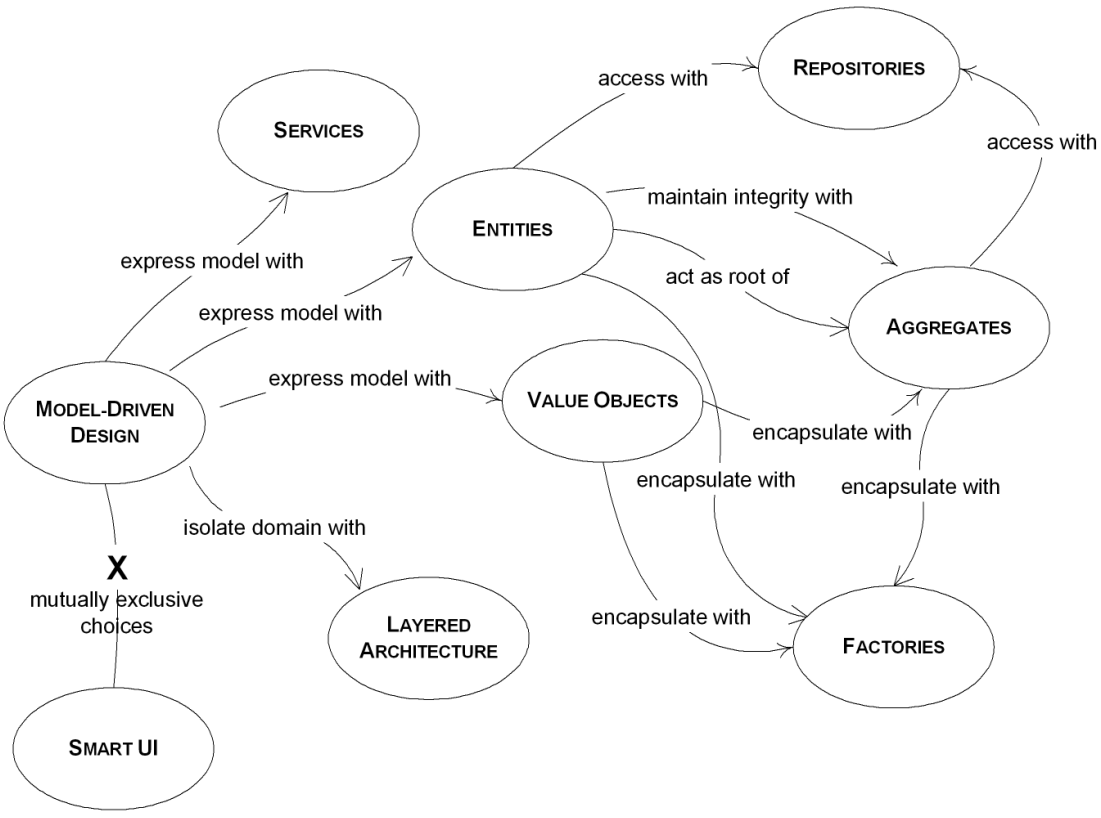
Domain-Driven Design *Quickly*

日本語版



by Abel Avram & Floyd Marinescu

edited by: Dan Bergh Johnsson, Vladimir Gitlevich

日本語訳　徳武 聡

InfoQ *Enterprise Software Development Series*

本文書は、Abel Avram 氏と Floyd Marinescu 氏による文書「Domain-Driven Design Quickly」を、徳武 聡氏により日本語へ訳したものです。

原文の最新版は InfoQ.com からダウンロードすることができます。
http://infoq.com/minibooks/domain-driven-design-quickly

◆オリジナル文書（英語）の印刷版は、下記 Web サイトで購入することができます($30.00)。
http://www.lulu.com/content/325231

◆翻訳者の紹介
徳武 聡(とくたけ さとし) 氏
プログラマ。主に.Net プラットフォームでの開発を行っている。
現在は C#/ASP.NET で Web アプリケーションを構築している。

目次

はじめに: なぜドメイン駆動設計なのか

私が初めてドメイン駆動設計についての話を聞き、Eric Evansに会ったのは、Bruce Eckelがある山の頂上で2005年に開催したアーキテクトの集まりに参加したときでした。その集まりには私が尊敬している方々が参加していました。Martin Fowler, Rod Johnson, Cameron Purdy, Randy Stafford, Gregor Hohpeといった方々です。

その集まりに参加した人々は、ドメイン駆動設計の思想にとても感銘を受けていたようで、より詳しく学びたがっていました。誰もが、この思想がもっと主流になってほしい、と願っているのがありありと感じられました。そして参加した人々が話し合っている様々な技術的問題について議論するために、Ericが提示したドメインモデルの使い方や、また彼が特定の技術を大げさに解説する代わりに、業務ドメインを大変重視していることに気づいたとき、私はドメイン駆動設計の考え方はこのコミュニティに必要不可欠なものだということがすぐにわかりました。

私たちエンタープライズ分野の開発コミュニティ、とりわけWebシステムの開発コミュニティに属する者は、きちんとしたオブジェクト指向ソフトウエア開発を邪魔するような誇大広告に長年にわたって毒されてきました。例えばJavaのコミュニティでは、1999年から2004年の間にEJBとオブジェクト/コンテナモデルの大げさな宣伝のせいで、優れたドメインモデリングをおこなわなくなってしまいました。幸いにも技術動向の変化や、ソフトウエア開発コミュニティに蓄積された経験のおかげで、私たちは伝統的なオブジェクト指向のパラダイムへと回帰しています。しかし、コミュニティにはオブジェクト指向をエンタープライズ分野へどのように適用すればいいのかについてはっきりとした考え方がありません。だからこそ私はDDDが重要だと考えるのです。

残念なことに、熟練のアーキテクトの小さなグループの外では、DDDはほとんど知られていないようです。InfoQがこの本の執筆を依頼したのは、このような現状を改善するためです。

DDDの要約と基本思想の紹介を、短くて素早く読めるかたちで公表して、InfoQのサイトから自由にダウンロードできるようにし、また安価で小さな本として売り出すことで、DDDの考え方が主流になることを望みます。（InfoQ Japan注：安価で小さい本は、日本語版は提供していません。）

この本では新しい概念を紹介することはありません。DDDとは何なのかについて正確な要約を試みたのがこの本です。内容のほとんどはEric Evansが書いたこの分野の原典に負っていますし、またJimmy NilssonのApplying DDDのような書

籍や、フォーラム上でのDDDに関する議論も同じくらい情報源にしています。この本の内容はDDDの基本の速習講座です。Ericの本に載っている数多くのサンプルやケーススタディ、あるいはJimmyの本に載っている実習課題の代わりにはなりません。このふたつのすばらしい本をぜひ読んでみてください。また、私たちのグループの問題意識を共有するために、コミュニティにはDDDの考え方が必要だということに賛同してくださるなら、周囲の人にこの本やEricの仕事について教えてあげてください。

Floyd Marinescu
Co-founder & Chief Editor of InfoQ.com

イントロダクション

ソフトウエアは複雑な現代社会を生きる私たちを助けるための装置です。ソフトウエアは何らかの目的のための手段であり、多くの場合、その目的はとても実践的で現実的なものです。例えば航空監視のためにソフトウエアを使います。この使い方は私たちを取り巻く世界と直接関係しています。どこかから別のどこかへ飛行機で移動したいとき、洗練された仕組みがその移動を実現してくれます。この場合のソフトウエアは常に空をとんでいる何千機もの飛行機の運行を調整するために作られているのです。

ソフトウエアは現実に即していて使いやすくなければなりません。さもなければソフトウエアの作成に多くの時間と資源を費やしはしないでしょう。現実に即していて使いやすいソフトウエアは、私たちの生活の特定の側面と強く結びつきます。便利なソフトウエアパッケージは、そのソフトウエアが適用される現実から切り離せません。そしてその現実とは、私たちが思い通りに扱おうとするドメインのことです。ソフトウエアはドメインを思い通りに扱うのを手助けしてくれるはずです。しかし、現実にはソフトウエアとドメインが複雑に絡み合ってしまっています。

ソフトウエアの設計は芸術です。したがって芸術と同様、厳密な科学のように定理や公式を利用して教わったり学んだりできません。私たちはソフトウエアの作成過程全体に適用できるような、原則や技法を発見することはできます。しかし、現実世界で生まれる要求から、その要求を満たすモジュールまでの道筋を、正確に描くことはできません。絵画や建築のように、ソフトウエアには設計者や開発者の個人的なタッチや、作成開始から徐々に成熟していく過程に貢献した人の直感や才能（またはその欠如）が含まれます。

ソフトウエアの設計にはいくつかの方法があります。ここ20年のあいだ、ソフトウエア産業は製品を作成するいくつかの方法を学び、それぞれに利用してきました。そうして出来上がったのは、長所や欠点をもつ製品です。この本の目的は、この20年間に現れて徐々に発展してきたある設計方法に光を当てることです。この方法はここ数年でさらに具体的になってきました。この方法をドメイン駆動設計といいます。Eric Evansはドメイン駆動設計についての知識を集めひとつの本にまとめ上げ、この分野に大きく貢献しました。この分野についてさらに詳しい解説が必要なら、彼が書いた本“Domain-Driven Design: Tackling Complexity in the Heart of Software”, published by Addison- Wesley, ISBN: 0-321-12521-5. を読むことをおすすめします。

また、下記のドメイン駆動設計ディスカッショングループでも多くの有用な洞察を学ぶことができます。

http://groups.yahoo.com/group/domaindrivendesign

この本は単なるイントロダクションにすぎません。ドメイン駆動設計を詳しく理解するためではなく、基礎を素早く学ぶための本です。この本を読んだ皆さんが、ドメイン駆動設計の原則やガイドラインを使ってソフトウエアの設計を、より優れたものしたいと考えてくれるようになることを望んでいます。

1

ドメイン駆動設計とは何か

ソフトウエアを開発する目的は、実世界の作業を自動化したり、ビジネス上の問題を解決することです。つまりソフトウエアは、業務の自動化や現実世界の問題など、具体的なドメインのためのソフトウエアなのです。ソフトウエアは特定のドメインから生まれ、そしてそのドメインと深く関わっている。まずは、このことを理解しなければなりません。

ソフトウエアを構成するのはコードです。したがって、コードをいじるのに多くの時間を費やしたくなるかもしれません。また、ソフトウエアはオブジェクトとメソッドの塊だ、と単純に考えたくもなるでしょう。

例えば、車の製造について考えてみましょう。自動化された製造過程に携わる労働者たちは、自動車の部品の専門家なのかもしれません。しかし、そうして働くうちは、彼らは自動車製造の全体の流れの一部分しか把握できません。労働者たちはまず、自動車を互いに組み合う様々な部品の集積とみなします。しかし実際は、自動車はそれ以上のものです。優れた自動車の製造はその理想像を出発点とし、仕様を注意深く記述することから始まります。そして、設計過程が続きます。非常に多くの設計をおこないます。何ヶ月も、ひょっとしたら何年も設計過程に費やされるかもしれません。完璧になるまで、理想像を反映したものになるまで試行錯誤が続けられるでしょう。設計過程は机上の作業だけにとどまりません。多くの場合、実際に自動車のモデルを開発し、特定の状況下で動作するかテストします。そのテスト結果をもとにして、設計に更なる変更が加えられます。そして最終的に生産ラインに送られ、各部品が作られ組み立てられていきます。

ソフトウエア開発もよく似ています。ただ座ってコーディングしているだけではだめです。そんなことは簡単なことですし、些細な案件ならそれで十分でしょう。しかし、複雑なソフトウエアは作れません。

優れたソフトウエアを作るためには、そのソフトウエアが一体何なのか知らなければなりません。銀行業務にとってもっとも大切なことを的確に理解していなければ、つまり銀行業務のドメインを理解していなければ、銀行業務システムを構築できません。

ドメインに対する十分な知識を持たずに、複雑な銀行業務システムを構築することは本当に不可能なのでしょうか。もちろん不可能です。絶対に。それでは、誰が銀行業務のドメインを知っているのでしょう。ソフトウエアアーキテクトでしょうか。いいえ。彼は自分のお金を安全に保管したり、必要なときに引き出したりするために銀行を利用しているだけです。ではソフトウエアアナリストでしょうか。違います。彼は必要な材料をすべて与えられたときに、与えられた案件について分析することができるだけです。それなら開発者は。だめです。では一体誰が。もちろん、銀行員です。銀行業務システムは銀行の中で働く人々や専門家がとてもよく理解しています。すべての細部、すべての落とし穴、すべての潜在的な問題点、すべてのルールを知っているのです。私たちは常にここから、すなわちドメインから出発しなければなりません。

ソフトウエア開発のプロジェクトを始めるとき、私たちはそのソフトウエアが適用されるドメインに注目すべきです。ソフトウエアを導入する真の目的は、そのドメインの業務をより効率的にするためです。そのためには、そのソフトウエアは改善対象となるドメインに円滑に適用できなければなりません。さもなければソフトウエアはドメインに余計なストレスを加え、機能不全と障害を引き起こし、あげくの果てには大混乱を生み出してしまうでしょう。

では、どうすればドメインに円滑に適合するソフトウエアを作成できるのでしょうか。もっともよい方法はドメインの反映としてのソフトウエアを作ることです。そのためには、ソフトウエアはドメインの核となる概念や要素を取り入れ、それらの関係を正確に再現する必要があります。ソフトウエアはドメインをモデル化しなければならないのです。

銀行業務の知識がなくても、ドメインのモデルのコードを読むだけで多くのことが理解できるでしょう。これはきわめて重要なことです。ドメインに深く根を下ろしていないソフトウエアは、時間が経ってドメインが変化しても対応できないでしょう。

では、私たちもドメインから始めましょう。でも何を。ドメインはこの世界についての何かです。 それは、ただ単にキーボードを使ってコンピュータに取り込んだり注ぎ込んだりすれば、コードになるというものではありません。ドメインの抽象を作る必要があります。ドメインの専門家との会話からドメインについて多くの知識を学べます。しかし、私たちの頭の中にドメインの抽象的な概念、つまりドメインの青写真がなければ、この生のままの知識をソフトウエアの構造に変換するのは難しいです。始めの青写真は不完全です。しかし、作業を続けるうちに、よりよいものになり、そしてどんどん明瞭になっていきます。では「抽象」とは何でしょう。それはモデルです。ドメインのモデルのことです。Eric Evansによれば、ドメインモデルは特定の図で表されるものではなく、そのような図が伝えようとする概念です。そしてドメインモデルはドメインの専門家の頭の中にある単なる知識ではありません。ドメインモデルは

そういった知識を厳密に取捨選択しながら作成する抽象です。図はモデルを表現しモデルの意味を伝えることができます。しかし、それは注意深く書かれたコードでも、また文章でもできます。

ドメインモデルは、対象のドメインをプロジェクトの関係者に向けて表現したもので、設計と開発過程でとても重要になります。設計過程では、ドメインモデルを常に意識し、何度も参照します。私たちを取り巻く世界をそのまま扱うと、頭に大変な負担がかかってしまいます。特定のドメインですら、ひとりの人間の頭脳では一度に扱えません。したがって、情報を整理し、システム化し、より小さな部分に分割し、論理的な単位にまとめて、一度に扱えるようにする必要があります。ドメインのある部分は捨てなければならないでしょう。ドメインはとても多くの情報を含んでいるので、すべてをモデルに取り込めません。また、ドメインの大部分は考慮する必要さえないでしょう。こういった取捨選択自体が大きな課題となります。何をモデルに取り込み、何を捨てるのか。これが設計作業であり、ソフトウエアの作成です。銀行業務システムはきっと顧客の住所録を保持するでしょうが、顧客の目の色は無視します。もっともこれはわかりやすい例です。他の場合はこれほどわかりやすくないでしょう。

モデルはソフトウエア設計において本質的なものです。複雑さに対処するためにはモデルが必要です。ドメインについて考えたことは、すべてモデルに取り込まれます。これはすばらしいことですが、モデルは私たちの頭の中にだけに現れます。モデルが頭の中にあるままならあまり扱いやすくない、でしょう。私たちは、ドメインの専門家や一緒に作業する設計者や開発者に、モデルを伝えなければなりません。

モデルはソフトウエアの核ですが、これを表現し他人に伝える方法が必要です。私たちは一人で作業するわけではありません。上手に、正確に、完璧に、曖昧さを排除して、知識と情報を共有しなければなりません。これにはいくつかの方法があります。ひとつは図による表現、すなわちダイヤグラム、ユースケース、スケッチ、絵などです。もうひとつは記述による表現、つまりドメインについての見通しを文章で記述する方法です。そして独自言語で表現する方法もあります。ドメインについての特定の問題を伝達するために私たちは独自の言語を作成することができますし、そうするべきなのです。これらの詳細は後述しますが、もっとも大切なのは「モデルを他人に伝えなければならない」ということです。

ドメインモデルを表現したものがあれば、コードの設計を始められます。この作業はソフトウエアの設計とは異なります。ソフトウエアの設計は家を構成するのに似ています。いわば全体的な展望を作る作業です。一方、コードの設計は詳細部についての作業であり、どの壁にどの絵画を掛けるのかを決めるような作業です。もちろんコードの設計はとても重要な作業ですが、ソフトウエアの設計のような基礎となる作業ではありません。ほとんどの場合、コードの設

計上のミスは簡単に修正できますが、ソフトウエアの設計上の失敗を修正するには、多くのコストがかかります。絵画をもっと左側に掛けるのと、家の側面を取り壊して今までとは違うように改築するのはまったく違う作業です。しかし、優れたコード設計なしによい製品は完成しません。だからこそコード設計のパターン集を手元に置き、必要になったら適用します。優れたコーディング技術は、きれいで保守しやすいコードを記述する手助けをしてくれます。

ソフトウエア設計にはいくつかの手法があります。ひとつはウォーターフォール設計と呼ばれる手法です。この手法はいくつかの段階を含んでいます。まずは、業務の専門家が要求を書き起こし、それをアナリストに渡します。アナリストはその要求をもとにモデルを作成して開発者に渡し、開発者はそのモデルをもとにコーディングを始めます。この手法はソフトウエア設計の伝統的な手法であり、一定の成果を上げ、長く使われてきました。しかし、この手法には欠陥と限界があります。最大の問題はアナリストから業務の専門家へ、または開発者からアナリストへのフィードバックがないことです。

もうひとつの手法はエクストリームプログラム（XP）に代表されるようなアジャイルと呼ばれるいくつかの方法論です。これらの方法論はウォーターフォール設計に対抗してうまれたものです。すべての要求を事前に明らかにして、さらに要求の変更も考慮するのは難しいです。この難しさに対処するためにうまれたのがアジャイルの方法論です。事前にドメインのすべての側面を含む、完璧なモデルを作成することは、本当に大変なことです。ドメインはたくさんの事柄を抱えています。ほとんどの場合、最初にすべての問題を見切るのは無理ですし、設計の副作用やミスを予見するのは不可能です。アジャイルが解決しようとするもうひとつの問題は、”分析麻痺”とでも呼ぶべき、チームのメンバが設計上の決定を過度に恐れるあまり、全く作業が進まなくなってしまう状態です。もちろんアジャイルの支持者は設計上の決定の重要さを認識しています。しかし、事前に完璧な設計をしようとはしません。そのかわり、彼らは実装の柔軟性を最大限に利用し、反復的に開発しながら、関係者と継続的に通じ合い、多くのリファクタリングを実行します。このような過程を通じて、開発チームは顧客のドメインについて多くを学び、そして顧客のニーズに合致したよりよいソフトウエアを作成することができるのです。

アジャイルの手法には特有の問題点と限界があります。誰もが単純にアジャイルを支持しますが、アジャイルが何を意味するかについてそれぞれ独自の見解を持っています。また、設計について確固とした基準を持っていない開発者が継続的にリファクタリングをしても、理解しづらく変更しにくいコードが生み出されるだけでしょう。そしてウォーターフォール設計は実装過剰を生み出しましたが、実装過剰をおそれるアジャイルの手法は別の恐ろしい事態を生み出します。それは、深く完璧に考え抜いて設計を行うこと、いわば設計過剰ともいうべきものです。

この本はドメイン駆動設計の原則を説明します。この設計原則を適用すれば、どのような開発過程であれ、その能力を十分に発揮してドメインの複雑な問題に対して継続的にモデリングし、実装できるようになります。ドメイン駆動設計は設計と開発を兼ね備え、どのように設計と開発が協調すればよりよいソフトウエアが出来上がるのかを示しています。優れた設計は開発を加速させ、開発からのフィードバックは設計の精度を高めるでしょう。

ドメインの知識を構築する

飛行機の飛行制御システムを例に、どのようにドメインの知識を構築できるのか考えてみましょう。

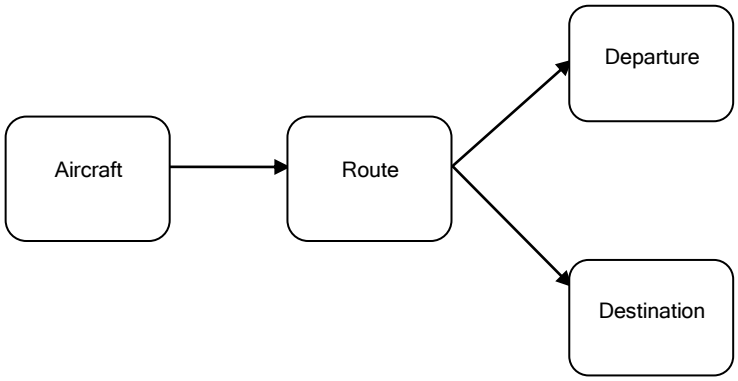
ある瞬間に地球上の様々な場所で何千もの飛行機が飛んでいます。それぞれが目的地にむかって独自の航路を飛行しています。このとき重要なのは、飛行機同士が衝突しないようにすることです。ここでは飛行制御システム全体について詳述せずに、その一部である航空監視システムについて考えてみます。提案された企画は、特定の領域で飛行しているすべての飛行機を追跡して、想定通りのルートを飛行しているかどうか、衝突する可能性はあるかどうかを判断する監視システムです。

さて、ソフトウエア開発の観点から考えた場合、どこから作業を始めたらいいのでしょう。前節ではドメインを理解することから始めるべきだと説きました。このケースでは、ドメインは航空監視業務になります。航空管制官がこのドメインの専門家です。しかし、管制官はシステム設計者でもソフトウエアの専門家でもありません。問題のドメインを完璧に説明してくれるとは思えません。

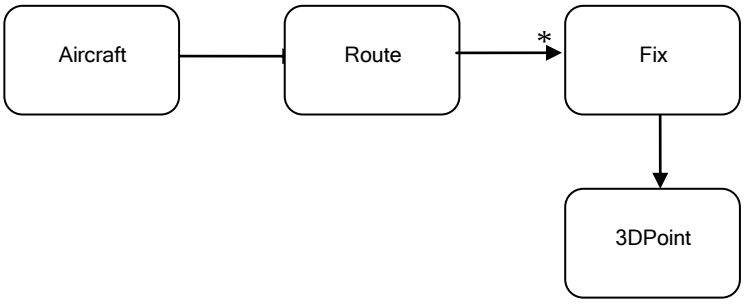
航空管制官はドメインの知識をたくさん持っています。しかしモデルを構築するためには、その知識から重要な部分を抜き出して一般化する必要があります。管制官と話し合いを始めると、飛行機の離発着、飛行中、衝突の危険性、着陸許可が出るまで空中で待機する飛行機などについて多くのことを聞けるでしょう。このような混沌としているようにみえる情報の集まりから秩序を見いだすため、何かの作業を始めなければなりません。

まず、管制官もあなたも同意するのは、各飛行機は出発する空港と到着する空港を持っていることです。したがって下図のように、飛行機と出発地と目的地がある、となるでしょう。

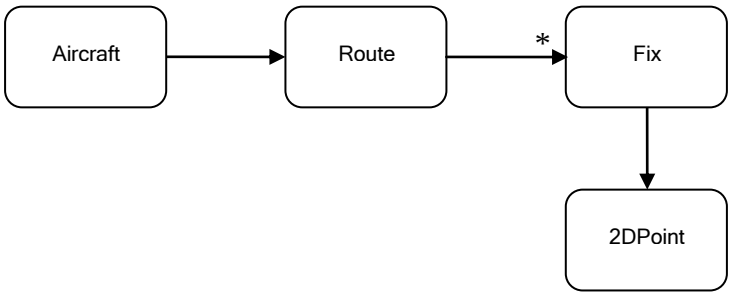
これで、飛行機はどこかから出発し、そしてどこかに到着するのがわかりました。では、飛行中は何が起こっているのでしょう。どのような経路を飛行しているのでしょうか。実のところ、離発着よりも、飛行中に起こっていることのほうが私たちの興味をひきます。管制官によると、どの飛行機も飛行行程のすべてを記述した運行計画が割り当てられている、とのことです。運行計画について聞きながら、あなたは内心、これは飛行機の経路についての話だと考えるかもしれません。さらに議論すると、あなたは興味深い単語を聞きます。それは「航路」です。あなたがこの単語にすぐに着目したのはそれなりの理由があります。航路は飛行行程の重要な概念を含んでいるからです。航路にしたがう。これが飛行機が飛行中におこなうことです。飛行機の出発地と目的地は航路の始点と終点でもあることが明らかになります。したがって、飛行機を出発地と目的地に関連させるよりも、航路と関連させた方が自然な気がします。この場合、航路は出発地と発着地に関連します。



管制官と航路について話し合いをしていると、実際の航路は小さな区分で形成されているのを発見します。この小さな区分を集めると出発地から目的地を結ぶ一本の折れ線ができます。この線は事前に決められた定点を通っているはずです。したがって、航路は一定の順序で並んだ定点の連なりだと考えられます。こうなると、もはやあなたは出発地と目的地が航路の端点であると考えずに、単なる２つの定点だと考えます。ひょっとしたら管制官の見方とは大きく異なっているのかもしれませんが、こういう抽象的な見方は不可欠であり、後になって役に立ちます。これらの発見をモデルに反映すると下図のようになります。



図には新しい要素が現れています。この図はそれぞれの定点が空間上にあり、そこを航路が通っていることを表します。定点は３次元の点です。しかし、管制官と話していると、管制官はこのような見方をしていないことがわかるでしょう。実際、彼は航路を文字通り飛行機が運航する路だと考えています。定点は単なる地球上の点で、経度と緯度で一意に決まります。したがって、正しい図は、



さて、これまでのところ何が起こったのでしょうか。あなたとドメインの専門家が議論をして、知識を交換します。まずはあなたが質問をして専門家が答えます。そうしている間に、専門家は航空運行のドメインからもっとも重要な概念を掘り起こします。見つかったときには、このような概念は未熟で混乱したままかもしれません。しかし、これらはドメインを理解するために必要です。したがって、専門家からドメインについてできるだけ多くのことを学ばなければなりません。そして簡単な質問を投げかけたり、情報を軽く整理してみたりして、あなたと専門家はドメインモデルをスケッチし始めるでしょう。完璧でも正確でもないスケッチですが、なくてはならないはじめの一歩です。まずは、ドメインの中の最も重要な概念を明らかにするように努力しましょう。
これは設計の中でも重要な作業です。多くの場合、ソフトウエアアーキテクトや開発者とドメインの専門家は長時間にわたり議論します。ソフトウエアの専門家はドメインの専門家から知識を引き出したいと思っていますし、その知識を扱いやすい形式に変換しなければなりません。また、早めにプロトタイプを

作って、どのように動作するかを確かめたくなるでしょう。動作を確かめれば、モデルや自分たちの手法に問題点をみつけるかもしれませんし、モデルを変更したくなるかもしれません。もちろん、ドメインの専門家からソフトウエアアーキテクトや開発者へと一方的に知識が伝わるわけではありません。フィードバックが発生します。これはよりよいモデルを作るのに役立ちますし、より明瞭に正確にモデルを理解するのを助けてくれます。ドメインの専門家は高度な専門知識を持っていますが、その知識を独特の方法で組み立てて使います。そして通常、この組み立て方はソフトウエアシステムとして実装するのに最適ではありません。ソフトウエアの設計者は分析的に考えることで、専門家との議論の中から、いくつかの鍵となる概念を掘り出します。そして、次の章で説明するような議論の枠組みを組み立てやすくします。私たちソフトウエアの専門家（アーキテクトと開発者）とドメインの専門家はドメインモデルを一緒に作成します。ドメインモデルはこの２つの専門領域が出会う場所でもあるあります。これは多くの時間を費やさなければならない作業に思えるでしょう。そして、実際その通りです。そうするべきです。なぜなら、ソフトウエアの最終的な目的は実際の生きたドメインでの業務上の問題を解決することであり、そのためにはソフトウエアはドメインと完璧に一体になる必要があるのですから。

2

ユビキタス言語

共通言語の必要性

前章で説明したのは、ソフトウエアの専門家とドメインの専門家が一緒に働いてドメインモデルを作成していくことが不可欠となる事例でした。しかし、ソフトウエアの専門家とドメインの専門家の間にはコミュンケーションを阻む大きな壁があります。したがって、この手法は始めから困難を抱え込みます。ソフトウエアの開発者はクラスやメソッド、アルゴリズム、パターンなどについて頭を悩ませ、常に実世界の概念とプログラムを結びつけようとします。どんなクラスを作るべきか、どんな関係をモデリングすべきかを見極めたいのです。彼らは継承やポリモリフィズム、OOPなどのプログラミングの言葉で考えます。話をするときも常にプログラミングの言葉を使います。そうするのが普通なのです。開発者は開発者のままなのですから。しかし、ドメインの専門家はこんなことは何も知りません。ソフトウエアライブラリ、フレームワーク、永続化はもちろん、データベースさえ知らないことが多いでしょう。彼らは専門とする特定の領域しか知らないのです。

航空監視システムの例では、ドメインの専門家は飛行機について、航路について、高度、経度、緯度について知っています。通常の航路からの逸脱や飛行機の軌道についても知っています。そして、これらのことについて独自の専門用語で話し合います。部外者が理解できない専門用語を使うこともあるでしょう。

ドメインモデルを作成するときは、このようなコミュニケーションのやり方の違いを克服するため、互いの考えを交換しなければなりません。モデルについて、モデルに含まれる要素について、それらの要素を結びつける方法について、なにが適切でなにがそうでないかを話し合う必要があります。この段階のコミュニケーションはプロジェクトが成功するかどうかを決めるとても重要なものです。誰かの発言を、他の誰かが理解できなかったり、さらに悪いことに誤解したりすれば、プロジェクトが成功する可能性はどのくらいあるでしょう。

プロジェクトが深刻な問題に直面するのは、チームのメンバがドメインについて議論するための共通の言語を共有していないときです。ドメインの専門家は独自の専門用語を使います。一方で技術チームのメンバが使うのも独自の専門

用語です。その用語は設計の観点からドメインについて議論するために調節されています。しかし、両者の共通の言語にはなりません。

これでは、日々の議論で使う用語はコード（究極的にはこれがソフトウエア開発プロジェクトでもっとも重要な成果物になります）で使われる用語と結びつきません。同じ人物が、話すときと書くときで違う用語を使うことさえあります。これではドメインについてのもっとも的確な表現が一瞬だけ現れて、コードにも文書にさえも反映されないまま消えてしまいます。

このような会議で考えを交換するときには、他の人が理解できるように用語を言い換えるのが一般的です。開発者は門外漢でもわかる用語でデザインパターンを説明しようとするかもしれませんが、うまくいかないこともあります。ドメインの専門家はなんとかして開発者の用語を理解するために、新しい専門用語を作ってしまうかもしれません。こんなやりとりを繰り返しても苦しいだけです。それにこの種の言い換えは、ドメインに関する知識を確立する作業には役に立ちません。

私たちは設計会議で専門用語を使いたがります。しかし、このような専門用語は共通言語になりません。誰の要求も満たさないからです。

したがって、ドメインモデルについて議論し、ドメインモデルを定義する会議では、全員が同じ言語を使わなければなりません。ではどのような言語が共通のものになり得るでしょうか。開発者の言語でしょうか。またはドメインの専門家の言語でしょうか。それともその中間の何かでしょうか。

ドメイン駆動設計の核となる原則は、ドメインモデルに基づく言語を使うことです。ドメインモデルはソフトウエアとドメインが出会う場所にあるのですから、共通言語の基盤として適切です。

共通言語の拠り所としてドメインモデルを使います。そしてこの言語をコミュニケーションに、さらにはコードにも使うようにチームのメンバに要求しましょう。知識を共有しドメインモデルを構築しているあいだにも、チーム内ではレビューをしたり文書や図を作ったりします。どんな形態のコミュニケーションであれ、常にこの言語で表現しましょう。このような性質からこの言語は「ユビキタス言語」と呼ばれます。

ユビキタス言語を使うことで設計のすべての部分が結びつき、設計チームが効率よく作業するための前提が生まれます。大規模なプロジェクトでは、設計が形になるまでに数週間、または数ヶ月かかることもあります。チームのメンバはプロジェクト初期に定義された概念が間違っていたり、不適切に使われていることに気づきます。あるいは設計上の新しい要素を発見して、その要素がド

メイン全体と適合するように注意しなければならない場合もあります。このような作業では、共通言語が不可欠です。

共通言語として使える用語はいつでも現れるわけではありません。主要な用語を確実に見つけるのはとても難しく、集中力が必要です。ドメインの定義と設計の方向を決めるような重要な要素を見つけ、これらの要素に適切な名前をつけて、その名前を使っていかなければなりません。簡単に見つかる名前もあり、見つけるのが難しい名前もあります。

設計上の困難を取り除くには、試しに代わりとなる表現をつかってみるのもいいでしょう。この代わりとなるの表現は、代替となる他のモデルを反映します。その表現を使って、コードをリファクタリングし、クラス名、メソッド名、モジュール名を変更して新しいモデルに適合させていきます。会話をしているときに、すでに共有している日常的な言葉と用語の間で混乱が起きないようにしましょう。

このような共通言語を構築すると、その結果ははっきりと現われます。つまり、ドメインモデルと言語が強く結びつくようになります。言語を変更したらモデルも変更しなければならないようになります。

ドメインの専門家は、ドメインを表現するのには不十分で、不適切な用語やモデルの構造には反対しなければなりません。ドメインモデルや共通言語の中で、ドメインの専門家が理解できないものがあったときは、きっと何か間違いがあります。一方、開発者は設計に紛れ込みやすい曖昧さや矛盾が現われていないか注視しなければなりません。

ユビキタス言語を創造する

では、どのようにして言語の構築を始めればよいでしょう。次の対話は、航空監視プロジェクトでのソフトウエア開発者とドメインの専門家との架空の対話です。太字で書かれている単語に注目してください。

開発者：飛行機の運行を監視したいのですが、何から始めればいいのでしょうか。

専門家：基本的なことから始めましょう。どのような飛行機を監視するにせよ監視対象の**飛行機**は必ず存在します。それぞれの飛行機は**出発地**から飛び立ち、**目的地**に到着します。

開発者：なるほど、簡単ですね。飛行しているときに、パイロットは好きな進路を選ぶことができるのですか。目的地に着くのであれば、どの方向へ進むのかは自分たちの責任で決めてしまっていいのですか。

専門家：いいえ、それは違います。パイロットには飛ばなければならない**航路**が指示されます。パイロットは可能な限り、航路に沿って飛行しなければなりません。

開発者：私は**航路**を空中に浮かぶ３次元の路として考えています。しかし、３次元のデカルト座標系を利用すれば、**航路**は３次元上の点の連なりにすぎないのですね。

専門家：私はそうは思いません。航路をそのように考えていません。実際は、航路は飛行すべき道筋を地球の表面に投影したものです。なので航路は地表上の点の連なりです。その点は**経度**と**緯度**で決まります。

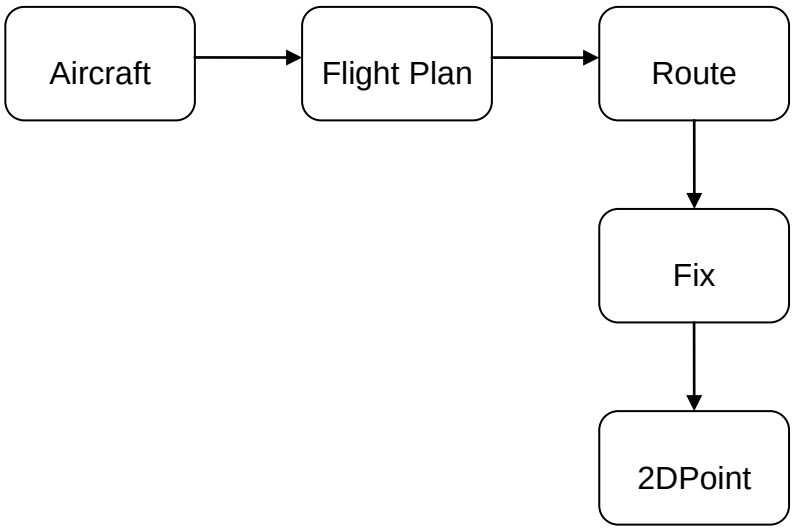
開発者：わかりました。ではそのような経度と緯度で決まる点を**運行定点**と呼びましょう。地球上の固定された点ですからね。それから、航路は２次元の点の連なりとして考えましょう。とすると、**出発地**も**目的地**も**運行定点**ですから、特別な概念として考えなくてもよさそうです。**航路**は他の**運行定点**に結びつけられるのと同じように目的地に結びつけられるというわけです。ところで、飛行機が航路に沿って飛ばなければならないのはわかりましたが、高いところを飛んだり低いところを飛んだりするのは自由にできるのですか。

専門家：いいえ。飛行機が飛んでいる間に維持すべき**高度**も**運行計画**によって事前に決められています。

開発者：**運行計画**？なんですか、それは？

専門家：空港を離陸する前に、パイロットは詳細な**運行計画**を受け取ります。**運行計画**は運行に関するあらゆる情報を含んでいます。**航路**、維持すべき**高度**、**飛行速度**、飛行機の型式、さらには乗務員の情報も**運行計画**に含まれています。

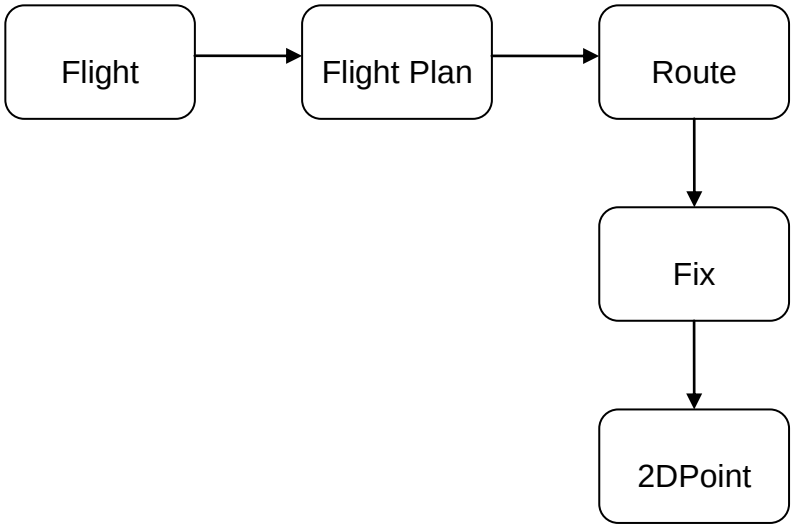
開発者：ふむ、**運行計画**はかなり重要そうですね。モデルに取り込みましょう。



開発者：これでもっとよくなりましたね。この図のおかげで、気づいたことがあります。運行を監視しているときは、じつは飛行機そのものに関心があるのではない、機体の色が白だろうと青だろうと、機種がボーイングだろうとエアバスだろうと関係ないということです。関心があるのはその飛行機の**フライト**なのです。実際に追跡し、計測するのは**フライト**なのです。モデルをもっと正確にするために少し変更する必要がありますね。

このチームが、航空監視業務のドメインとその初期のモデルについて議論しながら、太字で表記した単語を使ってゆっくりと言語を作り上げていく方法に注目してください。そして、その言語がいかにしてモデルを変更するかについても注意しましょう。

けれども、実生活ではこのような対話は冗長すぎます。それに人が婉曲的に話したり、話題の詳細に立ち入りすぎたり、概念を取り違えたりするのはよくあることです。しかし、そういう会話では共通言語を発見するのは難しいです。この困難な作業に取り組み始めるには、チームのすべてのメンバが共通言語を作らなければならないと自覚し、重要な点に常に注目するように気をつけ、必要なときはいつでも作成した共通言語を使うようにします。このような作業のときには、独自の専門用語を可能な限り使わないようにします。そしてユビキタス言語を使いましょう。ユビキタス言語は明確に、そして正確に意思伝達をする手助けをしてくれるのですから。



開発者はドメインモデルの主要な概念をコードへ実装するべきです。航路のためのクラスや運行定点のためのクラスを記述できるでしょう。運行定点クラスは２次元座標クラスから派生してもよいですし、２次元座標を主な属性とするクラスでもいいでしょう。実際にどのようにするかは、この後の章で説明するような他の要因に依存します。ドメインモデルの概念をクラスへと実装することで、ドメインモデルとコードを対応づけることができます。そして、それは共通言語とコードを対応づけることでもあります。しっかり対応づけられていると、コードが読みやすくなり、コードからドメインモデルを再作成できるようになるのでとても役に立ちます。また、プロジェクトの後半になると、コードがドメインモデルを表現している利点があらわれます。例えば、ドメインモデルが大きくなったときや、設計が間違っていたので、コードを変更すると望ましくない結果になりそうなときに役に立ちます。

ここまで、どのようにしてユビキタス言語がチーム全体で共有されるのか、さらにユビキタス言語がどのようにして知識の確立とドメインモデルの作成に役立つのかを説明しました。では、ユビキタス言語はどのように表現するべきでしょうか。会話だけで十分なのでしょうか。上に見てきた例では図でユビキタス言語を表現しました。ほかにはどうでしょうか。文書として記述するのはどうでしょう。

モデルの作成にはUMLを使えば十分だという人もいるかもしれません。確かに主要な概念をクラスとして記述し、クラス間の関係をあらわすのにはUMLはとても便利なツールです。UMLを使えば、あなたはスケッチブックに４つ、５つのクラスを描いて、それらに名前をつけて、さらにクラス同士の関係をあらわすことができます。この方法なら、他の人はあなたの考えていることをとても簡単に追いかけられます。また、考えていることが図で表現されていると理解

しやすいです。特定の話題についてすぐに認識があうので、簡潔にコミュニケーションできます。新しい考えが浮かんだときは、図を修正して概念上の変更を反映するようします。

図に含まれる要素の数が少なければUMLはとても便利です。しかし、夏の心地いい雨を浴びたキノコのようにUML図が巨大化するかもしれません。例えば、ミシシッピ川と同じくらい長くて大きい紙を埋め尽くす何百ものクラスを扱うとしたらどうしますか。このようなUML図はソフトウエアの専門家ですら読むのは難しいでしょう。ドメインの専門家については言わずもがな、です。彼らは巨大化したUML図を理解しないでしょう。しかし、中規模のプロジェクトでさえUML図は巨大化します。

もちろんUMLは、クラスやその属性、クラス間の関係を表現するのが得意です。しかし、クラスの振る舞いや制約を表現するのは簡単ではありません。そのため、UMLでは図のなかにメモを入れて文章で補足します。したがって、UMLではモデルについての２つの重要な側面、つまりモデルがあらわす概念の意味とオブジェクトがするべきことを伝えられません。でもそれでいいのです。コミュニケーションのための別の道具を導入すればいいのですから。

文書を使ってもいいでしょう。モデルについて認識をあわせるためのおすすめの方法は、モデルの部分をあらわす小さな図を作成することです。これらの図は、いくつかのクラスを含み、それらの関係を表し、概念全体の一部分を適切に含んでいます。さらに、図に文章を付加します。文章を使えば、図が説明できない振る舞いや制約を説明できるでしょう。図も文章も、それぞれにドメインの重要な側面を説明しようとします。それぞれがドメインのある部分に光を当てます。

これらの文書は手書きでもかまいません。文書が伝えるのは一時的な感触だからです。この感触は遠からず変更されます。ドメインモデルは、作成を始めたときから安定した状態になるまで何度も変更されるからです。

ドメインモデル全体をあらわす大きな図を描くのは魅力的かもしれません。しかしほとんどの場合、そのような図を作成するのは不可能です。そのような図を作成できたとしても雑然としすぎているので、小さな図の集まりの方がよりよく理解できるでしょう。

また、長い文書には気をつけてください。というのは、書くのに多くの時間が費やされるので、書き終わる前に古くなってしまうかもしれないからです。文書はドメインモデルと同期しなければなりません。古い文書は、間違った言語で書かれていて、ドメインモデルを反映していないので役に立ちません。古いドキュメントはなるべく使わないようにしてください。

もちろん、コードを使ってコミュニケーションをすることも可能です。この方法はXPプログラミングのコミュニティで広く支持されています。丁寧に書かれたコードはコミュニケーションにとても適しています。しかし、例えばコードを読むことで、メソッドが表現する振る舞いが理解できても、そのメソッド名がその振る舞いと同じくらい明瞭に理解できるでしょうか。テストのためのアサーションは、アサーション自体の内容を十分に伝えてくれますが、変数名やコードの構造全体については何か伝えてくれるでしょうか。すべての挙動を一目瞭然に教えてくれるでしょうか。コードは正しく振る舞いますが、必ずしも正しい表現をするわけではありません。コードを使ってモデルを表現するのはとても難しいことです。

設計では他にもコミュニケーションの方法があります。この本の目的はその方法をすべて紹介することではありません。しかし、次の点は明らかです。すなわち、ソフトウエアアーキテクト、開発者、ドメインの専門家を含む設計チームに必要なのは、作業を統一し、モデルの作成とコーディングを助ける共通言語だということです。

3

モデル駆動設計

前章では、業務ドメインを中心に据えたソフトウエア開発手法の重要さを強調しました。ドメインに深く根ざしたモデルをつくることがとても重要であり、そのモデルは、ドメインの中心にある概念を正確に反映するべきだと説明しました。「ユビキタス言語」はモデリングの作業全体を通して、ソフトウエア開発者とドメインの専門家とのコミュニケーションをとても楽にします。また、モデルに取り込むべきドメインの中心概念の発見にも役立ちます。モデリングの目的は、よいモデルを作成することです。そして次は、モデルをコードとして実装していく作業です。ソフトウエア開発においては、この作業もモデリングと同様に重要です。すばらしいドメインモデルを作成しても、コードの中へ正確に移植できなければ、品質の悪いソフトウエアになってしまいます。

システムアナリストと業務ドメインの専門家が一緒に作業して、ドメインの基本的な要素を見つけ出し、関連を結び、適切なモデルを作成し、そのモデルはドメインを正確に捉えているとします。そして、このモデルはソフトウエア開発者へ渡されます。開発者はモデルを検討して、モデルの中の概念や関連にはコードで正確に表現できないことがあるのに気づくかもしれません。そのため、開発者はモデルから着想を得て、独自の設計をします。その設計は部分的にモデルからアイディアを借用していますが、開発者独自の要素も加わっています。そして開発が進むと、コードにより多くのクラスが追加され、元のモデルと最終的な実装はさらに乖離します。これが良くない結果をもたらすかどうかはわかりません。優秀な開発者なら正しく動作するソフトウエアを作りあげるかもしれません。しかし、それは性能テストに耐えられるでしょうか。拡張性はあるでしょうか。保守性はどうでしょう。

どのようなドメインでも様々なモデルで表せます。そしてどのようなモデルでも様々な方法でコードに落とし込めます。どんな問題にもひとつ以上の解決方法があります。ではどの方法を選べばいいのでしょうか。よく分析されている正確なモデルが、必ずしもコードでそのまま表現できるモデルであるとはかぎりません。それどころか、ソフトウエア設計の原則を無視した実装になることもあるでしょう。原則を無視するのは勧められません。重要なのは簡単に、そして正確にコードに落とし込めるモデルを選ぶことです。ではここで基本的な質問です。私たちはどのような手法でモデルからコードへ変換するのでしょうか。

推奨できる設計手法のひとつに分析モデルと呼ばれるものがあります。この手法はコード設計の手法とは別のもので、コード設計者とは別の人が使うのが一般的です。分析モデルは業務ドメインを分析した結果であり、ソフトウエアとして実装することを考慮していません。このようなモデルを使うのはドメインを理解するためです。このモデルを使えば、ある程度ドメインの知識が確立でき、よく分析された正確なモデルが出来上がるでしょう。この段階ではソフトウエアは考慮されません。混乱の原因になると思われるからです。そして、このモデルが設計を担当する開発者へ渡されます。しかし、このモデルは設計の原則を念頭において作られていないため、開発者の作業の基礎にならないかもしれません。開発者はモデルを変更したり、別のモデルを作成しなければならないでしょう。こうなると、元のモデルとコードは無関係になります。結局、分析モデルはコーディングが始まったあとすぐに捨てられてしまいます。

この方法の主な問題点は、アナリストがモデルの欠点や複雑な点をすべて予見できないことです。アナリストはモデルの一部の構成要素は詳しく分析しても、残りは十分に分析していないかもしれません。このため、とても重要な細部が設計や実装の段階になって初めて見つかってしまいます。仮にドメインを正確にあらわしたモデルを使ってみれば、オブジェクトの永続化に深刻な問題があったり、性能が許容できないほど悪いことがわかるでしょう。

また、開発者は独自に決断をせざるを得ないことがあるでしょう。モデルを作成した時には考慮していなかった問題を解決するために、設計を変更することもあります。モデルから抜け落ちてしまっている部分を設計するのですから、さらにモデルとの関連性が希薄になります。

設計や開発のチームとは無関係にアナリストが働いていれば、最終的には分析モデルが出来上がるでしょう。しかし、モデルを設計者に渡せば、アナリストが持っているドメインやドメインモデルに関する知識はいくらか失われてしまいます。図や文章でモデルが表現されているでしょうが、それだけでは設計者はモデルの全体像やオブジェクト同士の関連や振る舞いを把握できないからです。モデルの細部は図では表現できないこともあります。文章でも完全に表現できないかもしれません。開発者はこのような部分をはっきりとさせるのに苦労するでしょう。モデルが意図している振る舞いについて勝手な想定をするかもしれません。こうなると間違った実装してしまう可能性があります。その結果、不適切な機能を持つプログラムになってしまいます。

アナリストたちは自分たちだけが参加する会議で、ドメインについて様々な議論をして多くの知識を共有します。そして、ドメインのすべての情報が圧縮されているようなモデルを作成し、開発者は渡された文書からそのすべてを学ばなければなりません。開発者がコードの設計をする前に、アナリストの会議に参加して、ドメインとそのモデルについての見通しを明確で完璧なものにできたら、もっと生産的でしょう。

つまり、より良い方法はドメインモデリングと設計を密接にすることです。モデルを作成するときは、ソフトウエアとその設計を考慮するべきです。また開発者もモデリングに参加すべきです。モデルに基づいた設計作業を後戻りすることなく進めるには、ソフトウエアを正確に表現するモデルを選択することが重要です。コードが基礎になるモデルとしっかり結びついていれば、コードの意味が明確になり、モデルもより適切になるでしょう。

開発者を巻き込むことでフィードバックが得られます。開発者からのフィードバックは、確実にモデルをソフトウエアとして実装するために役立ちます。モデリングに間違いがある場合も、フィードバックがあれば、早い段階で発見して簡単に修正できます。

コードを書く人はモデルをよく知り、モデルが完全であることに責任を感じるべきです。そして、コードの変更はモデルの変更を伴うことを自覚しなければなりません。そうでないと元のモデルと無関係になるまで、コードとリファクタリングしてしまうでしょう。また、実装に関心を示さないアナリストは、開発中に初めて見つかった実装上の制限に興味を示さないでしょう。その結果、モデルは実践に適したものではなくなります。

どんな技術者であれ、ドメインモデルの作成に関係するのであれば、いくらか時間を割いてコードを触ってみなければなりません。たとえそのプロジェクトを主導する役割を担っていても技術者であればそうすべきです。コードの変更に責任がある者はだれでも、コードを通してモデルを表現することを学ばなければなりません。すべての開発者はドメインモデルについての議論に参加し、ドメインの専門家と意見交換をする必要があります。プロジェクトのメンバは様々な方法でドメインモデルの作成に関わりますが、彼らは実際にコードを触る技術者とユビキタス言語を使って活発に意見を交換しなければなりません。

もし、設計の中心となる部分がドメインモデルと対応していないのなら、そのドメインモデルの価値は低いです。そのモデルに基づいて作成するソフトウエアの品質も疑わしいでしょう。また、ドメインモデルと設計の対応関係が複雑だと理解しにくく、実際に設計を変更すると対応関係は失われてしまいます。分析と設計の間に致命的な乖離が生まれるのは、作業を通じて各自が得た知見をチームの他のメンバに伝えていないからです。

ドメインモデルを正しく設計に反映させるためには、ソフトウエアシステムを部分ごとに設計します。そうすれば、ドメインモデルと設計の対応関係も明確になります。また、ドメインモデルを見直して修正を加えることで、より自然にソフトウエアを実装できるようにします。ドメインの詳細な内容をモデルに表現しようとする場合も同様です。設計との対応関係が明確なモデルであること。より自然にソフトウエアを実装できるモデルであること。この二つを満た

すひとつのモデルが必要です。加えて、ユビキタス言語が十分に使われていなければなりません。

ドメインモデルから設計に使われる用語を抜き出しましょう。また、モデルのどの要素にどの責務を割り当てるのかも、大まかに考えておきます。コードはドメインモデルを表現するので、コードの変更はモデルの変更と同じです。そして、その変更の影響はプロジェクトのその他の作業にも、相応に波及していかなければなりません。

実装とモデルを強く結びつけるためには、オブジェクト指向プログラミングのようなモデリングのパラダイムに基づいているプログラミング言語やソフトウエア開発ツールが必要です。

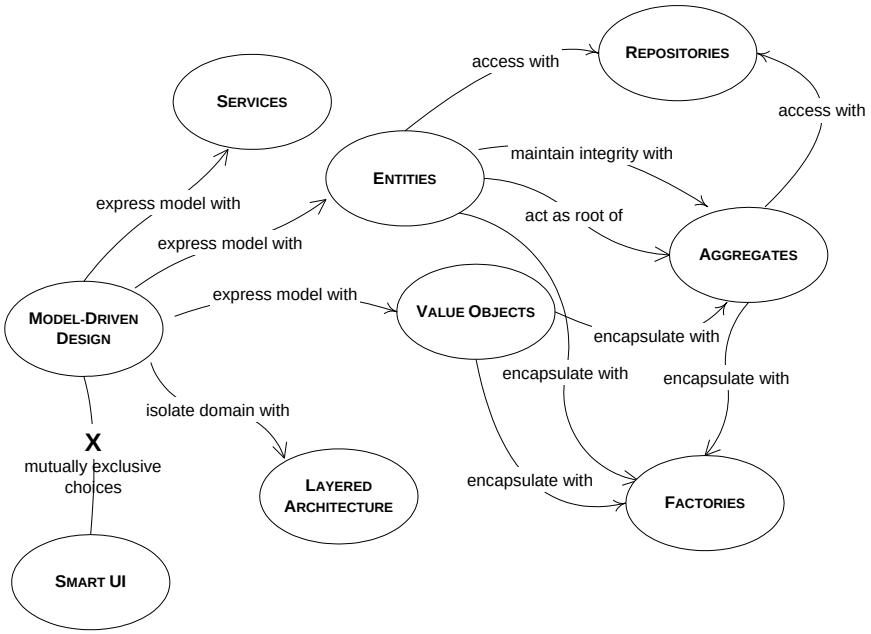
オブジェクト指向プログラミングは、モデルをソフトウエアとして実装するのに適しています。モデリングもOOPも同じパラダイムに基づいているからです。オブジェクト指向プログラミングはオブジェクトのクラスやクラス同士の関連、オブジェクトのインスタンス、そしてインスタンス間のメッセージングを提供します。そのため、OOPに基づく言語はモデルの要素とその関係を、直接プログラムの要素と対応づけることができます。

手続き型の言語はモデル駆動設計を十分にサポートしません。モデルの重要な部分を実装するために必要な概念を提供しないからです。OOPもC言語のような手続き型言語と同じ使い方ができると言う人もいます。そして実際にOOPの機能を使えば手続き型言語と同じような使い方もできます。例えば、オブジェクトをデータ構造とみなして、振る舞いを割り当てないようにします。そして、振る舞いはファンクションとしてオブジェクトとは別に実装します。しかし、こうするとデータの意味は開発者にしかわかりません。コードそのものが意味をはっきりと表現しないからです。手続き型の言語で書かれたプログラムは、ファンクションの集合として理解されます。プログラムを走らせれば、ひとつのファンクションが別のファンクションを呼び出しながら処理を進めて結果を出力します。このようなプログラムでは、概念的に関係のあるプログラムの要素をカプセル化できません。また、ドメインとコードを対応づけるのも難しいです。

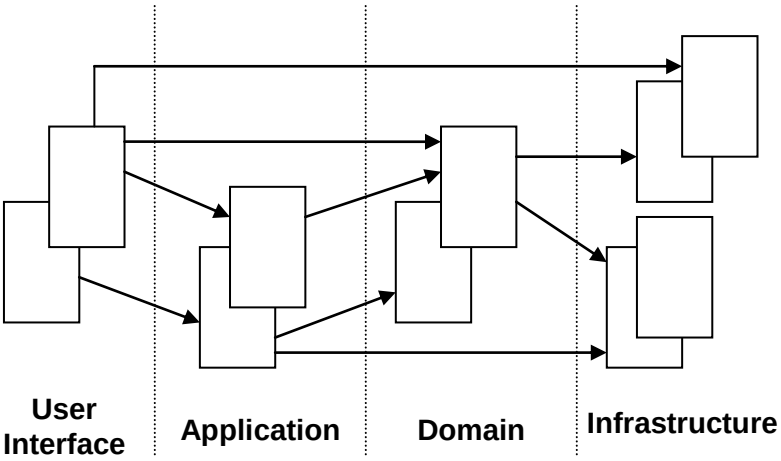
手続き型言語を使って簡単にモデリングし実装できる特定の領域もあります。例えば数学です。ほとんどの数学理論は計算で表現できるので、ファンクションの呼び出しとデータ構造を使って単純に処理を実行できます。しかし、複雑なドメインは計算のような抽象的な概念を集めたものではなく、アルゴリズムの集合に単純化できません。したがって、手続き型言語は様々な種類のドメインを表現するには役不足です。モデル駆動設計での手続き型言語の使用は推奨しません。

モデル駆動設計の基本要素

この節ではモデル駆動設計で使われる最も重要なパターンを紹介します。これらのパターンの目的は、モデリングや設計の主要な概念をドメイン駆動設計の観点から表現することです。次の図はパターンと各パターンの関係を表します。



レイヤアーキテクチャ



アプリケーションを作成するとき、アプリケーションの大部分はドメインと直接関係しません。しかし、そのような部分はインフラを構成し、ソフトウエア自体を支えます。アプリケーション内でドメインに関係する部分は、残りの部分と比べてとても小さくなるかもしれませんが、それで構いません。典型的なアプリケーションは、データベースへのアクセスやファイルやネットワークへのアクセス、ユーザインターフェイス等と関係する多くのコードが含まれるので、当然そうなります。

多くの場合、オブジェクト指向のプログラムでは、UIやデータベースに関係するコードやその他の補助的なコードはビジネスオブジェクトに直接書かれます。さらにビジネスロジックがUIウィジェットやデータベーススクリプトのなかに埋め込まれることもあります。こんなことをしてしまうのは、この方法が最も簡単に素早く目的を達成できるからです。

しかし、ドメインと関係するコードが他のレイヤに混ざると、コードを読んで検討するのがとても難しくなります。また、UIを表面的に変更したら、ビジネスロジックも一緒に変更されてしまうこともあります。ビジネスルールを変更するために、UIのコードやデータベースアクセス部分のコード、またはその他のプログラム要素を丁寧に追いかけて、変更箇所を探す必要があるかもしれません。機能が密着しすぎている実装では、モデル駆動で作成したオブジェクトは役に立ちません。テストも自動化しにくいです。どのような処理に関係するプログラムであっても、常に簡潔にしなければなりません。そうしないと、プログラムは理解できなくなります。

したがって、複雑なプログラムをレイヤに分割する必要があります。各レイヤの凝集度が強く、下位のレイヤだけ依存するレイヤを設計しましょう。下に挙げるのはレイヤ分割の標準的なパターンです。このパターンでは上位のレイヤと緩やかに結合するようになっています。また、ドメインモデルに関係のあるコードをひとつのレイヤに集めて、UIレイヤやアプリケーションレイヤ、インフラレイヤのコードと分離しています。ドメインオブジェクトは、自身を表示したり、保存したり、アプリケーションのタスクを管理したりする責任を負いません。ドメインモデルを表現することに集中できます。こうするとモデルは十分な表現力と明晰さを持ち、ビジネスについての本質的な知識を捉え、与えられた役割を果たすようになります。

ドメイン駆動設計のための一般的なソリューションでは４つの抽象的なレイヤを使います。

ユーザインター フェイス（プレゼンテーションレイヤ）	ユーザに対して情報を表示し、ユーザの命令を解釈する。
アプリケーションレイヤ	アプリケーションの活動を調整する薄いレイヤ。ビジネスロジックを含まない。また、ビジネスオブジェクトの状態を保持しない。しかし、アプリケーションの処理の進み具合を保持することがある。
ドメインレイヤ	ドメインについての情報を含むレイヤ。業務ソフトウエアの心臓部。ビジネスオブジェクトの状態を保持する。ビジネスオブジェクトの永続化処理をインフラストラクチャ層に委託する。まれにビジネスオブジェクトの状態もインフラストラクチャ層に委託することがある。
インフラストラクチャレイヤ	他のすべてのレイヤを補助するライブラリとして働く。レイヤ間の情報のやり取りを制御し、ビジネスオブジェクトの永続化を実装する。また、ユーザインターフェイス等を補助するライブラリを含む。

アプリケーションをレイヤに分割し、レイヤ間の相互作用のルールを確立することが重要です。レイヤがはっきりと分割されていないと、すぐに各レイヤの役割が絡み合うようになり、変更の管理がとても難しくなるでしょう。コードの一部分に小さな変更をひとつ加えただけで、他の部分に思いがけない悪影響が現れるかもしれません。ドメインレイヤはドメインの核心部分に焦点を絞るべきです。インフラストラクチャの機能を含むべきではありません。UIはビジネスロジックやインフラストラクチャレイヤに属する機能と強く結びついてはいけません。多くの場合、アプリケーションレイヤが必要です。アプリケーションレイヤは、ビジスロジックを統括してアプリケーション全体の動作を管理し調整するために必要です。

例えば、典型的なアプリケーションではドメインとインフラストラクチャは以下のようになります。まず、フライトを予約したいユーザがアプリケーションレイヤのサービスに問い合わせます。アプリケーション層は関連するドメイン

オブジェクトをインフラストラクチャから取り出し、関連するメソッドを実行します。例えば、その他の既に予約されているフライトとの間には十分な余裕があるかどうかチェックします. ドメインオブジェクトがすべてのチェックをして状態を”決定”に更新すると、サービスはオブジェクトを永続化してインフラストラクチャに保存します。

エンティティ

一意性を持つものとして分類できるオブジェクトがあります。ソフトウエアの様々な状態の中で、常に同一であり続けるオブジェクトです。これらのオブジェクトにとって問題になるのは属性ではありません。むしろ、同一のオブジェクトが存在し続けることが問題になります。オブジェクトの存在期間はシステムの寿命に及びます。またそれを超えることもあります。このようなオブジェクトをエンティティと呼びます。

OOP言語はオブジェクトをメモリ上に保持し、参照、つまりメモリのアドレスを各オブジェクトに結びつけます。この参照は与えられた時点ではユニークですが、いつまでもユニークであるとは保証されていません。そして実際、参照がユニークであるとは限らないのです。オブジェクトは頻繁にメモリの外へ移動したり戻ってきたりします。また、シリアライズされてネットワークに送り込まれ送信先で再作成されることもあります。破棄されることもあるでしょう。参照はプログラムが実行されている環境内では一意ですが、ここで私たちが考えている意味では一意ではありあません。例えば、気温のような天候についての情報を保持するクラスのインスタンスがふたつ存在して、両方とも全く同じ値をもつことは十分にあります。その場合はオブジェクトの値は完全に同じで両者は交換可能ですが、このふたつのオブジェクトの参照は異なります。こういうオブジェクトはエンティティではありません。

プログラムを使って人物の概念を実装することになったら、おそらくPersonクラスを作成して属性を加えるでしょう。名前、生年月日、出身地等の属性です。しかし、これらの属性で人物を一意に特定できるでしょうか。同姓同名の人物がいるかもしれませんから、名前では一意に特定できません。だから名前だけでは、同姓同名の人物を区別することができません。同じ日に生まれる人もいるので、誕生日でも一意に特定できません。出身地も同様です。同じ属性値を持っていたとしても、オブジェクトは他のオブジェクトと区別できなければなりません。そうでないと、データが汚れてしまいます。

したがって、エンティティをソフトウエアに実装するということは、一意性を作成することです。例えば、人物は属性を組み合わせることで一意性を保証できるかもしれません。名前、誕生日、誕生地、両親の名前、現在の住所等の組

み合わせです。アメリカ合衆国では社会保険番号で国民を一意に識別します。銀行口座を一意に識別するには口座番号で十分でしょう。一意性を保証するのは、オブジェクトの単一の属性や属性の組み合わせや、一意性を保証するための特別な属性や、さらにはオブジェクトの振る舞いの場合もあります。重要なのは、システムが一意性の異なるオブジェクトを簡単に区別できることであり、一意性を保証する属性や振る舞いが同じであれば、そのオブジェクト同士が同一だと判別できることです。そうでないと、システム全体が台無しになるかもしれません。

それぞれのオブジェクトに一意性を付加するには、いくつかの方法があります。例えば、モジュールによって自動的にIDを生成して、ユーザには見えないようにソフトウエアの内部で使う方法です。また、データベースの主キーはデータベース内で一意性を保証しますが、オブジェクトのIDにもなり得ます。オブジェクトがデータベースから取得されたときはいつでも、IDもデータベースから取得してメモリ上に再作成されるからです。空港のシステムの場合は、各空港に結びついた番号を利用してユーザがIDを設定できます。 それぞれの空港は文字列型のIDを持ちます。そのIDは、世界中の旅行代理店が旅行のスケジュールを作成するとき、空港を一意に識別するために使います。また、この他にもオブジェクトのいくつかの属性を使ってIDを作成する方法が考えられます。それらの属性だけではIDを作成できない場合は、さらに他の属性を追加してオブジェクトを識別します。

属性ではなくIDでオブジェクトを識別する場合、モデル内のオブジェクトの定義のなかでもIDを優先して定義します。このとき、クラスの定義をシンプルに保ち、オブジェクトの生存期間と一意性に注目します。そして、オフジェクトの属性や生存期間に依存せずに、それぞれのオブジェクトを識別できる方法を定義しましょう。オブジェクトには適切な属性を持たせなければならないことに注意してください。そして、どんなオブジェクトに対しても一意な値を割り当てる操作を定義しましょう。それはオブジェクトにユニークな記号を割り当てる操作を定義することで実現できるのかもしれません。また、システムの外部から一意性を割り当てる操作でもいいですし、システムによってシステムのために作成される恣意的なIDを使ってもいいかもしれません。しかし、どのような方法にしろ、モデル内のオブジェクトの一意性が持つ特徴に適合しなければなりません。同時にモデルには、オブジェクトの同一性をどのように保証するのか定義する必要があります。

エンティティはドメインモデルにおいて重要なオブジェクトであり、モデリングの始めから考慮するべきものです。オブジェクトをエンティティにするべきかそうでないのかを決めるのも重要です。これは、次のパターンの説明で取り上げます。

バリューオブジェクト

ここまでエンティティについて、そしてモデリングの初期にエンティティを見つけ出すことの大切さについて議論しました。エンティティはドメインモデルに必須のオブジェクトです。では、すべてのオブジェクトをエンティティとして作成すべきでしょうか。すべてのオブジェクトが一意性を保持しなければならないのでしょうか。

すべてのオブジェクトをエンティティとして作成したいと思うかもしれません。エンティティは追跡が可能です。しかし、追跡したり一意性を作成したりするにはコストがかかります。それぞれのインスタンスを一意に識別できることを保証しなければなりませんが、一意性を識別し追跡する処理は単純ではありません。どのように一意性を作成するのか決めるには、たくさんのことを注意深く考えなければなりません。間違った決定をすると同じIDをもつオブジェクトができてしまったり、なにか望ましくない問題が起きてしまうでしょう。また、すべてのオブジェクトをエンティティとして作成すると性能にも影響があります。設計上のオブジェクトはソフトウエア実行時にはひとつのインスタンスになります。例えば、Customerオブジェクトがエンティティであり、このオブジェクトのインスタンスは銀行の特定の顧客ひとりを表します。この場合、このインスタンスは他の顧客の口座に対する操作には再利用できません。そうなるとすべての顧客のためにこのようなインスタンスを作成する必要があります。このようなインスタンスを何千も同時に扱わなければならないとき、システムのパフォーマンスは低下してしまいます。

描画アプリケーションについて考えてみましょう。ユーザはキャンバスを前にして点や、様々な太さ、スタイル、色の線を描きます。このアプリケーションを作成するとき、Pointというクラスを作成すると便利です。プログラムはキャンバス上の点を表すために、このクラスのインスタンスを作成できるというわけです。このような点はふたつの属性を持ち、それらはキャンバス上の座標と結びついています。ではこれらの点は一意性を保持する必要があるでしょうか。他のオブジェクトとの関連はどうでしょう。このオブジェクトにとって関心があるのは自身の座標だけのようです。

ドメインの要素を属性にしなければならないとき、その属性を含むオブジェクトを識別することよりも、どんな属性を含むのかに関心がある場合があります。このようなオブジェクトはドメインのひとつの側面を表現するためのオブジェクトで、一意性を保証する必要はありません。このようなオブジェクトをバリューオブジェクトと呼びます。

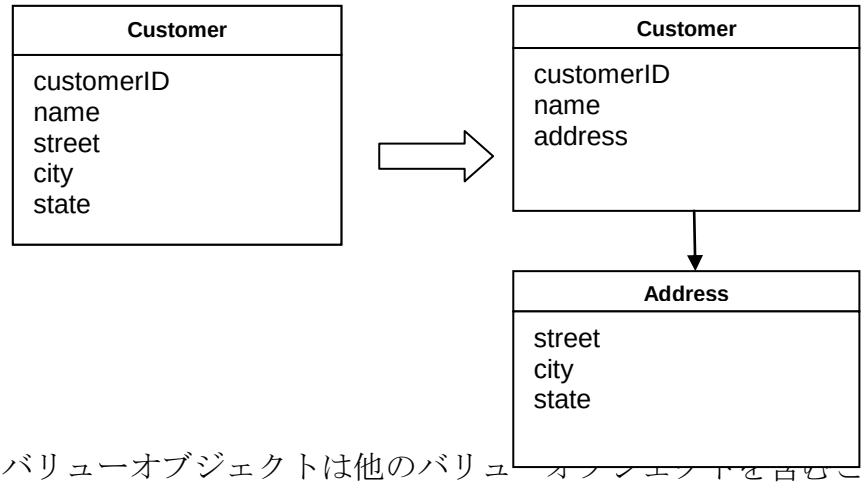
エンティティオブジェクトとバリューオブジェクトを区別しなければなりません。オブジェクトを統一したいがために、すべてのオブジェクトをエンティテ

ィにするのは有益ではありません。それより、エンティティの定義に一致するオブジェクトだけをエンティティとして選びだすほうがいいでしょう。そして残りのオブジェクトをバリューオブジェクトにします。（次の節でもうひとつのオブジェクトの種類を紹介しますが、今はエンティティオブジェクトとバリューオブジェクトしかないと仮定します。）こうすると設計は単純になります。また、他の利点もあります。

一意性を保持しないバリューオブジェクトは簡単に作成し、破棄できます。だれも一意性の確保を気にしなくていいのです。唯一ガーベッジコレクタが、オブジェクトがいつ他のオブジェクトから参照されなくなるかを気にするだけです。設計がとても単純になります。

バリューオブジェクトは変更できないようにするべきです。コンストラクタで作成された後、破棄されるまで変更できないようにします。別の値をもつバリューオブジェクトが必要なら、単純にもうひとつ作成すればいいのです。これは設計に重要な影響を与えます。不変であり一意性をもたないバリューオブジェクトは共有できます。設計によってはオブジェクトを共有しなければなりません。不変のオブジェクトを共有すれば性能を大きく改善できます。また、オブジェクトが変更できないということは、完全性、言い換えればデータの完全性の表明することでもあります。変更できるオブジェクトを共有するとどうなるのか想像してみましょう。例えば、航空予約システムではそれぞれのフライトを表すオブジェクトを作成します。ある顧客がある目的地へのフライトを予約したとします。もうひとりの顧客が同じフライトを予約しようとします。このときシステムはフライトコードを保持するオブジェクトを再利用します。同じフライトを予約しようとしたからです。そうしているうちに、顧客は考えを変えて、違うフライトを予約します。このときシステムはフライトコードを変更してしまいます。変更ができるオブジェクトだからです。その結果、最初の顧客のフライトコードまで一緒に変更されてしまいます。

ここに厳守すべきルールが出来上がります。つまりバリューオブジェクトが共有可能ならば、変更不可能にすること。バリューオブジェクトは小さくてシンプルなオブジェクトにすること。他のオブジェクトがバリューオブジェクトを必要とするときは、単純に値だけを渡してやるか、バリューオブジェクトそのものをコピーして渡してやること。バリューオブジェクトのコピーの作成はとても簡単です。通常は何の影響もありません。一意性を保持する必要がないのですから、好きなだけコピーを作成できます。そして好きなときに破棄できるのです。



バリューオブジェクトは他のバリューオブジェクトを含むことができます。またエンティティの参照さえも含むことができます。バリューオブジェクトは単純にドメインオブジェクトの属性を含むために使われますが、ひとつのバリューオブジェクトにドメインオブジェクトのすべての属性を含む必要はありません。属性はいくつかのオブジェクトにグループ化できます。ひとつのバリューオブジェクトを形成するための一揃えの属性は、ひとつの概念全体を表すべきです。例えば、顧客情報には名前と県と市町村と番地があるとします。この場合、別のオブジェクトが住所情報を保持し、顧客オブジェクトは住所オブジェクトの参照だけを保持するほうが優れた設計になります。上の図に示したように、県と市町村と番地をばらばらに顧客オブジェクトに持たせるよりも、住所オブジェクトにまとめるべきなのは、それらが共通の概念に属しているからです。

サービス

ドメインを分析し、ドメインモデルを構成する主要なオブジェクトを定義しようとすると、ドメインのある側面は簡単にオブジェクトに対応づけられないことがわかります。一般的にオブジェクトは属性を持ちます。また、オブジェクト自身によって管理される内部的な状態を持ち、振る舞いを外部に公開すると考えられています。ユビキタス言語を構築するとき、ユビキタス言語はドメインの主要な概念を取り込みます。ユビキタス言語に含まれる名詞をオブジェクトに対応づけるのは簡単です。動詞は適切な名詞と関連し、オブジェクトの振る舞いの一部になります。しかし、ドメインにはどのオブジェクトにも属さないような振る舞い、つまり動詞がいくつかあります。これらの振る舞いはドメインにとって重要です。無視したり、よく考えずに何らかのエンティティやバ

リューオブジェクトに含めることはできません。しかし、特定のオブジェクトにこの振る舞いを加えれば、そのオブジェクトは台無しになってしまうでしょう。本来はどのオブジェクトにも属さない振る舞いを加えることになるからです。にもかかわらず、オブジェクト指向言語を使うと、ドメインに属する振る舞いでも、何らかのオブジェクトに属さなければなりません。どのオブジェクトにも属していない振る舞いは定義できません。振る舞いは必ず何らかのオブジェクトに属しています。このような振る舞いの多くは、複数のオブジェクトを横断して作用します。ひょっとしたら複数のクラスに対しても作用するかもしれません。例えば、ある銀行口座から他の口座へと送金する場合、送金機能は送金元の口座と送金先の口座のどちらにあるべきでしょうか。どちらにあっても間違っているように思えます。

ドメインにこのような振る舞いが見つかった場合、最もよい方法はこの振る舞いをサービスとして定義することです。サービスは内部に状態を保持せず、単純に機能だけを提供します。サービスが提供する機能は重要です。サービスは特定のエンティティやバリューオブジェクトのための機能をまとめます。サービスはわかりやすく定義したほうがいいでしょう。そうすればドメイン内部の区分が明確になり、概念もカプセル化されます。サービスとして提供される機能を、エンティティやバリューオブジェクトに含めてしまえば混乱が起きるでしょう。このような機能はどのオブジェクトに含めればいいのか明らかでないからです。

サービスは操作を提供するインターフェイスとして働きます。サービスはアプリケーション内の枠組みの中では共通部分になりますが、ドメインレイヤからも使うことができます。サービスとはサービスとしてふるまうオブジェクトのことではありません。サービスは、操作を実行するオブジェクトや操作の対象となるオブジェクトと関係します。サービスは多くのオブジェクトを結びつけるポイントになるというわけです。ポイントになるからこそ、サービスとして定義できる振る舞いをドメインオブジェクトに含めてはなりません。ドメインオブジェクトにこの振る舞いを持たせてしまうと、振る舞いを持つオブジェクトとその振る舞いの結果を受け取るオブジェクトが強く複雑に結びついてしまいます。オブジェクト同士の結合度が強いのはよくない設計の兆候です。コードが理解しにくくなってしまい、さらに変更するのが難しくなるからです。

普通であればドメインオブジェクトに属するような操作をサービスにしてはいけません。また、必要な操作をすべてサービスとして作成するのもよくないです。その操作がドメインの概念のなかで重要であるならば、サービスとして作成すべきです。サービスには次の３つの特徴があります。

１．サービスとして作成される操作はドメインの概念をあらわしているが、エンティティやバリューオブジェクトに含めると違和感がある。

２．サービスとして作成される操作はドメイン内の他のオブジェクトから参照される。

３．サービスとして作成される操作は状態をもたない。

ドメインの重要な操作が、エンティティやバリューオブジェクトに割り当てると不自然な場合は、その処理をサービスとして定義し、独立したインターフェイスとしてモデルに追加しましょう。このインターフェイスはドメインモデルの言語を使って定義します。また、処理の名前もユビキタス言語に含まれていなければなりません。そして、サービスが状態を持たないようにします。

サービスを使う一方で、他のレイヤからドメインレイヤを分離しておくのも大切です。ドメインレイヤに属するサービスなのか、インフラストラクチャに属するのかを見極めるのは難しいです。さらには、アプリケーションレイヤにもサービスが存在することもあります。アプリケーションレイヤは全体の構成を少し複雑にしますが、そこに属するサービスはドメインレイヤに属する相手方のオブジェクトと分離するのがとても難しいです。ドメインモデルを設計するとき、ドメインレイヤが他のレイヤから独立するようにしなければなりません。

アプリケーションレイヤに属するにしろ、ドメインレイヤに属するにしろ、サービスはドメインの中のエンティティやバリューオブジェクトに基づいて作成され、これらのオブジェクトに対して必要な処理を提供しなければなりません。したがって、サービスが属するレイヤを決定するのは難しいです。概念的な観点から考えて、その操作がアプリケーションレイヤに属するのであれば、アプリケーションレイヤにサービスを配置するべきですし、その処理がドメインオブジェクトのための処理で、間違いなくドメインに関係し、ドメインのニーズを満たすための操作であれば、ドメインレイヤに配置します。

実践的な例で考えてみましょう。レポート出力をするWebアプリケーションです。レポートはデータベースのデータを使い、テンプレートを基に作られます。そして、最終的にはブラウザにHTMLページとして表示されます。

WebページはUIレイヤを含みます。UIレイヤはユーザにログイン画面を提供します。また、好きなレポートを選択して、ボタンをクリックしてレポート出力を要求できるようにします。アプリケーションレイヤはUIレイヤと、ドメインレイヤ、インフラストラクチャレイヤの間にある小さなレイヤです。アプリケーションレイヤは、ログイン操作では、データベースを扱うインフラストラクチャレイヤとデータのやり取りをします。また、レポート出力が必要になったら、ドメインレイヤとデータのやり取りをします。ドメインレイヤはドメインの中核部分を含みます。つまりレポートと直接関係するオブジェクトを含みま

す。この中にはレポートオブジェクトとテンプレートオブジェクトがあります。レポートはこのふたつのオブジェクトに基づいています。インフラストラクチャレイヤはデータベースへのアクセスやファイルへのアクセスをサポートします。

ユーザがレポートを出力するとき、ユーザはレポート名の一覧から出力対象のレポート名を選びます。このレポート名がレポート ID になる文字列です。他のパラメータ、例えばレポートに含まれる項目やレポートの出力対象期間も渡しますが、話を簡単にするためにレポート ID だけに着目します。このレポート名は、アプリケーションレイヤからドメインレイヤへ渡されます。ドメインレイヤは、渡された名前のレポートを作成してアプリケーションレイヤへ返す責任があります。レポートはテンプレートに基づいているので、サービスを作成してレポート ID に一致するテンプレートを取得できるでしょう。テンプレートはデータベースかファイルに格納されています。このような操作をレポートオブジェクト自体に割り当てるのは適切ではありません。テンプレートオブジェクトも適切ではありません。したがって、レポート ID をもとにレポートのテンプレートを取得するサービスを別に作成します。このサービスはドメインレイヤに属します。テンプレートを検索するのはファイルシステムの機能を使えばいいでしょう。

モジュール

大規模で複雑なアプリケーションの場合、ドメインモデルはどんどん大きくなっていく傾向にあります。一度に議論するのが困難なほど大きくなると、それぞれの要素の関係や相互作用を理解するのは大変です。このため、ドメインモデルを複数のモジュールの集まりとして構築する必要があります。モジュール化は関係する概念やタスクを組織して複雑性を低減する方法です。

モジュールはほとんどのプロジェクトで広く使われています。巨大なモデルの中からモジュールを見つけて、それらのモジュールの関係を見つければ、そのモデルを簡単に把握できます。モジュール間の相互作用が理解できたら、モジュール内部の詳細を調べ始められます。この方法は複雑さを管理するための単純で効果的な方法です。

モジュールを使うもうひとつの理由は、コードの品質に関係があります。ソフトウエアのコードは高い凝集度と低い結合度を持つべきだ、という考えは広く受け入れられています。凝集度はクラスやメソッドに対して使われていた考えですが、モジュールにも適用できます。関係の強いクラス同士をグループ化してモジュールに含めて、モジュールの凝集度をできるだけ高くするのがいいで

しょう。凝集度にはいくつかの種類があります。最も使われているのは通信的凝集と機能的凝集です。モジュールの各部分が同じデータを操作する場合は通信的凝集になります。同じデータを扱う部分はグループ化したほうがいいように思えます。それらは強く関係しているからです。他方、モジュール全体にまたがってひとつのうまく定義されたタスクを実行する場合は機能的凝集になります。これは最良の凝集だと考えられています。

モジュールを使って設計をするのは、モジュール内の凝集度を高めて、モジュール間の結合度を弱めるためです。モジュールを構成するのは、論理的にも機能的にも関係があり、凝集度が高まる要素にするべきです。また、モジュールは正確に定義されたインターフェイスを持たなければなりません。他のオブジェクトからアクセスできるのはそのインターフェイスだけにするべきです。モジュール内の３つのオブジェクトを呼び出すよりも、ひとつのインターフェイスへアクセスするほうが優れた方法です。モジュール間の結合度が弱まるからです。モジュール間の結合度が弱まれば複雑さも解消し、メンテナンスも容易になります。各モジュールが他のすべてのモジュールと相互作用しているシステムよりも、モジュール間に結びつきが少なく、各モジュールが丁寧に定義されたタスクを実行するシステムのほうが、動作を理解するのが簡単です。

システムの一連の動きをあらわし、関係の強い概念を含むモジュールを作成しましょう。そうすればモジュール間の結合度は弱まります。また、重要な要素を含んでいるモジュールの基礎になる、見落としていた概念を探り出しても、モデルを変更する方法は見つかりません。したがって、 他の概念とは独立して理解でき、議論できるような弱く結びついた概念の集まりを探しましょう。モデルの精度を上げて、ドメイン内の高いレベルの概念でモデルが分割できるようにします。そうすれば対応するコードも同じように疎結合になります。

モジュールにはユビキタス言語に含まれる名前を付けてください。モジュールとその名前はドメイン内部に対する洞察を反映していなければなりません。

設計者は最初からモジュールを作成するのに慣れています。設計ではモジュールは一般的なものだからです。モジュールの役割が決まった後は、その役割は変更しません。他方、モジュールの内部は盛んに変更されるでしょう。モジュールには柔軟性を持たせ、プロジェクトとともに進化させましょう。凍結してはいけません。クラスのリファクタリングよりも、モジュールのリファクタリングのほうがコストがかかりますが、モジュールの設計ミスが見つかった場合、回避する方法を見つけるよりもモジュールを修正するほうがいいでしょう。

アグリゲート

本章の最後の３つのパターンは、これまでとは異なるモデリングの難題に対処するパターンです。ひとつはドメインオブジェクトのライフサイクルに関係します。ドメインオブジェクトは生存期間中に様々な状態をとります。作成され、メモリ上に配置され、利用され、そして破棄されます。データベースのようなシステムの中に保存されて永続化される場合もあります。保存されたデータは後から取り出したり、アーカイブできます。そして、どこかの時点でデータベースやアーカイブストレージを含むすべてのシステムから完全に削除されることもあります。

ドメインオブジェクトのライフサイクルの管理は、それ自体に難題を含んでいます。適切に管理できなければ、ドメインモデルに悪影響を与えるかもしれません。ここでは、ライフサイクルを管理するのに役立つ３つのパターンを説明します。アグリゲートはオブジェクトの所有権と境界を定義するのに使うパターンです。ファクトリとリポジトリはオブジェクトの作成と保管に役立つパターンです。まずはアグリゲートから説明しましょう。

ひとつのモデルはたくさんのドメインオブジェクトを含むことができます。どんなによく考えて設計しても、多くのオブジェクト同士が互いに関連して、複雑な関係の網目が出来上がります。関連にはいくつかの種類がありますが、モデル内に広がるどんな関連でも、ソフトウエアとして実現する仕組みが必要です。実際にはドメインオブジェクト同士の関連はコードになり、さらには多くの場合データベースにも反映されます。例えば、顧客とその顧客が自身の名義で開設した口座の一対一の関係は、ふたつのオブジェクト間の参照関係として表現されます。そして、ふたつのデータベーステーブルの関係として実装されます。ひとつは顧客の情報を、もう１つは口座の情報を保持します。

モデルを作成するとき、完璧なモデルを作り上げるのが課題になることはほとんどありません。むしろ、可能な限り単純で理解しやすいモデルにする必要があります。作業時間の多くは、モデルの要素間の関係を取り除いたり、簡単にする作業に費やされます。そして、この作業はドメインを深く理解するまで続きます。

オブジェクトの間に１対多の関連があるとモデルは複雑になります。関係のあるオブジェクトが多くなるからです。オブジェクト同士の関連を、１つのオブジェクトとオブジェクトのコレクションの１対１の関係に変更することで、複雑な関係は単純になります。しかし、この方法はいつも使えるとは限りません。

多対多の関連もあります。それらの多くは双方向に通信しています。このような関連があるとモデルはとても複雑になり、オブジェクトのライフサイクル管理もとても難しくなります。関連はできるだけ少なくすべきです。まず、モデルにとって本質的でない関連は排除しましょう。これらの関連はドメインには存在しているかもしれませんが、作成しているモデルには必要ないので排除します。次に、制約を導入して多重の関係を解消します。多数のオブジェクトが関連している場合、関連に適切な制約を与えることで、１つのオブジェクトだけが関連を持つように変更できます。そして、多くの場合、双方向の関連は一方向の関連へと変更できます。例えば、自動車はそれぞれエンジンを持っています。そしてすべてのエンジンは車に搭載されています。この関連は双方向ですが、自動車がエンジンを持ちエンジンから自動車への関係を考えないようにすれば、簡単に単純な関係にできます。

オブジェクト同士の関連を少なく単純にしても、まだ多くの関連が残っているでしょう。例えば、銀行システムは顧客情報を扱います。顧客情報には顧客の個人情報が含まれています。名前、住所、電話番号、職業、口座情報（口座番号、残金、操作履歴）等です。顧客情報をアーカイブに入れる、あるいは完全に削除する場合、顧客情報へのすべての参照を確実に削除しなければなりません。多くオブジェクトがこれらの参照を保持していたら、すべてを確実に削除するのは難しいです。顧客情報を変更するときも、システムはその変更がシステム全体に反映して、データの完全性を保証しなければなりません。ふつうこのような問題はデータベースレベルまで残されます。データベースでは、データの完全性を保証するためにトランザクションを利用してこの問題を解決します。しかし、モデルが注意深く設計されていなかったら、トランザクションの競合が頻繁に発生して、パフォーマンスが低下するでしょう。たしかにデータベースのトランザクションはデータの完全性を保証する上でとても重要な役割を果たしますが、もっと望ましいのはデータの一貫性についての問題をモデル内で直接解決することです。

また、データの不変性も確保する必要があります。不変性とはいつデータが変更されても維持しなければならない決まりのことです。多くのオブジェクトが変更対象のオブジェクトの参照を保持していると、不変性の維持は難しくなります。

オブジェクト同士が複雑に関連しているモデルでは、オブジェクトの変更の一貫性を保証するのは難しいです。多くの場合、不変性は互いに離れているオブジェクトではなく、強く結びついたオブジェクト同士に適用されます。しかし、注意深くつくられた排他制御の仕組みがあると、複数のオブジェクトの間に無駄な干渉を作りだし、使いにくいシステムになります。

このような様々な困難を解決するため、アグリケートを使います。アグリケートは関連するオブジェクトの集まりで、データの変更について一括して扱うこ

とができます。あるアグリゲートを他のアグリゲートから区別するのは、そのアグリゲートに含まれるオブジェクトと含まれないオブジェクトの間にある境界線です。それぞれのアグリゲートはひとつのルートを持ちます。ルートになるのはエンティティで、アグリゲートの外部からアクセスできる唯一のオブジェクトです。ルートはアグリゲート内のオブジェクトの参照を保持します。アグリゲート内のオブジェクトは互いに参照を持つことができますが、アグリゲートの外部のオブジェクトはルートの参照しか保持できません。アグリゲートの内部にルート以外のエンティティがある場合、そのエンティティはそのアグリゲート内で一意であればいいでしょう。

ではアグリゲートはどのようにデータの完全性を保証し、不変性を維持するのでしょうか。アグリゲートの外部にあるオブジェクトがルートの参照しか保持できないということは、アグリゲート内のオブジェクトを直接変更できないということです。できるのはルートの変更か、ルートに何らかの処理を要求することだけです。ルートはアグリゲート内の他のオブジェクトを変更できますが、この変更はアグリゲート内の処理なのでコントロールできます。ルートが削除されメモリ上からも取り除かれた場合、アグリゲート内の他のオブジェクトも削除されます。アグリゲート内のオブジェクトを参照するオブジェクトがなくなるからです。アグリゲートの内部のオブジェクトに対して間接的に作用するような変更かルートに対しておこなわれても、その変更はルートが制御できるので、不変性を維持するのは簡単です。アグリゲートの外部のオブジェクトが内部のオブジェクトに直接アクセスできると不変性の維持は難しくなります。そのような場合に不変性を維持するには、外部のオブジェクトに不変性を維持するためのロジックを追加して対処することになりますが、その方法は望ましくありません。

ルートはアグリゲート内部のオブジェクトの一時的な参照を外部のオブジェクトへ渡すことができます。この場合、外部のオブジェクトは受け取った参照を処理が終わった後も保持してはなりません。この方法はバリューオブジェクトのコピーを外部のオブジェクトに渡せば簡単に実現できます。この方法を使えば外部に渡したオブジェクトがどうなろうと問題ありません。アグリゲート内部の完全性には影響がないからです。

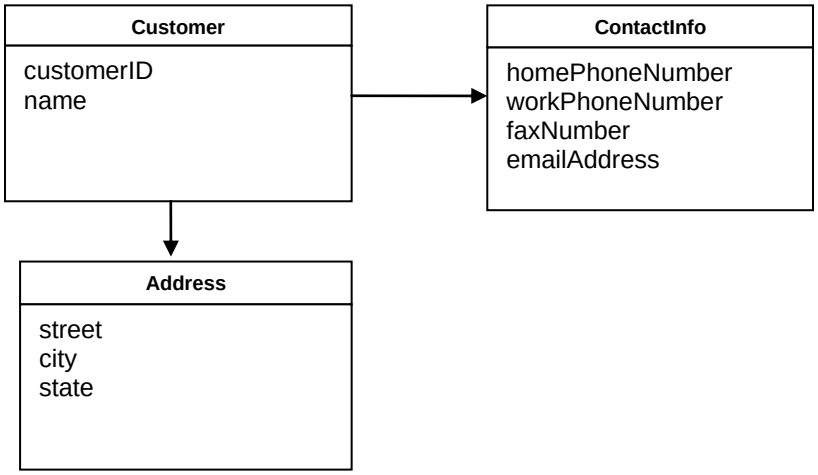
アグリゲートに含まれるオブジェクトがデータベースに保存される場合、外部のオブジェクトがクエリを使って取得できるのはルートオブジェクトだけです。ルート以外のオブジェクトは関連をたどって取得しなければなりません。

アグリゲート内のオブジェクトは他のアグリゲートのルートの参照を保持してもかまいません。

ルートとなるエンティティはシステム全体で一意です。また、不変性の保持に責任を持ちます。その他のオブジェクトはアグリゲート内部で一意です。

エンティティとバリューオブジェクトを集めてアグリゲートにして、それぞれのアグリゲートの境界を決めましょう。そしてそれぞれのアグリゲートでルートとなるエンティティを選び出し、必ずルートを通して内部のオブジェクトへアクセスするようにします。外部のオブジェクトにはルートの参照だけを持たせるようにしましょう。内部のオブジェクトの参照は1回の処理で使う場合にのみ、一時的な参照として外部のオブジェクトへ渡せます。こうすることでどんな変更が加えられても、アグリゲート内のオブジェクトやアグリゲート全体の不変性を維持できます。

次の図はアグリゲートの簡単なサンプルです。このアグリゲートのルートは Customer オブジェクトで、他のオブジェクトはすべてアグリゲートの内部に含まれています。もし Address オブジェクトが必要なら、そのコピーを外部のオブジェクトに渡すことができます。



ファクトリ

エンティティとアグリゲートは大きく複雑になりがちです。ルートとなるオブジェクトのコンストラクタでは構築できないくらい複雑になってしまいます。それに複雑なアグリゲートを作成することはドメインがおこなうべき処理と矛盾します。ドメインのあるものは別の場所で作られます（例えば電子機器は工場で作られます）。複雑なアグリゲートをコンストラクタで作るのは、プリンタが自分自身を作るようなものです。

クライアントとなるオブジェクトが他のオブジェクトを作成したいとき、作成対象のオブジェクトのコンストラクタを呼び出します。パラメータを渡すこともあるかもしれません。しかし、オブジェクトの作成が複雑な処理の場合、そのオブジェクトの内部構造や、内部に含むオブジェクトとの関係や、それらに

適用されているルールについての詳細な知識が必要です。つまり、クライアントのオブジェクトは作成対象のオブジェクトの具体的な知識を保持します。しかし、これはドメインオブジェクトやアグリゲートのカプセル化を無視します。例えば、クライアントがアプリケーションレイヤに属している場合、作成対象のドメインオブジェクトの知識がドメインレイヤの外側で必要になり、設計全体が汚くなってしまいます。実生活でいえば、プラスチックやゴムや金属、シリコンを与えられて、プリンタを手作りするようなものです。不可能ではないでしょうが、そうすることに価値があるのでしょうか。

オブジェクトの生成は主要な操作のひとつですが、生成の際に複雑な操作が集中すると場合、そのような操作は作成されるオブジェクトの責務として不適切です。そのような責務を割り当ててしまうと汚い設計になり、理解しにくくなります。

このような問題を解決するためには新しいパターンを導入する必要があります。オブジェクト作成の複雑な操作をカプセル化するのに役立つパターンです。このパターンを**ファクトリ**と呼びます。ファクトリはオブジェクトの作成に必要な知識をカプセル化します。またアグリゲートの作成にも便利です。アグリゲートのルートを作成するとき、そのアグリゲートに含まれるすべてのオブジェクトを作成して、不変性を適用できるからです。

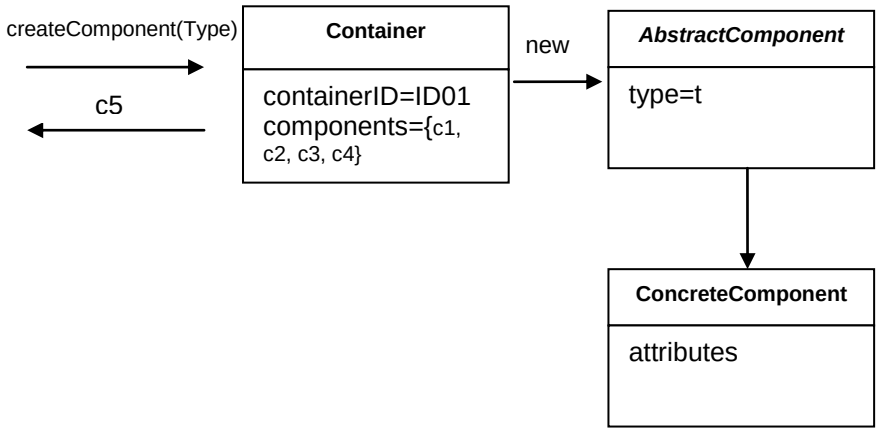
オブジェクトの生成過程を分割不可能にすることが重要です。そうしないと、オブジェクトの生成処理が途中で終わってしまい、オブジェクトがはっきりしないままになってしまう可能性があります。これは、オブジェクトの生成よりもアグリゲートの生成にとって重要なことです。ルートが作成されたとき、不変性の維持の対象になるオブジェクトも同時に作成しなければなりません。さもなければ不変性は維持できないでしょう。バリューオブジェクトは後から値を変更できないので、この生成過程ですべての属性を有効な値に初期化します。オブジェクトが適切に生成できないのなら、例外を発生させ、無効な値が返却されないようにしなければなりません。

複雑なオブジェクトやアグリゲートの生成は、別のオブジェクトの責務にしましょう。そのオブジェクトはドメインモデル内では何の役割も果たしませんが、ドメインの設計には含まれます。このオブジェクトはすべての複雑な部品をカプセル化するインターフェイスを提供します。クライアントには、作成対象のオブジェクトへの参照は必要ありません。そして、アグリゲート全体をひとつのかたまりとして生成して、不変性を維持します。

ファクトリを実装するためのデザインパターンがいくつかあります。Gammaらのデザインパターンの本には詳細が記述されていますが、その中からふたつのパターンを紹介しましょう。ファクトリメソッドとアブストラクトファクトリ

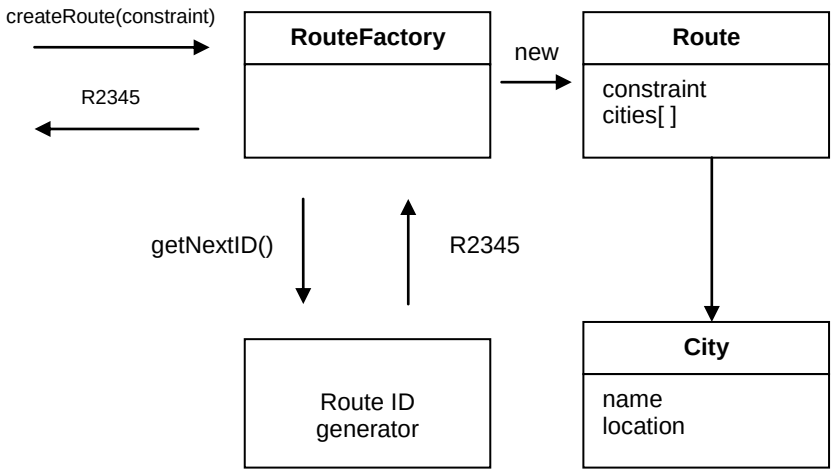
です。ここではクラス設計の観点からではなく、ドメインモデリングの観点から紹介します。

ファクトリメソッドは他のオブジェクトを作成するのに必要な知識を保持し隠蔽するメソッドです。このパターンがとても役に立つのは、クライアントがアグリゲートに属するオブジェクトを生成したいときです。たとえば、次のような使い方があります。アグリゲートのルートにメソッドを追加し、そのメソッドがオブジェクトを生成して不変性を維持しながら、作成したオブジェクトの参照やコピーを返します。



上の図を見てください。コンテナは複数のコンポーネントを保持し、それらのコンポーネントの型は具体的なクラスです。コンポーネントは作成された時に自動的にコンテナに属さなければなりません。クライアントはコンテナの createComponent(Type t) メソッドを呼びます。コンテナはコンポーネントのインスタンスを新規に作成します。コンポーネントの具体的な型はメソッドの Type型パラメータで決まります。作成された後、コンポーネントはコンテナが保持するコレクションに追加されます。そしてクライアントにはそのコピーが返されます。

オブジェクトの生成がもっと複雑な場合や、オブジェクトの生成が他の複数のオブジェクトの生成を含む場合があります。例えばアグリゲートの生成です。アグリゲートが含まなければならない内部オブジェクトの作成を隠蔽するのは、専用のファクトリオブジェクトです。次のようなプログラムの例を考えてみましょう。プログラムは出発点から目的地までの自動車の走行経路を算出します。走行経路には条件が与えられます。ユーザはこのアプリケーションを利用するため Web サイトにログインして、最も短い経路、最も速く到着する経路、最も料金の安い経路からひとつの条件を選択します。算出された走行経路には、保存する必要のあるユーザの情報を付加できます。こうすることで、ユーザが再びログインしたときに以前に算出した走行経路を検索できます。



上の図を見てください。Route ID generatorは走行経路のためのユニークなIDを生成します。IDはエンティティに必要です。

ファクトリを作成する場合、どうしてもオブジェクトのカプセル化を破らざるを得ませんが、これは慎重に行う必要があります。オブジェクトの作成ルールや不変性に影響するような変更をする場合はいつでも、そのオブジェクトのファクトリも更新して新しい状態に対応しなければなりません。ファクトリは作成対象のオブジェクトと密接に関係します。これは弱点ですが、よいこともあります。アグリゲートは密接な関係をもつ複数のオブジェクトを含みます。ルートの作成はアグリゲート内の他のオブジェクトの作成と関係しています。つまり、ルートの作成にはアグリゲートを作成するロジックが含まれます。しかし、このようなロジックは何らかのオブジェクトに自然に含むことができません。他のオブジェクトを作成するロジックだからです。したがって、アグリゲート全体を作成する特別なファクトリクラスを使うのが適切です。このクラスはアグリゲートを正しく作成するためのルールや制約や定数を含んでいます。このファクトリクラスを使えば、オブジェクトは複雑な作成ロジックで汚くなることはありません。単純で特定の目的にのみ奉仕するオブジェクトになります。

エンティティのファクトリとバリューオブジェクトのファクトリは異なっています。バリューオブジェクトは属性値が変化しません。必要な属性には、オブジェクトを作成する時に値を設定する必要があります。オブジェクトが作成された時には、既に有効な状態であり、また最終的な状態になっていなければなりません。つまり、作成後は変更しません。他方、エンティティは属性値が変化します。すべての不変性を守りながら、属性に値を設定することで、作成後も変更できます。また、エンティティは一意性を持ち、バリューオブジェクトは持たないことからもファクトリは違うものになります。

ファクトリが必要なく、単純なコンストラクタで十分なときもあります。たとえは下記のような場合です。

- オブジェクトの作成処理が複雑でない。
- オブジェクトの作成処理内で他のオブジェクトを作成しない。必要な属性はすべてコンストラクタで設定する。
- クライアントはオブジェクトの実装に興味がある。例えば、ストラテジパターンを使って処理を選択したい。
- クラスは具体的な型であり、その型から派生しているクラスはない。したがって具体的な実装を選ばなくていい。

もうひとつ注意しなければならないことがあります。それは、ファクトリがオブジェクトをゼロから作成するのか、それとも以前に作成されていてデータベースに保存されているオブジェクトを再作成するのか、ということです。データベースからメモリ上へエンティティを戻すのは、新しいオブジェクトを作成するよりもあらゆる点で難しい処理を含んでいます。明らかに違うのは、新たに一意性を付与する必要はないということです。すでに保持しているからです。また、不変性を侵した場合、修復するのは難しいです。オブジェクトをゼロから作成した場合、不変性がおかされていると最終的には例外が発生します。しかし、データベースから再作成したオブジェクトではそういうわけにはいきません。オブジェクトを使えるようにするには、どうにかして修復をしなければなりません。さもなければ、データは失われてしまいます。

リポジトリ

モデル駆動設計では、オブジェクトは作成されて始まり、破棄または保存されて終わるライフサイクルを持ちます。コンストラクタ、またはファクトリがオブジェクトを作成します。オブジェクトの作成はオブジェクトを使えるようにすることだけを目的にしています。オブジェクト指向言語では、オブジェクトを使うには、そのオブジェクトの参照を保持しなければなりません。クライアントが参照を保持するには、直接オブジェクトを作成するか、または他のオブジェクトとの関連をたどって参照を手に入れる必要があります。例えばアグリゲート内のバリューオブジェクトを取得するためには、クライアントはそのアグリゲートのルートへ要求します。問題はクライアントがルートの参照を保持していなければならないということです。大規模なアプリケーションではこれは問題になります。クライアントは必要とするオブジェクトや、必要なオブジェクトへの参照を持つオブジェクトへの参照を確実に保持しなければならないからです。この規則を守るように設計すると、オブジェクトはもともと保持しなくてもいいかもしれないオブジェクトへの参照を保持しなければなりません。

そうなるとオブジェクト間の結合が強くなります。そして、必要のない関連が生み出されてしまいます。

オブジェクトを使うということは、そのオブジェクトは既に作成されているということです。そのオブジェクトがアグリゲートのルートであるなら、それはエンティティであり、データベースなどの永続化層に保存されるでしょう。バリューオブジェクトならエンティティの関連をたどって手に入れることができるかもしれません。つまり、多くのオブジェクトはデータベースから直接取得できます。これでオブジェクトの参照を取得するときの問題が解決します。クライアントがオブジェクトを使いたくなったら、データベースにアクセスしてオブジェクトを取得して使えばいいのです。これは単純で時間もかからない解決策のように思えます。しかし、設計には悪影響を与えます。

データベースはインフラストラクチャの一部です。下手な解決策を適用すると、クライアントはデータベースへのアクセス処理を詳しく知らなければならなくなります。例えば、必要なデータを取得するために、クライアントはSQL文を作成しなければなりません。また、データベースに問い合わせた結果のレコードセットには、クライアントには必要のない詳細なデータが含まれているかもしれません。多くのクライアントが、データベースから直接オブジェクトを作成しなければならないと、ドメイン全体にデータベースへアクセスするコードがまき散らされます。その結果、ドメインモデルは汚くなってしまいます。本来はドメインの概念だけを扱えばいいのに、インフラストラクチャに対する細かな処理をしなくてはなりません。さらに、アプリケーションの基盤となるデータベースを変更することになったらどうなるでしょうか。新しいデータベースにアクセスできるようにするために、ドメインモデル内にばらまかれたデータアクセスのコードをすべて修正しなければなりません。また、クライアントがデータベースへ直接アクセスしてオブジェクトを取得すると、クライアントは取得したオブジェクトをアグリゲート内へと戻せます。そうなると、知らないうちにアグリゲートのカプセル化が破られてしまいます。

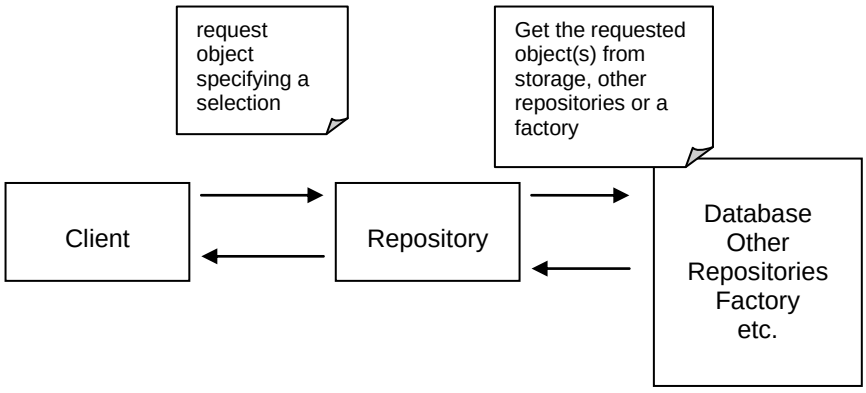
クライアントが必要としているのは、既存のオブジェクトの参照を取得する便利な方法です。インフラストラクチャが簡単な方法を提供してくれるのなら、クライアントの開発者はあちこちを参照できるような関連を付け加えるかもしれません。その結果、混乱したモデルになってしまうでしょう。逆に、SQL文を使って必要なデータをデータベースから直接取得するかもしれません。つまり、アグリゲートのルートを介さずに特定のオブジェクトを取得してしまうということです。そうなると、ドメインのロジックはSQL文やクライアント内に移動し、エンティティもバリューオブジェクトも単なるデータのコンテナになってしまいます。ほとんどのデータベースアクセス処理は技術的な複雑さを含んでいますが、それがクライアントのコードに浸透すると、開発者はその複雑さを低減しようとしてドメインレイヤを簡単な実装にします。しかし、そうす

ればドメインレイヤはドメインモデルとは無関係になります。ドメインの焦点は失われ、設計は不完全になります。

こうした問題を解決するために、リポジトリパターンを使います。リポジトリはオブジェクトの参照を取得するのに必要なロジックをすべてカプセル化するためのパターンです。リポジトリパターンを使えば、ドメインオブジェクトは必要な他のドメインオブジェクトの参照を取得するために、インフラストラクチャを使って直接アクセスする必要はなくなるでしょう。その代わりにリポジトリから参照を取得すればいいのです。モデルもきれいなままで本来の役割を果たすことができます。

リポジトリはオブジェクトの参照を保持できます。オブジェクトが作成されたとき、そのオブジェクトをリポジトリに保存して、あとで使うときはリポジトリから取得できます。また、クライアントがリポジトリにオブジェクトを要求したとき、リポジトリがオブジェクトを持っていなかったらデータベースから取得します。どちらの場合も、リポジトリはどこからでもアクセスできる、オブジェクトの保管場所として働きます。

リポジトリはストラテジパターンを含むこともあります。この場合、特定のストラテジに基づいてアクセスする永続化ストレージを決定します。異なる型のオブジェクトのために別のストレージを使ってもいいでしょう。こうすると、ドメインモデルからオブジェクトやその参照の保管処理、そして基盤となる永続化層へのアクセスを分離できます。

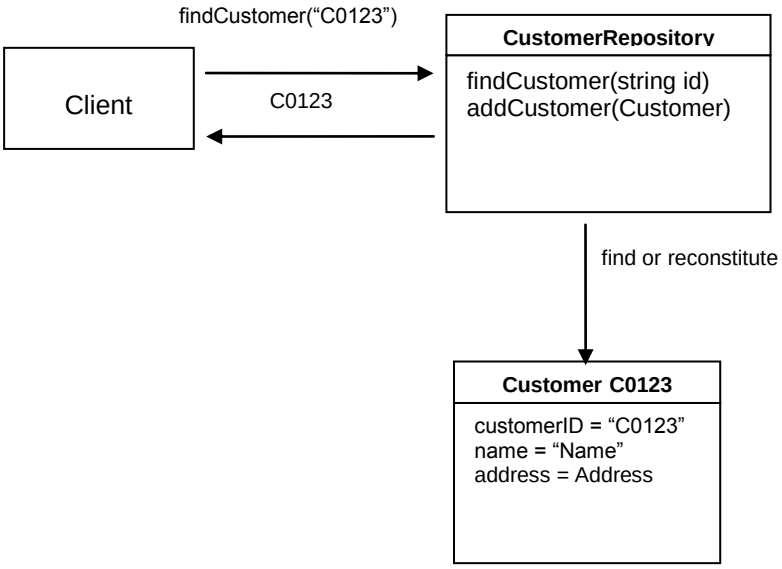


どこからでもアクセスできなければならないオブジェクトのために、そのような型のオブジェクトをすべて格納できるコレクションがメモリ上にあるように見せかけ、そのコレクションを保持するオブジェクトを作成します。そのオブジェクトにはわかりやすいインターフェイスを通じてどこからでもアクセスで

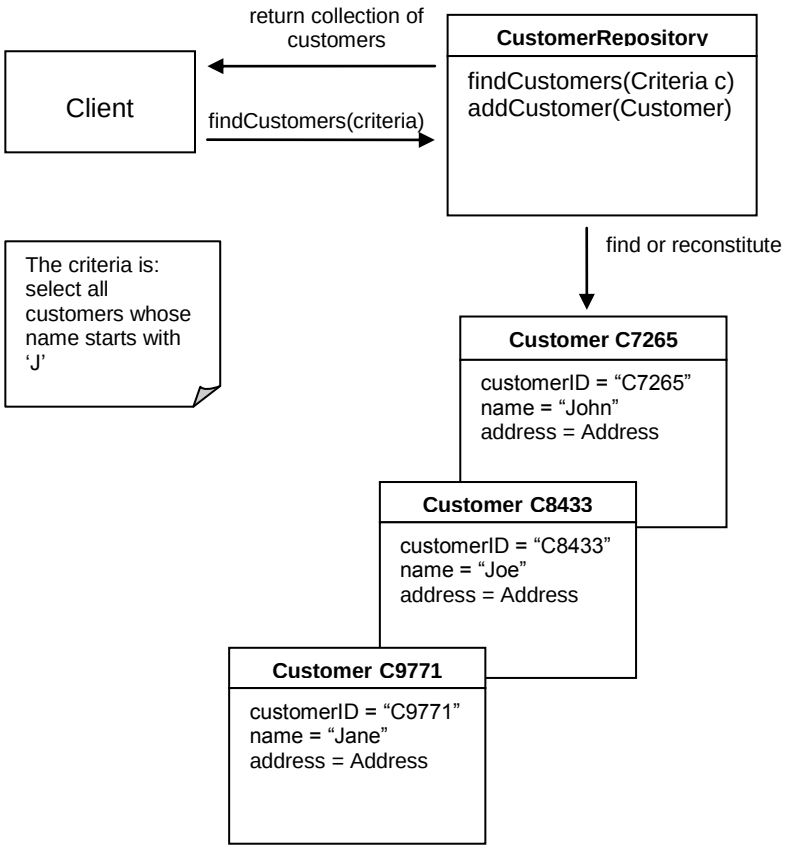
きるようにします。そして、オブジェクトを追加、または削除するメソッドを提供します。これらのメソッドは実際のデータストアへのデータの挿入や削除をカプセル化します。また、何らかの条件をもとにオブジェクトを取得するメソッドを提供します。このメソッドはその属性値が条件に適合するオブジェクトやオブジェクトのコレクションを返します。こうすることで、実際のオブジェクトの保管や問い合わせ処理をカプセル化します。リポジトリへ直接アクセスする必要があるのはアグリゲートのルートだけになるように設計しましょう。クライアントはモデルに集中して、オブジェクトの保管やオブジェクトの取得はすべてリポジトリに任してしまいましょう。

リポジトリはインフラストラクチャへアクセスするための詳細な情報を含みますが、インターフェイスは簡素にすべきです。リポジトリはオブジェクトを取得するためのメソッドを揃えておかなければなりません。クライアントはこのようなメソッドを呼び出していくつかのパラメータを渡します。これらのパラメータは取得条件をあらわし、条件に合致するひとつのオブジェクトあるいは複数のオブジェクトを取得するのに使います。エンティティはメソッドにIDを渡せば簡単に取得できます。他の取得条件はオブジェクトの属性の組み合わせで作ることができます。リポジトリはすべてのオブジェクトと与えられた条件を比較して、条件を満たすオブジェクトを返します。またリポジトリは補助的な計算をするインターフェイスを含むこともできます。例えば、特定の型のオブジェクトの数を数えるメソッドです。

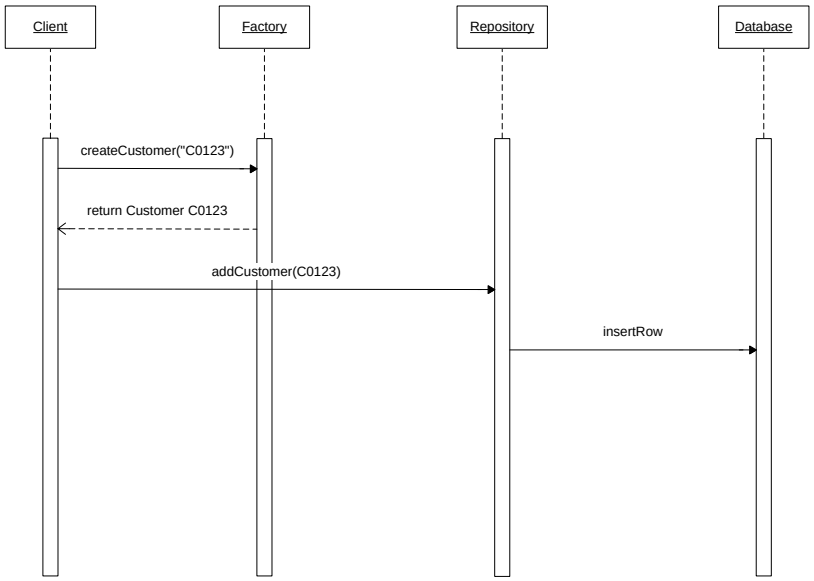
リポジトリの実装はインフラストラクチャによく似ていることに気付くかもしれません。しかし、リポジトリのインターフェイスは完全にドメインモデルの要素になります。



もうひとつの方法は、取得条件を仕様として定義することです。仕様には複雑な取得条件を定義することができます。例えば下図のような条件です。



ファクトリとリポジトリには関係があります。両方ともモデル駆動設計で使われるパターンであり、ドメインオブジェクトのライフサイクルを管理するのに役立ちます。ファクトリがオブジェクトの作成を管理する一方、リポジトリは既存のオブジェクトを管理します。リポジトリはオブジェクトをキャッシュするかもしれませんが、多くの場合は永続化ストレージからオブジェクトを取得してくる必要があります。つまり、オブジェクトはコンストラクタを使って作成されるか、またはファクトリを通じて作成されます。そう考えると、リポジトリがファクトリのように見えるかもしれません。リポジトリもオブジェクトを作成するからです。ゼロからオブジェクトを作成するわけではありませんが、既存のオブジェクトを再作成します。しかし、リポジトリとファクトリを一緒にするべきではありません。ファクトリは新しいオブジェクトだけを作成するべきですし、リポジトリは作成済みのオブジェクトの取得だけをおこなうべきです。新しいオブジェクトをリポジトリに追加する場合は、まずファクトリを使ってオブジェクトを作成し、それからそのオブジェクトをリポジトリに渡して保存します。一連の流れは下図の通りです。



ファクトリとリポジトリの違いをあらわすもうひとつの特徴は、ファクトリはドメインの要素であり、ドメイン以外とは関係を持っていないということです。一方でリポジトリはインフラストラクチャ、例えばデータベースと結びきます。

4

リファクタリングのためのさらに深い洞察

継続的なリファクタリング

ここまでドメインについて、そしてドメインを表すモデルを作成することの重要さについて議論してきました。また、役に立つモデルを作成するためのテクニックについてのガイドラインを紹介してきました。モデルは対象となるドメインと強く結びつかなければなりません。また、モデルに基づいてコードを設計して、モデルは設計上の決定に従って改善していく必要があることも説明しました。モデルを無視した設計は、対象となるドメインを正確にあらわさないソフトウエアを作り出してしまいます。期待している振る舞いをしないかもしれません。設計からのフェードバックなしでモデリングをしたり、開発者抜きでモデリングをすると、実装担当者にとって理解しにくいモデルになってしまいます。使い慣れた技術が使えないかもしれません。

設計と実装では、時々作業を止めてコードを見直す必要があります。つまり、リファクタリングする時間が必要です。リファクタリングはアプリケーションの振る舞いを変えずにコードを再設計し改善する作業です。多くの場合、制御できる小さな範囲で、段階を踏みながら細心の注意を払ってリファクタリングをするので、振る舞いを変更したりバグを埋め込んだりはしません。リファクタリングの目的はコードを改善することであり、悪くすることではありません。自動化されたテストはコードがおかしくなっていないことを保証するのにとても役立ちます。

コードのリファクタリングには様々な方法があります。リファクタリングパターンも存在します。そのようなパターンがあるということは、リファクタリングを自動化する方法があるということです。このようなパターンを元にして作られたツールもあり、開発者の作業を簡単にしてくれます。このようなツールなしではリファクタリングはとても難しい作業になるかもしれません。コードを対象とするリファクタリングが扱うのは、大量のコードとその品質だからです。

リファクタリングにはもうひとつの種類があります。それはドメインとそのモデル対するリファクタリングです。このリファクタリングをおこなうと、ドメインに対して新しい洞察が得られることがあります。例えば、ドメインの要素が明確になったり、ふたつの要素の間に新しい関係が見つかることもあります。このような改善は、設計段階でリファクタリングを通じておこなわなければなりません。表現に富み、読みやすいコードを書くことは重要です。そのコードを読めば何をしているのかがわかるだけではなく、なぜそれをしているのかも理解できなくてはなりません。そのようなコードだけがモデルの真の本質を捉えることができます。

コードを対象としたリファクタリングはパターン化できますし、組織化し体系化することができます。しかし、モデルに対する、さらに深い洞察を得るためのリファクタリングはそうはいきません。パターンを作ることができないのです。モデルは複雑で様々な形態をとります。したがって、機械的な方法でモデリングに取り組むことはできません。よいモデルは深い思慮と洞察、そして経験と直感によって生み出されるのです。

モデリングについて最初に学んだのは、仕様書を読んで名詞と動詞を見つけることでした。見つかった名詞はクラスになり、動詞はメソッドになります。この作業は要件を単純にして、おおまかなモデルを作り出します。モデリングを開始したときは、どんなモデルでも深みが欠けているものですが、より深い洞察の獲得に向けてリファクタリングをすればいいのです。

設計には柔軟さが必要です。柔軟さのない設計はリファクタリングを妨げます。柔軟さを考慮せずに記述したコードは扱うのに苦労します。そのようなコードは変更するときはいつでもコードと格闘しなければなりませんし、リファクタリングに費やす時間も多くなります。

設計が、問題点のない基本的な構成要素によって組み立てられ、適切な用語が使われていれば、開発者の作業はある程度順調に進むでしょう。しかし、適切なモデルを見つけ出すという難題は残っています。適切なモデルとは、ドメインの専門家の複雑な関心事までとらえ、実践的な設計の基礎になるようなものです。表面的ではなく、ドメインの本質をとらえたモデルは深いモデルです。深いモデルに基づいて作成するソフトウエアは、ドメインの専門家の思考方法とより調和し、ユーザの要求にも確実に応答できるでしょう。

これまで、リファクタリングは技術的な要求を満たすためのコード変更という観点から説明されてきました。しかし、リファクタリングするのはドメインをより深く理解し、その理解に応じてモデルやコードを洗練するためでもあるのです。

洗練されたドメインモデルを作成するには反復的なリファクタリングが欠かせません。そしてそのようなリファクタリングにはドメインの専門家とドメインを理解しなければならない開発者の密接な関与が必要です。

鍵となる概念を明らかにする

リファクタリングは少しずつ進めていきます。小さな改善の積み重ねが、リファクタリングの結果になります。小さな変更をたくさん加えても、設計がほとんど改善しない場合もあります。しかし、わずかな変更で大きく改善することもあります。このような大きな変化がブレイクスルーなのです。

まず粗い、浅いモデルから始めて、そのモデルに洗練を加え、ドメインについてのより深い知識とより良い理解に基づいた設計へと磨き上げていきます。新しい概念や抽象を付け加え、それからリファクタリングします。このように洗練を加えていくことで設計はますます明瞭になります。そして、このような洗練がブレイクスルーを準備します。

多くの場合、ブレイクスルーが起こると、考え方、つまりモデルの見方が変化します。またブレイクスルーによってプロジェクトが一気に進展することもあります。しかし、不都合なことも起こります。例えば、ブレイクスルーによって大量のリファクタリングが必要になることがあります。これには時間や工数など、決して豊富にあるわけではないリソースを多く費やさなければなりません。また大量にリファクタリングすると、アプリケーションの振る舞いが変ってしまうかもしれません。

ブレイクスルーにたどり着くには、潜在している概念を明確にする必要があります。ドメインの専門家と話しているときには多くの概念や知識を交換します。ユビキタス言語に取り込まれている概念もありますが、はじめのうちは見つかっていない概念もあります。そのような概念が潜在している概念であり、既にモデルになっている概念を説明するために使われます。設計を洗練していると、このような潜在している概念が注意を惹きます。そのうちのいくつかは、設計のなかで重要な役割を果たす概念であることがわかります。このとき見つかった概念は明確にしなければなりません。クラスを作成して、関係を結びます。このように潜在している概念を明確にするときが、ブレイクスルーを生み出すチャンスかもしれません。

潜在している概念をそのままにしてはいけません。ドメインの概念であるなら、モデルや設計に表現されるべきです。では、潜在している概念はどうすれば見つけることができるのでしょうか。第一の方法は言葉に耳を傾けることです。モデリングや設計のときに使われる言葉にはドメインの情報がたくさん含

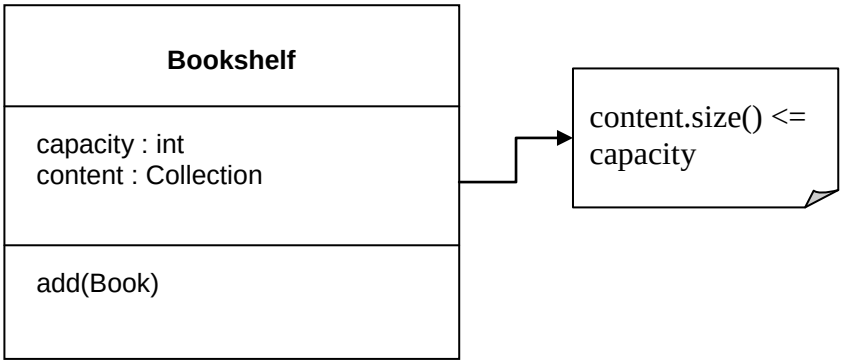
まれています。始めは多くの情報が含まれていないかもしれませんし、含まれている情報も正しくないかもしれません。十分に理解されていない概念もあるでしょうし、完全に誤解されている概念すらあるかもしれません。これは新しいドメインを理解するときには、必ず起こることです。しかし、ユビキタス言語を構築していくと、主要な概念はユビキタス言語へ取り込まれます。そして、潜在している概念を探し始めるのはこのときです。

設計のある部分があまり明確でないことがあります。そこには多数の関係が結ばれていて、処理の流れを追いかけるのが難しいかもしれませんし、また理解しにくい複雑な処理があるかもしれません。この部分の設計は未熟ですが、隠れた概念を探すのに都合のいい部分でもあります。きっと何かが足りないのです。この部分に主要な概念が欠けている場合、欠けている概念の役割はその他の概念が担っているはずです。そうなると、そのオブジェクトは肥大化します。本来はあるべきではない振る舞いが付け加えられるからです。そして、ドメインの明瞭さは損なわれてしまうでしょう。こんなふうにしないためには、足りない概念がないかどうか調べてみましょう。もし足りない概念が見つかったら、それを設計に反映しましょう。そしてその概念が単純に過不足なく欠落を埋めるように、設計をリファクタリングしましょう。

ドメインの知識を構築しているとき、矛盾に陥ることがあります。例えば、ドメインの専門家が言ったことが他の人が主張することと矛盾するように感じたり、ある要求が他の要求と矛盾したりすることがあります。しかし、そのような矛盾の中には、本当は矛盾しているわけではなく、同じものを違う見方で見ていたり、単純に説明不足な場合もあります。矛盾が見つかったら解消してみましょう。そうすると重要な概念が明らかになるかもしれません。何も見つからなくても、すべてを明確にしておくことは重要です。

モデルに取り入れるべき概念を掘り出すためのもうひとつの簡単な方法は、ドメインについての書籍を使うことです。様々な書籍が存在し、ありとあらゆるテーマを扱っています。それぞれのドメインについての知識をたくさん含んでいます。多くの場合、書籍は扱っているドメインモデルそのものを含んでいません。したがって、含んでいる情報を加工し、純化し、洗練する必要があります。それでも書籍から得られる情報には価値があります。書籍の情報はドメインの奥深くまで見通すのに役立ちます。

明確になっていると役に立つ概念は他にもあります。それは、制約、プロセスそして仕様です。制約は不変性を表すための単純な方法です。この方法を使えば、オブジェクトにどんなことが起こっても、不変性が考慮されます。この方法は制約の中に不変性を表すロジックを記述するだけで実現できます。次の図は簡単な例です。ただし、この図は概念を説明することだけを目的としています。類似のケースで制約を使うための指針ではありません。



本は本棚へ入れられますが、本棚の容量以上は入れられません。この制約は本棚クラスの振る舞いと見なせます。下記のようなJavaのコードで表現できるでしょう。

```
public class Bookshelf {
      private int capacity = 20;
      private Collection content;
      public void add(Book book) {
            if(content.size() + 1 <= capacity) {
                  content.add(book);
            } else {
            throw new IllegalOperationException(
            "The bookshelf has reached its limit.");
            }
      }
}
```

さらにコードをリファクタリングして制約を別のメソッドに抜き出します。

```
public class Bookshelf {
      private int capacity = 20;
      private Collection content;
      public void add(Book book) {
            if(isSpaceAvailable()) {
                  content.add(book);
```

```
            } else {
            throw new IllegalOperationException(
                  "The bookshelf has reached its limit.");
            }
      }
      private boolean isSpaceAvailable() {
            return content.size() < capacity;
      }
}
```

別のメソッドに抜き出すことで、制約をより明確に表現できます。Add()メソッドが制約の対象である、と簡単に読むことができますし、誰もが気づくでしょう。また制約が複雑になったとしても、制約を表すメソッドを抜き出してあれば、新たなロジックを簡単に追加することができます。

多くの場合、プロセスは手続きによって表現されます。しかし、私たちは手続き型の手法は使いません。オブジェクト指向言語を使っているからです。したがって、プロセスを表現するためには、プロセスのためのオブジェクトを選び、そこに振る舞いを追加しなければなりません。プロセスを実装するための最も良い方法はサービスを使うことです。プロセスを実行する方法がいくつかある場合は、そのアルゴリズムをカプセル化してストラテジパターンを使います。すべてのプロセスを明確にする必要はありません。しかし、プロセスがユビキタス言語含まれているのがはっきりしているなら、明確に実装する必要があります。

概念を明確にするための方法としてここで最後に紹介するのは仕様です。簡単に言えば、仕様とはオブジェクトが条件を満たしているかどうか検証する時に使います。

ドメインレイヤはエンティティやバリューオブジェクトに適用するためのビジネスルールを含んでいます。多くの場合、これらのルールはそのルールが適用されるオブジェクトに組み込まれます。”はい”か”いいえ”で答えられる一連の質問がルールを形成する場合もあるでしょう。このようなルールはBoolen型の値をとる一連の操作で表すことができ、最終的な結果もBoolen型の値になります。例えば、あるCustomerオブジェクトに一定の信用があるかどうかの検証が考えられます。このルールはIsEligible()という名のメソッドで表すことができ、Customerオブジェクトに組み込めます。しかし、Customerオブジェクトのデータを厳密に検証しようとすれば、ルールはひとつの単純なメソッドには収まりません。顧客の信用証明の検証や、過去に債務を返済したかどうか、負債があるかどうか等を検証します。このようなビジネスルールは検証項目が多くて複雑です。このような規則が組み込まれたオブジェクトは肥大化して本来

の目的を果たせなくなります。こうなるとすべてのルールをアプリケーションレベルに移行したいと思うかもしれません。ルールがドメインレベルを超えて広がりそうだからです。ここでリファクタリングをします。

このようなルールは専用のひとつのオブジェクトにカプセル化します。上の例で言えば、ルールをCustomerオブジェクトの仕様オブジェクトにカプセル化します。このオブジェクトはドメインレイヤに含んでおけばいいでしょう。この新しいオブジェクトが含むのは、特定のCustomerオブジェクトに信用があるかどうかを検証するためのBoolen型の値を返す一連のメソッドです。各メソッドは個別の項目を検証し、すべてのメソッドの結果をあわせた値が元の問い合わせの結果になります。ビジネスルールをひとつの仕様オブジェクトにまとめないと、多くのオブジェクトにルールを表すコードが散らばって、一貫性がなくなってしまいます。

仕様はオブジェクトがある要件を満たしているか、または何かの処理をする準備ができているかを検証するために使います。また、コレクションから特定のオブジェクトを取得するのにも使えます。オブジェクトの初期値をあらわすこともできるでしょう。

多くの場合、ひとつの仕様は単純なルールが守られているかどうかを判断します。そして、複数の仕様がひとつにまとめられ、下記のように複雑なルールを表現します。

```
Customer customer = customerRepository.findCustomer(customerIdentiy);

…
Specification customerEligibleForRefund = new Specification(

      new CustomerPaidHisDebtsInThePast(),

      new CustomerHasNoOutstandingBalances());

if(customerEligibleForRefund.isSatisfiedBy(customer) {
      refundService.issueRefundTo(customer);

}
```

このように、個々の単純なルールを検証すれば、ルールを単純に記述できます。また、どういうルールを満たせば顧客が借金をできるのかが、コードを読むだけで理解できます。

5

モデルの完全性を維持する

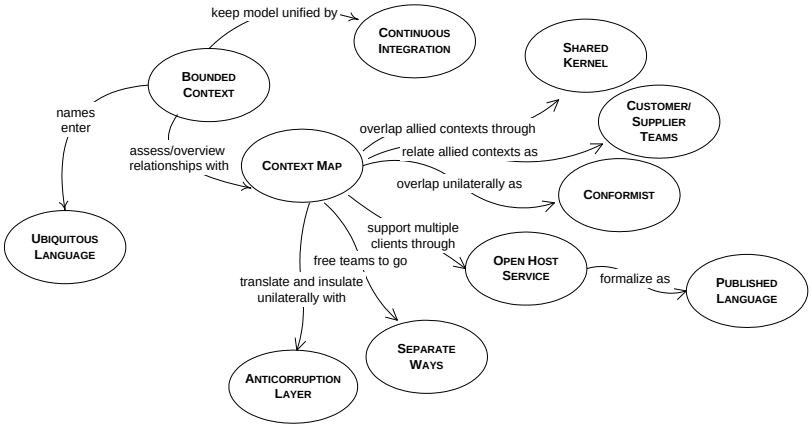
この章では複数のチームが協力して作業しなければならないような巨大なプロジェクトについて説明します。複数のチームが、それぞれ別の管理の下にプロジェクトでの開発作業を担う場合、様々な種類の困難に直面します。多くの場合、エンタープライズ分野ではプロジェクトは巨大になり、様々な技術やリソースが投入されます。このようなプロジェクトでもドメインモデルを基づいて設計をするべきです。そしてプロジェクトを確実に成功させるために適切な手段を採用する必要があります。

プロジェクトで複数のチームが作業する場合、各チームのコーディングは平行します。この場合、それぞれのチームにはモデルの特定の部分に割り当てられます。しかし、これらの部分は独立しているわけではなく、少なからず互いに繋がっています。最初はひとつの大きなモデルから作業が始まり、実装するためにそのモデルを分割して各チームに割り当てます。例えば、ひとつのチームがあるモジュールを作成し、他のチームが使えるようします。他のチームの開発者がそのモジュールを使ったところ、自分が作成したモジュールに機能的な要求を満たしていない部分を発見しました。彼はそのミスを修正してチェックインします。これで修正したモジュールをすべての人が使えるようになりました。しかし、彼はおそらく次のことに気づいていません。すなわち、この修正は本当は、モデルの変更であり、この変更がアプリケーションの機能を損なうことが十分にありうるということです。これは簡単に起こります。誰もモデル全体を完璧に把握するための時間を持っていないからです。皆自分の作業の背景は知っています。しかし、他の領域については細かい部分まで知りません。

作業を始めたときは優れたモデルだったのに、最終的には矛盾したモデルになってしまうのはよくあることです。モデルが満たさなければならない第一の条件は、矛盾がなく一定の用語が使われていて首尾一貫していることです。モデル内部の一貫性は「統一」と呼ばれます。エンタープライズ分野のプロジェクトでも、矛盾も用語の重複もなくドメイン全体を表せるかもしれません。しかし、この分野では統一されたひとつのモデルを作り上げるのは、難しく非現実的で、挑戦する価値すらない場合もあります。このようなプロジェクトでは多くのチームが力を合わせる必要があります。チームは十分に独立して開発できなければなりません。各チームが定期的に会って議論や設計をする時間がないからです。このような複数のチームの間を調整するのは気が滅入る仕事です。各チームはそれぞれ異なった部門に属し、別々に管理されているかもしれませ

ん。モデルの設計が部分的に独立して進められているとしたら、モデルの完全性が失われる可能性があります。モデルの完全性を保つためにひとつの巨大な統一モデルを保守しようと努力してもうまくいきません。この問題の解決方法は簡単には理解できません。なぜなら、この解決方法はこれまで学習してきたことと真っ向から対立するからです。つまり、粉々になってしまいそうなひとつの巨大なモデルを保守しようとするかわりに、意識的にいくつかのモデルに分割するのが解決方法になります。きちんとまとまっているモデルであれば、従わなければならない契約を守っている限り独立して設計できます。各モデルは明確な境界線を持たなければなりません。そして各モデルの間の関係は正確に定義しなければなりません。

これからモデルの完全性を維持するための方法を紹介します。下記の図はそれらの方法と各方法の関係をあらわします。



コンテキスト境界

各モデルはコンテキストを持ちます。ひとつのモデルを扱っている場合は、コンテキストは潜在的なものです。明確に定義する必要はありません。他のソフトウエアと相互に作用するようなアプリケーションを作成する場合、新しいアプリケーションは独自のモデルとコンテキストを持ち、古いモデルやコンテキストとは一線を画します。これは当たり前のことです。新旧のモデルやコンテキストを合成したり混ぜたりして複雑にはできません。しかし、エンタープライズ分野で巨大なアプリケーションを作成する場合、作成する各モデルのコンテキストを明示的に定義する必要があります。

どのような巨大プロジェクトでも複数のモデルが活動します。ところが、個々のモデルに基づくコードが結合されたとき、ソフトウエアにバグが埋め込まれ、信頼性が低くなり、理解するのが難しくなります。各チームのメンバのコミュニケーションも混乱するでしょう。それぞれのモデルがどんなコンテキストに適用されるのか、明らかでないからです。

ひとつの巨大なモデルを、複数の小さなモデルへと分割するための、決まったやり方はありません。関係のある要素や、ある概念を自然に構成する要素をモデルの中に置いてみましょう。**モデルは、ひとつのチームに割り当てることができるくらい小さくするべきです。**そうすればチーム内の協調性やコミュニケーションは、より柔軟で完全になり、同じモデルに取り組む開発者が楽になります。モデルのコンテキストとは、そのモデルに適用される一群の条件のことです。これらの条件は、モデル内で使われる用語が、特定の意味を確実に持つようにするために必要です。

モデルを分割するときに最も大事なのはモデルの範囲を確定すること、つまりコンテキストの境界を描くことです。さらに、モデルの統一を可能な限り乱さないようにしなければなりません。エンタープライズ分野のプロジェクト全体に渡るモデルを、純粋に保つのは難しいですが、特定の部分に限られているモデルであれば、きれいなままにしておくのは簡単です。モデルが適用されるコンテキストを明確にしましょう。そして、チームの体制や作成するアプリケーションの使い方、コードベースやデータベースのスキーマのような実際に構築するものを考慮に入れながら、境界線を決めていきます。境界内のモデルには全く矛盾がないようにする必要があります。このときは境界の外部の問題に気を使ったり、惑わされたりしないように注意しましょう。

コンテキスト境界はモジュールではありません。コンテキスト境界が提供するのは論理的な枠組みであり、その中でモデルは成長していきます。モジュールはモデルの要素を組織するために使われます。したがって、コンテキスト境界はモジュールを包み込む枠組みです。

複数のチームが同じモデルに取り組まなければならないとき、互いが気を悪くしないように注意しなければなりません。この場合はモデルの変更が既存の機能を損なうかもしれないことを常に意識している必要があります。しかし、たくさんのモデルに分割して扱えば、自身に割り当てられた部分で自由に作業できます。誰もがモデルの限界も知っていますから、境界内にとどまって作業ができます。しなければならないのはモデルを純粋で矛盾がなく統一されている状態に保つことだけです。そうすれば他のモデルに影響を与えずに簡単にリファクタリングできます。可能な限り汚くならないように設計を改善することができるのです。

複数のモデルに分割するには、ある程度の対価を支払う必要があります。境界線を定義し、異なるモデルの間に関係を結ばなくてはなりません。これには特別な作業と設計が必要です。おそらくモデル間のデータのやり取りのための、何らかの変換操作が出来上がるでしょう。変換操作がなければ、異なるモデルにオブジェクトを転送できないでしょうし、境界が存在しないかのように自由

に振る舞いを実行することもできません。しかし、これらは重大な問題ではありません。それに、これらの問題を解決して得られる利益は大きいです。

例えば、インターネットで商品を販売するためにe-コマースアプリケーションを作成したいとします。顧客はこのアプリケーションにメンバ登録することができ、アプリケーションはクレジットカードの番号を含む顧客の個人情報を収集します。これらのデータはリレーショナルデータベースに保存されます。顧客はアプリケーションにログインし、サイト内で商品を探して注文をします。アプリケーションは注文を受けたときはいつでもイベントを発生させる必要があるでしょう。注文を受けた商品を郵送しなければならないかもしれないからです。また、レポート出力画面を作ってレポートを作成したいです。そうすれば、販売中の商品の状態、顧客の好き嫌い等を観察できます。作業開始時は、e-コマースのドメイン全体を覆うモデルから出発します。大きなひとつのアプリケーションを作成するのが最終的な目的なのですから、ひとつの大きなモデルから始めようとするわけです。しかし、もっと慎重に作業を進めていくうちに、e-ショッピングアプリケーションとそのレポーティング機能は、本当は関係ないことがわかります。このことに気づけば、関心事を分離してそれぞれを異なる概念として扱います。異なる技術が必要になるかもしれません。本当に共通なのは、顧客と商品のデータが同じデータベースに保存してあり、e-ショッピングアプリケーションもレポーティング機能もこのデータベースにアクセスするということだけです。

この場合は、それぞれのドメインために個別のモデルを作成すればいいでしょう。e-コマースのドメインのためにひとつのモデルを作り、レポーティングのドメインのためにひとつのモデルを作ります。レポーティングアプリケーションが、e-コマースアプリケーションがデータベースに保存するはずの特定のデータを必要とするかもしれませんが、他の点では両者は独立して作業できます。

商品倉庫の職員に注文を受けたことを伝え、職員が商品を発送できるようにするためのシステムも必要です。また配達職員は顧客が購入した商品の品質や届け先、そして配達期日についての詳細な情報を調べられるアプリケーションを使うかもしれません。e-ショッピングのモデルはこのふたつの業務ドメインを含む必要はありません。そんなことをするよりも購入情報を含むバリューオブジェクトを非同期通信で倉庫へ送るほうが、e-ショッピングアプリケーションにとっては単純な仕組みになります。個別に開発できるふたつの明確に定義されたモデルがあれば、その間のインターフェイスが確実に動作するようにすればいいだけです。

継続的な統合

一度コンテキスト境界が定義できたら、むやみに変更してはなりません。複数の人々が同じコンテキストで作業をしていると、モデルの完全性が破られて断片化しやすいです。大きなチームほど大きな問題が起きますが、３人から４人のチームでも深刻な問題に直面します。システムを小さいコンテキストに落とし込もうとしても、結局は様々なレベルで統合性と一貫性が失われてしまいます。

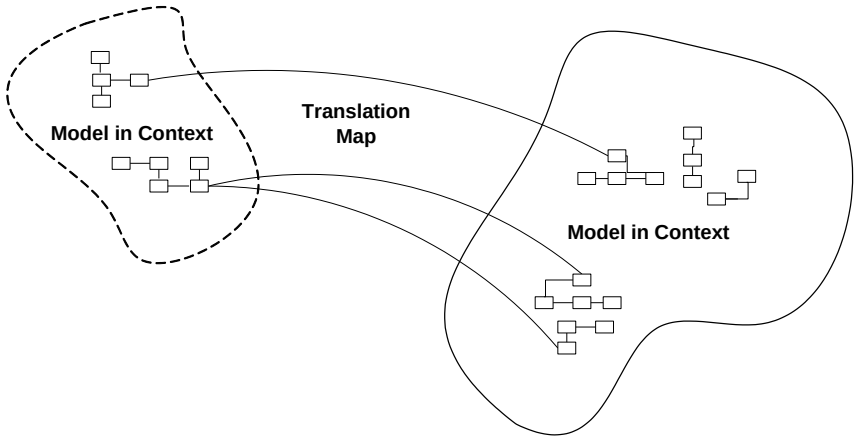
ひとつのチームがひとつのコンテキスト境界で作業している場合でさえ失敗する可能性があります。チーム内のメンバはコミュニケーションをとって、メンバ全員がモデルの各要素の役割を確実に理解しなければなりません。もしひとりでもオブジェクト間の関係を理解できていなかったら、メンバは意図しているのとは矛盾するようなコードの変更をするかもしれません。モデルの純度を保つことに１００パーセントの意識を集中していないと、このようなミスが発生しやすいです。メンバのひとりが知らないうちに、既存のコードと重複したコードを書いてしまうかもしれません。または、既存の機能を損なうのを恐れて、現在のコードを変更するかわりに重複したコードを書くこともあるでしょう。

モデルは最初から完璧に定義されていません。ドメインについての新しい洞察や、開発からのフィードバックを基にして継続的に成長していきます。したがって、新しい概念がモデルへ導入されることもあるでしょうし、新しい要素がコードへ追加されることもあるでしょう。このような新しい変更は、最終的にはひとつの統一されたモデルへと統合されて、その変更にしたがって実装されます。このため、コンテキスト境界を扱うときは継続的な統合が必要です。付け加えられた新しい要素をモデルの残りの部分と適合させて正確に実装するためです。そしてこの作業には、変更されたコードを統合する方法が不可欠です。素早くマージするほうが良いでしょう。単一の小さなチームであれば、毎日統合するのが良いでしょう。また適切なビルド手順も必要です。統合されたコードは自動的にビルドされる必要があります。そうすれば正しく統合されているかどうかのテストになります。もうひとつ必要なのは自動化されたテストを行うことです。テストツールがあり、テストコードが実装済みであれば、各ビルド毎に自動的にテストを実行してエラーを通知できます。このような手順が確立されていれば、エラーを修正するためにコードを変更するのは簡単です。エラーは簡単に検出できるからです。そして修正が完了したら、統合、ビルド、テストの各作業を再び実行します。

継続的な統合はモデル内の概念の統合に基づきます。そして、概念の統合に基づいていることは、テスト対象となる実装にまで影響します。モデルのどんな矛盾も実装時には見つけられます。継続的な統合はひとつのコンテキスト境界に適用されます。隣接するコンテキスト間の関係を扱うものではありません。

コンテキストマップ

エンタープライズ分野のアプリケーションは複数のモデルを持ちます。各モデルはそれぞれコンテキスト境界を持ちます。このコンテキストを基本にしてチームを組織するのが賢いやり方です。同じチームのメンバがコミュニケーションしやすくなりますし、モデルの作成、実装などの仕事がはかどります。各チームが割り当てられたモデルに取り組んでいる間、すべてのモデルを合わせた全体像がどんなものになるのか考えておくことは関係者全員にとって有益です。コンテキストマップは異なるコンテキスト境界と、それらの関係についての概略を示すための文書です。コンテキストマップは下記のような図になりますが、ほかの方法で記述してもよいでしょう。どのくらい詳細に記述するのかも様々です。大切なことはプロジェクトのメンバ全員がこの文書を共有し理解することです。



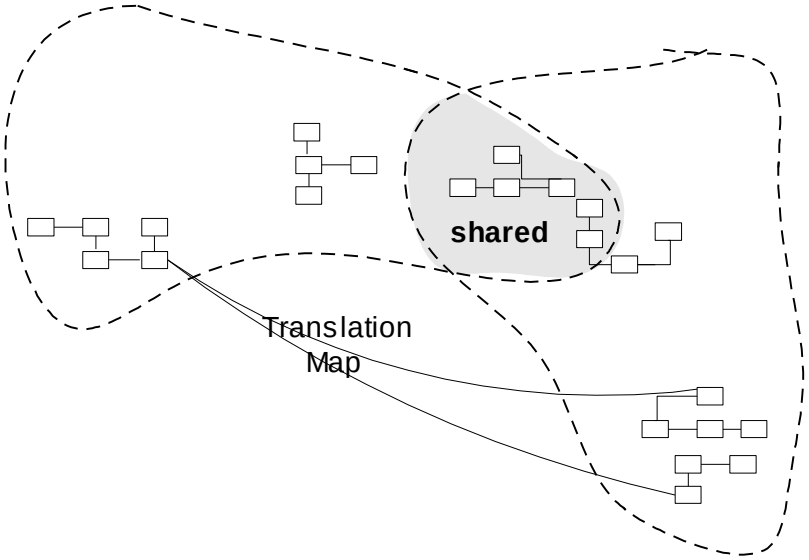
個々の、統一されたモデルがあるだけでは不十分で、それらを統合しなければなりません。各モデルの機能がシステム全体の一部分となるからです。最終的には部分が集められて完全なシステムとなり、きちんと動作しなければなりません。コンテキストがはっきりと定義されていない場合は、コンテキスト同士が重なり合ってしまうかもしれません。また、コンテキスト同士の関係がどうなっているのか大まかにわかっていないと、統合したときにきちんと動作しないかもしれません。

コンテキスト境界はそれぞれ名前を持っていますが、それらの名前はユビキタス言語の一部分です。名前を付けておけばシステム全体について話すときにと

ても役に立ちます。誰もがコンテキスト境界やコンテキストとコードの対応関係を知っているべきです。まずコンテキストを定義します。そして、モジュールを作成し、そして命名規則を使ってモジュールの名前を付けます。その名前で、どのモジュールがどのコンテキストに属するのがわかります。

ここからは、異なるコンテキスト同士の相互作用について説明します。これから説明するいくつかのパターンは、コンテキストマップを作成するのに役立ちます。これらのパターンを使ってコンテキストマップを作成すれば、各コンテキストの役割が明確になり、コンテキスト間の関係もはっきりします。共有カーネルとカスタマ−サプライヤはコンテキスト間に密接な関係がある場合に使われるパターンです。また別々の道はコンテキスト同士に高い独立性を持たせて個別に発展させていきたい場合に使われます。その他にふたつのパターンを紹介します。いずれもレガシシステムや外部システムと連携するために使うパターンです。それは公開ホストサービスと防腐レイヤと呼ばれるパターンです。

共有カーネル



機能的な統合が不十分だと、継続的な統合で発生するオーバヘッドは大きくなると思われます。とりわけ技術力のないチームや、継続的な統合の実行を維持する体制ができていないチームでは顕著かもしれません。またはひとつのチームが大きすぎて身動きが取れない場合でもあり得ることです。こういう場合はコンテキスト境界を定義し、複数のチームを作成して、作業を進めるのがいいでしょう。

密接な関係にある複数のアプリケーションを複数のチームで開発する場合、チーム同士が協調して作業を進めなければ、少しの間は互いに競い合って前進し

ますが、成果物を一緒にしても、うまく適合しないでしょう。結局、互換レイヤの作成や部分的な改良により多くの時間を費やしまい、継続的な統合は優先されなくなるでしょう。そして、そうこうしているうちに、同じ作業が重複してしまい、せっかく定義したユビキタス言語もその威力を発揮しなくなります。

こうならないためには、ドメインモデルのある部分をふたつのチームで共有します。もちろんその部分と一緒に、その部分に含まれるコードやデータベースの設計も共有します。この明示的に共有される部分は特別な部分であり、もうひとつのチームと相談しなければ変更を加えてはなりません。

この方法の場合、定期的にシステムを統合する必要がありますが、その頻度はチーム内の継続的な統合の頻度よりも少なくていいでしょう。チーム内での統合作業では、各チームでテストを実行します。

共有カーネルの目的は重複を少なくしながら、ふたつのコンテキストを分離しておくことです。共有カーネル部分を開発するには、細心の注意を払う必要があります。両方のチームが共有カーネルのコードを変更してもよいのですが、統合も一緒に行わなければなりません。各チームが共有カーネルのコードのコピーを使って作業しているのなら、できるだけ早くマージしなければなりません。最低でも週単位でマージしましょう。自動テストの準備もしておきます。共有カーネルに対するすべての変更がすぐにテストできるようにするためです。そして、共有カーネルに変更を加えたら他のチームに連絡し、各チームとも変更について理解するため、皆に変更内容を周知します。

カスタマ-サプライヤ

ふたつのサブシステムが特別な関係を持っている場合があります。ひとつのサブシステムがもうひとつに一方的に依存している場合です。ふたつのサブシステムはそれぞれ別のコンテキストに属しています。そして、一方のサブシステムの処理結果がもう一方への入力になります。このふたつのサブシステムの間には共有カーネルは存在しません。共有カーネルを持つことは概念的に正しくないでしょうし、さらにはコードを共有すること自体が技術的に不可能でしょう。このふたつのシステムはカスタマ-サプライヤ関係にあるのです。

前に取り上げた例に戻ってみましょう。先ほどはe－コマースアプリケーションのモデルを例に取り上げました。このアプリケーションにはレポーティング機能とメッセージ機能があります。これらの機能は、コンテキストごとに別々のモデルを作成したほうが良いことは既に述べました。ひとつのモデルで作業すると一定のボトルネックが発生してしまいます。ひとつのモデルだと開発は作業の重複が発生します。では、モデルを複数に分離する場合、Webショッピ

ングサブシステムとレポーティングサブシステムはどんな関係を持つべきでしょうか。共有カーネルは正しい選択肢ではなさそうです。サブシステムは異なるテクノロジーで実装されている場合が多いからです。例えば、ひとつはブラウザ上で動作し、もう一方はリッチなGUIアプリケーションとして動作します。レポートティングアプリケーションがWeb上で動作するとしても、Webショッピングサブシステムとはモデルの主要な概念が異なります。重複する部分はあるかもしれませんが、わざわざ共有カーネルを使うほどではないでしょう。したがって、他の関係を探します。別の面から考えてみましょう。Webショッピングサブシステムはレポーティングサブシステムにまったく依存しません。Webショッピングアプリケーションのユーザはwebからアクセスしてきて商品を閲覧し注文をする顧客です。すべての顧客、商品、注文のデータはデータベースにあります。これでわかりました。本当はWebショッピングアプリケーションは、それぞれのデータに何が起きているのかについて関心がありません。その一方でレポーティングアプリケーションは、Webショッピングアプリケーションによって保存されたデータに対してとても関心があり、そのデータを必要としています。またレポーティング機能を使うにはその他にも必要な情報があります。例えば、顧客は何らかの商品を買い物かごに入れ、支払いをする前にその商品をかごから出すことがあります。また、他のリンクをクリックして買い物を中止することもあるでしょう。このような情報は電子ショッピングアプリケーションには何の意味もありませんが、レポーティングアプリケーションにとっては大きな意味を持ち得ます。このように考えてみると、サプライヤになるサブシステムはカスタマとなるサブシステムの要件を満たす必要があります。この点でふたつのシステムが結びつきます。

もうひとつの必要なことは、データベースに関係します。正確に言えばデータベースのスキーマです。ふたつのアプリケーションは同じデータベースを使います。もしWebショッピングサブシステムだけがデータベースを使うのなら、Webショッピングサブシステムの都合に合わせて何度でもスキーマを変更できます。しかし、レポーティングサブシステムも同じデータベースを使う必要があります。したがってスキーマはある程度安定していなければなりません。データベースのスキーマが開発中に全く変更されない、というのは想像できないことです。Webショッピングサブシステムの都合で変更すれば、Webショッピングサブシステムには問題が起きませんが、レポーティングサブシステムには間違いなく問題が起きるでしょう。この問題を回避するには、ふたつのチームがコミュニケーションをとって作業を進める必要があります。データベースに関わる作業は協働して、いつスキーマの変更するのかを決定しなければならないかもしれません。しかし、一緒に作業することはレポーティングシステムの開発にとっては制約になります。レポーティングサブシステムを開発しているチームは、Webショッピングサブシステムの開発を待つよりも、素早く変更しながら開発を続けていきたいからです。もしWebショッピングサブシステムの開発チームが変更を拒否する権利を持っていれば、データベースの変更を制限するかもしれません。そうなるとレポーティングサブシステムの開発チームの

作業が進まなくなります。また、もしWebショッピングサブシステムの開発チームが単独で作業をしていると、レポーティングサブシステムの開発チームが約束を破って予期していなかった変更をするでしょう。ふたつのチームが一緒に働くのはチームが同じマネジメントの下にあるときにうまくいきます。作業を進めるための意思決定が簡単で、協調して作業できるからです。

このような状況に直面したら、行動しましょう。レポーティングサブシステム開発チームはカスタマの役割を、Webショッピングサブシステム開発チームはサプライヤの役割を、それぞれ引き受けます。ふたつのチームは定期的に、あるいは必要に応じて顔を合わせて、業者とその顧客がするような会話をします。つまり、カスタマチームは要求を提出してサプライヤチームがその要求に応じて計画をします。カスタマチームのすべての要求を聞き終わるまでに、サプライヤチームは要求を実現するための予定を立てます。本当に重要な要求があったら、他の要求を差し置いてすぐに実装するべきです。カスタマチームにはサプライヤチームの知識が成果物として必要です。サプライヤからカスタマへ一方的に情報を与えることになりますが、そういう場合もあるのです。

ふたつのサブシステムのインターフェイスは正確に定義しなければなりません。そのインターフェイスに適したテストを準備して、インターフェイスに関する要求を考えるときはいつもテストします。サプライヤチームは大胆に設計してもいいかもしれません。インターフェイスのためのテストがセーフネットになり、問題のあるときはいつでも警告してくれるからです。

ふたつのチームの間のカスタマ／サプライヤの関係を明確にしましょう。計画を決める会議ではカスタマチームは顧客としてサプライヤチームと向かい合うようにします。そして作業とスケジュールを理解するために、カスタマチームの要求を満たすための作業予定について話し合います。

自動化したテストも協力しながら開発しましょう。インターフェイスが期待通りに動作するかどうかを検証するテストです。これらのテストをサプライヤチームのテスト項目に加え、継続的な統合作業の一環としてテストを実行します。このように継続的にテストすると、カスタマチームの開発しているアプリケーションに影響を与えずに、自由にインターフェイスを変更できます。

順応者

カスタマ-サプライヤ関係は、両方のチームがこの関係に注意を払っているときにうまくいきます。カスタマチームはサプライヤチームに深く依存する一方で、サプライヤチームはカスタマチームに依存しません。この関係がうまくい

くように管理されていれば、サプライヤチームはこの関係を維持するのに必要な注意を怠ることなく、カスタマチームの要求に耳を傾けるでしょう。逆に、ふたつのチームの関係をはっきりと決めていない場合や、管理能力が不足していたり、管理自体がされていない場合、サプライヤチームは次第にモデルやその設計を相手にして作業することが多くなり、カスタマチームを助けることにあまり注意を払わなくなります。また、チームの作業には必ず期限があります。サプライヤチームの優秀なメンバが他のチームを助けようとしても、期限の重圧がものを言って、なかなか支援できずにカスタマチームが苦しむことになります。このような問題は各チームが別の会社に属している場合でも起こります。コミュニケーションをとるのが難しく、サプライヤチームの会社がカスタマ–サプライヤ関係を強固にするためにリソースを割こうとしないかもしれません。ときどき助けてくれるか、または全く協力してくれないかのどちらかでしょう。その結果、モデルと設計を基にしてカスタマチームだけで最善を尽くすしかありません。

ふたつのチームがカスタマ–サプライヤ関係を持ち、サプライヤチームがカスタマチームの必要とする情報を提供しない場合、カスタマチームは身動きがとれません。サプライヤチームは利他的精神に動かされて約束を守ろうとするかもしれませんが、十分な成果はあがりません。カスタマチームはその善意を信じて、決して手に入らないであろう成果物を基にして計画を立てます。その結果、カスタマチームの予定は、サプライヤチームからとりあえず与えられた不十分な情報で満足するしかないとカスタマチームが考えるようになるまで遅れてしまいます。これではカスタマチームに必要なインターフェイスは出来上がりそうにありません。

このような場合、カスタマチームの選択肢は限られています。最もはっきりとした選択肢は、サプライヤチームから離れて自分たちだけですべての作業を完結することです。この方法については後述する「別々の道」パターンの説明のなかで見ていきます。サプライヤとなるサブシステムから、有用な情報がときどき与えられるだけではこの問題を解決できません。分離したモデルを作成し、サプライヤのモデルから与えられる情報を考慮しないで設計をする方がより単純にこの問題に対処できます。しかし、これ以外のやり方もあります。

サプライヤのモデルにはカスタマチームにとって意味のある情報が含まれているので、関係を維持しなければならないときがあります。しかし、サプライヤチームがカスタマチームを手助けしないので、カスタマチームにはサプライヤチームのモデルの変更から自身のモデルを保護する方法が必要です。カスタマとサプライヤのふたつのコンテキストの間をつなげる変換レイヤを実装しなければならないでしょう。サプライヤチームのモデルが、考えなしに作られていて役に立たないかもしれません。カスタマのコンテキストでも使えるかもしれませんが、この場合は後述する防腐レイヤを使ってカスタマのコンテキストを保護したほうがいいでしょう。

もしカスタマチームがサプライヤチームのモデルを利用してうまく作業を進められるのなら、そのモデルに従います。カスタマチームがサプライヤチームのモデルに従うことで、カスタマチームのモデルは完全にサプライヤチームのモデルに適合するでしょう。これは共有カーネルにとても似ていますが、重要な違いがあります。カスタマチームはカーネル部分を変更できません。モデルの一部分として使ったり、提供された既存のコードを使えるだけです。この方法は様々な場面で使えます。多くの機能をもつコンポーネントとそのインターフェイスを誰かが提供してくれたとき、そのコンポーネントを含んだモデルを作成できます。コンポーネントが自分のモデルの一部であるかのように作成できるのです。コンポーネントのインターフェイスが小さければ、自身のモデルとコンポーネントのモデルの間にアダプタを作って、データのやり取りをするだけで済むかもしれません。そうすれば自身のモデルは独立させておけるので、より柔軟に開発できます。

防腐レイヤ

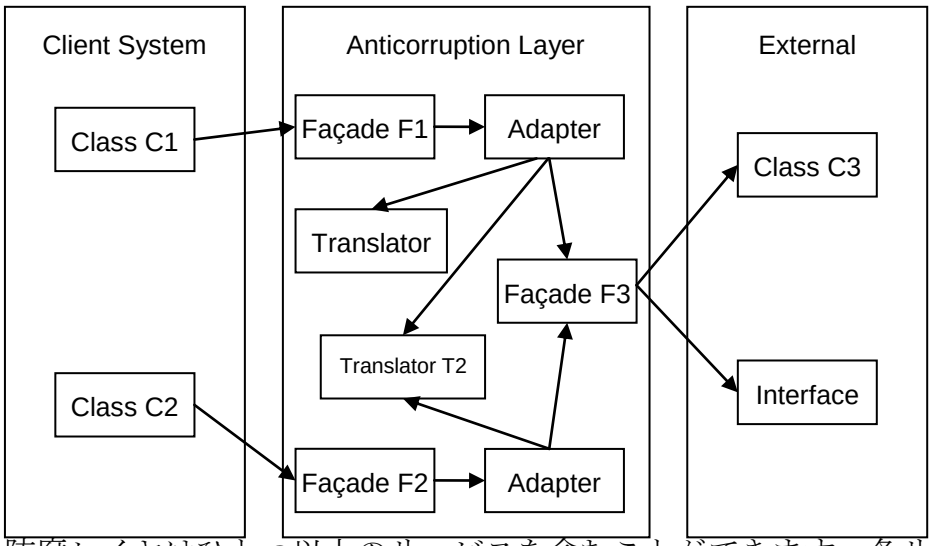
開発するアプリケーションが、レガシアプリケーションや外部アプリケーションと連携する場合があります。ドメインモデリングをする人にとっては、これもひとつの難問です。多くのレガシアプリケーションは、ドメインモデリングの技術を使って作成されていないので、そのモデルは混乱していて、複雑で、理解し実際に作業するのはとても大変です。うまく作られていた場合でも、レガシアプリケーションのモデルは使いやすくありません。私たちのモデルはレガシアプリケーションのモデルとは大きく異なっているだろうからです。にもかかわらず、レガシアプリケーションを使うためには、私たちのモデルとレガシアプリケーションのモデル同士を一定の水準で統合しなければなりません。

外部のシステムと連携するにはいくつかの方法があります。ひとつはネットワークを介して接続する方法です。この場合、ふたつのアプリケーションは同じプロトコルを使って連携する必要があります。そしてクライアントになるシステムは外部システムが提供するインターフェイスを信用しなければなりません。もうひとつの連携方法はデータベースを使います。外部システムがデータベースに保存されたデータを使い、クライアントとなるシステムも同じデータベースに接続します。このふたつの方法では数値や文字列のような基本的なデータがふたつのシステムの間を行き来するように処理します。これはとても単純な方法のように思えますが、基本的なデータはドメインモデルについての情報を含んでいません。データベースから取り出したデータをすべて基本データとして扱うことはできません。データは隠れた意味をたくさん持っているからです。リレーショナルデータベースでは、基本的なデータが他の基本的なデータと関

連することで関係の網目を作っています。データの意味はとても重要なので無視できません。扱っているデータの意味を理解していなければ、クライアントアプリケーションはデータベースにアクセスして、データを書き込めません。データベースに外部アプリケーションのモデルが反映されているのなら、それを自分のモデルに取り込みましょう。

また、何も注意していないと、外部のモデルによって自分たちのモデルに変更を強いられる危険があります。外部のモデルとの連携は無視できませんが、クライアントのモデルを外部のモデルからなるべく独立させなければなりません。この場合、クライアントのモデルと外部のモデルの間に防腐レイヤを作成します。クライアントのモデルから見ると、防腐レイヤは違和感なくクライアントのモデルの一部分になります。異質なものにはなりません。したがって、クライアントのモデルと親和的な概念や振る舞いを扱います。しかし、防腐レイヤが外部のモデルと対話をするときは、クライアントのモデルではなく外部のモデルの言語を使います。つまりこのレイヤはふたつのドメインと言語の間で双方向に働く変換機なのです。クライアントのモデルが外部のモデルから悪影響を受けることなく、きれいで矛盾のない状態を保つのがこの防腐レイヤの優れた機能です。

では、どのようにして防腐レイヤを実装すればいいでしょうか。防腐レイヤをクライアントのモデルが使うサービスとみなすのが優れた方法のひとつです。サービスとして実装すれば防腐レイヤはとても単純になります。サービスは外部システムを抽象化し、クライアント側の言葉で表現できるからです。このサービスは必要なデータ変換を行うので、クライアントのモデルは外部システムから分離したままにできます。サービスはファザードパターン（『デザインパターン』Gamma そのほか. 1995 をご覧ください）として実装するのが現実的です。その上、防腐レイヤはアダプタパターンにもよく似ています。アダプタパターンを使えば、クラスのインターフェイスをクライアント側が理解できるように変換できます。防腐レイヤを実装する場合は、アダプタはクラスを包む必要はありません。ふたつのシステムの間でデータを変換するのが仕事だからです。



防腐レイヤはひとつ以上のサービスを含むことができます。各サービスにはひとつのファザードがあり、各ファザードにアダプタをひとつ加えます。すべてのサービスをひとつのアダプタで構築するべきではありません。 アダプタに多くの機能が混ざってしまうからです。

また、もうひとつコンポーネントを付け加えなければなりません。アダプタは外部システムの振る舞いを包みますが、オブジェクトやデータも変換する必要があります。そこでトランスレータを使います。これはとても単純なオブジェクトで、ほとんど機能を持っていません。データ変換に最低限必要な機能を提供するだけです。もし、外部システムが複雑なインターフェイスを持っているなら、アダプタとその複雑なインターフェイスの間にファザードをもうひとつ加えたほうがいいかもしれません。そうすればアダプタと外部システムとのデータのやり取りは単純になりますし、分離も実現できます。

別々の道

ここまで各サブシステムをそのモデルや設計を変更することなく統合し、連携する方法を見つけようとしてきました。このような要件を満たすには努力と妥協が不可欠です。各サブシステムで作業をするチームは、サブシステム同士の関係が最適なものになるように、多くの時間を費やします。定期的にコードを結合する必要があるでしょう。また、正常に動作するのを確かめるために、何回もテストしなければならないでしょう。チームのひとりのメンバが、他のチームの要求を実装するためにだけ多大な時間を費やしてしまうこともあります。妥協も必要です。なるべく独立して開発を進めて、概念や発想を自由に取捨選択するのも大切ですが、自分が作成したモデルが他のシステムの枠組みの中に適切に当てはまるようにするのも大切です。このふたつを両立するためには、他のサブシステムと連携するためだけに、モデルを変更しなければならないか

もしれません。また、ふたつのサブシステムの間にデータを変換する特別なレイヤを導入する必要があるかもしれません。しかし、これ以外の方法でも実現できる場合もあります。まずは統合にどのくらい価値があるのかしっかりと評価して、本当に価値があるときだけ統合するようにします。評価の結果、統合には価値よりも問題のほうが大きいという結論になった場合、別々の道を進んだほうがいいでしょう。

「別々の道」パターンが扱うのは、エンタープライズ分野のアプリケーションがいくつかの小さなアプリケーションで作成されていて、モデルの観点から見ると、各アプリケーションはほとんど、あるいは全く共通部分をもたない場合です。この場合、ひとつのまとまった要求があり、ユーザからはひとつのアプリケーションに見えますが、モデリングや設計の観点から考えると、別々のモデルを使って独立して実装を進められます。まず要求が共通部分をもたない２、３のまとまりに分割できないかどうか確認します。分割できそうなら、そのまとまり毎にコンテキスト境界を作成して別々にモデリングします。この方法の利点は各モデルを実装するための技術を自由に選択できることです。各アプリケーションはGUIを共有してゆるやかに統合されます。このGUIはポータルサイトのようなもので、各アプリケーションに接続するためのリンクやボタンがあります。この緩やかな統合は、アプリケーションの背後のモデルを統合せずに、各アプリケーションを組織化することで実現します。

このパターンを使う前に確かめておかなければならないのは、作業を進めていってもアプリケーションを完全に統合する方針に戻ることはないということです。個別に作成されたモデルを統合するのはとても難しいです。また、共通部分がほとんどないので統合する価値もありません。

公開ホストサービス

ふたつのサブシステムを統合する場合、変換レイヤを挟むのが一般的です。このレイヤはクライアントになるサブシステムと統合したい外部サブシステムの間でバッファとして働きます。このレイヤは柔軟ではありません。サブシステム同士の関係の複雑さや外部サブシステムがどのように設計されたかに依存するからです。外部サブシステムがひとつのクライアントシステムだけではなく複数のシステムから使われる場合は、それらのひとつひとつに対応する変換レイヤを個別に作らなければなりません。しかし、それらのレイヤは同じ変換処理をおこない、似たようなコードを含むでしょう。

ひとつのサブシステムが多くのサブシステムと連携しなければならない場合、個別に変換レイヤを作成していてはチームの作業が行き詰まってしまうでしょ

う。メンテナンスしなければならないことが膨れ上がり、変更をするときには多くのことに気をつけなければなりません。

この問題を解決するには外部サブシステムをサービスのプロバイダと見なすことです。外部サブシステムを一群のサービスで包んでしまえれば、他のすべてのサブシステムはそのサービスを使うようになります。変換レイヤは必要なくなります。ただしこの方法にも難点があります。それは、各サブシステムはそれぞれ独自の方法で外部サブシステムと連携する必要があるかもしれない場合です。この場合、 サブシステムを均一なサービス群で包んでしまうと問題になるでしょう。

サービスに対するアクセスと同様の方式で、サブシステムにアクセスできるように連携の方式を定義します。さらにその方式を公開し、そのサブシステムと連携する必要のある他のシステムが使えるようにしましょう。そして、連携に関する新しい要求に対処するために方式を強化したり拡張したりします。ただし、ひとつのチームが特有の要求をしてきた場合は、拡張するべきではありません。特別な要求は限定的な変換処理で対処して、共通の方式が単純で矛盾のない状態を保つようにします。

蒸留

蒸留とは混ざり合っているものを分離することです。蒸留の目的は混合物から特定の物質を抽出することです。蒸留中に副産物が手に入ることもありますが、これも必要なものです。

大きなドメインを表現すると大きなモデルが出来上がります。磨き上げ、何度も抽象化した後ですらモデルは大きいままです。リファクタリングを繰り返しても小さくはなりません。このような状況になったら、蒸留するときなのかもしれません。蒸留の考え方はドメインの中核を表すコアドメインを定義するというものです。そして蒸留して得られた副産物はドメインの他の部分を構成する一般的なサブドメインとなります。

大きなシステムを設計するときには、手がかりとなる情報が大量にあります。それらはとても複雑ですが、設計を成功させるには必要不可欠なものです。この大量の情報に目を奪われて、ドメインモデルの本質や実際の業務でのシステムの価値が曖昧になったり無視されたりする可能性があります。

大きなモデルを扱っている場合は、一般的な概念から本質的な概念を区別するべきです。本書のはじめに航空監視システムの例を挙げました。そこでは運行計画には飛行機が従わなければならない航路が含まれていると説明しました。

この航路はシステム内で常に存在する概念です。しかし、実は航路は本質的な概念ではなくて一般的な概念なのです。航路は様々なドメインで使われる概念です。航路を表現するために作られるモデルは他にもあります。航空監視の本質は他の部分にあります。監視システムは飛行機が従うべき航路を知っていますが、一方で実際に飛行している飛行機を追尾するレーダー網からデータを受け取ります。このデータは飛行機が実際に飛行した航路を表示します。そしてこの航路は事前の設定とは異なっているのが普通でしょう。システムは現在の飛行状況を表す各種の数値や飛行機の特徴、天候などの情報から飛行機の軌道を算出します。この軌道は４次元の航路であり、飛行機がこれから飛んでいく航路を完全に表現しています。この航路が算出されるのは、次の２、３分の間かもしれませんし、数十分、数時間かかるかもしれません。この計算結果は業務上の意思決定を支援します。飛行機の航路を計算する真の目的は、その飛行機の航路が他の飛行機の航路と交わる可能性があるかどうかを確認することです。空港の周辺では、離発着の間に多くの飛行機が中空で旋回したり方向転換したりしています。もし一機が規定の航路から外れてしまったら、他の飛行機と衝突してしまう可能性が高くなります。航空監視システムとは飛行機の航路を計算し、衝突の可能性がある場合には警告を発するシステムです。航空管制官は素早い意思決定をして、飛行機に衝突を避けるように指示しなければなりません。飛行機が遠方にあれば航路を計算するのに時間がかかります。結果を受け取ってから指示を出すまでにはもっと時間がかかるでしょう。入手した情報から飛行航路を算出する部分が、この業務システムの心臓です。この部分をコアドメインにしなければなりません。航路についてのモデルは一般的なドメインのモデルにすぎません。

システムのコアドメインが何なのかはシステムをどのように見るかに依存します。単純な航路作成システムなら航路と航路に依存する概念がコアドメインでしょう。しかし航空監視システムの場合は航路はサブドメインに属します。このように、あるアプリケーションのコアドメインが別のアプリケーションのサブドメインになることもあります。重要なのはコアドメインを正確に識別し、コアドメインとモデルの他の部分の関係を決めることです。

モデルを煮詰めて、コアドメインを見つけ出し、その他のモデルやコードと簡単に区別がつくようにしておきます。最も価値がある特別な概念を強調しましょう。また、コアドメインはなるべく小さくしましょう。

コアドメインを担当するのはチームの中で最も優秀な人物です。または適切な人物を連れてくる必要があるかもしれません。担当者はコアドメインから詳細なモデルを作成し、柔軟な設計、つまりシステムの目的を十分に満たすような設計をします。ドメインの他の部分については、その部分がどのようにコアドメインを支えるのかによって作業の進め方を決めます。

コアドメインの実装に最も優秀な開発者を割り当てるのは重要なことです。一般的に、開発者はより技術的な分野を好む傾向があります。例えば、最も優れている最新の言語を習得したりビジネスロジックよりもインフラを実装することに魅力を感じます。彼らにとってドメインのビジネスロジックは、退屈で得るものがないと感じられるようです。飛行機の軌道について詳細に学ぶことが、なぜ重要なのでしょう。プロジェクトが終了したときにはすべての知識は過去のものとなり、その価値も少なくなってしまいます。しかし、ドメインのビジネスロジックはそのドメインの心臓です。この心臓部分の実装や設計に間違いがあれば、そのプロジェクトを途中で放棄しなければならないかもしれません。ビジネスロジックの核心部が正常に動作しなかったら、その他のすべての技術も無駄になってしまいます。

通常、コアドメインは１回の作業では完成しません。手を加えて改善し効果のあるリファクタリングによって、はじめてコアドメインは明確な姿を現します。したがって、コアドメインを設計の中心に据えて、その他の部分との境界を確定する必要があります。また、そのコアドメインとの関係を基にして、モデルの他の部分を見直します。おそらくモデルの他の部分もリファクタリングが必要でしょう。変更しなければならない機能もあるかもしれません。

モデルには、専門的な知識が取り入れられていないのに、複雑になっている部分もあります。異質なものが混入するとコアドメインは理解しにくくなります。例えば、誰もが知っている一般的な原則や、焦点はあたっていないけれど補助的な役割を持つ特別な知識が不自由にします。しかし、どんなに一般的であっても、コアドメイン以外の要素はシステムの機能にとっても、モデルを完全に表現するためにも不可欠です。

プロジェクトの動きを左右しないサブドメインのまとまりを特定しましょう。そして、それらのサブドメインの一般的なモデルを取り出して、別々のモジュールに配置します。自分が扱っている専門的な部分がその中に残らないように注意しましょう。

別のモジュールにすることができたら、それらの開発は後回しにしてコアドメインを優先します。サブドメインに、中核となる開発者を割り当てないようにしましょう（サブドメインの開発をしてもドメインの知識はほとんど得られないからです）。また、これらの一般的なサブドメインに適用できる既製のソリューションや公開されているモデルを使うことも検討します。

すべてのドメインは他のドメインで使われている概念を使います。通貨や交換レードのように金銭にまつわる概念は、異なるシステムにも含まれているでしょう。図表化も広く使われている概念です。この概念自体は複雑ですが、多くのアプリケーションで使われています。

一般的なサブドメインを実装するためのいくつかの方法を下記に示します。

1. **既製のソリューション。**利点は既にソリューションが完成していることです。しかし、そのソリューションについて習熟しなければ使いこなせないかもしれません。また、そのソリューションに依存してしまうこともあるでしょう。例えば、もしコードにバグがあったら修正されるのを待たなければなりません。またコンパイラやライブラリのバージョンも正しいものを選ぶ必要があります。後述する自前の実装と比べ、コアドメインと結合するのは簡単ではありません。
2. **アウトソーシング。**他のチームが設計と実装をおこないます。おそらく別の会社のチームです。こうするとコアドメインに注力することができ、サブドメインの作業を負担しなくて済みます。しかし、アウトソーシングしたコードと結合するのは面倒です。サブドメインと連携するためのインターフェイスを定義して相手チームに使ってもらわなければなりません。
3. **既存のモデル。**既存のモデルを使うのは手軽な方法です。分析パターンが紹介されている本があります。そのような本はサブドメインをモデリングするのに良い着想を与えてくれるでしょう。そのままでは適用できないかもしれませんが、多くの場合は少し手を加えれば使えます。
4. **自前の実装。**この方法には最善の水準で統合を実現できるという利点があります。しかし、それは同時にメンテナンス作業を含む余計な労力を割くことを意味します

6

今日における DDD の諸問題: Eric Evans へのインタビュー

InfoQ.com はドメイン駆動設計の提唱者である Eric Evans 氏に現在の DDD を取り巻く状況についてインタビューを行いました。

DDDが今までと同様、今日でも重要なのはなぜでしょうか。

基本的には、DDDはソフトウエア開発での原則です。それは、例えば、ソフトウエアのユーザの業務ドメインに含まれる深い問題に注力しなければならない、という原則であり、また、自分自身の最も優れた能力をドメインを理解することに費やし、そのドメインの専門家といっしょになんとかしてドメインを概念的な形式に落とし込み、強力で柔軟なソフトウエアを作成するのに利用できるようにしなければならない、という原則です。

この原則は決して廃れることがないでしょう。複雑に入り組んだドメインを扱うときにはいつでも適応できる原則だからです。

長い目で見れば、ソフトウエア開発は業務の心臓部の奥深くにある、より複雑な問題を扱う方向へ進んでいます。この動向はWebが一気に普及した数年の間、中断したように思えます。Web上で簡単にデータを処理できることが重要になり、優れたロジックを使って深いソリューションを実現することから関心が失われてしまいました。ソフトウエア開発がWebに対応するのは大変なことで、当初は単純な処理も簡単には実現できなかったため、Webに対応すること自体に開発の努力のすべてが費やされてしまいました。

しかし、いまやWebの基本的な使い方が多様になり、開発プロジェクトも再び、ビジネスロジックについて大きな要望を寄せるようになってきています。

そして最近になって、Web開発のプラットフォームはDDDを適用できるだけの十分な生産性を持つものへと成熟し始めました。例えばSOAを上手に利用すれば、簡単にドメインを分離できます。

また、この間にアジャイルプロセスが十分に普及しました。いまやプロジェクトのほとんどが反復開発や、ビジネスパートナーと一緒に働くことや、モジュールを継続的に統合すること、コミュニケーションしやすい環境で作業をすることに最低限の注意を払っています。

だからDDDは、今後さらに重要になるでしょう。そしてそのための基盤が作られていくように思えます。

技術プラットフォーム（Java .NET, Ruby 等）は進化し続けています。ドメイン駆動設計はこれらのプラットフォームにどのように適合するのでしょうか。

新しい技術や開発手法は、チームがドメインに集中するのを助けてくれるかどうかで評価するべきです。DDDは特定の技術プラットフォームに依存しませんが、ビジネスロジックを作成するのに優れた方法をもつプラットフォームもありますし、また障害物の少ないプラットフォームもあります。後者について言えば、1990年代後半のとくにひどかった状況が過ぎ去ったあとの、ここ数年の動向は期待できます。

ここ数年、Javaを利用することは標準的な選択肢でした。Javaは典型的なオブジェクト指向言語です。障害となりそうな構文上の不備についても、基本的な言語レベルでは大きな問題はありません。ガーベッジコレクション機能が実装されていて、実際にこれは本質的な機能です（それに比べてC＋＋は低いレベルのことに注意を払わなくてはなりません）。構文は多少汚いものの、plain old java objects（POJOs）は比較的読みやすく、Java5では可読性が向上しました。

しかし、J2EEが現れると、フレームワークの巨大なコードの山の下にJavaの基本的な表現方法が埋もれてしまいました。そして初期にできた慣例（例えばEJBHomeやすべての変数にget／setプレフィックス付きのアクセサをつける、というような）に従うことで恐ろしいオブジェクトが出来上がりました。開発ツールもとても使いにくく、チームの労力のすべてがツールを動かすことに費やされてしまいました。 そして一度汚いコードやxmlを生成してしまったら、オブジェクトを変更するのはとても難しく、その結果、だれも変更しようとはしませんでした。このプラットフォームでは効果的なドメインモデリングはほとんど不可能でしょう。

さらにこのフレームワーク上でhttpとhtml（ふたつともこういう目的で使えるためには設計されていません）を介して、ブラウザ上にUIを作成しなければなりません。しかも、とても原始的な第一世代のツールを使って、です。この頃、まともなUIを作成しメンテナンスしていくのはとても大変なことで、内部の機能の複雑な設計にまで注意する余裕はありませんでした。そして皮肉なことに、ちょうどその頃になってオブジェクト関連の技術が花開き、複雑なモデリングや設計手法に大きな影響を与えました。

.NETプラットフォームも似たような状況でした。Javaにくらべれば扱いやすい部分もありましたが、ひどい部分もありました。

こういう困難な状況がありましたが、ここ４年間で事態は変わりました。Javaについて言えばフレームワークを選択的に使う方法についての洗練された考え方がコミュニティに集まり、 様々な新しいフレームワークが生まれ（そのほとんどはオープンソース）継続的に改善されています。HibernateやSpringのようなフレームワークはJ2EEの特定の領域を扱いますが、よりライトな方法をとります。AJAXのような手法はより少ない労力でUIの諸問題を解決しようとする試みでしょう。価値あるJ2EEの要素を選び出し、新しい要素と組み合わせることについてはプロジェクトは賢くなっています。このような新しい動向の中でPOJOという言葉が生まれました。

こういった新しい動向の成果は段階的です。しかし、必要な技術的労力は徐々にですが明らかに減っています。そしてビジネスロジックをシステムの他の部分から分離して、個別に改良を加えることができるようになったので、ロジックをP0J0 に基づいて実装できます。自動的にドメイン駆動設計を適用できるようになったわけではありませんが、DDDを適用するのがより現実的になっています。

これはJavaの世界の話です。それからRubyのような新しい言語があります。Rubyはとても表現豊かな構文を持っています。この点でRubyはDDDと相性の良い言語でしょう（とはいっても私はDDDに基づいたRubyでの開発の実例を知りませんが）。また、Railsはとても面白いフレームワークのようです。WebのUIを1990年代の前半、まだWebがなかった頃のUI作成技術と同じくらい簡単に作ることができるようだからです。今はRailsの力はおびただしい数のWebアプリケーションを構築するのに使われています。それらのアプリケーションはドメインをしっかりと反映していませんが、かつては構築がとても難しかったものです。 UI実装の問題が少なくなったら、今度はドメインにもっと注意を払う番だと開発者が思ってくれると良いですね。こういう方向にRubyが使われだしたら、きっとDDDのためのすばらしいプラットフォームになると思いますよ。（それにはインフラ部分を若干補強する必要がありそうですが）

さらに最先端の試みとしてドメイン特化言語（DSL）の分野があります。私はずっと、この分野がDDDの次の大きな一歩になると思っています。今のところはニーズを本当に満たしてくれるツールはまだありません。でも、かつてないほど様々な試みがなされていますので期待しています。

現在は私の言える限りでは、ほとんどがJavaか.NETでDDDを実践しています。あとはSmalltalkも少し使われています。だからJavaの世界でDDDの影響が広がりつつあるのは好ましい動きです。

あなたの本が出版されてからDDDのコミュニティでどんなことが起こっていますか。

私がすばらしいと思ったのは、本に書いたDDDの原則を理解し、それを思いもしなかった方法で利用するようになったことです。例えばノルエーの国有企業のStatOil社での例が挙げられます。設計をおこなったアーキテクトがこの経験をレポートにまとめています。（http://domaindrivendesign.org/articles/で読むことができます。）

この中には、既製のソフトウエアを使うのかそれとも自前で作るのか評価するのにコンテキストマップを使っている例があります。

これとは全く違う例ですが、私たちの中には、様々なプロジェクトで必要になる基本的なドメインオブジェクトのライブラリをJavaで開発してなんらかの成果をあげようとしている人もいます。この試みは下記のURLから確認できます。

http://timeandmoney.domainlanguage.com

また、オブジェクトをJavaで実装しながらも、洗練されたドメイン特化言語の概念をどこまで押し進められるのかについての調査もおこなわれています。

このように、たくさんのことがおこなわれています。私に連絡をくれて自分が何をやっているのか教えてくれることをありがたく思っています。

DDD を学ぼうとしている人になにかアドバイスはありますか。

私の本を読んでください（笑）。プロジェクトでtimeandmoneyのライブラリを使ってみてください。私たちの当初の目的のひとつは、よいサンプルを提供して学習に役立ててもらうことでした。

気に留めておいてほしいのは、DDDの大部分はチームで実践する作業なのだということです。なので、誰かが伝道師の役割を果たす必要があるかもしれません。実際にDDDを実践しているプロジェクトを探したくなるのも、もっともなことです。

ドメインモデリングでは、いくつか注意しなければならないことがあります。

1) 開発に参加してください。モデリングする人にはコードが必要です。

2) 具体的なシナリオに取り組みましょう。抽象的な考えは具現化しなければなりません。

3) すべてに DDD を適用しようとしないでください。コンテキストマップを描いて、どこに DDD を適用して、どこに適用しないのかを決めましょう。そうすれば、適用外の部分を心配しなくてもよくなります。

4) たくさん実験をしてください。そして多くの失敗を想定してください。モデリングは創造的な作業です。

Eric Evans氏について

Eric Evans氏は『Domain-Driven Design: Tackling Complexity in Software』(Addison-Wesley 発行 2004）の著者です。

1990年代の前半から、氏は大規模な業務システムを開発する多くのプロジェクトに関わってきました。これらのプロジェクトで経験した様々な手法や成果の集大成が『Domain-Driven Design』です。この本には、成功するチームが複雑なソフトウエアの要求を考慮しながら整理し、プロジェクトをアジャイルに動かしながらシステムを大きく育てていく方法が詰まっています。

現在、氏は Domain Language 社を率いています。同社はドメイン駆動設計を取り入れるチームを指導し訓練するコンサルティング会社であり、チームの開発の価値を高めより生産的になる手助けをしています。