relate (such as implementations of MVC and the like). Such an "architectural framework" has much more impact on the design of the other parts of the program.

Architectural Frameworks

When infrastructure is provided in the form of SERVICES called on through interfaces, it is fairly intuitive how the layering works and how to keep the layers loosely coupled. But some technical problems call for more intrusive forms of infrastructure. Frameworks that integrate many infrastructure needs often require the other layers to be implemented in very particular ways, for example as a subclass of a framework class or with structured method signatures. (It may seem counterintuitive for a subclass to be in a layer higher than that of the parent class, but keep in mind which class reflects more knowledge of the other.) The best architectural frameworks solve complex technical problems while allowing the domain developer to concentrate on expressing a model. But frameworks can easily get in the way, either by making too many assumptions that constrain domain design choices or by making the implementation so heavyweight that development slows down.

Some form of architectural framework usually is needed (though sometimes teams choose frameworks that don't serve them well). When applying a framework, the team needs to focus on its goal: building an implementation that expresses a domain model and uses it to solve important problems. The team must seek ways of employing the framework to those ends, even if it means not using all of the framework's features. For example, early J2EE applications often implemented all domain objects as "entity beans." This approach bogged down both performance and the pace of development. Instead, current best practice is to use the J2EE framework for larger grain objects, implementing most business logic with generic Java objects. A lot of the downside of frameworks can be avoided by applying them selectively to solve difficult problems without looking for a one-size-fits-all solution. Judiciously applying only the most valuable of framework features reduces the coupling of the implementation and the framework, allowing more flexibility in later design decisions. More important, given how very complicated many of the current frameworks are to use, this minimalism helps keep the business objects readable and expressive.

Architectural frameworks and other tools will continue to evolve. Newer frameworks will automate or prefabricate more and more of the technical aspects of an application. If this is done right, application developers will increasingly concentrate their time on modeling the core business problems, greatly improving productivity and quality. But as we move in this direction, we must guard against our enthusiasm for technical solutions; elaborate frameworks can also straitjacket application developers.

The Domain Layer Is Where the Model Lives

LAYERED ARCHITECTURE is used in most systems today, under various layering schemes. Many styles of development can also benefit from layering. However, domain-driven design requires only one particular layer to exist.

The domain model is a set of concepts. The "domain layer" is the manifestation of that model and all directly related design elements. The design and implementation of business logic constitute the domain layer. In a MODEL-DRIVEN DESIGN, the software constructs of the domain layer mirror the model concepts.

It is not practical to achieve that correspondence when the domain logic is mixed with other concerns of the program. Isolating the domain implementation is a prerequisite for domain-driven design.

The Smart UI "Anti-Pattern"

. . . That sums up the widely accepted LAYERED ARCHITECTURE pattern for object applications. But this separation of UI, application, and domain is so often attempted and so seldom accomplished that its negation deserves a discussion in its own right.

Many software projects do take and should continue to take a much less sophisticated design approach that I call the SMART UI. But SMART UI is an alternate, mutually exclusive fork in the road, incompatible with the approach of domain-driven design. If that road is taken, most of what is in this book is not applicable. My interest is in the situations where the SMART UI does not apply, which is why I call it, with tongue in cheek, an "anti-pattern." Discussing it here provides a useful contrast and will help clarify the circumstances that justify the more difficult path taken in the rest of the book.

A project needs to deliver simple functionality, dominated by data entry and display, with few business rules. Staff is not composed of advanced object modelers.

If an unsophisticated team with a simple project decides to try a MODEL-DRIVEN DESIGN with LAYERED ARCHITECTURE, it will face a difficult learning curve. Team members will have to master complex new technologies and stumble through the process of learning object modeling (which is challenging, even with the help of this book!). The overhead of managing infrastructure and layers makes very simple tasks take longer. Simple projects come with short time lines and modest expectations. Long before the team completes the assigned task, much less demonstrates the exciting possibilities of its approach, the project will have been canceled.

Even if the team is given more time, the team members are likely to fail to master the techniques without expert help. And in the end, if they do surmount these challenges, they will have produced a simple system. Rich capabilities were never requested.

A more experienced team would not face the same trade-offs. Seasoned developers could flatten the learning curve and compress the time needed to manage the layers. Domain-driven design pays off best for ambitious projects, and it does require strong skills. Not all projects are ambitious. Not all project teams can muster those skills.

Therefore, when circumstances warrant:

Put all the business logic into the user interface. Chop the application into small functions and implement them as separate user interfaces, embedding the business rules into them. Use a relational database as a shared repository of the data. Use the most automated UI building and visual programming tools available.

Heresy! The gospel (as advocated everywhere, including else-where in this book) is that domain and UI should be separate. In fact, it is difficult to apply any of the methods discussed later in this book without that separation, and so this SMART UI can be considered an "anti-pattern" in the context of domain-driven design. Yet it is a legitimate pattern in some other contexts. In truth, there are advantages to the SMART UI, and there are situations where it works best—which partially accounts for why it is so common. Considering it here helps us understand why we need to separate application from domain and, importantly, when we might not want to.

Advantages

- Productivity is high and immediate for simple applications.
- Less capable developers can work this way with little training.
- Even deficiencies in requirements analysis can be overcome by releasing a prototype to users and then quickly changing the product to fit their requests.
- Applications are decoupled from each other, so that delivery schedules of small modules can be planned relatively accurately. Expanding the system with additional, simple behavior can be easy.
- Relational databases work well and provide integration at the data level.
- 4GL tools work well.
- When applications are handed off, maintenance programmers will be able to quickly redo portions they can't figure out, because the effects of the changes should be localized to each particular UI.

Disadvantages

- Integration of applications is difficult except through the database.
- There is no reuse of behavior and no abstraction of the business problem. Business rules have to be duplicated in each operation to which they apply.
- Rapid prototyping and iteration reach a natural limit because the lack of abstraction limits refactoring options.
- Complexity buries you quickly, so the growth path is strictly toward additional simple applications. There is no graceful path to richer behavior.

If this pattern is applied consciously, a team can avoid taking on a great deal of overhead required by other approaches. It is a common mistake to undertake a sophisticated design approach that the team isn't committed to carrying all the way through. Another common, costly mistake is to build a complex infrastructure and use industrial-strength tools for a project that doesn't need them.

Most flexible languages (such as Java) are overkill for these applications and will cost dearly. A 4GL-style tool is the way to go.

Remember, one of the consequences of this pattern is that you can't migrate to another design approach except by replacing entire applications. Just using a general-purpose language such as Java won't really put you in a position to later abandon the SMART UI, so if you've chosen that path, you should choose development tools geared to it. Don't bother hedging your bet. Just using a flexible language doesn't create a flexible system, but it may well produce an expensive one.

By the same token, a team committed to a MODEL-DRIVEN DESIGN needs to design that way from the outset. Of course, even experienced project teams with big ambitions have to start with simple functionality and work their way up through successive iterations. But those first tentative steps will be MODEL-DRIVEN with an isolated domain layer, or the project will most likely be stuck with a

SMART UI. The SMART UI is discussed only to clarify why and when a pattern such as LAYERED ARCHITECTURE is needed in order to isolate a domain layer.

There are other solutions in between SMART UI and LAYERED ARCHITECTURE. For example, Fowler (2002) describes the TRANSACTION SCRIPT, which separates UI from application but does not provide for an object model. The bottom line is this: *If the architecture isolates the domain-related code in a way that allows a cohesive domain design loosely coupled to the rest of the system, then that architecture can probably support domain-driven design.*

Other development styles have their place, but you must accept varying limits on complexity and flexibility. Failing to decouple the domain design can really be disastrous in certain settings. If you have a complex application and are committing to MODEL-DRIVEN DESIGN, bite the bullet, get the necessary experts, and *avoid* the SMART UI.

Other Kinds of Isolation

Unfortunately, there are influences other than infrastructure and user interfaces that can corrupt your delicate domain model. You must deal with other domain components that are not fully integrated into your model. You have to cope with other development teams who use different models of the same domain. These and other factors can blur your model and rob it of its utility. Chapter 14, "Maintaining Model Integrity," deals with this topic, introducing such patterns as BOUNDED CONTEXT and ANTICORRUPTION LAYER. A really complicated domain model can become unwieldy all by itself. Chapter 15, "Distillation," discusses how to make distinctions within the domain layer that can unencumber the essential concepts of the domain from peripheral detail.

But all that comes later. Next, we'll look at the nuts and bolts of co-evolving an effective domain model and an expressive implementation. After all, the best part of isolating the domain is getting all that other stuff out of the way so that we can really focus on the domain design.

Chapter Five. A Model Expressed in Software

To compromise in implementation without losing the punch of a MODEL-DRIVEN DESIGN requires a reframing of the basics. Connecting model and implementation has to be done at the detail level. This chapter focuses on those individual model elements, getting them in shape to support the activities in later chapters.

This discussion will start with the issues of designing and streamlining associations. Associations between objects are simple to conceive and to draw, but implementing them is a potential quagmire. Associations illustrate how crucial detailed implementation decisions are to the viability of a MODEL-DRIVEN DESIGN.

Turning to the objects themselves, but continuing to scrutinize the relationship between detailed model choices and implementation concerns, we'll focus on making distinctions among the three patterns of model elements that express the model: ENTITIES, VALUE OBJECTS, and SERVICES.

Defining objects that capture concepts of the domain seems very intuitive on the surface, but serious challenges are lurking in the shades of meaning. Certain distinctions have emerged that clarify the meaning of model elements and tie into a body of design practices for carving out specific kinds of objects.

Does an object represent something with continuity and identity—something that is tracked through different states or even across different implementations? Or is it an attribute that describes the state of something else? This is the basic distinction between an ENTITY and a VALUE OBJECT. Defining objects that clearly follow one pattern or the other makes the objects less ambiguous and lays out the path toward specific choices for robust design.

Then there are those aspects of the domain that are more clearly expressed as actions or operations, rather than as objects. Although it is a slight departure from object-oriented modeling tradition, it is often best to express these as SERVICES, rather than forcing responsibility for an operation onto some ENTITY or VALUE OBJECT. A SERVICE is something that is done for a client on request. In the technical layers of the software, there are many SERVICES. They emerge in the domain also, when some activity is modeled that corresponds to something the software must do, but does not correspond with state.

There are inevitable situations in which the purity of the object model must be compromised, such as for storage in a relational database. This chapter will lay out some guidelines for staying on course when you are forced to deal with these messy realities.

Finally, a discussion of MODULES will drive home the point that every design decision should be motivated by some insight into the domain. The ideas of high cohesion and low coupling, often thought of as technical metrics, can be applied to the concepts themselves. In a MODEL-DRIVEN DESIGN, MODULES are part of the model, and they should reflect concepts in the domain.

This chapter brings together all of these building blocks, which embody the model in software. These ideas are conventional, and the modeling and design biases that follow from them have been written about before. But framing them in this context will help developers create detailed components that will serve the priorities of domaindriven design when tackling the larger model and design issues. Also, a sense of the basic principles will help developers stay on course through the inevitable compromises.

Associations

The interaction between modeling and implementation is particularly tricky with the associations between objects.

For every traversable association in the model, there is a mechanism in the software with the same properties.

A model that shows an association between a customer and a sales representative corresponds to two things. On one hand, it abstracts a relationship developers deemed relevant between two real people. On the other hand, it corresponds to an object pointer between two Java objects, or an encapsulation of a database lookup, or some comparable implementation.

For example, a one-to-many association might be implemented as a collection in an instance variable. But the design is not necessarily so direct. There may be no collection; an accessor method may query a database to find the appropriate records and instantiate objects based on them. Both of these designs would reflect the same model. The design has to specify a particular traversal mechanism whose behavior is consistent with the association in the model.

In real life, there are lots of many-to-many associations, and a great number are naturally bidirectional. The same tends to be true of early forms of a model as we brainstorm and explore the domain. But these general associations complicate implementation and maintenance. Furthermore, they communicate very little about the nature of the relationship.

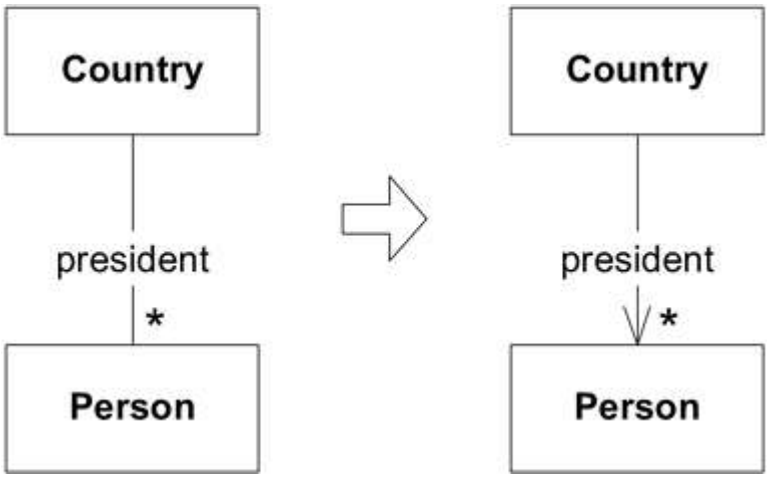
There are at least three ways of making associations more tractable.

1. Imposing a traversal direction
2. Adding a qualifier, effectively reducing multiplicity
3. Eliminating nonessential associations

It is important to constrain relationships as much as possible. A bidirectional association means that both objects can be understood only together. When application requirements do not call for traversal in both directions, adding a traversal direction reduces interdependence and simplifies the design. Understanding the domain may reveal a natural directional bias.

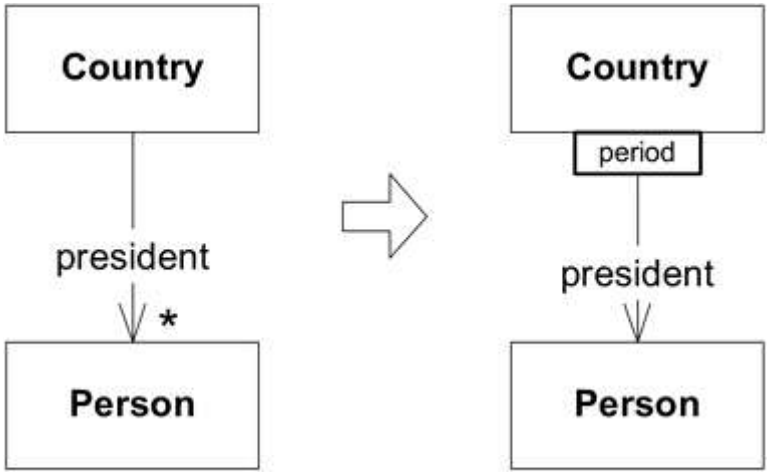
The United States has had many presidents, as have many other countries. This is a bidirectional, one-to-many relationship. Yet we seldom would start out with the name "George Washington" and ask, "Of which country was he president?" Pragmatically, we can reduce the relationship to a unidirectional association, traversable from country to president. This refinement actually reflects insight into the domain, as well as making a more practical design. It captures the understanding that one direction of the association is much more meaningful and important than the other. It keeps the "Person" class independent of the far less fundamental concept of "President."

Figure 5.1. Some traversal directions reflect a natural bias in the domain.



Very often, deeper understanding leads to a "qualified" relationship. Looking deeper into presidents, we realize that (except in a civil war, perhaps) a country has only one president at a time. This qualifier reduces the multiplicity to one-to-one, and explicitly embeds an important rule into the model. Who was president of the United States in 1790? George Washington.

Figure 5.2. Constrained associations communicate more knowledge and are more practical designs.



Constraining the traversal direction of a many-to-many association effectively reduces its implementation to one-to-many—a *much* easier design.

Consistently constraining associations in ways that reflect the bias of the domain not only makes those associations more communicative and simpler to implement, it also gives significance to the remaining bidirectional associations. When the bidirectionality of a relationship is a semantic characteristic of the domain, when it's needed for application functionality, the retention of both traversal directions conveys that.

Of course, the ultimate simplification is to eliminate an association altogether, if it is not essential to the job at hand or the fundamental meaning of the model objects.

Example
Associations in a Brokerage Account

据说他们起源于以色列12部落之一的利未部落，但是，从来没有任何传说提及该部落最初的定居地，也没有提到该部落在被征服的迦南分到了一席之地。他们虽然占据着最重要的祭司职位，但他们与其他祭司却有区别。利未并不是社会等级的名称，一个利未人并不必然就是一名祭司。我们关于摩西其人的估计可能为此提供一种解释。像摩西那样一个埃及的大人物要只身去接近一个陌生民族的人民是令人无法相信的，他肯定带着他的卫队、侍从、最坚定的拥护者、书记员、仆人等等。这些人就是最初的利未人。传说中一直把摩西当作一个利未人，这看来是对真实状况的明显歪曲。因为利未人是摩西的随从。我在前一篇论文中提到过，在后代犹太人中我们只在利未人的名 [46]
字中发现了埃及人的姓氏，这一点为上述解释提供了证据[45]。我们也许可以这样估计，摩西的随从中有相当一部分人逃脱了他和他的宗教所遭受的厄运，在其后的年月中，他们和自己混居其中的民族融合起来，但是却保持着对主人摩西的忠诚，缅怀着他的业绩，并且铭记着他的教诲。在耶和华的信徒们联合起来的时候，他们在其中形成了一支颇有影响的少数派，在文化水准上远远超过其他的人。

我设想，——至此为止仅仅是一种设想——在摩西身败名裂和夸底斯的宗教建立之间的时期，有整整两代人出生和消亡，甚至可能过了整整一个世纪。我无法断定，从埃及归来的那些新埃及人——我这样称呼他们以示同其他犹太人的区别——是在什么时候同他们的骨肉兄弟们重逢的，是在夸底斯的犹太人已经接受了信奉耶和华的宗教之前呢？还是在那之后？后一种可能性也许更大一些，但不管情况如何，那对于最终结果并无影响。在夸底斯达成的是某种妥协，摩西的部落在其中所占的

地位是明显的。

这里我们可以再注意一下关于割礼的风俗，这种"标准化石"(Leitfossil)已经反复起到了重要的作用。这种风俗也成了 [47]
耶和华宗教中的法律；并且，由于它与埃及的纠缠不清的联系，接受这种风俗就标志着对摩西的随从们的一种让步。他们——或他们当中的利未人——不会放弃自己视为神圣的标志，他们想尽力挽救自己的古老宗教，甚至不惜代价地宁愿承认一个新神，承认米底亚祭司们所说的关于这个新神的一切。也许他们还设法取得了其它让步。我已经提到过，犹太人的仪式在使用他们的神名时有着某种规定，凡说到耶和华处，必须以阿东赖(Adonai)来代替。我们禁不住要把这一戒律作为自己的论据，但这仅仅是一种臆测。禁止呼叫上帝的名字是一种原始禁忌，这是众所周知的。但是为什么恰好在犹太人的戒律中恢复这种禁忌，我们并不十分清楚。我们认为这种恢复是在某种新动机的影响之下发生的，这并非没有道理，我们没有理由假定那种戒律是一直被遵守的。耶和华这个名字被人们自由使用来给人定名，例如在约克兰(Jochanan)、杰休(Jehu)、乔舒亚(Joshua)等合成名字中。然而，这个名字还是有其独特之处。众所周知，《圣经》注释中承认，旧约全书中开始的六卷书(the Hexatecuh)有两个来源，这两个来源被称为J和E，因为一个用了神名耶和华(Jahve)，而另一个用了神名埃洛希恩[46](Elohim)；虽然用的神名是埃洛希恩而不是阿东赖，我们仍然可以在这里引用一位作者的话来说明："不同的名字是不同来源的神的明显标志。"[47]

我们承认过，严守割礼的风俗证明了夸底斯的新宗教建立时曾经达成过某种妥协，我们从J和E中都可以了解到该种妥协

的目的：这两种原因互相吻合，因此肯定能回溯到一个共同的 [48]
来源，即一种书面记载或一种口头传说。它们的主导目的在于证明新神耶和华的伟大和力量。既然摩西旧部的人认为他们从埃及逃亡的经历具有那样伟大的重要性，引导他们奔向自由的业绩当然应该归功于耶和华；这项业绩必须装饰得富于特色，以证明这位火山神令人害怕的威风，比如，那根在夜间变成火柱的烟柱，或者那股分开红海海水以便淹死追来的埃及人的风暴[20]等等。由此，逃出埃及和耶和华新宗教建立的时间被重合起来，两件事情之间的时间间隔被抹煞了。耶和华授予十大戒律的事据说也不是发生在夸底斯，而是在圣山脚下，在一处火山爆发的迹象之中。然而，这种描述却严重地亏待了摩西，因为是他带领他的人民逃出埃及奔向自由，而不是那位火山神。因此，摩西势必应该得到某种补偿，这种补偿的办法就是把摩西置换到了夸底斯或者西奈－霍内布山，并且把他放到一个米底亚祭司的地位上。我们后面将会认识到这种解决办法如何满足了另一种无法压制的紧迫需要。可以说，这种方法建立了下述这种平衡：耶和华的活动范围被允许从他在米底亚的圣山扩大到了埃及，而摩西的生活和活动却被转移到了夸底斯和约旦河东岸的国家。这就是他后来怎样变成另一个摩西，建立起一种宗教，成为米底亚祭司叶忒罗的女婿以及把自己的名字 [49]
摩西借用出去的经过。然而，我们却一点不知道这另一个摩西的个人情况，除了圣经中自相矛盾的描述之外，他完全被第一个摩西即埃及人摩西弄模糊了。他经常被描绘为性格专横，暴躁甚至凶猛，但是他也被说成是所有男人中最耐心最谦和的人。非常清楚，对于那个为他的人民谋划了那样巨大而困难的事业的埃及人摩西来说，上述后一种素质不会起任何作用，也

许这样的素质属于另一个摩西，即米底亚人摩西。将上述二者区分开来，假设埃及人摩西从没到过米底亚，也从没听到过耶和华的名字；而米底亚人摩西却从没有涉足过埃及，一点不知道阿顿，我认为是很有道理的。为了把这两者合而为一，传说和传奇都必须把埃及人摩西带到米底亚，我们已经看到，这一点已经有了不止一种解释。

六

我已作好准备再次遭受责难，人们指责我以很不恰当而且毫无道理的确信态度提出了关于以色列部落早期历史的构想；
[50] 我并不觉得这种批评过分严厉，因为我自己也有同感。我自己也知道这种构想有其薄弱环节，但是也有它的有力之处。总的说来，继续沿此构想进行探讨的有力之处更占优势。我们面前的《圣经》记载包含着很有价值——也很无价值——的历史证据。然而，这种记载却被有意施加的影响和苦心孤诣的诗歌创作歪曲了。我们在探讨中已经推测出一种这类歪曲倾向。这一发现将指引我们的道路，它是一种提示，使我们能由此发现其它类似的歪曲性影响。如果我们找到理由来辨认出这些影响造成的歪曲，我们就能使事态的演进更真实地反映出来。

让我们首先借助对《圣经》的批评性研究工作，弄清《圣经》前三卷书是怎样写成的，[21] 这三卷书指摩西五卷书和约书亚书。因为弄清这一点于我们是有利的。最早的来源据认为是J，即用耶和华为名称呼上帝的人，当代的研究者们认为他们能在三卷书作者当中鉴别出一个名叫埃布加塔(Ebjartar)的祭司，他是大卫王的一个同时代人[22]。过了不久，我们不清楚到底过了多久，三卷书的作者成了所谓的E，即用埃洛希恩为名称呼上帝

的人，他们属于北部王国[50]。公元前722年，北部王国毁灭之后，一个犹太祭司综合了J和E部分，并加进了他自己的部分，他编辑的书被定名为JE。在第七世纪中，第五部分书即申命记(Deuteronomy)被加进，据说整部书新近已在耶路撒冷的庙宇中发现。公元前586年，该庙宇被毁。在巴比伦之囚期间和获释重返耶路撒冷期间，整部书经过改写，被称为祭司法；在第五世纪又经过一次决定性的校正，此后就一直没作过材料上的变动[51]。

有关大卫王和其时代的历史记载极可能是他的某位同时代人所作的工作。它是"历史之父"希罗多德之前五百年的真实历史。如果人们按照我的假说来假设埃及人的影响，就能开始理解这一成就的意义。人们甚至设想过[52]摩西的书记员们对字母的发明作出过贡献[53]。我们当然无法知道，这些早期时代的记载到底有多少是以更早期的来源或口头传说为依据的，在事件与其文字记载形成之间到底经过了多长时间。然而，就我们现在所能发现的而言，那些记载却足以使我们看出有关的历史内容。两种完全相反的力量在那些文字记载上留下了各自的痕迹。一方面，某些改写人肯定会怀着隐密的动机篡改、肢解和夸大那些文字记载，至使它们面目全非。另一方面，一种宽容的虔诚态度占着统治地位，它急于维持现状而不管记载的细节是否互相吻合或抵触，在记载的每一处几乎都可以发现惊人的省略，扰人的重复，明显的矛盾以及一些人们决非有意传播的事情的痕迹。对文学记载的歪曲可以说无异于实施谋杀，其困难之处不在于抹煞事件本身而在于消除它的痕迹。虽然"歪曲"这个词不再具备双重含意，我却但愿能有权力来如此使用它。它不仅应该意味"改变某事情的表面"，而且应该意味使

[51]

其"发生剧烈变化"，"移置其内容"。这样，我们就可望在那些记载的许多失真之处，找出已经改头换面、支离破碎的材料，当然，要作到这一点并非易事。

我们想要识破的这类歪曲倾向肯定在那些传说被记载下来之前就已经对之产生了影响。其中的一种，也许是最严重的一种已经被我们发现。我说过，当夸底斯的新神耶和华被创立的时候，人们必须作些事情来给他增添光彩。更确切地说，他必须被树立起来，传诸久远，原来宗教的痕迹必须消除干净。那些定居部落似乎成功地做到了这一点，原来的宗教销声匿迹了。对那些从埃及归来的部落来说事情则不那么简单，他们决心捍卫从埃及出走的历史，捍卫摩西这个伟人和割礼风俗。确实，他们曾经身在埃及，但是他们又离开了那个国家，而且从现在起，埃及影响的每一点痕迹都必须予以否认。于是，摩西被移置到了米底亚和夸底斯，成了建立耶和华宗教的那个米底亚祭司。由于必须保留割礼这种最容易同埃及发生牵连的风俗，因此尽管存在着种种证据，人们还是尽一切努力来消除这种风俗同埃及的联系。在《出埃及记》中，有一神秘的段落用令人无法思议的风格写道，摩西忽略了施行割礼，上帝对他大发雷霆，他的米底亚妻子为了挽救他的生命，匆忙为他作了割礼手术。这一段落只可能解释为是对那件重要事实精心设置的矛盾。我们很快将会看到为了消除一件不利证据而有意虚构的其它故事。

[53]

我们发现，那些记载不遗余力地肯定耶和华不是一个外来神，而是犹太人的旧神，这很难说是一种新的倾向，只不过是上述歪曲企图的继续。为此目的，有关亚伯拉罕、以撒[54]和雅各[55]等族长的神话被炮制出来。耶和华坚持说他是这些族长们

[54]

的上帝；但他自己也必须承认，他们并没有在耶和华这个名字之下来崇拜他[59]。

耶和华也没有说清他曾经在其他什么名字之下受崇拜。这一机会被利用来对割礼风俗的埃及来源施予了决定性的一击。据说耶和华要求过亚伯拉罕，把割礼立为他和亚伯拉罕的子孙之间关系的标志。然而，这是一个特别笨拙的发明，如果谁想用一个标志来使某人有别于其他人，他肯定会选择其他人所不具备的特征，那当然不会是成千上万的人都能显露的东西。受着那种关系的束缚，以色列人就必须把所有的埃及人都认作自己的同胞。撰立《圣经》文字的以色列人不可能不知道割礼是埃及人的本地风俗这一事实。爱德华·迈耶所引的《约书亚书》中的有关段落坦率地承认这一点，但是，这一事实却必须不惜一切代价予以否认。

我们不能指望宗教神话会小心翼翼地注意逻辑联系。要不然以色列人就有理由抗议那位曾经和他们的祖先达成过契约的神的行为。那项契约包含着共同的义务，可是耶和华却在数百年间漠视了他的合伙人，然后才突然想到要对他们的子孙显圣。一个神突然"选定"一个民族，让它作为"他的"人民，使 〔55〕
自己成为它的上帝，这种设想更令人惊奇。我相信这在人类宗教史上是唯一的一次例外。在其它情况下，一个民族同它的上帝是不可分割的，他们从历史伊始就同属一体。当然，我们有时所说某个民族接受了另一位上帝，但却从未听说过某位上帝选定了某个新的民族。当我们回顾摩西与犹太民族之间的联系时，也许能理解这种唯一的事件。摩西屈尊降驾来到犹太人中间，把他们作为他的臣民；他们就是他的"选民"[60]。

然而，把那些先辈族长带入新的耶和华宗教还有另一个 〔56〕

目的。他们曾经居住在迦南；他们的记忆与这个国家的某些区域性有联系。他们自身可能就曾是迦南人的英雄或本地神祇。迁徙中的以色列人把他们纳入了自己的早期历史，追忆起他们，人们就可以证明自己诞生和生活在这个国家，就可以否认掉依附于外来征服者的非难。这不愧是一个聪明的改变：上帝耶和华给予他们的只是他们的祖先曾经拥有的东西。

在《圣经》的后期记载中，避免提及夸底斯的倾向获得了成功，建立耶和华新宗教的地点明确无误地成了西奈-霍内布的圣山，这种动机是非常明显的，也许他们不愿回忆起米底亚的影响。但是，所有后期的歪曲，特别是对祭司法的那些歪曲都服务于另一目的。现在再也没有必要在某一特别的方面去改变久远的事件；因为那早已经完成了。另一方面，人们竭力把现在的某些法律和惯例确立的时间推回到早期，使摩西律法成为它们照例的依据，由此使它们获得神圣性和约束力。无论这幅过去时代的图画被窜改得多么面目全非，其程序却仍然包含着某种心理学的理由。它反映出下述事实，即在许多世纪的过程 〔57〕
中——在出埃及记和以斯拉与尼希米编定的《圣经》之间，大约800年时光过去了——耶和华的宗教经历了一种逆向发展，这种发展与摩西的原始宗教融合而达到高潮。（也许达到了真实同一性的顶峰。）

这就是最本质的结果：犹太人宗教历史的决定性的内容。

七

在后世的诗人、祭司和历史学家们着意描绘的犹太人史前史的所有事件中，有一件事情最为突出，人们煞费苦心地竭力掩盖这件事，这就是伟大的领袖和解放者摩西所遭受的谋杀。

塞林的这种假设不能被斥之为异想天开；它是完全可能的事情。在埃赫那顿的祭司学校里受过训练的摩西使用了和那位国王一样的方法：他发布命令，把他的宗教强加到他的臣民头上[58]。摩西的信条也许比他的老师的信条更为坚定；他没有必要同太阳神教保持任何联系，因为古老北方的祭司学校对于他的外族臣民来说没有任何重要性。摩西遭遇了和埃赫那顿同样的命运，所有开明的专制君主都面临着那种命运。摩西的犹太族臣民同第十八王朝的埃及人一样，无法忍受那样一种高度精神化了的宗教，无法从其中为他们的需要寻求满足。在这两种情况下发 [58]
生了同样的事情：那些觉得自己受着训戒或感到无依无靠的人起而造反，扔掉了强加在自己身上的宗教负担。但是，当温顺的埃及人等待着命运除去他们神圣的法老时，野蛮的闪米特人却把命运操在自己手中，除掉了他们的暴君[59]。

我们也无法从现存的《圣经》中肯定摩西没有遭受那样的结局。"在荒野里漫游"的记载——它可能代表着摩西统治的时代——描述了一系列反抗他的权威的严重背叛，由于耶和华的命令，那些背叛被用野蛮的责罚镇压下去。我们很容易设想那些背叛中的一次走到了《圣经》记载所能承认的另一极端。虽然只是一段插曲，《圣经》中还是提到了犹太人抛弃了摩西的新宗教。那就是关于那只金牛犊(the golden Calf)的故事。故事中巧妙地将打坏两块写着诫命的石板的责任——这必须象征性地加以理解(＝"他已经破坏了那条戒律")——归咎于摩西本人，归罪于他愤怒时的义愤[60]。

后来，那些犹太人后悔谋杀了摩西，并且努力想忘掉那件事。在夸底斯达成联合时情形肯定是如此。然而，如果人 [59]
们把逃出埃及的时间安排在离沙漠绿洲中建立宗教的时间更

近一些，并且让摩西而不是其他的创立者来建立宗教，那么，犹太人不仅会对摩西的主张感到满意，而且也会成功地否认掉他被野蛮杀害的痛苦事实。实际上，即使摩西没有过早地被杀害，他也极不可能参与过夸底斯发生的事件。

这里我们必须尽量说明这些事件的顺序。我已将从埃及出逃的时间定在第十八王朝覆灭之后(公元前1350年)。实际上它可能发生在其时或者稍后，因为埃及的编年史家把随后在荷伦希布统治下的混乱年月包括在十八朝中，荷伦希布结束了第十八朝，一直统治到公元前1315年。确定这一编年史的后一个证据——也是唯一的证据——是美楞普塔[61](1225—1215, BC)石碑，它颂扬了对以色列人的胜利和以色列人子孙的毁灭。不幸的是，那块石碑的价值值得怀疑，它被当作以色列部落在当时已在迦南定居的证据[62]。就如人们以前要假定的一样，迈耶正确地从那块石碑得出结论说，美楞普塔不可能是犹太人出逃时埃及的法老，逃出埃及肯定是在更早的时期。就我看来，要深究出埃及当时的法老是谁毫无益处，那段时期没有法老，因为犹太人逃出埃及的事发生在埃及没有正常统治者的空位期。但是美楞普塔石碑并没有说清夸底斯联合和接受新宗教的可 [60]
能时期。所有我们能有把握说定的就是，它们发生在公元前1350年至1215年之间的某段时间。在这一世纪中，让我们假定逃出埃及的时间挨近公元前1350年，夸底斯发生事件的时间离公元前1215年不远，那么这两个事件之间相隔的时期就会更长些，回归的部落要从杀害摩西的狂热中清醒过来，摩西的旧部利未人要恢复起能在夸底斯的妥协中发挥的影响，这样一段长时期都是必要的。两代人的时间，60年，可能足够了，但是仅仅足够而已。从美楞普塔石碑上推知的时间太早了一点，正如

我们所知，在我们的假设中一环扣着一环，我们必须承认这种讨论显出了整个结构中的薄弱点。不幸的是，和犹太人在迦南定居相关的每一件事都非常模糊而且混乱，当然，我们也许可以暂且估计石碑上的以色列这个名字并不是指的我们正在探究的那些后来才联合进以色列民族的部落，可是，无论如何，从阿马尔那时代起，哈比鲁(即希伯来)这个名字就已是指的这个民族了。

当不同的部落接受同一种宗教而联合起来时，对于世界历史很可能并不会发生重大的影响。那种新宗教很可能已 [61]
经被一系列事件卷走，耶和华也会在福楼拜[60]所想象的往昔神祇的行列中占据一席之地，至于他的人民，不仅盎格鲁·撒克逊人长久以来追寻的那拾个部落不见踪影，所有的12个部落可能都已经"消失"[61]。上帝耶和华可能根本就不是一个值得注意的神，而是一个粗暴、狭隘、凶猛而且嗜血成性的本地神，他许诺他的信奉者，要给他们"一块流着牛奶与蜂蜜的土地"；他鼓励他们"用剑的锋刃"去除掉这块土地上现在居住着的人。令人惊奇的是，尽管《圣经》记载经过了反复的改写，却仍然有那么多使我们足以认清他的天性的内容被允许保存下来。他的宗教是否是真正的一神教，它是否否认其它的神具有上帝的性格，这还值得怀疑。也许，它只要自己的神比其他所有陌生的神更强大就足够了。当事件发生的顺序沿着另一个方向转变而没有产生像这类开端引导我们去预期的结果时，就只可能有一个理由来解释这种现象。埃及人摩西给犹太族的部分人带去了关于上帝的另一个更崇高的概念，这是一个拥抱全世界的唯一的神，因为他是全能的，所以他博爱一切人，他厌恶所有的仪式和巫术，把人道作为一种充满真理和正义的生活的最高目标。不管

我们关于阿顿神教的伦理学方面的资料多不完整，埃赫那顿在他的题铭里经常称自己"生活在真理和正义中"，[62]却肯定是
[62] 富有意义的。在一段短时期后，犹太人可能就抛弃了摩西的教诲并且杀害了他，可是从长远看来，这却并无多大关系。摩西的传统本身存留下来了，在长达数世纪的过程中，它的影响虽然扩展得缓慢，但是却达到了摩西本人所未达到的目标。从夸底斯开始，上帝耶和华就获得了非分的荣誉，摩西为犹太人谋求解放的业绩被记到了他的帐上；但是他不得不为这种僭取付出沉重代价。在历史发展的终点，在他之外出现了已被忘却的摩西神(Mosaic god)，这个被他僭位的神的庇护力渐渐胜过了他。谁也无法怀疑，就是由于对摩西神的希望，才使以色列的人民克服了所有的艰难而生存到我们的时代。

我们再也无法确定，在摩西神对耶和华的最后胜利中，利未人到底起了多大的作用。在夸底斯达成妥协的时候，他们曾经为摩西呐喊，他们是摩西的信徒和同胞，对自己的主人仍然记忆犹新。从那以后的数世纪中，利未人已经和犹太民族融为一体，成为了祭司阶层；而祭司的主要任务，就是举行和监督典礼仪式，关心《圣经》内容，按照自己的目的对之加以改写。但是，难道他们这些祭祀和仪式从本质上说不是魔法和巫术，就像摩西的古老信条无条件地加以责难的那些东西一样吗？从
[63] 这个民族中间，不断地产生出一批批新人，这些人不一定都是摩西旧部传下的子孙，但他们却被这个逐渐在黑暗中发展起来的伟大有力的传统所吸引，就是这些人，这些先知，小心翼翼地宣讲了古老的摩西信条：神摈斥祭祀和仪式，他只要求人们信仰他，过一种充满真理和正义的生活(Maat)。那些先知们的努力获得了不朽的成功；他们藉以重建起古老信仰的

那些信条成了犹太教亘古不变的内容。对于犹太民族来说，这足以称得上一种荣誉。即使那种激励首先是来自外界，来自一个伟大的陌生人，犹太民族还是一直保持了这样一个传统，并且养育出了为之献身的人。

如果我没有参考其他人的判断，对于上述事件的描述可能会给我留下一种不踏实的感觉。老练的研究工作者们虽然没有认识到摩西的埃及根源，但他们却以同样的眼光看待摩西在犹太宗教史上的重要性。例如，塞林⑯说："因此，从现在起，我们必须把真正的摩西宗教以及他所宣扬的一个伦理神的信仰想象为事实上是在该民族中拥有一个小集团。我们无法指望从一开始就在官方的崇拜，祭司的宗教和人民的普遍信仰中找到它。所有我们能够指望的就是从他点燃的精神火花中升起的点点星火，他的思想还没有消失，而是一直在静静地影响着信仰和习俗，直到某个时候，在某些特殊事件的影响下，或是由 [64]
于某些特别醉心于这种信仰的个人的影响下，它的力量又更强烈的爆发出来，在广大的人民中取得了统治地位。就是从这种观点出发，我们必须重视古代犹太人的早期宗教史。如果我们按照历史文献中描述的最初5世纪时迦南的宗教模式来设想摩西宗教，我们就会陷入最严重的方法上的错误。"沃尔兹(Volz)更清楚地表明了他的见解，他说："摩西的天马行空的学说最初几乎没有被人理解而且难于推行，在数世纪的过程中，它才慢慢地渗入到人民的精神中；最后，它在伟大的先知们中寻到了知音，那些先知们继承了这位孤独的创业者的学说。"⑰

至此，我的文章应该告一结束。我唯一的目的就是使埃及人摩西的肖像适合于犹太人历史的镜框。现在我可以用最简捷的方式来表述我的结论：犹太民族的历史具有众所周知的二重

性(duality)——两个民族融合而为一个民族，这个民族又分裂而为两个王国，依据《圣经》，他们的神有两个名称——我们现在要在这种二重性上加入两项新的二重性：在那两种新宗教的建立中，第一种被第二种取代了，但又重新出现并获得胜利；该宗教的两个创立者，被称为同一个名字：摩西，而我们又必须把他的个性一分为二。所有这些二重性都是第一种二重性 [65]
的必然结果：这个民族的一部分人经历了一种可以恰当地称之为创伤体验的阶段，其他的人则免除了这种体验。我们此外还有许多东西要讨论、解释和判断，只有那样，我们的纯历史研究的兴趣才能得到充分保证。一种传说的内在实质到底是什么？它的特殊的力量存在于哪里？人们怎么不可能否认伟大人物在世界历史上的个人影响？如果我们把那些从物质需要中萌发出来的东西认为是唯一的目的，我们会怎样亵渎了人类壮丽而多彩多姿的生活？某些思想，特别是宗教思想，从什么源泉获得压抑众多的个人和民族的力量？在犹太人历史这种特殊的情况中来研究所有这些问题将是一个诱人的任务。我的论文的这个续篇将把我25年前在《图腾和禁忌》中贡献的结论联成一体。但是，我很难相信自己还有更进一步的力量。

第三章　摩西，他的人民和一神教

[66]

引言性说明

一、写于 1938 年 3 月之前(维也纳)

怀着一个无牵无挂无所失的人的胆量，我打算再一次打破自己久已下定的决心，用至今还保留在手的这最后一个部分来接续我关于摩西的两篇论文(载《意向》杂志合订本 23 卷，第一、三章)。当我完成后一篇论文时，我曾说过，我充分了解自己的力量不足以胜任那次任务。当然，我指的是伴随老年而来的创造力的衰退①，但也还有一道其它障碍。我们生活在非常令人注目的时代，我们惊奇地发现，进步与野蛮结成了同盟。苏[67]维埃俄国已经作出努力来改善至今还受着压迫的成千万人民的生活，当局非常勇敢地取缔了他们的宗教镇痛剂，并且非常明智地允许了他们享有相当合理的性自由，但与此同时，当局却使他们服从于最残酷的高压政治并掠夺了他们的思想自由的每一种可能性。意大利人民受着同样野蛮的统治，正在被灌输着纪律和责任感。在德国人民现在的状况下，我们如释重负地发现，进步思想居然还能在各方面向史前的野蛮状态全面退化的情况下产生。即使如此，各种事件还是在自然发展，保守的民主党人已经成为文化进步的保护者；更为奇怪的是，正是天主

教会对破坏文化的行径进行了顽强的抵抗，而天主教会至今一直是所有自由思想不共戴天的敌人，它一直顽固地反对让任何向往真理的进步思想来左右这个世界。

我们现在生活在一个在天主教会保护之下的天主教国家，而且尚不清楚这种保护还要持续多久。既然如此，我当然就要踌躇，是否应该去作注定要唤起对天主教会的敌意的事情。这不是怯懦，而是谨慎；那个新的敌人② 比这个宿敌更危险，我将提高警惕，不作符合他的利益的任何事情，我们已经学会同[68]他和平共处。精神分析学的研究在任何情况下都会受到天主教组织的怀疑和注意。我没有坚持说这种怀疑是不恰当的。如果我们的研究使我们得出某种结果，将宗教贬低到人类神经症的地位；如果我们像看待个体病人的神经性强迫症那样来解释宗教的巨大力量，那么可以确信，我们在这个国家将招致这些势力的极大仇恨。我现在并没有什么新内容要说，四分之一世纪之前，我已将所有的东西表达得清清楚楚。但是，所有那些都已被忘却，如果我现在旧话重提，并且用典型例子来阐明宗教建立的方式，肯定将会产生某些效果。这可能会使我们从事的精神分析学的工作遭到禁止。这种野蛮的镇压手段对于天主教会来说并不陌生；虽然当其他人要诉诸同样的手段时，她会觉得是对她的特权的冒犯。在我漫长的一生中，精神分析学已经四通八达，但是在它诞生和成长的这个城市里，它却比在其它任何地方都更能发挥作用。

我不仅仅这样认为，而且清楚这种外部危险将阻挠我发表关于摩西的论文的最后部分。我一直告诫自己，这种恐惧是因为过高估计了自己的重要性，当局对我要说的关于摩西和一神教[69]起源的话可能是很无所谓的。我力图以此消除这种恐惧。然

而，我不清楚自己的判断是否正确。就我看来，在世人的眼中，除了上述原因之外，更大可能是将我看作一个用心险恶并喜欢危言耸听的人，因此我不会发表这篇论文，但是那不会阻碍我撰写它；尤其是两年之前我已写过一遍，因此现在只需重抄并把它加到前两篇论文中去。它将偷偷地等待，直到可以安全地见到天日，或者，直到有人产生了与此相同的见解并作出了同样的结论，那时他将会听说："在那些更黑暗的日子里，曾经生活着一个像你一样思想的人。"

二、1938年6月(伦敦)

在我写作这篇探讨摩西的论文期间，那些压在我身上的异乎寻常的巨大压力——内心的疑惧和外部的阻碍——是这第三即最后部分为什么会有两篇自相矛盾甚至互相抵消的序言的原因，因为在撰写这两篇序言的短暂期间，我的外部环境发生了急剧的改变。我原来居住在天主教会的保护之下，惟恐发表这篇论文之后会失去那种保护，也害怕在奥地利从事精神分析学的实习医师和学生们失去工作。其后，德国对我们发动突然袭击，而天主教组织就像《圣经》上所说的那样，只不过是"软弱得不能依靠的东西。"由于确信将会受到迫害——现在不仅因为我的工作，而且因为我的"种族"——我和许多朋友一道，离开 (70)
了从孩提时代一直生活到78岁的城市。

我在美丽、自由、慷慨的英国受到了最亲切的欢迎。我这个从压迫中解脱出来的人，作为一个受欢迎的客人居住在这里。我欣幸自己又能按自己的意愿来说话和写作，甚至"思想"了。我现在敢于将这篇论文的最后部分公诸于世了。

再也没有外部阻碍了，至少再也没有什么使我惊愕的东西。

在我来到后短短几个星期里，我已经收到大批的问候信，朋友们告诉我说，他们在这里看到我觉得非常高兴，陌生的、对我的工作很少兴趣的人们给我来信，为我在这里找到了自由和安全而感到愉快。除此之外还有另一类信件，常常使我这个外国人不知所措，它们表达了对我内心幸福的关怀，殷切地为我指引向往基督的道路，并且鼓励我相信以色列的未来。这些给我来信的善良的人们看来还不太了解我。然而我估计，当我的这一新作在我的新同胞中传播的时候，我将失去我的来信者，也将失去他们现在对我表示的同情。

内心的压力不会被不同的政治制度和新的定居处改变。在面对我自己的工作时，我仍然难以愉快；我怀念那种整体意识和应该存在于作者和他的工作之间的亲密感。这并不意味着我缺乏说服力来说明我所作出的结论的正确性。我在四分之一世纪 (71)
之前(1912年)写成《图腾与禁忌》的时候就已经获得了这种说服力，从那以后，这种说服力变得越来越强。我从来没有怀疑过，宗教现象只能在个体的神经病症状的模式中来理解，宗教现象作为人类大家庭原始时期发生的久已被人类遗忘的那些事件的再现形式，是我们十分熟悉的。由于这种特殊根源，宗教现象获得了其强迫性特征，并且由于它们包含了历史的真理而对人类产生着影响。只有当我试问自己，我是否已经从犹太一神教这个例子中成功地证明了这一点时，我才开始感到怀疑。对于我的持批评态度的同事们来说，这篇从研究摩西开始的论文似乎难于立论。如果我不能在对有关弃婴的神话的分析说明中发现论据，不能由此证明塞林关于摩西之死的设想的正确性，这篇论文便无法写成。然而，我还是要写下去。

我从概括我的第二篇纯粹研究摩西历史的论文开始来撰写

这篇论文。在这里我将不对那篇论文的结果作任何批评，因为它们构成了下述心理学讨论的前提，这种讨论以它们为基础，而又不断地反过来影响它们。

第 一 节 〔72〕

1.历史的前提

以下是激起我们兴趣的那些事件的历史背景。在第十八朝时期，埃及通过征服，已经成为一个世界性大国。其新的帝国主义实质在某些宗教思想的发展中反映出来；这些思想即便没有存在于它的全体人民之中，也存在于它的统治阶层和能起作用的上等知识阶层之中。在古老北方(赫里阿波利斯)太阳神祭师们的影响下，可能也在来自亚洲的影响的推动下，出现了一个宇宙神阿顿的思想；这个神不再局限于一个民族、一个国家。年轻的法老阿蒙霍特普四世接掌权柄之后，对发展这样一个神的思想表现出超乎一切的热情。他把阿顿神教抬高到了国教的地位。因此，这个宇宙神也就成了唯一的上帝；所有有关其他神的说法都成了欺骗和罪恶。他抱着誓不两立的态度，抵制巫术思想的一切诱惑，摒弃了埃及人特别珍视的关于死后生活 〔73〕的幻想；他惊人地预见到后世的科学知识，认为太阳辐射的能量是地球上所有生命的源泉，并且把太阳作为上帝力量的象征来崇拜。他满腔喜悦地赞美上帝创造天地，为他自己生活在真理和正义中而自豪。

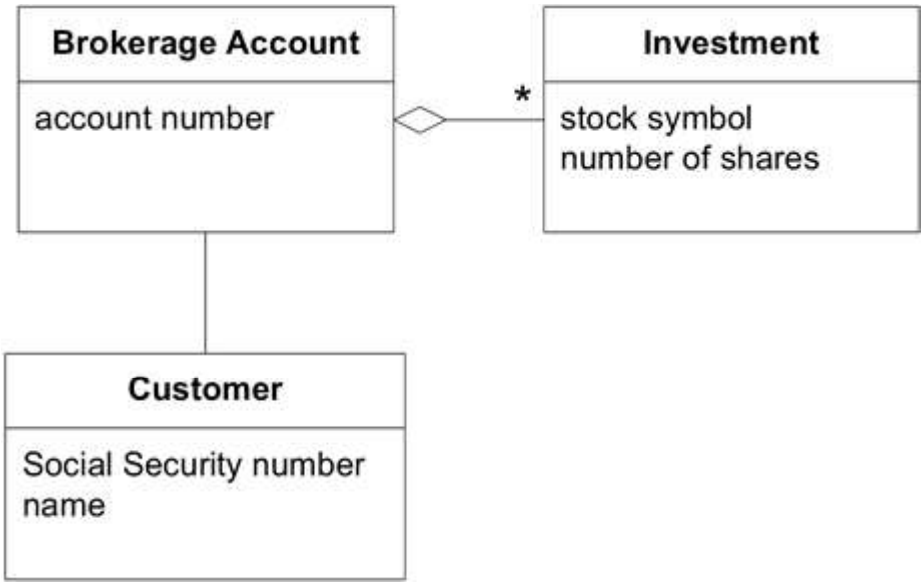
在人类历史上，这是一神教宗教的第一次出现，而且也许是最纯正的一次。对它发源的历史条件和心理学条件作更深入的了解将具有无可估量的价值。然而，我们应该明白，有关阿

顿神教的资料不会有多少流传下来。在埃赫那顿的懦弱的继承者们的统治时期，他所创立的一切都已经土崩瓦解。曾经被他镇压的祭师集团现在发泄出了他们的怨恨，阿顿神教被废止了；这个持异端邪说的法老的首都遭到破坏和劫掠。公元前1350年，第十八朝寿终正寝；经过一段混乱时期之后，[3]荷伦希布将军恢复了秩序并统治到公元前 1315 年。埃赫那顿的改革看来只不过是一段注定要被遗忘的插曲。

这就是已经建立起来的历史背景，我们的假设由此开始。在埃赫那顿的亲近者中，可能有一个名叫图特摩斯(Tho thmes)的人，就像当时有许多人叫这个名字一样④，这个名字本身没有什么关系，但是它的第二部份肯定是"摩西"(-mose)。他身居高位，是阿顿神教的一个忠实信徒，并且，他坚强有力而且充 〔74〕满热情，与他那位优柔寡断的国王恰成对比。对这个人而言，埃赫那顿的死和阿顿神教的倾覆意味着所有希望的破灭。他留在埃及就只有改变信仰或遭受排斥。如果他当时是边境省的总督，他很可能已经和几代人之前就移居进来的某些闪米特人部落建立了联系，他在绝望和孤独的处境中转向了这些外族人，在他们中间寻求补偿。他选择了这些人作为自己的臣民，力图通过他们来实现自己的理想。他由自己最亲信的随从陪同，带着这些人离开了埃及，然后使他们接受了割礼风俗，为他们建立了法律，并且使他们皈依了刚被埃及人抛弃的阿顿神教。也许摩西强加给这些犹太人的规矩比他的主人和老师埃赫那顿那些规矩还要严厉；但他也可能放弃了埃赫那顿坚持不渝的同古老北方的太阳神的联系。

我们必须把出埃及的时间确定在公元前 1350 年之后那段空位期。此后一直到占领迦南那段时期的情况特别模糊。在《圣

Figure 5.3.



One Java implementation of **Brokerage Account** in this model would be

```
public class BrokerageAccount {
    String accountNumber;
    Customer customer;
    Set investments;
  // Constructors, etc. omitted

  public Customer getCustomer() {
    return customer;
  }
  public Set getInvestments() {
    return investments;
  }
}
```

But if we need to fetch the data from a relational database, another implementation, equally consistent with the model, would be the following:

Table: BROKERAGE_ACCOUNT

ACCOUNT_NUMBER	CUSTOMER_SS_NUMBER

Table: CUSTOMER

SS_NUMBER	NAME

Table: INVESTMENT

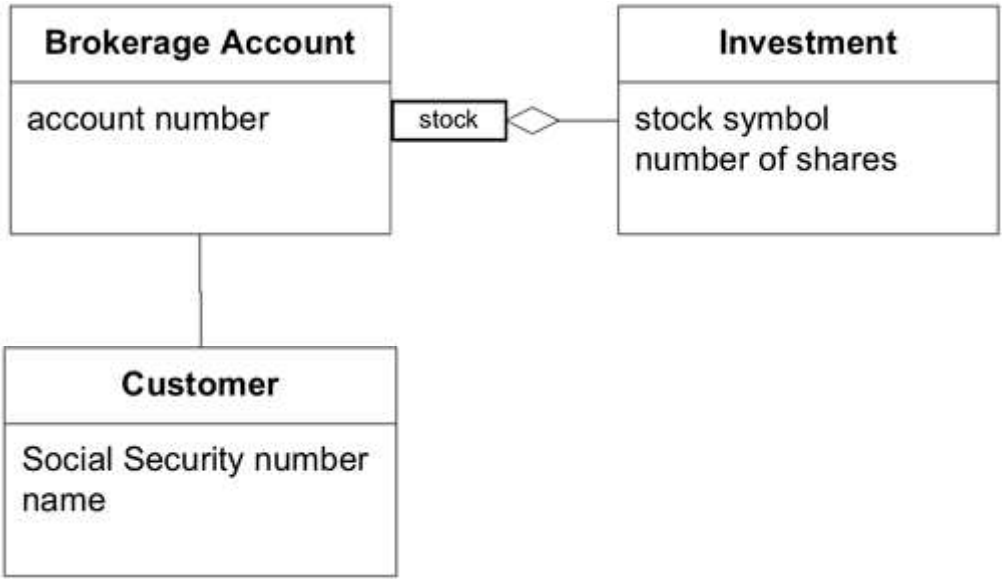
ACCOUNT_NUMBER	STOCK_SYMBOL	AMOUNT

```
public class BrokerageAccount {
  String accountNumber;
  String customerSocialSecurityNumber;

  // Omit constructors, etc.

  public Customer getCustomer() {
    String sqlQuery =
      "SELECT * FROM CUSTOMER WHERE" +
      "SS_NUMBER='"+customerSocialSecurityNumber+"'";
    return QueryService.findSingleCustomerFor(sqlQuery);
  }
  public Set getInvestments() {
    String sqlQuery =
      "SELECT * FROM INVESTMENT WHERE" +
      "BROKERAGE_ACCOUNT='"+accountNumber+"'";
    return QueryService.findInvestmentsFor(sqlQuery);
  }
}
```

(*Note:* The QueryService, a utility for fetching rows from the database and creating objects, is simple for explaining examples, but it's not necessarily a good design for a real project.)

Let's refine the model by qualifying the association between **Brokerage Account** and **Investment**, reducing its multiplicity. This says there can be only one investment per stock.

Figure 5.4.



This wouldn't be true of all business situations (for example, if the lots need to be tracked), but whatever the particular rules, as constraints on associations are discovered they should be included in the model and implementation. They make the model more precise and the implementation easier to maintain.

The Java implementation could become:

```
public class BrokerageAccount {
  String accountNumber;
  Customer customer;
  Map investments;

  // Omitting constructors, etc.

  public Customer getCustomer() {
    return customer;
  }
  public Investment getInvestment(String stockSymbol) {
    return (Investment)investments.get(stockSymbol);
  }
}
```

And an SQL-based implementation would be:

```
public class BrokerageAccount {
  String accountNumber;
  String customerSocialSecurityNumber;

  //Omitting constructors, etc.
  public Customer getCustomer() {
    String sqlQuery = "SELECT * FROM CUSTOMER WHERE SS_NUMBER='"
      + customerSocialSecurityNumber + "'";
    return QueryService.findSingleCustomerFor(sqlQuery);
  }
  public Investment getInvestment(String stockSymbol) {
```

```
    String sqlQuery = "SELECT * FROM INVESTMENT "
      + "WHERE BROKERAGE_ACCOUNT='" + accountNumber + "'"
      + "AND STOCK_SYMBOL='" + stockSymbol +"'";
    return QueryService.findInvestmentFor(sqlQuery);

  }
}
```

Carefully distilling and constraining the model's associations will take you a long way toward a MODEL-DRIVEN DESIGN. Now let's turn to the objects themselves. Certain distinctions clarify the model while making for a more practical implementation. . . .

PREVIOUS NEXT

Entities (a.k.a. Reference Objects)

Many objects are not fundamentally defined by their attributes, but rather by a thread of continuity and identity.

❊ ❊ ❊

A landlady sued me, claiming major damages to her property. The papers I was served described an apartment with holes in the walls, stains on the carpet, and a noxious liquid in the sink that gave off caustic fumes that had made the kitchen wallpaper peel. The court documents named me as the tenant responsible for the damages, identifying me by name and by my then-current address. This was confusing to me, because I had never even visited that ruined place.

After a moment, I realized that it must be a case of mistaken identity. I called the plaintiff and told her this, but she didn't believe me. The former tenant had been eluding her for months. How could I prove that I was not the same person who had cost her so much money? I was the only Eric Evans in the phone book.

Well, the phone book turned out to be my salvation. Because I had been living in the same apartment for two years, I asked her if she still had the previous year's book. After she found it and verified that my listing was the same (right next to my namesake's listing), she realized that I was not the person she wanted to sue, apologized, and promised to drop the case.

Computers are not that resourceful. A case of mistaken identity in a software system leads to data corruption and program errors.

There are special technical challenges here, which I'll discuss in a bit, but first let's look at the fundamental issue: Many things are defined by their identity, and not by any attribute. In our typical conception, a person (to continue with the nontechnical example) has an identity that stretches from birth to death and even beyond. That person's physical attributes transform and ultimately disappear. The name may change. Financial relationships come and go. There is not a single attribute of a person that cannot change; yet the identity persists. Am I the same person I was at age five? This kind of metaphysical question is important in the search for effective domain models. Slightly rephrased: Does the user of the application *care* if I am the same person I was at age five?

In a software system for tracking accounts due, that modest "customer" object may have a more colorful side. It accumulates status by prompt payment or is turned over to a bill-collection agency for failure to pay. It may lead a double life in another system altogether when the sales force extracts customer data into its contact management software. In any case, it is unceremoniously squashed flat to be stored in a database table. When new business stops flowing from that source, the customer object will be retired to an archive, a shadow of its former self.

Each of these forms of the customer is a different implementation based on a different programming language and technology. But when a phone call comes in with an order, it is important to know: Is this the customer who has the delinquent account? Is this the customer that Jack (a particular sales representative) has been working with for weeks? Is this a completely new customer?

A conceptual identity has to be matched between multiple implementations of the objects, its stored forms, and real-world actors such as the phone caller. Attributes may not match. A sales representative may have entered an address update into the contact software, which is just being propagated to accounts due. Two customer contacts may have the same name. In distributed software, multiple users could be entering data from different sources, causing update transactions to propagate through the system to be reconciled in different databases asynchronously.

Object modeling tends to lead us to focus on the attributes of an object, but the fundamental concept of an ENTITY is an abstract continuity threading through a life cycle and even passing through multiple forms.

Some objects are not defined primarily by their attributes. They represent a thread of identity that runs through time and often across distinct representations. Sometimes such an object must be matched with another object even though attributes differ. An object must be distinguished from other objects even though they might have the same attributes. Mistaken identity can lead to data corruption.

An object defined primarily by its identity is called an ENTITY.[1] ENTITIES have special modeling and design considerations. They have life cycles that can radically change their form and content, but a thread of continuity must be maintained. Their identities must be defined so that they can be effectively tracked. Their class definitions, responsibilities, attributes, and associations should revolve around who they are, rather than the particular attributes they carry. Even for ENTITIES that don't transform so radically or have such complicated life cycles, placing them in the semantic category leads to more lucid models and more robust implementations.

> [1] A model ENTITY is not the same thing as a Java "entity bean." Entity beans were meant as a framework for implementing ENTITIES, more or less, but it hasn't worked out that way. Most ENTITIES are implemented as ordinary objects. Regardless of how they are implemented, ENTITIES are a fundamental distinction in a domain model.

Of course, most "ENTITIES" in a software system are not people or entities in the usual sense of the word. An ENTITY is anything that has continuity through a life cycle and distinctions independent of attributes that are important to the application's user. It could be a person, a city, a car, a lottery ticket, or a bank transaction.

On the other hand, not all objects in the model are ENTITIES, with meaningful identities. This issue is confused by the fact that object-oriented languages build "identity" operations into every object (for example, the "==" operator in Java). These operations determine if two references point to the same object by comparing their location in memory or by some other mechanism. In this sense, every object instance has identity. In the domain of, say, creating a Java runtime environment or a technical framework for caching remote objects locally, every object instance may indeed be an ENTITY. But this identity mechanism means very little in other application domains. Identity is a subtle and meaningful attribute of ENTITIES, which can't be turned over to the automatic features of the language.

Consider transactions in a banking application. Two deposits of the same amount to the same account on the same day are still distinct transactions, so they have identity and are ENTITIES. On the other hand, the amount attributes of those two transactions are probably instances of some money object. These values have no identity, since there is no usefulness in distinguishing them. In fact, two objects can have the same identity without having the same attributes or even, necessarily, being of the same class. When the bank customer is reconciling the transactions of the bank statement with the transactions of the check registry, the task is, specifically, to match transactions that have the same identity, even though they were recorded by different people on different dates (the bank clearing date being later than the date on the check). The purpose of the check number is to serve as a unique identifier for this purpose, whether the problem is being handled by a computer program or by hand. Deposits and cash withdrawals, which don't have an identifying number, can be trickier, but the same principle applies: each transaction is an ENTITY, which appears in at least two forms.

It is common for identity to be significant outside a particular software system, as is the case with the banking transactions and the apartment tenants. But sometimes the identity is important only in the context of the system, such as the identity of a computer process.

Therefore:

When an object is distinguished by its identity, rather than its attributes, make this primary to its definition in the model. Keep the class definition simple and focused on life cycle continuity and identity. Define a means of distinguishing each object regardless of its form or history. Be alert to requirements that call for matching objects by attributes. Define an operation that is guaranteed to produce a unique result for each object, possibly by attaching a symbol that is guaranteed unique. This means of identification may come from the outside, or it may be an arbitrary identifier created by and for the system, but it must correspond to the identity distinctions in the model. The model must define what it *means* to be the same thing.

Identity is not intrinsic to a thing in the world; it is a meaning superimposed because it is useful. In fact, the same real-world thing might or might not be represented as an ENTITY in a domain model.

An application for booking seats in a stadium might treat seats and attendees as ENTITIES. In the case of assigned seating, in which each ticket has a seat number on it, the seat is an ENTITY. Its identifier is the seat number, which is unique within the stadium. The seat may have many other attributes, such as its location, whether the view is obstructed, and the price, but only the seat number, or a unique row and position, is used to identify and distinguish seats.

On the other hand, if the event is "general admission," meaning that ticket holders sit wherever they find an empty seat, there is no need to distinguish individual seats. Only the total number of seats is important. Although the seat numbers are still engraved on the physical seats, there is no need for the software to track them. In fact, it would be erroneous for the model to associate specific seat numbers with tickets, because there is no such constraint at a general admission event. In such a case, seats are *not* ENTITIES, and no identifier is needed.

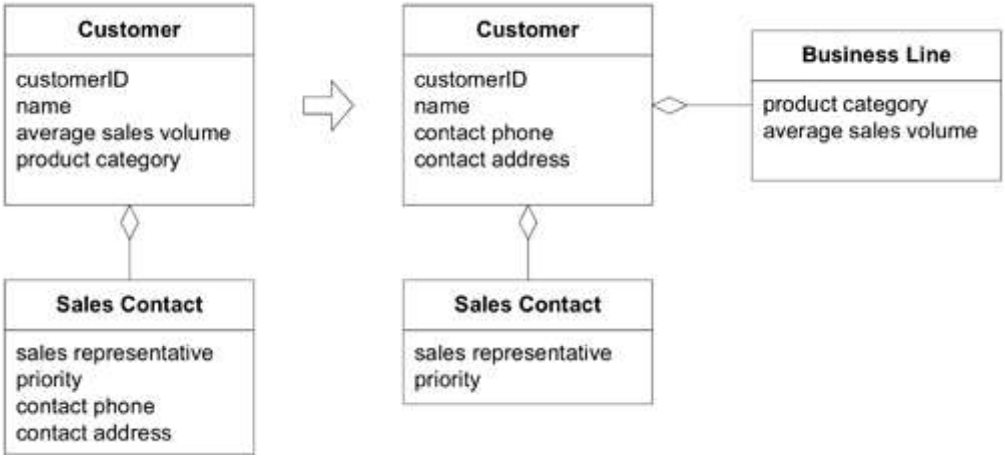
Modeling ENTITIES

It is natural to think about the attributes when modeling an object, and it is quite important to think about its behavior. But the most basic responsibility of ENTITIES is to establish continuity so that behavior can be clear and predictable. They do this best if they are kept spare. Rather than focusing on the attributes or even the behavior, strip the ENTITY object's definition down to the

most intrinsic characteristics, particularly those that identify it or are commonly used to find or match it. Add only behavior that is essential to the concept and attributes that are required by that behavior. Beyond that, look to remove behavior and attributes into other objects associated with the core ENTITY. Some of these will be other ENTITIES. Some will be VALUE OBJECTS, which is the next pattern in this chapter. Beyond identity issues, ENTITIES tend to fulfill their responsibilities by coordinating the operations of objects they own.

The customerID is the one and only identifier of the **Customer** ENTITY in Figure 5.5, but the phone number and address would often be used to find or match a **Customer**. The name does not *define* a person's identity, but it is often used as part of the means of determining it. In this example, the phone and address attributes moved into **Customer**, but on a real project, that choice would depend on how the domain's customers are typically matched or distinguished. For example, if a **Customer** has many contact phone numbers for different purposes, then the phone number is not associated with identity and should stay with the **Sales Contact**.

Figure 5.5. Attributes associated with identity stay with the ENTITY.



Designing the Identity Operation

Each ENTITY must have an operational way of establishing its identity with another object—distinguishable even from another object with the same descriptive attributes. An identifying attribute must be guaranteed to be unique within the system however that system is defined—even if distributed, even when objects are archived.

As mentioned earlier, object-oriented languages have "identity" operations that determine if two references point to the same object by comparing the objects' locations in memory. This kind of identity tracking is too fragile for our purposes. In most technologies for persistent storage of objects, every time an object is retrieved from a database, a new instance is created, and so the initial identity is lost. Every time an object is transmitted across a network, a new instance is created on the destination, and once again the identity is lost. The problem can be even worse when multiple versions of the same object exist in the system, such as when updates propagate through a distributed database.

Even with frameworks that simplify these technical problems, the fundamental issue exists: How do you know that two objects represent the same conceptual ENTITY? The definition of identity emerges from the model. Defining identity demands understanding of the domain.

Sometimes certain data attributes, or combinations of attributes, can be guaranteed or simply

constrained to be unique within the system. This approach provides a unique key for the ENTITY. Daily newspapers, for example, might be identified by the name of the newspaper, the city, and the date of publication. (But watch out for extra editions and name changes!)

When there is no true unique key made up of the attributes of an object, another common solution is to attach to each instance a symbol (such as a number or a string) that is unique within the class. Once this ID symbol is created and stored as an attribute of the ENTITY, it is designated immutable. It must never change, even if the development system is unable to directly enforce this rule. For example, the ID attribute is preserved as the object gets flattened into a database and reconstructed. Sometimes a technical framework helps with this process, but otherwise it just takes engineering discipline.

Often the ID is generated automatically by the system. The generation algorithm must guarantee uniqueness within the system, which can be a challenge with concurrent processing and in distributed systems. Generating such an ID may require techniques that are beyond the scope of this book. The goal here is to point out when the considerations arise, so that developers are aware they have a problem to solve and know how to narrow down their concerns to the critical areas. The key is to recognize that identity concerns hinge on specific aspects of the model. Often, the means of identification demand a careful study of the domain, as well.

When the ID is automatically generated, the user may never need to see it. The ID may be needed only internally, such as in a contact management application that lets the user find records by a person's name. The program needs to be able to distinguish two contacts with exactly the same name in a simple, unambiguous way. The unique, internal IDs let the system do just that. After retrieving the two distinct items, the system will show two separate contacts to the user, but the IDs may not be shown. The user will distinguish them on the basis of their company, their location, and so on.

Finally, there are cases in which a generated ID *is* of interest to the user. When I ship a package through a parcel delivery service, I'm given a tracking number, generated by the shipping company's software, which I can use to identify and follow up on my package. When I book airline tickets or reserve a hotel, I'm given confirmation numbers that are unique identifiers for the transaction.

In some cases, the uniqueness of the ID must apply beyond the computer system's boundaries. For example, if medical records are being exchanged between two hospitals that have separate computer systems, ideally each system will use the same patient ID, but this is difficult if they generate their own symbol. Such systems often use an identifier issued by some other institution, typically a government agency. In the United States, the Social Security number is often used by hospitals as an identifier for a person. Such methods are not foolproof. Not everyone has a Social Security number (children and nonresidents of the United States, especially), and many people object to its use, for privacy reasons.

In less formal situations (say, video rental), telephone numbers are used as identifiers. But a telephone can be shared. The number can change. An old number can even be reassigned to a different person.

For these reasons, specially assigned identifiers are often used (such as frequent flier numbers), and other attributes, such as phone numbers and Social Security numbers, are used to match and verify. In any case, when the application requires an external ID, the users of the system become responsible for supplying IDs that are unique, and the system must give them adequate tools to handle exceptions that arise.

Given all these technical problems, it is easy to lose sight of the underlying conceptual problem: What does it mean for two objects to be the same thing? It is easy enough to stamp each object with an ID, or to write an operation that compares two instances, but if these IDs or operations don't correspond to some meaningful distinction in the domain, they just confuse matters more.

This is why identity-assigning operations often involve human input. Checkbook reconciliation software, for instance, may offer likely matches, but the user is expected to make the final determination.