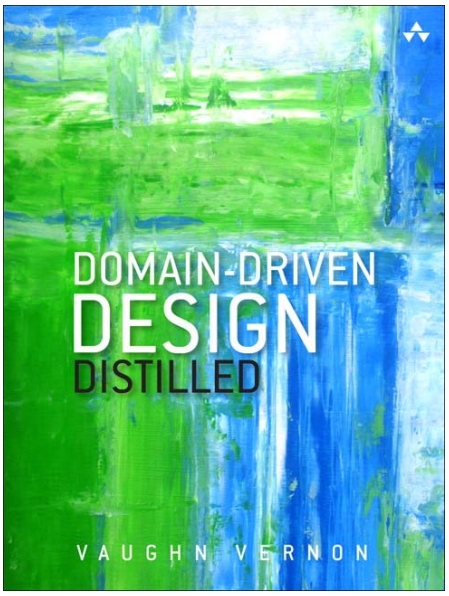
Bring DDD to Life



ISBN: 9780134434421 (PRINT)

ISBN: 9780134434988 (EBOOK)

ORDER & SAVE

SAVE 35% WHEN YOU ORDER

from informit.com/vernon and enter the discount code **LIGHTBEND** during checkout

FREE US SHIPPING on print books

Major eBook Formats

Only InformIT offers PDF, EPUB, & MOBI together for one price

OTHER AVAILABILITY

Through O'Reilly Safari subscription service

Booksellers and online retailers including Amazon/Kindle store and Barnes and Noble/bn.com

Domain-Driven Design (DDD) software modeling delivers powerful results in practice, not just in theory, which is why developers worldwide are rapidly moving to adopt it. Now, for the first time, there's an accessible guide to the basics of DDD: What it is, what problems it solves, how it works, and how to quickly gain value from it.

Concise, readable, and actionable, ***Domain-Driven Design Distilled*** never buries you in detail–it focuses on what you need to know to get results. Vaughn Vernon, author of the best-selling *Implementing Domain-Driven Design,* draws on his twenty years of experience applying DDD principles to real-world situations. He is uniquely well-qualified to demystify its complexities, illuminate its subtleties, and help you solve the problems you might encounter.

Vernon guides you through each core DDD technique for building better software. You'll learn how to segregate domain models using the powerful Bounded Contexts pattern, to develop a Ubiquitous Language within an explicitly bounded context, and to help domain experts and developers work together to create that language. Vernon shows how to use Subdomains to handle legacy systems and to integrate multiple Bounded Contexts to define both team relationships and technical mechanisms.

Coverage includes

- What DDD can do for you and your organization—and why it's so important
- The cornerstones of strategic design with DDD: Bounded Contexts and Ubiquitous Language
- Strategic design with Subdomains
- Context Mapping: helping teams work together and integrate software more strategically
- Tactical design with Aggregates and Domain Events
- Using project acceleration and management tools to establish and maintain team cadence

Pearson

informIT®
the trusted technology learning source

Contents

Vernon_Book.indb vii 4/28/16 2:07 PM

Preface

Why is model building such a fun and rewarding activity? Ever since I was a kid I have loved to build models. At that time I mostly built models of cars and airplanes. I am not sure where LEGO was in those days. Still, LEGO has been a big part of my son's life since he was very young. It is so fascinating to conceive and build models with those small bricks. It's easy to come up with basic models, and it seems you can extend your ideas almost endlessly.

You can probably relate to some kind of youthful model building.

Models occur in so many situations in life. If you enjoy playing board games, you are using models. It might be a model of real estate and property owners, or models of islands and survivors, or models of territories and building activities, and who knows what all. Similarly, video games are models. Perhaps they model a fantasy world with fanciful characters playing fantastic roles. A deck of cards and related games model power. We use models all the time and probably so often that we don't give most models a well-deserved acknowledgment. Models are just part of our lives.

But why? Every person has a learning style. There are a number of learning styles, but three of the most discussed are auditory, visual, and tactile styles. The auditory learners learn by hearing and listening. The visual learners learn by reading or seeing imagery. The tactile learners learn by doing something that involves touching. It's interesting that each learning style is heavily favored by the individual to the extent that he or she can sometimes have trouble with other types of learning. For example, tactile learners likely remember what they have done but may have problems remembering what was said during the process. With model building, you would think that visual and tactile learners would

Vernon_Book.indb xi 4/28/16 2:07 PM

have a huge advantage over the auditory learners, because model building seems to mostly involve visual and tactile stimulation. However, that might not always hold true, especially if a team of model builders uses audible communication in their building process. In other words, model building holds out the possibility to accommodate the learning style of the vast majority of individuals.

With our natural affinity to learning through model building, why would we not naturally desire to model the software that ever increasingly assists and influences our lives? In fact, to model software appears to be, well, human. And model software we should. It seems to me that humans should be elite software model builders.

It is my strong desire to help you be as human as you can possibly be by modeling software using some of the best software modeling tools available. These tools are packaged under the name "Domain-Driven Design," or DDD. This toolbox, actually a set of patterns, was first codified by Eric Evans in the book *Domain-Driven Design: Tackling Complexity in the Heart of Software* [DDD]. It is my vision to bring DDD to everyone possible. To make my point, if I must say that I want to bring DDD to the masses, then so be it. That is where DDD deserves to be, and DDD is the toolbox that model-oriented humans deserve to use to create their most advanced software models. With this book, I am determined to make learning and using DDD as simple and easy as possible and to bring that to the broadest conceivable audience.

For auditory learners, DDD holds out the prospect of learning through the team communication of building a model based on the development of a *Ubiquitous Language*. For visual and tactile learners, the process of using DDD tools is very visual and hands-on as your team models both strategically and tactically. This is especially true when drawing *Context Maps* and modeling the business process using *Event Storming*. Thus, I believe that DDD can support everyone who wants to learn and achieve greatness through model building.

Who Is This Book For?

This book is for everyone interested in learning the most important DDD aspects and tools and in learning quickly. The most common readers

are software architects and software developers who will put DDD into practice on projects. Very often, software developers quickly discover the beauty of DDD and are keenly attracted to its powerful tooling. Even so, I have made the subject understandable for executives, domain experts, managers, business analysts, information architects, and testers alike. There's really no limit to those in the information technology (IT) industry and research and development (R&D) environments who can benefit from reading this book.

If you are a consultant and you are working with a client to whom you have recommended the use of DDD, provide this book as a way to bring the major stakeholders up to speed quickly. If you have developers—perhaps junior or midlevel or even senior—working on your project who are unfamiliar with DDD but need to use it very soon, make sure that they read this book. By reading this book, at minimum, all the project stakeholders and developers will have the vocabulary and understand the primary DDD tools being used. This will enable them to share things meaningfully as they move the project forward.

Whatever your experience level and role, read this book and then practice DDD on a project. Afterward, reread this book and see what you can learn from your experiences and where you can improve in the future.

What This Book Covers

The first chapter, "DDD for Me," explains what DDD can do for you and your organization and provides a more detailed overview of what you will learn and why it's important.

Chapter 2, "Strategic Design with Bounded Contexts and the Ubiquitous Language," introduces DDD strategic design and teaches the cornerstones of DDD, *Bounded Contexts* and the *Ubiquitous Language*. Chapter 3, "Strategic Design with Subdomains," explains *Subdomains* and how you can use them to deal with the complexity of integrating with existing legacy systems as you model your new applications. Chapter 4, "Strategic Design with Context Mapping," teaches the variety of ways that teams work together strategically and ways that their software can integrate. This is called *Context Mapping*.

Chapter 5, "Tactical Design with Aggregates," switches your attention to tactical modeling with *Aggregates*. An important and powerful tactical modeling tool to be used with *Aggregates* is *Domain Events*, which is the subject of Chapter 6, "Tactical Design with Domain Events."

Finally, in Chapter 7, "Acceleration and Management Tools," the book highlights some project acceleration and project management tools that can help teams establish and maintain their cadence. These two topics are seldom if ever discussed in other DDD sources and are sorely needed by those who are determined to put DDD into practice.

Conventions

There are only a few conventions to keep in mind while reading. All of the DDD tools that I discuss are printed in italics. For example, you will read about *Bounded Contexts* and *Domain Events*. Another convention is that any source code is presented in a monospaced Courier font. Acronyms and abbreviations for works listed in the References on pages 136-137 appear in square brackets throughout the chapters.

Even so, what this book emphasizes most, and what your brain should like a lot, is visual learning through lots of diagrams and figures. You will notice that there are no figure numbers in the book, because I didn't want to distract you with so many of those. In every case the figures and diagrams precede the text that discusses them, which means that the graphic visuals introduce thoughts as you work your way through the book. That means that when you are reading text, you can count on referring back to the previous figure or diagram for visual support.

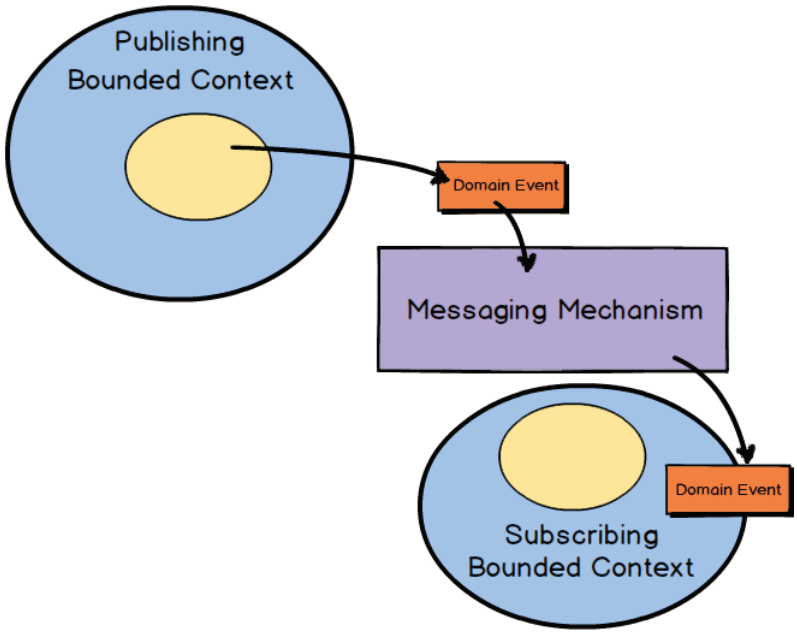
About the Author

Vaughn Vernon is a veteran software craftsman and thought leader in simplifying software design and implementation. He is the author of the best-selling books *Implementing Domain-Driven Design* and *Reactive Messaging Patterns with the Actor Model*, also published by Addison-Wesley. He has taught his IDDD Workshop around the globe to hundreds of software developers. Vaughn is a frequent speaker at industry conferences. He is most interested in distributed computing, messaging, and in particular with the Actor model. Vaughn specializes in consulting around Domain-Driven Design and DDD using the Actor model with Scala and Akka. You can keep up with Vaughn's latest work by reading his blog at www.VaughnVernon.co and by following him on his Twitter account @VaughnVernon.

Vernon_Book.indb xvii 4/28/16 2:07 PM

Chapter 6

Tactical Design with Domain Events



You've already seen a bit in previous chapters about how *Domain Events* are used. A *Domain Event* is a record of some business-significant occurrence in a *Bounded Context*. By now you know that *Domain Events* are a very important tool for strategic design. Still, often during tactical design *Domain Events* are conceptualized and become a part of your *Core Domain*.

To see the full power that results from using *Domain Events*, consider the concept of causal consistency. A business domain provides causal consistency if its operations that are causally related—one operation

Vernon_Book.indb 99 4/28/16 2:07 PM

causes another—are seen by every dependent node of a distributed system in the same order [Causal]. This means that causally related operations must occur in a specific order, and thus one thing cannot happen unless another thing happens before it. Perhaps this means that one *Aggregate* cannot be created or modified until it is clear that a specific operation occurred to another *Aggregate*:

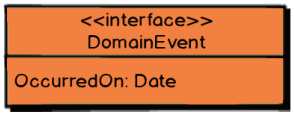
1. Sue posts a message saying, "I lost my wallet!"
2. Gary says in reply, "That's terrible!"
3. Sue posts a message saying, "Don't worry, I found my wallet!"
4. Gary replies, "That's great!"

If these messages were replicated on distributed nodes, but not in a causal order, it could appear that Gary said, "That's great!" to the message "I lost my wallet!" The message "That's great!" is not directly or causally related to "I lost my wallet!" and that's definitely not what Gary wants Sue or anyone else to read. Thus, if causality is not achieved in the proper way, the overall domain would be wrong or at least misleading. This sort of causal, linearized system architecture can be readily achieved through the creation and publication of correctly ordered *Domain Events*.

From tactical design efforts *Domain Events* become a reality in your domain model and can as a result be published and consumed in your own *Bounded Context* and by others. It's a very powerful way to inform interested listeners of important occurrences that have taken place. Now you will learn how to model *Domain Events* and use them in your *Bounded Contexts*.

Designing, Implementing, and Using Domain Events

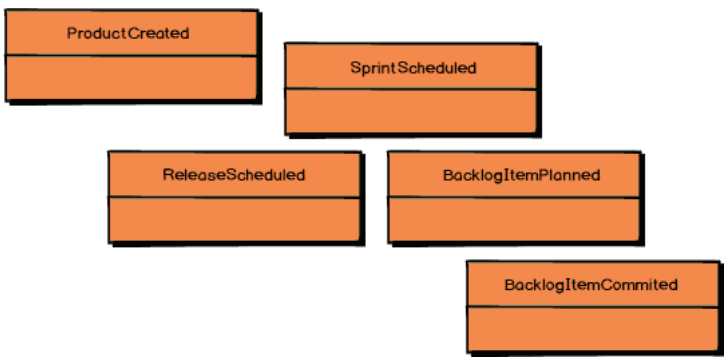
The following guides you through the steps needed to effectively design and implement *Domain Events* in your *Bounded Context*. Following this, you will also see examples of how *Domain Events* are used.




```
public interface DomainEvent
{
  public Date OccurredOn
  {
    get;
  }
}
```

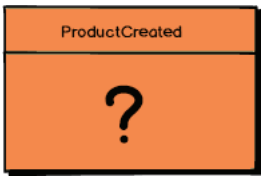
This C# code might be considered the minimum interface that every *Domain Event* should support. You generally want to convey the date and time when your *Domain Event* occurred, so that's provided by the `OccurredOn` property. This detail is not an absolute necessity, but it is often useful. So your *Domain Event* types would likely implement this interface.

You must show care in how you name your *Domain Event* types. The words you use should reflect your model's *Ubiquitous Language*. These words will form a bridge between the happenings in your model and the outside world. It's vital that you communicate your happenings well.



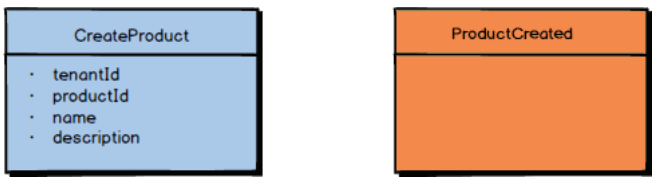
Your *Domain Event* type names should be a statement of a past occurrence, that is, a verb in the past tense. Here are some examples from the *Agile Project Management Context*: `ProductCreated`, for instance, states that a Scrum product was created at some past time.

Other *Domain Events* are `ReleaseScheduled`, `SprintScheduled`, `BacklogItemPlanned`, and `BacklogItemCommitted`. Each of the names clearly and concisely states what happened in your *Core Domain*.



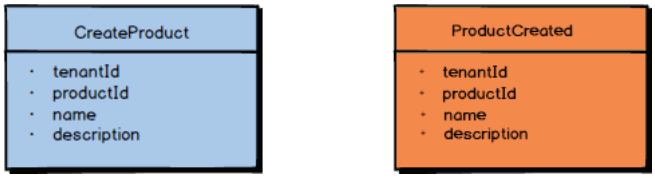
It's a combination of the *Domain Event*'s name and its properties that fully conveys the record of what happened in the domain model. But what properties should a *Domain Event* hold?

Ask yourself, "What is the application stimulus that causes the *Domain Event* to be published?" In the case of `ProductCreated` there is a command that causes it (a command is just the object form of a method/action request). The command is named `CreateProduct`. So you can say that `ProductCreated` is the result of a `CreateProduct` command.

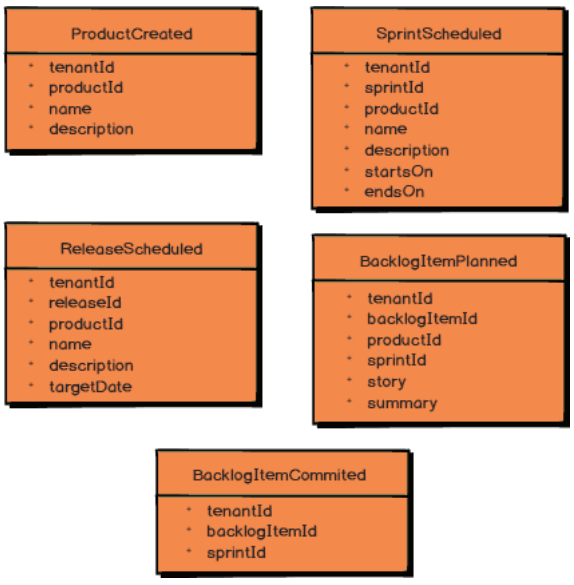


The `CreateProduct` command has a number of properties: (1) the `tenantId` that identifies the subscribing tenant, (2) the `productId` that identifies the unique `Product` being created, the (3) `Product name`,

and (4) the `Product description`. Each of these properties is essential to creating a `Product`.



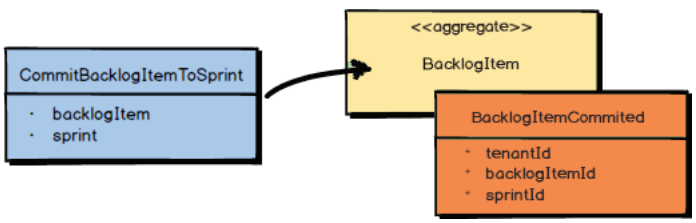
Therefore, the `ProductCreated` *Domain Event* should hold all the properties that were provided with the command that caused it to be created: (1) `tenantId`, (2) `productId`, (3) `name`, and (4) `description`. This will fully and accurately inform all subscribers what happened in the model; that is, a `Product` was created, it was for the tenant identified with the `tenantId`, the `Product` was uniquely identified with `productId`, and the `Product` had the `name` and `description` assigned to it.



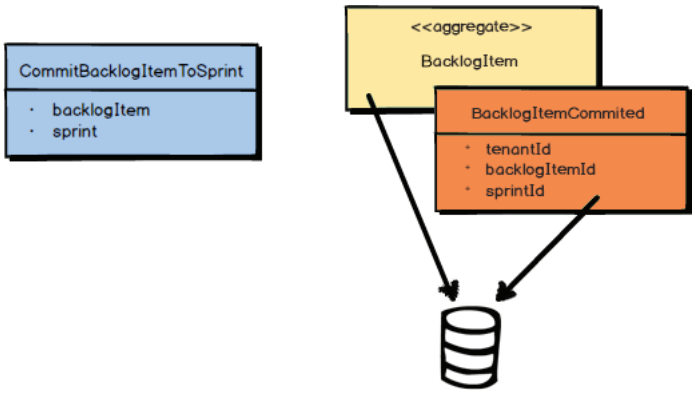
These five examples give you a good idea of the properties that should be included with the various *Domain Events* published by the *Agile*

Project Management Context. For instance, when a `BacklogItem` is committed to a `Sprint`, the `BacklogItemCommitted` *Domain Event* is instantiated and published. This *Domain Event* contains the `tenantId`, the `backlogItemId` of the `BacklogItem` that was committed, and the `sprintId` of the `Sprint` to which it was committed.

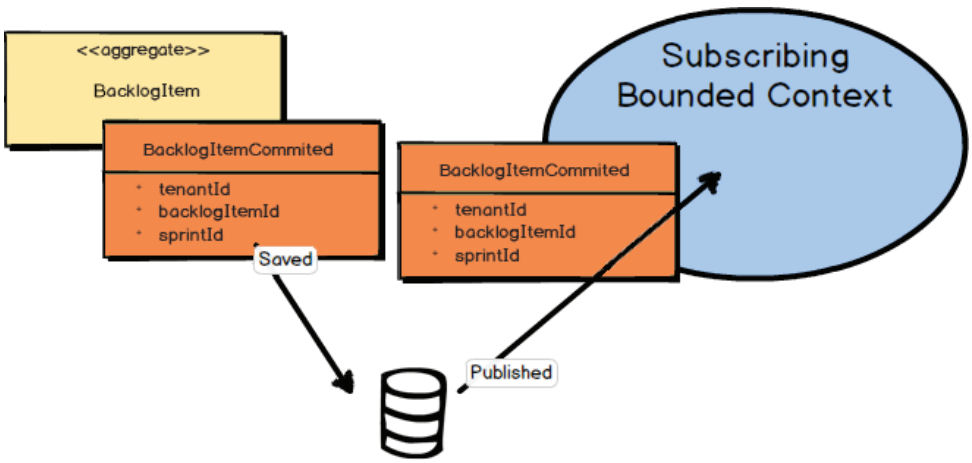
As described in Chapter 4, "Strategic Design with Context Mapping," there are times when a *Domain Event* can be enriched with additional data. This can be especially helpful to consumers that don't want to query back on your *Bounded Context* to obtain additional data that they need. Even so, you must be careful not to fill up a *Domain Event* with so much data that it loses its meaning. For example, consider the problem with `BacklogItemCommitted` holding the entire state of the `BacklogItem`. According to this *Domain Event,* what actually happened? All the extra data may make it unclear, unless you require the consumer to have a deep understanding of your `BacklogItem` element. Also, consider using `BacklogItemUpdated` with the full state of the `BacklogItem`, as opposed to providing `BacklogItemCommitted`. What happened to the `BacklogItem` is very unclear, because the consumer would have to compare the latest `BacklogItemUpdated` to the previous `BacklogItemUpdated` in order to understand what actually occurred to the `BacklogItem`.



To make the proper use of *Domain Events* clearer, let's walk through one scenario. The product owner commits a `BacklogItem` to a `Sprint`. The command itself causes the `BacklogItem` and the `Sprint` to be loaded. Then the command is executed on the `BacklogItem` *Aggregate*. This causes the state of the `BacklogItem` to be modified, and then the `BacklogItemCommitted` *Domain Event* is published as an outcome.



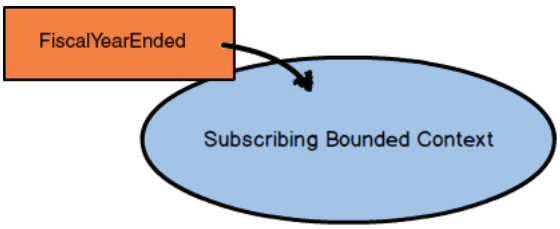
It's important that the modified *Aggregate* and the *Domain Event* be saved together in the same transaction. If you are using an object-relational mapping tool, you would save the *Aggregate* to one table and the *Domain Event* to an event store table, and then commit the transaction. If you are using *Event Sourcing*, the state of the *Aggregate* is fully represented by the *Domain Events* themselves. I discuss *Event Sourcing* in the next section of this chapter. Either way, persisting the *Domain Event* in the event store preserves its causal ordering relative to what has happened across the domain model.



Once your *Domain Event* is saved to the event store, it can be published to any interested parties. This might be within your own *Bounded*

Context and to external *Bounded Contexts*. This is your way of telling the world that something noteworthy has occurred in your *Core Domain*.

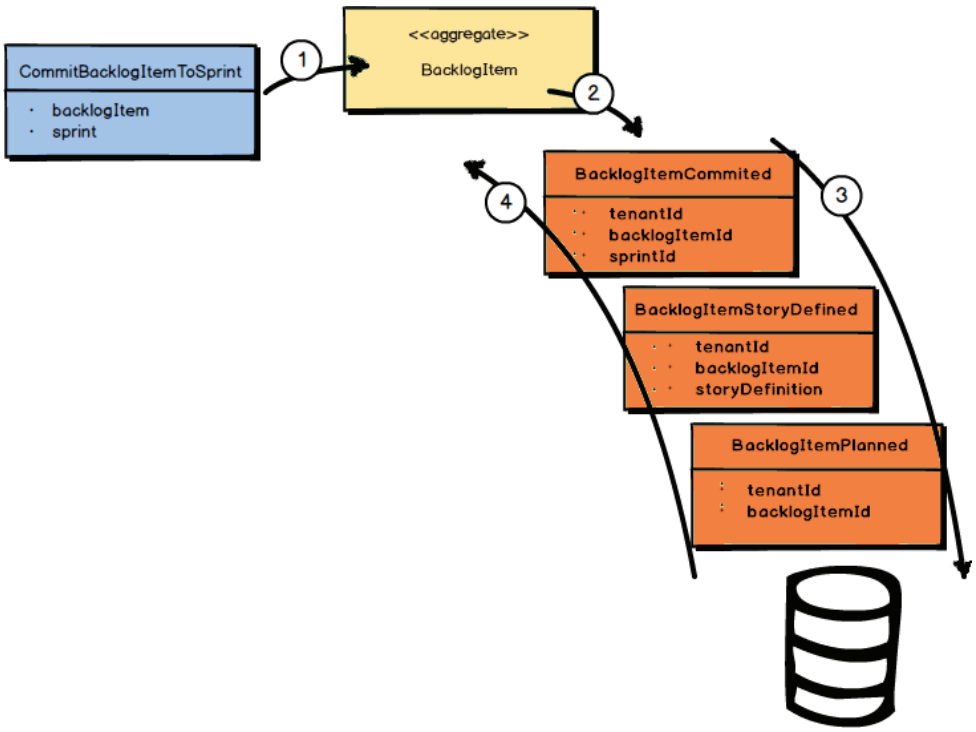
Note that just saving the *Domain Event* in its causal order doesn't guarantee that it will arrive at other distributed nodes in the same order. Thus, it is also the responsibility of the consuming *Bounded Context* to recognize proper causality. It might be the *Domain Event* type itself that can indicate causality, or it may be metadata associated with the *Domain Event,* such as a sequence or causal identifier. The sequence or causal identifier would indicate what caused this *Domain Event,* and if the cause was not yet seen, the consumer must wait to apply the newly arrived event until its cause arrives. In some cases it is possible to ignore latent *Domain Events* that have already been superseded by the actions associated with a later one; in this case causality has a dismissible impact.



One more point about what can cause a *Domain Event* is noteworthy. Although often it is a user-based command emitted by the user interface that causes an event to occur, sometimes *Domain Events* can be caused by a different source. This might be from a timer that expires, such as at the end of the business day or the end of a week, month, or year. In cases like this it won't be a command that causes the event, because the ending of some time period is a matter of fact. You can't reject the fact that some time frame has expired, and if the business cares about this fact, the time expiration is modeled as a *Domain Event,* and not as a command.

What is more, such an expiring time frame will generally have a descriptive name that will become part of the *Ubiquitous Language*. For example, "Fiscal Year Ended" may be an important event that your business needs to react to. Furthermore, 4:00 p.m. (16:00) on Wall Street is known as "Markets Closed" and not just as 4:00 p.m. Therefore, you have a name for that particular time-based *Domain Event*.

A command is different from a *Domain Event* in that a command can be rejected as inappropriate in some cases, such as due to supply and availability of some resources (product, funds, etc.), or another kind of business-level validation. So, a command may be rejected, but a *Domain Event* is a matter of history and cannot logically be denied. Even so, in response to a time-based *Domain Event* it could be that the application will need to generate one or more commands in order to ask the application to carry out some set of actions.



Event Sourcing

Event Sourcing can be described as persisting all *Domain Events* that have occurred for an *Aggregate* instance as a record of what changed about that *Aggregate* instance. Rather than persisting the *Aggregate*

state as a whole, you store all the individual *Domain Events* that have happened to it. Let's step through how this is supported.

All of the *Domain Events* that have occurred for one *Aggregate* instance, ordered as they originally occurred, make up its event stream. The event stream begins with the first *Domain Event* that ever occurred for the *Aggregate* instance and continues until the last *Domain Event* that occurred. As new *Domain Events* occur for a given *Aggregate* instance, they are appended to the end of its event stream. Reapplying the event stream to the *Aggregate* allows its state to be reconstituted from persistence back into memory. In other words, when using *Event Sourcing,* an *Aggregate* that was removed from memory for any reason is reconstituted entirely from its event stream.

In the preceding diagram, the first *Domain Event* to occur was `BacklogItemPlanned`; the next was `BacklogItemStoryDefined`; and the event that just occurred is `BacklogItemCommitted`. The full event stream is currently composed of those three events, and they follow the order described and seen in the diagram.

Each of the *Domain Events* that occurs for a given *Aggregate* instance is caused by a command, just as described previously. In the preceding diagram, it is the `CommitBacklogItemToSprint` command that has just been handled, and this has caused the `BacklogItemCommitted` *Domain Event* to occur.

Stream Id	Stream Version	Event Type	Event Content
backlogItem123	1	BacklogItemPlanned	{ ... }
backlogItem123	2	BacklogItemStoryDefined	{ ... }
backlogItem123	3	BacklogItemCommitted	{ ... }
...	N	...	{ ... }
...	N	...	{ ... }
...	N	...	{ ... }

The event store is just a sequential storage collection or table where all *Domain Events* are appended. Because the event store is append-only, it makes the storage mechanism extremely fast, so you can plan on a *Core Domain* that uses *Event Sourcing* to have very high throughput, low latency, and be capable of high scalability.

Performance Conscious

If one of your primary concerns is performance, you will appreciate knowing about caching and snapshots. First of all, your highest-performing *Aggregates* will be those that are cached in memory, where there is no need to reconstitute them from storage each time they are used. Using the Actor model with actors as *Aggregates* [Reactive] is one of the easier ways to keep your *Aggregates'* state cached.

Another tool at your disposal is snapshots, where the load time of your *Aggregates* that have been evicted from memory can be reconstituted optimally without reloading every *Domain Event* from an event stream. This translates to maintaining a snapshot of some incremental state of your *Aggregate* (object, actor, or record) in the database. Snapshots are discussed in more detail in *Implementing Domain-Driven Design* [IDDD] and in *Reactive Messaging Patterns with the Actor Model* [Reactive].

One of the greatest advantages of using *Event Sourcing* is that it saves a record of everything that has ever happened in your *Core Domain*, at the individual occurrence level. This can be very helpful to your business for many reasons, ones that you can imagine today, such as compliance and analytics, and ones that you won't realize until later. There are also technical advantages. For example, software developers can use event streams to examine usage trends and to debug their source code.

You can find coverage of *Event Sourcing* techniques in *Implementing Domain-Driven Design* [IDDD]. Also, when you use *Event Sourcing* you are almost certainly obligated to use CQRS. You can also find discussions of this topic in *Implementing Domain-Driven Design* [IDDD].

Summary

In this chapter you learned:

- How to create and name your *Domain Events*
- The importance of defining and implementing a standard *Domain Event* interface

- That naming your *Domain Events* well is especially important
- How to define the properties of your *Domain Events*
- That some *Domain Events* may be caused by commands, while others may happen due to the detection of some other changing state, such as a date or time
- How to save your *Domain Events* to an event store
- How to publish your *Domain Events* after they are saved
- About *Event Sourcing* and how your *Domain Events* can be stored and used to represent the state of your *Aggregates*

For a thorough treatment of *Domain Events* and integration, see Chapters 8 and 13 of *Implementing Domain-Driven Design* [IDDD].