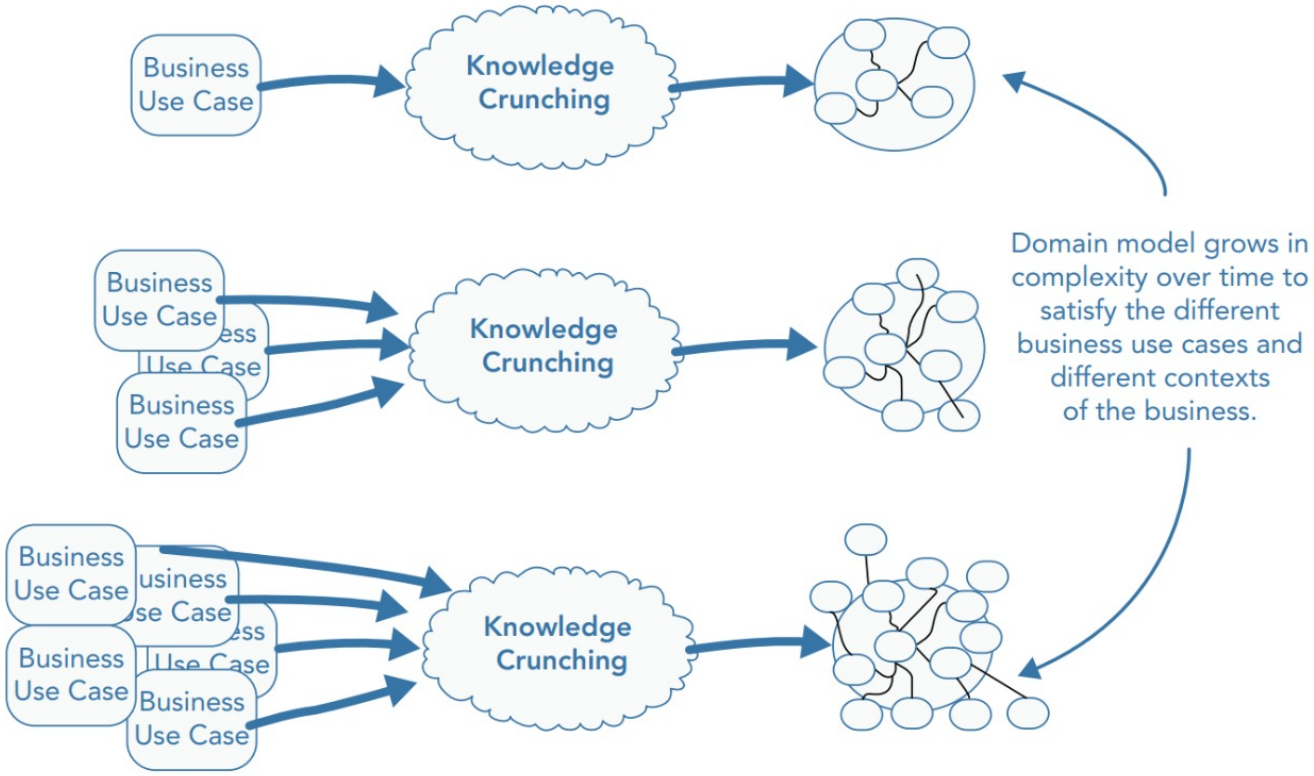


FIGURE 6-1: A model will grow in complexity.

_

Domain Terms Mean Different Things in Different Contexts

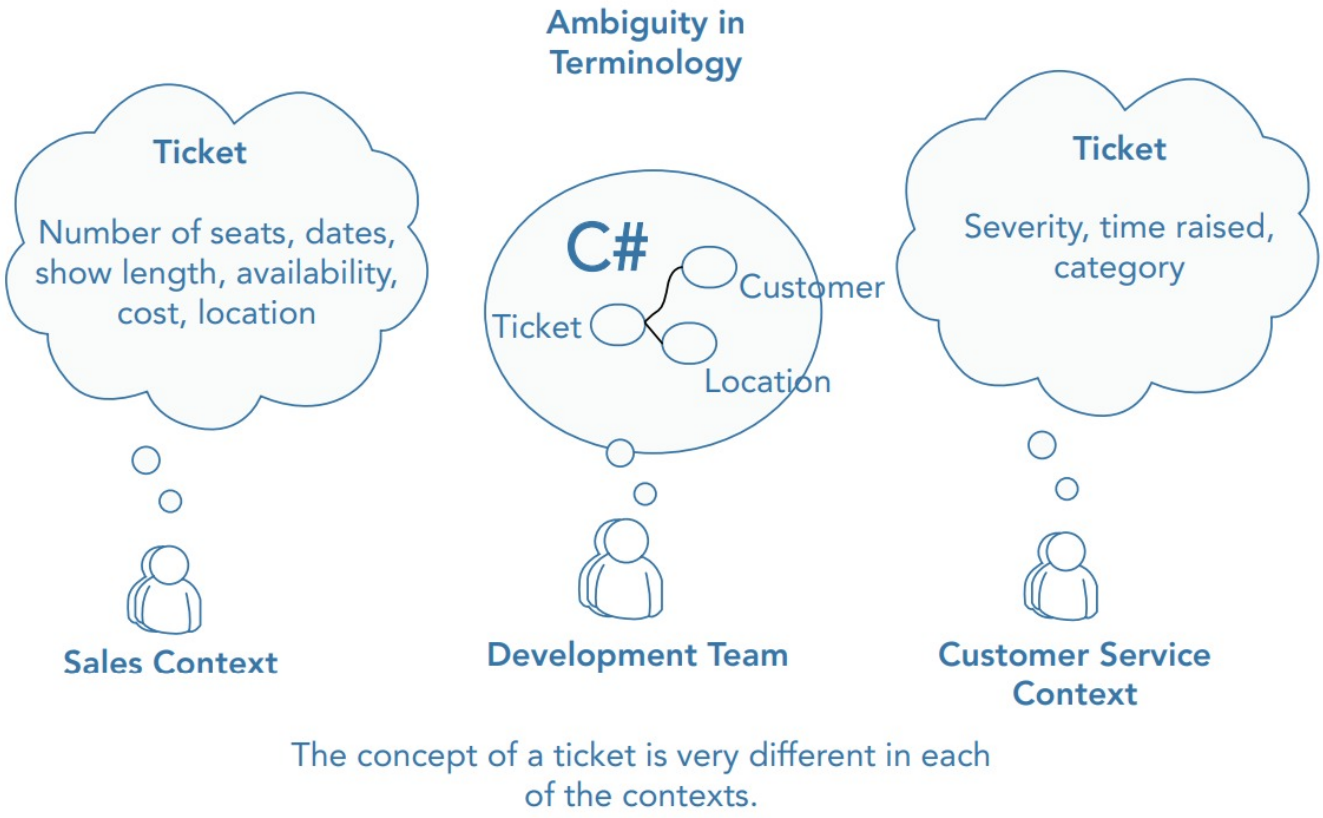


FIGURE 6-3: Domain terms mean different things in different contexts.

_

The Same Concept should be Understood within Different Contexts

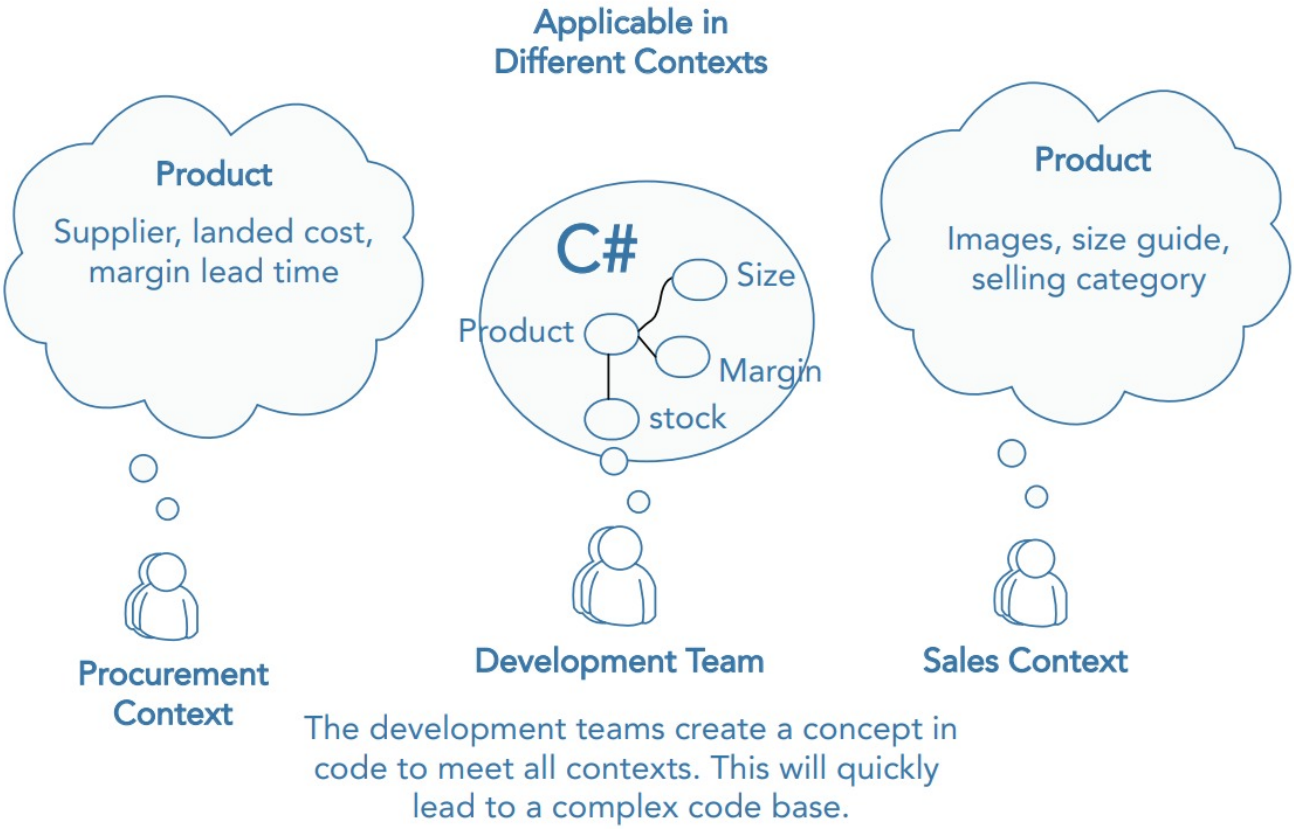


FIGURE 6-4: The same concept should be understood within different contexts.

_

A Single View of An Entity in The Domain for All Subdomains can Quicky Become A Problem

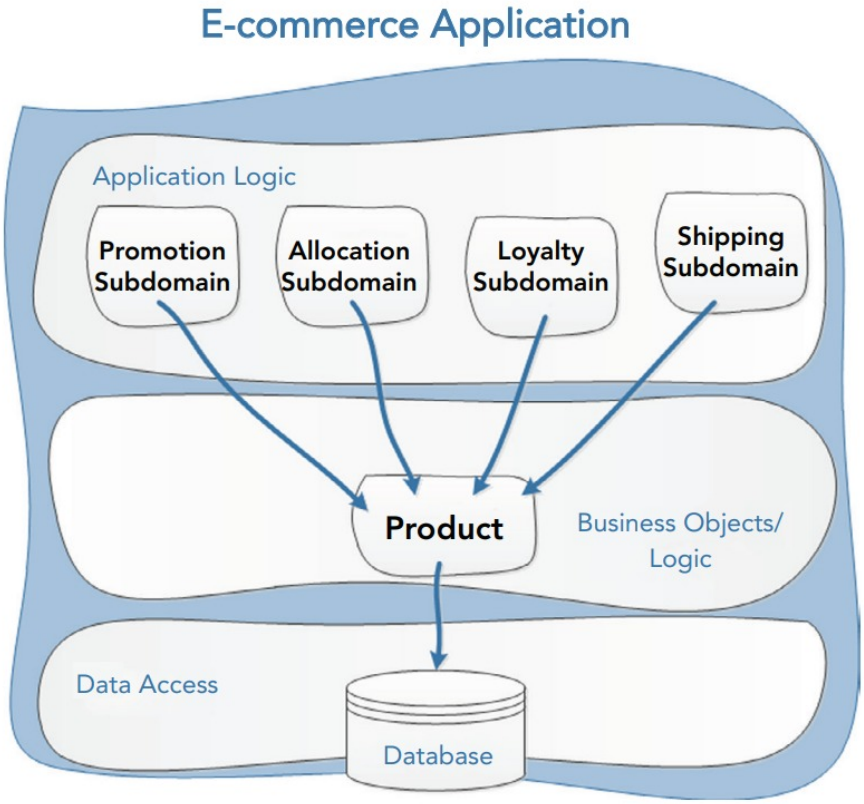


FIGURE 6-5: A single view of an entity in the domain for all subdomains can quickly become a problem.

_

The Product, An Implementation of The God Object Antipattern

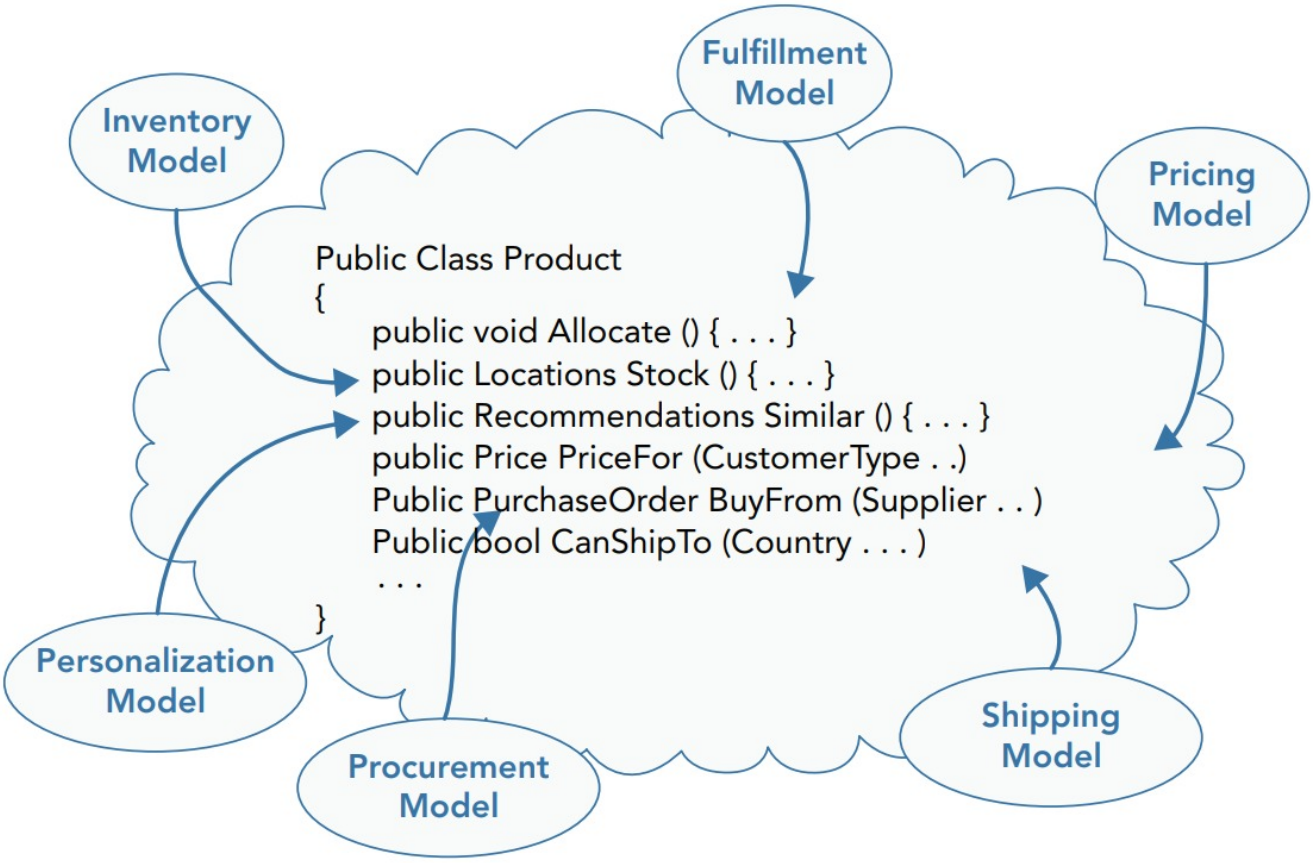


FIGURE 6-6: The product, an implementation of the god object antipattern.

_

Putting Terms into Context and Identifying Multiple Models

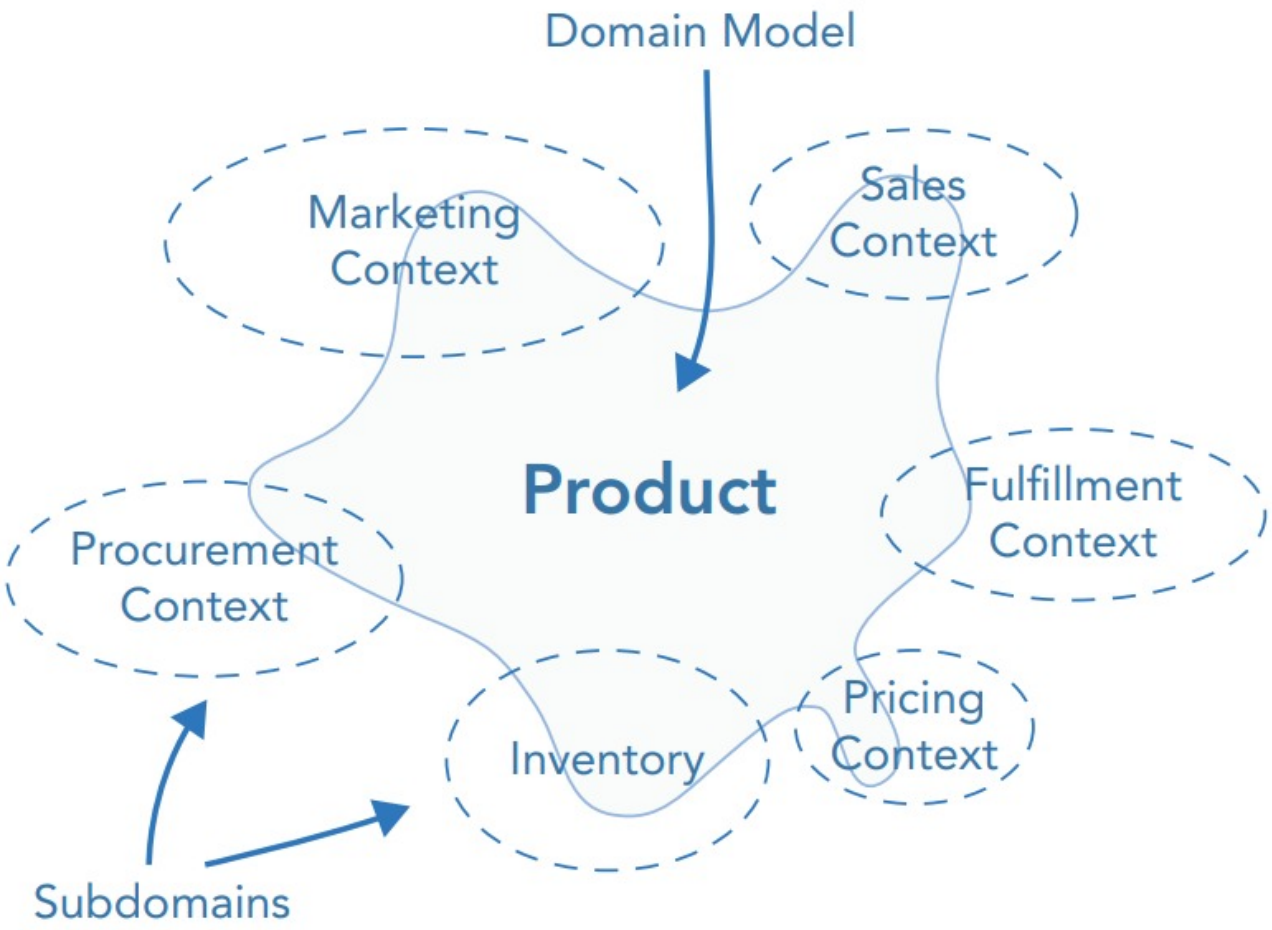


FIGURE 6-8: Putting terms into context and identifying multiple models.

Consider that a Bounded Context is a conceptual boundary around a system. The Ubiquitous Language inside a boundary has a specific contextual meaning. Concepts outside of this context can have different meanings.

_

Define Each Model within Its Own Context

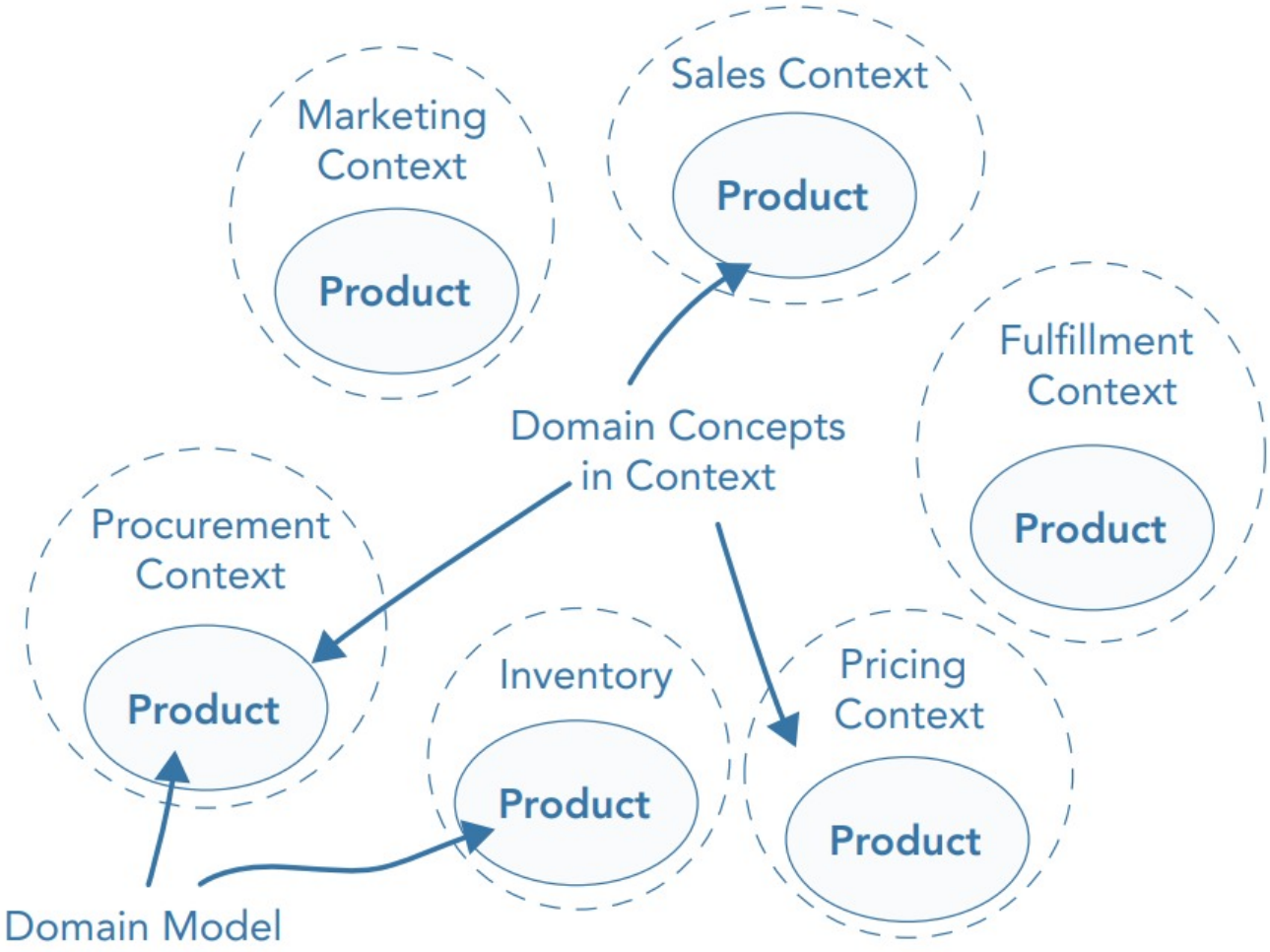


FIGURE 6-9: Define each model within its own context.

_

The Product Class in Different Contexts

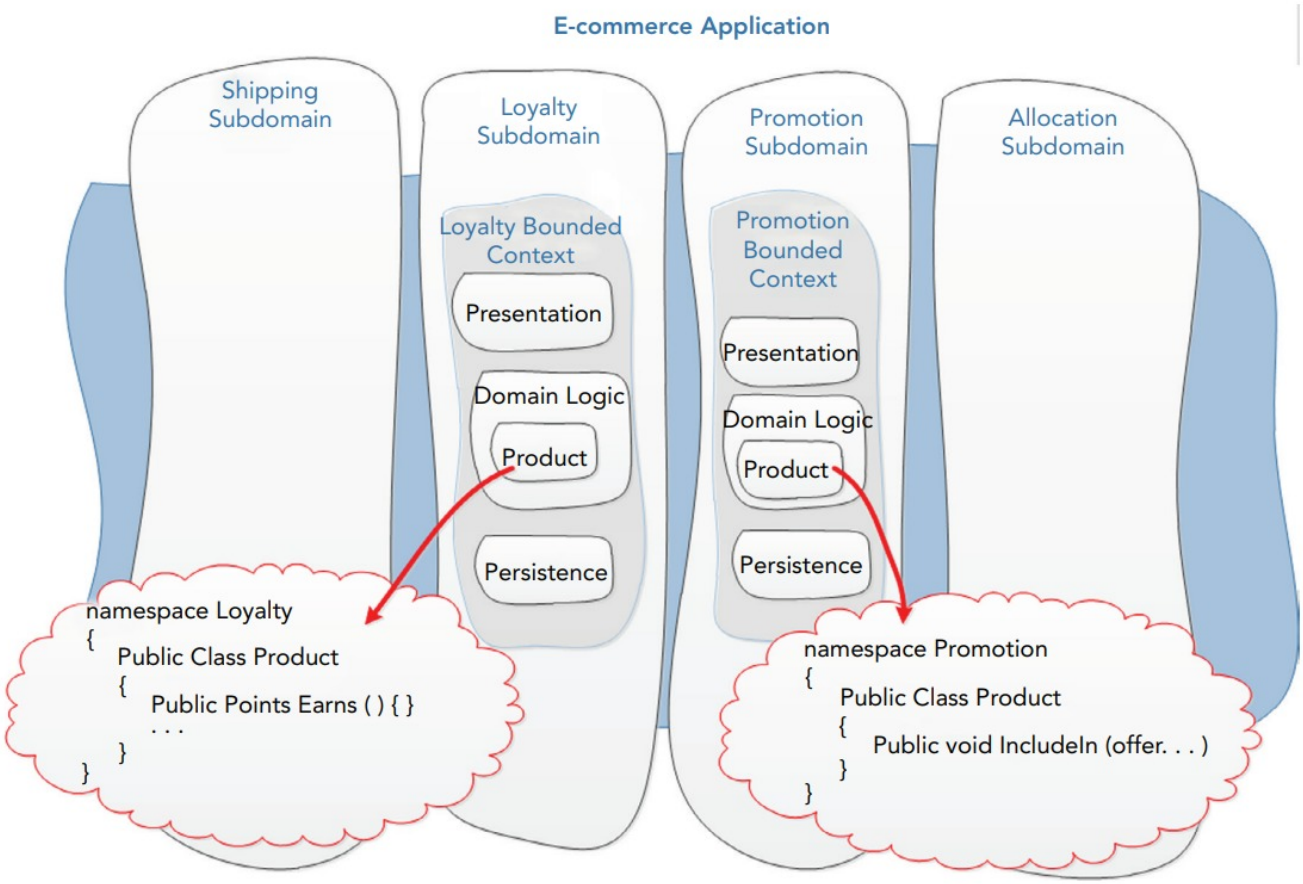


FIGURE 6-11: The Product class in different contexts.

_

The Anatomy of A Bounded Context

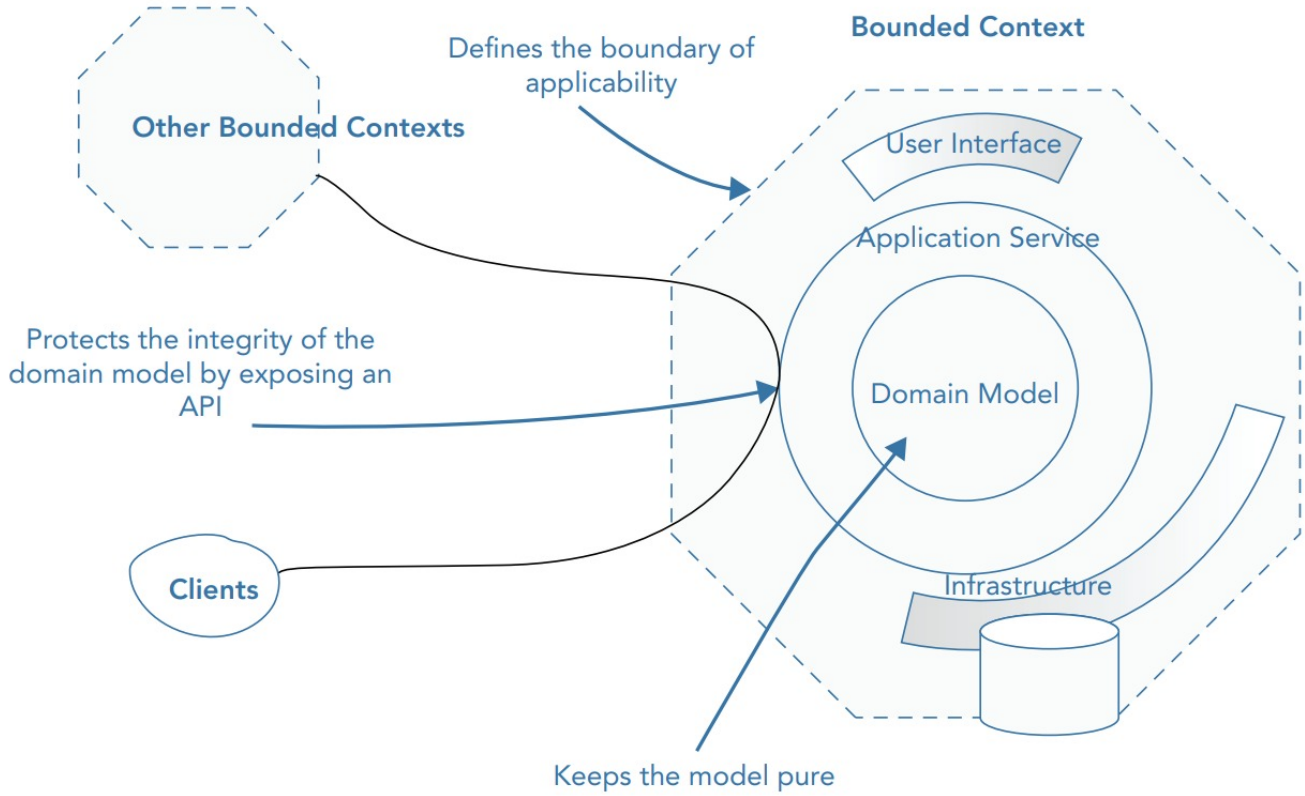


FIGURE 6-13: The anatomy of a bounded context.

_

The Technical Integration on A Context Map

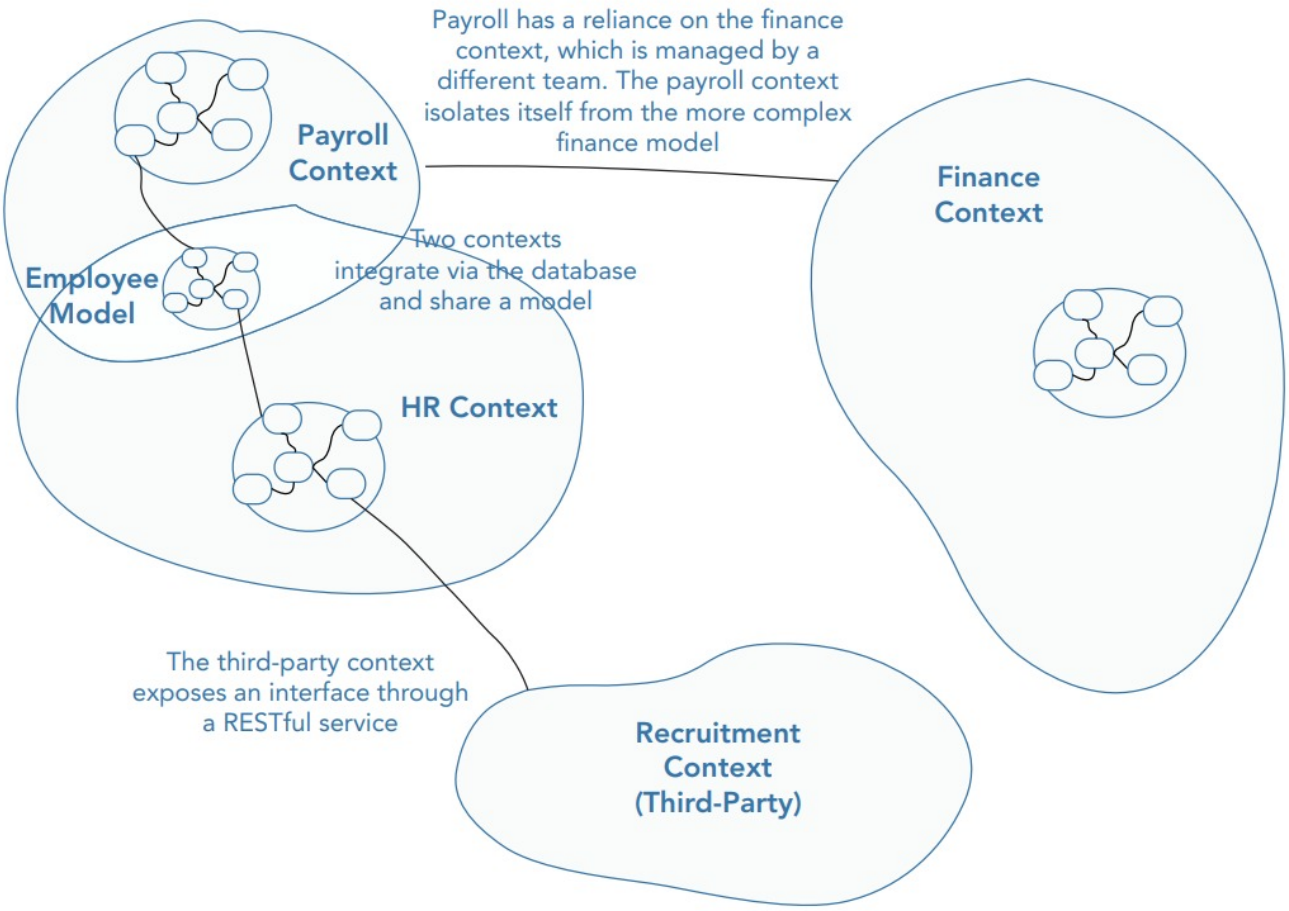


FIGURE 7-2: The technical integration on a context map.

_

Dependency Inversion within A Layered Architecture

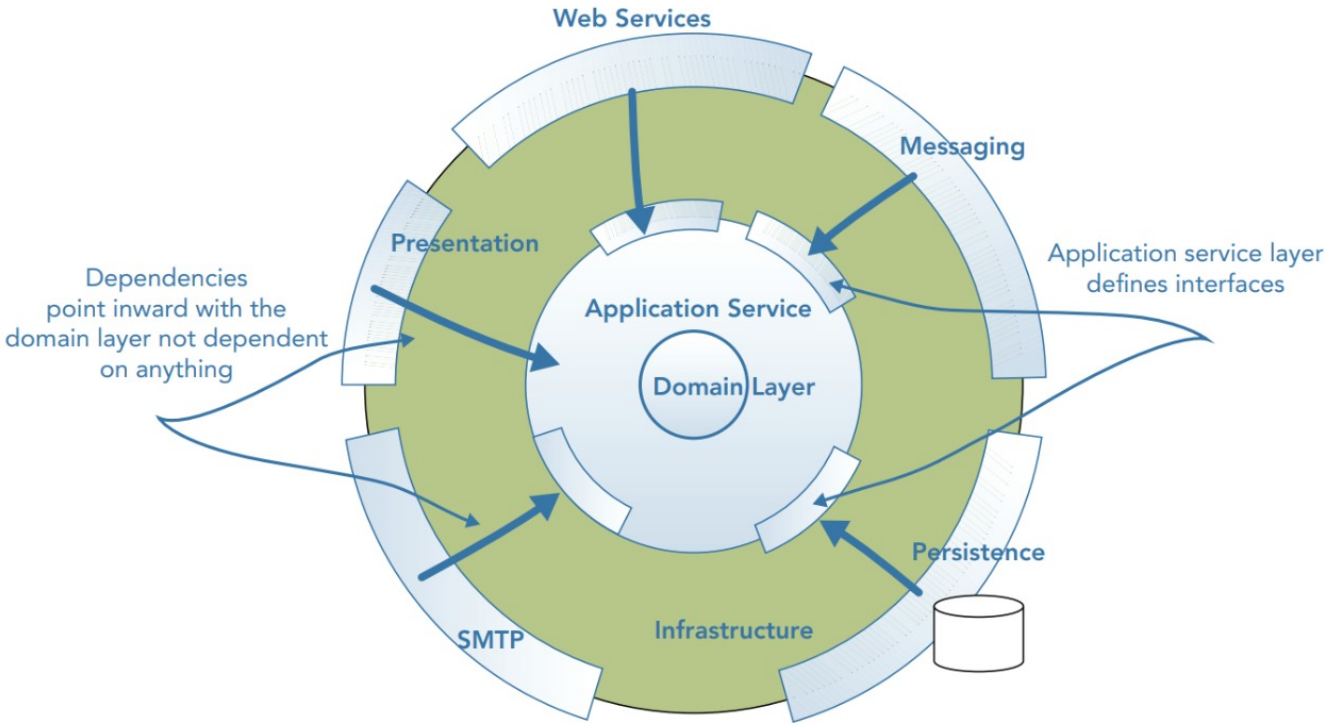


FIGURE 8-2: Dependency inversion within a layered architecture.

_

The Process of DDD

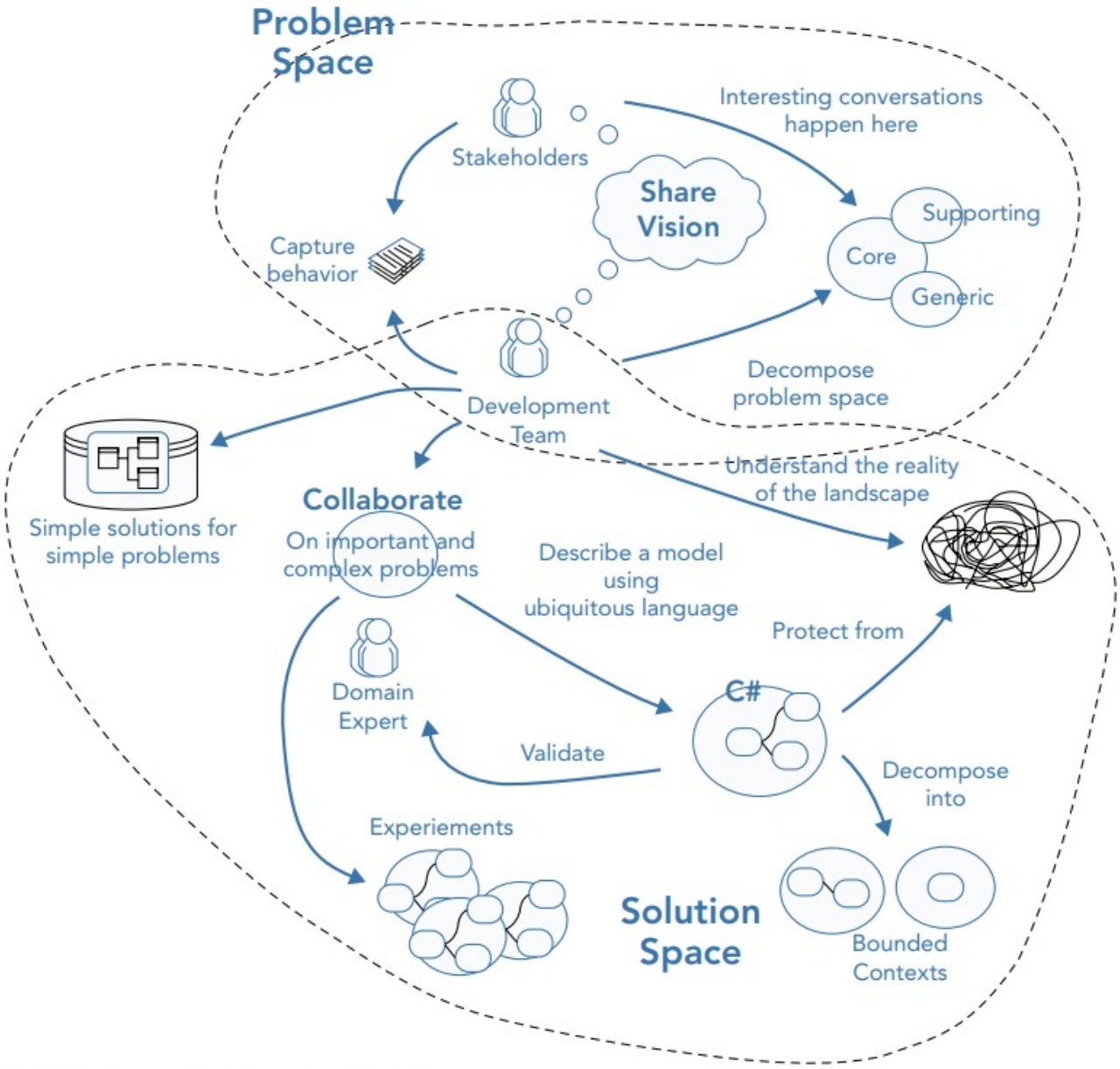


FIGURE 10-1: The process of DDD.

_

Tactical Patterns, Domain Model Building Blocks

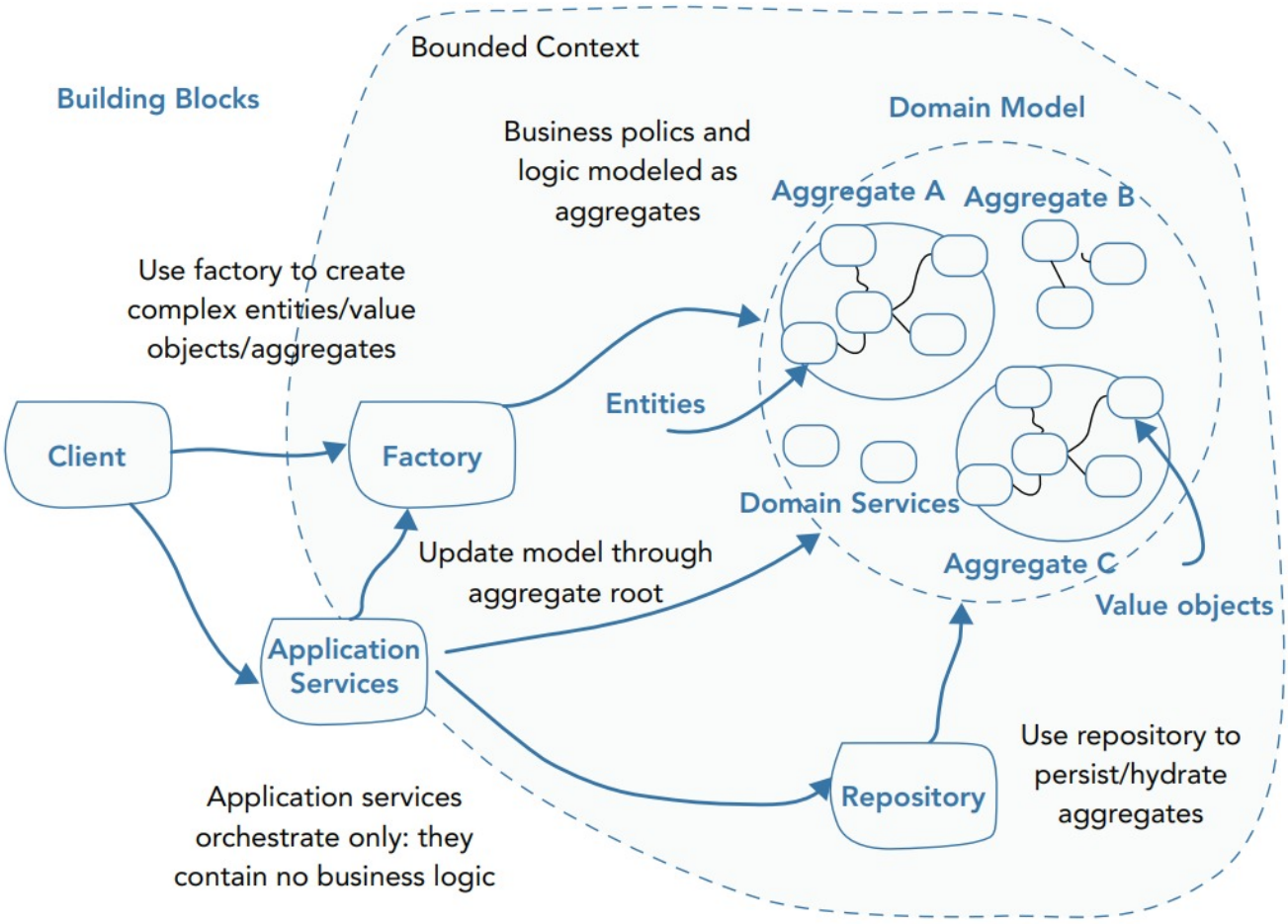


FIGURE 14-1: Tactical patterns—domain model building blocks.

_

Composing a Web Page with HTML Provided By Bounded Contexts

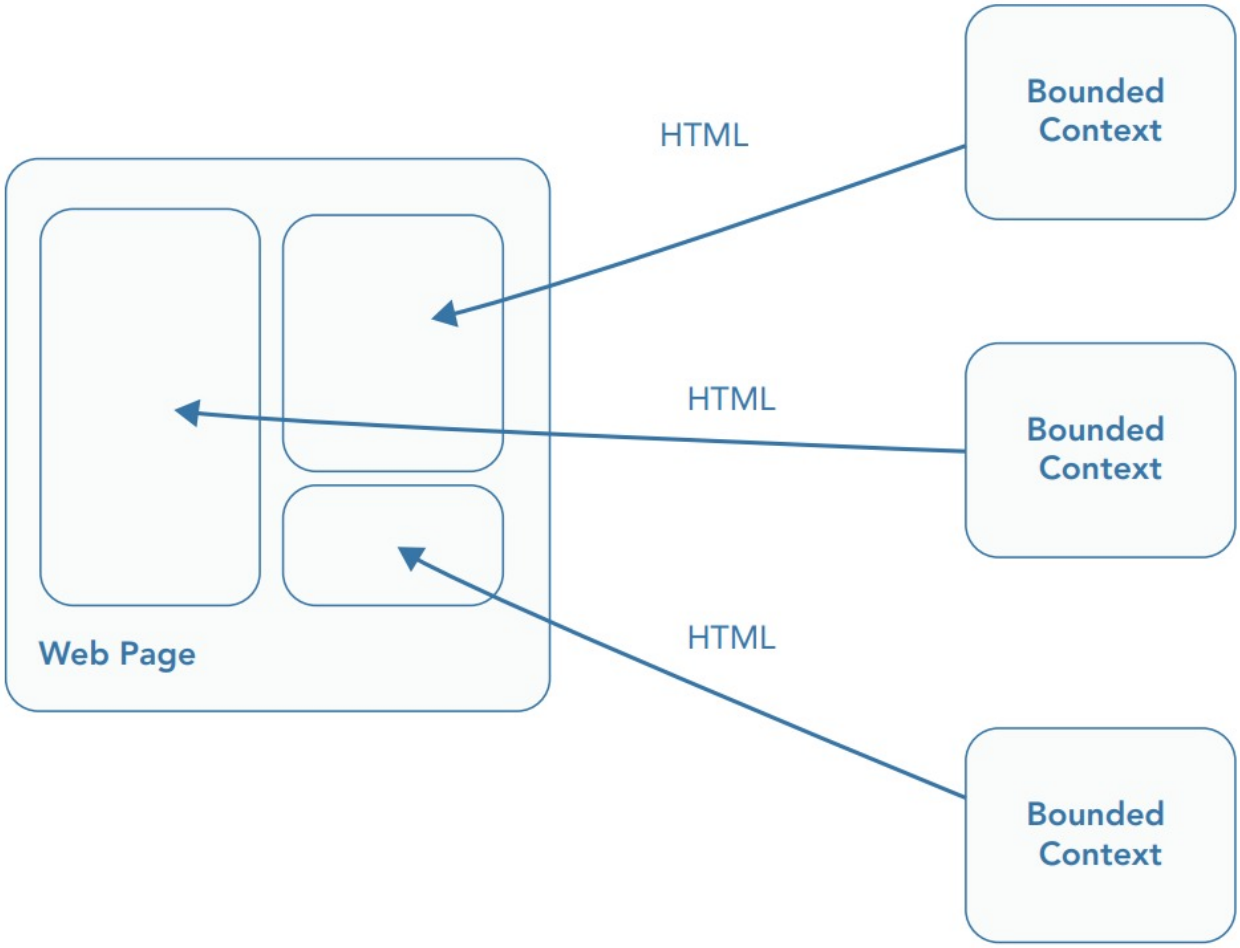


FIGURE 23-3: Composing a web page with HTML provided by bounded contexts.

_

The Design for This Example

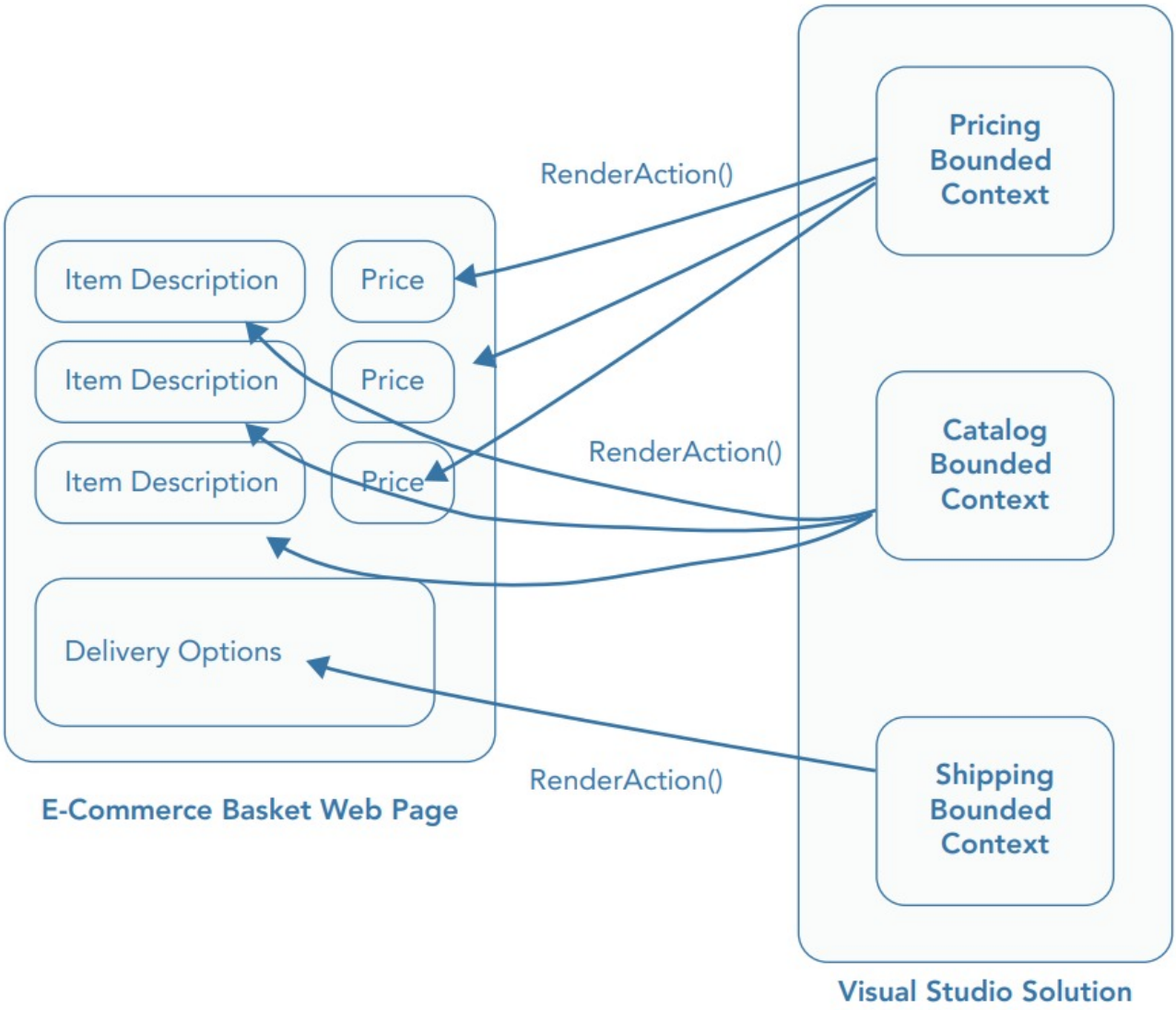
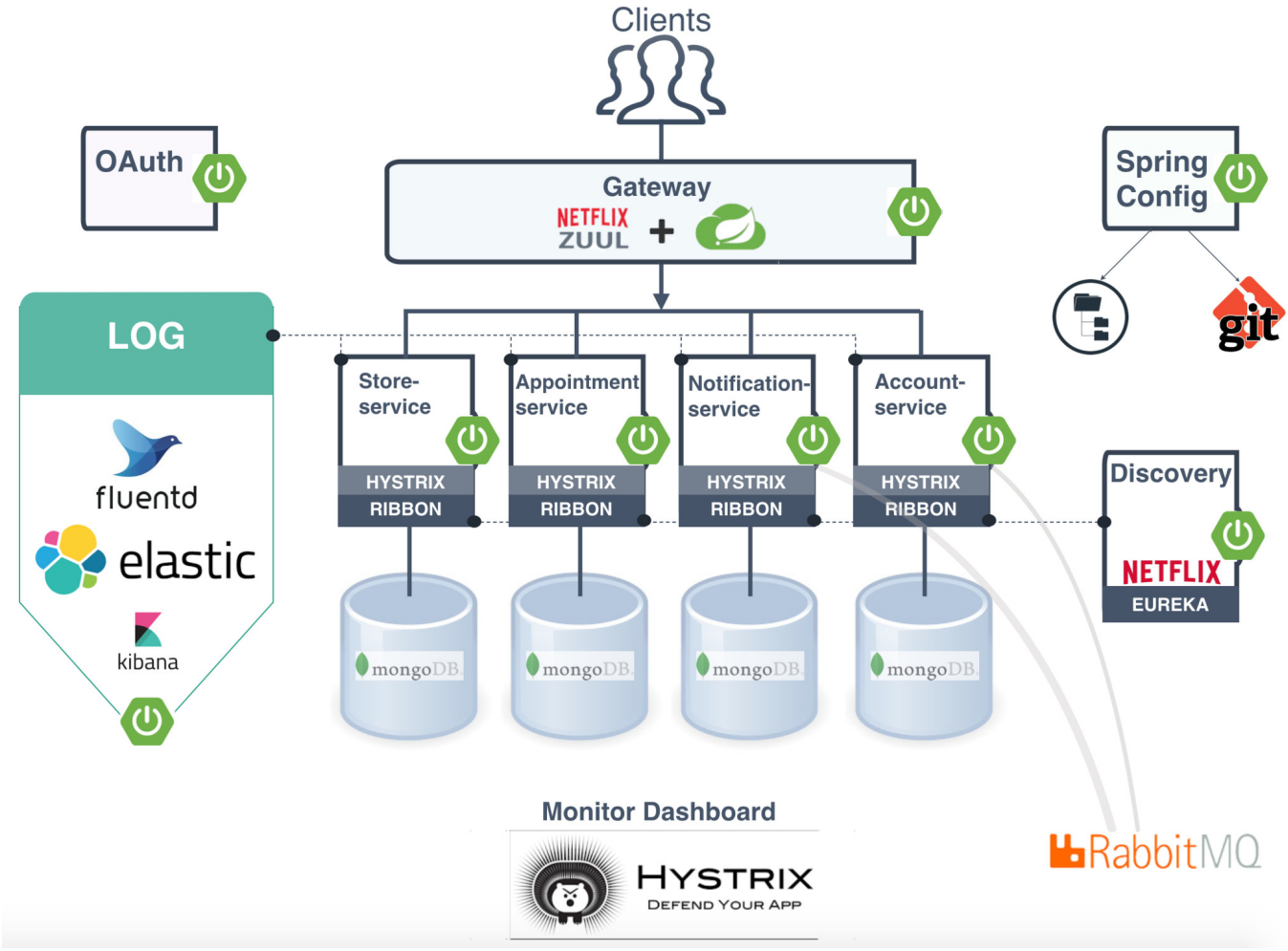


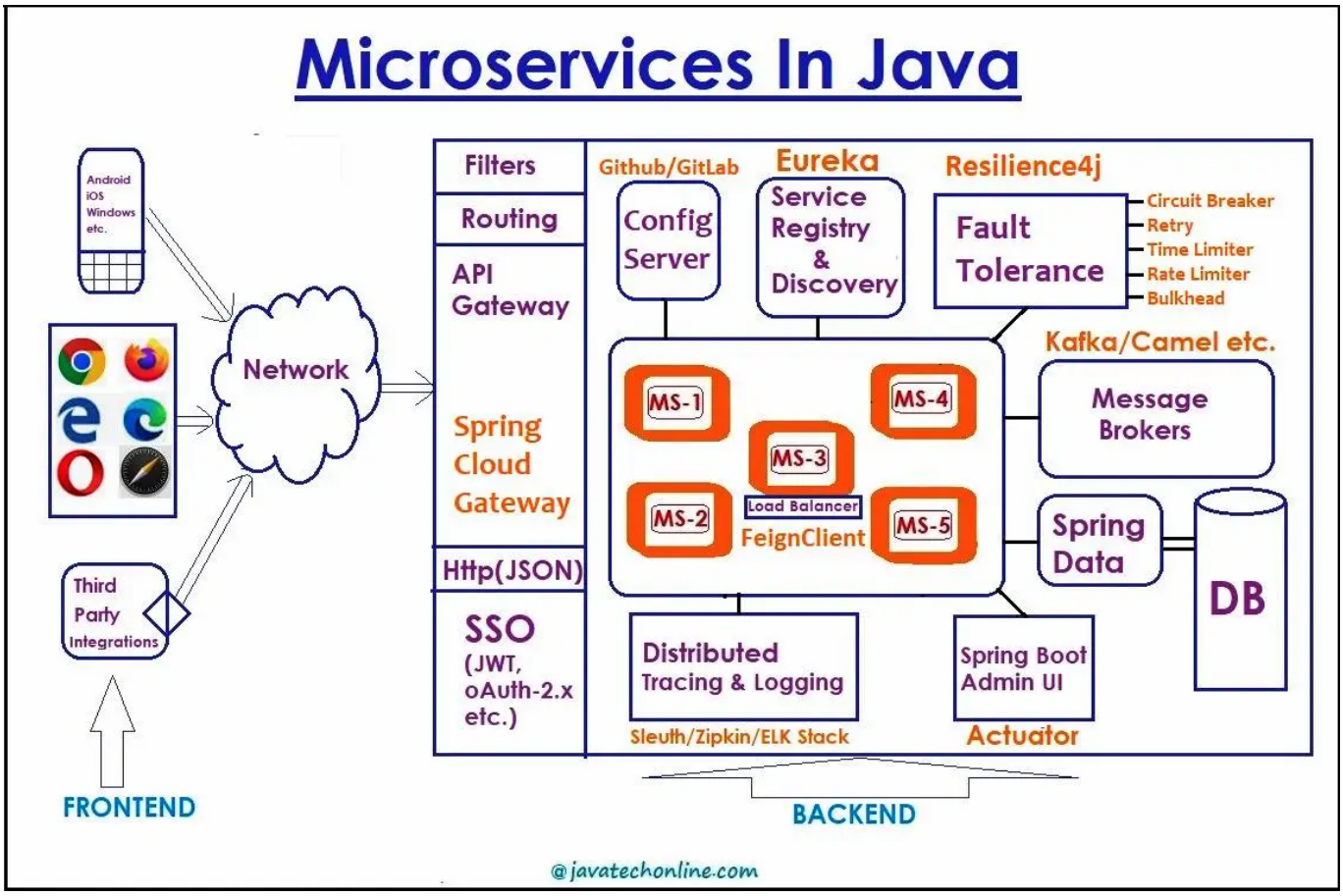
FIGURE 23-7: The design for this example.

.

_

Microservice




Microservices In Java
Android
iOS
Windows
etc.
Network
Third
Party
Integrations
FRONTEND
Filters
Routing
API
Gateway
Spring
Cloud
Gateway
Http(JSON)
SSO
(JWT,
oAuth-2.x
etc.)
Github/GitLab
Config
Server
Eureka
Service
Registry
&
Discovery
Resilience4j
Fault
Tolerance
Circuit Breaker
Retry
Time Limiter
Rate Limiter
Bulkhead
Kafka/Camel etc.
Message
Brokers
MS-1
MS-2
MS-3
MS-4
MS-5
Load Balancer
FeignClient
Spring
Data
DB
Distributed
Tracing & Logging
Sleuth/Zipkin/ELK Stack
Spring Boot
Admin UI
Actuator
BACKEND
@javatechonline.com

.

_

Related Topics

- Agile Management
- DevOps
- Containers Orchestration
- Dashboard Design (Business Analysis)
- System Analysis
- System Performance
- System Security
- UI/UX Analysis & Design