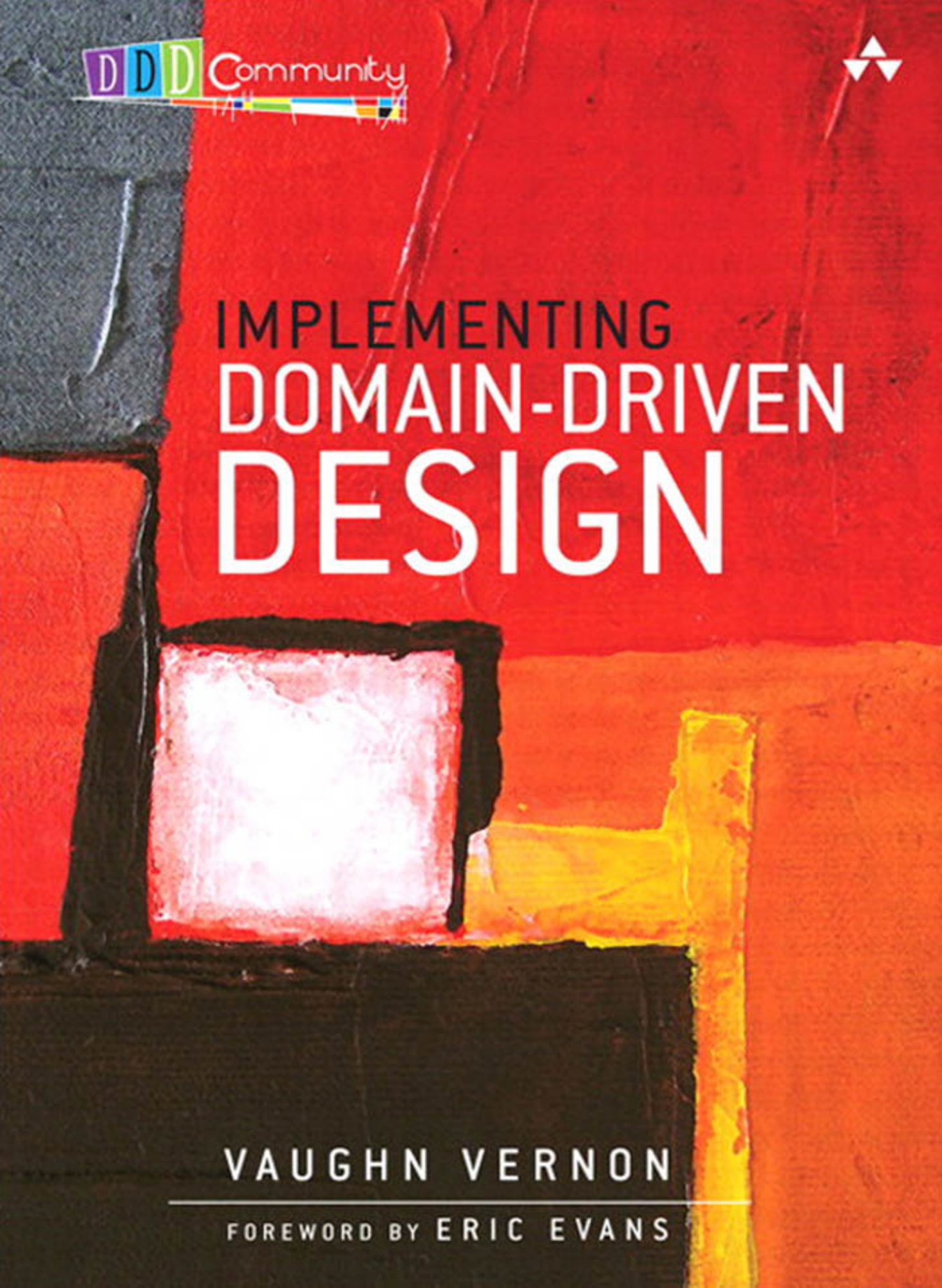
DDD Community
IMPLEMENTING
DOMAIN-DRIVEN
DESIGN
VAUGHN VERNON
FOREWORD BY ERIC EVANS

Praise for *Implementing Domain-Driven Design*

"With *Implementing Domain-Driven Design*, Vaughn has made an important contribution not only to the literature of the Domain-Driven Design community, but also to the literature of the broader enterprise application architecture field. In key chapters on Architecture and Repositories, for example, Vaughn shows how DDD fits with the expanding array of architecture styles and persistence technologies for enterprise applications—including SOA and REST, NoSQL and data grids—that has emerged in the decade since Eric Evans' seminal book was first published. And, fittingly, Vaughn illuminates the blocking and tackling of DDD—the implementation of entities, value objects, aggregates, services, events, factories, and repositories—with plentiful examples and valuable insights drawn from decades of practical experience. In a word, I would describe this book as *thorough*. For software developers of all experience levels looking to improve their results, and design and implement domain-driven enterprise applications consistently with the best current state of professional practice, *Implementing Domain-Driven Design* will impart a treasure trove of knowledge hard won within the DDD and enterprise application architecture communities over the last couple decades."

—Randy Stafford, Architect At-Large, Oracle Coherence Product Development

"Domain-Driven Design is a powerful set of thinking tools that can have a profound impact on how effective a team can be at building software-intensive systems. The thing is that many developers got lost at times when applying these thinking tools and really needed more concrete guidance. In this book, Vaughn provides the missing links between theory and practice. In addition to shedding light on many of the misunderstood elements of DDD, Vaughn also connects new concepts like Command/Query Responsibility Segregation and Event Sourcing that many advanced DDD practitioners have used with great success. This book is a must-read for anybody looking to put DDD into practice."

—Udi Dahan, Founder of NServiceBus

"For years, developers struggling to practice Domain-Driven Design have been wishing for more practical help in actually implementing DDD. Vaughn did an excellent job in closing the gap between theory and practice with a complete implementation reference. He paints a vivid picture of what it is like to do DDD in a contemporary project, and provides plenty of practical advice on how to approach and solve typical challenges occurring in a project life cycle."

—Alberto Brandolini, DDD Instructor, Certified by Eric Evans and
Domain Language, Inc.

"*Implementing Domain-Driven Design* does a remarkable thing: it takes a sophisticated and substantial topic area in DDD and presents it clearly, with nuance, fun and finesse. This book is written in an engaging and friendly style, like a trusted advisor giving you expert counsel on how to accomplish what is most important. By the time you finish the book you will be able to begin applying all the important concepts of

Cowboy Logic

LB: "*Dallas* and *Dynasty*, now *those* are what I call sagas!"

AJ: "For all you German readers, y'all know *Dynasty* as *Der Denver Clan*."

Extending the previous example, we could create parallel pipelines by adding just one new Filter, `TotalPhoneNumbersCounter`, as an additional subscriber to `AllPhoneNumbersListed`. It receives the Event `AllPhoneNumbersListed` virtually in parallel with the `PhoneNumberFinder`. The new Filter has a very simple goal, counting all existing contacts. This time, however, `PhoneNumberExecutive` both starts the Long-Running Process and tracks it through completion. The executive may or may not reuse the `PhoneNumbersPublisher`, but the important thing is what's new about it. The executive, implemented as an Application Service or Command Handler, tracks the progress of the Long-Running Process and understands when it is completed and what to do when that happens. Refer to Figure 4.9 as we step through the sample Long-Running Process.

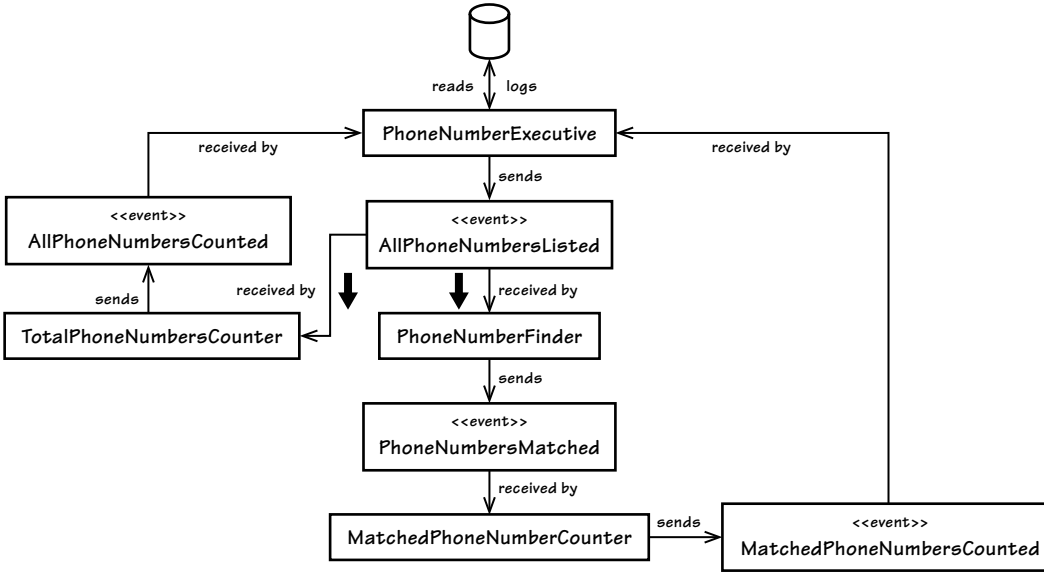

Figure 4.9 The single Long-Running Process executive initiates the parallel processing and tracks it to completion. The wider arrows indicate where the parallelism begins when two Filters receive the same Event.

Different Ways to Design a Long-Running Process

Here are three approaches to designing a Long-Running Process, although there may be more:

- Design the process as a composite task, which is tracked by an executive component that records the steps and completeness of the task using a persistent object. This is the approach discussed most thoroughly here.
- Design the process as a set of partner Aggregates that collaborate in a set of activities. One or more Aggregate instances act as the executive and maintain the overall state of the process. This is the approach promoted by Amazon's Pat Helland [Helland].
- Design a stateless process in that each message handler component that receives an Event-carrying message must enrich the received Event with more task progress information as it sends the next message. The state of the overall process is maintained only in the body of each message sent from collaborator to collaborator.

Since the initial Event is now subscribed to by two components, both Filters receive the same Event virtually simultaneously. The original Filter goes about as it always has, matching the specific `303` text pattern. The new Filter only counts all lines, and when it has completed, it sends the Event `AllPhoneNumbersCounted`. The Event includes the count of total contacts. If there are, for example, 15 total phone numbers, the Event `count` property is set to 15.

Now it is the responsibility of `PhoneNumberExecutive` to subscribe to two Events, both `MatchedPhoneNumbersCounted` and `AllPhoneNumbersCounted`. The parallel processing is not considered completed until both of these Domain Events are received. When completion is reached, the results of the parallel processing are merged into a single result. The executive now logs

```
3 of 15 phone numbers matched on July 15, 2012 at 11:27 PM
```

The log output is enhanced with the total count of phone numbers in addition to the previous matching, date, and time information. Although the tasks performed to yield results were really simple, they were performed in parallel. And if at least some of the subscriber components were deployed to different computing nodes, the parallel processing was also distributed.

There is a problem with this Long-Running Process, however. The `PhoneNumberExecutive` currently has no way of knowing that it has

received the two completion Domain Events associated with the specific, corresponding parallel processes. If many such processes were started in parallel, and completion Events for each were received out of order, how would the executive know which parallel process was ending? For our synthetic example, logging with mismatched events is hardly tragic. But when dealing with corporate business domains, an improperly aligned Long-Running Process could be disastrous.

The first step in the solution to this troublesome situation is to *assign a unique Process identity* that is carried by each of the associated Domain Events. This could be the same identity assigned to the originating Domain Event that causes the Long-Running Process to begin (for example, `AllPhoneNumbersListed`). We could use a universally unique identifier (UUID) allocated specifically to the Process. See **Entities (5)** and **Domain Events (8)** for a discussion of providing unique identity. The `PhoneNumberExecutive` would now write output to the log only upon receiving completion Events with equal identities. However, we can't expect the executive to wait around until all the completion Events are received. It, too, is an Event subscriber that comes and goes with the receipt and handling of each delivery.

Executive *and* Tracker?

Some find that merging the concepts of *executive* and *tracker* into a single object—an Aggregate—to be the simplest approach. Implementing such an Aggregate as a part of the domain model that naturally tracks just a part of the overall Process can be a liberating technique. For one, we avoid developing a separate tracker as state machine, in addition to the Aggregates that must also exist. In fact, the most basic Long-Running Processes are best implemented just that way.

In a Hexagonal Architecture, a Port-Adapter message handler would simply dispatch to an Application Service (or Command Handler), which would load the target Aggregate and delegate to its appropriate command method. Since the Aggregate would in turn fire a Domain Event, the Event would be published in part as an indication that the Aggregate has completed its role in the Process.

This approach closely follows that promoted by Pat Helland, which he refers to as *partner activities* [Helland], and is the second approach described in the sidebar "Different Ways to Design a Long-Running Process." Ideally, however, discussing a separate executive and tracker is a more effective way to teach the overall technique, and a more intuitive way to learn it.

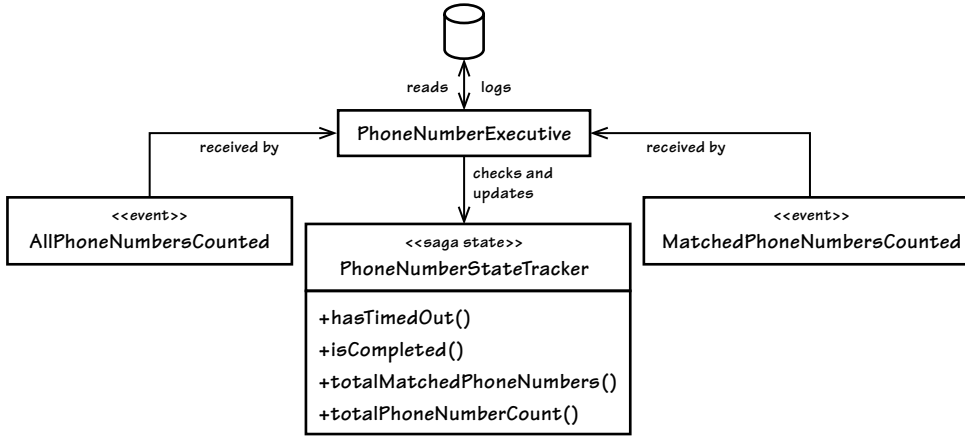

Figure 4.10 A `PhoneNumberStateTracker` serves as a Long-Running Process state object to track progress. The tracker is implemented as an Aggregate.

In an actual domain each instance of a Process executive creates a new Aggregate-like state object for tracking its eventual completion. The state object is created when the Process begins, associating the same unique identity that each related Domain Event must carry. It may also be useful for it to hold a timestamp of when the Process began (the reasons are discussed later in the chapter). The Process state tracker object is illustrated in Figure 4.10.

As each pipeline in the parallel processing completes, the executive receives a corresponding completion Event. The executive retrieves the state tracking instance by matching the unique Process identity carried by the received Event and sets a property that represents the step just completed.

The Process state instance usually has a method such as `isCompleted()`. As each step is completed and recorded on this state tracker, the executive checks `isCompleted()`. This method checks for the recorded completion of all required parallel processes. When the method answers `true`, the executive has the option to publish a final Domain Event if required by the business. This Event could be required if the completing Process is just a branch in a larger parallel process, for example.

A given messaging mechanism may lack features that guarantee *single delivery* of each Event.[7] If it is possible for the messaging mechanism to deliver a Domain Event message two or more times, we can use the Process state object to de-duplicate. Does this require special features to be provided by the messaging mechanism? Consider how it can be handled without them.

7. This does not mean guaranteed delivery, but guaranteed single delivery, or once and only once.

When each completion Event is received, *the executive checks the state object for an existing record of completion for that specific Event*. If the completion indicator is already set, the Event is considered a duplicate and is ignored, yet acknowledged.[8] Another option is to *design the state object to be idempotent*. That way, if duplicate messages are received by the executive, the state object absorbs the duplicate occurrence recordings equally. While only the second option designs the state tracker itself as idempotent, both of these approaches support idempotent messaging. See **Domain Events** (8) for further discussion of Event de-duplication.

Some Process completion tracking may be time-sensitive. We can deal with Process time-outs passively or actively. Recall that the Process state tracker can hold a timestamp of its inception. Add to this a total allowable time constant (or configuration) value and the executive can manage time-sensitive Long-Running Processes.

A passive time-out check is performed each time a parallel processing completion Event is received by the executive. The executive retrieves the state tracker and asks it if a time-out has occurred. A method such as `hasTimedOut()` can serve that purpose. If the passive time-out check indicates that the allowable time threshold has been exceeded, the Process state tracker can be marked as abandoned. It's also possible to publish a corresponding failure Domain Event. Note that a disadvantage of the passive time-out check is that the Process could remain active well past its threshold if one or more completion Events are for some reason never received by the executive. This may be unacceptable if a larger parallel process is dependent on certain success or failure of this Process.

An active Process time-out check can be managed using an external timer. For example, a JMX `TimerMBean` instance is one way to get a Java-managed timer. The timer is set for the maximum time-out threshold just as the Process begins. When the timer fires, the listener accesses the Process state tracker. If the state is not already completed (always checked in case the timer fires just as an asynchronous Event completes the Process), it is then marked as abandoned, and a corresponding failure Event is published. If the state tracker is marked as completed prior to the timer firing, the timer can then be terminated. One disadvantage of the active time-out check is that it requires more system resources, which may burden a high-traffic environment. Also, a race condition between the timer and the arriving completion Event could incorrectly cause failure.

8. When the messaging mechanism finally receives acknowledgment of receipt, the message will not be delivered again.

Long-Running Processes are often associated with distributed parallel processing but have nothing to do with distributed transactions. They require a mindset that embraces eventual consistency. We must enter any effort to design a Long-Running Process soberly, with the expectation that when infrastructure or the tasks themselves fail, well-designed error recovery is essential. Every system participating in a single instance of a Long-Running Process must be considered inconsistent with all other participants until the executive receives the final completion notification. True, some Long-Running Processes may be capable of succeeding with only partial completion, or they may delay for even a number of days before full completion. But if the Process runs aground and the participating systems are left in inconsistent states, compensation may be necessary. If compensation is mandatory, it could surpass the complexity of designing the success path. Perhaps business procedures could allow for failures and offer workflow solutions instead.

The SaaSOvation teams employ an Event-Driven Architecture across Bounded Contexts, and the ProjectOvation team will use the simplest form of a Long-Running Process to manage the creation of `Discussions` assigned to `Product` instances. The overarching style is Hexagonal to manage the outside messaging and publishing of Domain Events around the enterprise.

Not to be overlooked is that the Long-Running Process executive can publish one, two, or more Events to initiate the parallel processing. There may also be not only two, but three or more subscribers to any initiating Event or Events. In other words, a Long-Running Process may lead to many separate business process activities executing simultaneously. Thus, our synthetic example is limited in complexity only for the sake of communicating the basic concepts of a Long-Running Process.

Long-Running Processes are often useful when integration with legacy systems can have high latency. Even if latency and legacy are not the chief concerns, we still benefit from the distribution and parallelism with elegance, which can lead to highly scalable, highly available business systems.

Some messaging mechanisms have built-in support for Long-Running Processes, which can greatly expedite adoption. One such is [NServiceBus], which specifically calls them Sagas. Another Saga implementation is provided with [MassTransit].