Chapter Three. Binding Model and Implementation

The first thing I saw as I walked through the door was a complete class diagram printed on large sheets of paper that covered a large wall. It was my first day on a project in which smart people had spent months carefully researching and developing a detailed model of the domain. The typical object in the model had intricate associations with three or four other objects, and this web of associations had few natural borders. In this respect, the analysts had been true to the nature of the domain.

As overwhelming as the wall-size diagram was, the model did capture some knowledge. After a moderate amount of study, I learned quite a bit (though that learning was hard to direct—much like randomly browsing the Web). I was more troubled to find that my study gave no insight into the application's code and design.

When the developers had begun implementing the application, they had quickly discovered that the tangle of associations, although navigable by a human analyst, didn't translate into storable, retrievable units that could be manipulated with transactional integrity. Mind you, this project was using an object database, so the developers didn't even have to face the challenges of mapping objects into relational tables. At a fundamental level, the model did not provide a guide to implementation.

Because the model was "correct," the result of extensive collaboration between technical analysts and business experts, the developers reached the conclusion that conceptually based objects could not be the foundation of their design. So they proceeded to develop an ad hoc design. Their design did use a few of the same class names and attributes for data storage, but it was not based on the existing, or any, model.

The project had a domain model, but what good is a model on paper unless it directly aids the development of running software?

A few years later, I saw the same end result come from a completely different process. This project was to replace an existing C++ application with a new design implemented in Java. The old application had been hacked together without any regard for object modeling. The design of the old application, if there was one, had accreted as one capability after another had been laid on top of the existing code, without any noticeable generalization or abstraction.

The eerie thing was that the end products of the two processes were very similar! Both had functionality, but were bloated, very hard to understand, and eventually unmaintainable. Though the implementations had, in places, a kind of directness, you couldn't gain much insight about the purpose of the system by reading the code. Neither process took any advantage of the object paradigm available in their development environment, except as fancy data structures.

Models come in many varieties and serve many roles, even those restricted to the context of a software development project. Domain-driven design calls for a model that doesn't just aid early analysis but is the very foundation of the design. This approach has some important implications for the code. What is less obvious is that domain-driven design requires a different approach to modeling. . . .

Model-Driven Design

***The astrolabe, used to compute star positions, is a mechanical implementation of a model of the sky*.**

Tightly relating the code to an underlying model gives the code meaning and makes the model relevant.

A Medieval Sky Computer

Ancient Greek astronomers devised the astrolabe, which was perfected by medieval Islamic scientists. A rotating web (called a *rete*) represented the positions of the fixed stars on the celestial sphere. Interchangeable plates engraved with a local spherical coordinate system represented the views from different latitudes. Rotating the rete against the plate enabled a calculation of celestial positions for any time and day of the year. Conversely, given a stellar or solar position, the time could be calculated. The astrolabe was a mechanical implementation of an object-oriented model of the sky.

Projects that have no domain model at all, but just write code to fulfill one function after another, gain few of the advantages of knowledge crunching and communication discussed in the previous two chapters. A complex domain will swamp them.

On the other hand, many complex projects do attempt some sort of domain model, but they don't maintain a tight connection between the model and the code. The model they develop, possibly useful as an exploratory tool at the outset, becomes increasingly irrelevant and even misleading.

All the care lavished on the model provides little reassurance that the design is correct, because the two are different.

This connection can break down in many ways, but the detachment is often a conscious choice. Many design methodologies advocate an *analysis model*, quite distinct from the design and usually developed by different people. It is called an analysis model because it is the product of analyzing the business domain to organize its concepts without any consideration of the part it will play in a software system. An analysis model is meant as a tool for understanding only; mixing in implementation concerns is thought to muddy the waters. Later, a design is created that may have only a loose correspondence to the analysis model. The analysis model is not created with design issues in mind, and therefore it is likely to be quite impractical for those needs.

Some knowledge crunching happens during such an analysis, but most of it is lost when coding begins, when the developers are forced to come up with new abstractions for the design. Then there is no guarantee that the insights gained by the analysts and embedded in the model will be retained or rediscovered. At this point, maintaining any mapping between the design and the loosely connected model is not cost-effective.

The pure analysis model even falls short of its primary goal of understanding the domain, because crucial discoveries always emerge during the design/implementation effort. Very specific, unanticipated problems always arise. An up-front model will go into depth about some irrelevant subjects, while it overlooks some important subjects. Other subjects will be represented in ways that are not useful to the application. The result is that pure analysis models get abandoned soon after coding starts, and most of the ground has to be covered again. But the second time around, if the developers perceive analysis to be a separate process, modeling happens in a less disciplined way. If the managers perceive analysis to be a separate process, the development team may not be given adequate access to domain experts.

Whatever the cause, software that lacks a concept at the foundation of its design is, at best, a mechanism that does useful things without explaining its actions.

If the design, or some central part of it, does not map to the domain model, that model is of little value, and the correctness of the software is suspect. At the same time, complex mappings between models and design functions are difficult to understand and, in practice, impossible to maintain as the design changes. A deadly divide opens between analysis and design so that insight gained in each of those activities does not feed into the other.

An analysis must capture fundamental concepts from the domain in a comprehensible, expressive way. The design has to specify a set of components that can be constructed with the programming tools in use on the project that will perform efficiently in the target deployment environment and will correctly solve the problems posed for the application.

MODEL-DRIVEN DESIGN discards the dichotomy of analysis model and design to search out a single model that serves both purposes. Setting aside purely technical issues, each object in the design plays a conceptual role described in the model. This requires us to be more demanding of the chosen model, since it must fulfill two quite different objectives.

There are always many ways of abstracting a domain, and there are always many designs that can solve an application problem. This is what makes it practical to bind the model and design. This binding mustn't come at the cost of a weakened analysis, fatally compromised by technical considerations. Nor can we accept clumsy designs, reflecting domain ideas but eschewing software design principles. This approach demands a model that works well as both analysis and design. When a model doesn't seem to be practical for implementation, we must search for a new one. When a model doesn't faithfully express the key concepts of the domain, we must search for a new one. The modeling and design process then becomes a single iterative loop.

The imperative to relate the domain model closely to the design adds one more criterion for choosing the more useful models out of the universe of possible models. It calls for hard thinking and usually takes multiple iterations and a lot of refactoring, but it makes the model *relevant*.

Therefore:

Design a portion of the software system to reflect the domain model in a very literal way, so that mapping is obvious. Revisit the model and modify it to be implemented more naturally in software, even as you seek to make it reflect deeper insight into the domain. Demand a single model that serves both purposes well, in addition to supporting a robust UBIQUITOUS LANGUAGE.

Draw from the model the terminology used in the design and the basic assignment of responsibilities. The code becomes an expression of the model, so a change to the code may be a change to the model. Its effect must ripple through the rest of the project's activities accordingly.

To tie the implementation slavishly to a model usually requires software development tools and languages that support a modeling paradigm, such as object-oriented programming.

Sometimes there will be different models for different subsystems (see Chapter 14), but only one model should apply to a particular part of the system, throughout all aspects of the development effort, from the code to requirements analysis.

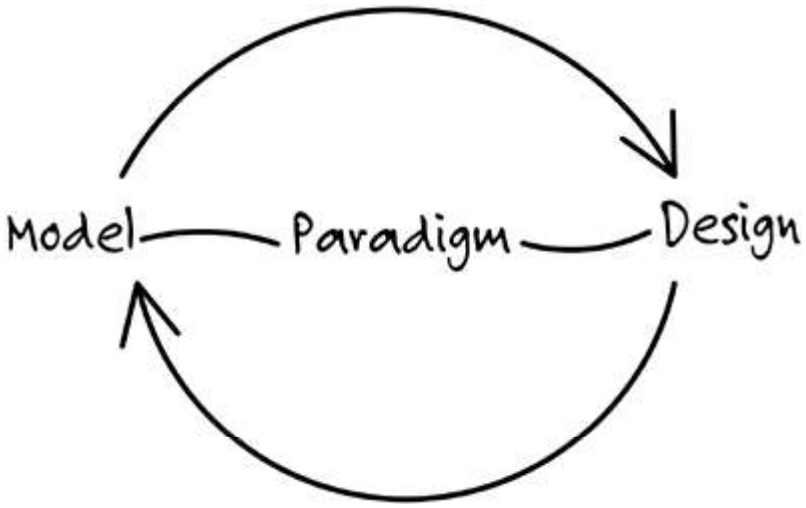
The single model reduces the chances of error, because the design is now a direct outgrowth of the carefully considered model. The design, and even the code itself, has the communicativeness of a model.

Developing a single model that captures the problem and provides a practical design is easier said than done. You can't just take any model and turn it into a workable design. The model has to be carefully crafted to make for a practical implementation. Design and implementation techniques have to be employed that allow code to express a model effectively (see Part II). Knowledge crunchers explore model options and refine them into practical software elements. Development becomes an iterative process of refining the model, the design, and the code as a single activity (see Part III).

Modeling Paradigms and Tool Support

To make a MODEL-DRIVEN DESIGN pay off, the correspondence must be literal, exact within bounds of human error. To make such a close correspondence of model and design possible, it is almost essential to work within a modeling paradigm supported by software tools that allow you to create direct analogs to the concepts in the model.

Figure 3.1.



Object-oriented programming is powerful because it is based on a modeling paradigm, and it provides implementations of the model constructs. As far as the programmer is concerned, objects really exist in memory, they have associations with other objects, they are organized into classes, and they provide behavior available by messaging. Although many developers benefit from just applying the technical capabilities of objects to organize program code, the real breakthrough of object design comes when the code expresses the concepts of a model. Java and many other tools allow the creation of objects and relationships directly analogous to conceptual object models.

Although it has never reached the mass usage that object-oriented languages have, the Prolog language is a natural fit for MODEL-DRIVEN DESIGN. In this case, the paradigm is logic, and the model is a set of logical rules and facts they operate on.

MODEL-DRIVEN DESIGN has limited applicability using languages such as C, because there is no modeling paradigm that corresponds to a purely *procedural* language. Those languages are procedural in the sense that the programmer tells the computer a series of steps to follow. Although the programmer may be thinking about the concepts of the domain, the program itself is a series of technical manipulations of data. The result may be useful, but the program doesn't capture much of the meaning. Procedural languages often support complex data types that begin to correspond to more natural conceptions of the domain, but these complex types are only organized data, and they don't capture the active aspects of the domain. The result is that software written in procedural languages has complicated functions linked together based on

anticipated paths of execution, rather than by conceptual connections in the domain model.

Before I ever heard of object-oriented programming, I wrote fortran programs to solve mathematical models, which is just the sort of domain in which fortran excels. Mathematical functions are the main conceptual component of such a model and can be cleanly expressed in fortran. Even so, there is no way to capture higher level meaning beyond the functions. Most non-mathematical domains don't lend themselves to MODEL-DRIVEN DESIGN in procedural languages because the domains are not conceptualized as math functions or as steps in a procedure.

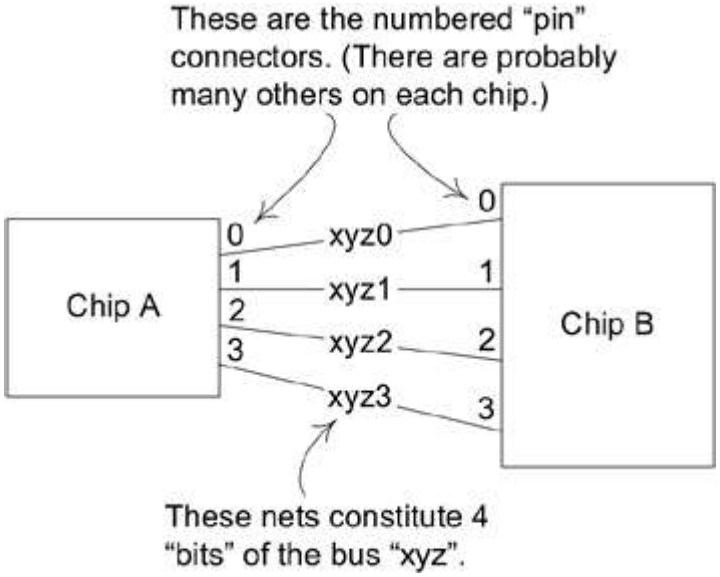
Object-oriented design, the paradigm that currently dominates the majority of ambitious projects, is the approach used primarily in this book.

Example
From Procedural to MODEL-DRIVEN

As discussed in Chapter 1, a printed circuit board (PCB) can be viewed as a collection of electrical conductors (called *nets*) connecting the pins of various components. There are often tens of thousands of nets. Special software, called a PCB layout tool, finds a physical arrangement for all the nets so that they don't cross or interfere with each other. It does this by optimizing their paths while satisfying an enormous number of constraints placed by the human designers that restrict the way they can be laid out. Although PCB layout tools are very sophisticated, they still have some shortcomings.

One problem is that each of these thousands of nets has its own set of layout rules. PCB engineers see many nets as belonging to natural groupings that should share the same rules. For example, some nets form *buses*.

Figure 3.2. An explanatory diagram of buses and nets



By lumping nets into a bus, perhaps 8 or 16 or 256 at a time, the engineer cuts the job down to a more manageable size, improving productivity and reducing errors. The trouble is, the layout tool has no such concept as a bus. Rules have to be assigned to tens of thousands of nets, one net at a time.

A Mechanistic Design

Desperate engineers worked around this limitation in the layout tool by writing scripts that parse the layout tool's data files and insert rules directly into the file, applying them to an entire bus at a time.

The layout tool stores each circuit connection in a *net list* file, which looks something like this:

```
Net Name    Component.Pin
--------    -------------
Xyz0        A.0, B.0
Xyz1        A.1, B.1
Xyz2        A.2, B.2
. . .
```

It stores the layout rules in a file format something like this:

```
Net Name    Rule Type        Parameters
--------    ---------        ----------
Xyz1        min_linewidth    5
Xyz1        max_delay        15
Xyz2        min_linewidth    5
Xyz2        max_delay        15
. . .
```

The engineers carefully use a naming convention for the nets so that an alphabetical sort of the data file will place the nets of a bus together in a sorted file. Then their script can parse the file and modify each net based on its bus. Actual code to parse, manipulate, and write the files is just too verbose and opaque to serve this example, so I'll just list the steps in the procedure.

```
1. Sort net list file by net name.
2. Read each line in file, seeking first one that starts with bus name pattern.
3. For each line with matching name, parse line to get net name.
4. Append net name with rule text to rules file.
5. Repeat from 3 until left of line no longer matches bus name.
```

So the input of a bus rule such as this:

```
Bus Name    Rule Type        Parameters
--------    ---------        ----------
Xyz         max_vias         3
```

would result in adding net rules to the file like these:

```
Net Name    Rule Type        Parameters
--------    ---------        ----------
. . .
Xyz0        max_vias         3
Xyz1        max_vias         3
Xyz2        max_vias         3
. . .
```

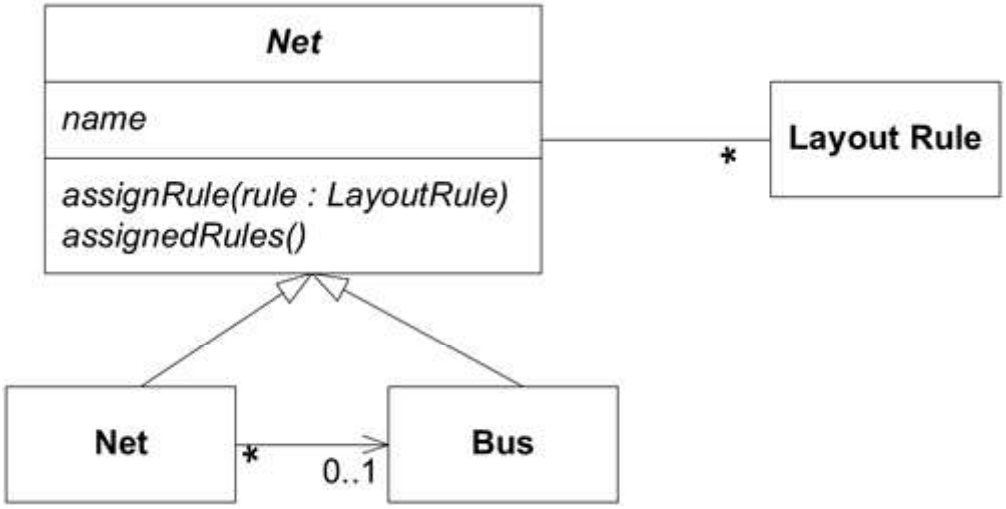
I imagine that the person who first wrote such a script had only this simple need, and if this were the only requirement, a script like this would make a lot of sense. But in practice, there are now dozens of scripts. They could, of course, be refactored to share sorting and string matching functions, and if the language supported function calls to encapsulate the details, the scripts could begin to read almost like the summary steps above. But still, they are just file manipulations. A different file format (and there are several) would require starting from scratch, even though the concept of grouping buses and applying rules to them is the same. If you wanted richer functionality or interactivity, you would have to pay for every inch.

What the script writers were trying to do was to supplement the tool's domain model with the concept of "bus." Their implementation infers the bus's existence through sorts and string matches, but it does not explicitly deal with the concept.

A Model-Driven Design

The preceding discussion has already described the concepts the domain experts use to think about their problems. Now we need to organize those concepts explicitly into a model we can base software on.

Figure 3.3. A class diagram oriented toward efficient assignment of layout rules



With these objects implemented in an object-oriented language, the core functionality becomes almost trivial.

The `assignRule()` method can be implemented on **Abstract Net.** The `assignedRules()` method on **Net** takes its own rules and its **Bus**'s rules.

```
abstract class AbstractNet {
    private Set rules;

    void assignRule(LayoutRule rule) {
        rules.add(rule);
    }

    Set assignedRules() {
```

```
        return rules;
    }
}

class Net extends AbstractNet {
    private Bus bus;

    Set assignedRules() {
        Set result = new HashSet();
        result.addAll(super.assignedRules());
        result.addAll(bus.assignedRules());
        return result;
    }
}
```

Of course, there would be a great deal of supporting code, but this covers the basic functionality of the script.

The application requires import/export logic, which we'll encapsulate into some simple services.

Service	Responsibility
Net List import	Reads Net List file, creates instance of Net for each entry
Net Rule export	Given a collection of Nets, writes all attached rules into the Rules File

We'll also need a few utilities:

Class	Responsibility
Net Repository	Provides access to Nets by name
Inferred Bus Factory	Given a collection of Nets, uses naming conventions to infer Buses, creates instances
Bus Repository	Provides access to Buses by name

Now, starting the application is a matter of initializing the repositories with imported data:

```
Collection nets = NetListImportService.read(aFile);
NetRepository.addAll(nets);
Collection buses = InferredBusFactory.groupIntoBuses(nets);
BusRepository.addAll(buses);
```

Each of the services and repositories can be unit-tested. Even more important, the core domain logic can be tested. Here is a unit test of the most central behavior (using the JUnit test framework):

```
public void testBusRuleAssignment() {
    Net a0 = new Net("a0");
    Net a1 = new Net("a1");
    Bus a = new Bus("a"); //Bus is not conceptually dependent
    a.addNet(a0);         //on name-based recognition, and so
    a.addNet(a1);         //its tests should not be either.
```

```
    NetRule minWidth4 = NetRule.create(MIN_WIDTH, 4);
    a.assignRule(minWidth4);

    assertTrue(a0.assignedRules().contains(minWidth4));
    assertEquals(minWidth4, a0.getRule(MIN_WIDTH));
    assertEquals(minWidth4, a1.getRule(MIN_WIDTH));
}
```

An interactive user interface could present a list of buses, allowing the user to assign rules to each, or it could read from a file of rules for backward compatibility. A façade makes access simple for either interface. Its implementation echoes the test:

```
public void assignBusRule(String busName, String ruleType,
      double parameter){
   Bus bus = BusRepository.getByName(busName);
   bus.assignRule(NetRule.create(ruleType, parameter));
}
```

Finishing:

```
NetRuleExport.write(aFileName, NetRepository.allNets());
```

(The service asks each **Net** for `assignedRules()`, and then writes them fully expanded.)

Of course, if there were only one operation (as in the example), the script-based approach might be just as practical. But in reality, there were 20 or more. The MODEL-DRIVEN DESIGN scales easily and can include constraints on combining rules and other enhancements.

The second design also accommodates testing. Its components have well-defined interfaces that can be unit-tested. The only way to test the script is to do an end-to-end file-in/file-out comparison.

Keep in mind that such a design does not emerge in a single step. It would take several iterations of refactoring and knowledge crunching to distill the important concepts of the domain into a simple, incisive model.

第二章　如果摩西是埃及人……

[16]

在本书第一章中，我力图用一个新论据来说明犹太民族的解放者，法律奠基人摩西并非犹太人，而是埃及人。他的名字来源于埃及词汇，这一点虽然没有得到应有的重视，但早已被人注意到了。此外，我认为对婴儿摩西被遗弃的神话的分析必然得出下述结论，即他是一个埃及人，但有个民族需要把他说成是犹太人。在我的文章结尾处，我曾说过，由摩西是埃及人的设想可以作出重要而意义深远的结论；但是我没有打算公开提出这类论点，因为它们建立在心理学或然性基础上，缺乏客[17]观的证据。这样辨析出来的可能性意义越重大，我在把它暴露出去面对外界批评时就应该越谨慎，因为它就像一座放在泥土上的钢铸纪念碑，没有任何安全基础，不论这些可能性多么具有诱惑力，我们都不可能保证它们没有漏洞；即使一个问题的所有部分都吻合得丝丝入扣，我们也必须记住，可能的不一定就是真理，真理也并不一定总是可能的。并且，归根结底，被人认为与学究和捧着犹太法典不放的人为伍，满足于卖弄智巧，而不管自己的结论离真理有多远，终究不是令人向往的事情。

虽然这些疑虑至今仍然像过去一样使人感到沉重，我的各种目的也互相冲突，我还是下定决心在此基础上继续补充我的第一篇论文。但是我要再次重申，它仅仅是全文的一个组成部

分，而且不是最重要的部分。

一

按照前文所述，如果摩西是一个埃及人，由此设想而来的第一个收获就是一个难于解开的新迷。当某个部落[①]的人民准备进行一项伟大的事业时，可以预料，他们当中的一员将自居为领袖，或者被推选来担任这个角色。但是，到底是什么能够[18]引诱一个身世显赫的埃及人——也许是一个王子、祭司或高级官员——自愿去充当一群更不开化的移民的首脑，并且同他们一起离开自己的国家呢？这一点不容易推测。众所周知，埃及人对外国人十分蔑视，这使得那样一个行动更不可能。确实，我倾向于认为，这就是使那些甚至承认摩西这个名字是埃及人的名字，并且把所存的埃及智慧都归之于摩西一人的历史学家们不愿意考虑摩西明显可能是埃及人的原因。

第二个困难接踵而至。我们必须记住，摩西不仅是定居埃及的犹太人的唯一政治领袖：他还为他们制定法律，教育他们并强迫他们接受了一种新宗教，这种宗教至今仍然称为摩西律法。但是，单独一个人能够那样轻而易举地创立一种新宗教吗？当信仰某种宗教的人想影响信仰另一种宗教的人时，最自然不过的作法难道不是使他改信自己的宗教吗？埃及的犹太人当然不会没有某种形式的宗教，如果为他们创立新宗教的摩西是埃及人，那么，猜测这种新宗教是埃及的宗教是立得住脚的。

这种可能性面临着一道障碍：以摩西名字命名的犹太教与埃及宗教之间的尖锐对立。犹太教是一种庞大而严格的一神教，它只有一个神，一个唯一的、全能的、无法接近的神。没有人能够看到他的容貌；人们不能塑造他的肖像，甚至也不能提到

他的名字。另一方面，在埃及宗教中，不同地位和起源的神却多得叫人数不清。其中有些是大自然威力的化身，如天和地、[19]太阳和月亮等；有些是抽象概念，如马特(Maat，指正义、真理)，或者是某种怪异的创造物，如形状似侏儒般的伯斯(Bes)；然而，其中大多数是本地神，是在埃及土地划分成许多省时起源的。他们具有动物的形状，好像还没有摆脱他们的古老图腾动物的出身似的。这些神没有明确的区别，只是由赋予其中某些神的特殊作用而得以模糊地区分。人们毫无顾忌地把他们一视同仁，对其中的每一个神都唱着同样的颂歌，以致于我们完全无法指望能区分他们。这些神的名字互相掺杂，甚至于其中一个神的名字成为另一个名字的组成部分。比如，在"新帝国"的最好时期中，底比斯城② 的主神名叫阿蒙－赖(Amon－Re)，这个名字的第一部分阿蒙(Amon)是个长着公羊头的城市守护神，而赖(Re)却是古老北方的长着鹰头的太阳神的名字。巫术和仪式，种种驱邪符和俗套在祭祀这些神的活动中占着重要的地位，就像它们在埃及人的日常生活中，占据着重要的地位一样。

某些这类区别大概是由于严厉的一神教和松散的多神教之间的对立而产生的。另一些明显的是不同智力水平的结果：多神教非常接近于原始宗教，而一神教却已达到使人惊奇的抽象概念的高度。也许就是这两种特征不时使人们产生这种印象，即犹太教和埃及宗教之间的对立是被有意促成的。例如，受犹太[20]教激烈谴责的各种魔法或巫术在埃及教里十分盛行；当埃及人兴致勃勃，不知满足地用泥土、石头和金属塑造他们的众多的神像时，犹太人却毫不含糊地禁止制作任何活着的人或虚幻的神的肖像。

在这两种宗教之间还有一种区别，我不打算加以解释。在

古代没有任何其它的民族像埃及人那样费尽心力去抵制死亡，去为死后的生活作精心的安排；与此一致，另一世界的统治者，死神奥西里斯(Osiris)③，在所有埃及神中是最著名而且最不容争议的。另一方面，早期犹太教却完全放弃了不朽的概念，它从来没有提到过人死后仍然存在的可能性，这一点是最值得注意的，因为后来的经验表明，相信人死后能得到超渡并能够与一神教宗教和谐共处。

我曾经希望，摩西是埃及人这种设想在许多不同的方面将表明是富于启发性和刺激性的。但是，我们由此而来的第一个推断——摩西给予犹太人的新宗教是一种埃及宗教——却建立在这两种宗教的不同之点上，甚至建立在两者的鲜明对立上。

二

[21] 埃及宗教史上的一个奇怪的事实使我们产生了另一个观点，这个事实比较晚期才得到鉴定和承认。摩西给犹太民族带去的宗教虽然不是当时埃及奉行的那一种宗教，却仍然可能是他自己的宗教，即一种埃及宗教。

在光荣的第十八朝时期，埃及第一次成为了一个世界强国，有个年轻的法老在公元前1375年前后登上了王位，他最初仿效他的父亲将自己称为阿蒙霍特普④(意为阿蒙之子)，但是后来改变了名字——而且不仅仅改变了名字。这位国王强迫他的臣民接受了一种严格的一神教，这种新宗教同他们的传统和习俗完全相反。就我们所知，这种新宗教在世界历史上还是首次尝试；在那之前以及之后很长时期，人们还没有产生宗教偏执的问题，然而当时却无可避免地产生了只有一个神的信仰。但是，阿蒙霍特普的统治只持续了17年；公元前1358年他去世后不久，

那种新宗教即被废除，这位持异端邪说的国王也被人们逐出了记忆。从他新建的以他的神命名的首都的废墟中，从那里的石头坟墓的雕饰上，我们得以稍许了解一点有关他的情况，关于这位杰出的、绝无仅有的人的一切有关情况确实是值得极大注意的⑤。

一切新事物都必然有其历史渊源，埃及一神教宗教的起 [22]
源可以比较有把握地追溯到一个相当久远的时期⑥。在古老北方（赫利阿波利斯Heliopolis）⑦的太阳神庙的祭司学校里，在那之前一度已经出现了一种倾向，发展出了关于一个宇宙神的概念，并且强调他的伦理学方面的意义。真理、秩序和正义女神马特是太阳神赖的女儿。还在阿蒙霍特普三世（改革者阿蒙霍特普四世的父亲和前任）当权时，对太阳神的崇拜就在日益上升，这种崇拜可能与对当时已经太过显著的底比斯的阿蒙崇拜互不相容。太阳神的一个古老的名字，阿顿或阿吞（Aton or Atum）又被抬出，在这个阿顿宗教中，年轻的国王阿蒙霍特普四世找到了一个他无需创立的运动，他只需参与其中就足够了。

大约在那段时期，埃及的政治环境开始对宗教产生持续的影响。由于伟大的征服者图特摩斯三世⑧（Thothmes）的武功，埃及已经成为一个世界强国，南方的努比亚（nubia），北方的巴勒斯坦，叙利亚和美索不达米亚的一部分已被纳入了帝国的版图。这种帝国主义在宗教中表现为包容万物性和一神教。既然法老操心的范围现在扩及到埃及之外的努比亚和叙利亚，神本身也就必须放弃它的国界，埃及的新神就必须像法老一样，在埃及的新世界中树立起唯一的和无限的权威。此外，随着疆 [23]
界的扩大，埃及自然变得容易接受外来影响；阿蒙霍特普四世的一些妻子是亚洲的公主⑨，她们甚至可能是一神教宗教从叙

利亚渗入埃及的直接鼓励因素。

阿蒙霍特普从来没有否认过他对古老北方的太阳神的崇拜。从保存下来的当时的石墓上雕刻的可能系他所写的两首对阿顿的颂歌来看，他称赞太阳是埃及内、外所有生物的创造者和保护者，其狂热程度，只有在许多世纪之后犹太人颂扬上帝耶和华（Jahve）的赞美诗里出现过。但是他没有停留在那种对太阳光作用的惊人的科学预见上，毫无疑问，他走得更远：他不是把太阳作为一个物质对象来崇拜，而是作为一个神的象征来崇拜，这个神的力量由他的光线得到了证实⑩。

但是，如果仅仅把这位国王看着在他之前就已存在的阿顿宗教的坚持者和保护人，我们确实就贬低了他的作用。他的活动更为充满活力。他在阿顿教中加入了某些新东西，把一神教教义变成了全宇宙一个神的教义：即排他性的教义。在他所写的一首颂歌中，他用了许多词语来赞颂阿顿神："啊，您唯一的神，除您之外没有其他的神⑪。"我们必须记住，要赞扬这种新教义，仅仅了解支持它的方面是不够的；了解它的反对方面，即它所弃绝的内容，对我们同样显得重要。设想这种新宗教会象雅典娜跳出宙斯的额头⑫那样一蹴而就，也会是一个错误。实际上每件事情都显示出，在阿蒙霍特普统治期间，阿顿教被强化了，以便变得更纯洁、更持久、更严厉、更狂热。这种趋势可能是由于那些起而抗拒国教改革的阿蒙教祭司们的强烈反对引 [24]
起的。在阿蒙霍特普统治的第六年，这种敌意急剧增长，以致于阿蒙霍特普干脆更改了自己的名字，因为在新教中被禁止的神阿蒙的名字是他的名字的一部分。于是他把自己改名为埃赫那顿（Ikhnaton）⑬。他不仅从自己的名字中清除了被仇视的神阿蒙的痕迹，而且清除了所有有关的雕刻碑铭，即使是以他

父亲阿蒙霍特普三世名义建立的也不例外。他更名为埃赫那顿后不久离开了仍在阿蒙神影响之下的底比斯，在上尼罗河建立新都，定名为阿克塔顿(Akhetaton，意为阿顿之光)。它的遗址现名为泰尔·埃尔·阿马尔那⑭。

[25]

阿蒙霍特普四世掀起的宗教迫害首先指向阿蒙神，但并非仅此而已。帝国内的每一处阿蒙神庙均被关闭，仪式被禁止，神庙的财产被侵占。这位国王越来越狂热，以致于下令清查古老纪念物上的题铭，凡是使用复数的"神"这个词的地方均予抹除⑮。毫不足怪，这些命令在被压迫的祭司和心怀不满的人民中引起了狂热的报复反应，阿蒙霍特普四世死后，这种反应才得以自由发泄出来。阿顿神教还没有引起人民的兴趣；它可能只局限在埃赫那顿周围的人当中。他的结局显得极为神秘，我们约略知道他家庭中几个短命的继承人情况，他的女婿图坦卡顿⑯(Tutankhaton)被迫迁都回底比斯，并且用阿蒙神的名字替换了他名字中阿顿神名字部分，改名为图坦卡蒙。其后经过了一段混乱时期，至到公元前1350年，他的将军荷伦希布(Haremhab)才成功地恢复了秩序。光荣的第十八朝寿终正寝了；同时，它的征服地努比亚和亚洲诸国也丧失了。在这段缺乏正常统治者的时期中，埃及的古老宗教得以复苏，阿顿神教走到了末路，埃赫那顿的首都被劫掠、捣毁，他也变得声名狼藉。

[26]

如果我们现在注意阿顿神教的几个排它性特征，将会达到某种目的。首先，所有的神话、巫术和魔法都被排除在它之外了⑰。

其次，代表太阳神的方式不再是早期的小金字塔或鹰，而是一个圆盘，无论在阿马尔那时期人们多么崇尚艺术，我们却没有发现任何象征太阳神阿顿的东西，而且我们可以有把握地

认为将来也不会发现任何这类物件⑱。

最后，关于死神奥西里斯和冥界，人们再也没有任何议论。从坟墓里发现的颂歌和题铭上也看不出当时埃及首都的任何事情。阿顿神教与埃及流行宗教的对立在此表现得再生动不过了⑲。

三

[27]

我现在冒昧地作出以下结论：如果摩西是一个埃及人，如果他把自己的宗教传给了犹太人，那么那种宗教是埃赫那顿的阿顿神教。

在此之前我比较过犹太教和埃及的宗教，注意到了两者之间的极大区别。现在我们将比较犹太教和阿顿神教，并且可以预料它们从起源上说是同一的。我们知道这不是一件容易事。由于阿蒙神教祭司们的报复导致的后果，我们对阿顿神教也许了解得并不充分。对于摩西宗教，我们只知道它的最后形式，即巴比伦之囚⑳后大约800年时由犹太祭司们固定下来的形式。不管这个材料多么不可靠，如果我们能够从中找到某些与我们预料一致的东西，那么也是值得我们高度重视的。

我们可以由一条捷径来证明摩西宗教就是阿顿神教，也就是说，由获准入教前所作的信仰声明或誓言来证明。但是恐怕有人会说这种方法是不现实的。众所周知，犹太教信仰的信条是这样说的，"Schema Jisroel Adonai Elohenu Adonai Echod."如果埃及神名阿顿(Aton或Atun)与希伯来语阿东赖(Adonai)，以及叙利亚神名阿东尼斯(Adonis)之间的相同之处不是一种巧

[28]

合，而是在语言和含义方面的原始统一性的结果，则我们可以把这句犹太教信条翻译如下："听着，啊，以色列(犹太人)，我

们的神阿顿〔Aton(Adonai)〕是唯一的神。"("Hear, O Israel, Our God Aton(Adonai)is the only God.")可惜！我完全无法回答这一问题，也无法在有关文献里找到答案[29]，可是我们最好不要把事情看得如此简单，当然，我们还得重新回头探讨这个神名的问题。

上述两种宗教中的异同之处很容易辨别，但却并不能使我们得到多大启迪。这两种宗教都是严格的一神教形式，我们将倾向于把两者之中的相同之处化简为这种基本的特征。犹太一神教就某些方面而言比埃及一神教更没有通融性，比如它禁止用任何可见的代替物来代表它的神。除了神名不同之外，两者之间最本质的区别就在于犹太教完全放弃了对太阳的崇拜，而埃及一神教却仍然坚持这一点。在比较犹太教和埃及民间宗教时，我们有这样的印象，即除了两者之间原则上的对立之外，还有一种有意形成的对立因素存在。我们知道，埃赫那顿怀着敌意针对当时的流行宗教精心培植起阿顿神教，当我们在比较中用它来代替犹太教时，这种印象就显得有道理了。我们惊奇而且不无理由地发现，犹太教不谈及阴间的任何事情，因为这种信条和犹太教这种最严格的一神教是谐调一致的。既然埃赫那 〔28〕
顿必须对抗当时流行的宗教，而死神奥西里斯在其中起着比上埃及地区任何其它神祇都更重大的作用，如果我们从犹太教回顾阿顿神教，并猜度犹太教的这种特点来源于阿顿神教，就不会再觉得惊奇了。犹太教与阿顿神教在此重要之点上的一致，为我们的论点提供了最有力的论据，我们将看到这还不是唯一的论据。

摩西不仅给予了犹太人新的宗教；同样可以肯定的是他传去了割礼的风俗。这对我们的问题具有决定性的意义，而这一

点几乎至今还未受到重视，当然，《圣经》上的内容经常与此相矛盾。一方面，它把这个风俗的起始期定到了希伯来族始祖亚伯拉罕[30]的时候，把它作为上帝和亚伯拉罕之间协约的标志；另一方面，《圣经》里一段特别含混的内容里又提到，上帝因为摩西疏忽了这个神圣的习俗而感到愤怒，因此决定将他杀掉以示惩罚。摩西的妻子是个米底亚人，她为了从上帝的盛怒之下挽救丈夫，慌忙中为他作了割礼手术。然而，这些只不过是些失真的记载，我们不应该因此误入歧途；我们现在就将揭开这些失真记载的目的。事实说明，涉及割礼起源的问题只有一种
〔30〕答案：它起源于埃及。"历史之父"希罗多德[31](Herodotus)告诉我们，割礼风俗在埃及已经流传了很长时期，他的说法已经从对木乃伊作的检查和古墓壁上的画中得到了证实。就我们所知，东地中海地区没有任何其他民族追随过这个风俗；我们可以肯定，闪米特人[32](Semites)、巴比伦人(Babylonias)和苏美尔人(Sumerians)都没有行过割礼。《圣经》历史上记载迦南(Canaan)的居民同样未行过割礼，在雅各(Jacob)的女儿和示剑的王子之间冒险经历[33]的故事中，行割礼的事也是假定的[34]。假如人们认为，在埃及的犹太人可能通过其它途径接受了割礼习俗，与摩西给予他们的宗教并无关系，这种说法也是很容易被驳倒的。现在让我们记住，割礼在埃及是被人们当成普遍风俗而施行的，同时也让我们暂时接受下述通常的假设：摩西是一个犹太人，他想把他的同胞们从埃及大地主的束缚下解救出来，带领他们离开埃及去开创独立自主的生活，并且，他确实完成了这一业绩。既然如此，他在那种时候强迫他们接受一种会使他们变成埃及人，并且注定会使他们保持对埃及的记忆
〔31〕的习俗，到底会有什么意义？他的目的是完全与此相反的，也

就是说，他的人民应当完全忘掉这个奴役他们的国家，应当克制对"埃及的销金窟"的留恋。上述历史事实和我由此事实而进行的推理之间如此不能相容，因此我冒昧作出下述结论：如果摩西不仅给犹太人带去了宗教，而且也带去了要他们接受割礼的法律，那么他就不是犹太人而是埃及人，而且，摩西宗教可能是一种埃及宗教，也就是说，由于摩西宗教与流行埃及的阿顿神教形成对照，我们可以得出这个结论，犹太教在某些显著之处与阿顿神教如出一辙。

如前所述，我关于摩西不是犹太人而是埃及人的假设形成了一个新的疑团。如果他是犹太人，他的所作所为不难理解，但在埃及人看来就会显得不可思议。但是，如果我们把摩西的生平放在埃赫那顿当政期间，并且假设他和这位法老有某种联系，那么这种疑团就会解开，他的动机可能就会显现出来，而使我们所有的问题都得到解答。让我们假定摩西是一个贵族，一个很有身份的人，或者像神话中传说的一样，确实是皇室的一个成员。那么，他肯定就意识得到自己巨大的能力，抱负和精力；或许他还预见到未来时期不容乐观，自己将成为人民的领袖，帝国的统治者。在同法老的密切接触中，他成了新宗教的自觉追随者，他充分理解了新教的基本原理，并且加以融汇贯通。随着埃赫那顿的死和其后的反应，他看到自己所有的希望和憧憬都断送了。如果他不愿改变自己十分珍视的信念，埃及对他来说就不再值得眷恋；他已经失去了自己的祖国，在这种 [31]
非常时刻，他发现了一种异常的解决方法。梦想家埃赫那顿已经使自己和人民变得疏远，已经使他的辽阔帝国走向崩溃。摩西的活跃天性中孕育了一项计划，他要建立一个新的帝国，寻找新的臣民，他要将遭到埃及人蔑视的宗教传给他们。我们

能够理解，这是反抗命运，从两个方向来弥补他在埃赫那顿灾祸中蒙受的损失的英勇努力。也许他在那段时期是边疆省（戈森〔Gosen〕）的总督，那些地区当时可能已进入"喜克索人时期"[2]（The Hyksos Period），某些闪米特部落已经在那里定居。他把这些闪米特人选择为他的新臣民，这不愧是一个勇敢的决定！[3]

他同他们建立起关系，充任了他们的首领，并且"用手的力量"带领他们离开埃及。我们可以估计这次大迁徙与《圣经》记载的情况完全相反，离开埃及是和平地进行的，并没有受到追踪。摩西的权威使之成为可能；并且，当时也没有中央政权的力量来阻止那次出逃。

根据我们的构想，离开埃及的大迁徙可能发生在公元前1358年至前1350年之间，那就是说，在埃赫那顿死后，荷伦希布恢复国家权威之前的一段时期[3]。那次飘泊的目的地只可能是迦南。在埃及的霸权崩溃之后，一批批好战的阿拉米人 [32]
（Arameans）已经席卷了整个国家，他们的征服和掠夺，表明了一个强悍的民族能够攫取新的土地。我们从1887年在阿马尔那废墟中发现的信件中了解了这些阿拉米武士的事迹。那时他们被称为哈比鲁人（Habiru），这种称号不知怎么传给了后来入侵的犹太人，改成了希伯来人（Hebrews），阿马尔那的信件中没有提到过这些犹太人。他们与随摩西逃离埃及的犹太人的关系最为接近，当时仍居住在巴勒斯坦以南的迦南。

我们曾猜测那次大迁徙的动机是为了寻找一片安全的乐土，割礼的创立也是出于这次迁徙的需要。虽然割礼现在很难被人理解，我们知道人类——无论是整个民族或个体——对这种古老风俗的反应。那些没有施行割礼的人觉得它非常荒唐可

憎，而那些受了割礼的则引以为自豪。他们为之感到高贵，充满优越感，并且鄙视未受割礼的人，觉得他们肮脏不堪。就是在今天，土耳其人还激烈地咒骂基督教徒，称他们为"未受割礼的狗"。我们可以相信，身为埃及人的摩西受过割礼，并且抱着这种态度。他带领着离开埃及的犹太人应该成为比他抛在身后的埃及人更好的臣民，在任何方面都必须胜过原来的臣民。他希望把他们造就为一个"神圣的民族"——《圣经》里这样明 [34]
明白白地写着——他把这个风俗介绍给他们，作为他们献身的标记，使他们至少可以成为与埃及人不相上下的人。如果这个风俗能够使他们在迁徙途中隔绝开来，防止他们与其他民族的人混交，就像埃及人处于和其他外国人隔绝的状态一样，摩西当然是会觉得高兴的[33]。

然而，后期的犹太传统却似乎与我们刚才谈及这一系列观点格格不入。如果承认割礼是由摩西传播的一种埃及风俗，就简直无异于承认摩西传下来的宗教是一种埃及宗教，但是，犹太人 [35]
也有很充足的理由来否认这一事实。因此，关于割礼的真实情况也只好如此互相矛盾。

四

在这一点上，我预料会遭受责难，因为我完成了自己的构想，把摩西当作生活在埃赫那顿时代的埃及人，并且认为他保护犹太人的决定是由于当时埃及的政治状况引起的，还把当时刚在埃及被废除了的阿顿神教当成他传给犹太人或强加给他们的那种宗教。我之所以预料会遭受责难，就是因为我太自以为是地完成了这种臆测式的构想，而在有关材料中找不到充分的依据。但是我认为这种责难是没有道理的，我已经在引言部分

强调了质疑的因素，并且在文章开首就提出了疑问。因此，我在行文之中可以省掉反复提及这一点的麻烦。

我的某些批评性的评论也许能维持这场讨论。我的论文的核心，即犹太一神教依赖于埃及历史上的一神教阶段，已经被好几位研究者们猜测和暗示过。这里我无需引证他们的著述，因为他们当中谁也没能说明这种影响到底是怎么产生的。况且，就算如我所述，这种影响是同摩西其人分不开的，我们也必须审视其它的可能性。我们不能假定官方的阿顿神教的倾复就完 [36]
全使埃及的这种一神教趋势走到了尽头。阿顿神教发源的古老北方的祭司学校，逃脱了那场倾复的灾变，并且还可能吸引了埃赫那顿之后的整整一代人遵循阿顿教的思想。因此，即使摩西没有生活在埃赫那顿时代，没有在他周围接受过影响，即使他只是北方的祭司学校的追随者或成员，他所从事的事业也是可以想像的。这种推测可以推迟那次离开埃及的大迁徙的时期，使它更接近通常假定的时期即公元前13世纪，否则提及这一点就没有什么意义。我们不应该自认为洞悉了摩西的目的，不应该认为当时盛行的无政府状态促成了那次大迁徙。继埃赫那顿之后统治埃及的第十九王朝的国王们都是些铁腕人物，有利于大迁徙的所有内、外部条件都只可能在那位持异端邪说的国王死后那段时期。

犹太人拥有丰富的圣经外文学，其中包含着在漫长的历史岁月中编撰出来的有关他们的鼻祖，宗教创立者摩西的巨人形象的神话和迷信，那些神话和迷信使摩西的形象显得空洞而模糊，没有载入摩西五卷书(the Rentateuch)的某些可靠传说的片断可能零星地散布在那类材料中。其中的一则传奇文学引人入胜地描述了摩西在童年时就已经表现了他的野心。当法老把他

抱起来逗着玩，把他高高举过头顶时，年仅三岁的摩西揭起了
法老头上的皇冠，把它戴在自己头上。那位法老被这一不祥之 [37]
兆震惊了，于是诚惶诚恐地去和他的哲人商量[36]。那些传奇还讲到摩西作为埃及军队的首领在埃塞俄比亚作战取得了胜利。他之所以逃离埃及，就是因为担心在朝庭的倾轧中引起别人甚至法老本人的嫉妒。《圣经》故事中也赋予了摩西某些使人易于相信的特色。它把摩西描述为一个暴躁易怒的人，当他义愤填膺的时候，他杀掉了虐待犹太劳工的野蛮监工；当他为手下的人变节而愤慨时，他在西奈山上打碎了为他摆好的桌子。确实，上帝最终因为他的暴躁亲自惩罚了他，我们不知道是怎样惩罚的。既然这种特色本身并不能给摩西添加光彩，它很可能就会是历史的真实事实。我们甚至不能排除这样的可能性，在犹太人早期关于上帝的概念中，他们之所以认为上帝是善嫉妒、难和解并且令人害怕的，从根底上说可能是出自他们对摩西其人的记忆，因为事实上并非一个不可见的上帝，而是摩西其人带领他们逃出了埃及。

有关摩西的另一个特征值得我们特别注意。据说他"讲话迟钝"，那就说明他肯定有口吃或发音抑制的毛病，以致于在他同法老讨论时，他只好叫亚伦(Aaron)[37]协助。这也许又是一个历史事实，并且可以使这位伟大人物的真实性增色不少。而且，这一传奇可能还具有另一种更为重要的意义。从某种可能被 [38]
讹传的情况来看，它表明摩西在没有翻译帮助时无法同他的闪米特人组成的新臣民进行语言交流，他说的另一种语言，至少在开始阶段情况是这样。由此，摩西是埃及人这一论点又有了一个新颖的证据。

现在看来，这一连串想法似乎已经走到尽头，至少暂时是

这样。不论摩西是埃及人这种臆测能否证实，我们暂时还无法推论出更多东西。没有任何史学家能够认定圣经中关于摩西和出埃及记的内容不是一个虔诚的神话，这种神话出于编撰者自身的需要而偷换了古老的传说。我们不知道那种古老传说的原始真像，也极欲探究歪曲那些传说的目的，但是却由于对历史事件茫然无知而无从着手。我们的构想没有为《圣经》记载中许多壮观的特色提供解释，例如十大灾祸[39]，穿过红海的通道，西奈山上接受十大戒律的神圣场面等，这不会使我们误入歧途。但是，我们的构想与当代历史学家们的审慎研究背道而驰，却是不能掉以轻心的事情。

这些当代历史学家以爱德华·迈耶为代表，在一个决定性的问题上追随着《圣经》记载[39]。他们赞成说，那些后来成为以 [39]
色列人的犹太部落在某个时期接受了一种新宗教，但是事情并不发生在埃及，也没发生在西奈半岛的某个山脚下，而是在一个叫麦内巴特—夸底斯(Meribat-Qades)的地方，位置在阿拉伯半岛西部与西奈半岛东部之间的南巴勒斯坦荒野中的一个绿州，该地以泉水众多而闻名。犹太人在那里接受了对上帝耶和华的崇拜，很可能是从附近的阿拉伯米底亚人部落接受过来的。那个地区附近的其它部落可能也是耶和华的信徒。

耶和华显然是一个火山神，然而，就我们所知，埃及并没有火山，西奈半岛的山脉中也从没有火山爆发。另一方面，阿拉伯半岛的西部边缘后来可能才发现有火山活动，其中之一肯定是传说中耶和华住处的西奈—霍内布[38](Sinai-Horeb)，不论《圣经》内容遭受了多少篡改，根据迈耶的说法，我们还是能够重视耶和华的原始性格：他是一个怪诞的嗜血成性的恶魔，他避开白昼的阳光而在黑夜里出没[38]。

在这个新宗教诞生时，犹太人与耶和华之间的媒介者就是摩西。他是米底亚祭司叶忒罗(Jethro)的女婿，他在放牧羊群的时候受到了上帝耶和华的召唤。叶忒罗在夸底斯会见了摩西，给了他一些教诲。

爱德华·迈耶说，他历来确信犹太人在埃及遭受奴役的故 (40)
事与埃及人那场异常的突变之间有真实的历史联系㊲，但是很明显，他不知道那一已被确认的事实归属何处，有何作用。他只愿意承认割礼风俗是由埃及人而来的，他用两项重要的建议丰富了我们上述的讨论：第一，约书亚(Joshua)叫人们接受割礼，"以便卷走埃及人的责难"；第二，据希罗多德所述，腓尼基人(可能就是指的犹太人)和巴勒斯坦的叙利亚人自己曾承认割礼风俗是从埃及人学来的㊳。但是，埃及人摩西并没有引起他的兴趣。他说："我们所知的摩西是夸底斯的祭司们的祖先；因此，他只与那种崇拜有关，他只是宗谱神话中的一个人物，而不是一个历史人物。"除了那些把传说统统当成历史真实来接受的人之外，在把摩西当成真实历史人物看待的人中，没有任何人曾经成功地在这个空洞形象中充实过任何内容，把他描述为一个具体的人：他们没能告诉我们他在历史上的成就或使命㊴。

另一方面，迈耶不厌其烦地告诉我们关于摩西与夸底斯和米底亚的关系。"摩西其人与米底亚和沙漠中的圣地紧密相关……"㊵"摩西这个人与夸底斯（玛撒和麦内巴(Massa and Meriba)）结下了不解之缘；通过婚姻与一个米底亚祭司结成的 (41)
关系更促成了这种联系。另一方面，他与那次大迁徙的联系，以及他青年时代的全部故事，都绝对是次要的，并且只不过是摩西必须符合一个连续相关的故事的结果。"㊶他也观察到了摩西青年时代故事里包含的特征后来都被省略了。"摩西在米底亚

不再是一个埃及人，不再是法老的孙子，而是一个牧羊人，耶和华对他显了圣，在十大灾祸的故事中，虽然他原先的关系很可以被有效的利用，《圣经》里却再也没有提及；杀死以色列男婴的命令也完全给遗忘了㊷。在出埃及记中和那些埃及人的毁灭中，摩西没有起任何作用，甚至没有被提到。他的童年故事里的英雄特色在成年摩西身上完全看不到了，他只不过成了上帝的使者，奇迹的执行人，他显示的超自然力量都是由耶和华提供的。"㊸

我们无法打消这样的印象，虽然传说中把制造救命铜蛇㊹也归功于摩西，可是这个在夸底斯和米底亚的摩西与我们所推论出的那个把严禁一切巫术的宗教带给他的新臣民的威风凛凛的埃及人还是判若两人。埃及人摩西与米底亚人摩西的区别之大，也许不亚于宇宙神阿顿与米底亚神山上的魔神耶和华的区 (42)
别。如果我们果真相信现代历史学家们提供的资料是真实的，则我们就必须承认，我们指望从摩西是埃及人这一推测中引出的线索将再度中断，而且，这次中断看起来没有任何希望再接续起来。

五

然而，柳暗花明又一村。继迈耶之后，格雷斯曼(Gressmann)和其他人继续努力研究摩西，摆脱了夸底斯的祭司一说的局限，并且证实了传说中赋予摩西的声誉。1922年，厄恩斯特·塞林(Ernst Sellin)获得了一个具有决定性意义的发现㊺。他在先知何西阿书㊻(公元前8世纪中叶)中找到了一些不容质疑的线索，说明犹太教的创立者摩西在他的倔强固执的人民的一次反叛中遭受了厄运，他所创立的宗教也同时被抛弃。这一

传说不仅出现在何西阿书中，在其后大多数先知的著述中都出现过；按塞林的见解，它确实是后来犹太人期待救主弥赛亚⑰(Messiah)的根本原因。巴比伦之囚的末期，犹太人中产生了一种希望，希望那被他们无情地杀害了的摩西从阴间归来，带领他的悔罪的人民——也许还不止他的人民——进入永恒的极乐世界。在我们目前的探讨过程中，我们看不出这种希望与那位后期宗教的建立者(指基督)的命运之间的明显联系。 [43]

自然，我无法确定塞林是否正确地解释了何西阿书中的有关章节。但是如果他是正确的，我们则可以把他所辨析出的传说当作历史上真实可信的事件，因为那种事件不是信手可以杜撰的，况且也真的不存在这样作的目的。如果那类事件真正发生过，则要想忘掉它们是很容易理解的。我们用不着接受那一传说的所有细节。塞林认为约旦河东岸的希廷 (Shittim) 是那场暴力发生的地点。然而我们将会看到，这个地点的选择与我们的论据不相一致。

让我们接受塞林的猜测，即认为埃及人摩西是被犹太人所杀，他所创立的宗教也被犹太人抛弃了。这样，我们就能够避免同历史研究中令人信服的结果发生矛盾，进一步理清我们的问题。但是在其它方面我们却要冒险排开那些历史学家们的见解，开辟出自己的道路。脱离埃及的大迁徙仍然是我们讨论问题的出发点，随同摩西离开埃及的犹太人数量肯定极为可观；一小批人不值得雄心勃勃的摩西大动干戈。埃及的犹太移民到那时可能已经居留了相当长的时间，繁衍成了一个数目可观的民族。然而，如果我们和大多数研究者一样，估计后来形成犹太民族的那一部分人在埃及遭受过奴役，当然也不至于有多大的失误。换句话说，从埃及归来的那些部落后来同其它散布 [44]

在埃及与迦南之间的有关部落联合起来，形成了以色列民族。那次联合以所有部落都接受一种信仰耶和华的新宗教为标志；据迈耶的见解，那次联合是由于米底亚人的影响在奇底斯发生的。自此，以色列民族觉得自身已经强大得足以入侵迦南。可是，这种见解与摩西蒙难以及他的宗教遭抛弃的一系列事件发生在约旦河东岸的说法不相吻合，入侵迦南肯定发生在那次联合之前很久的时间。

可以肯定，在犹太民族的形成过程中，有许多非常不同的因素起了作用，但是其中最重要的一点就在于他们是否曾在埃及居留，其后发生过什么事件。从这个观点出发，我们也许可以说，这个民族是由两种成份组成的。在一段短期的政治联合之后，犹太民族分裂成两部分——即以色列王国和犹太王国。合久必分，分久必合，历史偏爱这样的轮回。欧洲16世纪时期的宗教改革为此提供了一个最有说服力的众所周知的例子，经过一千多年的间歇之后，曾经是罗马天主教势力范围的德国与那一直保持独立的部分之间的分裂又暴露出来。对于犹太人，我们无法证实是否又出现过原来那种分裂局面，我们对那些时 [45] 代的情况不了解，无法推测北部王国(以色列王国)是否吸收了原来的定居者，南部王国(犹太王国)是否吸收了那些从埃及归来的人；但是，在这种情况下，后来的分裂不可能与早期的联合没有牵连。原来在埃及居住的犹太人数量可能比其他犹太人少，但是他们却表明具有较高的文化水准，因为他们从埃及带来了一种其他犹太人所没有的传统，他们在犹太民族后来的发展中产生了更为重要的影响。

也许他们还带来了其它东西，某种比上述传统更为具体的东西。犹太人史前时期的最难解之迷是利未人(Levites)的身世。

Letting the Bones Show: Why Models Matter to Users

In theory, perhaps, you could present a user with any view of a system, regardless of what lies beneath. But in practice, a mismatch causes confusion at best—bugs at worst. Consider a very simple example of how users are misled by superimposed models of bookmarks for Web sites in current releases of Microsoft Internet Explorer.[1]

[1] Brian Marick mentioned this example to me.

A user of Internet Explorer thinks of "Favorites" as a list of names of Web sites that persist from session to session. But the implementation treats a Favorite as a file containing a URL, and whose filename is put in the Favorites list. That's a problem if the Web page title contains characters that are illegal in Windows filenames. Suppose a user tries to store a Favorite and types the following name for it: "Laziness: The Secret to Happiness". An error message will say: "A filename cannot contain any of the following characters: \/ : * ? " < > |". What filename? On the other hand, if the Web page title already contains an illegal character, Internet Explorer will just quietly strip it out. The loss of data may be benign in this case, but not what the user would have expected. Quietly changing data is completely unacceptable in most applications.

MODEL-DRIVEN DESIGN calls for working with only one model (within any single context, as will be discussed in Chapter 14). Most of the advice and examples go to the problems of having separate analysis models and design models, but here we have a problem arising from a different pair of models: the user model and the design/implementation model.

Of course, an unadorned view of the domain model would definitely not be convenient for the user in most cases. But trying to create in the UI an illusion of a model other than the domain model will cause confusion unless the illusion is perfect. If Web Favorites are actually just a collection of shortcut files, then expose this fact to the user and eliminate the confusing alternative model. Not only will the feature be less confusing, but the user can then leverage what he knows about the file system to deal with Web Favorites. He can reorganize them with the File Explorer, for example, rather than use awkward tools built into the Web browser. Informed users would be more able to exploit the flexibility of storing Web shortcuts anywhere in the file system. Just by removing the misleading extra model, the power of the application would increase and become clearer. Why make the user learn a new model when the programmers felt the old model was good enough?

Alternatively, store the Favorites in a different way, say in a data file, so that they can be subject to their own rules. Those rules would presumably be the naming rules that apply to Web pages. That would again provide a single model. This one tells the user that everything he knows about naming Web sites applies to Favorites.

When a design is based on a model that reflects the basic concerns of the users and domain experts, the bones of the design can be revealed to the user to a greater extent than with other design approaches. Revealing the model gives the user more access to the potential of the software and yields consistent, predictable behavior.

Hands-On Modelers

Manufacturing is a popular metaphor for software development. One inference from this metaphor: highly skilled engineers design; less skilled laborers assemble the products. This metaphor has messed up a lot of projects for one simple reason—software development is *all* design. All teams have specialized roles for members, but overseparation of responsibility for analysis, modeling, design, and programming interferes with MODEL-DRIVEN DESIGN.

On one project, my job was to coordinate different application teams and help develop the domain model that would drive the design. But the management thought that modelers should be modeling, and that coding was a waste of those skills, so I was in effect forbidden to program or work on details with programmers.

Things seemed to be OK for a while. Working with domain experts and the development leads of the different teams, we crunched knowledge and refined a nice core model. But that model was never put to work, for two reasons.

First, some of the model's intent was lost in the handoff. The overall effect of a model can be very sensitive to details (as will be discussed in Parts II and III), and those details don't always come across in a UML diagram or a general discussion. If I could have rolled up my sleeves and worked with the other developers directly, providing some code to follow as examples, and providing some close support, the team could have taken up the abstractions of the model and run with them.

The other problem was the indirectness of feedback from the interaction of the model with the implementation and the technology. For example, certain aspects of the model turned out to be wildly in-efficient on our technology platform, but the full implications didn't trickle back to me for months. Relatively minor changes could have fixed the problem, but by then it didn't matter. The developers were well on their way to writing software that did work—without the model, which had been reduced to a mere data structure, wherever it was still used at all. The developers had thrown the baby out with the bathwater, but what choice did they have? They could no longer risk being saddled with the dictates of the architect in the ivory tower.

The initial circumstances of this project were about as favorable to a hands-off modeler as they ever are. I already had extensive hands-on experience with most of the technology used on the project. I had even led a small development team on the same project before my role changed, so I was familiar with the project's development process and programming environment. Even those factors were not enough to make me effective, given the separation of modeler from implementation.

If the people who write the code do not feel responsible for the model, or don't understand how to make the model work for an application, then the model has nothing to do with the software. If developers don't realize that changing code changes the model, then their refactoring will weaken the model rather than strengthen it. Meanwhile, when a modeler is separated from the implementation process, he or she never acquires, or quickly loses, a feel for the constraints of implementation. The basic constraint of MODEL-DRIVEN DESIGN—that the model supports an effective implementation and abstracts key domain knowledge—is half-gone, and the resulting models will be impractical. Finally, the knowledge and skills of experienced designers won't be transferred to other developers if the division of labor prevents the kind of collaboration that conveys the subtleties of coding a MODEL-DRIVEN DESIGN.

The need for HANDS-ON MODELERS does not mean that team members cannot have specialized

roles. Every Agile process, including Extreme Programming, defines roles for team members, and other informal specializations tend to emerge naturally. The problem arises from separating two tasks that are coupled in a MODEL-DRIVEN DESIGN, modeling and implementation.

The effectiveness of an overall design is very sensitive to the quality and consistency of fine-grained design and implementation decisions. With a MODEL-DRIVEN DESIGN, a portion of the code is an expression of the model; changing that code changes the model. Programmers are modelers, whether anyone likes it or not. So it is better to set up the project so that the programmers do good modeling work.

Therefore:

Any technical person contributing to the model must spend some time touching the code, whatever primary role he or she plays on the project. Anyone responsible for changing code must learn to express a model through the code. Every developer must be involved in some level of discussion about the model and have contact with domain experts. Those who contribute in different ways must consciously engage those who touch the code in a dynamic exchange of model ideas through the UBIQUITOUS LANGUAGE.

The sharp separation of modeling and programming doesn't work, yet large projects still need technical leaders who coordinate high-level design and modeling and help work out the most difficult or most critical decisions. Part IV, "Strategic Design," deals with such decisions and should stimulate ideas for more productive ways to define the roles and responsibilities of high-level technical people.

Domain-driven design puts a model to work to solve problems for an application. Through knowledge crunching, a team distills a torrent of chaotic information into a practical model. A MODEL-DRIVEN DESIGN intimately connects the model and the implementation. The UBIQUITOUS LANGUAGE is the channel for all that information to flow between developers, domain experts, and the software.

The result is software that provides rich functionality based on a fundamental understanding of the core domain.

As mentioned, success with MODEL-DRIVEN DESIGN is sensitive to detailed design decisions, which is the subject of the next several chapters.

Part II: The Building Blocks of a Model-Driven Design

To keep a software implementation crisp and in lockstep with a model, in spite of messy realities, you must apply the best practices of modeling and design. This book is not an introduction to object-oriented design, nor does it propose radical design fundamentals. Domain-driven design shifts the emphasis of certain conventional ideas.

Certain kinds of decisions keep the model and implementation aligned with each other, each reinforcing the other's effectiveness. This alignment requires attention to the details of individual elements. Careful crafting at this small scale gives developers a steady platform from which to apply the modeling approaches of Parts III and IV.

The design style in this book largely follows the principle of "responsibility-driven design," put forward in Wirfs-Brock et al. 1990 and updated in Wirfs-Brock 2003. It also draws heavily (especially in Part III) on the ideas of "design by contract" described in Meyer 1988. It is consistent with the general background of other widely held best practices of object-oriented design, which are described in such books as Larman 1998.

As a project hits bumps, large or small, developers may find themselves in situations that make those principles seem inapplicable. To make the domain-driven design process resilient, developers need to understand *how* the well-known fundamentals support MODEL-DRIVEN DESIGN, so they can compromise without derailing.

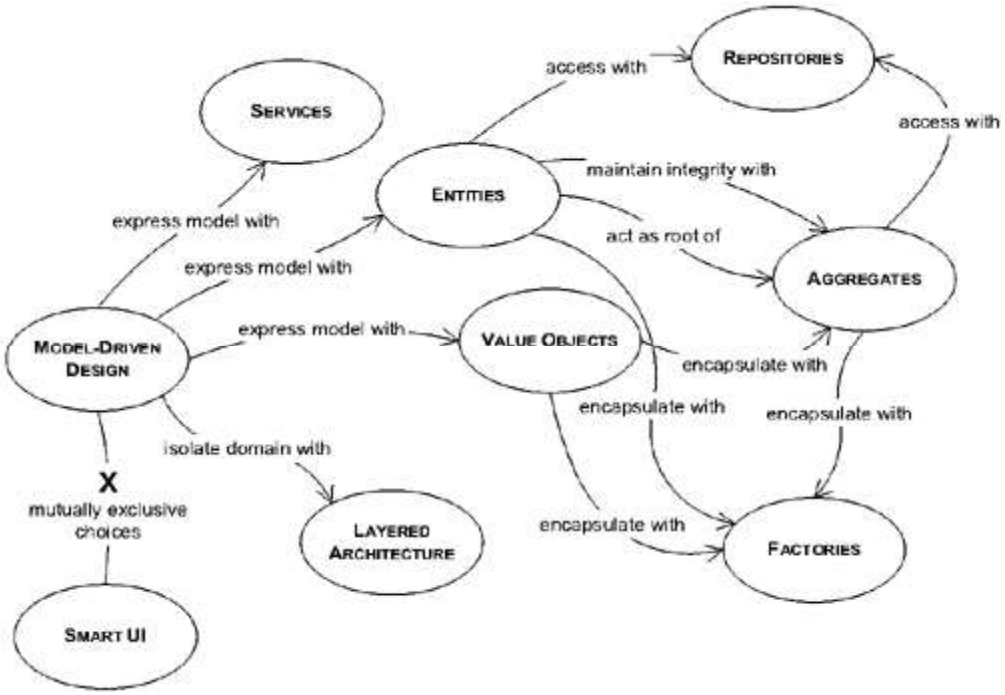
The material in the following three chapters is organized as a "pattern language" (see Appendix A), which will show how subtle model distinctions and design decisions affect the domain-driven design process.

The diagram on the top of the next page is a *navigation map*. It shows the patterns that will be presented in this section and a few of the ways they relate to each other.

Sharing these standard patterns brings order to the design and makes it easier for team members to understand each other's work. Using standard patterns also adds to the UBIQUITOUS LANGUAGE, which all team members can use to discuss model and design decisions.

Developing a good domain model is an art. But the practical design and implementation of a model's individual elements can be relatively systematic. Isolating the domain design from the mass of other concerns in the software system will greatly clarify the design's connection to the model. Defining model elements according to certain distinctions sharpens their meanings. Following proven patterns for individual elements helps produce a model that is practical to implement.

A navigation map of the language of MODEL-DRIVEN DESIGN



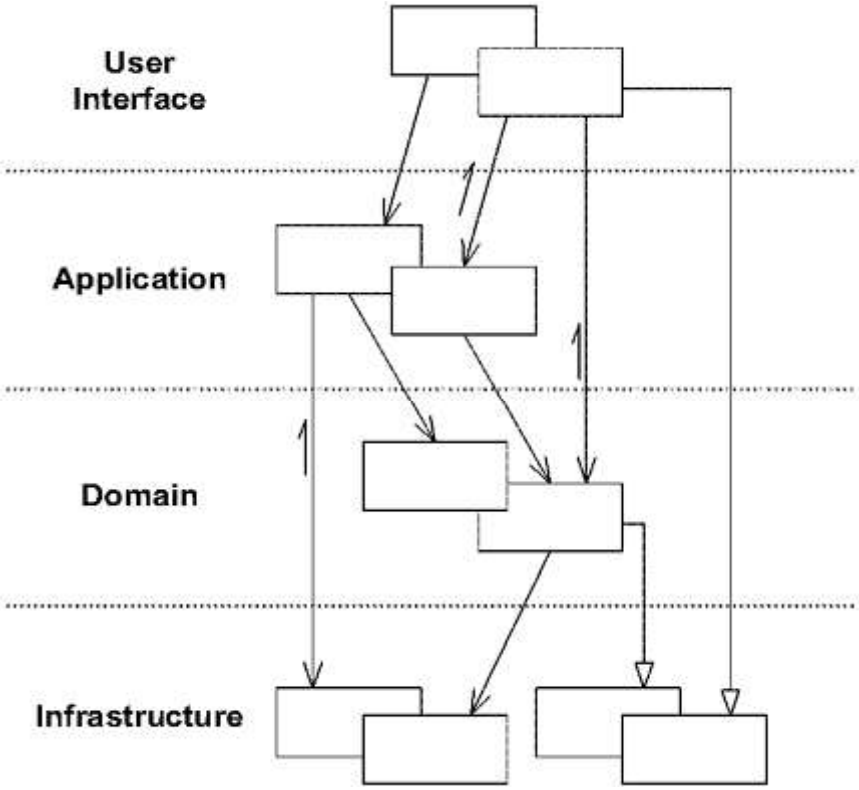
Elaborate models can cut through complexity only if care is taken with the fundamentals, resulting in detailed elements that the team can confidently combine.

Chapter Four. Isolating the Domain

The part of the software that specifically solves problems from the domain usually constitutes only a small portion of the entire software system, although its importance is disproportionate to its size. To apply our best thinking, we need to be able to look at the elements of our model and see them as a system. We must not be forced to pick them out of a much larger mix of objects, like trying to identify constellations in the night sky. We need to decouple the domain objects from other functions of the system, so we can avoid confusing the domain concepts with other concepts related only to software technology or losing sight of the domain altogether in the mass of the system.

Sophisticated techniques for this isolation have emerged. This is well-trodden ground, but it is so critical to the successful application of domain-modeling principles that it must be reviewed briefly, from a domain-driven point of view. . . .

Layered Architecture



For a shipping application to support the simple user act of selecting a cargo's destination from a list of cities, there must be program code that (1) draws a widget on the screen, (2) queries the database for all the possible cities, (3) interprets the user's input and validates it, (4) associates the selected city with the cargo, and (5) commits the change to the database. All of this code is part of the same program, but only a little of it is related to the business of shipping.

Software programs involve design and code to carry out many different kinds of tasks. They accept user input, carry out business logic, access databases, communicate over networks, display information to users, and so on. So the code involved in each program function can be substantial.

In an object-oriented program, UI, database, and other support code often gets written directly into the business objects. Additional business logic is embedded in the behavior of UI widgets and data-base scripts. This happens because it is the easiest way to make things work, in the short run.

When the domain-related code is diffused through such a large amount of other code, it becomes extremely difficult to see and to reason about. Superficial changes to the UI can actually change business logic. To change a business rule may require meticulous tracing of UI code, database code, or other program elements. Implementing coherent, model-driven objects becomes impractical. Automated testing is awkward. With all the technologies and logic involved in each activity, a program must be kept very simple or it becomes impossible to understand.

Creating programs that can handle very complex tasks calls for separation of concerns, allowing concentration on different parts of the design in isolation. At the same time, the intricate

interactions within the system must be maintained in spite of the separation.

There are all sorts of ways a software system might be divided, but through experience and convention, the industry has converged on LAYERED ARCHITECTURES, and specifically a few fairly standard layers. The metaphor of layering is so widely used that it feels intuitive to most developers. Many good discussions of layering are available in the literature, sometimes in the format of a pattern (as in Buschmann et al. 1996, pp. 31–51). The essential principle is that any element of a layer depends only on other elements in the same layer or on elements of the layers "beneath" it. Communication upward must pass through some indirect mechanism, which I'll discuss a little later.

The value of layers is that each specializes in a particular aspect of a computer program. This specialization allows more cohesive designs of each aspect, and it makes these designs much easier to interpret. Of course, it is vital to choose layers that isolate the most important cohesive design aspects. Again, experience and convention have led to some convergence. Although there are many variations, most successful architectures use some version of these four conceptual layers:

User Interface (or Presentation Layer)	Responsible for showing information to the user and interpreting the user's commands. The external actor might sometimes be another computer system rather than a human user.
Application Layer	Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.
Domain Layer (or Model Layer)	Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. *This layer is the heart of business software.*
Infrastructure Layer	Provides generic technical capabilities that support the higher layers: message sending for the application, persistence for the domain, drawing widgets for the UI, and so on. The infrastructure layer may also support the pattern of interactions between the four layers through an architectural framework.

Some projects don't make a sharp distinction between the user interface and application layers. Others have multiple infrastructure layers. But it is the crucial separation of the *domain layer* that enables MODEL-DRIVEN DESIGN.

Therefore:

Partition a complex program into layers. Develop a design within each layer that is cohesive and that depends only on the layers below. Follow standard architectural patterns to provide loose coupling to the layers above. Concentrate all the code related to the domain model in one layer and isolate it from the user interface, application, and infrastructure code. The domain objects, free of the responsibility of displaying themselves, storing themselves, managing application tasks, and so forth, can be

focused on expressing the domain model. This allows a model to evolve to be rich enough and clear enough to capture essential business knowledge and put it to work.

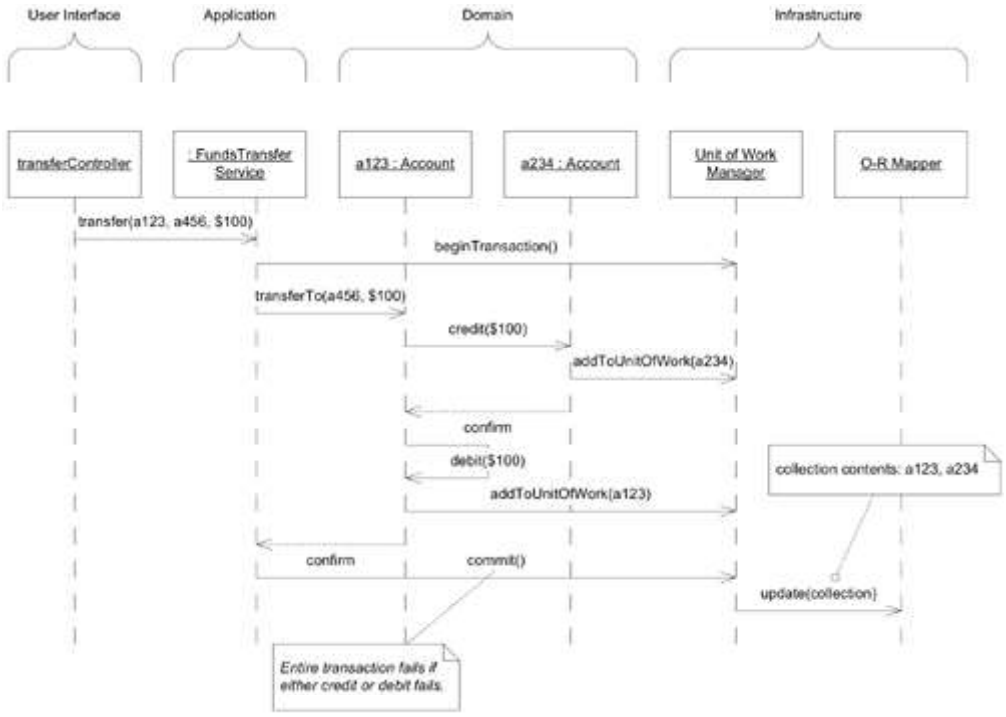
Separating the domain layer from the infrastructure and user interface layers allows a much cleaner design of each layer. Isolated layers are much less expensive to maintain, because they tend to evolve at different rates and respond to different needs. The separation also helps with deployment in a distributed system, by allowing different layers to be placed flexibly in different servers or clients, in order to minimize communication overhead and improve performance (Fowler 1996).

Example
Partitioning Online Banking Functionality into Layers

An application provides various capabilities for maintaining bank accounts. One feature is funds transfer, in which the user enters or chooses two account numbers and an amount of money and then initiates a transfer.

To make this example manageable, I've omitted major technical features, most notably security. The domain design is also oversimplified. (Realistic complexity would only increase the need for layered architecture.) Furthermore, the particular infrastructure implied here is meant to be simple and obvious to make the example clear—it is not a suggested design. The responsibilities of the remaining functionality would be layered as shown in Figure 4.1.

Figure 4.1. Objects carry out responsibilities consistent with their layer and are more coupled to other objects in their layer.



Note that the domain layer, *not the application layer*, is responsible for fundamental business rules—in this case, the rule is "Every credit has a matching debit."

The application also makes no assumptions about the source of the transfer request. The program

presumably includes a UI with entry fields for account numbers and amounts and with buttons for commands. But that user interface could be replaced by a wire request in XML without affecting the application layer or any of the lower layers. This decoupling is important not because projects frequently need to replace user interfaces with wire requests but because a clean separation of concerns keeps the design of each layer easy to understand and maintain.

In fact, Figure 4.1 itself mildly illustrates the problem of not isolating the domain. Because everything from the request to transaction control had to be included, the domain layer had to be dumbed down to keep the overall interaction simple enough to follow. If we were focused on the design of the isolated domain layer, we would have space on the page and in our heads for a model that better represented the domain's rules, perhaps including ledgers, credit and debit objects, or monetary transaction objects.

Relating the Layers

So far the discussion has focused on the separation of layers and the way in which that partitioning improves the design of each aspect of the program, particularly the domain layer. But of course, the layers have to be connected. To do this without losing the benefit of the separation is the motivation behind a number of patterns.

Layers are meant to be loosely coupled, with design dependencies in only one direction. Upper layers can use or manipulate elements of lower ones straightforwardly by calling their public interfaces, holding references to them (at least temporarily), and generally using conventional means of interaction. But when an object of a lower level needs to communicate upward (beyond answering a direct query), we need another mechanism, drawing on architectural patterns for relating layers such as callbacks or OBSERVERS (Gamma et al. 1995).

The grandfather of patterns for connecting the UI to the application and domain layers is MODEL-VIEW-CONTROLLER (MVC). It was pioneered in the Smalltalk world back in the 1970s and has inspired many of the UI architectures that followed. Fowler (2002) discusses this pattern and several useful variations on the theme. Larman (1998) explores these concerns in the MODEL-VIEW SEPARATION PATTERN, and his APPLICATION COORDINATOR is one approach to connecting the application layer.

There are other styles of connecting the UI and the application. For our purposes, all approaches are fine as long as they maintain the isolation of the domain layer, allowing domain objects to be designed without simultaneously thinking about the user interface that might interact with them.

The infrastructure layer usually does not initiate action in the domain layer. Being "below" the domain layer, it should have no specific knowledge of the domain it is serving. Indeed, such technical capabilities are most often offered as SERVICES. For example, if an application needs to send an e-mail, some message-sending interface can be located in the infrastructure layer and the application layer elements can request the transmission of the message. This decoupling gives some extra versatility. The message-sending interface might be connected to an e-mail sender, a fax sender, or whatever else is available. But the main benefit is simplifying the application layer, keeping it narrowly focused on its job: knowing *when* to send a message, but not burdened with *how*.

The application and domain layers call on the SERVICES provided by the infrastructure layer. When the scope of a SERVICE has been well chosen and its interface well designed, the caller can remain loosely coupled and uncomplicated by the elaborate behavior the SERVICE interface encapsulates.

But not all infrastructure comes in the form of SERVICES callable from the higher layers. Some technical components are designed to directly support the basic functions of other layers (such as providing an abstract base class for all domain objects) and provide the mechanisms for them to