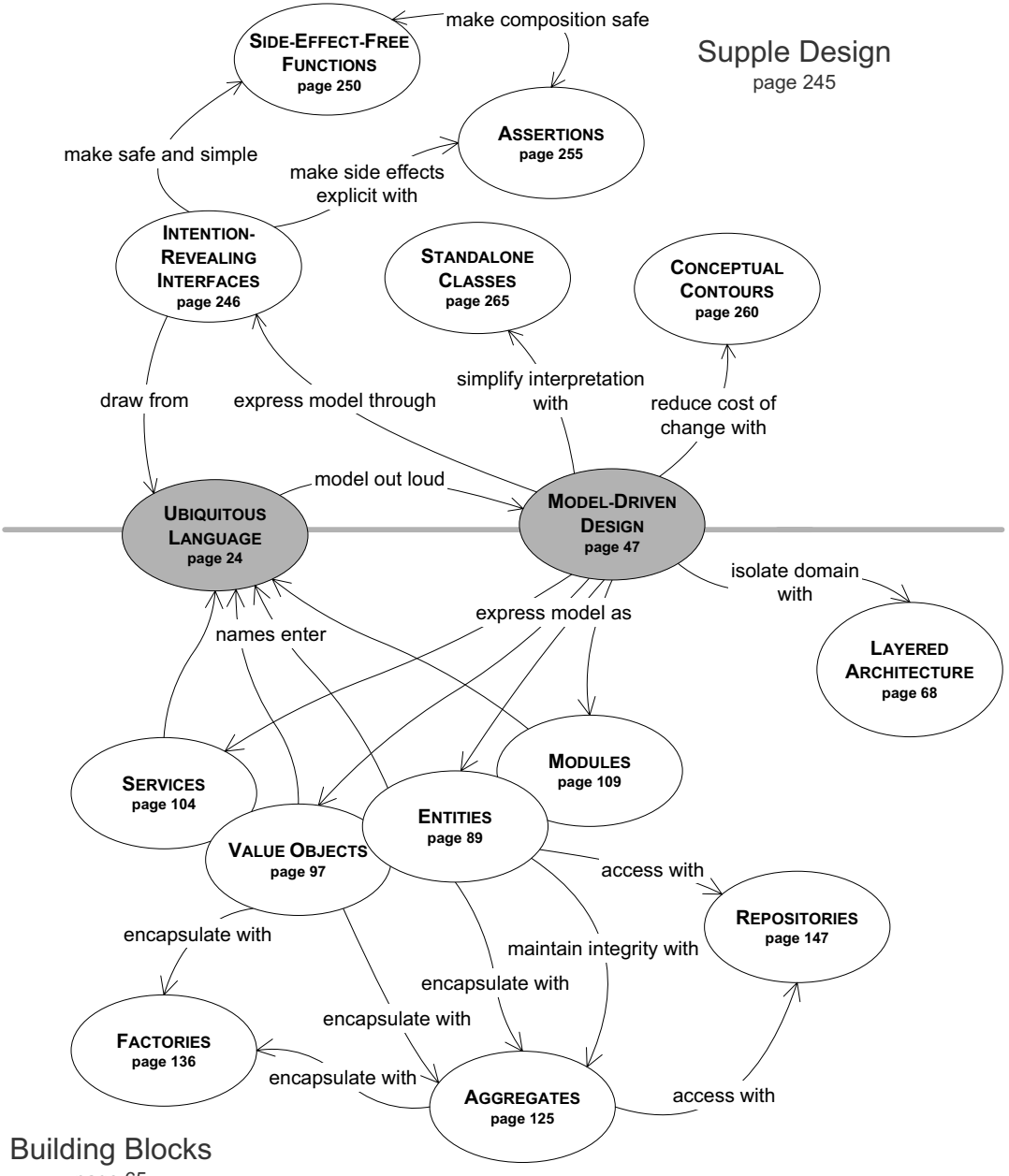
# Domain-Driven DESIGN

## Tackling Complexity in the Heart of Software

Eric Evans

Foreword by Martin Fowler


SIDE-EFFECT-FREE FUNCTIONS
page 250
make composition safe
Supple Design
page 245
ASSERTIONS
page 255
make safe and simple
make side effects explicit with
INTENTION-REVEALING INTERFACES
page 246
STANDALONE CLASSES
page 265
CONCEPTUAL CONTOURS
page 260
draw from
express model through
simplify interpretation with
reduce cost of change with
model out loud
UBIQUITOUS LANGUAGE
page 24
MODEL-DRIVEN DESIGN
page 47
isolate domain with
LAYERED ARCHITECTURE
page 68
express model as
names enter
SERVICES
page 104
MODULES
page 109
ENTITIES
page 89
VALUE OBJECTS
page 97
access with
REPOSITORIES
page 147
encapsulate with
maintain integrity with
encapsulate with
encapsulate with
FACTORIES
page 136
encapsulate with
AGGREGATES
page 125
access with
Building Blocks
page 65

# FACTORIES

When creation of an object, or an entire AGGREGATE, becomes complicated or reveals too much of the internal structure, FACTORIES provide encapsulation.

* * *

Much of the power of objects rests in the intricate configuration of their internals and their associations. An object should be distilled until nothing remains that does not relate to its meaning or support its role in interactions. This mid-life cycle responsibility is plenty. Problems arise from overloading a complex object with responsibility for its own creation.

A car engine is an intricate piece of machinery, with dozens of parts collaborating to perform the engine's responsibility: to turn a shaft. One could imagine trying to design an engine block that could grab on to a set of pistons and insert them into its cylinders, spark plugs that would find their sockets and screw themselves in. But it seems unlikely that such a complicated machine would be as reliable or as efficient as our typical engines are. Instead, we accept that something else will assemble the pieces. Perhaps it will be a human mechanic or perhaps it

will be an industrial robot. Both the robot and the human are actually more complex than the engine they assemble. The job of assembling parts is completely unrelated to the job of spinning a shaft. The assemblers function only during the creation of the car—you don't need a robot or a mechanic with you when you're driving. Because cars are never assembled and driven at the same time, there is no value in combining both of these functions into the same mechanism. Likewise, assembling a complex compound object is a job that is best separated from whatever job that object will have to do when it is finished.

But shifting responsibility to the other interested party, the client object in the application, leads to even worse problems. The client knows what job needs to be done and relies on the domain objects to carry out the necessary computations. If the client is expected to assemble the domain objects it needs, it must know something about the internal structure of the object. In order to enforce all the invariants that apply to the relationship of parts in the domain object, the client must know some of the object's rules. Even calling constructors couples the client to the concrete classes of the objects it is building. No change to the implementation of the domain objects can be made without changing the client, making refactoring harder.

A client taking on object creation becomes unnecessarily complicated and blurs its responsibility. It breaches the encapsulation of the domain objects and the AGGREGATES being created. Even worse, if the client is part of the application layer, then responsibilities have leaked out of the domain layer altogether. This tight coupling of the application to the specifics of the implementation strips away most of the benefits of abstraction in the domain layer and makes continuing changes ever more expensive.

**Creation of an object can be a major operation in itself, but complex assembly operations do not fit the responsibility of the created objects. Combining such responsibilities can produce ungainly designs that are hard to understand. Making the client direct construction muddies the design of the client, breaches encapsulation of the assembled object or AGGREGATE, and overly couples the client to the implementation of the created object.**

Complex object creation is a responsibility of the domain layer, yet that task does not belong to the objects that express the model. There are some cases in which an object creation and assembly corresponds to

a milestone significant in the domain, such as "open a bank account." But object creation and assembly usually have no meaning in the domain; they are a necessity of the implementation. To solve this problem, we have to add constructs to the domain design that are not ENTITIES, VALUE OBJECTS, or SERVICES. This is a departure from the previous chapter, and it is important to make the point clear: We are adding elements to the design that do not correspond to anything in the model, but they are nonetheless part of the domain layer's responsibility.

Every object-oriented language provides a mechanism for creating objects (constructors in Java and C++, instance creation class methods in Smalltalk, for example), but there is a need for more abstract construction mechanisms that are decoupled from the other objects. A program element whose responsibility is the creation of other objects is called a FACTORY.

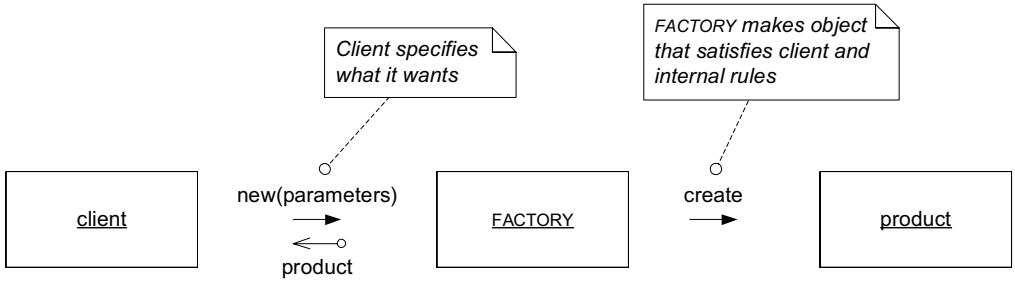



**Figure 6.12**
Basic interactions with a FACTORY

Just as the interface of an object should encapsulate its implementation, thus allowing a client to use the object's behavior without knowing how it works, a FACTORY encapsulates the knowledge needed to create a complex object or AGGREGATE. It provides an interface that reflects the goals of the client and an abstract view of the created object.

Therefore:

**Shift the responsibility for creating instances of complex objects and AGGREGATES to a separate object, which may itself have no responsibility in the domain model but is still part of the domain design. Provide an interface that encapsulates all complex assembly and that does not require the client to reference the concrete classes of the objects being instantiated. Create entire AGGREGATES as a piece, enforcing their invariants.**

* * *

There are many ways to design FACTORIES. Several special-purpose creation patterns—FACTORY METHOD, ABSTRACT FACTORY, and BUILDER—were thoroughly treated in Gamma et al. 1995. That book mostly explored patterns for the most difficult object construction problems. The point here is not to delve deeply into designing FACTORIES, but rather to show the place of FACTORIES as important components of a domain design. Proper use of FACTORIES can help keep a MODEL-DRIVEN DESIGN on track.

The two basic requirements for any good FACTORY are

1. Each creation method is atomic and enforces all invariants of the created object or AGGREGATE. A FACTORY should only be able to produce an object in a consistent state. For an ENTITY, this means the creation of the entire AGGREGATE, with all invariants satisfied, but probably with optional elements still to be added. For an immutable VALUE OBJECT, this means that all attributes are initialized to their correct final state. If the interface makes it possible to request an object that can't be created correctly, then an exception should be raised or some other mechanism should be invoked that will ensure that no improper return value is possible.

2. The FACTORY should be abstracted to the type desired, rather than the concrete class(es) created. The sophisticated FACTORY patterns in Gamma et al. 1995 help with this.

## Choosing FACTORIES and Their Sites

Generally speaking, you create a factory to build something whose details you want to hide, and you place the FACTORY where you want the control to be. These decisions usually revolve around AGGREGATES.

For example, if you needed to add elements inside a preexisting AGGREGATE, you might create a FACTORY METHOD on the root of the AGGREGATE. This hides the implementation of the interior of the AGGREGATE from any external client, while giving the root responsibility for ensuring the integrity of the AGGREGATE as elements are added, as shown in Figure 6.13 on the next page.

Another example would be to place a FACTORY METHOD on an object that is closely involved in spawning another object, although it doesn't own the product once it is created. When the data and possibly the rules of one object are very dominant in the creation of an

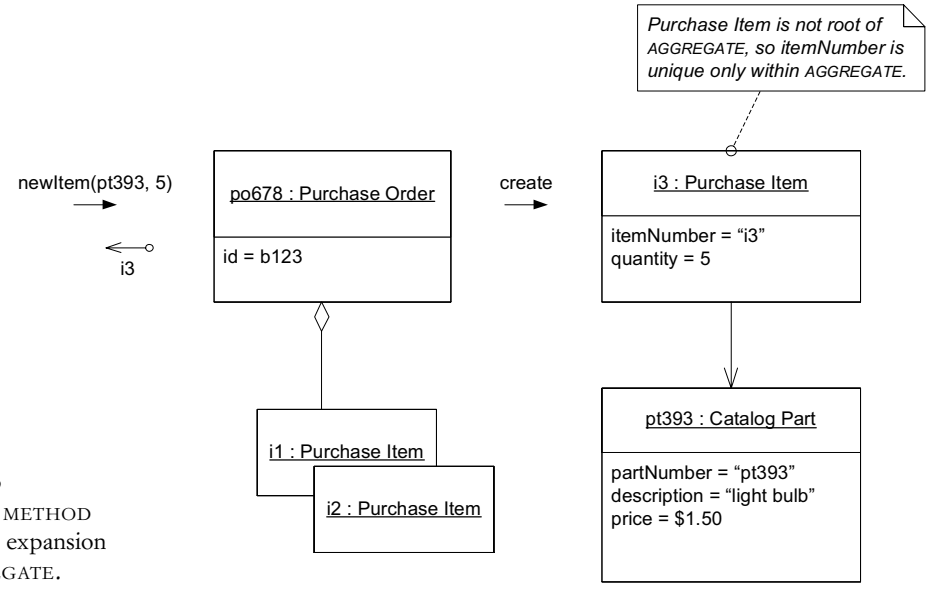
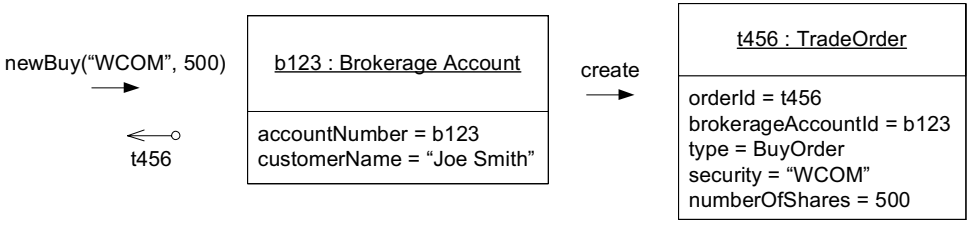



**Figure 6.13**
A FACTORY METHOD encapsulates expansion of an AGGREGATE.

object, this saves pulling information out of the spawner to be used elsewhere to create the object. It also communicates the special relationship between the spawner and the product.

In Figure 6.14, the **Trade Order** is not part of the same AGGREGATE as the **Brokerage Account** because, for a start, it will go on to interact with the trade execution application, where the **Brokerage Account** would only be in the way. Even so, it seems natural to give the **Brokerage Account** control over the creation of **Trade Orders**. The **Brokerage Account** contains information that will be embedded in the **Trade Order** (starting with its own identity), and it contains rules that govern what trades are allowed. We might also benefit from hiding the implementation of **Trade Order**. For example, it might be refactored into a hierarchy, with separate subclasses for **Buy Order** and **Sell Order**. The FACTORY keeps the client from being coupled to the concrete classes.

**Figure 6.14**
A FACTORY METHOD spawns an ENTITY that is not part of the same AGGREGATE.



A FACTORY is very tightly coupled to its product, so a FACTORY should be attached only to an object that has a close natural relationship with the product. When there is something we want to hide—either the concrete implementation or the sheer complexity of construction—yet there doesn't seem to be a natural host, we must create a dedicated FACTORY object or SERVICE. A standalone FACTORY usually produces an entire AGGREGATE, handing out a reference to the root, and ensuring that the product AGGREGATE'S invariants are enforced. If an object interior to an AGGREGATE needs a FACTORY, and the AGGREGATE root is not a reasonable home for it, then go ahead and make a standalone FACTORY. But respect the rules limiting access within an AGGREGATE, and make sure there are only transient references to the product from outside the AGGREGATE.

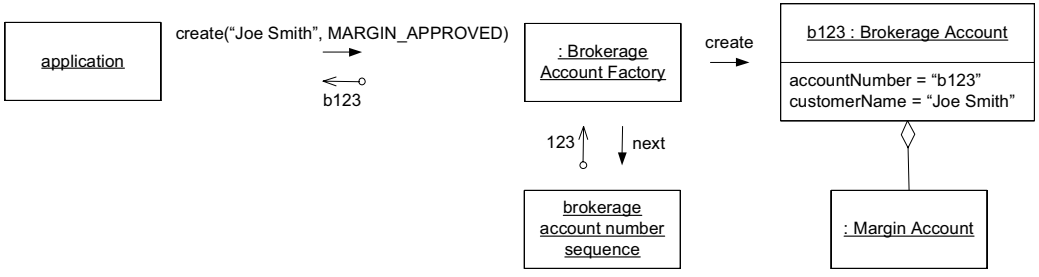



**Figure 6.15**
A standalone FACTORY builds AGGREGATE.

## When a Constructor Is All You Need

I've seen far too much code in which *all* instances are created by directly calling class constructors, or whatever the primitive level of instance creation is for the programming language. The introduction of FACTORIES has great advantages, and is generally underused. Yet there are times when the directness of a constructor makes it the best choice. FACTORIES can actually obscure simple objects that don't use polymorphism.

The trade-offs favor a bare, public constructor in the following circumstances.

- The class is the type. It is not part of any interesting hierarchy, and it isn't used polymorphically by implementing an interface.
- The client cares about the implementation, perhaps as a way of choosing a STRATEGY.