a pattern language for

EVENT SOURCING in Python

EVENT-ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS

JOHN BYWATER

a pattern language for

EVENT SOURCING in Python

EVENT-ORIENTED ANALYSIS AND DESIGN
WITH APPLICATIONS

JOHN BYWATER

A Pattern Language for Event Sourcing in Python

Event-oriented analysis and design with applications.

By John Bywater

Contents

Preface

I began writing the main parts of this book (in Parts 1-3) after giving a talk to the Domain-Driven Design London meetup group in November 2019 about event sourced building blocks for Domain-Driven Design, based on my experience of event sourcing in Python. During this talk, I presented a set of patterns for event-sourced Domain-Driven Design based on the design and capabilities of the Python eventsourcing library. I identified fifteen main patterns and the dependencies of each pattern on the others. The directed graph of patterns revealed a linear ordering that would allow the patterns to be discussed one-by-one without the need to refer to patterns which had not already been introduced.

The main chapters of this book follow the structure of that talk, with each of the main chapters describing one of the patterns. These fifteen patterns fell naturally into three groups of five. These three groups were taken as the three main parts of the talk, and have become the main parts of this book. These chapters present the material with more care and with more detail than I was able to provide in the talk. A map showing the graph of dependencies and the linear ordering is included in the final introductory chapter (Overview of the Patterns).

The first two chapters of the book (Prologue, and Whitehead's Scheme) were written following some earlier considerations about the general applicability of event sourcing. The material for these two chapters was developed during 2018 and 2019, and was gradually distilled and refined whilst preparing for presentations given at software conferences in 2018 and 2019.

In the Prologue, I discuss Christopher Alexander's pattern language scheme and its relationship to the modern process philosophy of Alfred North Whitehead. I make a respectful but slightly critical appraisal of how pattern language was taken up in software development, identifying the conceptual distance of classical object-oriented analysis and design from modern process philosophy, and the apparent overlooking of the event-oriented

character of Christopher Alexander's scheme. This conceptual gap left room for the unanticipated turn to event-oriented analysis and design, as expressed by event sourcing, event storming, and event modelling. I argue that the event-oriented approach is generally applicable, on the grounds that Whitehead's scheme is generally applicable. This argument goes against classical object-oriented analysis and design which views the world as built up of instances of substance-quality categories with events having a marginal status, and of the early view of event sourcing that it is only applicable in limited circumstances.

I had initially accepted those views, and was merely seeking how to identify the limits of an event-oriented approach, so that I would be able to tell when event sourcing is not appropriate in any given domain. The surprising result of my search was: to discover Whitehead's scheme; to realise that Alexander's scheme was a creative application of Whitehead's scheme; that there are no domains that are not fundamentally comprised of events; and that the limits of the applicability of event sourcing is determined by the skills of the software developers rather than the nature of the domain they are desiring to support.

In order to provide a logical and coherent conceptual foundation on which to construct a general event-oriented analysis and design approach for software development, a general introduction to modern process philosophy is provided as a second introductory chapter.

An introduction to the notions 'domain', 'domain model', and 'domain event' is presented in the third introductory chapter (Domains and Domain Models). The notions are explained in basic terms, and put on an event-oriented foundation by refering to Whitehead's scheme. This material has been separated from the patterns in the main parts of the book, in order to keep the patterns as concise as possible.

Connections are made with person-centred psychology, which is described in more detail in the final chapter of the book (Epilogue).

Acknowledgements

I am very grateful to Kacper Gunia and the other organisers of the DDD London meetup for inviting me to give the talk about event sourcing in Python. I was greatly encouraged by Kacper's generous remark, "You've come up with a particularly nice narrative and way of structuring the content."

I am similarly grateful for the support of the organisers of the software architecture and Python conferences at which I have spoken about these topics. I would like to express deep gratitude to Robert Smallshire and Mathias Verraes for involving me in the DDD EU conference in 2019, which gave me an opportunity to collect my thoughts about "events in software, process and reality" into a moderately communicable form.

I am especially grateful to Michael Mehaffy, Rebecca Wirfs-Brock, Eric Evans, Ward Cunningham, Jim Coplien, Vaughn Vernon, Grady Booch, and David Heath for detailed discussions during 2019 and 2020, for their interest in these topics, and for the encouragement these discussion have given me to write this book.

I have also greatly appreciated discussions I have been having with various academic philosophers, in particular Prof James R Williams and Dr Peter Sjöstedt-H. Their guidance and conversation about the work of Alfred North Whitehead helped to reassure me in my understanding of Whitehead's work.

I would also like to thank the many companies that have asked me to work with them professionally over the past five years to create event-sourced applications and systems. Without these practical encounters, the Python eventsourcing library would likely not have come into existence, and my sustained interest in these topic would not have been possible.

Last but not least, I would also like to thank the users and supporters of the Python eventsourcing library for their friendliness, expressions of gratitude, contributions, good will, pleasantness, and encouragement.

INTRODUCTION

Prologue

> *"A civilization which cannot burst through its current abstractions is doomed to sterility after a very limited period of progress."*
>
> —Alfred North Whitehead

This book presents a pattern language for event-sourced applications and reliable distributed systems. The patterns are organised in three parts. Part 1 describes how to make an event-sourced domain model. Part 2 describes how to make an event-sourced application. Part 3 describes how to make a reliable distributed system from a set of event-sourced applications.

Since this is a book of patterns, it seems worthwhile firstly to say a few words about what pattern language means to me.

Pattern language

The term 'pattern language' was coined by Christopher Alexander. Alexander's book *The Timeless Way of Building* presents pattern language as a scheme for describing events. Each pattern language is concerned with a particular domain. For example, *A Pattern Language for Towns and Buildings* is concerned with the built environment.

As we shall see, the timeless way is found by describing and becoming familiar with the recurring events that give living character to a world. A pattern language can help us to develop our feelings for its domain of concern, so that we may become receptive to what is real in that domain, and so that any designs we may introduce positively contribute to sustaining those events.

Alexander's novel scheme is most understandable as an application of the event-oriented process philosophy of Alfred North Whitehead. For this reason, Alexander's scheme will be discussed with reference to

Whitehead's scheme. The discussion raises the question 'what is an event?' — a question answered in the next chapter on Whitehead's scheme.

Pattern language is commonly understood as an organised and coherent set of patterns, each of which describes a problem in a context, and the core of a solution. Alexander wrote:

> *"A pattern is a careful description of a perennial solution to a recurring problem within a building context, describing one of the configurations that brings life to a building. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice".*

Pattern language was intended by Alexander to describe the events that make our worlds what they are. Alexander suggested in *The Timeless Way of Building* that places are characterised by recurring patterns of events, and that we must start by recognising the centrality of events in reality.

> *"We must begin by understanding that every place is given its character by certain patterns of events that keep on happening there."*

Alexander explained his idea by referring to particular occasions of experience.

> *"To understand this clearly, we must first recognize that what a town or building is, is governed, above all, by what is happening there. I mean this in the most general sense. Activities; events; forces; situations; lightning strikes; fish die; water flows; lovers quarrel; a cake burns; cats chase each other; a hummingbird sits outside my window; friends come by; my car breaks down; lovers' reunion; children born; grandparents go broke…. My life is made of episodes like this. The life of every person, animal, plant, creature, is made of similar episodes. The character of a place, then, is given to it by the episodes which happen there."*

By explaining places as shaped by what happens there, Alexander echoes Whitehead's first category of explanation: that a world is a process made up of 'actual occasions'.

> *"That the actual world is a process, and that the process is the becoming of actual entities. Thus actual entities are creatures; they are also termed 'actual occasions'."*

Pattern languages describe the events that make up the reality we seek to shape as developers. The atomicity and molecularity of events that we find in Whitehead's scheme — actual occasions of experience and 'societies' of actual occasions — also appears in Alexander's *The Timeless Way of Building*.

> *"Indeed, as we shall see, each building and each town is ultimately made out of these patterns in the space, and out of nothing else: they are the atoms and the molecules from which a building or a town is made."*

Being social and linguistic creatures living in a world made of its events, we experience the need to discuss and define them. Understanding this is vital to our understanding of the world because, as Alexander mentioned, the actual world is made of nothing else.

Alexander is quite definite in his book about the relationship between pattern language and events. Each pattern describes a characteristic event (or episode). The event is a 'thing' in that it can happen, its 'happening' is a process by which the occasion comes to pass, and the description explains how the process works.

> *"The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing."*

We can see in Alexander's writing a definite unity of 'process' and 'thing' which echoes Whitehead's eighth and ninth categories of explanation.

"That two descriptions are required for an actual entity: one which is analytical of its potentiality for 'objectification' in the becoming of other actual entities, and another which is analytical of the process which constitutes its own becoming."

"That how an actual entity becomes constitutes what that actual entity is; so that the two descriptions of an actual entity are not independent. Its 'being' is constituted by its 'becoming'; this is the 'principle of process'."

We can also see in Alexander's writing a distinction between existence and explanation. Since existence and explanation are distinct categories in Whitehead's scheme, we can understand the word 'pattern' is being used by Alexander to mean two different things: the existence of that which is explained; and the explanation of that which exists. As existence, Alexander reflects the idea that each actual occasion is a unique novel advance with the comment that "you can use the solution a million times over, without ever doing it the same way twice". As an explanation, a pattern is a "careful description", but it is speculative and is not the only possible description.

Furthermore, Alexander describes pattern language as a gate that one approaches, moves through, and leaves behind. A pattern language is a means to an end. The end is to become more receptive to what is real.

"To reach the quality without a name we must then build a living pattern language as a gate."

"Once we have built the gate, we can pass through it to the practice of the timeless way."

"And yet the timeless way is not complete, and will not fully generate the quality without a name, until we leave the gate behind."

"It is the gate which leads you to the state of mind, in which you live so close to your own heart that you no longer need a language."

"At this final stage, the patterns are no longer important: the patterns have taught you to be receptive to what is real."

What is real are actual occasions of experience. One occasion of experience relates to others through its feelings. Each of its feelings makes a contribution to the occasion of experience, and the occasion of experience is made of the feelings that it feels.

A pattern language describes events and gestures towards what there is to feel within a particular domain. By moving through a pattern language, we can practise feeling what has been felt by its authors and become familiar and more receptive to what is real in its domain. Since feelings condition creativity, by conditioning our feelings a pattern language can help us to become more creative and therefore more alive. This is the intuitive appeal of any pattern language.

Similarly, modern process philosophy can help us to develop our conceptual feeling for events in general. As I will try to explain in the next chapter on Whitehead's scheme, by understanding what is an event we can develop a more adequate grasp of the universe we inhabit as living creatures. This is the intuitive appeal of modern process philosophy.

Both pattern languages and process philosophy are important for developers and architects to understand. Once we recognise that pattern languages describe events, we can recognise the importance of obtaining an adequate conceptual feeling for events, and become more deeply receptive to what is real.

Understood in this way, a pattern language encourages a process of empathy, of listening through words and other kinds of expression to understand the feelings that are being felt, to identify the needs that are or are not being met, and to develop and apply strategies by which those needs can be met. Pattern language appears comparable with the moment-by-moment empathy that is encouraged by Marshall Rosenberg's scheme for collaborative or 'nonviolent' communication. Marshall Rosenberg said nonviolent communication was not just about meeting human needs, but about meeting the needs of all the phenomena on the planet. Christopher Alexander's later work, *The Nature of Order*, has a similarly broad scope of

concern. Pattern language functions effectively as empathetic nonviolent communication, and nonviolent communication functions effectively as a pattern language. The coherence of pattern language and nonviolent communication is no mere coincidence since both are process philosophic approaches to life.

The fact that Alexander's pattern language scheme was intended to describe events has not been widely appreciated in the software development community. What seems even less well-known is that Christopher Alexander was enormously influenced by the work of Alfred North Whitehead.

Design patterns

The influence of Alfred North Whitehead's philosophy on Christopher Alexander's work, and the centrality of events to both pattern language and modern conceptions of reality, has been almost entirely overlooked by the authors of the well-known software pattern languages and design pattern books.

Alexander expressed some dissatisfaction with the way his work was adopted by some of the early adopters of pattern language in software development. The early adopters also expressed some difficulties reaching satisfaction with their work. It seems to me, these difficulties follow from overlooking Whitehead's scheme and its influence on the work of Alexander. The possibility of applying Whitehead's scheme directly in software development was not considered.

The problem with understanding patterns merely as solutions to problems, as a communication technique rather than as an event, is that the question 'what is an event?' does not arise, and does not receive an answer. The solution to this problem is to recognise the centrality of events in Alexander's work, and to seek to understand and apply the modern concept of the event that we find in Whitehead's scheme.

An outstanding example of successfully apprehending the impression made by Whitehead's scheme on the work of Alexander is Ward Cunningham's

Episodes. Episodes is a pattern language for software development which is an acknowledged precursor to Extreme Programming (XP). It was largely ignored and is today almost entirely unknown.

Cunningham was the first to appreciate the potential of applying Alexander's pattern language scheme in software development. The following quote from the introduction to Episodes brilliantly expresses the core concept at the heart of Whitehead's scheme: the creative becoming of an actual occasion; a decision that takes place in the context of existing facts and leaves behind new additional facts which may be taken up by subsequent occasions. Cunningham wrote:

> *"An episode builds toward a climax where the decision is made. Before the decision, we find facts, share opinions, build concentration and generally prepare for an event that cannot be known in advance. After the climax, the decision is known, but the episode continues. In the tail of an episode we act on our decision, promulgate it, follow it through to its consequences. We also leave a trace of the episode behind in its products. It is from this trace that we must often pick up the pieces of thought in some future episode."*

Since we cannot know what will be decided in a creative process of development, the primary function of software design patterns will be to condition occasions of design. The common idea that patterns allow developers to communicate using well-known names for well-understood tropes is interesting and useful, but the view that patterns aid communication overlooks the creative process of becoming that distinguishes a creative occasion of design event from an act of copying-and-pasting. It is the creative process of finding a unity or harmony or settlement across a selection of feelings that results in something new being created. As described in Whitehead's scheme, creativity is found only under particular conditions, and design patterns contribute to those conditions.

Martin Fowler's book *Refactoring* is a splendid book of patterns that condition occasions of improving existing code. Its notion of a 'code smell' is as good an expression of feeling for incoherence and reaching for a better settlement as anything that has been written in the domain of software development. In Eric Evans's book *Domain-Driven Design* the practice of

reaching for a ubiquitous language and modelling out loud is, similarly, a process of feeling and of the becoming of something new.

Evans' book is a very good example of how pattern language has been applied to describe software architecture, not least because Ward Cunningham encouraged Eric Evans to remain as close as possible to his experience. Although *Domain-Driven Design* does not include a discussion of event sourcing, the Cargo Shipping example in the middle of the book does have cargo handling events modelled as domain entities that do not change. The strange but necessary involvement of atomic database transactions in the book's discussion about the domain layer, which is explicitly cast as being entirely independent of any infrastructure layer, constructs the useful notion of a 'consistency boundary'. A consistency boundary means that the potentially many changes to the state of an aggregate that result from the execution of a command are not divisible in their recording. Without this atomicity, the recorded state of the aggregate may become inconsistent. The event of changing the recorded state of an aggregate in response to executing a command is by design, to use Whitehead's term, an actual occasion. And because these occasions of recording are effectively serialised, so that they occur one after the other without branching the state, they are ordered serially. As we shall see in the next chapter on Whitehead's scheme, this is a kind of order known as 'personal order'. It was considerations such as these which meant that Domain-Driven Design as a movement of thought and practice, absorbed without difficulty the later developments of event sourcing and event storming. However, the actual occasions of the aggregates in themselves were not generally analysed or designed as events. This is a matter that will be discussed in more detail in Chapter 1 and Chapter 2.

By contrast with the growing interest in *Domain-Driven Design* , Martin Fowler's important book *Patterns of Enterprise Application Architecture* with its stronger focus on domain objects and mappers of various kinds, in a magnificent struggle with the "impedance mismatch" between an object-oriented domain model and a relational database management system, seems to have become much less interesting to people over the years. Nevertheless, these patterns are still important, and some of them figure in this book.

Unfortunately, software design patterns were mostly not, at least at that time, conceived as Alexander had explicitly intended them: as events, as both explanation and existence, as both process and thing, in other words as actual occasions of experience, as the atoms of structured or living societies which a pattern language seeks to describe. Yet, the word 'event' appears in *The Timeless Way of Building* more than one hundred times.

We can see from remarks by Grady Booch in the February 2002 issue of MSDN Magazine, that the patterns of pattern language were taken up in software engineering by some as an abstraction from UML diagrams which were an abstraction from object-oriented software, but the general notion of 'event' and the more specific notion of an 'actual occasion' seems to be missing.

> *"From a technical perspective, I think there are several key understandings that have happened over the last five years or so, in this order: the first is the mainstreaming of object-oriented technology; the second is the creation of UML, which has given us a basis for a common language and a blueprint for software, and last, the notion of software patterns as best typified by the work of the Gang of Four. What is interesting about patterns is that they represent a step up in the next level of abstraction. Indeed, if you look across the whole history of software engineering, it's one of trying to mitigate complexity by increasing levels of abstraction. And object-oriented technology followed by design patterns is indeed that natural trend."*

It seems to be a simple historical fact that these great developers and most popular authors were not aware of the indirect illumination of their work provided by the work of Alfred North Whitehead. This situation was anticipated by Henry Nelson Wieman in his 1930 review of Whitehead's book *Process and Reality*.

> *"Not many people will read Whitehead's recent book in this generation; not many will read it in any generation. But its influence will radiate through concentric circles of popularization until the common man will think and work in the light of it, not knowing whence the light came. After a few decades of discussion and*

analysis one will be able to understand it more readily than can now be done."

Our not knowing that Whitehead's scheme was operating within Alexander's work has resulted in a situation that can be described as 'cargo culting'. The sadness of cargo culting is that one does not have a way of knowing that something important is missing from one's understanding, except perhaps for vague feelings of dissatisfaction. The real consequence is that object-oriented analysis and design, the practice of writing patterns, and the organisation of software development have all operated a degree of abstraction away from what is warranted. There is no question that thought depends on abstractions. What is important is that the predominant mode of abstraction active in our general understanding of things is generally adequate and applicable.

The two-hierarchies scheme

If events were not the predominant mode of thought amongst the thought-leaders of objected-oriented programming, then what was? Let us briefly consider The Canonical Form of a Complex System in Chapter 1 of Grady Booch's masterpiece *Object-Oriented Analysis and Design*, which offers examples taken from biology, physics, technology, and social institutions. Booch wrote:

"Plants consist of three major structures (roots, stems, and leaves), and each of these has its own structure. For example, roots encompass branch roots, root hairs, the root apex, and the root cap. Similarly, a cross-section of a leaf reveals its epidermis, mesophyll, and vascular tissue. Each of these structures is further composed of a collection of cells, and inside each cell we find yet another level of complexity, encompassing such elements as chloroplasts, a nucleus, and so on. As with the structure of a computer, the parts of a plant form a hierarchy, and each level of this hierarchy embodies its own complexity."

"The study of fields as diverse as astronomy and nuclear physics provides us with many other examples of incredibly complex systems. Spanning these two disciplines, we find yet another structural hierarchy. Astronomers study galaxies that are arranged in clusters, and stars, planets, and various debris are the constituents of galaxies. Likewise, nuclear physicists are concerned with a structural hierarchy, but one on an entirely different scale. Atoms are made up of electrons, protons, and neutrons; electrons appear to be elementary particles, but protons, neutrons, and other particles are formed from more basic components called quarks."

"As a final example of complex systems, we turn to the structure of social institutions. Groups of people join together to accomplish tasks that cannot be done by individuals. Some organizations are transitory, and some endure beyond many lifetimes. As organizations grow larger, we see a distinct hierarchy emerge. Multinational corporations contain companies, which in turn are made up of divisions, which in turn contain branches, which in turn encompass local offices, and so on. If the organization endures, the boundaries among these parts may change, and over time, a new, more stable hierarchy may emerge."

The central claim of Chapter 1 of *Object-Oriented Analysis and Design* is that the complex systems we encounter in everyday life can be adequately characterised as wholes made of parts (the 'part of' hierarchy), with the wholes and the parts understood as instances of kinds of things (the 'is a' hierarchy). I will refer to this arborescent characterisation as the 'two-hierarchies scheme'.

"For example, an aircraft may be studied by decomposing it into its propulsion system, flight-control system, and so on. This decomposition represents a structural, or 'part of' hierarchy. Alternately, we can cut across the system in an entirely orthogonal way. For example, a turbofan engine is a specific kind of jet engine, and a Pratt and Whitney TF30 is a specific kind of turbofan engine. Stated another way, a jet engine represents a generalization of the properties common to every kind of jet engine; a turbofan engine is

simply a specialized kind of jet engine, with properties that distinguish it, for example, from ramjet engines."

"This second hierarchy represents an 'is a' hierarchy. In our experience, we have found it essential to view a system from both perspectives, studying its 'is a' hierarchy as well as its 'part of' hierarchy. For reasons that will become clear in the next chapter, we call these hierarchies the class structure and the object structure, respectively."

These structures are described as an economic statement of an expressive God.

"In a computer, we find NAND gates used in the design of the CPU as well as in the hard disk drive. Likewise, a considerable amount of commonality cuts across all parts of the structural hierarchy of a plant. This is God's way of achieving an economy of expression."

In addition to a divine process of expression, other processes are acknowledged. However, these processes are not explicable as 'part of' and 'is a' hierarchies. The parts may change but the behaviours are preordained by their types. The notion of emergence is stated as wholes being greater than the sum of their parts, but since the parts are only parts because they are parts of the whole, how the whole comes to be greater than the sum of the parts isn't explained.

"In studying the morphology of a plant, we do not find individual parts that are each responsible for only one small step in a single larger process, such as photosynthesis. In fact, there are no centralized parts that directly coordinate the activities of lower level ones. Instead, we find separate parts that act as independent agents, each of which exhibits some fairly complex behavior, and each of which contributes to many higher-level functions. Only through the mutual cooperation of meaningful collections of these agents do we see the higher-level functionality of a plant. The science of complexity calls this emergent behavior: The behavior of the whole is greater than the sum of its parts."

There is a very interesting quote in Section III Applications of *Object-Oriented Analysis and Design* which concerns 'basic phenomena'. It warns about mistaken abstractions, and advises that we remain with particular examples:

> *"To build a theory, one needs to know a lot about the basic phenomena of the subject matter. We simply do not know enough about these, in the theory of computation, to teach the subject very abstractly. Instead, we ought to teach more about the particular examples we now understand thoroughly, and hope that from this we will be able to guess and prove more general principles."*

Having abstractions is important and useful, and I do not disagree with Grady Booch's observation that, "In many ways, the entire history of software engineering can be seen as one of raising levels of abstraction." The trouble seems to have been that the non-abstract, real actuality that these levels of abstraction are abstracting from wasn't clearly identified. It hasn't been clear whether it is "abstractions all the way down", or whether it is the "tangible" objects were supposed to be the ground from which the abstractions are made, and if so then what is the status of an event? It has been very hard to obtain a settled understanding.

Altogether, the analysis of the examples in Chapter 1 of *Object-Oriented Analysis and Design* under the two-hierarchies scheme appears as an application of substance-quality categories.

Substance-quality categories are categories of classical Greek thought. They are both inadequate as explanations and mistaken for real actuality. The 'presentational immediacy' of the ordinary physical objects in the conscious occasions of experience is taken as the actual reality of the things that are perceived. The degree of abstraction our perception of these objects depends on is overlooked.

The label Whitehead gave to this mistake is the 'fallacy of misplaced concreteness'. The desire to generalise from our experiences of the physical objects we encounter in the world is to be admired, but there are aspects of existence and of explanation that are not included in the two-hierarchies scheme. Whitehead described the substance-quality categories as an

incubus, and suggested the use of the substance-quality categories should be avoided by any adequate conceptual scheme. Whitehead put it like this:

> *"The aim at generalization is sound, but the estimate of success is exaggerated. There are two main forms of such overstatement. One form is what I have termed, elsewhere, the 'fallacy of misplaced concreteness'. This fallacy consists in neglecting the degree of abstraction involved when an actual entity is considered merely so far as it exemplifies certain categories of thought. There are aspects of actualities which are simply ignored so long as we restrict thought to these categories. Thus the success of a philosophy is to be measured by its comparative avoidance of this fallacy, when thought is restricted within its categories."*

> *"The current accounts of perception are the stronghold of modern metaphysical difficulties. They have their origin in the same misunderstanding which led to the incubus of the substance-quality categories. The Greeks looked at a stone, and perceived that it was grey. The Greeks were ignorant of modern physics; but modern philosophers discuss perception in terms of categories derived from the Greeks."*

The particular individuality of a particular thing can be adequately understood only by considering its particular history within its particular world. This is also what distinguishes the person-centred psychology of Carl Rogers and Marshall Rosenberg from the identification with personality types and the diagnosis of types of personality disorder. To illustrate this point, Whitehead discusses the case of Rome and Europe.

> *"The complex nexus of ancient imperial Rome to European history is not wholly expressible in universals. It is not merely the contrast of a sort of city, imperial, Roman, ancient, with a sort of history of a sort of continent, sea-indented, river-diversified, with alpine divisions, begirt by larger continental masses and oceanic wastes, civilized, barbarized, christianized, commercialized, industrialized. The nexus in question does involve such a complex contrast of universals. But it involves more. For it is the nexus of that Rome with that Europe. We cannot be conscious of this nexus purely by the*

aid of conceptual feelings. This nexus is implicit, below consciousness, in our physical feelings. In part we are conscious of such physical feelings, and of that particularity of the nexus between particular actual entities. This consciousness takes the form of our consciousness of particular spatial and temporal relations between things directly perceived."

Whitehead's gesture towards "things directly perceived" is echoed by Christopher Alexander in *The Timeless Way of Building* with the comment that after we move through a pattern language, after we leave the gate behind, the patterns are "no longer important" because they have "taught you to be receptive to what is real". The problem with using the two-hierarchies scheme proposed in *Object-Oriented Analysis and Design* as the canonical form of a complex system is that it does not teach us to be receptive to what is real.

Object-Oriented Analysis and Design is an outstanding and highly influential example of how the foundational and now prevailing conceptions in software development have fallen back onto substance-quality categories.

The two-hierarchies scheme fails as a canonical form because it does not reduce things to their most basic form. It is not the general rule, because it is not general enough. The two-hierarchies scheme is neither an adequately general nor a generally applicable description of things. Could the two-hierarchies scheme be used to explain poetry or painting, novels or cinema, psychology or the history of philosophy? By thinking about the world in terms of the substance-quality categories, thought itself becomes restricted. We misunderstand biology, physics, technology, and social institutions by thinking about them through the two-hierarchies scheme. What evades the two-hierarchies scheme is the adequate conception of events that we find in the modern process philosophy of Alfred North Whitehead.

There is nothing essential about viewing a system from the perspective of the two-hierarchies scheme. The use of substance-quality categories as a proposal for the canonical form of a complex systems in analysis and design is a mistake.

The wrong objects

The term ‘event’ is not listed as a primary term of importance in the index of either Fowler’s or Evans’ books, or the *Design Patterns* book, or *Object-Oriented Analysis and Design*. The term is certainly present in these books. For example, the State pattern in the book *Design Patterns* concerns changes of behaviour, and state machines in *Object-Oriented Analysis and Design* are discussed in terms of events. Booch’s book defines ‘event’ in the glossary as an occurrence which causes the state to change. The term ‘event’ is used in many places in *Object-Oriented Analysis and Design*, and in many different ways for example as integration events in the development process, as the members of a sequence that algorithmic programming emphasises, as the things that a software system responds to or handles or is interrupted by, as elements in the history of the development of ideas in computer programming. The notion of an object in *Object-Oriented Analysis and Design* receives a rich variety of definitions. Software objects are defined in a range of ways, including a general unit of concern. Nevertheless, the consideration of non-software objects often leans towards “tangible” enduring and changing objects, which are considered to be “real”, and events are put to the side, and considered separately as things which somehow happen *to* these objects, rather than as the things which the tangible objects are built up of. The notion of a subject is also brought in. However, for all the variety and richness, as far as I can see the “tangible” or visible objects themselves were *never* considered or classified or analysed or designed as events. There is a definite style of thinking in the two-hierarchies scheme, but it is a style of thinking that remaining unaffected by modern process philosophy. In contrast with Cunningham’s work, this style of thinking does not apprehend the impression made by Whitehead’s scheme on the work of Christopher Alexander.

The two-hierarchies scheme is not a concept of the creative becoming of actual occasions of experience. In the two-hierarchies scheme, the objects are not processes, the processes are not objects, and the relation between object and process is missing. By contrast, in both Whitehead’s and Alexander’s schemes, there is an identity between process and object. An actual occasion (and a pattern in a pattern language) is a selective process of

synthesis that decides a thing, and every thing that exists is the result of its decisive process of selection and synthesis that makes it what it is.

The incoherence of using pre-modern concepts in the modern world of science and technology is real, and has been detrimental. Software development has modelled the world and itself as being made up of instances of substance-quality categories, rather than events. The deployment of substance-quality categories in the formative considerations of software development has inhibited software developers from thinking adequately about events. Due to the way thought becomes restricted by investing in the substance-quality categories, feelings that something isn't quite right tend to be suppressed rather than pursued.

The difficulties of making distributed systems reliable, of understanding and applying pattern language, and of making development processes habitable all follow directly from the mistake of conceiving of the world as filled with instances of substance-quality categories, rather than as built up of actual occasions.

In contrast to this view, as an enhancement to previous considerations, as an addition to the history of development of the ideas, the proposition of event-oriented analysis and design is that every actual thing is an event. The atoms of our actual world are the actual occasions (decisions) which do not change, which may be superseded like an erroneous financial transaction may be corrected by another financial transaction but is not itself changed. And every actual thing that may be considered to exist is derived by abstraction from these atoms, just like the balance of an account is derived from its transactions, as societies of actual occasions which are events that are not actual occasions, except in the limiting case where a society has only one member. This is the canonical form through which domains and domain models are considered in event-oriented analysis and design.

It was illuminating to hear from Ward Cunningham that during the early days of object-orientated programming the conception of objects was such that events were given hardly any attention. Alan Kay was perhaps right to steer attention away from the object class of object-oriented programming, if it was encouraging the world to be considered through substance-quality categories. But his paying attention to messaging instead of object classes

did not adequately redeem the situation. Messaging is not the "big idea". The big idea is events, understood as both actual occasions and societies of actual occasions. After characterising software objects with the passing of messages, distributed systems were also characterised by their passing of messages. The dual writing problem — writing to a database in one transaction followed by writing a message to a queue as a separate operation which may fail independently of the database transaction — is still prevalent today. The simple reason why dual writing is a problem is that it is an event that is not an actual occasion. The solution to this problem, as Vaughn Vernon described in his book *Implementing Domain-Driven Design*, is to write the messages into the database in the same atomic transaction as the state update. Software objects are useful, object classes are useful, and messaging is useful. But the best way to understand all of these is to consider the events, and to decide and define actual occasions.

The difference between object-oriented programming and functional programming is sometimes exaggerated as an opposition. Both function to determine actual occasions. With a more adequate notion of the meaning of 'function' OOP and FP can be more easily understood as complimentary. Consider Whitehead's definition of the verb 'to function':

> *"That to 'function' means to contribute determination to the actual entities in the nexus of some actual world. Thus the determinateness and self-identity of one entity cannot be abstracted from the community of the diverse functionings of all entities.*
> *'Determination' is analysable into 'definiteness' and 'position,' where 'definiteness' is the illustration of select eternal objects, and 'position' is relative status in a nexus of actual entities."*

Pattern language has had a marginal status in the daily life of many software developers because pattern language is not understood as a description of events and our work is not properly understood as an event. For a long time, I also did not appreciate that pattern language was intended to be descriptions of events, and I did not know about the influence of Whitehead's process philosophy on Christopher Alexander. I discovered the work of Alfred North Whitehead by seeking to identify how applicable is

the consideration of the events in a domain. Is event sourcing only something to be used in special cases? When in particular should it be avoided? It was only by understanding Whitehead's work that I realised what Christopher Alexander had actually done. The assistance of Michael Mehaffy was extremely useful in confirming that Whitehead was "enormously influential" for Alexander, that "Alexander may be Whitehead operationalized", that patterns correspond to actual occasions, and that pattern language corresponds to nexus of actual occasions. James Coplien was very helpful in pointing to particular examples of the prominence of events in *The Timeless Way of Building*. And it was illuminating to hear from Ward Cunningham that when Alexander's pattern language ideas were first applied in software development, the importance and relevance of Whitehead's scheme was entirely unknown.

Problems with the contemporary experiences of agile software development are sometimes blamed on a dilution of the original conception and sometimes excused by saying that agile is in fact nothing new. The real problem, however, appears to be more fundamental. The manifesto for agile software development sought to generalise and explain a variety of incremental and iterative approaches to software development. There are traces of actual occasions in the manifesto ('couple of weeks', 'couple of months', 'a plan', …) and the manifesto itself was an event. But the manifesto for agile software development is not an explicit explanation of creative eventuation, and it does not characterise the incremental and iterative approaches as built up of actual occasions. In contrast, most of the iterative and incremental approaches to software development are indeed proposed as societies of actual occasions that decide now what will happen in the future, for example Scrum events and the loops in XP. The iterative and incremental approaches explain themselves as built up of discrete, creative, actual occasions of experience (decisions). Even the end of a "fixed timebox" is a decision, albeit one that terminates the becoming of a new settlement in code that has not yet been found (and might not be). As Ward Cunningham wrote in Episodes:

> *"Programming is the act of deciding now what will happen in the future."*

However, the manifesto for agile software development is characterised by an overwhelming use of the present continuous tense ('uncovering', 'developing', 'helping', 'responding', 'changing', 'conveying', 'self-organizing'). The central problem with the manifesto for agile software development is that it characterises software development not as an event, but rather as a continuous process. The manifesto for agile software development does not summarise the iterative and incremental approaches to software development as built up of events. It does not explain that software development is "given its character by certain patterns of events" as Episodes did. To use Alexander's words, it therefore does not teach us to be "receptive to what is real".

As a characterisation and summary explanation of the iterative and incremental approaches to software development, the concept of an actual occasion of experience is missing. The text of the manifesto does not use words such as 'event' or 'decision' or 'episode'. In the name of agile, with thought conditioned by prevailing pre-modern substance-quality categories, software development work is assessed by the extent to which the substance of a project (or team, or organisation) has the quality of being agile ('agility'). Software development has been conceived as an instance of a continuous process (continuous attention, continuous integration, continuous delivery, continuous deployment, continuous development, a constant pace indefinitely) rather than as an unfolding series of irregular episodes of creative decision making that "cannot be known in advance". As Whitehead stated in *Process and Reality*, there is no continuity of becoming, only the becoming of continuity.

> *"There is a becoming of continuity, but no continuity of becoming. The actual occasions are the creatures which become, and they constitute a continuously extensive world. In other words, extensiveness becomes, but 'becoming' is not itself extensive. Thus the ultimate metaphysical truth is atomism. The creatures are atomic."*

Under these conditions, creativity has tended be squeezed out, and a world that can only be built up of novel, creative, discrete occasions — in which an incoherence is felt and a better settlement found in the working software

or the software development work — has tended to become barren and uninhabitable. In a talk given in 2011, Dave Snowden said:

> *"Part of the big problem with the whole Scrum/Agile movement is you've got some great practices but you haven't worked out the underlying theory yet, and until you work out the underlying theory it won't scale and it won't have bite, other than in isolated cases."*

Since then, it appears that things have got worse. When Kent Beck was recently asked how he feels about agile now, he replied:

> *"It's a devastated wasteland. The life has been sucked out of it. It's a few religious rituals carried out by people who don't understand the purpose that those rituals were intended to serve in the first place."*
>
> *"Compassion is a habit not a quality."*

Patterns of process were supposed to be a description that "brings life". Software developers need empathy and understanding, for software development involves a creative feeling for a deeper satisfaction in the person, the process and the software. For those who wish to start over with the process patterns of software development, the patterns of collaborative or nonviolent communication (observations, feelings, needs, requests, gratitude) may offer a humane foundation for approaching software development that can maximise the chances that everybody's needs will be met. Cunningham's Episodes could be revived. For what everybody needs, and what everybody actually has, is an eventual approach.

Event-oriented analysis and design

In general, the solution to the problem of falling back onto the substance-quality categories is to say that the canonical form of complex systems is that of events. Hierarchies of types and instances can occur and can be useful, but they are examples of things that can happen: they are examples of events and not the general rule.

Intuitive answers to the difficulties that have arisen from the application of substance-quality categories in software development have been found by turning to a consideration of the events in a domain. The invention of event sourcing by Greg Young and the invention of event storming by Alberto Brandolini are the best examples of how the underlying order of nature was intuited again by software developers.

We can follow up on this intuition by carefully understanding the modern process philosophy set out in *Process and Reality*. We can understand why the new techniques that focus on events are generally adequate and applicable. Event sourcing is not the only option for persisting the state of an application. However, non-event-sourced systems are most usefully analysed and designed in terms of their actual occasions.

Writing a pattern language for event-sourced applications and systems provides an opportunity to confront this old oversight.

We can understand that Whitehead's scheme provides the missing context for understanding Alexander's scheme. Alexander's scheme can be explained by using Whitehead's scheme, and becomes more meaningful by doing so. By understanding Alexander's scheme as an application of Whitehead's scheme, Alexander's scheme becomes more understandable and can be adopted and applied with more confidence.

By understanding that design patterns condition occasions of design, by realising that designs condition events ("and nothing else"), and by obtaining a more adequate concept of events, we can apply the original ideas of pattern language with more commitment, carry our designs further, and create better settlements that give deeper satisfaction.

We can also better understand how Alexander's scheme can be varied. Alexander's scheme can be varied so long as it remains consistent with Whitehead's scheme. When writing a new pattern language we can faithfully follow Alexander's scheme and describe the context-problems, the discussion-solutions, the solution-diagrams, the archetypal examples, and link each pattern to related patterns. But we can also do what Alexander seems to have done, and employ Whitehead's scheme directly.

By understanding that objects are made by subjects in a creative process of subjective objectification of other subjects, we can move beyond object-orientation to a more fundamental and more adequate canonical form.

Moving to an event-oriented analysis and design also allows a congruence between the analysis and design of software systems and the moment-by-moment humanist psychology of Carl Rogers and others, which, with Whitehead's scheme as the foundation, radically unifies and therefore simplifies the understanding of things in general that we need to understand the broad range of subjects and objects in the domains supported by software development, and in the domain of software development itself.

Whitehead's Scheme

The purpose of this chapter is to provide a short introduction to Whitehead's scheme. Whitehead's scheme is sophisticated, and in being selective I have focussed on an 'event' concept that is broad enough to include our personal thoughts and feelings, our moment-by-moment relations with other people, the life-bringing structures Christopher Alexander wanted to propagate with his pattern language scheme; as well as the domain event objects that we find in software development, other software objects such as the aggregates of Domain-Driven Design, a piece of working software as a whole, a team of software developers, and so on.

There is a habit in software development of referring to domain event objects as 'events', but this habit seems to make it more difficult to think of other things as events. The aim of this introduction is to instil a more adequate conceptual scheme by which the 'domain event object' of software development comes to be understood as a relatively simple and particular thing, and the term 'event' comes to be understood as a much broader concept that is adequate and applicable as the canonical form of simple and complex systems, of technology and living things, and of the considerations we have of such things.

An important building block in the design and development of software systems, and the processes they support, is our general understanding of the world in which we live. We cannot avoid using our general understanding of things whenever we engage with others in an effort to meet complex needs. For this reason, engineers find themselves, at least from time to time, interested in philosophy and psychology. However, obtaining an adequate grasp of philosophy and psychology, especially one that makes sense in simple terms, is congruent with our intution about the world, and is coherent with effective approaches to design and development, can be elusive. Whitehead's event-oriented scheme is appealing because it offers a modern, adequate, applicable, coherent, logical understanding of the world. Whitehead's scheme is worth understanding because its established uses, in architecture and psychology, have already informed our methods and

illuminated our experiences. We have already been thinking and working in the light of Whitehead's scheme. So it makes good sense to understand Whitehead's scheme directly and to see from whence this light came.

The core of Whitehead's scheme, of worlds built up of objects that do not change, will be immediately recognisable by any software developer acquainted with the persistence mechanism for Domain-Driven Design known as event sourcing. Although Whitehead's scheme predates electronic computers and computer software by several decades, it can be used to understand why tackling the complexity in a domain by its events is universally applicable. Whitehead's scheme is attractive because it offers a conceptual framework to express "any possible interconnection of things" that is generally adequate and applicable. Christopher Alexander's pattern language scheme, originally intended as a description of events, is appealing as a general approach to design and architecture for this reason.

Process philosophy

Alfred North Whitehead's modern process philosophy was described most fully in his masterpiece *Process and Reality*. Whitehead's philosophic scheme was first presented in a series of lectures given at Harvard University in 1927 and 1928. Whitehead's book *Process and Reality*, published in 1929, presents the content of those lectures.

Background

Whitehead's scheme is the product of a lifetime of study and thought. Whitehead read the Greeks in Greek when he was a boy. In *Process and Reality*, Whitehead brought forward ideas from many philosophers, and discarded ideas and interpretations which appeared to him as unreasonable mistakes.

> *"There is no point in endeavouring to force the interpretations of divergent philosophers into a vague agreement. What is important is that the scheme of interpretation here adopted can claim for each of its main positions the express authority of one, or the other, of some*

supreme master of thought — Plato, Aristotle, Descartes, Locke, Hume, Kant. But ultimately nothing rests on authority; the final court of appeal is intrinsic reasonableness."

Process philosophy in European thought is considered to date back to the ancient Greek philosopher Heraclitus, with the notion that 'everything flows' and the idea that 'you cannot step twice into the same stream'. In *Process and Reality*, Whitehead generalises the idea of a flowing river, to the thoughtful thinker, and to the experiencing subject.

"The ancient doctrine that 'no one crosses the same river twice' is extended. No thinker thinks twice; and, to put the matter more generally, no subject experiences twice."

Whitehead was inspired above all by Plato, and in particular Plato's *Timaeus*.

"This conception of an actual entity in the fluent world is little more than an expansion of a sentence in the Timaeus: 'But that which is conceived by opinion with the help of sensation and without reason, is always in a process of becoming and perishing and never really is.'"

Whitehead's book *Process and Reality* contains what is perhaps Whitehead's most famous quote, which characterises European philosophy as a series of footnotes to Plato.

"The safest general characterization of the European philosophical tradition is that it consists of a series of footnotes to Plato."

Overview

Modern process philosophy is largely attributed to Whitehead's philosophic scheme. The name Whitehead gave to his scheme is the 'philosophy of organisim'.

"These lectures are based upon a recurrence to that phase of philosophic thought which began with Descartes and ended with

Hume. The philosophic scheme which they endeavour to explain is termed the 'Philosophy of Organism'. There is no doctrine put forward which cannot cite in its defence some explicit statement of one of this group of thinkers, or of one of the two founders of all Western thought, Plato and Aristotle. But the philosophy of organism is apt to emphasize just those elements in the writings of these masters which subsequent systematizers have put aside."

Whitehead identifies his scheme as a 'cosmological scheme' which means he is concerned with ultimate generalities about the order of nature in the universe. The philosopher Pythagoras first used the term 'cosmos' to mean the 'order of the universe'. Using the word cosmos rather than the word universe implies viewing the universe as a complex and orderly system or entity; the opposite of chaos. A cosmological scheme presupposes that degrees of order are at least attainable. Whitehead contrasts 'order' with 'disorder', such that things can be more or less ordered, and also that different kinds of order can be distinguished.

Whitehead describes his cosmological scheme as a 'speculative philosophy' — an essay which seeks to approach universal principles whilst acknowledging both: the limitations of language and imagination; and the fact that one can only start from the particular occasions of experience one has, and then proceed towards ultimate generalities by noticing differences between them.

"Philosophers can never hope finally to formulate these metaphysical first principles. Weakness of insight and deficiencies of language stand in the way inexorably. Words and phrases must be stretched towards a generality foreign to their ordinary usage; and however such elements of language be stabilized as technicalities, they remain metaphors mutely appealing for an imaginative leap."

"There is no first principle which is in itself unknowable, not to be captured by a flash of insight. But, putting aside the difficulties of language, deficiency in imaginative penetration forbids progress in any form other than that of an asymptotic approach to a scheme of principles, only definable in terms of the ideal which they should satisfy."

> *"The difficulty has its seat in the empirical side of philosophy. Our datum is the actual world, including ourselves; and this actual world spreads itself for observation in the guise of the topic of our immediate experience. The elucidation of immediate experience is the sole justification for any thought; and the starting-point for thought is the analytic observation of components of this experience. But we are not conscious of any clear-cut complete analysis of immediate experience, in terms of the various details which comprise its definiteness. We habitually observe by the method of difference. Sometimes we see an elephant, and sometimes we do not."*

The central and novel idea of Whitehead's modern process philosophy is: the actual world is built up of occasions of experience (the "actual entities)".

> *"'Actual entities' — also termed 'actual occasions' — are the final real things of which the world is made up. There is no going behind actual entities to find anything more real. They differ among themselves: God is an actual entity, and so is the most trivial puff of existence in far-off empty space. But, though there are gradations of importance, and diversities of function, yet in the principles which actuality exemplifies all are on the same level. The final facts are, all alike, actual entities; and these actual entities are drops of experience, complex and interdependent."*

Whitehead described his scheme as a "cell" theory. The actual occasions of experience (the "true things" or "actual entities") are the "cells" of actuality.

> *"The philosophy of organism is a cell-theory of actuality. Each ultimate unit of fact is a cell-complex, not analysable into components with equivalent completeness of actuality."*

The "cells" can be considered as arising from other cells (when considered "genetically") and as having a formal structure (when considered "morphologically").

> *"The cell can be considered genetically and morphologically."*

"In the genetic theory, the cell is exhibited as appropriating for the foundation of its own existence, the various elements of the universe out of which it arises. Each process of appropriation of a particular element is termed a prehension. The ultimate elements of the universe, thus appropriated, are the already constituted actual entities, and the eternal objects."

The actual occasions are "atomic" in their subjective aim and unity of their final satisfaction, but are divisible by the constituent feelings so that each of the feelings can be felt subsequently.

"The atomic actual entities individually express the genetic unity of the universe. The world expands through recurrent unifications of itself, each, by the addition of itself, automatically recreating the multiplicity anew."

"The other type of indefinite multiplicity, introduced by the indefinite coordinate divisibility of each atomic actuality, seems to show that, at least for certain purposes, the actual world is to be conceived as a mere indefinite multiplicity."

"But this conclusion is to be limited by the principle of 'extensive order' which steps in. The atomic unity of the world, expressed by a multiplicity of atoms, is now replaced by the solidarity of the extensive continuum. This solidarity embraces not only the coordinate divisions within each atomic actuality, but also exhibits the coordinate divisions of all atomic actualities from each other in one scheme of relationship."

The development of Whitehead's scheme revolves around the distinctions between the becoming of the actual occasions of experience (the "present", or the "contemporary world"), the being of the actual occasions of experience (the "past"), and more importantly the relations of one actual occasion to others.

"The positive doctrine of these lectures is concerned with the becoming, the being, and the relatedness of 'actual entities'."

The "true things" or "actual entities" are the actual occasions of experience, but the 'relatedness' of each actual occasion to others is more important than any 'qualities' which it may exhibit.

> *"An 'actual entity' is a res vera [true thing] in the Cartesian [Descartian] sense of that term; it is a Cartesian substance and not an Aristotelian 'primary substance'. But Descartes retained in his metaphysical doctrine the Aristotelian dominance of the category of 'quality' over that of 'relatedness'."*

The complexity and interdependence follows from the relations between actual occasions of experience, whereby the content (or 'feelings') of an actual occasion of experience follows from the many 'prehensions' that appropriate previous actual occasions. In this way, the past is appropriated by the "living immediacy" of the development of a present moment which perishes in the completion of that moment. Hence, broadly speaking, Whitehead's modern process philosophy is characterised by the notion that the past is dead and the future doesn't exist.

> *"In these lectures 'relatedness' is dominant over 'quality'. All relatedness has its foundation in the relatedness of actualities; and such relatedness is wholly concerned with the appropriation of the dead by the living — that is to say, with 'objective immortality' whereby what is divested of its own living immediacy becomes a real component in other living immediacies of becoming. This is the doctrine that the creative advance of the world is the becoming, the perishing, and the objective immortalities of those things which jointly constitute stubborn fact."*

The actual occasions are complex and interdependent, but there is a limit to the interdependence of actual occasions, which is the independence of contemporary actual occasions.

> *"So far as physical relations are concerned, contemporary events happen in causal independence of each other."*

> *"Actual occasions R and S, which are contemporary with M, are those actual occasions which lie neither in M's causal past, nor in*

M's causal future."

"This is in fact the definition of contemporaneousness (cf. Part II, Ch. II, Sect. I); namely, that actual occasions, A and B, are mutually contemporary, when A does not contribute to the datum for B, and B does not contribute to the datum for A, except that both A and B are atomic regions in the potential scheme of spatio-temporal extensiveness which is a datum for both A and B."

Whitehead's speculative philosophy is constructed as a scheme built up of categories, which he calls a 'categoreal scheme'. The categories are proposed as approximations to unobtainable ultimate generalities. The gradual elaboration of such categories is proposed as the proper objective of philosophy.

"Philosophy will not regain its proper status until the gradual elaboration of categoreal schemes, definitely stated at each stage of progress, is recognized as its proper objective. There may be rival schemes, inconsistent among themselves; each with its own merits and its own failures. It will then be the purpose of research to conciliate the differences. Metaphysical categories are not dogmatic statements of the obvious; they are tentative formulations of the ultimate generalities."

That is to say, Whitehead's categories are provisional. Much like this book of mine, there will always be room for further improvement.

Categories

Whitehead's categories are proposed as the aim of his speculative philosophy and not the origin. That is, unlike for example Newton's mistaken assumption of absolute time and space and mass with which he derived his obsoleted mechanics, Whitehead's work does not assume first principles that are appealing due to their initial clarity and from which logical deductions are to be pursued. The origin of his considerations, what he draws on or appeals to, is acknowledged to be intuition. Although

unfamiliarity with Whitehead's scheme may present difficulties of understanding, its appeal to intuition makes it approachable.

> *"The analysis of the components abstracts from the concrescence. The sole appeal is to intuition."*

Through the pages of *Process and Reality*, there is an incremental development of the categories. It is an incremental approach to development that will be familiar to every modern software developer. As an approximation to ultimate generalities, Whitehead suggested the validity of his scheme is to be measured by its general success, by the adequacy and applicability of the categories. Whitehead's intention was that his scheme can be used to express 'any possible interconnection of things'.

> *"In each recurrence, these topics throw some new light on the scheme, or receive some new elucidation. At the end, in so far as the enterprise has been successful, there should be no problem of space-time, or of epistemology, or of causality, left over for discussion. The scheme should have developed all those generic notions adequate for the expression of any possible interconnection of things."*

Whitehead's scheme involves four sets of categories:

- the category of the ultimate;
- categories of existence;
- categories of explanation; and
- categoreal obligations.

The category of the ultimate expresses the general principle of 'creativity' presupposed in the three more special categories. Whitehead invented the word 'creativity' to name his principle of novelty. In Whitehead's scheme, creativity is the universal of universals characterising ultimate matter of fact.

> *"'Creativity is the universal of universals characterizing ultimate matter of fact. It is that ultimate principle by which the many, which are the universe disjunctively, become the one actual occasion,*

which is the universe conjunctively. It lies in the nature of things that the many enter into complex unity."

"'Creativity' is the principle of novelty. An actual occasion is a novel entity diverse from any entity in the 'many' which it unifies. Thus 'creativity' introduces novelty into the content of the many, which are the universe disjunctively. The 'creative advance' is the application of this ultimate principle of creativity to each novel situation which it originates."

"The novel entity is at once the togetherness of the 'many' which it finds, and also it is one among the disjunctive 'many' which it leaves; it is a novel entity, disjunctively among the many entities which it synthesizes. The many become one, and are increased by one. In their natures, entities are disjunctively 'many' in process of passage into conjunctive unity. This Category of the Ultimate replaces Aristotle's category of 'primary substance'."

"Creativity is without a character of its own in exactly the same sense in which the Aristotelian 'matter' is without a character of its own. It is that ultimate notion of the highest generality at the base of actuality. It cannot be characterized, because all characters are more special than itself. But creativity is always found under conditions, and described as conditioned. The non-temporal act of all-inclusive unfettered valuation is at once a creature of creativity and a condition for creativity. It shares this double character with all creatures. By reason of its character as a creature, always in concrescence and never in the past, it receives a reaction from the world; this reaction is its consequent nature. It is here termed 'God'; because the contemplation of our natures, as enjoying real feelings derived from the timeless source of all order, acquires that 'subjective form' of refreshment and companionship at which religions aim."

In Whitehead's scheme, there are eight categories of existence, twenty-seven categories of explanation, and nine categoreal obligations.

The categories of existence aim at distinguishing different kinds of existence. The categories of explanation aim at distinguishing different kinds of explanation. And the categories of obligation aim at distinguishing different kinds of obligation.

> *"Every entity should be a specific instance of one category of existence, every explanation should be a specific instance of categories of explanation, and every obligation should be a specific instance of categoreal obligations. The Category of the Ultimate expresses the general principle presupposed in the three more special categories."*

The categories of existence are:

- actual entities (also termed actual occasions);
- prehensions (concrete facts of relatedness, leading to feelings);
- nexus (public matters of fact);
- subjective forms (private matters of fact);
- eternal objects (pure potentials for the specific determination of fact);
- propositions (theories);
- multiplicities (pure disjunctions of diverse entities);
- contrasts (modes of synthesis of entities in one prehension, or patterned entities).

It's worth pointing out that the notion of patterned entities is at the root of the notion of 'pattern' in Alexander's pattern language scheme. We will return to this idea towards the end of this chapter, under the discussion of structured and living societies.

In Whitehead's scheme, the meaning of the word 'event' includes both individual atomic occasions ('actual entity' or 'actual occasion'), and inter-related sets of such things ('nexus'). As we shall see, while the 'actual occasions' become and perish but do not change, the ordinary physical objects we encounter in everyday life are the things that enjoy adventures of change, and are analysable as societies of actual occasions ('social nexus'). They can all be considered as events.

> *"The actual world is built up of actual occasions; and […] whatever things there are in any sense of 'existence' are derived by abstraction from actual occasions. I shall use the term 'event' in the more general sense of a nexus of actual occasions, inter-related in some determinate fashion in one extensive quantum. An actual occasion is the limiting type of an event with only one member."*

Whitehead's scheme is a one-substance ontology, with each actual entity having a mental and a physical pole. Conscious intellectuality is considered as a high grade of mental activity.

> *"The philosophy of organism abolishes the detached mind. Mental activity is one of the modes of feeling belonging to all actual entities in some degree, but only amounting to conscious intellectuality in some actual entities."*

What is an actual occasion?

An actual occasion is the passage from a set of public facts to a new public fact.

> *"It is a moment of passage from decided public facts to a novel public fact."*

Creative process

The passage of an actual occasion is said to swing from the public to the private and then back to the public.

> *"The creative process is rhythmic: it swings from the publicity of many things to the individual privacy; and it swings back from the private individual to the publicity of the objectified individual."*

It is helpful to understand from the outset that Whitehead's scheme involves two kinds of fluency: the creative process of becoming of an actual occasion ('concrescence'), and the creative process of becoming by which

one actual occasion comes to be one amongst many original elements in the becoming of subsequent actual occasions ('transition').

> *"The former swing is dominated by the final cause, which is the ideal; and the latter swing is dominated by the efficient cause, which is actual."*

The notions of 'final cause' and 'efficient cause' echo two of Aristotle's four causes. The 'final cause' is the purpose of the existence of something, what it exists in order to do, its 'end'. The 'efficient cause' is the agent or already existing potential by which it is possible for it to come into being, its 'agent'. For example, the final cause of a table is to be a platform on which things can be placed above the level on which the table itself stands, and the efficient cause of a table is perhaps a carpenter who is capable of making tables. The final cause of a seed is perhaps to be a plant, and the efficient cause of the the seed is a plant that has already existed (and perhaps continues to exist). It is interesting to note, when considering Whitehead's categories of explanation, that the Greek word that Aristotle used for 'cause' is perhaps best translated into modern English as 'explanation'. In modern English, the word 'cause' tends to mean 'agent' (or 'maker'), and the distinctly creative role of 'purpose' in the selective process of appropriation that making anything new involves, and indeed the creation or selection of a purpose itself, tends to be explained and understood less well. It is beyond the scope of this discussion to ask whether the relative loss of this meaning is because we have been conditioned by the conditions of factory production to expect always to make the same thing, or because there is normally somebody else deciding and telling us what the purpose of our work is, or because the purpose of making anything is commonly understood to be making money, or because making new things has been taken up in the identity of the creatives amongst us, or due to some other reason. Instead, one of the purposes of this discussion is to recover the an adequate understanding of the crucial function of the subjective aim (the final cause or ideal) of any and all actual occasions: to lure feeling. It can be noted that Whitehead made some interesting remarks about the stress that has been given to the notions of final and efficient causes, and the need to describe the "proper relation to each other".

> "*It is notable that no biological science has been able to express itself apart from phraseology which is meaningless unless it refers to ideals proper to the organism in question. This aspect of the universe impressed itself on that great biologist and philosopher, Aristotle. His philosophy led to a wild overstressing of the notion of 'final causes' during the Christian middle ages; and thence, by a reaction, to the correlative overstressing of the notion of 'efficient causes' during the modern scientific period. One task of a sound metaphysics is to exhibit final and efficient causes in their proper relation to each other. The necessity and the difficulty of this task are stressed by Hume in his Dialogues Concerning Natural Religion.*"

The passage of an actual occasion is firstly summarised and then explained below using the notions 'datum', 'subjective aim', 'process', 'concrescence', 'prehension', 'satisfaction', 'decision', and 'transition'.

The movement from public to the private begins from a given 'datum', which is the situation the actual occasion finds itself in and objectifies. Governed by a 'subjective aim' which the actual occasion creates for itself by feeling a proposition from its datum as a purpose, the actual occasion becomes itself through the immediate actual 'process' of its 'concrescence' of 'prehensions'. Its process of becoming ends in the 'satisfaction' of a new private individual caused finally by the subjective aim. This is the actual occasion separated from everything else. However, since the actual occasion is a process of becoming and the process has ended, the actual occasion in perishing swings back through its 'decision' as something that has forever happened and loses itself in the 'transition' to being an objective element in the 'datum' of other actual occasions that it functions efficiently to cause.

The swing from a set of 'decided public facts' to a 'private individual' is explained as a 'concrescence'. Concrescence means 'growth by assimilation' or 'the growing together and merging of similar or dissimilar parts' or 'increment'. The word concrescence dates from the 17th century and is still in use for example by dentists and botanists. In Whitehead's scheme, an actual occasion is the unity of a concrescence.

"An actual occasion is nothing but the unity to be ascribed to a particular instance of concrescence."

"An instance of concrescence is termed an 'actual entity' — or, equivalently, an 'actual occasion.'"

The settled world, of presented public facts that are perceived by the actual occasion through its limited perspective from its standpoint, is the situation or actual world in which the actual occasion finds itself. This is its fixed starting point, or 'datum'.

"The 'settlement' which an actual entity 'finds' is its datum."

The actual world of an actual occasion gives the objective content of its experience, and limits what the actual occasion can become.

"No actual entity can rise beyond what the actual world as a datum from its standpoint — its actual world — allows it to be. Each such entity arises from a primary phase of the concrescence of objectifications which are in some respects settled: the basis of its experience is 'given'."

That is, although an actual occasion is a subject that creates itself, it does so within particular objective obligations. This distinguishes Whitehead's scheme from solipsism.

"The phrase 'objective content' is meant to emphasize the doctrine of 'objectification' of actual entities. If experience be not based upon an objective content, there can be no escape from a solipsist subjectivism."

Concrescence is the process by which an actual occasion is built up from the public facts that it objectifies.

"'Concrescence' is the name for the process in which the universe of many things acquires an individual unity in a determinate relegation of each item of the 'many' to its subordination in the constitution of the novel 'one'."

"This concrescence is thus nothing else than the 'real internal constitution' of the actual occasion in question."

Whitehead's consideration of the 'real internal constitution' of an actual occasion brings in the notion of the 'privacy' of an actual occasion as a feeling subject.

"An actual entity considered in reference to the privacy of things is a 'subject'; namely, it is a moment of the genesis of self-enjoyment. It consists of a purposed self-creation out of materials which are at hand in virtue of their publicity."

Feelings

The process of feeling of an actual occasion is considered by Whitehead to have three phases: the initial feelings that are a response to settled public facts ('datum'); and then subsequent feelings that feel feelings that have already been felt ('process'); and in the end a single feeling that brings all of those feelings together as a single feeling ('satisfaction').

"The analysis of the formal constitution of an actual entity has given three stages in the process of feeling: (i) the responsive phase, (ii) the supplemental stage, and (iii) the satisfaction."

An actual occasion is a creative 'process' of becoming, but its creativeness is creativity as conditioned by its feelings.

"It is by means of its feelings that the subject objectively conditions the creativity transcendent beyond itself."

The process of becoming an actual occasion originates a 'subjective aim'. The subjective aim feels a proposition from the settled world as a purpose and functions as a lure for feeling. It is the 'private ideal' which governs the second stage of feeling towards the satisfaction which it finally causes.

"The 'subjective aim,' which controls the becoming of a subject, is that subject feeling a proposition with the subjective form of purpose to realize it in that process of self-creation."

> *"It is an essential doctrine in the philosophy of organism that the primary function of a proposition is to be relevant as a lure for feeling."*

By defining its own origins and aim, an actual occasion is the cause of itself ('causa sui').

> *"But the admission into, or rejection from, reality of conceptual feeling is the originative decision of the actual occasion. In this sense an actual occasion is causa sui."*

> *"To be causa sui means that the process of concrescence is its own reason for the decision in respect to the qualitative clothing of feelings. It is finally responsible for the decision by which any lure for feeling is admitted to efficiency. The freedom inherent in the universe is constituted by this element of self-causation."*

It is perhaps worth noting that computer programming (and other kinds of design) can be viewed an attempt to eliminate such freedom, in particular the freedom by which things can go wrong. In most cases, the ideal is for the actual occasions of a computer system to be fully determined in advance, for there to be no "bugs", and often they can be so long as we aren't especially fussy about exact timing. This is the sense in which computer programming, like other kinds of design, functions by contributing determination to actual occasions.

Whitehead indicates that every philosophy relies, in some way or other, on a notion of self-causation.

> *"Every philosophy recognizes, in some form or other, this factor of self-causation, in what it takes to be ultimate actual fact."*

> *"This mutual determination of the elements involved in a feeling is one expression of the truth that the subject of the feeling is causa sui."*

The distinction between the first phase of feeling (the responsive phase) and the second phase of feeling (the supplemental phase) is the distinction

between its receiving of the actual world as the 'alien' feelings of other things and the processing of these feelings by the creation of the particular feelings that belong to the actual occasion itself.

> *"The first phase is the phase of pure reception of the actual world in its guise of objective datum for aesthetic synthesis. In this phase there is the mere reception of the actual world as a multiplicity of private centres of feeling, implicated in a nexus of mutual presupposition. The feelings are felt as belonging to the external centres, and are not absorbed into the private immediacy. The second stage is governed by the private ideal, gradually shaped in the process itself; whereby the many feelings, derivatively felt as alien, are transformed into a unity of aesthetic appreciation immediately felt as private."*

A feeling is an episode of self-creation. A feeling is a contribution to the real internal constitution of an actual occasion.

> ""*Each actual entity is conceived as an act of experience arising out of data. It is a process of 'feeling' the many data, so as to absorb them into the unity of one individual 'satisfaction'. Here 'feeling' is the term used for the basic generic operation of passing from the objectivity of the data to the subjectivity of the actual entity in question*."

Prehension is a common word that means 'the act of grasping or gripping'. The adjective 'prehensile' is commonly used to describe the ability to take hold of and clasp objects, especially to grasp by wrapping around an object. For example, some monkeys are said to have prehensile tails.

The sole essence of an actual occasion is that it is a thing that prehends.

> *"In Cartesian language, the essence of an actual entity consists solely in the fact that it is a prehending thing (i.e., a substance whose whole essence or nature is to prehend)."*

Whereas a feeling contributes to the occasion of experience, a 'prehension' is the definite relation or 'bond' with an item of data that may or may not

result in the subject coming to feel that item.

> *"A positive prehension is the definite inclusion of that item into positive contribution to the subject's own real internal constitution. This positive inclusion is called its 'feeling' of that item."*

In Whitehead's scheme, there are physical prehensions which grasp other actual occasions, and conceptual prehensions which grasp eternal objects.

> *"Prehensions of actual entities — i.e., prehensions whose data involve actual entities — are termed 'physical prehensions'; and prehensions of eternal objects are termed 'conceptual prehensions'. Consciousness is not necessarily involved in the subjective forms of either type of prehension."*

In this way, Whitehead's philosophy of organism abolishes the notion of the detached mind. Although, the contrast between physicality and mentality is maintained as the dipolar essence of an actual entity.

> *"In each concrescence there is a twofold aspect of the creative urge. In one aspect there is the origination of simple causal feelings; and in the other aspect there is the origination of conceptual feelings. These contrasted aspects will be called the physical and the mental poles of an actual entity. No actual entity is devoid of either pole; though their relative importance differs in different actual entities. Also conceptual feelings do not necessarily involve consciousness; though there can be no conscious feelings which do not involve conceptual feelings as elements in the synthesis."*

> *"Thus an actual entity is essentially dipolar, with its physical and mental poles; and even the physical world cannot be properly understood without reference to its other side, which is the complex of mental operations. The primary mental operations are conceptual feelings."*

With these distinctions, Whitehead's objects to the "bifurcation of nature", an objection which Christopher Alexander follows in his writing.

Whitehead's ninth categoreal obligation suggests that physical feelings are all felt conceptually (registered on the mental pole).

> *"From each physical feeling there is the derivation of a purely conceptual feeling whose datum is the eternal object determinant of the definiteness of the actual entity, or of the nexus, physically felt."*

> *"For example, 'thirst' is an immediate physical feeling integrated with the conceptual prehension of its quenching."*

Whitehead's tenth category of explanation suggests an actual occasion is a creative concrescence of 'prehensions' that are created as it comes to be what it will become.

> *"That the first analysis of an actual entity, into its most concrete elements, discloses it to be a concrescence of prehensions, which have originated in its process of becoming. All further analysis is an analysis of prehensions."*

Whitehead's eleventh category of explanation suggests a 'prehension' of an actual occasion is the grasping by the actual occasion of an item of its 'datum' into a 'subjective form'.

> *"That every prehension consists of three factors: (a) the 'subject' which is prehending, namely, the actual entity in which that prehension is a concrete element; (b) the 'datum' which is prehended; (c) the 'subjective form' which is how that subject prehends that datum."*

Whitehead's twelfth category of explanation distinguishes between 'positive' and 'negative' prehensions. A positive prehension is a feeling and a negative prehension is an elimination from feeling.

> *"That there are two species of prehensions: (a) 'positive prehensions' which are termed 'feelings,' and (b) 'negative prehensions' which are said to 'eliminate from feeling'. Negative prehensions also have subjective forms. A negative prehension holds*

its datum as inoperative in the progressive concrescence of prehensions constituting the unity of the subject."

"A negative prehension is the definite exclusion of that item from positive contribution to the subject's own real internal constitution. This doctrine involves the position that a negative prehension expresses a bond."

Whitehead's thirteen category of explanation suggests the subjective forms which feelings involve include notions such as 'emotion', 'valuation', 'purpose', 'adversion', 'aversion', and 'consciousness'.

"That there are many species of subjective forms, such as emotions, valuations, purposes, adversions, aversions, consciousness, etc."

This means for example feeling something emotionally, or feeling the value of something, or feeling the purpose of something, or feeling inclined toward something, or feeling disinclined towards something, or feeling aware of something, and so on.

The feelings of an actual occasion are feelings of other actual occasions, feelings of eternal objects, and feelings of feelings that have already been felt.

"A feeling can be considered in respect to (i) the actual occasions felt, (ii) the eternal objects felt, (iii) the feelings felt, and (iv) its own subjective forms of intensity. In the process of concrescence the diverse feelings pass on to wider generalities of integral feeling."

In this way, a feeling can be an integral complex of feelings already felt.

"A feeling is in all respects determinate, with a determinate subject, determinate initial data, determinate negative prehensions, a determinate objective datum, and a determinate subjective form."

Satisfaction and decision

In the end, the creative process of becoming solves the problem of what the actual occasion is to be.

> *"The actual entity, in becoming itself, also solves the question as to what it is to be. Thus process is the stage in which the creative idea works towards the definition and attainment of a determinate individuality."*

The process of becoming, of feeling feelings, continues until 'satisfaction' is reached.

> *"The process continues till all prehensions are components in the one determinate integral satisfaction."*

The satisfaction is a feeling that brings together all the other feelings that belong to the actual occasion.

> *"The actual entity terminates its becoming in one complex feeling involving a completely determinate bond with every item in the universe, the bond being either a positive or a negative prehension. This termination is the 'satisfaction' of the actual entity."*

The private individuality of an actual occasion is its satisfaction. The satisfaction of an actual occasion is the settlement or harmony or unity that is the actual occasion in itself. Satisfaction is reached when there is no incoherence amongst the feelings.

> *"Thus 'becoming' is the transformation of incoherence into coherence, and in each particular instance ceases with this attainment."*

The process of becoming may involve the deselection of incompatible feelings, and the inclusion of other feelings. The satisfaction is the end of this process.

> *"The satisfaction is merely the culmination marking the evaporation of all indetermination; so that, in respect to all modes of feeling and to all entities in the universe, the satisfied actual entity embodies a*

determinate attitude of 'yes' or 'no'. Thus the satisfaction is the attainment of the private ideal which is the final cause of the concrescence. But the process itself lies in the two former phases."

Because a feeling is a fully determinate thing, and the satisfaction is a new feeling that brings together the other feelings that are felt by the actual occasion, the unity of the actual occasion remains to be determined until the many feelings of the actual occasion are united by the satisfaction.

"Actual occasions in their 'formal' constitutions are devoid of all indetermination. Potentiality has passed into realization. They are complete and determinate matter of fact, devoid of all indecision. They form the ground of obligation."

The unity of the satisfaction makes an actual occasion what it is in itself.

"In the conception of the actual entity in its phase of satisfaction, the entity has attained its individual separation from other things; it has absorbed the datum, and it has not yet lost itself in the swing back to the 'decision' whereby its appetition becomes an element in the data of other entities superseding it. Time has stood still — if only it could."

As the creative process of becoming ends, the actual occasion perishes into immortal fact.

"The process of concrescence terminates with the attainment of a fully determinate 'satisfaction'; and the creativity thereby passes over into the 'given' primary phase for the concrescence of other actual entities. This transcendence is thereby established when there is attainment of determinate 'satisfaction' completing the antecedent entity. Completion is the perishing of immediacy: 'It never really is'."

The actuality of an actual occasion is its 'decision', which is how its satisfaction contributes to actual worlds beyond itself. The 'decision' of an actual occasion establishes the individuality of the actual occasion as a stubborn fact.

"Just as 'potentiality for process' is the meaning of the more general term 'entity' or 'thing; so 'decision' is the additional meaning imported by the word 'actual' into the phrase 'actual entity'. 'Actuality' is the decision amid 'potentiality'. It represents stubborn fact which cannot be evaded."

The decision is also a relation that brings the satisfaction of an actual occasion into being a decision for the actual entities that objectify it.

"The ontological principle asserts the relativity of decision; whereby every decision expresses the relation of the actual thing, for which a decision is made, to an actual thing by which that decision is made. But 'decision' cannot be construed as a casual adjunct of an actual entity. It constitutes the very meaning of actuality. An actual entity arises from decisions for it, and by its very existence provides decisions for other actual entities which supersede it."

In this way, an actual occasion is constituted by four stages: 'datum', 'process', 'satisfaction', 'decision'.

"Thus the 'datum' is the 'decision received' and the 'decision' is the 'decision transmitted'. Between these two decisions, received and transmitted, there lie the two stages, 'process' and 'satisfaction'. The datum is indeterminate as regards the final satisfaction. The 'process' is the addition of those elements of feeling whereby these indeterminations are dissolved into determinate linkages attaining the actual unity of an individual actual entity."

Transition

Having become what it is, an actual occasion gives potential for the becoming of other actual occasions. It doesn't really exist except in so far as it figures in the becoming of other actual occasions. But in Whitehead's scheme, every actual occasion is felt by at least one other actual occasion. The creativity by which one actual occasion and another together become the datum for another actual occasion is called 'transition'.

> *"There is not one completed set of things which are actual occasions. For the fundamental inescapable fact is the creativity in virtue of which there can be no 'many things' which are not subordinated in a concrete unity. Thus a set of all actual occasions is by the nature of things a standpoint for another concrescence which elicits a concrete unity from those many actual occasions. Thus we can never survey the actual world except from the standpoint of an immediate concrescence which is falsifying the presupposed completion. The creativity in virtue of which any relative complete actual world is, by the nature of things, the datum for a new concrescence is termed 'transition'. Thus, by reason of transition, 'the actual world' is always a relative term, and refers to that basis of presupposed actual occasions which is a datum for the novel concrescence."*

This point is illuminated by the discussion in Whitehead's category of the ultimate regarding the mutual presupposition of 'the one' and 'the many'.

> *"The term 'many' presupposes the term 'one', and the term 'one' presupposes the term 'many'. The term 'many' conveys the notion of 'disjunctive diversity'; this notion is an essential element in the concept of 'being'. There are many 'beings' in disjunctive diversity."*

Concrescence is therefore distinguished from transition as two different becomings: the becoming of datum from decisions (transition), and the becoming of a satisfaction from a datum (immediate actual process).

> *"There is the becoming of the datum, which is to be found in the past of the world; and there is the becoming of the immediate self from the datum. This latter becoming is the immediate actual process. An actual entity is at once the product of the efficient past, and is also, in Spinoza's phrase, causa sui."*

Whitehead explains this distinction between two kinds of fluency in the universe as coming mostly from Locke.

"With all his inconsistencies, Locke is the philosopher to whom it is most useful to recur, when we desire to make explicit the discovery of the two kinds of fluency, required for the description of the fluent world. One kind is the fluency inherent in the constitution of the particular existent. This kind I have called 'concrescence.' The other kind is the fluency whereby the perishing of the process, on the completion of the particular existent, constitutes that existent as an original element in the constitutions of other particular existents elicited by repetitions of process. This kind I have called 'transition'. Concrescence moves towards its final cause, which is its subjective aim; transition is the vehicle of the efficient cause, which is the immortal past."

We can now fully understand the relation between 'subject' and 'object' in Whitehead's scheme, which Whitehead notes as an inversion of the generally accepted ideas in the Anglo-American world which we have received from Kant ("the Kantian doctrine of the objective world as a theoretical construct from purely subjective experience").

"For Kant, the world emerges from the subject; for the philosophy of organism, the subject emerges from the world — a 'super ject' rather than a 'subject.' The word 'object' thus means an entity which is a potentiality for being a component in feeling; and the word 'subject' means the entity constituted by the process of feeling, and including this process. The feeler is the unity emergent from its own feelings; and feelings are the details of the process intermediary between this unity and its many data. The data are the potentials for feeling; that is to say, they are objects. The process is the elimination of indeterminateness of feeling from the unity of one subjective experience."

Four grades

Whitehead describes four 'grades' of actual occasions: those in "empty space" such as the propagation of an electromagnetic wave; those of inorganic matter such as stone and metal; those of more simple living

objects such as bamboo and mushrooms and daffodils; and those of conscious living creatures such as cats and elephants and whales and apes.

> *"In the actual world we discern four grades of actual occasions, grades which are not to be sharply distinguished from each other. First, and lowest, there are the actual occasions in so-called 'empty space'; secondly, there are the actual occasions which are moments in the life-histories of enduring non-living objects, such as electrons or other primitive organisms; thirdly, there are the actual occasions which are moments in the life-histories of enduring living objects; fourthly, there are the actual occasions which are moments in the life-histories of enduring objects with conscious knowledge."*

What distinguishes the fourth grade from the others is that these actual occasions involve the mode of 'presentational immediacy'. These occasions correspond to the notion of 'qualia' of subjective experience. That's what was happening when the Greeks saw the stone was grey. Rather than abstracting from the occasion of experiencing grey stone and concluding with the substance-quality categories, Whitehead abstracts from the occasion of experience, and concludes that seeing grey stone is a particularly high grade of experience, and also that there lower grades.

> *"The fourth grade is to be identified with the canalized importance of free conceptual functionings, whereby blind experience is analysed by comparison with the imaginative realization of mere potentiality. In this way, experience receives a reorganization in the relative importance of its components by the joint operation of imaginative enjoyment and of judgment. The growth of reason is the increasing importance of critical judgment in the discipline of imaginative enjoyment."*

This leads to an interesting discussion about the "defining characteristic of a living person", the "evolution of personal mentality" in the higher animals, and the "central direction, which suggests that in their case each animal body harbours a living person". Which brings us, again, to the psychology of Rogers and Rosenberg.

Example: The Alexandrian form

The four stages of an actual occasion usefully explain the major parts of the 'Alexandrian form'. If a design pattern form has roughly four parts (context-problem, discussion-solution, solution-diagram, archetypal example), it is because design patterns are descriptions of actual occasions.

The context-problem (the 'datum') is a perspective on the actual world, a selection of relevant facts that are to be felt (e.g. "A, but also B"). The selection and objectification of other things from the particular standpoint of a pattern defines a 'context' and contrasts a 'problem' to be solved.

The discussion-solution (the 'process') considers the context-problem by developing feelings that find ways in which the concern or problem can be settled (e.g. "by doing C"), and so involves a 'discussion' that moves towards a solution.

The solution-diagram (the 'satisfaction') is a declaration of the 'solution', as an individual structure, and perhaps as a sketch or a diagram (e.g. "X has Y"), which brings together what has been felt in the discussion to create a complex and coherent unity.

The archetypal example (the 'decision') is illustrative of the settlement ("for example Z"), and gives a picture of the solution as something that can be easily grasped.

The parts of the Alexandrian form which link to other patterns represent what is "analytical of its potentiality for 'objectification' in the becoming of other actual entities". The name of a pattern reflects the 'subjective aim' of the occasion, which lures feelings and governs the development of feelings of the context.

As with Alexander's patterns, Whitehead's concept of an actual occasion can be applied a million times over, without ever doing it in the same way twice. Hence, the variety of 'pattern forms' which are listed in Ward Cunningham's Portland Pattern Repository.

What is an event?

Individual occasions of experience are related to others. One thing leads another. In Whitehead's scheme, the term 'event' is used in the general sense of a 'nexus' or set of inter-related actual occasions of experience, with an actual occasion being the limiting type of an event with only one member.

> *"Whenever we attempt to express the matter of immediate experience, we find that its understanding leads us beyond itself, to its contemporaries, to its past, to its future, and to the universals in terms of which its definiteness is exhibited."*

Continuing the discussions of the notions of efficient and final causation, Whitehead's eighteenth category of explanation which concerns the "search for a reason" is termed both the 'ontological principle' and alternatively 'the principle of efficient and final causation'. This brings together the notion of efficient cause and the notion of final cause in the notion of the reason why a thing happens to be what it is.

> *"That every condition to which the process of becoming conforms in any particular instance has its reason either in the character of some actual entity in the actual world of that concrescence, or in the character of the subject which is in process of concrescence. This category of explanation is termed the 'ontological principle'. It could also be termed the 'principle of efficient, and final, causation'. This ontological principle means that actual entities are the only reasons; so that to search for a reason is to search for one or more actual entities. It follows that any condition to be satisfied by one actual entity in its process expresses a fact either about the 'real internal constitutions' of some other actual entities, or about the 'subjective aim' conditioning that process."*

A nexus of actual occasions is either social or non-social. A social nexus has a social order, and is called a 'society'. A society is a nexus that has a social order of some kind or other. There is no such thing as social order in general.

A non-social nexus is a nexus that does not have the defining characteristic that give a social nexus its social character.

> *"A non-social nexus is what answers to the notion of 'chaos.'"*

Social order

A society of actual occasions is a set of related actual occasions, with a 'social order' of one kind or another. The actual occasions of a society share a defining characteristic of the society.

When the actual occasions of a society have serial ordering, the society is said to have the special kind of social order known as 'personal order'.

> *"A nexus enjoys 'personal order' when (a) it is a 'society', and (b) when the genetic relatedness of its members orders these members 'serially'."*

A society that has personal order is called an 'enduring object'.

> *"Thus the nexus forms a single line of inheritance of its defining characteristic. Such a nexus is called an 'enduring object'."*

A 'corpuscular society' is a society of actual occasions that has multiple strands of personal order. The ordinary physical objects we encounter in daily life are societies of actual occasions that can mostly be considered as corpuscular societies and in the simple case as an enduring object.

> *"An ordinary physical object, which has temporal endurance, is a society. In the ideally simple case, it has personal order and is an 'enduring object'. A society may (or may not) be analysable into many strands of 'enduring objects'. This will be the case for most ordinary physical objects. These enduring objects and 'societies', analysable into strands of enduring objects, are the permanent entities which enjoy adventures of change throughout time and space."*

Hence, the ordinary physical objects we encounter in everyday life are events.

Structured societies

Returning to the idea of pattern language as a collection of interrelated patterns, with some subordinate to others, we can refer to the idea of a structured society in Whitehead's scheme.

> *"It is obvious that the simple classification of societies into 'enduring objects', 'corpuscular societies', and 'non-corpuscular societies' requires amplification. The notion of a society which includes subordinate societies and nexus with a definite pattern of structural inter-relations must be introduced. Such societies will be termed 'structured'."*
>
> *"A structured society as a whole provides a favourable environment for the subordinate societies which it harbours within itself. Also the whole society must be set in a wider environment permissive of its continuance."*
>
> *"A 'structured society' may be more or less 'complex' in respect to the multiplicity of its associated sub-societies and sub-nexus and to the intricacy of their structural pattern."*
>
> *"Molecules are structured societies, and so in all probability are separate electrons and protons. Crystals are structured societies. But gases are not structured societies in any important sense of that term; although their individual molecules are structured societies."*

Living occasions and living societies

Returning to the notion of a pattern as describing one of the "configurations that brings life", and the common description of pattern language as "organised and coherent" collection of patterns, we can refer to the idea in Whitehead's scheme of 'living occasions' and the derivative idea of 'living societies'.

> *"A 'living society' is one which includes some 'living occasions.' Thus a society may be more or less 'living,' according to the prevalence in it of living occasions."*

The distinction between 'living societies' and 'non-living societies' is one of degree, and not of a binary opposition or duality. The distinction is also a matter of the purpose under which a society is prehended.

> *"It is obvious that a structured society may have more or less 'life' and that there is no absolute gap between 'living' and 'non-living' societies. For certain purposes, whatever 'life' there is in a society may be important; and for other purposes, unimportant."*

The distinction between a 'living occasion' and a 'non-living occasion' is also a matter of degree and purpose, in other words of the "importance" of its "novel factors".

> *"Also an occasion may be more or less living according to the relative importance of the novel factors in its final satisfaction."*

The 'meaning of life' is taken to be the origination of conceptual novelty.

> *"In accordance with this doctrine of life, the primary meaning of 'life' is the origination of conceptual novelty — novelty of appetition."*

Whitehead adds several characteristics to his notion of a living society, an obvious one being the need for food.

> *"Another characteristic of a living society is that it requires food. In a museum the crystals are kept under glass cases; in zoological gardens the animals are fed. Having regard to the universality of reactions with environment, the distinction is not quite absolute. It cannot, however, be ignored. The crystals are not agencies requiring the destruction of elaborate societies derived from the environment; a living society is such an agency. The societies which it destroys are its food."*

The idea of life as origination of novelty, should be contrasted with, or contexualised within, the stated purpose of 'God' in Whitehead's scheme, which is intensity of feeling and depth of satisfaction, rather than the production of innovation or the preservation of tradition. That is, novelty and repetition are both means to achieving deeper satisfaction.

> *"The primordial appetitions which jointly constitute God's purpose are seeking intensity, and not preservation. Because they are primordial, there is nothing to preserve. He, in his primordial nature, is unmoved by love for this particular, or that particular; for in this foundational process of creativity, there are no preconstituted particulars. In the foundations of his being, God is indifferent alike to preservation and to novelty. He cares not whether an immediate occasion be old or new, so far as concerns derivation from its ancestry. His aim for it is depth of satisfaction as an intermediate step towards the fulfilment of his own being. His tenderness is directed towards each actual occasion, as it arises."*

> *"Thus God's purpose in the creative advance is the evocation of intensities. The evocation of societies is purely subsidiary to this absolute end. The characteristic of a living society is that a complex structure of inorganic societies is woven together for the production of a non-social nexus characterized by the intense physical experiences of its members. But such an experience is derivate from the complex order of the material animal body, and not from the simple 'personal order' of past occasions with analogous experience. There is intense experience without the shackle of reiteration from the past. This is the condition for spontaneity of conceptual reaction. The conclusion to be drawn from this argument is that life is a characteristic of 'empty space' and not of space 'occupied' by any corpuscular society. In a nexus of living occasions, there is a certain social deficiency. Life lurks in the interstices of each living cell, and in the interstices of the brain. In the history of a living society, its more vivid manifestations wander to whatever quarter is receiving from the animal body an enormous variety of physical experience."*

The reason for caring about pattern language, then, is that through bringing out and feeling what is alive in any given moment, we can help sustain life, and thereby contribute to the making of adjustments that lead to the intensities that give deeper satisfaction.

There are many references in *Process and Reality* to the notions of pattern and of language, but the phrase "patterned intertwining" does stand out.

> *"There is yet another factor in living societies which requires more detached analysis. A structured society consists in the patterned intertwining of various nexus with markedly diverse defining characteristics."*

These notions, of 'structured' and 'living' societies can help us to understand the common description of Alexander's pattern language as "organized and coherent" and his statement about "configurations that brings life". They can seem to be echoed in the "person-centred theory" and the "actualising tendency" of Rogers' therapeutic psychology.

Living persons

It is well here to bring in person-centred psychology, because it is an advance which would have been welcomed by Whitehead. At time of writing *Process and Reality*, psychology as we know it today was in its early stages of development, as Whitehead noted in a remark that contrasts "physical physiology" with "psychological physiology".

> *"'Physical Physiology' deals with the subservient inorganic apparatus; and 'Psychological Physiology' seeks to deal with 'entirely living' nexus, partly in abstraction from the inorganic apparatus, and partly in respect to their response to the inorganic apparatus, and partly in regard to their response to each other. Physical Physiology has, in the last century, established itself as a unified science; Psychological Physiology is still in the process of incubation."*

It appears that Whitehead's scheme is perhaps the best way to understand the person-centred psychology of Carl Rogers. One of the founders of the humanistic approach to psychology, Rogers is regarded as one of the founding fathers of psychotherapy. In a survey of US and Canadian psychologists, Rogers was considered the first most influential psychotherapist in history. Rogers' person-centred theory is highly reminiscent of Whitehead's philosophy of organism — his nineteen propositions are almost a summary of Whitehead's scheme. Just as Whitehead's scheme sets aside the notion of the substance-quality category in favour of actual occasions of experience and feelings that relate one to another, so Rogers' scheme sets aside the diagnosis of personality types and types of personality disorder, in favour of working with the occasions of experience that each human person is at the centre of, and the feelings about those experiences that each of us has (or doesn't have).

In Whitehead's scheme, the idea that a 'living person' is a character that is sustained and developed moment-by-moment through its feelings combines the ideas of 'enduring object' and 'non-social living society'. Whitehead wrote:

> *"Thus the nexus forms a single line of inheritance of its defining characteristic. Such a nexus is called an 'enduring object'. It might have been termed a 'person' in the legal sense of that term. But unfortunately 'person' suggests the notion of consciousness, so that its use would lead to misunderstanding. The nexus 'sustains a character' and this is one of the meanings of the Latin word persona. But an 'enduring object' qua 'person' does more than sustain a character. For this sustenance arises out of the special genetic relations among the members of the nexus."*

Whitehead goes on to discuss what a 'living person' is, and how its thread of 'personal order' exists within a non-social living nexus that gives it life.

> *"The complexity of nature is inexhaustible. So far we have argued that the nature of life is not to be sought by its identification with some society of occasions, which are living in virtue of the defining characteristic of that society. An 'entirely living' nexus is in respect to its life, not social. Each member of the nexus derives the*

necessities of its being from its prehensions of its complex social environment; by itself the nexus lacks the genetic power which belongs to 'societies'. But a living nexus, though non-social in virtue of its life, may support a thread of personal order along some historical route of its members. Such an enduring entity is a living person. It is not of the essence of life to be a living person. Indeed a living person requires that its immediate environment be a living, non-social nexus."

The living non-social nexus is the animal body that has life. The personal order of the living person is a line of inheritance that sustains a character. The life of the animal body would be a unsustainable disaster without the line of inheritance in the moment-by-moment existence of the living person itself. The living person itself thereby provides central coordination for the body, and "binds originality within bounds" allowing for the "canalisation" of personal mentality. This is, more or less, an explanation of the "actualising tendency" of person-centred theory.

"The defining characteristic of a living person is some definite type of hybrid prehensions transmitted from occasion to occasion of its existence. The term 'hybrid' is defined more particularly in Part III. It is sufficient to state here that a 'hybrid' prehension is the prehension by one subject of a conceptual prehension, or of an 'impure' prehension, belonging to the mentality of another subject. By this transmission the mental originality of the living occasions receives a character and a depth. In this way originality is both 'canalized' — to use Bergson's word — and intensified. Its range is widened within limits. Apart from canalization, depth of originality would spell disaster for the animal body. With it, personal mentality can be evolved, so as to combine its individual originality with the safety of the material organism on which it depends. Thus life turns back into society: it binds originality within bounds, and gains the massiveness due to reiterated character."

Whitehead doesn't suppose that all living things have personality. But he seems to think it reasonable to assume that any animal which is directly

aware of its capability for central direction would effectively contain a living person.

> *"In the case of single cells, of vegetation, and of the lower forms of animal life, we have no ground for conjecturing living personality. But in the case of the higher animals there is central direction, which suggests that in their case each animal body harbours a living person, or living persons. Our own self-consciousness is direct awareness of ourselves as such persons."*

Whitehead's scheme is such that it is easy to imagine how this process might not always work out, such that the personal order of the living person is disrupted, giving "multiple personalities". But also that central direction can occur without the direct awareness that leads to the development of personality.

> *"There are limits to such unified control, which indicate dissociation of personality, multiple personalities in successive alternations, and even multiple personalities in joint possession. This last case belongs to the pathology of religion, and in primitive times has been interpreted as demoniac possession. Thus, though life in its essence is the gain of intensity through freedom, yet it can also submit to canalization and so gain the massiveness of order. But it is not necessary merely to presuppose the drastic case of personal order. We may conjecture, though without much evidence, that even in the lowest form of life the entirely living nexus is canalized into some faint form of mutual conformity. Such conformity amounts to social order depending on hybrid prehensions of originalities in the mental poles of the antecedent members of the nexus. The survival power, arising from adaptation and regeneration, is thus explained. Thus life is a passage from physical order to pure mental originality, and from pure mental originality to canalized mental originality."*

With this scheme in mind, we can understand that Rogers' scheme it is our feelings, and our awareness of our feelings, which makes up a well-adjusted relation with ourselves and our world, through which we can grow to become fully who we are. Mal-adjustment follows from "subception", unconsciously applying strategies to prevent a troubling stimulus from

entering consciousness. The notion of subception follows the more general notion in Whitehead's scheme of "negative prehension" (elimination from feeling).

Marshall Rosenberg used Rogers' person-centred approach to develop an even more decisively empathetic, life-bringing, moment-by-moment approach to our encounters with other people and ourselves. Rosenberg's book *Nonviolent Communication: A Language for Life* presents a common approach for the development of more habitable human relations. The goal of nonviolent communication is to do the spiritual work of bringing out what is alive in other people, from moment to moment. The general approach is to start from an attitude of genuine care for oneself and other people, to make observations without generalisation or interpretation, to express and try to understand what feelings are alive in a given moment, to identify needs that are met (and not being met) by considering what those feelings point to, to express gratitude for particular things that have been done that contributed to needs being met, to make requests without obligation or rewards or punishments. And perhaps when there is conflict to get to the point where each can express to the other what they are feeling and needing, and then from that position to find strategies and make requests by which everybody's needs can be met.

The general view of nonviolent communication is that all human activity is an attempt to meet needs. But it can be recognised that the moment when we have the greatest need for empathy is also the time when we are perhaps most likely to express what we are feeling and needing in a way that disinclines others (and even oneself) from understanding our feelings and contributing to meeting our needs. For this reason, the practice of nonviolent communication, especially listening through words and thoughts for feelings and needs, even to the point of not hearing insults and criticism and judgements, can make the difference between the breakdown of ones relationship (even with oneself), and relationships that are joyful and that make life wonderful. Relationships are made moment-by-moment, and each moment is made of feeling what is being felt. The purpose of emotion is to motivate action to meet needs. By connecting with and understanding what is being felt, by identifying what needs are not being met, by developing and proposing strategies by which needs are met, we can act to meet those

needs and all together feel better rather than worse. This isn't easy and it can feel exhausting. The importance of gratitude in this process is that gratitude provides the fuel that giving empathy tends to consume.

Nonviolent communication (also called compassionate communication, or collaborative communication) is congruent with Whitehead's scheme, and shares with Alexander's pattern language scheme the central desire for enhancing life through the creative use of feelings, firstly to make adequate connection with what has gone before and what is happening now, and then to create a harmony in the unity of an adequate satisfaction. Nonviolent communication appears highly applicable in software development as an approach that can be explained and shared for connecting with ourselves and other people, for developing and maintaining relationships, and for understanding and meeting needs. Microsoft CEO Satya Nadella bought all the members of his senior leadership team a copy of the book *Nonviolent Communication* in 2014 when he took over the company.

Person-centred theory and therapy, and collaborative or nonviolent communication are discussed in more detail in the Epilogue of this book.

Examples: Events in software

In an event-sourced domain model, a 'domain event' object is an event that is an actual occasion: it is indivisible in the sense that there cannot be half a domain event; and it does not change (it is what it is).

An event-sourced 'aggregate' whose state is determined by a sequence of domain events can be identified as an enduring object: it is a society with personal order since new domain events are appended to the end of a sequence. An aggregate with two or more domain events is therefore an event that is not an actual occasion.

Domain events are immutable, but aggregates can change. The word change is frequently used but infrequently defined. What is change? Whitehead defines change as the difference between actual occasions. To make a change, therefore, is to make an actual occasion that can be contrasted with a previous actual occasion.

> *"The fundamental meaning of the notion of 'change' is 'the difference between actual occasions comprised in some determinate event.'"*

An event-sourced domain model that has many such aggregates can be identified as a corpuscular society, since each aggregate has its own sequence of domain events. A domain model is therefore also an event; it is something that happens. If all the domain events of a domain model have a stable 'total order' then the domain model can be said to have its own strand of personal order too.

It is also worth noting that the software code of a domain object class is also an event, it is also something that happens. Kent Beck said, "all software is written one character at at time." The typing of a single character is an actual occasion (you cannot type half a character) and so are the individual changesets committed to a version control system (a changeset is either committed or not).

This demonstrates that actual occasions can be smaller (in that they exist as primitives) and larger (in that they are composed of smaller things). Actual occasions can be composed of other actual occasions. For example, the atomicity of a domain event can be contained within the atomicity provided by an aggregate's consistency boundary. As we shall see in the chapters of this book, the consistency boundary of an aggregate can be extended, from being the atomic change to an aggregate that results from executing a command, to containing other things such as event notifications for these domain events, domain events from other aggregates, and position in a sequence of domain event notifications that are being processed ('process event').

A piece of software that was developed using a version control system can also be identified as an enduring object, since it has personal order: for any particular version there is a serial ordering of changesets from the initial empty state of the repository. As a whole, a repository with more than one branch can be identified as a corpuscular society: each branch is one strand of personal order.

Software developers are living persons. The obligation of any collective effort to develop a piece of working software is to bring out the personal order of the software system from the corpuscular society of the group of individual developers. The various approaches to branching and merging can be viewed as different strategies for bringing out this personal order from that corpuscular society. Pair programming, trunk based development, and mob programming are different ways of, or patterns for, meeting this need.

All together, the progressive analysis and design, or development, of software is an event. The main chapters of this book contain a coherent and organised collection of software design patterns that can contribute to the development of deeply satisfying event-sourced applications and event-driven systems.

Some influences of Whitehead's scheme

Whitehead has been influential in the philosophy of Gilles Deleuze, Isabelle Stengers, Bruno Latour, Susanne Langer, and many others. Deleuze remarked that Whitehead "stands provisionally as the last great Anglo-American philosopher before Wittgenstein's disciples spread their misty confusion, sufficiency, and terror." Stengers wrote many interesting and useful things about Whitehead, including her book *Thinking with Whitehead*. Latour called Whitehead, "the greatest philosopher of the 20th century."

Bertrand Russell, a long-time friend and collaborator, was one of Whitehead's students at Cambridge. Russell wrote in his autobiography:

> "*Whitehead was extraordinarily perfect as a teacher. He took a personal interest in those with whom he had to deal. He would elicit from a pupil the best of which a pupil was capable. He was never repressive, or sarcastic, or superior, or any of the things that inferior teachers like to be. I think that in all the abler young men with whom he came in contact he inspired, as he did in me, a very real and lasting affection*."

Whitehead and Russell together wrote *Principia Mathematica*, the famous work on the foundations of mathematics. In their book, published 1910-1913, Whitehead and Russell described a complex system now called "the ramified theory of types". which involved the notion 'class' which compounds the notions of 'type' and 'set'. One of the three aims of this book, as stated in its introduction, was to solve paradoxes of logic and set theory. Russell's Paradox was identified in Gottlob Frege's set theory, but also the notion of types was also taken from Frege's work. Whitehead and Russell described a hierarchy of types which became the origin of type theory. Type systems are used in computer programming to ascribe meaning to code, and type theory is used to analyse and avoid bugs in computer code written using a type system. Kurt Gödel used the axioms of *Principia Mathmatica* when proving his incompleteness theorems. Gödel's theorums were followed by Alonzo Church's and Alan Turing's independently derived but equivalent theorum's regarding computability, which, amongst other things, establishes limits for the static analysis of computer programs, and motivates test-driven driven development and the development of dynamically-typed programming languages such as Python.

Russell's book *Our Knowledge of the External World*, published in 1914, used Whitehead's method of extensive abstraction (an important aspect of his process philosophy) for the purpose of "bridging the gulf between the world of physics and the world of sense". However, Russell then ran into difficulties carrying forward the development of the ideas, and apparently annoyed Whitehead by badgering him for unpublished notes, complaining that he was unable to work without them. Whitehead refused to disclose his notes, writing in a letter sent in 1917:

> *"I am awfully sorry, but you do not seem to appreciate my point. I don't want my ideas propagated at present either under my name or anybody else's – that is to say, as far as they are at present on paper. […] I do not want you to have my notes […] I am sorry that you do not feel able to get to work except by the help of these notes"*

Whitehead had previously persuaded a skeptical Russell of his process philosophic ideas, and this seems to have at least had some influence on the

development of Russellian monism. Russell wrote in his book *My Philosophical Development*, published in 1959:

> *"When, however after 1910, I had done all that I intended to do as regards pure mathematics, I began to think about the physical world and, largely under Whitehead's influence, I was led to new applications of Occam's razor, to which I had become devoted by its usefulness in the philosophy of arithmetic. Whitehead persuaded me that one could do physics without supposing points and instants to be part of the stuff of the world. He considered — and in this I came to agree with him — that the stuff of the physical world could consist of events, each occupying a finite amount of space-time. As in all uses of Occam's razor, one was not obliged to deny the existence of the entities with which one dispensed, but one was enabled to abstain from ascertaining it. This had the advantage of diminishing the assumptions required for the interpretation of whatever branch of knowledge was in question. As regards the physical world, it is impossible to prove that there are not point-instants, but it is possible to prove that physics gives no reason whatever for supposing that there are such things."*

Amongst other notables interested in and influenced by Whitehead, the minister Martin Luther King was very interested in Whitehead's work and quoted Whitehead in his Nobel Prize acceptance speech, Gertrude Stein was a good friend of the Whiteheads, and we know from library records that Alan Turing at least read Whitehead's *Science and the Modern World* when he was a boy.

The chemist Ilya Prigogine was deeply influenced by Whitehead. Prigogine won the Nobel prize for his work on 'dissipative structures' and their role in thermodynamic systems, and made connections with the Turing mechanism which explains how spatial patterns form autonomously in an organism. In The Tanner Lectures On Human Values, delivered in 1982, Prigogine stated:

> *"You may be astonished that I have spoken little about cosmological theories. […] This description of nature, in which order is generated out of chaos through nonequilibrium conditions provided by our*

cosmological environment, leads to a physics which is quite similar in its spirit to the world of 'processes' imagined by Whitehead. It leads to a conception of matter as active, as in a continuous state of becoming."

Dissipative structure theory led to research in self-organising systems, which contributed strongly to the science of complexity. Prigogine references Whitehead in several books, such as *From Being to Becoming*. He collaborated with Isabelle Stengers, co-authoring books *The End of Certainty* and *Order Out of Chaos*. Another contribution in this area was the understanding of anthropology through systems theory by Gregory Bateson, who was influenced by developmental biologist Conrad Waddington.

Whitehead's scheme was applied by Conrad Waddington to pioneer the study of growth and development of living things. Brian Goodwin was a student of Waddington. Christopher Alexander refers to Brian Goodwin in his later work, for example in a paper called Harmony Seeking Computation. Waddington coined the biological notion of epigenetics, "the branch of biology which studies the causal interactions between genes and their products, which bring the phenotype into being". The basic idea is that environmental factors influence an "epigenetic landscape" causing heritable changes in gene expression: a change in the 'phenotype' (without a change in the 'genotype') which affects how cells read their genes. Epigenetics is now a broad and widely accepted topic in biology, formalised using the science of complexity which Prigogine had contributed to developing. According to Wikipedia:

"In recent times, Waddington's notion of the epigenetic landscape has been rigorously formalized in the context of the systems dynamics state approach to the study of cell-fate. Cell-fate determination is predicted to exhibit certain dynamics, such as attractor-convergence (the attractor can be an equilibrium point, limit cycle or strange attractor) or oscillatory."

Whitehead's process philosophy is widely known for its influence on education theory. The influence was to reconsider the purpose and process of education as the stimulation of personal development rather than the development and injection of a curriculum. A range of models of teaching

have been developed such as the ANISA model developed by Daniel C. Jordan, and the FEELS model developed by Xie Bangxiu.

Process psychology was also largely inspired by Whitehead's process philosophy. Bernie Neville wrote an article entitled What Kind of Universe? Rogers, Whitehead and Transformative Process, which describes the influence of Whitehead's work on Carl Rogers.

> *"Rogers locates his thinking within the paradigm of the new science and lists a number of key figures who support his notion of an organic, evolving universe. Among them is the mathematician and philosopher Alfred North Whitehead. We may speculate that Rogers was introduced to Whitehead's process philosophy during his tenure at the University of Chicago, which was at the time a major center for Whitehead's thinking. Whether or not Rogers was directly influenced by Whitehead scholars, his psychology is essentially a process psychology, and his reflections on therapy in A Way of Being (1980) make most sense when viewed in the context of Whitehead's process view of cosmology."*

> *"When Rogers came to reflect on the assumptions underlying his own thinking, he located them on a trajectory which runs from Smuts, Bergson and Whitehead, through Szent-Gyoergyi and Whyte, to Prigogine, Murayama and Capra (Rogers, 1980, pp. 124–132)."*

Whitehead's thought has also been influential in environmental sustainability. John B. Cobb is notable for his work in this area, and for co-founding and co-directing the Center for Process Studies in Claremont, California, which is regarded as the leading Whitehead-related academic institute. In China, Whitehead's work has been blended with traditions of Taoism, Buddhism, and Confucianism. Whitehead's scheme has been promoted most intensely by the Chinese government through building at least twenty-three university-based centres for the study of Whitehead's philosophy, with the assistance of Cobb and his institute. These initiatives aim to help develop a more ecological civilization, which is a goal the Chinese government has written into its constitution. China's best-known environmentalist, Sheri Liao, has actively engaged with Whiteheadians in

the West and has won awards from Chinese government ministries for her work.

Whitehead's scheme was influenced by early developments in quantum theory, by the theory of relativity, and by the theory of electromagnetism. Amongst physicists known to have been influenced by Whitehead include David Bohm, and Henry Stapp. For me, the physicist who most exemplifies Whitehead's process thought is the Nobel laureate Roger Penrose: his conformal cyclic cosmology (which accords with the avoidance of an original temporal actual occasion); his use of cohomology in his explanation of non-locality in the wave function of a single particle which forbids the detection of a particle in two places and when discussing Bell's inequalities (which accords with Whitehead's fallacy of simple location); his interpretation of quantum mechanics and the collapse of the wave function as an objective reduction (which accords with Whitehead's notion of actual occasions being the cause of themselves); and his Platonic 'three realms' view of reality and generous use of geometric thought in the creation of physical theory. Penrose references Whitehead many times in a paper entitled Conscious Events as Orchestrated Space-Time Selections, written with Stuart Hameroff:

> *"Some philosophers have contended that 'qualia', or an experiential medium from which consciousness is derived, exists as a fundamental component of reality. Whitehead, for example, described the universe as being comprised of 'occasions of experience'. […] We contend that this type of objective self-collapse introduces non-computability, an essential feature of consciousness […] the climax of a self-organizing process in fundamental space-time–and a candidate for a conscious Whitehead 'occasion' of experience. […] 'Mentalists' such as Leibniz and Whitehead (e.g. 1929) contended that systems ordinarily considered to be physical are constructed in some sense from mental entities. […] Among these positions, the philosophy of Alfred North Whitehead (1929; 1933) may be most directly applicable. Whitehead describes the ultimate concrete entities in the cosmos as being actual 'occasions of experience', each bearing a quality akin to 'feeling'. Whitehead construes 'experience' broadly […] so that even 'temporal events in*

the career of an electron have a kind of protomentality'. Whitehead's view may be considered to differ from panpsychism, however, in that his discrete 'occasions of experience' can be taken to be related to quantum events. […] We take the self-reduction to be an instantaneous event – the climax of a self-organizing process fundamental to the structure of space-time - and apparently consistent with a Whitehead 'occasion of experience'. […] Thus Buddhist 'moments of experience', Whitehead 'occasions of experience', and our proposed Orch OR [orchestrated objective reduction] events seem to correspond tolerably well with one another."

Whether or not orchestrated objection reduction is a good theory, this paper demonstrates an influence of Whitehead's thought on at least some of Penrose's thought. Regarding the merits or otherwise of orchestrated objection reduction, perhaps we can remember Whitehead's remark that it is more important for an idea to be interesting than for it to be correct.

"It is more important that a proposition be interesting than it be true. But of course a true proposition is more apt to be interesting than a false one."

It is perhaps worth remarking that the status of 'decision' in physics is still being debated. From Newtonian mechanics, to Einstein's spacetime, to Feynman's diagrams, and the "many worlds" interpretation… most of the physical theories can run just as well "forwards" as "backwards" in time. The "arrow of time" that means that we remember the past but not the future, that eggs are broken and cooked, and that glass is smashed and wine is spilled, that differences in heat level out, that liquids mix… seems somewhat disconnected from these theories. The "growing block" model of the universe introduces theoretical and philosophical complications. The deterministic interpretations of quantum mechanical theory leave no room for creative decisions, or indeed any decision. The spontaneous or objective collapse interpretations do leave room for decisions to be made, but the difficulty seems to be that to progress the investigation, gravity needs to be quantised and quantum mechanics needs to be relativized, and we don't have a theory that goes beyond these difficulties. The merit of the Penrose

interpretation seems to be that at least these contrasting feelings are felt, even if a final satisfaction hasn't been reached. Relativizing the superpositions is problematic because the possibility arises for the particle to have ceased as a possibility in one position without having been realised in the other position, or for it to have been realised in one position without having ceased to be a possibility in the other position. The feeling that Penrose seems to have is that the branching into super position doesn't happen, that the decision is made before the branching would have occurred, and that that somehow the situation "feels forward" into the future. And perhaps also into the space it will occupy, as if the region of spacetime to be occupied by the realized particle is presupposed by the becoming of the collapsed state. This seems to take the consideration beyond physical time (and physical space) in exactly the way that Whitehead described in *Process and Reality* when he discusses the contrast between the "genetic theory" and the "morphological theory", the two considerations of an actual occasion, stating that "genetic passage from phase to phase is not in physical time", that "the genetic process is not the temporal succession", that "physical time expresses some features of the growth, but not the growth of the features", and that "each phase in the genetic process presupposes the entire quantum", that "there is a spatial element in the quantum as well as a temporal element", and that spatial region "is the determinate basis which the concrescence presupposes".

> *"There are two distinct ways of 'dividing' the satisfaction of an actual entity into component feelings, genetically and coordinately. Genetic division is division of the concrescence; coordinate division is division of the concrete. In the 'genetic' mode, the prehensions are exhibited in their genetic relationship to each other. The actual entity is seen as a process; there is a growth from phase to phase; there are processes of integration and of reintegration. At length a complex unity of objective datum is obtained, in the guise of a contrast of actual entities, eternal objects, and propositions, felt with corresponding complex unity of subjective form. This genetic passage from phase to phase is not in physical time: the exactly converse point of view expresses the relationship of concrescence to physical time. It can be put shortly by saying, that physical time*

expresses some features of the growth, but not the growth of the features. The final complete feeling is the 'satisfaction'."

"Physical time makes its appearance in the 'coordinate' analysis of the 'satisfaction'. The actual entity is the enjoyment of a certain quantum of physical time. But the genetic process is not the temporal succession: such a view is exactly what is denied by the epochal theory of time. Each phase in the genetic process presupposes the entire quantum, and so does each feeling in each phase. The subjective unity dominating the process forbids the division of that extensive quantum which originates with the primary phase of the subjective aim. The problem dominating the concrescence is the actualization of the quantum in solido [as a whole]. *The quantum is that standpoint in the extensive continuum which is consonant with the subjective aim in its original derivation from God. Here 'God' is that actuality in the world, in virtue of which there is physical 'law'."*

"There is a spatial element in the quantum as well as a temporal element. Thus the quantum is an extensive region. This region is the determinate basis which the concrescence presupposes."

Whitehead's scheme was also enormously influential for Christopher Alexander's pattern language scheme, but so far this influence hasn't been recognised within the software development community, despite the broad acceptance and application of Alexander's pattern language scheme.

"In an earlier chapter (Part II, Ch. IV, Sects. IV to IXt) the sense in which the world can be conceived as a medium for the transmission of influences has been discussed. This orderly arrangement of a variety of routes of transmission, by which alternative objectifications of an antecedent actuality A can be indirectly received into the constitution of a subsequent actuality B, is the foundation of the extensive relationship among diverse actual entities."

The introductory chapters of this book express my crude attempt to extend an awareness of Whitehead's scheme into the process and reality of

software development. My purpose in writing all of this is simply to clarify the situation for myself and others, to accomplish a greater intensity of feeling and a deeper satisfaction in our work.

Domains and Domain Models

The purpose of this chapter is to discuss the notions ‘domain’, ‘domain model’, ‘aggregate’, and ‘domain event’. The notions are explained both in conventional terms, and with reference to Whitehead’s scheme. By applying Whitehead’s scheme to our consideration of domains and domain models, we can obtain an event-oriented consideration that is generally adequate and applicable in which events and decisions are dominant.

Domains

According to Eric Evans’s book *Domain-Driven Design*, published in 2004, a domain is “the subject area to which the user applies a program”.

In practical terms, a domain is a little world of human activity that is characterised by certain desired outcomes. A domain consists of activities that accomplish those outcomes. A useful software application will usefully support those activities, helping its users to accomplish outcomes that are desired in the domain. (There are different ways of characterising the characterisation of a domain, but it seems to me that considering the final ‘outcomes’ leads to the most effective and decisive connective synthesis.)

In order to understand what a software application might usefully do, software developers can solicit advice from ‘domain experts’. Domain experts will have experience and a good understanding of the domain. Domain experts can explain the outcomes that are desired in the domain, and how those outcomes can be accomplished. Domain experts may suggest ways in which a software application could be supportive of those activities.

So that the terms used by the software developers are congruent with the terms used in the domain, the book *Domain-Driven Design* suggests that software developers and domain experts converge on a ‘ubiquitous

language'. The ubiquitous language can then be used to express a 'model' of the domain that makes sense to all.

The extent of a ubiquitous language is known in *Domain-Driven Design* as a 'bounded context'. A large domain may be responded to with more than one bounded context. A 'context map' is a diagram of many bounded contexts. Each bounded context will then address a particular 'subdomain'. The term 'domain model' can refer both to the domain model that exists in a bounded context, and also to all of the domain models across all of the inter-related bounded contexts. Admitting different bounded contexts allows for divergence of meanings, so that the same name can be used in different ways in different contexts. The domain model of a particular bounded context will be supportive of the particular 'subdomain' described by its ubiquitous language. Having multiple bounded contexts doesn't really reduce inherent complexity, since complications arise at the boundaries and in the interaction of the different domain models. The development of multiple bounded contexts is a way of tackling complexity inherent in a large domain, and avoids the instability of attempting to arrive at a unified model across a real multiplicity of subdomains.

Domain models

In order to support the domain, a software application will have a certain amount of 'domain logic' that codes for the important supportive behaviour of the application. The domain logic is an expression of how the software application supports the domain (the decisions that it needs to make). The domain logic will figure amongst other aspects that are needed to make the software application work, such as user interfaces that present the functionality of the application to its users, and mechanisms which persist the state of the application to a database.

Martin Fowler's book *Patterns of Enterprise Application Architecture*, published in 2003, describes a pattern called Transaction Script. For very simple domains, a number of 'transaction scripts' may be sufficient to express the domain logic that is needed to support a domain. There would be one 'transaction script' for each procedure supported by an application.

A transaction script may be implemented using the Command pattern, described in the book *Design Patterns: Elements of Reusable Object-Oriented Software*, published in 1994. Transaction scripts may or may not attempt to separate domain logic from any persistence mechanisms required to persist the state of the application.

The use of the Command pattern to implement transaction scripts gives opportunity to refactor aspects of domain logic which is involved in more than one procedure, which may otherwise be duplicated or repeated across the set of transaction scripts. The more sophisticated an application becomes, the more domain logic it is likely to have. Refactoring the domain logic in the transaction scripts may tend towards the development of a distinct 'domain model'. The book *Patterns of Enterprise Application Architecture* includes a design pattern called Domain Model, which describes that idea. Designing more sophisticated domain models is the central topic of the book *Domain-Driven Design*. Many authors have written about this topic. In all but the simplest cases, it is now a common practice when developing a supportive software application to segregate the domain logic in one or many domain models.

Several advantages follow from separating the domain logic from the persistence mechanisms of a software application. For example, the domain logic of the application can be expressed more clearly by being expressed independently of particular persistence mechanisms. And the particular mechanisms used to persist the state of the application can be expressed more clearly and changed more easily. By bringing the domain logic of a software application together in a domain model, aspects of the domain logic which may be involved in more than one command or procedure of the application can be expressed once and not repeated or duplicated. These considerations motivate the desire to have distinct and separate 'layers' in the software application, for the domain logic and for the persistence mechanisms. The 'n-tiered' architecture expresses this separation of concerns. The 'hexagonal architecture' and 'onion architecture' are variants of the same idea which emphasise an 'inversion of control' by which the domain layer can stand alone without depending directly on the persistence mechanisms.

When developing a domain model, developers work to create a productive coupling between the model and the domain. There may be things that happen in the supported domain that do not figure in the domain model, and there may be things in a domain model that are not in the domain itself. Although the word 'model' normally means 'approximation' or 'map', a domain model is a little world of software objects that is *supportive* of its domain. A supportive domain model does not need to be a good analogy of the domain, but it might be. Any so-called *representation* of the domain in the domain model is most likely supporting the *recording* of things that happen in the domain. It is important to remember that the purpose of a domain model is to be supportive of its domain, and not merely a reflection or an approximation.

Domain objects and aggregates

Domain models have commonly been developed as a collection of enduring and changing software objects. In Martin Fowler's book *Patterns of Enterprise Application Architecture* they are called 'domain objects'. In Eric Evans' book *Domain-Driven Design* they are called 'aggregates'. The 'domain objects' or 'aggregates' are commonly understood as modelling tangible objects, ordinary physical objects, and other objective concerns in the domain. If there is a 'person' in the domain, then there may be a 'person' in the domain model.

> *"An AGGREGATE is a cluster of associated objects that we treat as a unit for the purpose of data changes."*
>
> —Eric Evans

In *Domain-Driven Design*, the state of an aggregate is defined as a cluster of 'entities' and 'value objects'. Entities have variable attribute values, and a fixed identity. The attribute values of an entity may be value objects or other entities. Value objects have immutable attribute values but are not necessarily unique or individually identifiable.

One of the entities is used as the 'root' of the aggregate. This entity is referred to as the 'aggregate root', and its identity is used to uniquely

identify an aggregate within its domain model. As a rule, the non-root entities and the value objects are presented and manipulated through the aggregate root. Because the identity of an aggregate does not change, it can provide continuity of reference as the aggregate changes.

In *Patterns of Enterprise Application Architecture* and *Domain-Driven Design*, the notions of 'unit of work' and 'consistency boundary' respectively ensure that either all or none of the changes to the enduring and changing objects that result from executing a command or running a procedure of the application are recorded. The use of atomic database transactions to record potentially many changes determines the persistence of those changes as 'atomic'. Making sure that either all or none or changes are recorded avoids the risk that only some of the changes will be recorded, an outcome which might result in the recorded state of the application being internally inconsistent or otherwise undesirable.

The use of atomic database transactions to record domain model changes can be identified as defining the recording as an actual occasion. However, although the changes to the state of the enduring and changing objects are occasionally discussed as being caused by events (internal and external), it remains that in neither *Patterns of Enterprise Application Architecture* nor *Domain-Driven Design* (nor *Object-Oriented Analysis and Design*, first published in 1990) were the actual occasions of the enduring and changing objects themselves (their decisions) made explicit as generalities. In those books, the enduring and changing objects are emphasised and made explicit as generalities, but the actual occasions were not. The enduring and changing objects are not discussed as being built up of actual occasions. Furthermore, the enduring and changing objects themselves are never considered as events.

The emphasis in the software design patterns books on the enduring and changing objects of a domain model and not on their actual occasions, and the broad characterisation of the enduring and changing objects in a domain as instances of substance-quality categories and not as events, seems to have happened because the prevailing general understanding of things was conditioned by pre-modern categories derived from the Greeks. Since Whitehead's book *Process and Reality* was published in 1929, it was

certainly possible for object-oriented analysis and design in the 1980s and 1990s and 2000s to have been conditioned by modern process philosophy, but apparently it wasn't. Under the two-hierarchies scheme proposed as the canonical form of a complex system in *Object-Oriented Analysis and Design*, domains have mostly been conceived as built up of instances of substance-quality categories, and domain models have followed suit. When writing the great books of software design patterns, despite great efforts having been made to be faithful to Alexander's scheme, the enormous influence of Whitehead's event-oriented scheme on the work of Christopher Alexander was mostly overlooked. This is understandable, but it is also inadequate. It precipitates the object-relational impedance mismatch. It leads to dual writing when messaging is introduced. And it conditions developers to think like Linnaean taxonomists. None of these remarks are aimed at criticising object-oriented programming in itself, which as a technological advance can perhaps be understood in summary as an improvement on programming with data structures that have function pointers. Instead, these remarks are aimed at criticising a style of analysis and design of domains and domain models that is predominated by substance-quality categories rather than by a process-relational conception of events. The purpose of making this criticism is to make way for an event-oriented analysis and design. In event-oriented analysis and design, when considering domains and domain models, attention turns away from the substance-quality categories as the primary unit of analysis, to the analysis and design of events.

As we have seen in Whitehead's scheme, the "ordinary physical objects" that "enjoy adventures of change" are are all societies of actual occasions that are mostly analysable as strands of enduring objects, with the "ideally simple case" being a single enduring object. An 'enduring object' is a society that has 'personal order'. Personal order is the special kind of social order that a society has when its actual occasions have serial ordering. The aggregates of *Domain-Driven Design* can be recognised as enduring objects, because the 'consistency boundary' design causes the actual occasions of a domain model aggregate to occur one after the other. The "sequence of events" that figures in the common description of event sourcing makes this situation explicit.

The actual occasions of the enduring and changing objects of a domain model can be coded explicitly as domain event objects. Domain event objects represent decisions that are made in a domain model. All domain models make decisions. But not all domain models express or represent their decisions explicitly with domain event objects.

Domain events

When investigating a situation, it is always useful to ask the question "what happened?" Asking this question is always useful because actual worlds are built up of actual occasions. To quote from Whitehead's book *Process and Reality*, "whatever things there are in any sense of 'existence' are derived by abstraction from actual occasions." And indeed, to use the words of Christopher Alexander, they are built up "of nothing else".

Since the phrase 'domain event' was originally used to refer to occasions of experience in the domain itself, a better name for the actual occasions of the enduring and changing objects of a domain model would perhaps be 'domain model event'. But it is reasonable to use the shorter name 'domain event' for the software objects, so long as we remain sensitive to the problems of representation, and remember the need to be supportive of the actual occasions (and events more generally) in the domain itself.

Often the term 'domain event' is shortened to 'event', which is even more useful for brevity but unfortunately may encourage the conceptual feeling that other things are not events. Unless we understand each world as a society of actual occasions, we may find that our thoughts become restricted as we fall back onto substance-quality categories. That is the danger with privileging the enduring and changing objects above the actual occasions from which they arise. Fortunately, the existence of Whitehead's scheme provides the courage to be decisive in taking the view that actual worlds, including domain models, are built up of actual occasions, and that other things, such as the enduring and changing objects, are "derived by abstraction" (by which is meant "leaving out some of the truth").

When referring to ‘domain events’, we need to bear in mind the more general meaning of the term ‘event’ in Whitehead’s scheme, which justifies the enduring and changing objects themselves also being referred to as events. Hence, the importance of the distinction between ‘actual occasions’ (immutable domain events) and ‘societies of actual occasions’ (the enduring and changing objects).

The term ‘actual occasion’ corresponds to the common notion of ‘decision’. But when considering the term ‘decision’ we need to distinguish between the choice between a set of given alternatives, and the decisions that create the alternatives in the first place. We may choose chocolate ice cream rather than vanilla ice cream. But before we can make this decision, the chocolate ice cream itself has to be decided.

Decisions

Exceptionally and commendably, Ward Cunningham wrote in his pattern language for programming called Episodes, published in 1995, that programming is the act of deciding now what will happen in the future.

> “*Programming is the act of deciding now what will happen in the future. A programming language offers an operationally precise way to encode decisions through a process called simply* coding. *Programmers reason about future behavior by interpreting previously coded decisions and integrating these with their own decisions and their interpretations of other sources like Technical Memos and domain experts. The depth, quality and value of programming decisions will be limited by the programmers ability to concentrate*.”
>
> “*Therefore: Develop a program in discrete episodes*.”

Cunningham’s Episodes, in name and content, is outstanding in its following the idea that the world is built up of actual occasions of experience. We do best when we remain close to our experience, which we can abstract into our general understand of things as happening moment-by-moment, as a sequence of episodes, as discrete occasions of experience.

Cunningham's use of the word 'decision' is decisive. Actual occasions can be effectively understood as decisions. The great philosopher of science Isabelle Stengers emphasised this point in her book *Thinking with Whitehead: a free and wild creation of concepts*:

> *"Sometimes, in the course of this text, I have been unable not to anticipate, and to use the word 'decision', which Whitehead was to use in Process and Reality to name the 'breaking off' that turns the occasion into the affirmation of a 'thus and not otherwise'."*

We can also refer to the following description in *Process and Reality* of the 'ontological principle' which views an actual occasion as the 'relation' of that which decides to that which is decided, as that which arises from decisions and that which provides decisions for later decisions.

> *"The ontological principle asserts the relativity of decision; whereby every decision expresses the relation of the actual thing, for which a decision is made, to an actual thing by which that decision is made. But 'decision' cannot be construed as a casual adjunct of an actual entity. It constitutes the very meaning of actuality. An actual entity arises from decisions for it, and by its very existence provides decisions for other actual entities which supersede it. Thus the ontological principle is the first stage in constituting a theory embracing the notions of 'actual entity', 'givenness,' and 'process', Just as 'potentiality for process' is the meaning of the more general term 'entity', or 'thing', so 'decision' is the additional meaning imported by the word 'actual' into the phrase 'actual entity'. 'Actuality' is the decision amid 'potentiality'. It represents stubborn fact which cannot be evaded."*

The patterns of a pattern language seek to condition occasions of design. The designs condition subsequent occasions of experience of those affected by the designs. In the case of software-intensive systems, between the designer and the user, there is the active software itself.

The decisions made by a software domain model are both conditioned by actual occasions of design during software development, and then condition (and are conditioned by) the actual occasions of experience in the supported

domain. However, until relatively recently in the history of software development, the primary importance of the actual occasions of a domain model was not made explicit. The domain model events were not emphasised in the software design pattern books. Software applications conditioned by these books mostly did not express the actual occasions of a domain model explicitly with software objects.

Software developers work carefully to understand the decisions in the domain itself (the outcomes that are desired and the decisions that lead to those outcomes). We must decide what decisions need to be made by our software in order that it will be supportive of its domain. The function of software design patterns is to enhance the creativity of our development work, by conditioning our occasions of analysis and design.

In summary, what is most important about the domain logic of a software application is that it makes decisions. Such decisions are the actual occasions of the domain model. So that the software will usefully support decisions made in the domain, software developers work to make good decisions about the decisions that will made by their software. These decisions also need to be expressed in a way that makes it easy to pick up the traces of previous episodes in future episodes. A domain model contains the domain logic, and is usefully expressed as a collection of enduring and changing objects that are built up of actual occasions. Therefore, when developing a domain model, developers above all need to focus on the domain model's actual occasions, whilst also giving consideration to how those actual occasions form interrelated sets that define the enduring and changing objects and the domain model itself.

To make the actual occasions of a domain model explicit, it makes good sense to express the domain model in terms of immutable domain event objects. The enduring and changing objects, for example the aggregates of *Domain-Driven Design*, can then be expressed as deriving from, and as originating, immutable domain event objects. This style of coding a domain model is known as 'event sourcing'.

An event-sourced domain model is a domain model that has its current state determined by immutable domain event objects from the past. In this case, immutable domain event objects are an explicit statement of individual

decisions made by the ‘command methods’ of enduring and changing objects, or by ‘factory methods’ by which the enduring and changing objects are brought about.

Four stages of immutable domain events

We can consider the occasion of experience of an immutable domain model event object by referring to the four stages of an actual occasion in Whitehead’s scheme: ‘datum’, ‘process’, ‘satisfaction’, and ‘decision’.

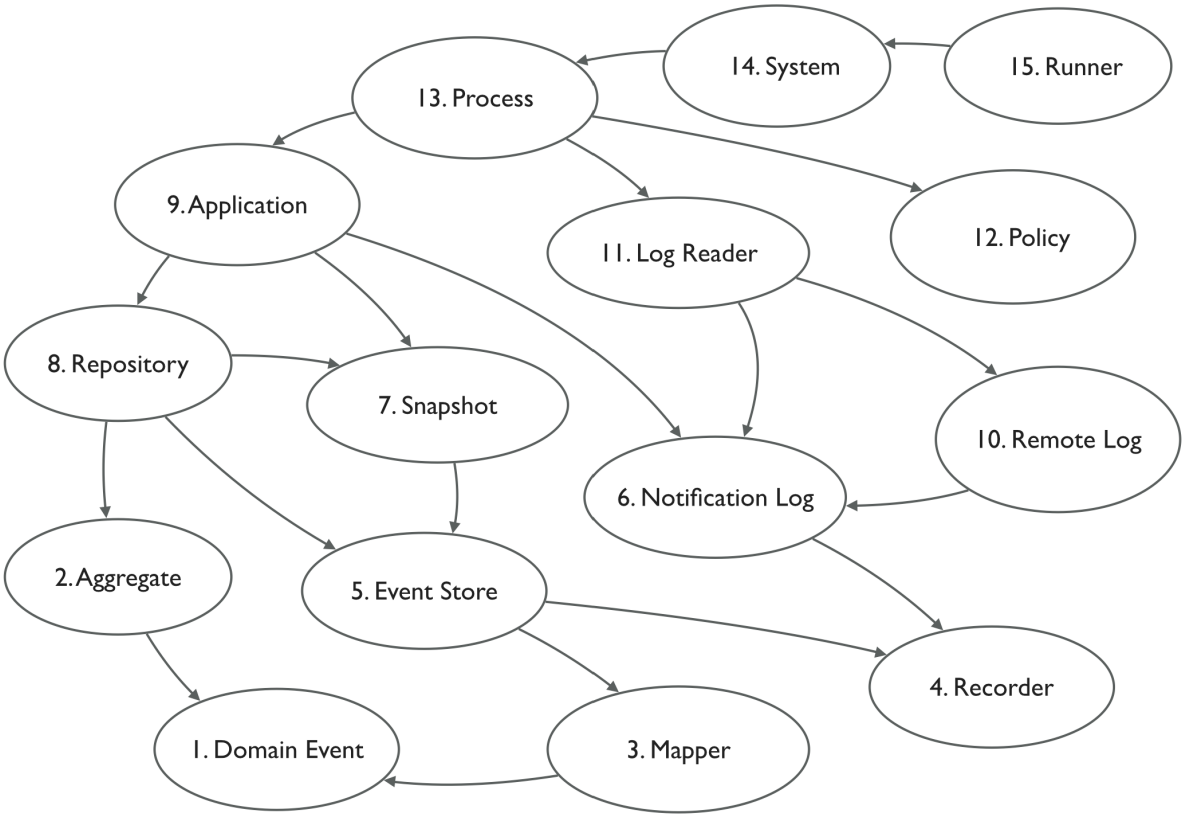
The creation of a new domain event object is the ‘satisfaction’ of an actual occasion of a domain model. The central topic of Chapter 1 is the creation of immutable domain event objects themselves. To put this in context, whilst looking forward to the other chapters of Part 1, it may help briefly to consider the other stages.

As we shall see in Chapter 2, immutable domain event objects from the past contribute determination to the current state of the domain model’s enduring and changing objects (the ‘aggregates’). New immutable domain event objects are created by the command methods of the enduring and changing objects, and are a function of the command method arguments and the state of the enduring and changing object. Hence, existing domain events object contribute to the context, or settled world, or ‘datum’, from which a new domain event object originates. This context also includes the domain model code, the Python language interpreter (or equivalent), and the hardware on which the software runs. The immediate ‘process’ by which a new domain event object comes to be, draws on this context and works towards the “private individuality” and creation of a new domain event object.

The ‘decision’ which brings the immutable domain model event into the “publicity of many things” follows both from applying the domain event object to (and thereby mutating) the aggregate from which it originated, and (as we shall see in Chapter 3 and Chapter 4) from the recording of the domain event object as a ‘stored event’. The applying and recording of a domain event object gives the domain event object the publicity that it

needs to function as part of the datum for the creation of subsequent domain model objects. As mentioned above, sometimes several domain event objects will be included in the actual occasions decided by a consistency boundary (or unit of work) so that either all or none of those domain event objects will be recorded in a database. The electrical creativity of the domain model then "passes over" to the creative becoming of future occasions of experience of a domain model ('transition').

Overview of the Patterns



Each chapter describes one pattern, one characteristic occasion of design, one building block for event-sourced Domain-Driven Design. The descriptions are each intended to contribute determination to future design events that have the particular character of that pattern. Each chapter includes working examples that illustrate the characterised occasion of design, but which could be varied by the reader in different ways. The chapter examples build on examples from previous chapters.

Part 1 is about domain models. It has patterns to define, trigger, and store domain model events, and to project domain model events into the enduring objects which trigger them.

1. Domain Event
2. Aggregate
3. Mapper
4. Recorder
5. Event Store

Part 2 is about applications. It has patterns to unify the components of an event-sourced application, and to propagate the state of the application.

6. Notification Log
7. Snapshot
8. Repository
9. Application
10. Remote Log

Part 3 is about systems, and has patterns to process events and to define and run systems of applications that process domain model events.

11. Log Reader
12. Policy
13. Process
14. System
15. Runner

As we have seen, a pattern language is a description of events. In order to develop a world, we need to design and contribute definition to its events. We need an adequate conceptual feeling of events, of actual occasions, and of societies of actual occasions. We need a form of pattern language, but we only need the simplest adequate way of describing events.

Rather than patterning off the application of Whitehead's scheme in the architecture of the built environment (Alexander's scheme) it is possible to apply Whitehead's scheme directly in software development. That is what I have attempted to do in the chapters of this book.

The simplest adequate design pattern form would help condition an occasion of design. It would start with an initial datum of feeling things that already exist, perhaps an incoherence but perhaps not, a discord or

incitement to novelty or a zest for the enhancement of some dominant element of feeling. At least one feeling, or perhaps two or more feelings contrasted together calling for a new harmony or unity. What then remains is for a new satisfaction to be achieved through a process of consideration or discussion, and then revealed and described with an illustrative example. The name of the pattern may summarise the subjective aim that lures these feelings. So long as it helps to condition an actual occasion of design, it's a design pattern. This is the simple way that pattern language has been used in the chapters of this book.

The chapter examples have been written in the Python programming language (v3.8). The examples only use modules in the Python Standard Library, except for the use of the `AES` cipher from the pycryptodome in Chapter 3, and the use of the `psycopg2` driver for interacting with PostgreSQL.

The examples in the earlier chapters effectively provide a lightweight framework for event sourcing in Python, and the later chapters provide a lightweight framework for event-sourced applications and event-driven systems. The later chapters follow through on the consequences by showing how a distributed system can be defined independently of particular infrastructure and mode of running.

Following the "cohesive mechanism" pattern of Domain-Driven Design, event sourcing as a persistence mechanism for Domain-Driven Design can be understood as a cohesive mechanism, which can be "partitioned into a separate lightweight framework".

> *"Partition a conceptually COHESIVE MECHANISM into a separate lightweight framework. Particularly watch for formalisms for well-documented categories of algorithms. Expose the capabilities of the framework with an INTENTION-REVEALING INTERFACE. Now the other elements of the domain can focus on expressing the problem ('what'), delegating the intricacies of the solution ('how') to the framework."*
>
> —Eric Evans

The Domain-Driven Design community and the Python community have not so far overlapped greatly. However, Python is one of the world's most popular programming languages and allows rapid development of highly maintainable code. Domain-Driven Design is an approach to designing software systems, and focuses on modelling language and the development of fluent models in code. Python and Domain-Driven Design can work well together, and do.

PART 1 DOMAIN MODEL

Chapter 1 Domain Event

"The actual world is built up of actual occasions."

therefore…

Tackle complexity in a domain by modelling events.

The purpose of this chapter is to consider a particular kind of software object commonly referred to as the 'domain event'.

As we have seen in the introductory chapters, actual occasions are the unique, creative advances of the universe. They become and perish into "stubborn fact" but "do not change". They are what they are. The immutable character of actual occasions is exhibited by immutable objects in software. The 'value objects' in *Domain-Driven Design* are examples of immutable objects. Immutable objects can also be found in reactive user interfaces.

What is important about a domain model is that it makes decisions. These decisions can be represented with immutable domain model event objects. Immutable domain event objects represent the immutable actual occasions of the enduring and changing objects of a domain model. There is no object-relational impedance mismatch between a sequence of immutable domain model event objects and an append-only table of stored events in a relational database management system.

Coding domain event objects as software object classes allows software developers to define different templates for recording the different types of decisions that a domain model will make. Such object classes are a record of design decisions that made by software developers about the kinds of decisions that the domain model will make.

Different types of domain event objects will affect the current state of the domain model in different ways, and the software objects classes can help

to make this clear.

The established convention for naming domain event object classes is to use past participles, such as 'created', 'paid', 'done'. In *Domain-Driven Design*, the names of domain event object classes, and their attributes, will draw upon and contribute to a 'ubiquitous language'. The attribute values of domain event objects will be either simple native types or custom value objects or perhaps non-root entities. The attributes values should not refer to the aggregate root entity.

> *"Events are typed, uniquely identified and contain a set of attribute/value pairs."*
>
> —Rebecca Wirfs-Brock

Except for the value objects which they may contain, domain event objects do not depend on any other parts of an application. However, it can help to factor out recurring or common aspects into supertypes. For example, as we shall see, there is a practical advantage to arranging domain event objects in sequences that are originated by an enduring object. For this reason it is useful to identify the sequence by referring to identity of the originator of a domain event object, and to identify the position in that sequence as the version of the enduring object that the domain event will determine. The identity and version of the originator can then be used to uniquely identify domain event objects. As we shall see in the example below, to avoid repetition these attributes can be usefully pulled up to an extracted base class which all the domain event classes in a domain model will extend.

Domain event objects may also usefully be timestamped, so that we can tell roughly when they were created.

Domain events can sometimes be marked as being directly 'caused' by a preceding event, or recognised as a 'continuity' from an earlier event. But the individuality of different domain event classes will follow from the particular kind of decision that each represents.

Domain event objects may also be versioned, with older versions up-cast to new versions. In this case, the version number of the domain event object

class may be included in the state of the domain event object. When making changes to an event-sourced domain model that is already being used in production, the general guiding principle is to adopt an "append-only" attitude when making changes to code. Hence changes can be made by adding attributes to existing classes, and by adding new classes. Renaming or otherwise changing attributes of existing classes, or removing attributes and classes, is possible but will generally involve rewriting existing stored event records, so that the records can be used by the new code.

In summary, immutable domain event objects:

- Express decisions in a domain model
- Are immutable (they occur but do not change)
- Have unique identities within their domain model
- Have different types, and may be coded as object classes
- Named as past participles (regular and irregular)
- Include a set of named attribute values
- Have no dependencies on other parts of the application
- May be versioned, to allow for additions to the model

Example

The example below introduces a type of domain event object for opening an account. The object class is then refactored into a base class for domain event objects and a concrete class for opening accounts. The base class will be used in the following chapters of this book.

Let's say the object class `AccountOpened` defined below expresses the occasion of deciding to open an account of some kind.

```
from dataclasses import dataclass
from datetime import datetime
from uuid import UUID

@dataclass(frozen=True)
class AccountOpened:
```

```
    originator_id: UUID
    originator_version: int
    timestamp: datetime
    full_name: str
```

The `originator_id` may be used to identify an enduring and changing object that originated an instance of this domain event object class. The `originator_version` would distinguish the position in a sequence of other domain event objects.

Together, the `originator_id` and the `originator_version` attributes may be used to uniquely identify domain event objects within in a domain model. This uniqueness contributes to determining the object as an entity.

The `timestamp` attribute may be used to show when the event object was created, and then used to indicate when its originator was created or last modified.

The `full_name` attribute may represent the "full name" of the owner of the account.

The `AccountOpened` object class can be constructed by calling it with suitable arguments. Let's construct the `AccountOpened` object class with a new version-4 UUID, a version number of 1, with the current date and time, and with a full name "Alice".

```
from uuid import uuid4

originator_id = uuid4()

event1 = AccountOpened(
    originator_id=originator_id,
    originator_version=1,
    timestamp=datetime.now(),
    full_name="Alice",
)
```

The domain event object can be tested by checking the attribute values are what we would expect them to be.

```
assert isinstance(event1.originator_id, UUID)
assert event1.originator_id == originator_id
assert event1.originator_version == 1
assert isinstance(event1.timestamp, datetime)
assert event1.full_name == "Alice"
```

The class `AccountOpened` is defined as a frozen Python dataclass, so that its attributes cannot be assigned new values. This effectively determines the object as an immutable object — as a model of an event that comes to be, but that does not then change.

We can test the domain event attribute values cannot be changed by checking that an exception is raised when an attempt is made to assign new values to the object.

```
from dataclasses import FrozenInstanceError

try:
    event1.originator_id = uuid4()
except FrozenInstanceError:
    pass
else:
    raise AssertionError("Not immutable")
```

We can also model other domain event objects by defining other domain event object classes. For example, the occasion of deciding to update the full name may be modelled with an object class `FullNamedUpdated` such as the one defined below. The class name is different from `AccountOpened` but the fields are all exactly the same.

```
@dataclass(frozen=True)
class FullNameUpdated:
    originator_id: UUID
    originator_version: int
```

```
    timestamp: datetime
    full_name: str
```

We might also want to define an object class for when the account is closed, such as the class `AccountClosed` that is defined below. The fields differ from those of `AccountOpened` and `FullNameUpdated` only by not having a `full_name`.

```
@dataclass(frozen=True)
class AccountClosed:
    originator_id: UUID
    originator_version: int
    timestamp: datetime
```

We can see that there is some commonality between the above object class definitions of `AccountOpened`, `FullNamedUpdated`, and `AccountClosed`. With a little bit of care, base classes can be usefully extracted from the concrete domain event classes.

The metaclass `FrozenDataClass` will be used to avoid decorating all the domain event object classes with `@dataclass(frozen=True)`.

```
class FrozenDataClass(type):
    def __new__(cls, *args):
        new_cls = super().__new__(cls, *args)
        return dataclass(frozen=True)(new_cls)
```

The object class `ImmutableObject` will be used as the base class for immutable objects.

```
class ImmutableObject(metaclass=FrozenDataClass):
    pass
```

The object class `DomainEvent` is an `ImmutableObject` and has the `originator_id`, `originator_version`, and `timestamp` attributes. It can be used as the base class for concrete domain event objects.

```
class DomainEvent(ImmutableObject):
    originator_id: UUID
    originator_version: int
    timestamp: datetime
```

The base class `DomainEvent` makes it easier to define concrete domain event object classes. For example, the domain event object classes above can be defined more simply in the following way.

```
class AccountOpened(DomainEvent):
    full_name: str

class FullNameUpdated(DomainEvent):
    full_name: str

class AccountClosed(DomainEvent):
    pass
```

These refactored domain event object classes can be used to construct domain event objects in exactly the same way as before.

```
event1 = AccountOpened(
    originator_id=originator_id,
    originator_version=1,
    timestamp=datetime.now(),
    full_name="Alice",
)

assert event1.originator_id == originator_id
assert event1.originator_version == 1
assert isinstance(event1.timestamp, datetime)
assert event1.full_name == "Alice"
```

Let's say the full name is updated from Alice to Bob.

```
event2 = FullNameUpdated(
    originator_id=originator_id,
    originator_version=2,
    full_name="Bob",
    timestamp=datetime.now(),
)

assert event2.originator_id == originator_id
assert event2.originator_version == 2
assert isinstance(event2.timestamp, datetime)
assert event2.full_name == "Bob"
```

Let's say the account is then closed.

```
event3 = AccountClosed(
    originator_id=originator_id,
    originator_version=3,
    timestamp=datetime.now(),
)

assert event3.originator_id == originator_id
assert event3.originator_version == 3
assert isinstance(event3.timestamp, datetime)
```

Simple domain event object classes just like these can be used to model the actual occasions, or decisions, of a more sophisticated domain model. They are perfectly usable in a professional development project.

Furthermore, as part of an ordered sequence, they may contribute to determining the current state of the enduring and changing objects from which they originated, such as the event-sourced aggregates of Domain-Driven Design.

Chapter 2 Aggregate

Facts do not make sense alone. One thing leads to another. The ordinary physical objects we encounter in daily life enjoy adventures of change.

therefore…

Model enduring objects with a series of domain event objects that are triggered by command methods and projected into the current state of an aggregate.

In Chapter 1 we considered the creation of individual domain model event objects that represent individual decisions, or actual occasions, of the domain model. Domain model event objects were defined as both immutable objects that do not change and as entities that are uniquely identifiable.

The purpose of this chapter is to consider objects that enjoy adventures of change, and to model such things with series of actual occasions that do not change.

In this chapter, we will limit ourselves to the simple case of modelling ordinary physical objects and other concerns as being dominated by a single strand of personal order. This is the traditional scope of concern of the aggregates in *Domain-Driven Design*.

In *Domain-Driven Design* aggregates are defined as a cluster of entities and value objects. The aggregate has a 'root entity', which is the 'aggregate root', which is used to access the cluster of objects. The ID of the root entity is used as the ID of the aggregate, and is expected to be unique within its domain model. The unique identity of the aggregate is a 'defining characteristic' that all of its actual occasions inherit (for example as the `originator_id` of the domain event objects of event-sourced aggregates).

In *Domain-Driven Design*, atomic database transactions are used to make sure that either all or none of the changes to the state of an aggregate that result from a single client request are recorded. The use of atomic database transactions defines the 'consistency boundary' of an aggregate, so that the recorded state of the aggregate jumps from one desirable state to another, without the risk that intermediate states (which may be inconsistent but certainly aren't intended as the current state) will be recorded as the current state. An aggregate can't have two states simultaneously. Branching of the state of an aggregate is therefore not allowed. Hence, the atomic changes made to an aggregate have serial ordering. This is true whether or not the aggregate is event-sourced.

In the case of event sourcing, the state of an event-sourced aggregate is determined by a sequence of domain event objects. So that an event-sourced aggregate's sequence of domain event objects has definite positions, each domain event object must have a unique identity within the sequence. The sequence of positions can be reliably constructed and followed using counting, for example by adding one to the last position to get the next position. Hence, the position in the sequence can be understood as corresponding to the version of the aggregate that a domain event object will contribute to determining (the "next version" rather than the last).

The current state of an event-sourced aggregate is defined as a function of its already existing domain event objects. This function is often referred to as a 'projection', it is the aggregate's projection of its domain event objects. Whenever an aggregate needs to be reconstructed, its already existing domain event objects will be used, or projected, in the order they were created to obtain the current state of the aggregate. Each domain event object may cause many changes to the entities and value objects of the aggregate. The aggregate's projection must not trigger further domain event objects.

The state of an aggregate will be changed using 'command' methods and accessed using 'query' methods. Distinguishing between command methods and query method is a design pattern known as 'command-query separation', or CQS, a term coined by Bertrand Meyer and described in his book *Object-Oriented Software Construction*. Command methods will

change the state of an object, but will not return anything. Query methods will return particular aspects of the state, but will not change anything.

An aggregate command method uses the method arguments in conjunction with the current state of the aggregate to decide some new values that pertain to the state of the application. In the trivial case, the new values are simply the command arguments. In the special case of an event-sourced aggregate, these new values (the decision) will be encapsulated within a domain event object (see Chapter 1). The domain event object will be used to adjust the state of the aggregate and eventually to record the decision.

When an event-sourced aggregate that has been changed is eventually "saved", all of the domain event objects that are waiting to be recorded will need to be recorded together atomically. A single client request may result in the creation of many domain event objects, either by calling many command methods or by calling one command method that triggers more than one event. The aggregate's consistency boundary (its use of atomic database transactions to persist its data changes) means that either all of the domain event objects that result from a single client request will be recorded or none of them will. Hence, an aggregate should only be "saved" only once for any given client request.

It should be remembered, bearing in mind Whitehead's scheme, when developing a supportive domain model most of the ordinary physical objects are analysable into societies of actual occasions with one or many strands of enduring objects. So far, we have been concerned with the 'ideally simple case' of the enduring object with its personal order. But sometimes we will need to model an ordinary physical object as a corpuscular society with many strands of enduring objects. For such things, we will need to extend the notion of the consistency boundary to include many aggregates that are "saved" together in the same atomic database transaction. (As we shall see in Chapter 9, this need will be met by the `save()` method of an application object.)

The identity of an aggregate is usually coded as a UUID value. Rather than using an attribute of the model, it usually makes good sense to generate a random version-4 UUID that is independent of any other attribute of the aggregate, so that names and other attribute values that are perhaps initially

thought to be unique can become variable in a later phase of development. By making all the aggregate IDs in an event-sourced domain model share the same type of value, it is easier to store and retrieve the domain event objects of the aggregates of a domain model in the same way.

Where desirable, aggregate IDs can be generated from a namespace using version-5 UUIDs. This allows an aggregate to be identified by a unique name in a namespace. This can help to avoid needing to keep track of otherwise random UUIDs.

These two techniques, of generating a version-5 UUID for a particular position in a namespace, and a random version-4 UUID can be combined to create an index of aggregate IDs, allowing aggregates to be discovered by name, and for the names of an aggregate to be changed. In this arrangement, the indexed aggregates will use version-4 UUIDs and the index aggregates will use version-5 (namespaced) UUIDs. When updating the name of an aggregate, it would make good sense to "save" the named aggregate and the index aggregate together in the same atomic database transaction.

When designing aggregates, it can help to think about what needs to be deleted together. If two entities are usually deleted together, it can make sense to put them together in a single aggregate.

When designing event-sourced aggregates, care needs to be taken about the number of domain model events each will generate during its lifetime. Aggregates that generate a very large number of events may benefit from snapshotting, which is discussed in Chapter 7.

An aggregate has no dependencies except for necessarily depending on its domain event object classes.

For brevity in later chapters, I will use the term 'aggregate events' to refer to the immutable domain event objects of an aggregate.

In summary, event-sourced domain model aggregates:

- are comprised of entities and value objects

- have a unique identity within a domain model
- are changed by atomic and immutable decisions
- order their domain event objects serially
- position their domain event objects using counting
- are projections of existing domain event objects
- have no dependencies (except on their events)
- trigger domain event objects in command methods
- are "saved" once per client request in a consistency boundary

Example

In the example below, a bank account is modelled as an event-sourced aggregate. A base class for aggregates is firstly defined, and the bank account aggregate is defined by extending the aggregate base class. The domain event object classes used by these aggregate classes will extend the `DomainEvent` class defined in Chapter 1. However, the definition will be extended with the addition of a `mutate()` method that will be used to project the state of the domain event into the state of the aggregate.

The class `Aggregate` below defines a base class for aggregates. The nested class `Aggregate.Event` extends the base class `DomainEvent` (defined in Chapter 1). A `mutate()` method is defined, which is a convenient way of defining the aggregate projection. The `mutate()` method is called with the previous state of the aggregate and returns the subsequence state of the aggregate. It checks the `originator_version` value is actually the "next" version number, and then increments the `version` of the aggregate object. It also assigns the `timestamp` value as the `modified_on` attribute of the aggregate object.

The `mutate()` method may be called with `None` as the prior state, and it may return `None` as the subsequent state. For this reason, if this method is implemented in subclasses, those methods must call `super()` and must always return an object. For this reason, the `mutate()` method calls `apply()` which isn't expected to return a value and doesn't necessarily need to call `super()`. Concrete aggregate events classes can then apply their values to the aggregate in a more convenient way by overriding the `apply()` method

instead of extending the `mutate()` method to, since the implementation of the `apply()` method won't need to call `super()` and won't need a `return` statement.

The nested class `Aggregate.Created` extends the base class `Aggregate.Event`. The `mutate()` method is overridden, and resolves the `originator_topic` to an aggregate class, which is then constructed with an `id` and `version`, and all the remaining attributes of the event. The constructed aggregate object is then returned.

The `__init__()` method is defined so that instances will be initialised with an `id`, a `version`, and a `timestamp`. The list `pending_events` is also initialised, which will be used to collect new aggregate events that are waiting to be recorded.

The aggregate's class method `_create_()` implements a factory method for creating new aggregates, which works by firstly constructing a domain event object from the given `event_class` (an `Aggregate.Created` class) with an `originator_id`, an `originator_version` representing the initial version of the aggregate object, an `originator_topic` that represents the object class of the aggregate being created, along with any other method arguments. The event's `mutate()` method is called with a value of `None`, and the newly constructed aggregate object is returned. Finally, the event object is appended to a list of pending events that are waiting to be recorded.

```
class Aggregate:
    """
    Base class for aggregate roots.
    """

    class Event(DomainEvent):
        """
        Base domain event class for aggregates.
        """

        def mutate(
            self, obj: Optional["Aggregate"]
```

```
    ) -> Optional["Aggregate"]:
        """
        Changes the state of the aggregate
        according to domain event attributes.
        """
        # Check event is next in its sequence.
        # Use counting to follow the sequence.
        assert isinstance(obj, Aggregate)
        next_version = obj.version + 1
        if self.originator_version != next_version:
            raise obj.VersionError(
                self.originator_version, next_version
            )
        # Update the aggregate version.
        obj.version = next_version
        # Update the modified time.
        obj.modified_on = self.timestamp
        self.apply(obj)
        return obj

    def apply(self, obj) -> None:
        pass

class VersionError(Exception):
    pass

def __init__(
    self, id: UUID, version: int, timestamp: datetime
):
    """
    Aggregate is constructed with an 'id'
    and a 'version'. The 'pending_events'
    is also initialised as an empty list.
    """
    self.id = id
    self.version = version
    self.created_on = timestamp
```

```
        self.modified_on = timestamp
        self.pending_events: List[Aggregate.Event] = []

    @classmethod
    def _create_(
        cls,
        event_class: Type["Aggregate.Created"],
        **kwargs,
    ):
        """
        Factory method to construct a new
        aggregate object instance.
        """
        # Construct the domain event class,
        # with an ID and version, and the
        # a topic for the aggregate class.
        event = event_class(
            originator_id=uuid4(),
            originator_version=1,
            originator_topic=get_topic(cls),
            timestamp=datetime.now(),
            **kwargs,
        )
        # Construct the aggregate object.
        aggregate = event.mutate(None)
        # Append the domain event to pending list.
        aggregate.pending_events.append(event)
        # Return the aggregate.
        return aggregate

    class Created(Event):
        """
        Domain event for when aggregate is created.
        """

        originator_topic: str
```

```
    def mutate(
        self, obj: Optional["Aggregate"]
    ) -> "Aggregate":
        """
        Constructs aggregate instance defined
        by domain event object attributes.
        """
        # Copy the event attributes.
        kwargs = self.__dict__.copy()
        # Separate the id and version.
        id = kwargs.pop("originator_id")
        version = kwargs.pop("originator_version")
        # Get the aggregate root class from topic.
        aggregate_class = resolve_topic(
            kwargs.pop("originator_topic")
        )
        # Construct and return aggregate object.
        return aggregate_class(
            id=id, version=version, **kwargs
        )

def _trigger_(
    self,
    event_class: Type["Aggregate.Event"],
    **kwargs,
) -> None:
    """
    Triggers domain event of given type,
    extending the sequence of domain
    events for this aggregate object.
    """
    # Construct the domain event as the
    # next in the aggregate's sequence.
    # Use counting to generate the sequence.
    next_version = self.version + 1
    try:
        event = event_class(
```

```
                originator_id=self.id,
                originator_version=next_version,
                timestamp=datetime.now(),
                **kwargs,
            )
        except AttributeError:
            raise
        # Mutate aggregate with domain event.
        event.mutate(self)
        # Append the domain event to pending list.
        self.pending_events.append(event)

    def _collect_(self) -> List[Event]:
        """
        Collects pending events.
        """
        collected = []
        while self.pending_events:
            collected.append(self.pending_events.pop(0))
        return collected
```

The method `_trigger_()` will be used by command methods to create new domain event objects using the `id` of the aggregate as the `originator_id` of the aggregate event, the "next" `version` of the aggregate as the `originator_version`, and a `timestamp` value, along with any other method arguments. The new event's `mutate()` method is called with the previous state of the aggregate. The `_trigger_()` method appends to event object to the aggregate's list of pending events.

The `_collect_()` method drains the list of pending events and returns a list of the events that are waiting to be recorded.

The helper functions `get_topic()` and `resolve_topic()` are defined below. A "topic" represents the location of a class in the software code. For example, the `_create_()` method includes the aggreate root class as a topic in the `Created` method using `get_topic()`, and the `mutate()` method of the

`Created` domain event uses `resolve_topic()` to locate the aggregate root class. This allows the aggregate to be constructed by the `Created` event without any other information, which will be important after the domain events have been stored.

```
def get_topic(cls: type) -> str:
    """
    Returns a string that locates the given class.
    """
    return f"{cls.__module__}#{cls.__qualname__}"


def resolve_topic(topic: str) -> type:
    """
    Returns a class located by the given string.
    """
    module_name, _, class_name = topic.partition("#")
    module = importlib.import_module(module_name)
    return resolve_attr(module, class_name)


def resolve_attr(obj, path: str) -> type:
    if not path:
        return obj
    else:
        head, _, tail = path.partition(".")
        obj = getattr(obj, head)
        return resolve_attr(obj, tail)
```

We can now define the `BankAccount` class as a subclass of the base class `Aggregate`. It has a `balance` and an `overdraft_limit`. It also has nested domain event classes that extend the base class `DomainEvent`.

The class method `_create_()` is a factory method which creates new `BankAccount` objects using the `Created` event class, which is triggered along with the `full_name` method argument.

The method `append_transaction()` firstly checks the account is not closed, and then checks there is sufficient balance to guard against the account becoming overdrawn by the `amount`. Finally, a `TransactionAppended` event is triggered with the given `amount`.

The `TransactionAppended` event extends the `DomainEvent` class. It's `mutate()` method increments the balance by the `amount` of the transaction.

In a similar way, the method `set_overdraft_limit()` changes the `overdraft_limit` attribute of the account, by triggering the `OverdraftLimitSet` domain event.

The `close()` method sets the `is_closed` attribute to a true value, by triggering the `Closed` domain event.

```
class BankAccount(Aggregate):
    """
    Aggregate root for bank accounts.
    """

    def __init__(
        self, full_name: str, email_address: str, **kwargs
    ):
        super().__init__(**kwargs)
        self.full_name = full_name
        self.email_address = email_address
        self.balance = Decimal("0.00")
        self.overdraft_limit = Decimal("0.00")
        self.is_closed = False

    @classmethod
    def open(
        cls, full_name: str, email_address: str
    ) -> "BankAccount":
        """
        Creates new bank account object.
        """
```

```
        return super()._create_(
            cls.Opened,
            full_name=full_name,
            email_address=email_address,
        )

    class Opened(Aggregate.Created):
        full_name: str
        email_address: str

    def append_transaction(self, amount: Decimal) -> None:
        """
        Appends given amount as transaction on account.
        """
        self.check_account_is_not_closed()
        self.check_has_sufficient_funds(amount)
        self._trigger_(
            self.TransactionAppended,
            amount=amount,
        )

    def check_account_is_not_closed(self) -> None:
        if self.is_closed:
            raise AccountClosedError(
                {"account_id": self.id}
            )

    def check_has_sufficient_funds(
        self, amount: Decimal
    ) -> None:
        if self.balance + amount < -self.overdraft_limit:
            raise InsufficientFundsError(
                {"account_id": self.id}
            )

    class TransactionAppended(Aggregate.Event):
        """
```

```
        Domain event for when transaction
        is appended to bank account.
        """

        amount: Decimal

        def apply(self, account: "BankAccount") -> None:
            """
            Increments the account balance.
            """
            account.balance += self.amount

    def set_overdraft_limit(
        self, overdraft_limit: Decimal
    ) -> None:
        """
        Sets the overdraft limit.
        """
        # Check the limit is not a negative value.
        assert overdraft_limit >= Decimal("0.00")
        self.check_account_is_not_closed()
        self._trigger_(
            self.OverdraftLimitSet,
            overdraft_limit=overdraft_limit,
        )

    class OverdraftLimitSet(Aggregate.Event):
        """
        Domain event for when overdraft
        limit is set.
        """

        overdraft_limit: Decimal

        def apply(self, account: "BankAccount"):
            account.overdraft_limit = self.overdraft_limit
```

```
    def close(self) -> None:
        """
        Closes the bank account.
        """
        self._trigger_(self.Closed)

    class Closed(Aggregate.Event):
        """
        Domain event for when account is closed.
        """

        def apply(self, account: "BankAccount"):
            account.is_closed = True


class AccountClosedError(Exception):
    """
    Raised when attempting to operate a closed account.
    """


class InsufficientFundsError(Exception):
    """
    Raised when attempting to go past overdraft limit.
    """
```

Below, an account is created, and the account is credited and debited. A debit that would make the account overdrawn results in a `InsufficientFundsError` error. Then, the overdraft limit is increased, and a subsequent attempt to debit the account succeeds and results in a negative balance. Then, the account is closed. A subsequent attempt to debit the account results in an `AccountClosed` error.

```
# Open an account.
account = BankAccount.open(
    full_name="Alice",
    email_address="alice@example.com",
```

```
)

# Check the full name and email address.
assert account.full_name == "Alice"
assert account.email_address == "alice@example.com"

# Check the initial balance.
assert account.balance == 0

# Credit the account.
account.append_transaction(Decimal("10.00"))

# Check the balance.
assert account.balance == Decimal("10.00")

# Credit the account again.
account.append_transaction(Decimal("10.00"))

# Check the balance.
assert account.balance == Decimal("20.00")

# Debit the account.
account.append_transaction(Decimal("-15.00"))

# Check the balance.
assert account.balance == Decimal("5.00")

# Fail to debit account (insufficient funds).
try:
    account.append_transaction(Decimal("-15.00"))
except InsufficientFundsError:
    pass
else:
    raise Exception("Insufficient funds error not raised")

# Increase the overdraft limit.
account.set_overdraft_limit(Decimal("100.00"))
```

```
# Debit the account.
account.append_transaction(Decimal("-15.00"))

# Check the balance.
assert account.balance == Decimal("-10.00")

# Close the account.
account.close()

# Fail to debit account (account closed).
try:
    account.append_transaction(Decimal("-15.00"))
except AccountClosedError:
    pass
else:
    raise Exception("Account closed error not raised")

# Collect pending events.
pending = account._collect_()
assert len(pending) == 7
```

Chapter 3 Mapper

Different events must be stored together in a common format.

therefore…

Convert domain events to stored events using a mapper.

An event mapper converts from domain event objects of different types to a standard format for storing any domain event. Using different types of domain events allows for a variety of domain events in the domain model. The common format for storing domain events allows different databases to be more easily adapted, since they only need to support the common format.

An event mapper must have at least two methods. One method will convert domain event objects to the stored event type. The other method will convert the stored event objects to the original domain event type.

The stored event type needs to have attributes that hold information about the type and the state of the domain event. It also needs to have information to identify the domain event aggregate's sequence and the position of the stored event in its sequence.

An important concern of a mapper is the serialisation and deserialisation of the state of the domain event. In particular, a domain model will usually involve custom types that will need to be serialised and deserialised. One approach is to use an extensible transcoder. The transcoder can be configured with individual transcoding strategies for non-standard types that support the custom types.

The mapper is a good place for compressing and encrypting the serialised state of the domain event. Compression decreases the size of the serialised state of the domain event, and can speed up the transporting of serialised state to and from a database. Encryption increases the size of the serialised

state, but by encrypting the state within the mapper, it will be encrypted within the application.

Encrypting domain events within the application this implements "application-level encryption", so that data will be encrypted in transit across a network ("on the wire") and at disk level including backups ("at rest"), which is a legal requirement in some jurisdictions when dealing with personally identifiable information (PII) for example the EU's GDPR.

In summary, an event mapper:

- Maps different types of domain event to one stored event type
- Supports extensible serialisation of custom model object types
- Serialised state can be compressed and encrypted

Example

The example below shows how an event mapper can be used to map from domain events to stored event objects.

Let's firstly define a common object type for storing events. The object class `StoredEvent` below has four attributes. The `originator_id` represents the `originator_id` of a domain event object. The `originator_version` represents the `originator_version` of a domain event object.

```
class StoredEvent(ImmutableObject):
    originator_id: uuid.UUID
    originator_version: int
    topic: str
    state: bytes
```

The `topic` represents the type of the domain event object. And the `state` represents the state of the domain event object.

Firstly, so that transcoding is customisable within a mapper, let's define an abstract base class for transcoders.

The abstract base class `AbstractTranscoder` defined below has abstract methods `encode()` and `decode()` that will be overridden by concrete transcoders to convert any type of object to bytes, and from bytes to the original type of object.

```
class AbstractTranscoder(ABC):
    @abstractmethod
    def encode(self, o: Any) -> bytes:
        pass

    @abstractmethod
    def decode(self, d: bytes) -> Any:
        pass
```

Next, let's define an extensible JSON transcoder. Our transcoder will be extensible by defining and registering custom transcoding objects. The `Transcoding` class below can be subclassed to define encodings for custom types of objects, for example custom value objects that are used in a domain model and consequently included in the state of its domain event objects.

```
class Transcoding(ABC):
    @property
    @abstractmethod
    def type(self) -> type:
        pass

    @property
    @abstractmethod
    def name(self) -> str:
        pass

    @abstractmethod
    def encode(self, o: Any) -> Union[str, dict]:
        pass

    @abstractmethod
```

```
    def decode(self, d: Union[str, dict]) -> Any:
        pass
```

The properties `type` and `name` will be overridden by concrete transcodings. The `type` will be the object class supported by a concrete transcoding. The `name` will combine the name of the type and the style of its encoding. The abstract method `encode()` will be overridden by concete transcodings to convert the object class to a Python `str` or a Python `dict` in a style that is suitable for the type. Simple types will be converted to a `str`. More complex types can be converted to a `dict`. Converting to a `dict` may involve returning with unconverted custom types that will subsequently be converted by their custom transcodings. The method `decode` will be overridden to reverse the `encode` method.

The class `Transcoder` defined below uses the JSON module from the Python Standard Library to encode Python objects as JSON strings that are encoded as UTF-8 byte strings.

It uses an extensible collection of transcoding objects to convert non-standard object types that can't be handled by the JSON encoder and decoder to a particular Python `dict` that is unlikely to figure in normal usage. Its method `register()` is used to register custom transcoding objects. The methods `encode()` and `decode()` are used to encode and decode the state of domain event objects.

```
class Transcoder(AbstractTranscoder):
    def __init__(self):
        self.types: Dict[type, Transcoding] = {}
        self.names: Dict[str, Transcoding] = {}
        self.encoder = json.JSONEncoder(
            default=self._encode_dict
        )
        self.decoder = json.JSONDecoder(
            object_hook=self._decode_dict
        )

    def register(self, transcoding: Transcoding):
```

```
    self.types[transcoding.type] = transcoding
    self.names[transcoding.name] = transcoding

def encode(self, o: Any) -> bytes:
    return self.encoder.encode(o).encode("utf8")

def decode(self, d: bytes) -> Any:
    return self.decoder.decode(d.decode("utf8"))

def _encode_dict(
    self, o: Any
) -> Dict[str, Union[str, dict]]:
    try:
        transcoding = self.types[type(o)]
    except KeyError:
        raise TypeError(
            f"Object of type "
            f"{o.__class__.__name__} "
            f"is not serializable"
        )
    else:
        return {
            "__type__": transcoding.name,
            "__data__": transcoding.encode(o),
        }

def _decode_dict(
    self, d: Dict[str, Union[str, dict]]
) -> Any:
    if set(d.keys()) == {
        "__type__",
        "__data__",
    }:
        t = d["__type__"]
        t = cast(str, t)
        transcoding = self.names[t]
        return transcoding.decode(d["__data__"])
```

```
        else:
            return d
```

We can now construct a transcoder using the `Transcoder` object class defined above.

```
transcoder = Transcoder()
```

Below, individual transcodings are defined for `UUID`, `Decimal` and `datetime` objects.

```
class UUIDAsHex(Transcoding):
    type = UUID
    name = "uuid_hex"

    def encode(self, o: UUID) -> str:
        return o.hex

    def decode(self, d: Union[str, dict]) -> UUID:
        assert isinstance(d, str)
        return UUID(d)


class DecimalAsStr(Transcoding):
    type = Decimal
    name = "decimal_str"

    def encode(self, o: Decimal):
        return str(o)

    def decode(self, d: Union[str, dict]) -> Decimal:
        assert isinstance(d, str)
        return Decimal(d)


class DatetimeAsISO(Transcoding):
    type = datetime
    name = "datetime_iso"
```

```
    def encode(self, o: datetime) -> str:
        return o.isoformat()

    def decode(self, d: Union[str, dict]) -> datetime:
        assert isinstance(d, str)
        return datetime.fromisoformat(d)
```

The custom transcoding objects defined above can be registered with the transcoder.

```
transcoder.register(UUIDAsHex())
transcoder.register(DecimalAsStr())
transcoder.register(DatetimeAsISO())
```

The transcoder can then encode objects that contain a `UUID`, or a `Decimal` or a `datetime`. Decoding the bytes gives an exact copy of the original object.

```
from uuid import uuid4
from decimal import Decimal
from datetime import datetime

a_uuid = uuid4()
a_decimal = Decimal("12.34")
a_datetime = datetime.now()

a_dict = {
    "a_uuid": a_uuid,
    "a_decimal": a_decimal,
    "a_datetime": a_datetime,
}

data = transcoder.encode(a_dict)
assert isinstance(data, bytes)

copy = transcoder.decode(data)
assert copy == a_dict
```

Next, let's define a mapper class. The `Mapper` class below is constructed with a transcoder, an optional compressor, and an optional cipher.

```
class Mapper(Generic[TDomainEvent]):
    def __init__(
        self,
        transcoder: AbstractTranscoder,
        compressor=None,
        cipher=None,
    ):
        self.transcoder = transcoder
        self.cipher = cipher
        self.compressor = compressor

    def from_domain_event(
        self, domain_event: TDomainEvent
    ) -> StoredEvent:
        topic: str = get_topic(domain_event.__class__)
        d = copy(domain_event.__dict__)
        d.pop("originator_id")
        d.pop("originator_version")
        state: bytes = self.transcoder.encode(d)
        if self.compressor:
            state = self.compressor.compress(state)
        if self.cipher:
            state = self.cipher.encrypt(state)
        return StoredEvent(
            domain_event.originator_id,
            domain_event.originator_version,
            topic,
            state,
        )

    def to_domain_event(
        self, stored: StoredEvent
    ) -> TDomainEvent:
        state: bytes = stored.state
```

```
        if self.cipher:
            state = self.cipher.decrypt(state)
        if self.compressor:
            state = self.compressor.decompress(state)
        d = self.transcoder.decode(state)
        d["originator_id"] = stored.originator_id
        d["originator_version"] = stored.originator_version
        cls = resolve_topic(stored.topic)
        assert issubclass(cls, DomainEvent)
        domain_event: TDomainEvent = object.__new__(cls)
        domain_event.__dict__.update(d)
        return domain_event
```

The method `from_domain_event()` maps a domain event object to a new stored event object, and the method `to_domain_event()` maps a stored event object to new domain event object. The helper functions `get_topic()` and `resolve_topic()` were introduced in Chapter 2.

We can now construct an event mapper using the `Mapper` object class defined above. The `mapper` object below has neither a compressor nor a cipher.

```
mapper = Mapper(transcoder)
```

We can then use the event mapper to map between domain event objects and stored events. The `TransactionAppended` event of aggregate `BankAccount` was introduced in Chapter 2.

```
# Create a domain event.
domain_event = BankAccount.TransactionAppended(
    originator_id=uuid.uuid4(),
    originator_version=123456789,
    timestamp=datetime.now(),
    amount=Decimal("10.00")
)
```

```
# Map from domain event to stored event.
stored_event = mapper.from_domain_event(domain_event)
assert isinstance(stored_event, StoredEvent)

# Map to domain event.
copy = mapper.to_domain_event(stored_event)

# Check copy has correct values.
assert copy.originator_id == domain_event.originator_id
assert copy.originator_version ==
domain_event.originator_version
assert copy.timestamp == domain_event.timestamp, copy.timestamp
assert copy.amount == domain_event.amount
```

Because we didn't use a cipher, the values of the domain event are visible in the stored event.

```
# Check values are visible.
assert "10.00" in str(stored_event.state)
```

This plaintext mapping gives a stored event that has 145 bytes of state.

```
assert len(stored_event.state) == 145, len(stored_event.state)
```

To encrypt the values of the domain event at rest and in transit, the mapper can be constructed with a cipher. The domain event will then be mapped to a stored event that will keep the domain event attributes secret.

The `AES` class is used from the `pycryptodome` package.

```
class AESCipher(object):
    """
    Cipher strategy that uses Crypto
    library AES cipher in GCM mode.
    """
```

```
def __init__(self, cipher_key: bytes):
    """
    Initialises AES cipher with ``cipher_key``.

    :param cipher_key: 16, 24, or 32 random bytes
    """
    assert len(cipher_key) in [16, 24, 32]
    self.cipher_key = cipher_key

def encrypt(self, plaintext: bytes) -> bytes:
    """Return ciphertext for given plaintext."""

    # Construct AES-GCM cipher, with 96-bit nonce.
    nonce = random_bytes(12)
    cipher = self.construct_cipher(nonce)

    # Encrypt and digest.
    result = cipher.encrypt_and_digest(plaintext)
    encrypted = result[0]
    tag = result[1]

    # Combine with nonce.
    ciphertext = nonce + tag + encrypted

    # Return ciphertext.
    return ciphertext

def construct_cipher(self, nonce: bytes) -> GcmMode:
    cipher = AES.new(
        self.cipher_key,
        AES.MODE_GCM,
        nonce,
    )
    assert isinstance(cipher, GcmMode)
    return cipher

def decrypt(self, ciphertext: bytes) -> bytes:
```

```
        """Return plaintext for given ciphertext."""

        # Split out the nonce, tag, and encrypted data.
        nonce = ciphertext[:12]
        if len(nonce) != 12:
            raise Exception(
                "Damaged cipher text: invalid nonce length"
            )

        tag = ciphertext[12:28]
        if len(tag) != 16:
            raise Exception(
                "Damaged cipher text: invalid tag length"
            )
        encrypted = ciphertext[28:]

        # Construct AES cipher, with old nonce.
        cipher = self.construct_cipher(nonce)

        # Decrypt and verify.
        try:
            plaintext = cipher.decrypt_and_verify(
                encrypted, tag
            )
        except ValueError as e:
            raise ValueError(
                "Cipher text is damaged: {}".format(e)
            )
        return plaintext


def random_bytes(num_bytes: int) -> bytes:
    return os.urandom(num_bytes)
```

We can now construct the cipher and use it with an event mapper to map domain model events to encrypted stored events.

```
# Construct cipher.
cipher = AESCipher(cipher_key=b"0123456789abcdef")

# Construct mapper with cipher.
mapper = Mapper(
    transcoder=transcoder,
    cipher=cipher
)

# Map from domain event.
stored_event = mapper.from_domain_event(domain_event)

# Check values are not visible.
assert "123456789" not in str(stored_event.state)

# Check decrypted copy has correct values.
assert copy.originator_id == domain_event.originator_id
assert copy.originator_version ==
domain_event.originator_version
```

This encrypted mapping gives a stored event that has 173 bytes of state. Encrypted stored events will always be slightly larger.

```
assert len(stored_event.state) == 173, len(stored_event.state)
```

To reduce the size of the stored event state, the mapper can be constructed with a compressor.

```
import zlib

# Construct mapper with cipher and compressor.
mapper = Mapper(
    transcoder=transcoder,
    cipher=cipher,
    compressor=zlib
)
```

```
# Map from domain event.
stored_event = mapper.from_domain_event(domain_event)

# Check decompressed copy has correct values.
assert copy.originator_id == domain_event.originator_id
assert copy.originator_version ==
domain_event.originator_version
```

This compressed and encrypted stored event has around 139 bytes of state, which is less that the total size of the unserialized domain object. Larger domain event objects are more likely to be compressed with a greater compression ratio than smaller ones.

```
assert len(stored_event.state) in [135, 136, 137, 138, 139,
140], len(stored_event.state)
```

It's also possible to serialise with a compressor but without a cipher.

```
# Construct mapper with cipher and compressor.
mapper = Mapper(
    transcoder=transcoder,
    compressor=zlib
)

# Map from domain event.
stored_event = mapper.from_domain_event(domain_event)

# Check decompressed copy has correct values.
assert copy.originator_id == domain_event.originator_id
assert copy.originator_version ==
domain_event.originator_version
```

This compressed but unencrypted stored event has around 111 bytes of state.

```
assert len(stored_event.state) in [107, 108, 109, 110, 111,
112], len(stored_event.state)
```

Chapter 4 Recorder

Stored event objects can be recorded in database management systems. But different database systems work in different ways.

therefore…

Adapt database systems to a common interface for recording stored events.

Different database systems work in different ways, but they can be adapted to a common interface for recording and retrieving stored events. Recorders implement this common interface. Each recorder records and retrieves stored events using a particular database management system, or a particular abstraction of database management systems such as an object-relational mapper, or indeed a lower level abstraction across a family of database management systems.

In order to retrieve the stored events of a particular aggregate, stored events must be recorded in a sequence that corresponds to the aggregate which originated the event that are being stored. So that the stored events can be retrieved in the same order as they were created, new stored events must be appended to the end of their aggregate sequence.

Each stored event occupies a position in its aggregate sequence. Occupation of each position must be unique, and an attempt to overwrite an already occupied position must fail.

A recorder must be able to write stored events atomically, so that each stored event is recorded completely or not at all. If many stored events need to be recorded together for some reason, then either all or none of them are recorded.

So that the aggregate sequence has definite positions, each domain event notification must have a unique identity within the sequence. Positioning domain events in an aggregate sequence with a contiguous sequence of

integers satisfies both the desire for the identities to increase, and the desire for the sequence to have no gaps. The sequence can be reliably constructed and followed using counting.

When retrieving stored events, it is sometimes desirable to select only some of the stored events in an aggregate sequence. For example, it can be useful to be able to select only the stored events from a particular position. It is also useful to be able to get the last stored event. These two abilities are needed for snapshotting which is discussed in Chapter 7.

In summary, a recorder:

- Adapts a particular database management system
- Reads and writes stored events in aggregate sequences
- Uses atomic transactions (everything or nothing)
- Ensures unique occupation of each position in each sequence

The limitation recording each stored event in only one aggregate sequence is that the stored events can only be retrieved by querying for the events of a particular aggregate. The total ordering of the stored events is not available.

Example

The abstract base class `Recorder` defines common recorder exception classes.

```
class Recorder(ABC):
    class OperationalError(Exception):
        pass

    class IntegrityError(Exception):
        pass
```

The common interface for recording stored events is defined by the abstract base class `AggregateRecorder`.

The `insert_events()` method will by called with a list of stored events. The `select_events()` method will be called with an `originator_id` a return a list of stored events.

The stored event objects passed into the method `insert_events()` are expected to be instances of object class `StoredEvent`, which was defined in Chapter 3. Similarly, the objects returned by `select_events()` are also instances of `StoredEvent`. The events selected by `select_events()` will be from a sequence identified by the argument `originator_id` which corresponds to the attribute `originator_id` of the dataclass `StoredEvent`. The arguments `gt` and `lte` indicate that events greater than and less than or equal to a position in the sequence are to be selected. The boolen argument `desc` will indicate that the selection should in descending order if a `True` value is passed. By default, the selection will be made in ascending order. The argument `limit` will limit the number of selected events.

```
class AggregateRecorder(Recorder):
    @abstractmethod
    def insert_events(
        self,
        stored_events: List[StoredEvent],
        **kwargs,
    ) -> None:
        """
        Writes stored events into database.
        """

    @abstractmethod
    def select_events(
        self,
        originator_id: UUID,
        gt: Optional[int] = None,
        lte: Optional[int] = None,
        desc: bool = False,
        limit: Optional[int] = None,
    ) -> List[StoredEvent]:
        """
```

```
        Reads stored events from database.
        """
```

Before implementing the abstract base class, let's firstly define a test for aggregate recorders. The test below inserts and selects stored events, and checks that events can't be overwritten. It also checks that we can select a limited number of events, events in a particular range, and in ascending and descending order.

```
from uuid import uuid4

def test(recorder: AggregateRecorder):

    # Write two stored events.
    originator_id = uuid4()
    stored_event1 = StoredEvent(
        originator_id=originator_id,
        originator_version=1,
        topic='topic1',
        state=b'state1'
    )
    stored_event2 = StoredEvent(
        originator_id=originator_id,
        originator_version=2,
        topic='topic2',
        state=b'state2'
    )

    recorder.insert_events([
        stored_event1, stored_event2
    ])

    results = recorder.select_events(originator_id)

    # Check we got what was written.
    assert len(results) == 2
    assert results[0].originator_id == originator_id
```

```
assert results[0].originator_version == 1
assert results[0].topic == 'topic1'
assert results[0].state == b'state1'
assert results[1].originator_id == originator_id
assert results[1].originator_version == 2
assert results[1].topic == 'topic2'
assert results[1].state == b'state2'

# Check recorded events are unique.
stored_event3 = StoredEvent(
    originator_id=originator_id,
    originator_version=3,
    topic='topic3',
    state=b'state3'
)

# Check events can't be overwritten.
try:
    recorder.insert_events([
        stored_event2, stored_event3
    ])
except Recorder.IntegrityError:
    pass

# Check writing of events is atomic.
results = recorder.select_events(originator_id)
assert len(results) == 2

# Check the third event can be written.
recorder.insert_events([stored_event3])
results = recorder.select_events(originator_id)
assert len(results) == 3
assert results[2].originator_id == originator_id
assert results[2].originator_version == 3
assert results[2].topic == 'topic3'
assert results[2].state == b'state3'
```

```
    # Check we can get events after the first.
    results = recorder.select_events(
        originator_id, gt=1
    )
    assert results[0].originator_id == originator_id
    assert results[0].originator_version == 2
    assert results[1].originator_id == originator_id
    assert results[1].originator_version == 3

    # Check we can get the last event.
    results = recorder.select_events(
        originator_id, limit=1, desc=True
    )
    assert len(results) == 1
    assert results[0].originator_id == originator_id
    assert results[0].originator_version == 3
```

POPO

Let's define an aggregate recorder which simply keeps the stored events objects in memory using "Plain Old Python Objects" (POPO). The recorder class `POPOAggregateRecorder` offers a simple way to "record" stored events when developing an application.

This recoder appends stored event objects to a list called `stored_events`. But it also maintains an index, called `stored_events_index` of the aggregate sequences, so that uniqueness can be checked. The method `insert_events()` firstly calls `assert_uniqueness()` and then `update_table()` so that if there are any conflicts then none of the stored events will be inserted. The threading lock `database_lock` is used to serialise access from multiple threads.

```
class POPOAggregateRecorder(AggregateRecorder):
    def __init__(self) -> None:
        self.stored_events: List[StoredEvent] = []
        self.stored_events_index: Dict[
            UUID, Dict[int, int]
```

```
        ] = defaultdict(dict)
        self.database_lock = Lock()

    def insert_events(
        self,
        stored_events: List[StoredEvent],
        **kwargs,
    ) -> None:
        with self.database_lock:
            self.assert_uniqueness(stored_events, **kwargs)
            self.update_table(stored_events, **kwargs)

    def assert_uniqueness(
        self, stored_events: List[StoredEvent], **kwargs
    ) -> None:
        for s in stored_events:
            if (
                s.originator_version
                in self.stored_events_index[
                    s.originator_id
                ]
            ):
                raise self.IntegrityError

    def update_table(
        self, stored_events: List[StoredEvent], **kwargs
    ) -> None:
        for s in stored_events:
            self.stored_events.append(s)
            self.stored_events_index[s.originator_id][
                s.originator_version
            ] = (len(self.stored_events) - 1)

    def select_events(
        self,
        originator_id: UUID,
        gt: Optional[int] = None,
```

```
        lte: Optional[int] = None,
        desc: bool = False,
        limit: Optional[int] = None,
    ) -> List[StoredEvent]:

        with self.database_lock:
            results = []

            index = self.stored_events_index[originator_id]
            positions: Iterable = index.keys()
            if desc:
                positions = reversed(list(positions))
            for p in positions:
                if gt is not None:
                    if not p > gt:
                        continue
                if lte is not None:
                    if not p <= lte:
                        continue
                s = self.stored_events[index[p]]
                results.append(s)
                if len(results) == limit:
                    break
            return results
```

Now let's run the test with the "Plain Old Python Objects" aggregate recorder.

```
# Construct the POPO recorder.
recorder = POPOAggregateRecorder()

# Run the test.
test(recorder)
```

It should be obvious that the "Plain Old Python Objects" recorder won't persist state permanently and so it is only suitable for development and not

for production.

SQLite

Let's define an aggregate recorder that uses SQLite. Below, the class `SQLiteAggregateRecorder` implements `AggregateRecorder`.

A database table for stored events is created by the method `create_table()`. This table is defined with a primary key that combines the `originator_id` and `originator_version` fields. The other fields, `topic` and `state`, are used to record the topic and the state of the stored event.

The methods `insert_events()` and `select_events()` are implemented as parameterised SQL statements.

```
class SQLiteDatabase:
    def __init__(self, db_uri):
        self.db_uri = db_uri
        self.connections = {}

    class Transaction:
        def __init__(self, connection: Connection):
            self.c = connection

        def __enter__(self) -> Connection:
            # We must issue a "BEGIN" explicitly
            # when running in auto-commit mode.
            self.c.execute("BEGIN")
            return self.c

        def __exit__(self, exc_type, exc_val, exc_tb):
            if exc_type:
                # Roll back all changes
                # if an exception occurs.
                self.c.rollback()
            else:
                self.c.commit()
```

```
    def transaction(self) -> Transaction:
        return self.Transaction(self.get_connection())

    def get_connection(self) -> Connection:
        thread_id = threading.get_ident()
        try:
            return self.connections[thread_id]
        except KeyError:
            c = self.create_connection()
            self.connections[thread_id] = c
            return c

    def create_connection(self) -> Connection:
        # Make a connection to an SQLite database.
        c = sqlite3.connect(
            database=self.db_uri,
            uri=True,
            check_same_thread=False,
            isolation_level=None,  # Auto-commit mode.
            cached_statements=True,
        )
        c.row_factory = sqlite3.Row
        # Use WAL (write-ahead log) mode.
        c.execute("pragma journal_mode=wal;")
        return c


class SQLiteAggregateRecorder(AggregateRecorder):
    def __init__(
        self,
        db: SQLiteDatabase,
        table_name: str = "stored_events",
    ):
        assert isinstance(db, SQLiteDatabase)
        self.db = db
        self.table_name = table_name
```

```
def create_table(self):
    with self.db.transaction() as c:
        self._create_table(c)

def _create_table(self, c: Connection):
    statement = (
        "CREATE TABLE IF NOT EXISTS "
        f"{self.table_name} ("
        "originator_id TEXT, "
        "originator_version INTEGER, "
        "topic TEXT, "
        "state BLOB, "
        "PRIMARY KEY "
        "(originator_id, originator_version))"
    )
    try:
        c.execute(statement)
    except sqlite3.OperationalError as e:
        raise self.OperationalError(e)

def insert_events(self, stored_events, **kwargs):
    with self.db.transaction() as c:
        self._insert_events(c, stored_events, **kwargs)

def _insert_events(
    self,
    c: Connection,
    stored_events: List[StoredEvent],
    **kwargs,
) -> None:
    statement = (
        f"INSERT INTO {self.table_name}"
        " VALUES (?,?,?,?)"
    )
    params = []
    for stored_event in stored_events:
        params.append(
```

```
                (
                    stored_event.originator_id.hex,
                    stored_event.originator_version,
                    stored_event.topic,
                    stored_event.state,
                )
            )
        try:
            c.executemany(statement, params)
        except sqlite3.IntegrityError as e:
            raise self.IntegrityError(e)

    def select_events(
        self,
        originator_id: UUID,
        gt: Optional[int] = None,
        lte: Optional[int] = None,
        desc: bool = False,
        limit: Optional[int] = None,
    ) -> List[StoredEvent]:
        statement = (
            "SELECT * "
            f"FROM {self.table_name} "
            "WHERE originator_id=? "
        )
        params: List[Any] = [originator_id.hex]
        if gt is not None:
            statement += "AND originator_version>? "
            params.append(gt)
        if lte is not None:
            statement += "AND originator_version<=? "
            params.append(lte)
        statement += "ORDER BY originator_version "
        if desc is False:
            statement += "ASC "
        else:
            statement += "DESC "
```

```
        if limit is not None:
            statement += "LIMIT ? "
            params.append(limit)
        c = self.db.get_connection()
        stored_events = []
        for row in c.execute(statement, params):
            stored_events.append(
                StoredEvent(
                    originator_id=UUID(
                        row["originator_id"]
                    ),
                    originator_version=row[
                        "originator_version"
                    ],
                    topic=row["topic"],
                    state=row["state"],
                )
            )
        return stored_events
```

Statements are executed in a separate connection for each thread, which takes advantage of SQLite's file locking mechanism to control concurrency. SQLite is used in its "WAL" mode which means that reading can be concurrent with writing (although there can only be one writer). The constructor argument `db_uri` defines the SQLite database URI (normally a file path, but also `:memory:` or `'file::memory:` can be used for in-memory databases).

Now let's run the test with the SQLite aggregate recorder.

```
# Construct the SQLite recorder.
recorder = SQLiteAggregateRecorder(
    SQLiteDatabase(':memory:')
)

# Create tables.
recorder.create_table()
```

```
# Run the test.
test(recorder)
```

Whilst the SQLite database can be used in production with a real file, and also the in-memory mode can be used for development, when it comes to accessing an in-memory SQLite database with multiple threads, we would need to use the "shared cache" feature. However, there are differences in the locking behaviour when using a shared cache. Although locking could be selectively applied when in-memory mode is detected, that adds complexity to the simple implementation. Of course, it's possible to use a real file on a RAM disk. But for these reasons, and also for extra performance, there is added value in implementing an aggregate recorder which simply keeps stored events in memory without using a database management system at all.

PostgreSQL

Let's define an aggregate recorder that uses PostgreSQL. Below, the class `PostgresAggregateRecorder` implements `AggregateRecorder`. It's very similar to the SQLite recorder.

```
class PostgresDatabase:
    def __init__(self):
        self.connections = {}

    def get_connection(self) -> connection:
        thread_id = threading.get_ident()
        try:
            return self.connections[thread_id]
        except KeyError:
            c = self.create_connection()
            self.connections[thread_id] = c
            return c

    def create_connection(self) -> connection:
```

```
    # Make a connection to a Postgres database.
    dbname = os.getenv("POSTGRES_DBNAME", "dddbb")
    host = os.getenv("POSTGRES_HOST", "127.0.0.1")
    user = os.getenv("POSTGRES_USER", "eventsourcing")
    password = os.getenv(
        "POSTGRES_PASSWORD", "eventsourcing"
    )

    c = psycopg2.connect(
        dbname=dbname,
        host=host,
        user=user,
        password=password,
    )
    return c

class Transaction:
    def __init__(self, connection: connection):
        self.c = connection

    def __enter__(self) -> cursor:
        cursor = self.c.cursor(
            cursor_factory=psycopg2.extras.DictCursor
        )
        # cursor.execute("BEGIN")
        return cursor

    def __exit__(self, exc_type, exc_val, exc_tb):
        if exc_type:
            # Roll back all changes
            # if an exception occurs.
            self.c.rollback()
        else:
            self.c.commit()

def transaction(self) -> Transaction:
    return self.Transaction(self.get_connection())
```

```
class PostgresAggregateRecorder(AggregateRecorder):
    def __init__(
        self,
        application_name: str = "",
    ):
        self.application_name = application_name
        self.db = PostgresDatabase()
        self.events_table = (
            application_name.lower() + "events"
        )

    def create_table(self):
        with self.db.transaction() as c:
            self._create_table(c)

    def _create_table(self, c: cursor):
        statement = (
            "CREATE TABLE IF NOT EXISTS "
            f"{self.events_table} ("
            "originator_id uuid NOT NULL, "
            "originator_version integer NOT NULL, "
            "topic text, "
            "state bytea, "
            "PRIMARY KEY "
            "(originator_id, originator_version))"
        )
        try:
            c.execute(statement)
        except psycopg2.OperationalError as e:
            raise self.OperationalError(e)

    def insert_events(self, stored_events, **kwargs):
        with self.db.transaction() as c:
            self._insert_events(c, stored_events, **kwargs)

    def _insert_events(
        self,
```

```
    c: cursor,
    stored_events: List[StoredEvent],
    **kwargs,
) -> None:
    statement = (
        f"INSERT INTO {self.events_table}"
        " VALUES (%s, %s, %s, %s)"
    )
    params = []
    for stored_event in stored_events:
        params.append(
            (
                stored_event.originator_id,
                stored_event.originator_version,
                stored_event.topic,
                stored_event.state,
            )
        )
    try:
        c.executemany(statement, params)
    except psycopg2.IntegrityError as e:
        raise self.IntegrityError(e)

def select_events(
    self,
    originator_id: UUID,
    gt: Optional[int] = None,
    lte: Optional[int] = None,
    desc: bool = False,
    limit: Optional[int] = None,
) -> List[StoredEvent]:
    statement = (
        "SELECT * "
        f"FROM {self.events_table} "
        "WHERE originator_id = %s "
    )
    params: List[Any] = [originator_id]
```

```
        if gt is not None:
            statement += "AND originator_version > %s "
            params.append(gt)
        if lte is not None:
            statement += "AND originator_version <= %s "
            params.append(lte)
        statement += "ORDER BY originator_version "
        if desc is False:
            statement += "ASC "
        else:
            statement += "DESC "
        if limit is not None:
            statement += "LIMIT %s "
            params.append(limit)
        # statement += ";"
        stored_events = []
        with self.db.transaction() as c:
            c.execute(statement, params)
            for row in c.fetchall():
                stored_events.append(
                    StoredEvent(
                        originator_id=row["originator_id"],
                        originator_version=row[
                            "originator_version"
                        ],
                        topic=row["topic"],
                        state=bytes(row["state"]),
                    )
                )
        return stored_events
```

Now let's run the test with the PostgreSQL aggregate recorder.

```
# Construct the PostgresSQL recorder.
recorder = PostgresAggregateRecorder()

# Create tables.
```

```
recorder.create_table()

# Run the test.
test(recorder)
```

Chapter 5 Event Store

We can map and record stored events. But applications just need to store and retrieve domain events.

therefore…

Hide the complexity of both mapping events and managing records behind an interface for storing and retrieving events.

An event store implements an interface for storing and retrieving domain model events, and uses an event mapper and a recorder to implement this interface.

When domain events are stored, the event store firstly maps the domain events to stored events using an event mapper, and then the stored events are recorded in the database using the recorder.

When retrieving domain events for a particular aggregate, firstly the recorder is used to read stored events from the database, and then the event mapper is used to map the stored events to domain events.

When retrieving domain events, it is sometimes desirable to select only some of the domain events of an aggregate. For example, it can be useful to be able to select only the domain events from a particular position. It is also sometimes useful to be able to get the last domain event. These two abilities are needed for snapshotting which is discussed in Chapter 7.

In summary, an event store:

- Uses both an event mapper and a recorder
- Stores domain events in their aggregate sequence
- Retrieves the domain events of aggregate

Example

The example below involves the event mapper from Chapter 3 and a recorder from Chapter 4. The bank account aggregate from Chapter 2 is used to generate a sequence of domain events. The pending events are then stored. Having been stored, the domain events are retrieved and used to reconstruct a copy of the bank account aggregate.

Firstly, let's define the event store as an object class. It is initialised with an `mapper` and a `recorder`. It has methods `store()` and `get()` which store and retrieve domain event objects.

```
class EventStore(Generic[TDomainEvent]):
    """
    Stores and retrieves domain events.
    """

    def __init__(
        self,
        mapper: Mapper[TDomainEvent],
        recorder: AggregateRecorder,
    ):
        self.mapper = mapper
        self.recorder = recorder

    def put(self, events, **kwargs):
        """
        Stores domain events in aggregate sequence.
        """
        self.recorder.insert_events(
            list(
                map(
                    self.mapper.from_domain_event,
                    events,
                )
            ),
            **kwargs,
```

```
        )

    def get(
        self,
        originator_id: UUID,
        gt: Optional[int] = None,
        lte: Optional[int] = None,
        desc: bool = False,
        limit: Optional[int] = None,
    ) -> Iterator[TDomainEvent]:
        """
        Retrieves domain events from aggregate sequence.
        """
        return map(
            self.mapper.to_domain_event,
            self.recorder.select_events(
                originator_id=originator_id,
                gt=gt,
                lte=lte,
                desc=desc,
                limit=limit,
            ),
        )
```

The ability to retrieve only the domain events from a particular position, and to retrieve only the last domain event in a particular sequence, will be used when discussing snapshotting in Chapter 7.

Now, let's construct an `EventStore` object with the mapper and transcoder from Chapter 3 and the recorder from Chapter 4.

```
transcoder = Transcoder()
transcoder.register(UUIDAsHex())
transcoder.register(DecimalAsStr())
transcoder.register(DatetimeAsISO())

event_store = EventStore(
```

```
    mapper=Mapper(transcoder),
    recorder=POPOAggregateRecorder()
)
```

Let's also use the bank account aggregate from Chapter 2 to generate some domain events.

```
# Open an account.
account = BankAccount.open(
    full_name="Alice",
    email_address="alice@exmaple.com"
)

# Credit the account.
account.append_transaction(Decimal("10.00"))
account.append_transaction(Decimal("25.00"))
account.append_transaction(Decimal("30.00"))
```

We can collect the pending domain events from the aggregate and store them in the event store.

```
# Collect pending events.
pending = account._collect_()

# Store pending events.
event_store.put(pending)
```

Having stored the domain events, we can retrieve them from the event store and project them using their `mutate()` methods to reconstruct the current state of the `account` aggregate.

```
# Get domain events.
domain_events = event_store.get(account.id)

# Reconstruct the bank account.
copy = None
```

```
for domain_event in domain_events:
    copy = domain_event.mutate(copy)

# Check copy has correct attribute values.
assert copy.id == account.id
assert copy.balance == Decimal("65.00")
```

The few lines above, that get and project the domain events of an aggregate, will be encapsulated by the `Repository` object in Chapter 8.

Intermission 1 Notifications

To propagate the state of an application, we need to follow a single sequence, but the application's domain model events are each only in their own aggregate sequence.

therefore…

Give the application "personal order" by writing a sequence of event notifications, along with the new domain events, all in the same atomic database transaction.

In Chapter 4, stored event recorders were defined which are capable of recording stored events in aggregate sequences. That is useful for persisting a collection of aggregates. But it isn't adequate for propagating a collection of aggregates, or the state of a domain model, as a whole. The set of domain model events does not have 'total order'. Without further modification, in order to check if anything new has happened, it would be necessary frequently to check the many aggregate sequences. As the number of aggregate sequences grows, the cost of checking for new domain events will increase. There would also need to be a separate mechanism to check for the addition of new sequences.

Therefore, to propagate the state of a domain model in a reliable way, all the domain events must be ordered serially. By positioning each domain event in both an aggregate sequence and an event notification sequence, the state of a domain model as a whole can be propagated reliably.

To avoid duplication, each domain event must be included in only one event notification. Each event notification must also have a unique identity in the sequence. So that the ordering of the event notifications is determined, it makes sense to identify event notifications using a countable identity, and to identify subsequent event notifications with a higher value than previous

ones. The sequence of event notification identities can be adequately and reliably constructed and followed using counting by ones.

To make sure that stored domain events are always positioned in both an aggregate sequence and the event notification sequence, the aggregate domain event and the domain event notification must be written together in an atomic transaction. If the event notification was written separately from the writing of the stored events, a sudden termination of the process might lead to a situation where only one or the other was recorded. Separate recording of the event notifications is an example of "dual writing", which was discussed in the Prologue. An example of "dual writing" is recording domain events in an event recorder and then posting the domain events to a message queue such as an AMQP server.

Specialist event stores such as EventStoreDB and Axon Server are capable of atomically storing domain events in two sequences just like this. A database such as Cassandra, with its light-weight transactions that only allow atomic writes within a single sequence, are not able to do this.

In summary, an application recorder:

- Writes stored events to both aggregate and event notification sequence
- Position notifications in event notification sequence using integers
- Commits notifications atomically with domain events
- Has uniqueness constraints on position in both aggregate and event notification sequence

Example

Using an application recorder, `StoredEvent` objects can be written to many aggregate sequences and then accessed as a single sequence of notifications.

In the example below, application recorders are defined which extend the recorders from Chapter 4 so that domain events objects can be stored in an aggregate sequence along with event notifications that position those domain model events in an event notification sequence.

The dataclass `Notification` is used to define domain event notification objects, with each having an integer `id`, a `topic` and `state`. It extends the `StoredEvent` class from Chapter 3, and adds a field `id` which represents the event notification ID.

```
class Notification(StoredEvent):
    id: int
```

Let's define an abstract base class for application recorders. Below, `ApplicationRecorder` extends `EventRecorder` from Chapter 4.

```
class ApplicationRecorder(AggregateRecorder):
    @abstractmethod
    def select_notifications(
        self, start: int, limit: int
    ) -> List[Notification]:
        """
        Returns a list of event notifications
        from 'start', limited by 'limit'.
        """

    @abstractmethod
    def max_notification_id(self) -> int:
        """
        Returns the maximum notification ID.
        """
```

The method `select_notifications()` will return a limited selection of `Notification` objects, with notification IDs from `start`. The method `max_notification_id()` will return the highest notification ID in the event notification sequence.

Before implementing the abstract base class, let's firstly define a test for application recorders. As with the test for aggregate recorders, we will insert events. But as well as selecting events, we will also select notifications.

```
def test(recorder: ApplicationRecorder):

    # Write two stored events.
    originator_id1 = uuid4()
    originator_id2 = uuid4()

    stored_event1 = StoredEvent(
        originator_id=originator_id1,
        originator_version=0,
        topic='topic1',
        state=b'state1'
    )
    stored_event2 = StoredEvent(
        originator_id=originator_id1,
        originator_version=1,
        topic='topic2',
        state=b'state2'
    )
    stored_event3 = StoredEvent(
        originator_id=originator_id2,
        originator_version=1,
        topic='topic3',
        state=b'state3'
    )

    recorder.insert_events([
        stored_event1, stored_event2
    ])
    recorder.insert_events([
        stored_event3
    ])

    stored_events1 = recorder.select_events(originator_id1)
    stored_events2 = recorder.select_events(originator_id2)

    # Check we got what was written.
    assert len(stored_events1) == 2
```

```
    assert len(stored_events2) == 1

    assert recorder.max_notification_id() == 3

    notifications = recorder.select_notifications(1, 10)
    assert len(notifications) == 3, len(notifications)
    assert notifications[0].id == 1
    assert notifications[0].topic == 'topic1'
    assert notifications[0].state == b'state1'
    assert notifications[1].id == 2
    assert notifications[1].topic == 'topic2'
    assert notifications[1].state == b'state2'
    assert notifications[2].id == 3
    assert notifications[2].topic == 'topic3'
    assert notifications[2].state == b'state3'

    notifications = recorder.select_notifications(3, 10)
    assert len(notifications) == 1, len(notifications)
    assert notifications[0].id == 3
    assert notifications[0].topic == 'topic3'
    assert notifications[0].state == b'state3'
```

POPO

Let's extend the "Plain Old Python Objects" aggregate recorder so that we can use it for developing an application. This time, the index of the list of stored events is used to select the stored events which are then converted to `Notification` objects.

```
class POPOApplicationRecorder(
    ApplicationRecorder, POPOAggregateRecorder
):
    def select_notifications(
        self, start: int, limit: int
    ) -> List[Notification]:
        with self.database_lock:
            results = []
```

```
            i = start - 1
            j = i + limit
            for notification_id, s in enumerate(
                self.stored_events[i:j], start
            ):
                n = Notification(
                    id=notification_id,
                    originator_id=s.originator_id,
                    originator_version=s.originator_version,
                    topic=s.topic,
                    state=s.state,
                )
                results.append(n)
            return results

    def max_notification_id(self) -> int:
        with self.database_lock:
            return len(self.stored_events)
```

We can now run the above test with the "Plain Old Python Objects" application recorder.

```
# Construct the POPO recorder.
recorder = POPOApplicationRecorder()

# Run the test.
test(recorder)
```

SQLite

We can also implement an application recorder using SQLite. Below, the class `SQLiteApplicationRecorder` implements the abstract base class `ApplicationRecorder` by extending the class `SQLiteEventRecorder` from Chapter 4.

Its method `select_notifications()` makes use of the SQLite table's `rowid` to select stored events and returns `Notification` objects. Its method `max_notification_id()` also makes use of the `rowid`. The internal `rowid` of the SQLite table effectively positions all the stored event records in an event notification sequence.

```
class SQLiteApplicationRecorder(
    ApplicationRecorder,
    SQLiteAggregateRecorder,
):
    def select_notifications(
        self, start: int, limit: int
    ) -> List[Notification]:
        """
        Returns a list of event notifications
        from 'start', limited by 'limit'.
        """
        statement = (
            "SELECT "
            "rowid, *"
            f"FROM {self.table_name} "
            "WHERE rowid>=? "
            "ORDER BY rowid "
            "LIMIT ?"
        )
        params = [start, limit]
        c = self.db.get_connection().cursor()
        c.execute(statement, params)
        notifications = []
        for row in c.fetchall():
            notifications.append(
                Notification(
                    id=row["rowid"],
                    originator_id=UUID(
                        row["originator_id"]
                    ),
                    originator_version=row[
```

```
                        "originator_version"
                    ],
                    topic=row["topic"],
                    state=row["state"],
                )
            )
        return notifications

    def max_notification_id(self) -> int:
        """
        Returns the maximum notification ID.
        """
        c = self.db.get_connection().cursor()
        statement = (
            f"SELECT MAX(rowid) FROM {self.table_name}"
        )
        c.execute(statement)
        return c.fetchone()[0] or 0
```

Now let's run the test with the SQLite application recorder.

```
# Construct the SQLite recorder.
recorder = SQLiteApplicationRecorder(
    SQLiteDatabase(':memory:')
)

# Create tables.
recorder.create_table()

# Run the test.
test(recorder)
```

PostgreSQL

Similarly, we can extend the PostgreSQL aggregate recorder.

```
class PostgresApplicationRecorder(
    ApplicationRecorder,
    PostgresAggregateRecorder,
):
    def _create_table(self, c: cursor):
        super()._create_table(c)
        statement = (
            f"ALTER TABLE {self.events_table} "
            "ADD COLUMN IF NOT EXISTS "
            f"rowid SERIAL"
        )
        try:
            c.execute(statement)
        except psycopg2.OperationalError as e:
            raise self.OperationalError(e)

        statement = (
            "CREATE UNIQUE INDEX IF NOT EXISTS rowid_idx "
            f"ON {self.events_table} (rowid ASC);"
        )
        try:
            c.execute(statement)
        except psycopg2.OperationalError as e:
            raise self.OperationalError(e)

    def select_notifications(
        self, start: int, limit: int
    ) -> List[Notification]:
        """
        Returns a list of event notifications
        from 'start', limited by 'limit'.
        """
        statement = (
            "SELECT * "
            f"FROM {self.events_table} "
            "WHERE rowid>=%s "
            "ORDER BY rowid "
```

```
            "LIMIT %s"
        )
        params = [start, limit]
        with self.db.transaction() as c:
            c.execute(statement, params)
            notifications = []
            for row in c.fetchall():
                notifications.append(
                    Notification(
                        id=row["rowid"],
                        originator_id=row["originator_id"],
                        originator_version=row[
                            "originator_version"
                        ],
                        topic=row["topic"],
                        state=bytes(row["state"]),
                    )
                )
        return notifications

    def max_notification_id(self) -> int:
        """
        Returns the maximum notification ID.
        """
        c = self.db.get_connection().cursor()
        statement = (
            f"SELECT MAX(rowid) FROM {self.events_table}"
        )
        c.execute(statement)
        return c.fetchone()[0] or 0
```

Now let's run the test with the PostgresSQL application recorder.

```
# Construct the Postgres recorder.
recorder = PostgresApplicationRecorder()

# Create tables.
```

```
recorder.create_table()

# Run the test.
test(recorder)
```

PART 2 APPLICATION

Chapter 6 Notification Log

The event notifications have been written into a sequence for the application, but we don't have a way to read them.

therefore…

Present a notification log with sections of event notifications.

All the domain events of an event-sourced domain model can usefully be positioned in a sequence. This is useful because it allows the state of an application to be propagated in a reliable way.

Once the domain events are positioned in a sequence, we need a way of reading them. The notification log is a pattern for presenting domain events in linked sections of event notifications.

By defining an interface for accessing linked sections of event notifications, the interface can be implemented locally or remotely.

Using sections of event notifications effectively batches the individual event notifications in an event notification sequence so that the overhead of reading event notifications from a database, and transporting them across a network, is reduced. The sections, once full, will not change and can therefore be cached.

In summary, a notification log:

- Presents all event notifications in a sequence
- Batches event notifications in linked sections
- Implements a standard interface can be used locally or remotely

Example

Let's firstly define a standard `Section` type that will be used to present event notification in batches.

```
def format_section_id(first, limit):
    return "{},{}".format(first, limit)


class Section(ImmutableObject):
    section_id: str
    items: List[Notification]
    next_id: Optional[str]
```

We need to define an abstract interface for notification logs, so that notification log readers can deal not only with local notification logs but also remote notification logs.

```
class AbstractNotificationLog(ABC):
    @abstractmethod
    def __getitem__(self, section_id: str) -> Section:
        """
        Returns section of notification log.
        """
```

The class `LocalNotificationLog` defined below inherits from the abstract base class defined above. It is constructed with an application recorder (see Intermission 1) and implements the abstract interface by selecting event notifications from its recorder, according to the section ID.

```
class LocalNotificationLog(AbstractNotificationLog):
    DEFAULT_SECTION_SIZE = 10

    def __init__(
        self,
        recorder: ApplicationRecorder,
        section_size: int = DEFAULT_SECTION_SIZE,
    ):
        self.recorder = recorder
```

```
        self.section_size = section_size

    def __getitem__(self, section_id: str) -> Section:
        # Interpret the section ID.
        parts = section_id.split(",")
        part1 = int(parts[0])
        part2 = int(parts[1])
        start = max(1, part1)
        limit = min(
            max(0, part2 - start + 1), self.section_size
        )

        # Select notifications.
        notifications = self.recorder.select_notifications(
            start, limit
        )

        # Get next section ID.
        if len(notifications):
            last_id = notifications[-1].id
            return_id = format_section_id(
                notifications[0].id, last_id
            )
            if len(notifications) == limit:
                next_start = last_id + 1
                next_id = format_section_id(
                    next_start, next_start + limit - 1
                )
            else:
                next_id = None
        else:
            return_id = None
            next_id = None

        # Return a section of the notification log.
        return Section(
            section_id=return_id,
```

```
            items=notifications,
            next_id=next_id,
        )
```

Then we can construct a notification log using an application recorder, store some events, and select them as a section of event notifications.

The section IDs are formatted as two integers separated by a comma. These integers are interpreted as a request for a range of event notifications. The section that is returned has a section ID that indicates what the section actually includes. The next section ID is provided when the section is full.

The number of event notifications in a section is constrained by the `section_size` constructor argument of the notification log. That means you can ask for less then the section size, but if you ask for more, the sections will contain at most the `section_size` number of items.

```
recorder = SQLiteApplicationRecorder(
    SQLiteDatabase(":memory:")
)
recorder.create_table()

# Construct notification log.
notification_log = LocalNotificationLog(
    recorder, section_size=5
)

# Get the first section of log.
section = notification_log["1,5"]
assert len(section.items) == 0
assert section.section_id == None
assert section.next_id == None

# Write 5 events.
originator_id = uuid4()
for i in range(5):
    stored_event = StoredEvent(
```

```
            originator_id=originator_id,
            originator_version=i,
            topic="topic",
            state=b"state"
        )
        recorder.insert_events([stored_event])

    # Get the first section of log.
    section = notification_log["1,5"]
    assert len(section.items) == 5
    assert section.section_id == "1,5"
    assert section.next_id == "6,10"

    # Write 4 events.
    originator_id = uuid4()
    for i in range(4):
        stored_event = StoredEvent(
            originator_id=originator_id,
            originator_version=i,
            topic="topic",
            state=b"state"
        )
        recorder.insert_events([stored_event])

    # Get the next section of log.
    section = notification_log["6,10"]
    assert len(section.items) == 4
    assert section.section_id == "6,9"
    assert section.next_id == None
```

Chapter 7 Snapshot

The more domain events an aggregate has generated, the longer it takes to reconstruct the aggregate from these domain events.

therefore…

Take a snapshot of the state of the aggregate, so that the access time is reduced.

A snapshot records the current state of an aggregate. Snapshots can be used to speed up the access time of event-sourced aggregates.

Rather than getting all the aggregate events and using them to reconstruct the current state of an aggregate, we can instead get the last snapshot and the subsequent aggregate events. By checking for an aggregate snapshot and then getting only the aggregate events that were generated since the snapshotted version, the time it takes to access the aggregate can be reduced. If snapshots are taken at regular intervals, the access time will be more or less constant rather than being proportional to the totla number of aggregate events.

Aggregate snapshots are very similar to aggregate events. They need to refer to the ID and version of an aggregate. Consequently, the same mechanism for storing, retrieving and projecting aggregate events can be used to store, retrieve and project aggregate snapshots. Hence, to make a snapshot store we can use an event store, a mapper, and an event recorder. However, snapshots need to be recorded in a different table from the aggregate events, so that the originator ID and version of the stored aggregate snapshot does not conflict with stored aggregate events.

Having snapshots in a separate table also makes it easier to discard old snapshots if the aggregate's projection of its domain events is changed.

Some specialist event sourcing databases, in particular AxonDB, return snapshots when querying for the domain events of an aggregate. For this reason it makes sense to define snapshots to work in the same way as aggregate events when it comes to projecting them into the current state of an aggregate.

In summary, snapshotting:

- Reduces time taken to reconstruct current aggregate state
- Records the current state of an aggregate as an event
- Records aggregate snapshots separately from aggregate events
- Reuses existing mechanismss for inserting, selecting and projecting

Example

Let's define an object class for snapshots. The class `Snapshot` defined below inherits from `DomainEvent` (introduced in Chapter 1). It has two attributes, `topic` and `state`, which are used to capture the topic of an aggregate and its state.

```
class Snapshot(DomainEvent):
    topic: str
    state: dict

    @classmethod
    def take(cls, aggregate: Aggregate) -> DomainEvent:
        return cls(
            originator_id=aggregate.id,
            originator_version=aggregate.version,
            timestamp=datetime.now(),
            topic=get_topic(type(aggregate)),
            state=aggregate.__dict__,
        )

    def mutate(self, _=None) -> Aggregate:
        cls = resolve_topic(self.topic)
```

```
        aggregate = object.__new__(cls)
        assert isinstance(aggregate, Aggregate)
        aggregate.__dict__.update(self.state)
        return aggregate
```

It has a class method `take()` which can be used to construct a new snapshot object from an aggregate object. The method `mutate()` will reconstruct the aggregate object from the snapshot object. It is defined with the same signature as `Aggregate.Event.mutate()` so that a snapshot can be used as the first domain event in a sequence that determines the current state of an aggregate.

In the example below, an aggregate is created. A snapshot of the aggregate is created, and stored using the same mechanism for storing events that is used for storing domain events.

```
# Open an account.
account = BankAccount.open(
    full_name="Alice",
    email_address="alice@example.com"
)

# Credit the account.
account.append_transaction(Decimal("10.00"))
account.append_transaction(Decimal("25.00"))
account.append_transaction(Decimal("30.00"))

transcoder = Transcoder()
transcoder.register(UUIDAsHex())
transcoder.register(DecimalAsStr())
transcoder.register(DatetimeAsISO())

snapshot_store = EventStore(
    mapper=Mapper(transcoder),
    recorder=SQLiteAggregateRecorder(
        SQLiteDatabase(':memory:'),
        table_name="snapshots"
```

```
    )
)
snapshot_store.recorder.create_table()

# Clear pending events.
account.pending_events.clear()

# Take a snapshot.
snapshot = Snapshot.take(account)

# Store snapshot.
snapshot_store.put([snapshot])

# Get snapshot.
snapshot = next(
    snapshot_store.get(
        account.id, desc=True, limit=1
    )
)

# Reconstruct the aggregate.
copy = snapshot.mutate()

# Check copy has correct attribute values.
assert copy.id == account.id
assert copy.balance == Decimal("65.00")
```

Chapter 8 Repository

The event store returns domain events for an aggregate ID. But application services deal with aggregates by ID.

therefore…

Use a repository to obtain the state of an aggregate by its ID.

A repository provides an interface for accessing aggregates by their ID.

The ID is used to retrieve the existing domain events for an aggregate. The domain events are then used to reconstruct the state of the aggregate. Finally, the aggregate object is returned.

Sometimes we want to access the aggregate is a previous, historical state. In this case, we would like to specify both the aggregate ID and a version number, and obtain the aggregate in the state it was in at that time.

A repository should also be able to make sure of any snapshots that are available. If there are several snapshots, we only need the last one. However, if we are obtaining an older version of the aggregate, we can use the last snapshot before the requested version.

If a snapshot if found, then we only need to retrieve the domain events since the snapshot was created. The aggregate can be firstly reconstructed from the snapshot, and then its state can be advanced by applying the subsequence domain events.

If the aggregate is not found, then it is useful to raise an exception.

In summary, a repository:

- Implements dictionary-like interface
- Gets domain events from event store

- Projects domain events into aggregate object
- Returns aggregate for given aggregate ID

Example

The class `Repository` defined below is constructed with an event store for aggregate domain events, and an optionally with an event store for aggregate snapshots. It defines a method `get()` which will try to find and return an aggregate for a given aggregate ID. If no aggregate is found, the exception class `AggregateNotFoundError` is raised.

```
class Repository:
    def __init__(
        self,
        event_store: EventStore[Aggregate.Event],
        snapshot_store: Optional[
            EventStore[Snapshot]
        ] = None,
    ):
        self.event_store = event_store
        self.snapshot_store = snapshot_store

    def get(
        self, aggregate_id: UUID, version: int = None
    ) -> Aggregate:

        gt = None
        domain_events: List[
            Union[Snapshot, Aggregate.Event]
        ] = []

        # Try to get a snapshot.
        if self.snapshot_store is not None:
            snapshots = self.snapshot_store.get(
                originator_id=aggregate_id,
                desc=True,
```

```
                limit=1,
                lte=version,
            )
            try:
                snapshot = next(snapshots)
                gt = snapshot.originator_version
                domain_events.append(snapshot)
            except StopIteration:
                pass

        # Get the domain events.
        domain_events += self.event_store.get(
            originator_id=aggregate_id,
            gt=gt,
            lte=version,
        )

        # Project the domain events.
        aggregate = None
        for domain_event in domain_events:
            aggregate = domain_event.mutate(aggregate)

        # Raise exception if not found.
        if aggregate is None:
            raise AggregateNotFoundError

        # Return the aggregate.
        assert isinstance(aggregate, Aggregate)
        return aggregate


class AggregateNotFoundError(Exception):
    pass
```

When called, the repository `get()` method uses the given `aggregate_id` firstly to obtain the latest snapshot of the requested aggregate, and then to select existing aggegate domain events. If a snapshot is found then the

version number will be used to restrict the selection of the aggregate domain events to those created since that version. In so far as they exist, the snapshot and the aggregate domain events will be used to obtain the current state of the aggregate. If the aggregate is requested at a particular version, by using the `version` argument of the `get()` method, the selections are further restricted to things that exist at or below that version.

Recalling the `EventStore` class from Chapter 5, we can construct the repository with an event store for aggregate events and an event store for aggregate snapshots. The event store for aggregate events will normally be constructed with an application recorder, so that is what we will do in this example. The event store aggregate snapshots only needs an event recorder. The mapper object can be used by both event store objects.

```
transcoder = Transcoder()
transcoder.register(UUIDAsHex())
transcoder.register(DecimalAsStr())
transcoder.register(DatetimeAsISO())
mapper = Mapper(transcoder)

event_store = EventStore(
    mapper=mapper,
    recorder=POPOApplicationRecorder()
)

snapshot_store = EventStore(
    mapper=mapper,
    recorder=POPOAggregateRecorder()
)

repository = Repository(event_store, snapshot_store)
```

Now let's create a `BankAccount` aggregate from Chapter 2, and append some transactions.

```
# Open an account.
account = BankAccount.open(
```

```
    full_name="Alice",
    email_address="alice@example.com"
)

# Remember the account ID.
account_id = account.id

# Credit the account.
account.append_transaction(Decimal("10.00"))
account.append_transaction(Decimal("25.00"))
account.append_transaction(Decimal("30.00"))
```

Then, let's collect and store the pending domain events.

```
# Collect pending events.
pending = account._collect_()

# Store pending events.
event_store.put(pending)
```

Now we can access the aggregate using the repository. The repository gets the domain events from the event store, and projects them to obtain the current state of the aggregate.

```
# Get the bank account.
account = repository.get(account_id)

# Check account has correct attribute values.
assert account.id == account_id
assert account.balance == Decimal("65.00")
```

Recalling the snapshot from Chapter 7, we can take a snapshot of the account aggregate.

```
# Store a snapshot.
snapshot = Snapshot.take(account)
```

```
snapshot_store.put([snapshot])
```

Now when we access the bank account aggregate, it will use the snapshot. It will select the snapshot from the snapshot store and reconstruct the aggregate from the snapshot.

```
# Get aggregate (uses stored snapshot).
account = repository.get(account_id)

# Check copy has correct attribute values.
assert account.balance == Decimal("65.00")
```

We can continue to use the bank account aggregate, and store subsequent domain events.

```
# Credit the account.
account.append_transaction(Decimal("10.00"))
event_store.put(account._collect_())
```

Now, when we access the bank account aggregate, the aggregate will be reconstructed from the stored snapshot and the domain events will be used to advance the state of the aggregate to its current state.

```
# Reconstruct aggregate from snapshot and domain events.
account = repository.get(account_id)
assert account.balance == Decimal("75.00")
```

We can also get old versions of the bank account object by using the `version` argument of the `get()` method. This uses snapshots if they are available prior to that version.

```
# Reconstruct version 1.
account = repository.get(account_id, version=1)
assert account.balance == Decimal("0.00")

# Reconstruct version 2.
```

```
account = repository.get(account_id, version=2)
assert account.balance == Decimal("10.00")

# Reconstruct version 3.
account = repository.get(account_id, version=3)
assert account.balance == Decimal("35.00")
```

The repository will raise an exception if the aggregate is not found.

```
try:
    repository.get(uuid4())
except AggregateNotFoundError:
    pass
else:
    raise AssertionError("AggregateNotFoundError not raised")
```

The recorder will raise an exception if an attempt is made to stored changes from an old version.

```
account = repository.get(account_id, version=3)
account.append_transaction(Decimal("10.00"))
try:
    event_store.put(account._collect_())
except Recorder.IntegrityError:
    pass
else:
    raise AssertionError("Integrity error not raise")
```

Chapter 9 Application

Interfaces use application services. But so far we only have repositories, notification logs, event stores, mappers, recorders, aggregates, snapshots, and domain events.

therefore…

Create a unified application layer object that hides the complexity of the domain and infrastructure layers. Add application services that support interfaces as application object methods.

In *Domain-Driven Design*, the purpose of the application layer is to bring together the domain layer and the infrastructure layer. An application layer offers application services that can be accessed by the interface layer.

In this chapter, the idea of an application object is developed. Following the CQS pattern, the application will have 'application services' implemented with 'command methods' and 'query methods' that manipulate and present a particular domain model supported by particular infrastructure. The application command methods will change the state of the application but will not return anything. The query methods will return particular aspects of the state of the application but will not change anything.

Application services allow interfaces to use a domain model without dealing directly with the aggregates. The domain model can be presented in a variety of interfaces without replicating the functionality of the application services in multiple interfaces.

Although the distinction between command and query methods is useful, some methods may usefully make changes and return values. For example, 'factory methods' may create and return a new aggregate ID. But such methods could instead add the new ID to a collection, so that in case the aggregate is successfully created there is no risk of losing the new

aggregate ID because the client happens not to receive the new ID. Alternatively, the aggregate ID could be a deterministic function of a unique name, so that the new aggregate can be obtained by name. In that case, there is no need to return anything, and the factory method is effectively a command method.

As discussed in Chapter 2, if the name of an aggregate will change, index aggregates can be used to keep track of which aggregate ID is referred to by any given name at any given time. In this case it makes sense to record both the named aggregate and the index aggregate in the same atomic database transaction, so that either both or neither are changed. The use of atomic database transactions to define the consistency boundary of an aggregate can then usefully be extended to include events from more than one aggregate.

Application objects will use a notification log (see Chapter 6) to present their state in a way that can be reliably propagated and processed, perhaps into a "read model" as seen in CQRS. A repository (see Chapter 8) will be used by the application services to access the aggregates of the application.

Whilst the application layer brings together the domain and infrastructure layers, the application services will naturally be defined independently of particular infrastructure. That means the particular database used by an application can be configured at "run time" when the application is actually used.

By using application objects rather than a less unified approach, the configuration of different applications can be encapsulated by the application object. A system of multiple applications can then be developed together using a single thread before perhaps being deployed in separate processes.

In *Domain-Driven Design* a bounded context is the extent of a ubiquitous language, and there may be many bounded contexts. One bounded context may involve many application objects, so that more than one application object is expressed using the same ubiquitous language. For example, in the case of CQRS where one application functions as the "write model" and another as the "read model" the ubiquitous language could be shared by

both applications. The idea can be extended to a chain of process applications.

The construction of application object components, such as the repository and the notification log, and the objects they are composed of, can be defined in a base class for all applications. The construction of these objects can also be delegated to an infrastructure factory. The infrastructure factory can be configured using environment variables.

In summary, an application:

- Has a repository to provide access to domain model aggregate
- Has a notification log to provide access to event notifications
- Has application services that handle client commands and queries
- Can be defined independently of particular infrastructure
- Can be configured to use particular infrastructure at run-time

Example

The class `Application` class below brings together all the components from the previous eight chapters. It will be subclassed by concrete applications that define command and query methods that manipulate the aggregates of a domain model (see Chapter 2). Existing aggregates are accessible in the application's `repository` (see Chapter 8). The repository uses a snapshot store that holds snapshots (see Chapter 7) of aggregates. Aggregate domain events (see Chapter 1) are stored in the application's events store (see Chapter 5), and presented as event notifications in the application's `log` (as discussed in Chapter 6).

The `Application` class has various methods to construct the components of the application object. This makes it possible to extend or replace application's components with other custom components. For example, the method `register_transcodings()` can be extended to extend the mapper's transcoder's transcodings, as discussed in Chapter 3. An alternative transcoder can be used by overriding the `construct_transcoder()` method. The `construct_factory()` method constructs the infrastructure factory that

much of the construction is delegated to. The infrastructure factory will be described later in this example.

The `save()` method collects pending domain events from a list of aggregates, and stores aggregate events using the `put()` method of the application's event store.

The `notify()` method exists to allow subclasses to push new events beyond the application objects. It is used in Part 3 to prompt "follower" applications to pull event notifications from "leader" applications. It could also be used to put messages on a message queue.

```
class Application(ABC):
    def __init__(self):
        self.factory = self.construct_factory()
        self.mapper = self.construct_mapper()
        self.recorder = self.construct_recorder()
        self.events = self.construct_event_store()
        self.snapshots = self.construct_snapshot_store()
        self.repository = self.construct_repository()
        self.log = self.construct_notification_log()

    def construct_factory(self) -> InfrastructureFactory:
        return InfrastructureFactory.construct(
            self.__class__.__name__
        )

    def construct_mapper(
        self, application_name=""
    ) -> Mapper:
        return self.factory.mapper(
            transcoder=self.construct_transcoder(),
            application_name=application_name,
        )

    def construct_transcoder(self) -> AbstractTranscoder:
        transcoder = Transcoder()
```

```
        self.register_transcodings(transcoder)
        return transcoder

    def register_transcodings(
        self, transcoder: Transcoder
    ):
        transcoder.register(UUIDAsHex())
        transcoder.register(DecimalAsStr())
        transcoder.register(DatetimeAsISO())

    def construct_recorder(self) -> ApplicationRecorder:
        return self.factory.application_recorder()

    def construct_event_store(
        self,
    ) -> EventStore[Aggregate.Event]:
        return self.factory.event_store(
            mapper=self.mapper,
            recorder=self.recorder,
        )

    def construct_snapshot_store(
        self,
    ) -> Optional[EventStore[Snapshot]]:
        if not self.factory.is_snapshotting_enabled():
            return None
        recorder = self.factory.aggregate_recorder()
        return self.factory.event_store(
            mapper=self.mapper,
            recorder=recorder,
        )

    def construct_repository(self):
        return Repository(
            event_store=self.events,
            snapshot_store=self.snapshots,
        )
```

```
    def construct_notification_log(self):
        return LocalNotificationLog(
            self.recorder, section_size=10
        )

    def save(self, *aggregates: Aggregate) -> None:
        events = []
        for aggregate in aggregates:
            events += aggregate._collect_()
        self.events.put(events)
        self.notify(events)

    def notify(self, new_events: List[Aggregate.Event]):
        pass

    def take_snapshot(
        self, aggregate_id: UUID, version: int
    ):
        aggregate = self.repository.get(
            aggregate_id, version
        )
        snapshot = Snapshot.take(aggregate)
        self.snapshots.put([snapshot])
```

The application class `BankAccounts` defined inherits from the `Application`. This application object class has methods to open and credit accounts. The command method `open_account()` can be used to open a new account. The command method `credit_account()` can be used to append a transaction to an existing account. The accounts are modelled using the `BankAccount` aggregate introduced in Chapter 2.

```
class BankAccounts(Application):
    def open_account(self, full_name, email_address):
        account = BankAccount.open(
            full_name=full_name,
            email_address=email_address,
```

```
        )
        self.save(account)
        return account.id

    def credit_account(
        self, account_id: UUID, amount: Decimal
    ) -> None:
        account = self.get_account(account_id)
        account.append_transaction(amount)
        self.save(account)

    def get_balance(self, account_id: UUID) -> Decimal:
        account = self.get_account(account_id)
        return account.balance

    def get_account(self, account_id: UUID) -> BankAccount:
        try:
            aggregate = self.repository.get(account_id)
        except self.repository.AggregateNotFoundError:
            raise self.AccountNotFoundError(account_id)
        else:
            if not isinstance(aggregate, BankAccount):
                raise self.AccountNotFoundError(account_id)
            return aggregate

    class AccountNotFoundError(Exception):
        pass
```

Application command methods cause new domain event objects to be created when they call the command methods of the aggregates they interact with. As we saw in Chapter 2, these new domain event objects will be "pending" in the account aggregates until they are collected and stored. Unless the domain events are stored, they will be lost when the aggregate objects go out of scope. For this reason, the command methods must use the `save()` method of the `Application` class, which will collect the pending domain events and store them in its event store.

The query method `get_balance()` can be used to get the current balance of an existing account.

The method `get_account()` helps the command and query methods by obtaining an account aggregate for a given ID. In the cases where the aggregate is not found or the ID of another type of aggregate is used by mistake, the error `AccountNotFoundError` will be raised.

Application objects can be constructed by calling the application class. So let's construct an instance of the `BankAccounts` application.

```
app = BankAccounts()
```

According to the definition of the `create_factory()` method of the `Application` class, and the `construct()` method of the `InfrastructureFactory` class, by default the `POPOInfrastructureFactory` will be used to construct a "Plain Old Python Objects" application recorder for the application to store its aggregate's domain events. The `InfrastructureFactory` classes will be explained towards the end of this chapter.

All we need to know at this point is that the application has an `ApplicationRecorder` object as its `aggregate_event_store`. The `ApplicationRecorder` was defined in Intermission 1. The application also has a `LocalNotificationLog` as its `log`, which uses the application recorder. As explained in Chapter 6, the notification log queries its application recorder for event notifications. In this way, the application can present the event notifications that represent the domain events stored by calling the application's `save()` method.

Since no accounts have so far been opened in this application, nor have any other aggregates have been created in this application, the notification log will initially be empty.

```
section = app.log["1,10"]

assert len(section.items) == 0
```

So let's create a `BankAccount` aggregate by calling the `BankAccounts` application's `open_account()` method. A new `BankAccount` aggregate will then exist in the application's repository. The method returns the `UUID` of the aggregate. The `BankAccount` aggregate was defined in Chapter 2.

```
account_id = app.open_account(
    full_name="Alice",
    email_address="alice@example.com"
)
```

The aggregate ID can be used to get the new account balance. We can get the account balance by calling the application's `get_balance()` method with the aggregate ID.

```
balance = app.get_balance(account_id)
```

The opening balance is zero.

```
assert balance == Decimal("0.00")
```

In general, application methods should be developed to allow interface clients to get and do what they need without the clients dealing directly with the domain model aggregates. This keeps application logic within the application and avoids repeating application logic across different interface clients. Different interfaces can then focus on implementing whatever protocol they exist to support.

Because we have created an aggregate, there will now be one event notification in the application's notification log.

```
assert len(app.log["1,10"].items) == 1
```

Let's use the application's command method `credit_account()` to credit the account we have opened.

```
app.credit_account(account_id, Decimal("10.00"))

app.credit_account(account_id, Decimal("25.00"))

app.credit_account(account_id, Decimal("30.00"))
```

The balance of the account is the sum of the amounts credited to the account. Since three amounts were credits, 10.00, 25.00, and 30.00, the balance will now be 65.00.

```
assert app.get_balance(account_id) == Decimal("65.00")
```

Since we have now created four domain events, one for opening the account and three for crediting the account, there will now be four domain events in the notification log.

```
section = app.log["1,10"]

assert len(section.items) == 4
```

The `InfrastructureFactory` deserves some explanation. It is the base class for concrete infrastructure classes.

Firstly, it should be noted that the application can be configured using environment variables. For example, the environment variable `INFRASTRUCTURE_FACTORY_TOPIC` is used to identify a particular infrastructure factory class to be used by an application object.

The class method `construct()` reads the infrastructure factory topic from the environment, resolves the topic to a class, then constructs and returns an infrastructure factory object. By default, the `POPOInfrastructureFactory` is used which means that an application will use the "Plain Old Python Objects" recorder classes.

The infrastructure factory object is constructed with the name of an application. The name of the application is used by the `getenv()` method to

read application-specific environment variables, if they are defined. Otherwise non-application-specific environment variable names are used.

```
class InfrastructureFactory(ABC):
    TOPIC = "INFRASTRUCTURE_FACTORY_TOPIC"
    CIPHER_TOPIC = "CIPHER_TOPIC"
    CIPHER_KEY = "CIPHER_KEY"
    MAPPER_TOPIC = "MAPPER_TOPIC"
    COMPRESSOR_TOPIC = "COMPRESSOR_TOPIC"
    IS_SNAPSHOTTING_ENABLED = "IS_SNAPSHOTTING_ENABLED"

    @classmethod
    def construct(cls, name) -> "InfrastructureFactory":
        topic = os.getenv(
            cls.TOPIC,
            "dddbb.poporecorders#POPOInfrastructureFactory",
        )
        try:
            factory_cls = resolve_topic(topic)
        except (ModuleNotFoundError, AttributeError):
            raise EnvironmentError(
                "Failed to resolve "
                "infrastructure factory topic: "
                f"'{topic}' from environment "
                f"variable '{cls.TOPIC}'"
            )

        if not issubclass(
            factory_cls, InfrastructureFactory
        ):
            raise AssertionError(
                f"Not an infrastructure factory: {topic}"
            )
        return factory_cls(application_name=name)

    def __init__(self, application_name):
        self.application_name = application_name
```

```
def getenv(
    self, key, default=None, application_name=""
):
    if not application_name:
        application_name = self.application_name
    keys = [
        application_name.upper() + "_" + key,
        key,
    ]
    for key in keys:
        value = os.getenv(key)
        if value:
            return value
    return default

def mapper(
    self,
    transcoder: AbstractTranscoder,
    application_name: str = "",
) -> Mapper:
    cipher_topic = self.getenv(
        self.CIPHER_TOPIC,
        application_name=application_name,
    )
    cipher_key = self.getenv(
        self.CIPHER_KEY,
        application_name=application_name,
    )
    cipher = None
    compressor = None
    if cipher_topic:
        if cipher_key:
            cipher_key = b64decode(
                cipher_key.encode("utf8")
            )
            cipher_cls = resolve_topic(cipher_topic)
```

```
            cipher = cipher_cls(cipher_key=cipher_key)
        else:
            raise EnvironmentError(
                "Cipher key was not found in env, "
                "although cipher topic was found"
            )
    compressor_topic = self.getenv(
        self.COMPRESSOR_TOPIC
    )
    if compressor_topic:
        compressor = resolve_topic(compressor_topic)
    return Mapper(
        transcoder=transcoder,
        cipher=cipher,
        compressor=compressor,
    )

def event_store(self, **kwargs) -> EventStore:
    return EventStore(**kwargs)

@abstractmethod
def aggregate_recorder(self) -> AggregateRecorder:
    pass

@abstractmethod
def application_recorder(self) -> ApplicationRecorder:
    pass

@abstractmethod
def process_recorder(self) -> ProcessRecorder:
    pass

def is_snapshotting_enabled(self) -> bool:
    default = "no"
    return bool(
        strtobool(
            self.getenv(
```

```
                    self.IS_SNAPSHOTTING_ENABLED, default
                )
                or default
            )
        )
```

The `POPOInfrastructureFactory` is the default infrastructure factory. It implements the abstract base class methods `event_recorder()`, `application_recorder()`, and `process_recorder()` by constructing the "Plain Old Python Objects" recorders defined in Chapter 4, Intermission 1, and Intermission 2.

```
class POPOInfrastructureFactory(InfrastructureFactory):
    def aggregate_recorder(self) -> AggregateRecorder:
        return POPOAggregateRecorder()

    def application_recorder(self) -> ApplicationRecorder:
        return POPOApplicationRecorder()

    def process_recorder(self) -> ProcessRecorder:
        return POPOProcessRecorder()
```

Similarly, the `SQLiteInfrastructureFactory` implements the abstract base class methods `event_recorder()`, `application_recorder()`, and `process_recorder()` by constructing the SQLite recorders defined in Chapter 4, Intermission 1, and Intermission 2. It uses environment variable `DB_URI` to identify the SQLite database name, and it checks `CREATE_TABLE` to see if the table should be created. The function `strtobool()` from the `distutils` package is used to convert the environment variable to a boolean value. So the strings 'y', 'yes', 't', 'true', 'on', '1' represent `True` values and the strings 'n', 'no', 'f', 'false', 'off', '0' represent `False` values.

```
class SQLiteInfrastructureFactory(InfrastructureFactory):
    DB_URI = "DB_URI"
    CREATE_TABLE = "CREATE_TABLE"
```

```
def __init__(self, application_name):
    super().__init__(application_name)
    self._database = None
    self.lock = Lock()

@property
def database(self):
    with self.lock:
        if self._database is None:
            db_uri = self.getenv(self.DB_URI)
            if not db_uri:
                raise EnvironmentError(
                    "Database URI not found "
                    "in environment with key "
                    f"'{self.DB_URI}'"
                )
            self._database = SQLiteDatabase(
                db_uri=db_uri
            )
        return self._database

def aggregate_recorder(self) -> AggregateRecorder:
    recorder = SQLiteAggregateRecorder(
        db=self.database
    )
    if self.do_create_table():
        recorder.create_table()
    return recorder

def application_recorder(self) -> ApplicationRecorder:
    recorder = SQLiteApplicationRecorder(
        db=self.database
    )
    if self.do_create_table():
        recorder.create_table()
    return recorder
```

```
    def process_recorder(self) -> ProcessRecorder:
        recorder = SQLiteProcessRecorder(db=self.database)
        if self.do_create_table():
            recorder.create_table()
        return recorder

    def do_create_table(self) -> bool:
        default = "no"
        return bool(
            strtobool(
                self.getenv(self.CREATE_TABLE, default)
                or default
            )
        )
```

The `PostgresInfrastructureFactory` defined below, is an infrastructure factory that uses the PostgreSQL recorders.

```
class PostgresInfrastructureFactory(InfrastructureFactory):
    CREATE_TABLE = "CREATE_TABLE"

    def __init__(self, application_name):
        super().__init__(application_name)
        self.lock = Lock()

    def aggregate_recorder(self) -> AggregateRecorder:
        recorder = PostgresAggregateRecorder(
            self.application_name
        )
        if self.do_create_table():
            recorder.create_table()
        return recorder

    def application_recorder(self) -> ApplicationRecorder:
        recorder = PostgresApplicationRecorder(
            self.application_name
        )
```

```
        if self.do_create_table():
            recorder.create_table()
        return recorder

    def process_recorder(self) -> ProcessRecorder:
        recorder = PostgresProcessRecorder(
            self.application_name
        )
        if self.do_create_table():
            recorder.create_table()
        return recorder

    def do_create_table(self) -> bool:
        default = "no"
        return bool(
            strtobool(
                self.getenv(self.CREATE_TABLE, default)
                or default
            )
        )
```

The `ProcessRecorder` classes mentioned in the infrastructure factories are introduced in Intermission 2.

Chapter 10 Remote Log

A client that is situated remotely, in a different operating system process on the same machine or separated further by a network, will not be able directly to process the state of an application object.

therefore…

Use a remote notification log to pull sections from a remote view of the local notifications log.

In Chapter 9 an application object was introduced that presented its state through its notification log, so that its state can reliably be propagated. However, an application object can only be used directly from within the same operating system process.

The purpose of this chapter is to venture into the interface layer and show how the the notification log can be presented for remote access.

A remote notification log presents notification events using the same interface as the local notification log. It functions effectively as a proxy for a local notification log, allowing a notification log to be used across operating system process boundaries, and across networks.

Whilst the interface of the remote notification log is the same as the notification log introduced in Chapter 6, the implementation of the remote notification log is quite different. A remote notification log makes calls to a notification log view which will normally read sections from a local notification log and serialise them before presenting them and passing them back to the remote notification log. The remote notification log will then deserialise the sections, and then return notification log sections to its readers.

There are many different ways of serialising and transferring notification log sections. For example, a notification log view can use JSON to serialise

the notification log sections and HTTP can be used to transfer the serialised sections from one process to another.

In summary, a remote notification log:

- Requests and deserialises sections from a notification log view
- Depends on a serialisation format such as JSON
- Depends on a protocol for transferring serialised sections

Example

The example below illustrates how a notification log can be used remotely. We will develop an API for interacting with application objects using JSON documents, and a client that uses that API. We will then introduce HTTP as a transfer protocol for mediating between the client and the application.

Application API

Firstly, let's define an API presenting serialised notification log sections.

The abstract base class `AbstractNotificationLogView` suggests that various serialisation formats could be used to represent notification log sections. Instances of this class will be constructed with a local notification log object (see Chapter 6).

```
class AbstractNotificationLogView(ABC):
    """
    Presents serialised notification log sections.
    """

    def __init__(self, log: LocalNotificationLog):
        self.log = log

    @abstractmethod
    def get(self, section_id: str) -> str:
        pass
```

The method `get()` will obtain a `Section` object (see Chapter 6) from the local notification log, and will return a serialised representation of that notification log section.

The class `JSONNotificationLogView` below implements the abstract base class by serialising notification log sections as JSON documents.

The `Notification` items are serialised by calling the helper method `serialise_item()`. The class `Notification` was introduced in Intermission 1.

```
class JSONNotificationLogView(AbstractNotificationLogView):
    """
    Presents notification log sections in JSON format.
    """

    def get(self, section_id: str) -> str:
        section = self.log[section_id]
        return json.dumps(
            {
                "section_id": section.section_id,
                "next_id": section.next_id,
                "items": [
                    self.serialise_item(item)
                    for item in section.items
                ],
            }
        )

    def serialise_item(self, item: Notification) -> dict:
        return {
            "id": item.id,
            "originator_id": item.originator_id.hex,
            "originator_version": item.originator_version,
            "topic": item.topic,
            "state": b64encode(item.state).decode("utf8"),
        }
```

We can construct the JSON view with the application log of the `BankAccounts` application from Chapter 9.

```
app = BankAccounts()

view = JSONNotificationLogView(app.log)
```

Let's call the `get()` method of the `view` object with a section ID. The view will return a JSON document representing a section without any items.

There are no event notifications in this notification log section because there are not yet any aggregates in this application object.

```
s = view.get(section_id="1,10")
assert s.startswith('{"section_id": ')
assert s.endswith('"items": []}')
assert "BankAccount.Opened" not in s
```

The JSON notification log view can be used within a broader interface that presents the command and query methods of an application. For example, the class `BankAccountsJSONAPI` defined below uses the class `JSONNotificationLogView` as part of a broader JSON API for the bank accounts application.

Following the adapter pattern, its interface is defined using abstract base classes to define an API that will be used by the client side later in this example.

Firstly, the abstract base class `NotificationLogAPI` suggests that any application log could be presented in a serialised format.

```
class NotificationLogAPI(ABC):
    @abstractmethod
    def get_log_section(self, section_id: str) -> str:
        pass
```

The abstract base class `BankAccountsAPI` extends the `NotificationLogAPI` and suggests that any serialisation format could be used to present the bank accounts application.

```
class BankAccountsAPI(NotificationLogAPI):
    @abstractmethod
    def open_account(self, body: str) -> str:
        pass
```

This interface will be used later in this example by the bank accounts client object. Remote clients won't have a direct reference to the application object. However, the `BankAccountsJSONAPI` will be a concrete interface for presenting application objects. Therefore, it will have an actual application object, and so will any other such interface.

For that reason, the abstract base class `ApplicationAdapter` is defined below. It expects to be constructed with an application object. It declares an abstract method `construct_log_view()` which must be implemented by concrete adapters. The adapter pattern is one of the twenty-three patterns in the book *Design Patterns: Elements of Reusable Object-Oriented Software*.

```
class ApplicationAdapter(ABC, Generic[TApplication]):
    def __init__(self, app: TApplication):
        self.app = app
        self.log = self.construct_log_view()

    @abstractmethod
    def construct_log_view(
        self,
    ) -> AbstractNotificationLogView:
        pass
```

The class `BankAccountsJSONAPI` inherits from both `BankAccountsAPI` and `ApplicationAdapter`, and implements a serialised interface to the bank accounts application that uses the JSON format.

```
class BankAccountsJSONAPI(
    BankAccountsAPI,
    ApplicationAdapter[BankAccounts],
):
    def construct_log_view(self):
        return JSONNotificationLogView(self.app.log)

    def get_log_section(self, section_id: str) -> str:
        return self.log.get(section_id)

    def open_account(self, request: str) -> str:
        kwargs = json.loads(request)
        account_id = self.app.open_account(**kwargs)
        return json.dumps({"account_id": account_id.hex})
```

We can imagine implementing serialised adapters to the bank accounts application that use other serialisation formats, but we will focus on JSON in this example.

Let's construct the JSON bank accounts adapter with the bank accounts application object.

```
adapter = BankAccountsJSONAPI(app)
```

Using this broader interface, we can both open an account and get the notification log using JSON documents. Once we have opened an account, the serialised notification log section will contain a serialised `BankAccount.Opened` event notification.

```
msg = adapter.open_account(
    json.dumps(
        {
            "full_name": "Alice",
            "email_address": "alice@example.com",
        }
    )
)
```

```
s = adapter.get_log_section(section_id="1,10")
assert s.startswith('{"section_id": ')
assert "BankAccount.Opened" in s

account_id1 = UUID(json.loads(msg)["account_id"])
```

Now that we can use the `adapter` to interact with the application object in terms of JSON documents, let's turn our attention to the client side.

API client

The class `RemoteNotificationLog` defined below implements the abstract base class `AbstractNotificationLog` that was introduced in Chapter 6. It therefore presents the same interface as the `LocalNotificationLog`. It will be constructed with a `NotificationLogAPI` object. It implements the `__getitem__()` method of `AbstractNotificationLog` by using the API object to obtain a serialised notification log section, which it deserialises into a `Section` object and returns. The `Notification` items are deserialised by calling the helper method `deserialise_item()`.

```
class RemoteNotificationLog(AbstractNotificationLog):
    def __init__(self, api: NotificationLogAPI):
        self.api = api

    def __getitem__(self, section_id: str) -> Section:
        body = self.api.get_log_section(section_id)
        section = json.loads(body)
        return Section(
            section_id=section["section_id"],
            next_id=section["next_id"],
            items=[
                self.deserialise_item(item)
                for item in section["items"]
            ],
        )
```

```
    def deserialise_item(self, item: dict) -> Notification:
        return Notification(
            id=item["id"],
            originator_id=UUID(item["originator_id"]),
            originator_version=item["originator_version"],
            topic=item["topic"],
            state=b64decode(item["state"].encode("utf8")),
        )
```

We can construct a remote notification log with the `adapter` object created earlier in this example.

```
remote_log = RemoteNotificationLog(adapter)
```

The remote notification log object can be used to obtain an actual `Section` object, containing actual `Notification` objects from the bank accounts application encapsulated by the API.

```
section = remote_log["1,10"]
assert isinstance(section, Section)
assert len(section.items) == 1

notification1 = section.items[0]
assert isinstance(notification1, Notification)
assert notification1.originator_id == account_id1
assert "Alice" in str(notification1.state)
```

We can take this a step further and, just like we used the notification log view within an application interface, we can use the remote notification log within an application client.

The class `BankAccountsJSONClient` defined below does just that. It is constructed with a bank account API object, like the `adapter` object we created and used before.

It is initialised with an instance of the `RemoteNotificationLog` class. The method `open_account` serialises the arguments `full_name` and `email_address` and call the API.

```
class BankAccountsJSONClient:
    def __init__(self, api: BankAccountsAPI):
        self.api = api
        self.log = RemoteNotificationLog(api)

    def open_account(
        self, full_name, email_address
    ) -> UUID:
        body = json.dumps(
            {
                "full_name": full_name,
                "email_address": email_address,
            }
        )
        body = self.api.open_account(body)
        return UUID(json.loads(body)["account_id"])
```

The application client can be used both to open new accounts and obtain sections from the notification log, in the same way as the actual bank accounts application object.

```
client = BankAccountsJSONClient(adapter)

account_id2 = client.open_account(
    full_name="Bob",
    email_address="bob@example.com"
)

section = client.log["1,10"]
assert isinstance(section, Section)
assert len(section.items) == 2

notification1 = section.items[0]
```

```
assert isinstance(notification1, Notification)
assert notification1.originator_id == account_id1
assert "Alice" in str(notification1.state)

notification2 = section.items[1]
assert isinstance(notification2, Notification)
assert notification2.originator_id == account_id2
assert "Bob" in str(notification2.state)
```

We can see that the `BankAccount.Opened` events are presented by the client in the order in which they were created in the application.

Transfer protocol

So far in this example we have defined an API that can be used to interact with an application object using JSON documents. And we have defined a client that can operate with an application object via the API. Now let's introduce HTTP as a transfer protocol for mediating between the client and the application's serialised interface adapter.

Firstly, the class `HTTPApplicationServer` defined below functions as generic single-threaded Web application server. It can only process one request at a time, but that will be sufficient for this example.

It is constructed with an address (host and port number) and a request handler class that will be used to handle the HTTP requests that it receives. It extends the class `Thread` from Python's `threading` module, so that we can run it in a separate thread, and it uses the class `HTTPServer` from Python's `http.server` module to implement the HTTP protocol.

```
class HTTPApplicationServer(Thread):
    prepare: List[Callable] = []

    def __init__(self, address, handler):
        super(HTTPApplicationServer, self).__init__(
            daemon=True
```

```
        )
        self.server = HTTPServer(
            server_address=address,
            RequestHandlerClass=handler,
        )

    def run(self):
        [f() for f in self.prepare]
        self.server.serve_forever()

    def stop(self):
        self.server.shutdown()
        self.join()

    @classmethod
    def before_first_request(cls, f):
        HTTPApplicationServer.prepare.append(f)
        return f
```

The class method `before_first_request()` can be used as a decorator to register functions that need to be called before requests are handled.

The function `init_bank_accounts()` constructs the bank accounts application within the JSON application adapter we defined above. It is decorated with `@HTTPApplicationServer.before_first_request` so that the bank accounts application and adapter are constructed when the server is started.

```
@HTTPApplicationServer.before_first_request
def init_bank_accounts() -> None:
    global adapter
    adapter = BankAccountsJSONAPI(BankAccounts())
```

The class `BankAccountsHTTPHandler` defined below functions effectively as a simple Web application that presents the bank accounts API as an HTTP API. It extends the base class `BaseHTTPRequestHandler` from the Python

Standard Library. It implements a method for handling `POST` requests that detects and fulfills requests to open new accounts. It implements a method for handling `GET` requests that detects and fulfills requests for notification log sections. It uses the `adapter` object that will be constructed by the `init_bank_accounts()` function when the HTTP server starts.

```
class BankAccountsHTTPHandler(BaseHTTPRequestHandler):
    def do_PUT(self):
        if self.path.startswith("/accounts/"):
            length = int(self.headers["Content-Length"])
            request_msg = self.rfile.read(length).decode(
                "utf8"
            )
            body = adapter.open_account(request_msg)
            status = 201
        else:
            body = "Not found: " + self.path
            status = 404
        self.send(body, status)

    def do_GET(self):
        if self.path.startswith("/notifications/"):
            section_id = self.path.split("/")[-1]
            body = adapter.get_log_section(section_id)
            status = 200
        else:
            body = "Not found: " + self.path
            status = 404
        self.send(body, status)

    def send(self, body: str, status: int):
        self.send_response(status)
        self.send_header("Content-type", "text/html")
        self.end_headers()
        self.wfile.write(body.encode("utf8"))
```

Let's start the application server with the bank accounts handler.

```
server_address = ("", 8000)

server = HTTPApplicationServer(
    address=server_address,
    handler=BankAccountsHTTPHandler
)
server.start()
```

Now let's define an HTTP client for interacting with the HTTP server. The `BankAccountsHTTPClient` class defined below implements the `BankAccountsAPI` interface defined above. It is constructed with a server address (host and port number). It is initialised with an HTTP connection object that will be used to interact with an HTTP server.

```
class BankAccountsHTTPClient(BankAccountsAPI):
    def __init__(self, server_address):
        self.connection = HTTPConnection(*server_address)

    def get_log_section(self, section_id: str) -> str:
        return self.request(
            "GET", "/notifications/{}".format(section_id)
        )

    def open_account(self, body: str) -> str:
        return self.request(
            "PUT", "/accounts/", body.encode("utf8")
        )

    def request(self, method, url, body=None):
        self.connection.request(method, url, body)
        return self.get_response()

    def get_response(self):
        response = self.connection.getresponse()
        return response.read().decode()
```

We can then construct the `BankAccountsJSONClient` defined above, with the `BankAccountsHTTPClient` using the same address as the HTTP server.

```
client = BankAccountsJSONClient(
    BankAccountsHTTPClient(
        server_address=server_address
    )
)
```

We can then interact with the bank accounts application via HTTP to open two accounts and get the resulting event notifications.

```
account_id1 = client.open_account(
    full_name="Alice",
    email_address="alice@example.com"
)
account_id2 = client.open_account(
    full_name="Bob",
    email_address="bob@example.com"
)

# Get the event notifications.
section = client.log["1,10"]
assert len(section.items) == 2

assert section.items[0].originator_id == account_id1
assert section.items[0].topic.endswith(
    "BankAccount.Opened"
), section.items[0].topic

assert section.items[1].originator_id == account_id2
assert section.items[1].topic.endswith(
    "BankAccount.Opened"
)

server.stop()
```

In this way, the state of an application can be reliably propagated across operating system process boundaries, and across computer networks.

As we have seen in Chapter 9, the application can be configured using environment variables to use alternative infrastructure. This example uses the "Plain Old Python Objects" infrastructure factory, but we could have used the SQLite infrastructure instead.

In the case where a cipher is used, the event notification state will be encrypted when presented by the API. The client will then need to use an appropriately constructed mapper to decipher the event notifications before the domain events can be reconstructed. This will be discussed further in Chapter 13.

The other command and query methods of the bank accounts application, `credit_account()` and `get_balance()`, could easily be added to the API, but they have not been implemented in the example above simply because this example is primarily intended to illustrate how to implement a remote notification log.

Intermission 2 Tracking

When processing events from a notification log, we can lose track of position.

therefore…

Write records to track position in a notification log.

Intermission 1 introduced a recorder which inserted and selected stored events from two sequences, one for an aggregate and one for the application. The aggregate sequence was used in Part 1 to obtain the events for an aggregate, and the event notification sequence was used in Part 2 to obtain event notifications so that the state of an application can be propagated reliably.

The reason for propagating the state of an application is to process the domain events. When processing the domain events of an application, we need to track the position in the event notification sequence in a durable way, so that we can resume from the correct place when restarting the processing. We can also use the recorded position to make sure that an event notification hasn't already been processed.

By tracking the position of an event notification along with the consequences of processing the event notification in an atomic database transaction, we can achieve "exactly once" event processing. The advantage of "exactly once" processing is that the event processing can be deterministic. If the position were to be recorded separately from the consequences of the processing, then sudden terminations of the processing may lead to inconsistencies. Recording the position of the processed event notification separately from consequences of the processing of the event notification is another example of "dual writing". An example of "dual writing" in this situation is sending an acknowledgement to an AMQP server before or after updating a database.

The consequences of processing a domain event may include new domain events and new event notifications, and it may include new or updated or deleted objects of other types. Together with the tracking information, we can identify a new type of object that I have named "process event".

A process event object can constructed with the tracking information and passed into a processing policy and used to collect the consequences of processing an event notification. We can then understand the purpose of a process recorder as recording all the factors of a process event atomically.

In summary, a process recorder:

- Records position in the event notification sequence being processed
- Records new stored events in aggregate sequence
- Records new event notifications in event notification sequence
- All factors of "process event" written atomically

Example

Using an process recorder, `StoredEvent` objects can be written along with a tracking information that records position in an event notification sequence that is being processed.

The dataclass `Tracking` defined below has two fields: `application_name` and `notification_id`.

```
class Tracking(ImmutableObject):
    application_name: str
    notification_id: int
```

Let's define an abstract base class for process recorders. The class `ProcessRecorder` defined below extends the `ApplicationRecorder` from Intermission 1.

```
class ProcessRecorder(ApplicationRecorder):
    @abstractmethod
```

```
    def max_tracking_id(
        self, application_name: str
    ) -> int:
        pass
```

The method `max_tracking_id()` will return the position in an application sequence of the last event notification that has been processed. This will be used to position a notification log reader when resuming to process event notifications.

Before implementing the abstract base class, let's firstly define a test for process recorders. As with the test for application recorders, we will insert events. But as along with stored event objects, we will also pass in a `Tracking` object. The test checks that recording events with the same tracking position results in an integrity error.

```
from uuid import uuid4

def test(recorder: ProcessRecorder):

    # Get current position.
    assert recorder.max_tracking_id('upstream_app') == 0

    # Write two stored events.
    originator_id1 = uuid4()
    originator_id2 = uuid4()

    stored_event1 = StoredEvent(
        originator_id=originator_id1,
        originator_version=0,
        topic='topic1',
        state=b'state1'
    )
    stored_event2 = StoredEvent(
        originator_id=originator_id1,
        originator_version=1,
        topic='topic2',
```

```
    state=b'state2'
)
stored_event3 = StoredEvent(
    originator_id=originator_id2,
    originator_version=1,
    topic='topic3',
    state=b'state3'
)
tracking1 = Tracking(
    application_name='upstream_app',
    notification_id=1,
)
tracking2 = Tracking(
    application_name='upstream_app',
    notification_id=2,
)

recorder.insert_events(
    stored_events=[
        stored_event1,
        stored_event2,
    ],
    tracking=tracking1
)

# Get current position.
assert recorder.max_tracking_id('upstream_app') == 1

try:
    recorder.insert_events(
        stored_events=[
            stored_event3
        ],
        tracking=tracking1
    )
except Recorder.IntegrityError:
    pass
```

```
    else:
        raise Exception("Did not raise IntegrityError")


    # Get current position.
    assert recorder.max_tracking_id('upstream_app') == 1

    recorder.insert_events(
        stored_events=[
            stored_event3
        ],
        tracking=tracking2
    )

    # Get current position.
    assert recorder.max_tracking_id('upstream_app') == 2
```

POPO

Now let's define a process recorder that uses "Plain Old Python Objects".

The class `POPOProcessRecorder` below implements the abstract base class `ProcessRecorder` by extending `POPOApplicationRecorder` from Intermission 1.

The method `create_table()` is extended to define a `dict` that records tracking information. The method `update_table` is extended to append the `Tracking` object's `notification_id` to the list identified by the `application` name. The `assert_uniqueness` method is extended to check the tracking record before stored events are inserted.

```
class POPOProcessRecorder(
    ProcessRecorder, POPOApplicationRecorder
):
    def __init__(self) -> None:
        super().__init__()
        self.tracking_table: Dict[str, int] = defaultdict(
```

```
            None
        )

    def assert_uniqueness(
        self, stored_events: List[StoredEvent], **kwargs
    ) -> None:
        super().assert_uniqueness(stored_events, **kwargs)
        tracking: Optional[Tracking] = kwargs.get(
            "tracking", None
        )
        if tracking:
            last = self.tracking_table.get(
                tracking.application_name, 0
            )
            if tracking.notification_id <= last:
                raise self.IntegrityError

    def update_table(
        self, stored_events: List[StoredEvent], **kwargs
    ) -> None:
        super().update_table(stored_events, **kwargs)
        tracking: Optional[Tracking] = kwargs.get(
            "tracking", None
        )
        if tracking:
            self.tracking_table[
                tracking.application_name
            ] = tracking.notification_id

    def max_tracking_id(
        self, application_name: str
    ) -> int:
        with self.database_lock:
            try:
                return self.tracking_table[
                    application_name
                ]
```

```
            except KeyError:
                return 0
```

We use the `POPOProcessRecorder` to "record" stored events along with a position in an event notification sequence in memory, without using a real database, which is the fastest and easiest way of developing a system.

The test defined above can be used to check the `POPOProcessRecorder` is working as it should.

```
# Construct the POPO recorder.
recorder = POPOProcessRecorder()

# Run the test.
test(recorder)
```

SQLite

Let's also implement a process recorder using SQLite. Below, the class `SQLiteProcessRecorder` implements the abstract base class `ProcessRecorder` by extending `SQLiteApplicationRecorder` from Intermission 1.

The method `_create_table()` is extended to create a database table that stores tracking records. And the method `_insert_events` is extended so that when stored events are inserted a tracking record is written atomically. The method `max_tracking_id()` is implemented to get the highest notification ID that has been recorded as a tracking record.

```
class SQLiteProcessRecorder(
    ProcessRecorder,
    SQLiteApplicationRecorder,
):
    def _create_table(self, c: Connection):
        super()._create_table(c)
        statement = (
            "CREATE TABLE tracking ("
```

```
            "application_name text, "
            "notification_id int, "
            "PRIMARY KEY "
            "(application_name, notification_id))"
        )
        c.execute(statement)

    def max_tracking_id(
        self, application_name: str
    ) -> int:
        params = [application_name]
        c = self.db.get_connection().cursor()
        statement = (
            "SELECT MAX(notification_id)"
            "FROM tracking "
            "WHERE application_name=?"
        )
        c.execute(statement, params)
        return c.fetchone()[0] or 0

    def _insert_events(
        self,
        c: Connection,
        stored_events: List[StoredEvent],
        **kwargs,
    ) -> None:
        super()._insert_events(c, stored_events, **kwargs)
        tracking: Optional[Tracking] = kwargs.get(
            "tracking", None
        )
        if tracking is not None:
            statement = (
                "INSERT INTO tracking " "VALUES (?,?)"
            )
            try:
                c.execute(
                    statement,
```

```
                (
                    tracking.application_name,
                    tracking.notification_id,
                ),
            )
        except sqlite3.IntegrityError as e:
            raise self.IntegrityError(e)
```

We can use the `SQLiteProcessRecorder` to record stored events along with a position in an event notification sequence, get the current position in the upstream event notification sequence so that we can resume pulling a processing event notifications.

```
# Construct the SQLite recorder.
recorder = SQLiteProcessRecorder(
    SQLiteDatabase(':memory:')
)

# Create tables.
recorder.create_table()

# Run the test.
test(recorder)
```

PostgreSQL

We can also implement a process recorder using PostgreSQL. Below, the class `PostgresProcessRecorder` implements the abstract base class `ProcessRecorder` by extending `PostgresApplicationRecorder` from Intermission 1.

```
class PostgresProcessRecorder(
    PostgresApplicationRecorder,
    ProcessRecorder,
):
    def __init__(
```

```
    self,
    application_name: str = "",
):
    super().__init__(application_name)
    self.tracking_table = (
        self.application_name.lower() + "tracking"
    )

def _create_table(self, c: cursor):
    super()._create_table(c)
    statement = (
        "CREATE TABLE IF NOT EXISTS "
        f"{self.tracking_table} ("
        "application_name text, "
        "notification_id int, "
        "PRIMARY KEY "
        "(application_name, notification_id))"
    )
    c.execute(statement)

def max_tracking_id(
    self, application_name: str
) -> int:
    params = [application_name]
    c = self.db.get_connection().cursor()
    statement = (
        "SELECT MAX(notification_id)"
        f"FROM {self.tracking_table} "
        "WHERE application_name=%s"
    )
    c.execute(statement, params)
    return c.fetchone()[0] or 0

def _insert_events(
    self,
    c: cursor,
    stored_events: List[StoredEvent],
```

```
        **kwargs,
    ) -> None:
        super()._insert_events(c, stored_events, **kwargs)
        tracking: Optional[Tracking] = kwargs.get(
            "tracking", None
        )
        if tracking is not None:
            statement = (
                f"INSERT INTO {self.tracking_table} "
                "VALUES (%s, %s)"
            )
            try:
                c.execute(
                    statement,
                    (
                        tracking.application_name,
                        tracking.notification_id,
                    ),
                )
            except psycopg2.IntegrityError as e:
                raise self.IntegrityError(e)
```

Now let's run the test with the PostgresSQL process recorder.

```
# Construct the Postgres recorder.
recorder = PostgresProcessRecorder()

# Create tables.
recorder.create_table()

# Run the test.
test(recorder)
```

PART 3 SYSTEM

Chapter 11 Log Reader

The notification logs present sections of event notifications, but we want to process events in a sequence from a given position.

therefore…

Use a notification log reader to iterate over the event notifications in local or remote notification log.

A notification log reader supports iterating over an event notification sequence. Whereas a notification log (see Chapters 6 and 10) presents a linked list of sections, a notification log reader will present a sequence of event notifications.

By using the abstract notification log interface, a notification log reader can read from local or remote notification logs.

Event notifications will usually be obtained so that they can be processed, and it makes sense to process each event notification only once. For this reason, it makes sense to start (or resume) reading event notifications from position of the next event notification that has not yet been processed. The position of the last event notification that has been processed may be recorded and obtained using tracking records (introduced in Intermission 2). Whichever way the position of the next event notification is determined, the start position will need to be given to the reader when reading from the event notification sequence.

In summary, a notification log reader:

- Reads event notifications from a notification log
- Starts from given position in the event notification sequence

Example

In the example below, the class `NotificationLogReader` implements a notification log reader. It is constructed with a notification log.

The `read()` method has an argument `start` which is used to construct a section ID. The section ID is used to make a request to the notification log for a section. The event notifications of the section are yielded, and the next section is then obtained, until the end of the sequence is reached.

```
class NotificationLogReader:
    DEFAULT_SECTION_SIZE = 10

    def __init__(
        self,
        notification_log: AbstractNotificationLog,
        section_size: int = DEFAULT_SECTION_SIZE,
    ):
        self.notification_log = notification_log
        self.section_size = section_size

    def read(
        self, *, start: int
    ) -> Iterable[Notification]:
        section_id = "{},{}".format(
            start, start + self.section_size - 1
        )
        while True:
            section: Section = self.notification_log[
                section_id
            ]
            for item in section.items:
                if item.id < start:
                    continue
                yield item
            if section.next_id is None:
                break
```

```
            else:
                section_id = section.next_id
```

We can use the log reader to iterate over a sequence of event notifications.

Firstly, let's construct one of the process recorders from Intermission 2.

```
recorder = SQLiteProcessRecorder(
    SQLiteDatabase(':memory:')
)
recorder.create_table()
```

Let's use the recorder to construct a local notification log, using the `LocalNotificationLog` class that was defined in Chapter 6.

```
# Construct notification log.
notification_log = LocalNotificationLog(
    recorder, section_size=5
)
```

We can now construct a log reader, using the `NotificationLogReader` class defined above.

```
reader = NotificationLogReader(notification_log)
```

At first, when we call the reader's `read()` method, there will not be any event notifications because there are no stored events in the process recorder.

```
notifications = list(reader.read(start=1))
assert len(notifications) == 0
```

So let's write five stored event objects into the process recorder.

```
# Write 5 events.
originator_id = uuid4()
```

```
for i in range(5):
    stored_event = StoredEvent(
        originator_id=originator_id,
        originator_version=i,
        topic='topic',
        state=b'state'
    )
    recorder.insert_events(
        stored_events=[stored_event],
    )
```

Now when we call the reader's `read()` method, there will be five event notifications.

```
notifications = list(reader.read(start=1))
assert len(notifications) == 5
```

If we start at the second position rather than the first, then four event notifications will be returned.

```
notifications = list(reader.read(start=2))
assert len(notifications) == 4
```

Similarly, if we start at the fifth, then one event notification will be returned.

```
notifications = list(reader.read(start=5))
assert len(notifications) == 1
```

And if we start at the sixth position, then zero event notifications will be returned.

```
notifications = list(reader.read(start=6))
assert len(notifications) == 0
```

Chapter 12 Policy

We can receive event notifications from notification logs. And we can trigger new events by calling aggregate command methods. But we need to decide which aggregate methods will be called in response to which events.

therefore…

Express event responses in an event processing policy.

The purpose of a processing policy is to define how domain events are processed. In processing a domain event, a policy may create event-sourced aggregates, or retrieve and call command methods on existing aggregates. (A processing policy may create or update or delete non-event-sourced objects.)

The definition of a processing policy can be expressed as a function that takes a domain event as an argument, and behaves according to the type of the domain event. There may be more than one aggregate affected by each occasion of processing. To make the occasion of processing explicit, it can help to define a "process event" object, and call the policy function with the process event object as well as the domain event object.

There may be zero, one, many new domain events created for each domain event processed by a policy. The process event object can be used to collect all the domain events that are created by an occasion of processing. (The process event can also be used to track which non-event-sourced objects need to be saved or deleted.)

The process event object can also keep track of the position in an event notification sequence of the existing domain event that is being processed.

All the factors of the process event will need to be recorded atomically, as described in Intermission 2, so that the processing can be safely be started again if it happens that it doesn't fully complete.

In summary, processing policy:

- Responds to event notifications by calling factory methods or command method on existing aggregates
- Defines occasions of processing that must be recorded atomically
- Can involve event-sourced or non-event-sourced objects

Example

In the example below, an event processing policy is defined as a function which processes `BankAccount.Opened` events into "email notification" aggregates. It will be called with a domain event and a "process event" object, and the process event object will be used to collect new domain events.

The class `EmailNotification` defined below inherits from the `Aggregate` class. It is constructed with arguments `to`, `subject`, and `message` which gesture towards the information required to create an email message. The class method `create()` calls the base class method `_create_()` and involves arguments `to`, `subject`, and `message` in the triggering of the `EmailNotification.Created` event. The base aggregate event class `Created` is extended to have these arguments as its attributes.

```
class EmailNotification(Aggregate):
    def __init__(self, to, subject, message, **kwargs):
        super(EmailNotification, self).__init__(**kwargs)
        self.to = to
        self.subject = subject
        self.message = message

    @classmethod
    def create(cls, to, subject, message):
        return super()._create_(
            cls.Created,
            to=to,
            subject=subject,
```

```
            message=message,
        )

    class Created(Aggregate.Created):
        to: str
        subject: str
        message: str
```

The class `ProcessEvent` defined below represents an actual occasion of processing. It will be constructed with a `Tracking` object (introduced in Intermission 2) that indicates the position of a `Notification` (introduced in Intermission 1) in a notification log.

```
class ProcessEvent:
    def __init__(self, tracking: Tracking):
        self.tracking = tracking
        self.events: List[Aggregate.Event] = []

    def collect(self, aggregates: List[Aggregate]):
        for aggregate in aggregates:
            self.events += aggregate._collect_()
```

Its method `collect()` is similar to the `save()` method of the `Application` class and collects aggregate events from a given list of aggregate objects. But rather than storing these events directly, it appends them to its own list of events waiting to be recorded. The process event objects are used by processing policies to collect new domain events from aggregates changed by a policy.

The function `policy()` defined below will be called with a domain event object and a process event object. It is decorated with the `@singledispatch` decorator from the Python `functools` module. By decorating the function in this way, we can register other functions that will be invoked when the first argument of a call to `policy()` matches a particular type.

A second function is registered with the policy function so that when the policy is called with a `BankAccount.Opened` object, the second function is invoked. The second function is implemented to create a new email notification aggregate using information from the bank account opened aggregate event. The process event is then used to collect the pending aggregate events.

```
from functools import singledispatch


@singledispatch
def policy(domain_event, process_event):
    pass

@policy.register(BankAccount.Opened)
def _(domain_event, process_event):
    notification = EmailNotification.create(
        to=domain_event.email_address,
        subject="Your New Account",
        message="Dear {}".format(domain_event.full_name)
    )
    process_event.collect([notification])
```

We can the open a new bank account, by calling the `open()` method of the `BankAccount` aggregate.

```
# Open an account.
account = BankAccount.open(
    full_name="Alice",
    email_address="alice@example.com"
)
```

We can collect the pending event from the aggregate, which will be a `BankAccount.Opened` event.

```
events = account._collect_()
account_opened = events[0]
```

Let's imagine this domain event exists in an "upstream" event notification sequence, and then let's create a "tracking" object that captures its notification ID.

```
tracking = Tracking(
    application_name='upstream_app',
    notification_id=5,
)
```

Let's prepare for calling the policy by constructing a process event object with this `tracking` object.

```
process_event = ProcessEvent(tracking)
```

We can then process the "account opened" domain event by calling the `policy()` function with both the domain event object and the process event object.

```
policy(account_opened, process_event)
```

The result is that the process event ends up with an `EmailNotification.Created` event in its list of `events`. The new domain event is associated with the tracking information in the process event.

```
assert len(process_event.events) == 1
event = process_event.events[0]
assert isinstance(event, EmailNotification.Created)

assert process_event.tracking.notification_id == 5
assert process_event.tracking.application_name == (
    'upstream_app'
)
```

This technique is used in the process applications that are the focus of the next chapter.

Chapter 13 Process

We can read event notifications from a notification log using a reader. We can process domain events using a policy to generate more events. And we can keep track of position in a notification log using tracking records.

therefore…

Extend the application to read and process event notifications, writing the resulting domain events with a tracking record in an atomic database transaction.

In Chapter 9 we discussed an application object that can propagate the state of its aggregates using a notification log. In Intermission 2 we developed a recorder that can store tracking records along with new domain events. In Chapter 11 we defined a notification log reader that can pull event notifications from a notification log. And in Chapter 12 we developed a processing policy that can take different actions depending on the type of domain event.

The purpose of this chapter is to bring these ideas together by extending the idea of the application object so that it can pull and process event notifications as domain events with "exactly once" semantics.

The general routine will involve getting event notifications from a log reader, extracting the domain event and its position in the event notification sequence, calling the policy with the domain event, and storing the consequences of the policy response atomically with the position in the event notification sequence.

We need to avoid any dual writing of the position of the event that was processed and the consequences of process the domain event at that position, so that either both are recorded or neither are. For if the position is recorded without the consequences then the domain event will effectively

have been ignored, and if the consequences are recorded without the position then the domain event may be processed twice. This is another example of avoiding “dual writing”, and is the counterpart to the avoiding of “dual writing” that is accomplished by recording event notifications along with domain events. The avoidance of both of these forms of “dual writing” is necessary to make a reliable distributed system.

The consequences of processing a domain event could be new domain events in an event-sourced domain model. That is, one event-sourced domain model can be projected into another. It is also possible to process a domain event by making changes to a non-event-sourced model, for example a search engine, or custom ORM objects, or pre-prepared views. So long as the position is recorded atomically with the consequences of the processing of the domain events, the results will be reliable.

Processing domain events by calling an external API that doesn’t register the position in the event notification sequence will require some more care. It may be that the calls are idempotent in some way or other. Otherwise, given the processing of a domain event may suddenly be terminated (for example due to a power cut or network partition), we need to choose between “at least once” semantics and “best effort” (or “perhaps never”) semantics.

If the state of an application depends is entirely the consequences of processing domain events from an event notification sequence, the state of the application may be fully a deterministic function of the event notification sequence. However, if the application also receives commands directly from users, or if the application pulls from two different event notification sequences, then the possibility arises for the state of the application to be indeterminate, since the order in which things are done can vary.

Another consideration is knowing when to read domain event notifications. An application that is processing domain event notifications can poll a notification log reader at regular intervals, in which case the latency of processing will be governed by the polling interval. If the polling frequency is increased, the latency of processing will be lower, but the chances of doing unnecessary work will increase. If the polling frequency is decreased,

the changes of doing unnecessary work will decrease, but the latency of processing will be higher. For these reasons, it can make sense for one application to prompt, or push another application into processing, new event notifications when they are created. A prompt can be handled by setting a flag that is cleared when processing is started, and waited upon when processing is completed, so that processing resumes as soon as there is something to process.

In summary, a process application:

- Follows application notification logs
- Calls event processing policy
- Writes "process events" with tracking record, domain events, and event notifications

Example

The example below defines and uses an "email notifications" process application to process the event notifications from the `BankAccounts` application defined in Chatper 9. It has a policy like the one introduced in Chapter 11, and uses the `NotificationLogReader` introduced in Chapter 12.

A base class for process applications is defined which can be used to more easily define process applications. The base class for process applications depends on a few other classes that will be defined next, `Promptable`, `Follower`, `Leader`. These classes offer a separation of concerns that makes more explicit the behaviour of process applications, and also opens the way for the development of system runners in Chapter 15.

The abstract base class `Promptable` declares an abstract method `receive_prompt()` that can be implemented to take action when a prompt is received. A prompt is no more than the name of a leader application that has notified its followers that new events notifications are available in its notification log. The "promptable" classes in this book are the `Follower` class (defined below), the `SingleThreadedRunner` class (defined in Chapter

15), and the `FollowerThread` class used by the `MultiThreadedRunner` (also defined in Chapter 15).

```
class Promptable(ABC):
    @abstractmethod
    def receive_prompt(self, leader_name: str) -> None:
        pass
```

Now, let's consider the `Follower` class defined below. It inherits both the `Promptable` class and the `Application` class from Chapter 9. It overrides the `create_recorder()` method of the `Application` class, so that a `ProcessRecorder` is constructed by the infrastructure factory instead of an `ApplicationRecorder`. The `ProcessRecorder` is needed as the application's aggregate event recorder so tracking information is recorded atomically with new domain events. This is needed to implement "exactly once" event processing.

It defines a method `follow()` which is called with the name of an application to be followed, and a notification log for that application. This method functions by constructing a notification log reader with the given notification log and this application's process recorder so that it can know the position of the last event in that log processed by this application. A mapper is also constructed using the name of application so that the correct configuration (e.g. correct cipher key) is picked up from the environment. The reader and the mapper are added to the collection of `followers`.

It implements the `receive_prompt()` method of the `Promptable` class by calling its `pull_and_process()` method with the given name of the promoting leader application.

The `pull_and_process()` method functions by using the given name to select one of the notification log readers and mappers constructed by the `follow()` method. It uses the reader to pull new event notifications from the leader. Each new event notification is firstly mapped to a domain event object, and then the processing policy method `policy()` is called with that domain event object and a new instance of the class `ProcessEvent` introduced in Chapter 12.

The `Follower` class defines an abstract method `policy()` which needs to be implemented by concrete follower applications so that they can process domain events obtained from their leaders. The `policy()` method will use the domain event and the current state of the application (as viewed through the aggregates in the application's repository) to create or update the application's aggregates. The policy will need to call `collect()` on the given process event object to collect new domain events to be stored after the policy has finished its work. The policy needs to use the process event rather than calling the `save()` method so that the tracking information is stored atomically with the new domain events.

When the processing policy has completed its work, the process event object is passed to the `record()` method. The `record()` method functions by passing the attributes of the process event to the `put()` method of the application's aggregate event store. Once the process event has been recorded, the `notify()` method is called.

```
class Follower(Promptable, Application):
    def __init__(self):
        super().__init__()
        self.readers: Dict[
            str,
            Tuple[
                NotificationLogReader,
                Mapper[Aggregate.Event],
            ],
        ] = {}

    def construct_recorder(self) -> ApplicationRecorder:
        return self.factory.process_recorder()

    def follow(
        self, name: str, log: AbstractNotificationLog
    ):
        assert isinstance(self.recorder, ProcessRecorder)
        reader = NotificationLogReader(log)
        mapper = self.construct_mapper(name)
```

```
        self.readers[name] = (reader, mapper)

    def receive_prompt(self, leader_name: str) -> None:
        self.pull_and_process(leader_name)

    def pull_and_process(self, name: str) -> None:
        reader, mapper = self.readers[name]
        start = self.recorder.max_tracking_id(name) + 1
        for notification in reader.read(start=start):
            domain_event = mapper.to_domain_event(
                notification
            )
            process_event = ProcessEvent(
                Tracking(
                    application_name=name,
                    notification_id=notification.id,
                )
            )
            self.policy(
                domain_event,
                process_event,
            )
            self.record(process_event)

    def record(self, process_event: ProcessEvent) -> None:
        self.events.put(
            **process_event.__dict__,
        )
        self.notify(process_event.events)

    @abstractmethod
    def policy(
        self,
        domain_event: Aggregate.Event,
        process_event: ProcessEvent,
    ):
        pass
```

Next, let's consider the `Leader` class defined below. It inherits from the `Application` class from Chapter 9. It defines a new method `lead()` which will be used to configure leader applications with a list of followers (promptable objects). The `notify()` method of the `Application` class is extended so that `prompt_followers()` is called whenever there are new event notifications. The method `prompt_followers()` iterates over the promptable followers that have been registered by the `lead()` method and calls each of their `receive_prompt()` methods with the name of the application. In this way, whenever aggregates are saved in application command methods or after processing an event notification by a policy, the followers of a leader will be prompted to pull the new domain event notifications.

```
class Leader(Application):
    def __init__(self):
        super().__init__()
        self.followers: List[Promptable] = []

    def lead(self, follower: Promptable):
        self.followers.append(follower)

    def notify(self, new_events: List[Aggregate.Event]):
        super().notify(new_events)
        if len(new_events):
            self.prompt_followers()

    def prompt_followers(self):
        name = self.__class__.__name__
        for follower in self.followers:
            follower.receive_prompt(name)
```

Finally, the class `ProcessApplication` is defined simply as a `Leader` and a `Follower`. This means that a process application can both be prompted by leaders to pull and process their event notifications, and also prompt followers of the existence of new event notifications.

```
class ProcessApplication(Leader, Follower):
    pass
```

Altogether, this means that whenever new domain events have been stored after aggregates have been changed, either in command methods or in processing policies, followers can be prompted to pull new event notifications. But the prompting and the pulling depends on configuring the leaders with promptables and the followers with notification log readers.

For example, the class `EmailNotifications` is defined as a process application that has a processing policy which will process the `BankAccount.Opened` events defined in Chapter 2 by creating `EmailNotification` aggregates defined in Chapter 12.

```
class EmailNotifications(ProcessApplication):
    @singledispatchmethod
    def policy(
        self,
        domain_event: Aggregate.Event,
        process_event: ProcessEvent,
    ):
        pass

    @policy.register(BankAccount.Opened)
    def _(
        self,
        domain_event: Aggregate.Event,
        process_event: ProcessEvent,
    ):
        assert isinstance(domain_event, BankAccount.Opened)
        notification = EmailNotification.create(
            to=domain_event.email_address,
            subject="Your New Account",
            message="Dear {}, ...".format(
                domain_event.full_name
            ),
```

```
        )
        process_event.collect([notification])
```

To demonstrate the `Leader` class, we can redefine the `BankAccounts` application that was defined in Chapter 9 by subclassing it as a `Leader`.

```
class BankAccounts(Leader, BankAccounts):
    pass
```

Let's construct both the bank accounts and the email notifications applications.

```
accounts = BankAccounts()
notifications = EmailNotifications()
```

Now that we have a two applications, we can firstly configure the email notifications application to follow the bank accounts application.

```
notifications.follow(
    name="BankAccounts",
    log=accounts.log
)
```

Initially, both of the applications have an empty notification log.

```
section = accounts.log["1,5"]
assert len(section.items) == 0

section = notifications.log["1,5"]
assert len(section.items) == 0
```

When we open an account, the accounts application notification log will have an event notification. But until the email notifications application is prompted, it will still have an empty notification log.

```
accounts.open_account("Alice", "alice@example.com")

section = accounts.log["1,5"]
assert len(section.items) == 1

section = notifications.log["1,5"]
assert len(section.items) == 0
```

When we prompt the email notifications application, it will pull and process the `BankAccount.Opened` event, and generate an `EmailNotification` aggregate.

```
notifications.receive_prompt("BankAccounts")
```

The `EmailNotification.Created` event will then appear in its notification log. The `Notification` class was introduced in Intermission 1.

```
section = notifications.log["1,5"]
assert len(section.items) == 1

event_notification = section.items[0]
assert isinstance(event_notification, Notification)
assert event_notification.topic.endswith(
    "EmailNotification.Created"
) is True
```

Due to the tracking of the position in the log that is being processed, prompting the email notifications application again will not generate another email notification.

```
notifications.receive_prompt("BankAccounts")

section = notifications.log["1,5"]
assert len(section.items) == 1
```

However, when the email notifications application is prompted after a second account is opened, so that a second event notification is presented by the notification log of the bank accounts application, then another email notification will be created.

```
accounts.open_account("Bob", "bob@example.com")

notifications.receive_prompt("BankAccounts")

section = notifications.log["1,5"]
assert len(section.items) == 2, len(section.items)
```

Now, because we subclassed the `BankAccounts` application as a `Leader`, we can configure the bank accounts application to lead the email notifications application.

```
accounts.lead(notifications)
```

Now when we open a third account, the email notifications application will be automatically prompted to pull and process the notification log of the bank accounts application, and a new email notification will be automatically created.

```
accounts.open_account("Jane", "jane@example.com")

section = notifications.log["1,5"]
assert len(section.items) == 3, len(section.items)
```

In this way, a set of applications can be configured to lead and follow each other. An event-driven system of event-sourced applications can be defined entirely independently of infrastructure and mode of running.

The definition and running of such a system of applications is encapsulated by the system and system runner classes that are discussed in Chapters 14 and 15.

Chapter 14 System

A system may involve several applications that process events from the others.

therefore…

Express the relations between application with a set of pipeline expressions in a system.

In Chapter 13, we were able to process the aggregate events of one application object with another by configuring the first to lead the second and the second to follow the first. The purpose of this chapter is to find a way to declare the relations between application classes.

By declaring the relations between application separately from defining the actual interactions between applications, we can defer the decision about the actual interactions and imagine (and develop) different modes of running a system of applications.

The actual interactions can be defined by runners, and will be in Chapter 15. The actual interaction will involve the particular way in which a process application is run, and the particular means of transporting event notifications between application objects.

Since both an application and the relations between a set of applications can be defined independently of particular infrastructure, a system of applications can be defined independently of infrastructure. Such a system can be run in a single thead with non-persistent infrastructure, so that development can be optimised. It can then be run in multiple processes so that a production service can be optimised. If the system is designed to be deterministic in its behaviour, exactly the same results will be obtained in production as in development.

The definition of a system of event-sourced process applications can be recognised as the old "pipes and filters" pattern.

In summary, a system definition:

- Associates applications in pipeline expressions, so that one can follow another
- Can be defined independently of infrastructure
- Can be fully deterministic

Example

The class `System` defined below can be used to define a system of applications. It is constructed with a list of `pipes` which are each a list of application classes. From this list of lists it builds a graph defined by a `dict` of nodes that mappes application class names to application topics, and a list of `edges` that are each a `tuple` that have the name of a "leader" and the name of a "follower". The method `get_app_class()` uses the `name` it is given to identify a topic for an application class, which it resolves to obtain the application class which it returns.

```
class System:
    def __init__(
        self,
        pipes: Iterable[Iterable[Type[Application]]],
    ):
        nodes: Dict[str, Type[Application]] = {}
        edges: Set[Tuple[str, str]] = set()
        # Build nodes and edges.
        for pipe in pipes:
            follower_cls = None
            for cls in pipe:
                nodes[cls.__name__] = cls
                if follower_cls is None:
                    follower_cls = cls
                else:
```

```
                leader_cls = follower_cls
                follower_cls = cls
                edges.add(
                    (
                        leader_cls.__name__,
                        follower_cls.__name__,
                    )
                )

        self.edges = list(edges)
        self.nodes: Dict[str, str] = {}
        for name in nodes:
            topic = get_topic(nodes[name])
            self.nodes[name] = topic
        # Identify leaders and followers.
        self.follows: Dict[str, List[str]] = defaultdict(
            list
        )
        self.leads: Dict[str, List[str]] = defaultdict(
            list
        )
        for edge in edges:
            self.leads[edge[0]].append(edge[1])
            self.follows[edge[1]].append(edge[0])

        # Check followers are followers.
        for name in self.follows:
            if not issubclass(nodes[name], Follower):
                raise TypeError(
                    "Not a follower class: %s"
                    % nodes[name]
                )

        # Check each process is a process application class.
        for name in self.processors:
            if not issubclass(
                nodes[name], ProcessApplication
```

```
        ):
            raise TypeError(
                "Not a follower class: %s"
                % nodes[name]
            )

@property
def leaders(self) -> Iterable[str]:
    return self.leads.keys()

@property
def leaders_only(self) -> Iterable[str]:
    for name in self.leads.keys():
        if name not in self.follows:
            yield name

@property
def followers(self) -> Iterable[str]:
    return self.follows.keys()

@property
def processors(self) -> Iterable[str]:
    return set(self.leaders).intersection(
        self.followers
    )

def get_app_cls(self, name) -> Type[Application]:
    cls = resolve_topic(self.nodes[name])
    assert issubclass(cls, Application)
    return cls

def leader_cls(self, name) -> Type[Leader]:
    cls = self.get_app_cls(name)
    if issubclass(cls, Leader):
        return cls
    else:
        cls = type(
```

```
                cls.__name__,
                (Leader, cls),
                {},
            )
            assert issubclass(cls, Leader)
            return cls

    def follower_cls(self, name) -> Type[Follower]:
        cls = self.get_app_cls(name)
        assert issubclass(cls, Follower)
        return cls
```

Let's define a system using the `BankAccounts` application that was defined in Chapter 9 and the `EmailNotifications` application that was defined in Chapter 13.

```
system = System(
    pipes=[
        [BankAccounts, EmailNotifications],
    ]
)
assert len(system.nodes) == 2

assert len(system.edges) == 1
assert (
    ('BankAccounts', 'EmailNotifications')
    in system.edges
)
```

This simple example defines a system of applications that has two nodes and one edge.

Chapter 15 Runner

We can define a system without infrastructure and without needing it to be run in one particular way. We may want to use different databases and run the system in different ways at different times.

therefore…

Develop different runners for different purposes. Run the system with a particular system runner, configured to use particular database infrastructure.

In Chapter 14 a system of applications was specified independently of particular persistence infrastructure and a particular mode of running. The purpose of this final chapter is to define a collection of system runners that show how a system of applications can be run in different ways.

We can execute the process application processing in different ways, for example there could be a single thread that loops over each application in turn and processes what is available at that point, or there could be separate threads for each process application running in a single operating process or in different operating system processes.

We can pull event notifications in different ways, for example by directly using the notification log of another application, or indirectly by calling into an application through an interface.

The event notifications can also be transported from one application to another in different ways, for example by directly obtaining event notification objects, or by serialising them and transporting them across an operating system process boundary. The serialisation of event notifications could involve different technologies, for example JSON documents or a binary format such as Protocol Buffers (Protobuf).

We can also push prompts, to inform of the existence of new domain event notifications, from one application to another in different ways, for example by calling directly into the application, or by pushing prompts asynchronously.

Calls from one application to another, to push prompts or to pull event notifications, can involve different technologies, such as HTTP or gRPC.

In summary, system runners:

- Offer various running modes (single-threaded, multi-threaded, multi-processing, etc.)
- Use various IPC technology (queues, RESTful, Reactive, Thrift, etc)
- Can introduce concrete infrastructure at runtime, allowing entire system behaviour to be defined without depending on particular infrastructure
- Can run multiple instances of the system pipeline

Example

In Chapter 9 we defined a `BankAccounts` application that uses the `BankAccount` aggregate defined in Chapter 2. In Chapter 13 we defined an `EmailNotifications` application that can process `BankAccount` aggregate events. And in Chapter 14, we defined a system to has the `EmailNotifications` following the `BankAccounts` application. The final piece of the puzzle is to define system runners.

In the examples below, a single-threaded system runner and a multi-threaded system runner are defined. The system defined in Chapter 14 is run in a single-threaded mode and then in a multi-threaded mode, using the "Plain Old Python Objects" and then the SQLite recorders introduced in Intermissions 1 and 2.

Since we will write several different runners, let's define an abstract base class for system runners.

Instances of the class `AbstractRunner` below will be constructed with an instance of the `System` object class, introduced in Chapter 14, which will define the system of applications.

Subclasses will need to implement the `start()` method. A runner will need to construct the applications of the system and set them to lead and follow each other according to the system definition. The `stop()` method will stop the system running.

```
class AbstractRunner:
    def __init__(self, system: System):
        self.system = system
        self.is_started = False

    @abstractmethod
    def start(self) -> None:
        if self.is_started:
            raise self.AlreadyStarted()
        self.is_started = True

    class AlreadyStarted(Exception):
        pass

    @abstractmethod
    def stop(self) -> None:
        pass

    @abstractmethod
    def get(self, cls: Type[A]) -> A:
        pass
```

The `get()` method is used by clients to obtain a reference to the applications of the system, so that their methods can be called. An application class is passed to `get()` and an application instance (or proxy) is returned. Its argument `cls` is typed with a type variable, so that IDEs are convinced that the object returned is an instance of the type requested. This helps with code navigation and code completion.

Single-threaded runner

Now let's define a single-threaded system runner.

The class `SingleThreadedRunner` runs a system of applications in a single thread. It inherits from both the `AbstractRunner` class and the `Promptable` class.

The applications are constructed in a straightforward manner in the `start()` method, and are set to lead and follow each other according to the edges of the system definition. The subtlety is that whilst the applications directly follow other applications, the leaders actually lead the runner. The is why the `SingleThreadedRunner` class extends the `Promptable` cass.

The method `receive_prompt()` method of the `Promptable` class is implemented by appending the names of the leaders to a list. This allows the prompts to be processed iteratively, rather than recursively which would happen if the leaders prompted the follower application directly. By processing the prompts iteratively, we can avoid the recursion depth problems that may otherwise arise.

```
class SingleThreadedRunner(Promptable, AbstractRunner):
    def __init__(self, system: System):
        super(SingleThreadedRunner, self).__init__(system)
        self.apps: Dict[str, Application] = {}
        self.prompts_received: List[str] = []
        self.is_prompting = False

    def start(self):
        super().start()

        # Construct followers.
        for name in self.system.followers:
            app = self.system.follower_cls(name)()
            self.apps[name] = app

        # Construct leaders.
```

```
        for name in self.system.leaders_only:
            app = self.system.leader_cls(name)()
            self.apps[name] = app

        # Lead and follow.
        for edge in self.system.edges:
            leader = self.apps[edge[0]]
            follower = self.apps[edge[1]]
            assert isinstance(leader, Leader)
            assert isinstance(follower, Follower)
            leader.lead(self)
            follower.follow(
                leader.__class__.__name__, leader.log
            )

    def receive_prompt(self, leader_name: str) -> None:
        if leader_name not in self.prompts_received:
            self.prompts_received.append(leader_name)
        if not self.is_prompting:
            self.is_prompting = True
            while self.prompts_received:
                prompt = self.prompts_received.pop(0)
                for name in self.system.leads[prompt]:
                    follower = self.apps[name]
                    assert isinstance(follower, Follower)
                    follower.receive_prompt(prompt)
            self.is_prompting = False

    def stop(self):
        self.apps.clear()

    def get(self, cls: Type[A]) -> A:
        app = self.apps[cls.__name__]
        assert isinstance(app, cls)
        return app
```

Let's run the system defined in from Chapter 14 with the single-threaded runner.

```
system = System(
    pipes=[
        [BankAccounts, EmailNotifications],
    ]
)

runner = SingleThreadedRunner(system)
runner.start()
```

As mentioned in Chapter 9 By default, applications by default use the "Plain Old Python Objects" infrastructure factory, so that the system use the "Plain Old Python Objects" recorders defined in Intermissions 1 and 2.

The `BankAccounts` application is just an `Application` so it will be using an `ApplicationRecorder` which doesn't have tracking records. The `EmailNotifications` application is a `ProcessApplication` and so it will be using a `ProcessRecorder`.

We can get hold of the application objects that have been constructed by the system using the `get()` method. Passing in the application class returns the application instance.

```
accounts = runner.get(BankAccounts)
notifications = runner.get(EmailNotifications)
```

It's important that we access the actual application objects constructed by the system so that we can access the data they have.

At first, the bank accounts and email notifications applications have an empty notification log.

```
section = accounts.log["1,10"]
assert len(section.items) == 0
```

```
section = notifications.log["1,10"]
assert len(section.items) == 0
```

So let's open a bank account, using the `open_account()` method of the `BankAccounts` application.

```
account_id = accounts.open_account(
    full_name="Alice",
    email_address="alice@example.com",
)
```

As we have seen in Chapter 13, the bank accounts application will now have a `BankAccount` aggregate in its repository. Because the account has just been opened, the balance of the account will be 0.00.

```
balance = accounts.get_balance(account_id)
assert balance == Decimal("0.00")
```

The bank accounts application will also have one event notification in its notification log.

```
section = accounts.log["1,10"]
assert len(section.items) == 1
```

Because the single-threaded runner is synchronous, the `open_account()` method will return after the `BankAccount.Opened` event has been processed into an `EmailNotification`. So we can expect the notification log of the email notifications application immediately to show that an email notification aggregate has been created.

Firstly, there is now an event notification in the email notifications application log.

```
section = notifications.log["1,10"]
assert len(section.items) == 1
```

Secondly, the event notification topic indicates that this event notification represents the creation of an `EmailNotification` aggregate.

```
event_notification = section.items[0]
assert isinstance(event_notification, Notification)
assert event_notification.topic.endswith(
    "EmailNotification.Created"
) is True
```

Thirdly, the event notification's `originator_id` attribute could be used to retrieve the `EmailNotification` aggregate. Its `subject` will be "Your New Account" as defined in Chapter 13.

```
email_notification = notifications.repository.get(
    event_notification.originator_id
)
assert isinstance(email_notification, EmailNotification)
assert email_notification.subject == "Your New Account"
```

Hence, we can see that by opening a new account in the bank accounts application, an email notification is created in the email notifications application. The "account created" event was successfully processed by the system to generate an email notification. We can imagine the email notifications application log can be processed to actually send email via an SMTP server. The main point of this demonstration is that the email notifications application is "event driven" by the domain events of the "event-sourced" bank accounts application.

Multi-threaded runner

Now let's define a multi-threaded runner. The class `MultiThreadedRunner` defined below implements the abstract base class `AbstractRunner`.

This time, the "followers" in the system definition are construct as separate threads. The "leaders" are constructed more simply as straightforward application classes.

The “follower” application objects follow the “leader” application objects directly. But the “leader” applications lead the runner threads, just like the “leader” applications in the single-threaded runner lead the runner itself.

```
class MultiThreadedRunner(AbstractRunner):
    """
    Runs system with thread for each follower.
    """

    def __init__(self, system):
        super().__init__(system)
        self.apps: Dict[str, Application] = {}
        self.threads: Dict[str, RunnerThread] = {}
        self.is_stopping = Event()

    def start(self) -> None:
        super().start()

        # Construct followers.
        for name in self.system.followers:
            thread = RunnerThread(
                app_class=self.system.follower_cls(name),
                is_stopping=self.is_stopping,
            )
            thread.start()
            thread.is_ready.wait(timeout=1)
            self.threads[name] = thread
            self.apps[name] = thread.app

        # Construct leaders.
        for name in self.system.leaders_only:
            app = self.system.leader_cls(name)()
            self.apps[name] = app

        # Lead and follow.
        for edge in self.system.edges:
            leader = self.apps[edge[0]]
```

```
            follower = self.apps[edge[1]]
            assert isinstance(leader, Leader)
            assert isinstance(follower, Follower)
            follower.follow(
                leader.__class__.__name__, leader.log
            )
            thread = self.threads[edge[1]]
            leader.lead(thread)

    def stop(self):
        self.is_stopping.set()
        for thread in self.threads.values():
            thread.is_prompted.set()
            thread.join()

    def get(self, cls: Type[A]) -> A:
        app = self.apps[cls.__name__]
        assert isinstance(app, cls)
        return app
```

The `MultiThreadedRunner` depends on the `RunnerThread` class, which is implemented by inheriting from the `Promptable` class and also the Python threading module's `Thread` class.

The `RunnerThread` class implement the `receive_prompt()` method of the `Promptable` class by setting a Python threading `Event` object called `is_prompted`. It implements the `run()` method of the `Thread` class by waiting for the `is_prompted` event to be set. When `is_prompted` is set, it calls on the application to pull and process event notifications from the "leaders" from which it has received prompts. This serialises the prompts from multiple applications, and keeps the processing of the event notifications single-threaded and sequential within individual process applications.

```
class RunnerThread(Promptable, Thread):
    def __init__(
        self,
```

```
        app_class: Type[Follower],
        is_stopping: Event,
    ):
        super(RunnerThread, self).__init__()
        if not issubclass(app_class, Follower):
            raise TypeError(
                "Not a follower: %s" % app_class
            )
        self.app_class = app_class
        self.is_stopping = is_stopping
        self.is_prompted = Event()
        self.prompted_names: List[str] = []
        self.setDaemon(True)
        self.is_ready = Event()

    def run(self):
        try:
            self.app: Follower = self.app_class()
        except:
            self.is_stopping.set()
            raise
        self.is_ready.set()
        while True:
            self.is_prompted.wait()
            if self.is_stopping.is_set():
                return
            self.is_prompted.clear()
            while self.prompted_names:
                name = self.prompted_names.pop(0)
                self.app.pull_and_process(name)

    def receive_prompt(self, leader_name: str) -> None:
        self.prompted_names.append(leader_name)
        self.is_prompted.set()
```

Now let's run the system multi-threaded with "Plain Old Python Objects".

```
runner = MultiThreadedRunner(system)
runner.start()
```

As above, both the bank accounts and the email notifications applications will each have an empty event notification log.

```
notifications = runner.get(EmailNotifications)
section = notifications.log["1,10"]
assert len(section.items) == 0

accounts = runner.get(BankAccounts)
section = accounts.log["1,10"]
assert len(section.items) == 0
```

We can then open a new account.

```
accounts.open_account(
    full_name="Alice",
    email_address="alice@example.com",
)
```

Because the multi-threaded runner is asynchronous, the `open_account()` method may return before the `BankAccount.Opened` event has actually been processed into an `EmailNotification`. So we just need to wait for a moment before we can expect the notification log of the email notifications application to show that an email notification has been created.

```
sleep(0.01)
```

Then we can expect the log of the email notifications application to show an `EmailNotification` aggregate has been created.

```
section = notifications.log["1,10"]
assert len(section.items) == 1
```

So far, we have run the system with both a single-threaded runner and a multi-threaded runner using "Plain Old Python Objects" recorders.

Running with SQLite

Now let's run the system with SQLite recorders.

Firstly, we need to configure the environment so that applications use the SQLite infrastructure factory. The environment variable `INFRASTRUCTURE_FACTORY_TOPIC` is used for this purpose. This environment variable is set to be the topic of the infrastructure factory class. We can use the `get_topic()` function introduced in Chapter 2 to generate the topic of the `SQLiteInfrastructureFactory`.

```
key = "INFRASTRUCTURE_FACTORY_TOPIC"
topic = get_topic(SQLiteInfrastructureFactory)
os.environ[key] = topic
```

Because we are using the SQLite infrastructure factory, we need to set the database URI in the environment too. The `SQLiteInfrastructureFactory` looks for the environment variable `DB_URI`. This variable is set to `:memory:` so that the runner's applications will each use their own in-memory SQLite database.

```
os.environ['DB_URI'] = ":memory:"
os.environ['CREATE_TABLE'] = "yes"
```

Now when we run the single-threaded runner, the applications will store events in an in-memory SQLite database. Given the system is running, when we open a new account, then an email notification will be immediately created.

```
# Given the system is running.
runner = SingleThreadedRunner(system)
runner.start()
```

```
notifications = runner.get(EmailNotifications)
section = notifications.log["1,10"]
assert len(section.items) == 0

# When we open a new account.
accounts = runner.get(BankAccounts)
accounts.open_account(
    full_name="Alice",
    email_address="alice@example.com",
)

# Then an email notification will be created.
section = notifications.log["1,10"]
assert len(section.items) == 1

# Clean up environment.
del(os.environ["DB_URI"])
del(os.environ["CREATE_TABLE"])
```

Now let's run multi-threaded with SQLite.

```
runner = MultiThreadedRunner(system)
```

This time we need some temporary files so that whilst different applications can have different databases, the different threads can also access the other databases. The function `tempfile_urls` is a generator that generates named temporary files and keeps them from being deleted so long as the generator exists.

```
def tmpfile_uris():
    tmp_files = []
    while True:
        tmp_file = NamedTemporaryFile(
            prefix="/Volumes/RAM DISK/"
        )
        tmp_files.append(tmp_file)
        yield "file:" + tmp_file.name
```

So then we can generate some temporary files and set application-specific database URIs in the environment.

```
uris = tmpfile_uris()
os.environ["BANKACCOUNTS_DB_URI"] = next(uris)
os.environ["EMAILNOTIFICATIONS_DB_URI"] = next(uris)
os.environ['CREATE_TABLE'] = "yes"
```

When an infrastructure factory looks for an environment variable, it firstly looks for an application-specific variable, and then looks for the non-application-specific value.

Now we can run the multi-threaded runner with SQLite.

```
runner.start()
```

At first, the email notifications application has an empty notification log

```
notifications = runner.get(EmailNotifications)

section = notifications.log["1,10"]
assert len(section.items) == 0
```

Let's open a new account.

```
accounts = runner.get(BankAccounts)

accounts.open_account(
    full_name="Alice",
    email_address="alice@example.com",
)
```

As above, because the multi-threaded runner is asynchronous, the `open_account()` method will return before the `BankAccount.Opened` event has been processed into an `EmailNotification`. So, again, we just need to

wait for a moment before we can expect the notification log of the email notifications application to show that an email notification has been created.

```
sleep(0.01)
```

Then we can expect the log of the email notifications application to show an email notification has been created.

```
section = notifications.log["1,10"]
assert len(section.items) == 1
```

The `tmpfile_uris` function uses a RAM disk, but if the SQLite database files were on a normal disk, this would function as a persistent system.

```
# Clean up environment.
del os.environ["BANKACCOUNTS_DB_URI"]
del os.environ["EMAILNOTIFICATIONS_DB_URI"]
del os.environ['CREATE_TABLE']
del os.environ['INFRASTRUCTURE_FACTORY_TOPIC']
```

Running with PostgreSQL

Now let's run the system with PostgreSQL recorders.

First, we need to set the environment variable for infrastructure factory topic.

```
key = "INFRASTRUCTURE_FACTORY_TOPIC"
topic = get_topic(PostgresInfrastructureFactory)
os.environ[key] = topic
```

We also need to set the environment variables that the PostgreSQL infrastructure factory will look for.

```
os.environ['CREATE_TABLE'] = "yes"
```

Now when we run the single-threaded runner, the applications will store events in a PostgreSQL database. As before, given the system is running, when we open a new account, then an email notification will be immediately created.

```
# Given the system is running.
runner = SingleThreadedRunner(system)
runner.start()

notifications = runner.get(EmailNotifications)
section = notifications.log["1,10"]
assert len(section.items) == 0, section.items

# When we open a new account.
accounts = runner.get(BankAccounts)
accounts.open_account(
    full_name="Alice",
    email_address="alice@example.com",
)

# Then an email notification will be created.
section = notifications.log["1,10"]
assert len(section.items) == 1, section.items

# Clean up environment.
del os.environ["CREATE_TABLE"]
del os.environ["INFRASTRUCTURE_FACTORY_TOPIC"]
```

Further extensions

The techniques above can be extended to use different modes of running systems of applications and different databases. For example, recorders can be developed that use for example MySQL or DynamoDB or EventStoreDB or AxonDB. And system runners can be developed that use the Python multithreading module or HTTP or gRPC.

Due to the fact that the system behaviour has been defined independently of particular database infrastructure or mode of running, a system of applications that is developed using “Plain Old Python Objects” recorders and a single-threaded runner will result in the same state as a system that is run against production databases in a distributed multi-operating system mode.

APPLICATIONS

Bank Accounts

This example extends the bank account examples that were used in the previous chapters of this book to illustrate the patterns.

As well as adding a few more methods and defining a few more events, this example shows how to save the events of more than one aggregate atomically. In order to atomically transfer an amount from one account to another, the amount needs to be debited from the former and credited to the later. Clearly, if any of the accounts is in a state where the transfer cannot be performed, then neither account should be affected. For example, if there are insufficient funds available on the account to be debited, or if the account to be credited is closed, then the transfer cannot be performed. Making the transfer between accounts atomic, by recording the events from both aggregates atomically, makes sure that either the transfer is completed successfully, or none of it happens at all.

Test case

Here is a test case that describes how an interface might use the application. It uses the `TestCase` class from the `unittest` package in the Python Standard Library.

```
class TestBankAccounts(unittest.TestCase):
    def test(self):
        app = BankAccounts()

        # Check account not found error.
        with self.assertRaises(AccountNotFoundError):
            app.get_balance(uuid4())

        # Create an account.
        account_id1 = app.open_account(
            full_name="Alice",
```

```
        email_address="alice@example.com",
    )

    # Check balance.
    self.assertEqual(
        app.get_balance(account_id1), Decimal("0.00")
    )

    # Deposit funds.
    app.deposit_funds(
        credit_account_id=account_id1,
        amount=Decimal("200.00"),
    )

    # Check balance.
    self.assertEqual(
        app.get_balance(account_id1), Decimal("200.00")
    )

    # Withdraw funds.
    app.withdraw_funds(
        debit_account_id=account_id1,
        amount=Decimal("50.00"),
    )

    # Check balance.
    self.assertEqual(
        app.get_balance(account_id1), Decimal("150.00")
    )

    # Fail to withdraw funds - insufficient funds.
    with self.assertRaises(InsufficientFundsError):
        app.withdraw_funds(
            debit_account_id=account_id1,
            amount=Decimal("151.00"),
        )
```

```
        # Check balance - should be unchanged.
        self.assertEqual(
            app.get_balance(account_id1), Decimal("150.00")
        )

        # Create another account.
        account_id2 = app.open_account(
            full_name="Bob",
            email_address="bob@example.com",
        )

        # Transfer funds.
        app.transfer_funds(
            debit_account_id=account_id1,
            credit_account_id=account_id2,
            amount=Decimal("100.00"),
        )

        # Check balances.
        self.assertEqual(
            app.get_balance(account_id1), Decimal("50.00")
        )
        self.assertEqual(
            app.get_balance(account_id2), Decimal("100.00")
        )

        # Fail to transfer funds - insufficient funds.
        with self.assertRaises(InsufficientFundsError):
            app.transfer_funds(
                debit_account_id=account_id1,
                credit_account_id=account_id2,
                amount=Decimal("1000.00"),
            )

        # Check balances - should be unchanged.
        self.assertEqual(
            app.get_balance(account_id1), Decimal("50.00")
```

```
)
self.assertEqual(
    app.get_balance(account_id2), Decimal("100.00")
)

# Close account.
app.close_account(account_id1)

# Fail to transfer funds - account closed.
with self.assertRaises(AccountClosedError):
    app.transfer_funds(
        debit_account_id=account_id1,
        credit_account_id=account_id2,
        amount=Decimal("50.00"),
    )

# Fail to transfer funds - account closed.
with self.assertRaises(AccountClosedError):
    app.transfer_funds(
        debit_account_id=account_id2,
        credit_account_id=account_id1,
        amount=Decimal("50.00"),
    )

# Fail to withdraw funds - account closed.
with self.assertRaises(AccountClosedError):
    app.withdraw_funds(
        debit_account_id=account_id1,
        amount=Decimal("1.00"),
    )

# Fail to deposit funds - account closed.
with self.assertRaises(AccountClosedError):
    app.deposit_funds(
        credit_account_id=account_id1,
        amount=Decimal("1000.00"),
    )
```

```
# Check balance - should be unchanged.
self.assertEqual(
    app.get_balance(account_id1), Decimal("50.00")
)

# Check overdraft limit.
self.assertEqual(
    app.get_overdraft_limit(account_id2),
    Decimal("0.00"),
)

# Set overdraft limit.
app.set_overdraft_limit(
    account_id=account_id2,
    overdraft_limit=Decimal("500.00"),
)

# Can't set negative overdraft limit.
with self.assertRaises(AssertionError):
    app.set_overdraft_limit(
        account_id=account_id2,
        overdraft_limit=Decimal("-500.00"),
    )

# Check overdraft limit.
self.assertEqual(
    app.get_overdraft_limit(account_id2),
    Decimal("500.00"),
)

# Withdraw funds.
app.withdraw_funds(
    debit_account_id=account_id2,
    amount=Decimal("500.00"),
)
```

```
        # Check balance - should be overdrawn.
        self.assertEqual(
            app.get_balance(account_id2),
            Decimal("-400.00"),
        )

        # Fail to withdraw funds - insufficient funds.
        with self.assertRaises(InsufficientFundsError):
            app.withdraw_funds(
                debit_account_id=account_id2,
                amount=Decimal("101.00"),
            )

        # Fail to set overdraft limit - account closed.
        with self.assertRaises(AccountClosedError):
            app.set_overdraft_limit(
                account_id=account_id1,
                overdraft_limit=Decimal("500.00"),
            )
```

Clearly, the test will not pass, unless the application is defined.

Application

The application supports an interface and depends on an event-sourced domain model.

```
class BankAccounts(Application):
    def open_account(
        self, full_name, email_address
    ) -> UUID:
        account = BankAccount.open(
            full_name=full_name,
            email_address=email_address,
        )
        self.save(account)
```

```
        return account.id

    def get_account(self, account_id: UUID) -> BankAccount:
        try:
            aggregate = self.repository.get(account_id)
        except AggregateNotFoundError:
            raise AccountNotFoundError(account_id)
        else:
            return aggregate

    def get_balance(self, account_id: UUID) -> Decimal:
        account = self.get_account(account_id)
        return account.balance

    def deposit_funds(
        self, credit_account_id: UUID, amount: Decimal
    ) -> None:
        account = self.get_account(credit_account_id)
        account.append_transaction(amount)
        self.save(account)

    def withdraw_funds(
        self, debit_account_id: UUID, amount: Decimal
    ) -> None:
        account = self.get_account(debit_account_id)
        account.append_transaction(-amount)
        self.save(account)

    def transfer_funds(
        self,
        debit_account_id: UUID,
        credit_account_id: UUID,
        amount: Decimal,
    ) -> None:
        debit_account = self.get_account(debit_account_id)
        credit_account = self.get_account(
            credit_account_id
```

```
        )
        debit_account.append_transaction(-amount)
        credit_account.append_transaction(amount)
        self.save(debit_account, credit_account)

    def set_overdraft_limit(
        self, account_id: UUID, overdraft_limit: Decimal
    ) -> None:
        account = self.get_account(account_id)
        account.set_overdraft_limit(overdraft_limit)
        self.save(account)

    def get_overdraft_limit(
        self, account_id: UUID
    ) -> Decimal:
        account = self.get_account(account_id)
        return account.overdraft_limit

    def close_account(self, account_id: UUID) -> None:
        account = self.get_account(account_id)
        account.close()
        self.save(account)
```

Obviously, the application will not work until the domain model has been developed.

Domain model

The domain model is defined below.

```
class BankAccount(Aggregate):
    def __init__(
        self, full_name: str, email_address: str, **kwargs
    ):
        super().__init__(**kwargs)
        self.full_name = full_name
```

```
        self.email_address = email_address
        self.balance = Decimal("0.00")
        self.overdraft_limit = Decimal("0.00")
        self.is_closed = False

    @classmethod
    def open(
        cls, full_name: str, email_address: str
    ) -> "BankAccount":
        return super()._create_(
            cls.Opened,
            full_name=full_name,
            email_address=email_address,
        )

    class Opened(Aggregate.Created):
        full_name: str
        email_address: str

    def append_transaction(
        self, amount: Decimal, transaction_id: UUID = None
    ) -> None:
        self.check_account_is_not_closed()
        self.check_has_sufficient_funds(amount)
        self._trigger_(
            self.TransactionAppended,
            amount=amount,
            transaction_id=transaction_id,
        )

    def check_account_is_not_closed(self) -> None:
        if self.is_closed:
            raise AccountClosedError(
                {"account_id": self.id}
            )

    def check_has_sufficient_funds(
```

```
    self, amount: Decimal
) -> None:
    if self.balance + amount < -self.overdraft_limit:
        raise InsufficientFundsError(
            {"account_id": self.id}
        )

class TransactionAppended(Aggregate.Event):
    amount: Decimal
    transaction_id: UUID

    def apply(self, obj: "BankAccount") -> None:
        obj.balance += self.amount

def set_overdraft_limit(
    self, overdraft_limit: Decimal
) -> None:
    assert overdraft_limit > Decimal("0.00")
    self.check_account_is_not_closed()
    self._trigger_(
        self.OverdraftLimitSet,
        overdraft_limit=overdraft_limit,
    )

class OverdraftLimitSet(Aggregate.Event):
    overdraft_limit: Decimal

    def apply(self, obj: "BankAccount") -> None:
        obj.overdraft_limit = self.overdraft_limit

def close(self):
    self._trigger_(self.Closed)

class Closed(Aggregate.Event):
    def apply(self, obj: "BankAccount") -> None:
        obj.is_closed = True
```

Run the test

Normally a suite of tests would be discovered and run from the command line (see the `unittest` documentation for details). But here we can define a little function `run()` to discover and run the test case programmatically.

```
import unittest

def run(test_case_class):
    suite = unittest.TestSuite()
    for name in dir(test_case_class):
        if name.startswith('test'):
            suite.addTest(test_case_class(name))
    runner = unittest.TextTestRunner()
    result = runner.run(suite)
    assert result.wasSuccessful()
```

The test case class can be run using the `run()` function defined above.

```
run(TestBankAccounts)
```

The output of the test runner is shown below.

```
.
----------------------------------------------------------------------
Ran 1 test in 0.020s

OK
```

Cargo Shipping

This example follows the definitive "Cargo Shipping" example in Eric Evans' book *Domain-Driven Design*. The example is the book was extended in a collaboration between Eric Evans' company and a Swedish software development company called Citerus.

The "DDD Sample" project website is available at `http://dddsample.sourceforge.net/` and the "DDD Sample" code is now available at `https://github.com/citerus/dddsample-core`. As it says on the project website,

> *"One of the most requested aids to coming up to speed on DDD has been a running example application. Starting from a simple set of functions and a model based on the cargo example used in Eric Evans' book, we have built a running application with which to demonstrate a practical implementation of the building block patterns as well as illustrate the impact of aggregates and bounded contexts."*

The "DDD Sample" project was written in the Java programming language, and has been ported to Python in the example below. In porting the "DDD Sample" project to an event-sourced application in Python, the distinctions between bounded contexts were discarded. Nevertheless, this example is worth studying for the rich quality of its ubiquitous language, as expressed partly with the custom value objects, the names of the command and query methods in the booking application and cargo aggregate.

This example is a good example of the use of custom value object types. They require custom transcoding to be used in the mapping of domain model event objects to the database. This example also shows how an interface object can be used to distance a concrete interface (e.g. a Web interface) from the custom value object types that give meaning to the application and domain layers.

Test case

Here is a test case that describes how the application might be used.

```
class TestCargoShipping(unittest.TestCase):
    def setUp(self) -> None:
        self.client = LocalClient(BookingApplication())

    def test_admin_can_book_new_cargo(self) -> None:
        arrival_deadline = datetime.now() + timedelta(
            weeks=3
        )

        cargo_id = self.client.book_new_cargo(
            origin="NLRTM",
            destination="USDAL",
            arrival_deadline=arrival_deadline,
        )

        cargo_details = self.client.get_cargo_details(
            cargo_id
        )
        self.assertTrue(cargo_details["id"])
        self.assertEqual(cargo_details["origin"], "NLRTM")
        self.assertEqual(
            cargo_details["destination"], "USDAL"
        )

        self.client.change_destination(
            cargo_id, destination="AUMEL"
        )
        cargo_details = self.client.get_cargo_details(
            cargo_id
        )
        self.assertEqual(
            cargo_details["destination"], "AUMEL"
        )
```

```
        self.assertEqual(
            cargo_details["arrival_deadline"],
            arrival_deadline,
        )

    def test_scenario_cargo_from_hongkong_to_stockholm(
        self,
    ) -> None:
        # Test setup: A cargo should be shipped from
        # Hongkong to Stockholm, and it should arrive
        # in no more than two weeks.
        origin = "HONGKONG"
        destination = "STOCKHOLM"
        arrival_deadline = datetime.now() + timedelta(
            weeks=2
        )

        # Use case 1: booking.

        # A new cargo is booked, and the unique tracking
        # id is assigned to the cargo.
        tracking_id = self.client.book_new_cargo(
            origin, destination, arrival_deadline
        )

        # The tracking id can be used to lookup the cargo
        # in the repository.
        # Important: The cargo, and thus the domain model,
        # is responsible for determining the status of the
        # cargo, whether it is on the right track or not
        # and so on. This is core domain logic. Tracking
        # the cargo basically amounts to presenting
        # information extracted from the cargo aggregate
        # in a suitable way.
        cargo_details = self.client.get_cargo_details(
            tracking_id
        )
```

```
self.assertEqual(
    cargo_details["transport_status"],
    "NOT_RECEIVED",
)
self.assertEqual(
    cargo_details["routing_status"], "NOT_ROUTED"
)
self.assertEqual(
    cargo_details["is_misdirected"], False
)
self.assertEqual(
    cargo_details["estimated_time_of_arrival"],
    None,
)
self.assertEqual(
    cargo_details["next_expected_activity"], None
)

# Use case 2: routing.
#
# A number of possible routes for this cargo is
# requested and may be presented to the customer
# in some way for him/her to choose from.
# Selection could be affected by things like price
# and time of delivery, but this test simply uses
# an arbitrary selection to mimic that process.
routes_details = (
    self.client.request_possible_routes_for_cargo(
        tracking_id
    )
)
route_details = select_preferred_itinerary(
    routes_details
)

# The cargo is then assigned to the selected
# route, described by an itinerary.
```

```
self.client.assign_route(
    tracking_id, route_details
)

cargo_details = self.client.get_cargo_details(
    tracking_id
)
self.assertEqual(
    cargo_details["transport_status"],
    "NOT_RECEIVED",
)
self.assertEqual(
    cargo_details["routing_status"], "ROUTED"
)
self.assertEqual(
    cargo_details["is_misdirected"], False
)
self.assertTrue(
    cargo_details["estimated_time_of_arrival"]
)
self.assertEqual(
    cargo_details["next_expected_activity"],
    ("RECEIVE", "HONGKONG"),
)

# Use case 3: handling

# A handling event registration attempt will be
# formed from parsing the data coming in as a
# handling report either via the web service
# interface or as an uploaded CSV file. The
# handling event factory tries to create a
# HandlingEvent from the attempt, and if the
# factory decides that this is a plausible
# handling event, it is stored. If the attempt
# is invalid, for example if no cargo exists for
# the specified tracking id, the attempt is
```

```
# rejected.
#
# Handling begins: cargo is received in Hongkong.
self.client.register_handling_event(
    tracking_id, None, "HONGKONG", "RECEIVE"
)
cargo_details = self.client.get_cargo_details(
    tracking_id
)
self.assertEqual(
    cargo_details["transport_status"], "IN_PORT"
)
self.assertEqual(
    cargo_details["last_known_location"],
    "HONGKONG",
)
self.assertEqual(
    cargo_details["next_expected_activity"],
    ("LOAD", "HONGKONG", "V1"),
)

# Load onto voyage V1.
self.client.register_handling_event(
    tracking_id, "V1", "HONGKONG", "LOAD"
)
cargo_details = self.client.get_cargo_details(
    tracking_id
)
self.assertEqual(
    cargo_details["current_voyage_number"], "V1"
)
self.assertEqual(
    cargo_details["last_known_location"],
    "HONGKONG",
)
self.assertEqual(
    cargo_details["transport_status"],
```

```
        "ONBOARD_CARRIER",
    )
    self.assertEqual(
        cargo_details["next_expected_activity"],
        ("UNLOAD", "NEWYORK", "V1"),
    )

    # Incorrectly unload in Tokyo.
    self.client.register_handling_event(
        tracking_id, "V1", "TOKYO", "UNLOAD"
    )
    cargo_details = self.client.get_cargo_details(
        tracking_id
    )
    self.assertEqual(
        cargo_details["current_voyage_number"], None
    )
    self.assertEqual(
        cargo_details["last_known_location"], "TOKYO"
    )
    self.assertEqual(
        cargo_details["transport_status"], "IN_PORT"
    )
    self.assertEqual(
        cargo_details["is_misdirected"], True
    )
    self.assertEqual(
        cargo_details["next_expected_activity"], None
    )

    # Reroute.
    routes_details = (
        self.client.request_possible_routes_for_cargo(
            tracking_id
        )
    )
    route_details = select_preferred_itinerary(
```

```
        routes_details
    )
    self.client.assign_route(
        tracking_id, route_details
    )

    # Load in Tokyo.
    self.client.register_handling_event(
        tracking_id, "V3", "TOKYO", "LOAD"
    )
    cargo_details = self.client.get_cargo_details(
        tracking_id
    )
    self.assertEqual(
        cargo_details["current_voyage_number"], "V3"
    )
    self.assertEqual(
        cargo_details["last_known_location"], "TOKYO"
    )
    self.assertEqual(
        cargo_details["transport_status"],
        "ONBOARD_CARRIER",
    )
    self.assertEqual(
        cargo_details["is_misdirected"], False
    )
    self.assertEqual(
        cargo_details["next_expected_activity"],
        ("UNLOAD", "HAMBURG", "V3"),
    )

    # Unload in Hamburg.
    self.client.register_handling_event(
        tracking_id, "V3", "HAMBURG", "UNLOAD"
    )
    cargo_details = self.client.get_cargo_details(
        tracking_id
```

```
)
self.assertEqual(
    cargo_details["current_voyage_number"], None
)
self.assertEqual(
    cargo_details["last_known_location"], "HAMBURG"
)
self.assertEqual(
    cargo_details["transport_status"], "IN_PORT"
)
self.assertEqual(
    cargo_details["is_misdirected"], False
)
self.assertEqual(
    cargo_details["next_expected_activity"],
    ("LOAD", "HAMBURG", "V4"),
)

# Load in Hamburg
self.client.register_handling_event(
    tracking_id, "V4", "HAMBURG", "LOAD"
)
cargo_details = self.client.get_cargo_details(
    tracking_id
)
self.assertEqual(
    cargo_details["current_voyage_number"], "V4"
)
self.assertEqual(
    cargo_details["last_known_location"], "HAMBURG"
)
self.assertEqual(
    cargo_details["transport_status"],
    "ONBOARD_CARRIER",
)
self.assertEqual(
    cargo_details["is_misdirected"], False
```

```
        )
        self.assertEqual(
            cargo_details["next_expected_activity"],
            ("UNLOAD", "STOCKHOLM", "V4"),
        )

        # Unload in Stockholm
        self.client.register_handling_event(
            tracking_id, "V4", "STOCKHOLM", "UNLOAD"
        )
        cargo_details = self.client.get_cargo_details(
            tracking_id
        )
        self.assertEqual(
            cargo_details["current_voyage_number"], None
        )
        self.assertEqual(
            cargo_details["last_known_location"],
            "STOCKHOLM",
        )
        self.assertEqual(
            cargo_details["transport_status"], "IN_PORT"
        )
        self.assertEqual(
            cargo_details["is_misdirected"], False
        )
        self.assertEqual(
            cargo_details["next_expected_activity"],
            ("CLAIM", "STOCKHOLM"),
        )

        # Finally, cargo is claimed in Stockholm.
        self.client.register_handling_event(
            tracking_id, None, "STOCKHOLM", "CLAIM"
        )
        cargo_details = self.client.get_cargo_details(
            tracking_id
```

```
        )
        self.assertEqual(
            cargo_details["current_voyage_number"], None
        )
        self.assertEqual(
            cargo_details["last_known_location"],
            "STOCKHOLM",
        )
        self.assertEqual(
            cargo_details["transport_status"], "CLAIMED"
        )
        self.assertEqual(
            cargo_details["is_misdirected"], False
        )
        self.assertEqual(
            cargo_details["next_expected_activity"], None
        )
```

Interface

The test case interacts with an interface object, which is defined below. The interface object uses the application methods, which are expressed in terms of custom value objects. The actual domain model aggregates are encapsulated by the application methods, so the interface object doesn't deal with those. Basically, the interface translates between the custom value object types and plain old Python objects which would be suitable for serialising in HTTP or JSON or XML.

```
class LocalClient(object):
    def __init__(self, app: BookingApplication):
        self.app = app

    def book_new_cargo(
        self,
        origin: str,
        destination: str,
```

```
    arrival_deadline: datetime,
) -> str:
    tracking_id = self.app.book_new_cargo(
        Location[origin],
        Location[destination],
        arrival_deadline,
    )
    return str(tracking_id)

def get_cargo_details(
    self, tracking_id: str
) -> CargoDetails:
    cargo = self.app.get_cargo(UUID(tracking_id))

    # Present 'next_expected_activity'.
    next_expected_activity: Optional[
        Union[Tuple[Any, Any], Tuple[Any, Any, Any]]
    ]
    if cargo.next_expected_activity is None:
        next_expected_activity = None
    elif len(cargo.next_expected_activity) == 2:
        next_expected_activity = (
            cargo.next_expected_activity[0].value,
            cargo.next_expected_activity[1].value,
        )
    elif len(cargo.next_expected_activity) == 3:
        next_expected_activity = (
            cargo.next_expected_activity[0].value,
            cargo.next_expected_activity[1].value,
            cargo.next_expected_activity[2],
        )
    else:
        raise Exception(
            "Invalid next expected activity: {}".format(
                cargo.next_expected_activity
            )
        )
```

```
        # Present 'last_known_location'.
        if cargo.last_known_location is None:
            last_known_location = None
        else:
            last_known_location = (
                cargo.last_known_location.value
            )

        # Present the cargo details.
        return {
            "id": str(cargo.id),
            "origin": cargo.origin.value,
            "destination": cargo.destination.value,
            "arrival_deadline": cargo.arrival_deadline,
            "transport_status": cargo.transport_status,
            "routing_status": cargo.routing_status,
            "is_misdirected": cargo.is_misdirected,
            "estimated_time_of_arrival":
cargo.estimated_time_of_arrival,
            "next_expected_activity": next_expected_activity,
            "last_known_location": last_known_location,
            "current_voyage_number":
cargo.current_voyage_number,
        }

    def change_destination(
        self, tracking_id: str, destination: str
    ) -> None:
        self.app.change_destination(
            UUID(tracking_id), Location[destination]
        )

    def request_possible_routes_for_cargo(
        self, tracking_id: str
    ) -> List[dict]:
        routes = (
```

```
            self.app.request_possible_routes_for_cargo(
                UUID(tracking_id)
            )
        )
        return [
            self.dict_from_itinerary(route)
            for route in routes
        ]

    def dict_from_itinerary(
        self, itinerary: Itinerary
    ) -> ItineraryDetails:
        legs_details = []
        for leg in itinerary.legs:
            leg_details: LegDetails = {
                "origin": leg.origin,
                "destination": leg.destination,
                "voyage_number": leg.voyage_number,
            }
            legs_details.append(leg_details)
        route_details: ItineraryDetails = {
            "origin": itinerary.origin,
            "destination": itinerary.destination,
            "legs": legs_details,
        }
        return route_details

    def assign_route(
        self,
        tracking_id: str,
        route_details: ItineraryDetails,
    ) -> None:
        routes = (
            self.app.request_possible_routes_for_cargo(
                UUID(tracking_id)
            )
        )
```

```
        for route in routes:
            if route_details == self.dict_from_itinerary(
                route
            ):
                self.app.assign_route(
                    UUID(tracking_id), route
                )

    def register_handling_event(
        self,
        tracking_id: str,
        voyage_number: Optional[str],
        location: str,
        handling_activity: str,
    ) -> None:
        self.app.register_handling_event(
            UUID(tracking_id),
            voyage_number,
            Location[location],
            HandlingActivity[handling_activity],
        )


# Stub function that picks an itinerary from a list of possible
itineraries.
def select_preferred_itinerary(
    itineraries: List[ItineraryDetails],
) -> ItineraryDetails:
    return itineraries[0]
```

Application

The application class `BookingApplication` supports an interface and depends on an event-sourced domain model. Custom transcodings are defined for the custom value object types used in the domain model event objects.

```
class LocationAsName(Transcoding):
    type = Location
    name = "location"

    def encode(self, o: Location) -> str:
        return o.name

    def decode(self, d: str) -> Location:
        assert isinstance(d, str)
        return Location[d]


class LegAsDict(Transcoding):
    type = Leg
    name = "leg"

    def encode(self, o: Leg) -> dict:
        return o.__dict__

    def decode(self, d: dict) -> Leg:
        assert isinstance(d, dict)
        return Leg(**d)


class ItineraryAsDict(Transcoding):
    type = Itinerary
    name = "itinerary"

    def encode(self, o: Itinerary) -> dict:
        return o.__dict__

    def decode(self, d: dict) -> Itinerary:
        assert isinstance(d, dict)
        return Itinerary(**d)


class HandlingActivityAsName(Transcoding):
    type = HandlingActivity
    name = "handling_activity"
```

```
    def encode(self, o: HandlingActivity) -> str:
        return o.name

    def decode(self, d: str) -> HandlingActivity:
        assert isinstance(d, str)
        return HandlingActivity[d]
```

The `BookingApplication` class is implemented as an extension of the `Application` class introduced in Chapter 9. The custom transcodings are defined and registered with the transcoder, as described in Chapter 3. The booking application implements command and query methods that support the interface object defined above.

```
class BookingApplication(Application):
    def register_transcodings(
        self, transcoder: Transcoder
    ):
        super(
            BookingApplication, self
        ).register_transcodings(transcoder)
        transcoder.register(LocationAsName())
        transcoder.register(HandlingActivityAsName())
        transcoder.register(ItineraryAsDict())
        transcoder.register(LegAsDict())

    def book_new_cargo(
        self,
        origin: Location,
        destination: Location,
        arrival_deadline: datetime,
    ) -> UUID:
        cargo = Cargo.new_booking(
            origin, destination, arrival_deadline
        )
        self.save(cargo)
        return cargo.id
```

```
def change_destination(
    self, tracking_id: UUID, destination: Location
) -> None:
    cargo = self.get_cargo(tracking_id)
    cargo.change_destination(destination)
    self.save(cargo)

def request_possible_routes_for_cargo(
    self, tracking_id: UUID
) -> List[Itinerary]:
    cargo = self.get_cargo(tracking_id)
    from_location = (
        cargo.last_known_location or cargo.origin
    ).value
    to_location = cargo.destination.value
    try:
        possible_routes = REGISTERED_ROUTES[
            (from_location, to_location)
        ]
    except KeyError:
        raise Exception(
            "Can't find routes from {} to {}".format(
                from_location, to_location
            )
        )

    return possible_routes

def assign_route(
    self, tracking_id: UUID, itinerary: Itinerary
) -> None:
    cargo = self.get_cargo(tracking_id)
    cargo.assign_route(itinerary)
    self.save(cargo)

def register_handling_event(
```

```
        self,
        tracking_id: UUID,
        voyage_number: Optional[str],
        location: Location,
        handing_activity: HandlingActivity,
    ) -> None:
        cargo = self.get_cargo(tracking_id)
        cargo.register_handling_event(
            tracking_id,
            voyage_number,
            location,
            handing_activity,
        )
        self.save(cargo)

    def get_cargo(self, tracking_id: UUID) -> Cargo:
        cargo = self.repository.get(tracking_id)
        assert isinstance(cargo, Cargo)
        return cargo
```

Domain model

The domain model is defined below. The custom value object types are defined first (`Location`, `Leg`, `Itinerary`, and `HandlingActivity`). Custom types are defined, which are used in the type hinting. A collection of `REGISTERED_ROUTES` are defined, as a simplification for the purposes of the example. The `Cargo` aggregate is defined as an event-sourced aggregate using the `Aggregate` class introduced in Chapter 2.

```
class Location(Enum):
    HAMBURG = "HAMBURG"
    HONGKONG = "HONGKONG"
    NEWYORK = "NEWYORK"
    STOCKHOLM = "STOCKHOLM"
    TOKYO = "TOKYO"
```

```
    NLRTM = "NLRTM"
    USDAL = "USDAL"
    AUMEL = "AUMEL"


class Leg(object):
    def __init__(
        self,
        origin: str,
        destination: str,
        voyage_number: str,
    ):
        self.origin: str = origin
        self.destination: str = destination
        self.voyage_number: str = voyage_number


class Itinerary(object):
    def __init__(
        self,
        origin: str,
        destination: str,
        legs: Tuple[Leg, ...],
    ):
        self.origin = origin
        self.destination = destination
        self.legs = legs


class HandlingActivity(Enum):
    RECEIVE = "RECEIVE"
    LOAD = "LOAD"
    UNLOAD = "UNLOAD"
    CLAIM = "CLAIM"


# Custom static types.
CargoDetails = Dict[
    str, Optional[Union[str, bool, datetime, Tuple]]
]
```

```
LegDetails = Dict[str, str]

ItineraryDetails = Dict[str, Union[str, List[LegDetails]]]

NextExpectedActivity = Optional[
    Union[
        Tuple[HandlingActivity, Location],
        Tuple[HandlingActivity, Location, str],
    ]
]


# Some routes from one location to another.
REGISTERED_ROUTES = {
    ("HONGKONG", "STOCKHOLM"): [
        Itinerary(
            origin="HONGKONG",
            destination="STOCKHOLM",
            legs=(
                Leg(
                    origin="HONGKONG",
                    destination="NEWYORK",
                    voyage_number="V1",
                ),
                Leg(
                    origin="NEWYORK",
                    destination="STOCKHOLM",
                    voyage_number="V2",
                ),
            ),
        )
    ],
    ("TOKYO", "STOCKHOLM"): [
        Itinerary(
            origin="TOKYO",
            destination="STOCKHOLM",
```

```
            legs=(
                Leg(
                    origin="TOKYO",
                    destination="HAMBURG",
                    voyage_number="V3",
                ),
                Leg(
                    origin="HAMBURG",
                    destination="STOCKHOLM",
                    voyage_number="V4",
                ),
            ),
        )
    ],
}


class Cargo(Aggregate):
    class Event(Aggregate.Event):
        pass

    @classmethod
    def new_booking(
        cls,
        origin: Location,
        destination: Location,
        arrival_deadline: datetime,
    ) -> "Cargo":
        return cls._create_(
            event_class=Cargo.BookingStarted,
            origin=origin,
            destination=destination,
            arrival_deadline=arrival_deadline,
        )

    class BookingStarted(Aggregate.Created):
        origin: Location
        destination: Location
```

```
    arrival_deadline: datetime

def __init__(
    self,
    origin: Location,
    destination: Location,
    arrival_deadline: datetime,
    **kwargs: Any,
) -> None:
    super().__init__(**kwargs)
    self._origin: Location = origin
    self._destination: Location = destination
    self._arrival_deadline: datetime = arrival_deadline
    self._transport_status: str = "NOT_RECEIVED"
    self._routing_status: str = "NOT_ROUTED"
    self._is_misdirected: bool = False
    self._estimated_time_of_arrival: Optional[
        datetime
    ] = None
    self._next_expected_activity: NextExpectedActivity = (
        None
    )
    self._route: Optional[Itinerary] = None
    self._last_known_location: Optional[
        Location
    ] = None
    self._current_voyage_number: Optional[str] = None

@property
def origin(self) -> Location:
    return self._origin

@property
def destination(self) -> Location:
    return self._destination

@property
```

```
def arrival_deadline(self) -> datetime:
    return self._arrival_deadline

@property
def transport_status(self) -> str:
    return self._transport_status

@property
def routing_status(self) -> str:
    return self._routing_status

@property
def is_misdirected(self) -> bool:
    return self._is_misdirected

@property
def estimated_time_of_arrival(
    self,
) -> Optional[datetime]:
    return self._estimated_time_of_arrival

@property
def next_expected_activity(self) -> Optional[Tuple]:
    return self._next_expected_activity

@property
def route(self) -> Optional[Itinerary]:
    return self._route

@property
def last_known_location(self) -> Optional[Location]:
    return self._last_known_location

@property
def current_voyage_number(self) -> Optional[str]:
    return self._current_voyage_number
```

```
def change_destination(
    self, destination: Location
) -> None:
    self._trigger_(
        self.DestinationChanged,
        destination=destination,
    )

class DestinationChanged(Aggregate.Event):
    destination: Location

    def apply(self, obj: "Cargo") -> None:
        obj._destination = self.destination

def assign_route(self, itinerary: Itinerary) -> None:
    self._trigger_(self.RouteAssigned, route=itinerary)

class RouteAssigned(Event):
    route: Itinerary

    def apply(self, obj: "Cargo") -> None:
        obj._route = self.route
        obj._routing_status = "ROUTED"
        obj._estimated_time_of_arrival = (
            datetime.now() + timedelta(weeks=1)
        )
        obj._next_expected_activity = (
            HandlingActivity.RECEIVE,
            obj.origin,
        )
        obj._is_misdirected = False

def register_handling_event(
    self,
    tracking_id: UUID,
    voyage_number: Optional[str],
    location: Location,
```

```
    handling_activity: HandlingActivity,
) -> None:
    self._trigger_(
        self.HandlingEventRegistered,
        tracking_id=tracking_id,
        voyage_number=voyage_number,
        location=location,
        handling_activity=handling_activity,
    )

class HandlingEventRegistered(Event):
    tracking_id: UUID
    voyage_number: str
    location: Location
    handling_activity: str

    def apply(self, obj: "Cargo") -> None:
        assert obj.route is not None
        if (
            self.handling_activity
            == HandlingActivity.RECEIVE
        ):
            obj._transport_status = "IN_PORT"
            obj._last_known_location = self.location
            obj._next_expected_activity = (
                HandlingActivity.LOAD,
                self.location,
                obj.route.legs[0].voyage_number,
            )
        elif (
            self.handling_activity
            == HandlingActivity.LOAD
        ):
            obj._transport_status = "ONBOARD_CARRIER"
            obj._current_voyage_number = (
                self.voyage_number
            )
```

```
        for leg in obj.route.legs:
            if leg.origin == self.location.value:
                if (
                    leg.voyage_number
                    == self.voyage_number
                ):
                    obj._next_expected_activity = (
                        HandlingActivity.UNLOAD,
                        Location[leg.destination],
                        self.voyage_number,
                    )
                    break
        else:
            raise Exception(
                "Can't find leg with origin={} and "
                "voyage_number={}".format(
                    self.location,
                    self.voyage_number,
                )
            )

    elif (
        self.handling_activity
        == HandlingActivity.UNLOAD
    ):
        obj._current_voyage_number = None
        obj._last_known_location = self.location
        obj._transport_status = "IN_PORT"
        if self.location == obj.destination:
            obj._next_expected_activity = (
                HandlingActivity.CLAIM,
                self.location,
            )
        elif self.location.value in [
            leg.destination
            for leg in obj.route.legs
        ]:
```

```
            for i, leg in enumerate(
                obj.route.legs
            ):
                if (
                    leg.voyage_number
                    == self.voyage_number
                ):
                    next_leg: Leg = obj.route.legs[
                        i + 1
                    ]
                    assert (
                        Location[next_leg.origin]
                        == self.location
                    )
                    obj._next_expected_activity = (
                        HandlingActivity.LOAD,
                        self.location,
                        next_leg.voyage_number,
                    )
                    break
        else:
            obj._is_misdirected = True
            obj._next_expected_activity = None

    elif (
        self.handling_activity
        == HandlingActivity.CLAIM
    ):
        obj._next_expected_activity = None
        obj._transport_status = "CLAIMED"

    else:
        raise Exception(
            "Unsupported handling event: {}".format(
                self.handling_activity
            )
        )
```

Run the test

The test case class can be run using the `run()` function defined in the bank accounts application example.

```
run(TestCargoShipping)
```

The output of the test runner is shown below.

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.051s

OK
```

Epilogue

Thank you for reading this book. We have covered a lot of ground. The book was mainly about event-sourced models, applications and systems. But the question of how generally applicable is event sourcing brought us to Whitehead's process philosophic scheme: the idea that the actual world is built up from actual occasions of experience, with human occasions of experience being of an especially high grade. Understanding Whitehead's scheme made it very apparent that Christopher Alexander's pattern language scheme is a creative application of Whitehead's scheme: pattern language was intended to describe events. Alexander's patterns correspond to Whitehead's actual occasions, and Alexander's pattern language corresponds to Whitehead's structured living societies. Alexander's remark that the world is built up of events *and nothing else* echoes Whitehead's remark that everything that exists is derived by abstraction from actual occasions. Alexander's remark that pattern languages teach you to be receptive to what is real echoes Whitehead's view that there is no going behind actual occasions to find anything more real. At the centre of all of this are the actual occasions, which are both a creative process of feeling and a decision that brings many feelings into a unity of feeling; things that both come to exist and that can be explained and expressed in terms of their interconnections and decisions. The notion of an event is then understood as an interrelated set of actual occasions (a social nexus, or society) with the limiting type being a nexus that has only one member.

In adopting Whitehead's scheme, we respond to the question of how generally applicable is event sourcing by acquiring an understanding that all domains are inherently constituted by what happens in that domain. In adopting Alexander's scheme, we acquire a technique for describing and conditioning creative occasions of design. The patterns of event-sourced applications presented in the main chapters of this book are generally applicable for these two reasons: firstly we can always tackle the complexity in a domain by building supportive event-sourced applications; secondly, when we build event-sourced applications we can condition our occasions of design by following the patterns in this book.

The patterns in this book are intended to be used in a particular set of human occasions of experience: occasions of designing event-sourced software applications and event-driven systems. Software code can be exciting and but isn't in itself alive, and however fascinating can be tiring to work with and understand. We have been through quite a lot of software code in the previous fifteen chapters. And so I wanted to end this book on a warmer note, by returning to the human level, and saying something more about the application of Whitehead's scheme in the world of human personal experience and human relationships, by discussing more fully the psychology topics that were mentioned in the prologue: Carl Rogers' person-centred theory, and Marshall Rosenberg's nonviolent or collaborative communication.

Person-centred theory

Person-centred theory is the view that a human person exists moment-by-moment through their feelings. Each moment is made of the feelings that are experienced in that moment. We say 'moment' because the many feelings are brought together in a unity of feeling that makes the moment what it is. These moments correspond to the actual occasions of experience in Whitehead's scheme, and the patterns of recurring moments that we have in our lives can be described using Alexander's patten language scheme.

Whitehead considered that the Greeks were mistaken to abstract from the occasion of experiencing "grey stone" to thinking of the world as built up of instances of substance-quality categories, because these categories restrict thought, and that it is much better to abstract from the occasion of experience itself, and say that our human occasions of experience are of a high grade (characterised by "presentational immediacy" that is our capacity for consciousness), that there are lower grades of actual occasions that populate the broader societies of actual occasions from which we have arisen as human subjects, and that these lower (and indeed higher) grades are the atoms of which reality made up. From this broad cosmology, Carl Rogers brings consideration back to the human occasions of experience themselves. That is why his nineteen propositions are both highly

reminiscent of Whitehead's scheme, but also very interesting as a characterisation of the condition of being a human being.

The descriptions of a fully functioning person and of the nineteen propositions of person-centred theory below are adapted from the Person Centred Association.

Fully functioning person

In *On Becoming a Person*, published in 1951, Carl Rogers listed the characteristics of a fully functioning person.

1. A growing openness to experience – they move away from defensiveness and have no need for subception (a perceptual defense that involves unconsciously applying strategies to prevent a troubling stimulus from entering consciousness).

2. An increasingly existential lifestyle – living each moment fully – not distorting the moment to fit personality or self concept but allowing personality and self concept to emanate from the experience. This results in excitement, daring, adaptability, tolerance, spontaneity, and a lack of rigidity and suggests a foundation of trust. ("To open one's spirit to what is going on now, and discover in that present process whatever structure it appears to have.")

3. Increasing organismic trust – they trust their own judgment and their ability to choose behavior that is appropriate for each moment. They do not rely on existing codes and social norms but trust that as they are open to experiences they will be able to trust their own sense of right and wrong.

4. Freedom of choice – not being shackled by the restrictions that influence an incongruent individual, they are able to make a wider range of choices more fluently. They believe that they play a role in determining their own behavior and so feel responsible for their own behavior.

5. Creativity – it follows that they will feel more free to be creative. They will also be more creative in the way they adapt to their own circumstances

without feeling a need to conform.

6 Reliability and constructiveness – they can be trusted to act constructively. An individual who is open to all their needs will be able to maintain a balance between them. Even aggressive needs will be matched and balanced by intrinsic goodness in congruent individuals.

7 A rich full life – he describes the life of the fully functioning individual as rich, full and exciting and suggests that they experience joy and pain, love and heartbreak, fear and courage more intensely.

Nineteen propositions

Optimal development, referred to below in proposition 14, results in a certain process rather than static state. Rogers describes this as the good life, where the organism continually aims to fulfill its full potential.

The following is taken from Tony Merry's book *Learning and Being in Person-Centred Counselling*:

> *"Rogers offered a group of nineteen hypothetical statements which, together constitute his person-centred theory of personality dynamics and behaviour . 'A theory of personality and Behaviour' can be found in Rogers (1951, pp. 481-533). Rogers makes the following statement: 'This theory is basically phenomenological in character, and relies heavily on the concept of the self as an explanatory construct. It pictures the end-point of personality development as being a basic congruence between the phenomenal field of experience and the conceptual structure of the self - a situation which, if achieved, would represent freedom from internal strain and anxiety, and freedom from potential strain; which would represent the maximum in realistically oriented adaptation; which would mean the establishment of an individualised value system having considerable identity with the value system of any other equally well-adapted member of the human race.' (p. 532)"*

The nineteen propositions repay careful reading because together they provide us with an eloquent theory of personality which is entirely consistent with Rogers' theory of how people can change for the better, and why certain qualities of relationship are necessary in order to promote that change. Interspersed with Rogers' original wording Merry has added some explanations in different and more familiar terms.

1. Every individual exists in a continually changing world of experiencing of which they are the centre.

2. The organism reacts to the field as it is experienced and perceived. This perceptual field is, for the individual, reality. We see ourselves as the centre of our reality; that is, our ever-changing world around us. We experience ourselves as the centre of our world, and we can only "know" our own perceptions.

3. The organism reacts as an organised whole to this phenomenal field. The whole person works together rather than as separate parts.

4. The organism has one basic tendency and striving – to actualize, maintain, and enhance the experiencing organism. Human beings have a basic tendency to fulfil their potential, to be positive, forward looking, to grow, improve, and protect their existence.

5. Behaviour is basically the goal-directed attempt of the organism to satisfy its needs as experienced in the field as perceived. The things we do (our behaviour in everyday life) in order to satisfy our fundamental needs. If we accept proposition 4, that all needs are related, then all complex needs are related to basic needs. Needs are "as experienced" and the world is "as perceived".

6. Emotion accompanies and in general facilitates such goal-directed behaviour, the kind of emotion being related to the seeking versus the consummatory aspects of the behaviour, and the intensity of the emotion being related to the perceived significance of the behaviour for the maintenance and enhancement of the organism. Feelings are associated with, and help us to get, satisfaction and fulfilment. Generally speaking,

pleasant feelings arise when we are satisfied, unpleasant feelings when we are not satisfied. The more important the situation, the stronger the feelings.

7. The best vantage point from which to understand behaviour is from the internal frame of reference of the individual himself. To understand the behaviour of a person, we must look at the world from their point of view.

8. A portion of the total perceptual field gradually becomes differentiated as the self. Some of what we recognise as "reality", we come to call "me" or "self".

9. As a result of interaction with the environment, and particularly as a result of evaluational interaction with others, the structure of the self is formed – an organised, fluid, but consistent conceptual pattern of perceptions of characteristics and relationships of the "I" or the "me" together with values attached to these concepts.

10. The values attached to experiences, and the values which are part of the self structure, in some instances are values experienced directly by the organism, and in some instances are values introjected or taken over from others, but perceived in a distorted fashion, as if they had been experienced directly. As we go about our everyday life, we build up a picture of ourselves, called the self-concept, from relating to and being with others and by interacting with the world around us. Sometimes we believe other people's version of reality and we absorb them into our self-concept as though they were our own.

11. As experiences occur in the life of an individual, they are either (a) symbolized, perceived and organized into some relationship to the self, (b) ignored because there is no relationship to the self-structure, (c) denied symbolization or given a distorted symbolization because the experience is inconsistent with the structure of the self. There are several things we can do with our everyday experience: we can see that it is relevant to ourselves or we can ignore it because it is irrelevant; or if we experience something that doesn't fit with our picture of ourselves we can either pretend it didn't happen or change our picture of it, so that it does fit.

12. Most of the ways of behaving which are adopted by the organism are those which are consistent with the concept of the self. Most of the time we do things and live our lives in ways which are in keeping with our picture of ourselves.

13. Behaviour may, in some instances, be brought about by organic experiences and needs which have not been symbolised. Such behaviour may be inconsistent with the structure of the self, but in such instances the behaviour is not "owned" by the individual. Sometimes we do things as a result of experiences from inside us we have denied, or needs we have not acknowledged. This may conflict with the picture we have of ourselves, so we refuse to accept it is really us doing it.

14. Psychological maladjustment exists when the organism denies to awareness significant sensory and visceral experiences, which consequently are not symbolized and organised into the gestalt of the self-structure. When this situation exists, there is a basic or potential psychological tension. When we experience something that does not fit with our picture of ourselves and we cannot fit it in with that picture, we feel tense, anxious, frightened or confused.

15. Psychological adjustment exists when the concept of the self is such that all the sensory and visceral experiences of the organism are, or may be, assimilated on a symbolic level into a consistent relationship with the concept of the self. We feel relaxed and in control when the things we do and the experiences we have all fit in with the picture we have of ourselves.

16. Any experience which is inconsistent with the organization or structure of self may be perceived as a threat, and the more of these perceptions there are, the more rigidly the self-structure is organized to maintain itself. When things happen that don't fit with the picture we have of ourselves, we feel anxious. The more anxious we feel, the more stubbornly we hang on to the picture we have of ourselves as "real".

17. Under certain conditions, involving primarily complete absence of any threat to the self-structure, experiences which are inconsistent with it may be perceived, and examined, and the structure of self revised to assimilate and include such experiences. When we are in a relationship where we feel

safe, understood and accepted for who we are, we can look at some of the things that don't fit in with our picture of ourselves and, if necessary change our picture to fit our experience more accurately. Or we can accept the occasional differences between our pictures of ourselves and our experience without becoming anxious.

18. When the individual perceives and accepts into one consistent and integrated system all his sensory and visceral experiences, then they are necessarily more understanding of others and is more accepting of others as separate individuals. When we see ourselves more clearly and accept ourselves more for what we are than as how others would like us to be, we can understand that others are equal to us, sharing basic human qualities, yet distinct as individuals.

19. As the individual perceives and accepts into their self-structure more of their organic experiences, they finds that they are replacing their value system – based so largely upon introjections which have been distortedly symbolized – with a continuing organismic valuing process. We stop applying rigid rules to govern our values and use a more flexible way of valuing based upon our own experience, not on the values we have taken in from others.

Applications

Person-centred theory emerged from therapeutic psychology, of counselling people who we struggling with their feelings and in their lives. Carl Rogers tells the story of counselling a child whose mother seemed to be the cause of some of the difficulties which the child was having. When the mother asked Carl Rogers if he also did counselling with adults, he said he did, and spent time listening to her. He realised that it was her feelings and needs that he needed to pay attention to, and changed from diagnosis of personality types and curing personality disorders to take the view that what mattered was bringing out what is alive in a person, and providing the conditions in which they could feel and then address concerns that they were shutting out because they were too painful or otherwise too troubling, because by learning not to feel what was troubling they were becoming unable fully to feel themselves, unable to grow as a person, unable fully to

live each moment, and unable to fully become who they are, itself is a cause of pain and grief, unable to enjoy themselves. Person-centred theory is analysis of these concerns, and person-centred therapy is an application of this theory.

The aim of person-centred therapy is to create the necessary conditions to assist clients in their growth process, to live each moment fully, enabling a person to cope with current and future problems, to find satisfaction in being who they are, and to enjoy life. Rogers proposed that therapists must have three attributes to create a growth-promoting climate in which individuals can move forward and become capable of fully becoming who they are.

1. Congruence (genuineness or realness).

2. Unconditional positive regard (acceptance and caring).

3. Accurate empathic understanding (an ability to deeply grasp the subjective world of another person).

These conditions can be understood as the conditions by which a person can begin to feel what they have been disinclined or have become unable to feel, and thereby make the connections with themselves that they need to live their lives. These conditions simply make way for the "actualising tendency" each person has to maintain and enhance the "experiencing organism". This depends on innate creativity, being the cause of oneself, supported and informed self-direction, and a subjective moment-by-moment process of feeling and becoming. As we have seen, these are all features of Whitehead's scheme. They are all also recognisable aspects of a productive software development project.

A very popular application of person-centred theory is Marshall Rosenberg's nonviolent or collaborative communication. Marshall Rosenberg was a student of Carl Rosenberg and worked closely with him. Much of what Carl Rogers said is reflected in Marshall Rosenberg's approach, much of what Marshall Rosenberg describes can be recognised as things Carl Rogers said. Nonviolent communication is centred on the need we have to connect with ourselves and others, and to try together to meet

the needs we share. We are social creatures, and by trying empathise, to understand or feel what we and others are feeling, we can hope to understand what we and others need, and then have a chance of meeting those needs. Emotions motivate us to meet our needs, and so we need to pay attention to them. One important need is the need for empathy, and also self-empathy, because that's how we connect with the feelings we have, through which we can understand, or at least guess at, the needs we have as living social human creatures, and whether these needs are being met or not. But this isn't something we are necessarily good at understanding naturally, it isn't something we are necessarily born fully understanding, and certainly we can forget and in different ways be drawn away from understanding that we need empathy and we need to empathise. And so it can help us a great deal to live more fully and more happily together to have an approach to connecting with ourselves and others that goes beyond the initial applications of person-centred theory in counselling in therapeutic psychology, something that takes us further than the therapist-client relationship into everyday life and common human relations. Not only do we need to enjoy ourselves, we need to enjoy being with other people, and helping others enjoy themselves. Nonviolent or collaborative communication is designed to help with that.

Collaborative communication

Nonviolent or collaborative communication is an approach to personal human relations that considers all human action is an attempt to meet needs. The trouble is that often we don't always understand what needs we have as human beings, and unless we are careful we can express ourselves in a way that is self-defeating. One of the greatest joys we can have is contributing to meeting the needs of others.

The moment when we have the greatest need for empathy is also the moment when we are most at risk of expressing ourselves in a way that makes it unlikely others will feel inclined to contribute to meeting our needs. When a person feels judged or blamed or abused by us, they may feel disinclined from giving that person the empathy they need or indeed disinclined to contribute to meeting any of their needs. Nonviolent

communication can help us to express what we are feeling and needing in a way that maximises the chances that our needs will be met.

If "violent communication" is the "tragic expression of unmet needs", a style of communication that breeds conflict and sometimes real physical violence, nonviolent communication offers a way to connect with feelings and needs and to make present requests that are more likely to result in everybody's needs being met.

Gratitude

Marshall Rosenberg's book *Nonviolent Communication: A Language For Life* starts with an expression of gratitude to the reader for reading his book. It seems to me that explaining gratitude is perhaps the best way to start explaining nonviolent communication.

Gratitude in nonviolent communication is a particular form of expression that gives thanks to a particular person or group of people for a particular thing that they did to contribute to meeting our needs. The suggestion is to say in particular what you observed that somebody did or said that contributed to meeting those needs, to say what you felt when they did what they did, and to say what needs were met by what they did. Gratitude is given as a gift, not as a reward, and when giving any gift there is an implied request to accept the gift.

We can notice from this style of expressing gratitude the more general form of expression used throughout nonviolent communication: observations, feelings, needs, and requests. These four parts must be preceded by a genuine care, a genuine intention to meet needs, otherwise they are nothing.

Gratitude helps others to know what they did that you value, and which needs they contributed to meeting. Receiving this kind of gratitude helps to meet our need to understand how we actually contributed to meeting needs. As well as understanding, an expression of gratitude can also help meet the needs we all have for acceptance, appreciation, belonging, cooperation, communication, closeness, companionship, consideration, respect, support, to be seen, understanding, warmth, presence, awareness, celebration of life,

clarity, growth, and self-expression. Gratitude is also something we can be grateful for. Clearly, being grateful for gratitude is an endless cycle that will need to be stopped gracefully sooner or later!

Meeting needs makes life feel more wonderful. The good feelings that follow from meeting these needs by expressing gratitude helps replenish the energy that contributing to meeting needs, such as giving empathy, can consume. Marshall Rosenberg calls gratitude 'giraffe fuel', which brings us to the two characterisations used to teach nonviolent communication: the jackal and the giraffe.

Jackals and giraffes

In nonviolent communication, the jackal is the character makes and hears judgments and criticisms, to and from both itself and others. A jackal rewards people for doing what it wants, and punishes people for not doing what it wants and doing what it doesn't want. Jackals might also punish you for doing what it wants, just to keep you down. Jackals bite. Jackals make like feel miserable, including their own.

By contrast, the giraffe is the character with a big heart. The giraffe character actually cares, and listens through words for feeling and needs. The giraffe makes requests that are to be fulfilled only if it would bring joy and would be done without any sense of obligation. Giraffes make life wonderful, especially their own.

Each person can choose, from moment to moment, whether to be a jackal or a giraffe. It's easy to act as a jackal because that's the way we are often accustomed to behaving, that's the way we often see people behaving. It's conventionally easy to think that being a jackal is the way to be successful because it's the character that is often associated with success, for example on television or at work.

It's also easy to be a giraffe, but it can take time to remember when we have forgotten. It may also be that we never learned how to listen for feelings and needs. That's where nonviolent communication can help.

The problem with the jackal strategy is that being a jackal makes it less likely that everybody's needs will be met. The benefit the giraffe strategy is maximising the chances the everybody's needs will be met.

Strategies

Nonviolent communication takes the view that all human behaviour is an attempt to meet needs. But unless we know what each person is needing, it's hard to meet those needs. Unless we understand what each person is feeling, it's hard to know what we need. Even when we do understand what we need, it can be hard to find strategies to meet those needs. Just because you are hungry, doesn't mean you know how to cook. Similarly, just because you need to give and receive empathy doesn't mean that you know how to understand other people or how to make it more likely you will be understood by others.

Developing strategies takes time and energy. It takes time to learn how to cook nutritious food, to learn how to sleep well, and to learn how to give ourselves and others empathy. As each of us grows as a person, we learn things that we didn't know before. As we learn, we may look back and feel pained by things we did when we didn't know something that we know now. We may feel sadness or pain for what happened, and understand, that our actions caused real pain and perhaps even trauma. Mourning and grieving is a strategy for giving ourselves and others the empathy we need to understand and connect with aspects of our previous selves that we have transcended. For the things we have learned, we can also be grateful. We can also try to "clean up" by giving empathy to the people we have hurt, although we have to be prepared to accept their 'No' to our request to receive such a gift.

Observations

It's easy to generalise and make interpretations and judgements. But it's better to make observations by stating what actually happened. It's better to say "you arrived later than you said you would twice this week" than "you are always late".

Similarly, it's useful to remember that when we hear observations it can help to avoid hearing generalisations and judgements. If somebody says "you arrived later that you said" it's better to understand the feelings and needs of the other person rather than to hear "you are always late". Equally, if somebody does say "you are always late" when you simply arrived later than you promised, it's useful to look through the words to the feelings that are alive in the other person and the needs they have that weren't met.

Feelings and needs

When needs are met we feel good. Similarly, when needs are not met we feel bad. This can get complicated when some needs are met and others are not met.

Our feelings, our emotions, move us to meet our needs. If we didn't experience thirst when we needed to drink, then we wouldn't be motivated to find water. The trouble is that sometimes we can learn not to respond to our feelings in a way that leads to our needs being met. Sometimes we can learn to think in ways that lead us away from connecting with our feelings. And sometimes we can learn to respond to our feelings in ways that make it unlikely our actual needs will be met.

Nonviolent communication suggests that we all have the same universal needs. The needs we have are many, and we are as needy as the need we have that aren't being met. There is no shame in having needs. We are each as needy as the needs we have that aren't being met.

Nonviolent communication provides a vocabulary for needs and feelings. The lists of feelings are not intended to be definitive or exhaustive.

Needs in nonviolent communication are often categorised under the following headings: connection, honesty, play, peace, physical well-being, meaning, and autonomy.

Feelings are categorised in two sets, firstly into feeling you have when needs are not met, and secondly into feelings you have when needs are met. The feelings we have when our needs are not met are often categorised

under the following headings: afraid, annoyed, angry, aversion, confused, disconnected, disquiet, embarrassed, fatigue, pain, sad, tense, vulnerable, yearning. The feelings we have when our needs are met are often categorised under these headings: affectionate, engaged, confident, excited, grateful, inspired, joyful, exhilarated, peaceful, refreshed.

Requests

A request in nonviolent communication is always followed by the qualification that you do not want the person to do what you ask unless they will only feel joy. A request in nonviolent communication is detached from any form of reward or punishment.

The giving of gratitude is also a request to accept a gift. As with any request, we have to be prepared to accept the answer we are given.

Hearing ‘no’

It’s important when practising nonviolent communication to understand how to hear a No. When we make a request and receive a No in response, we can look to the needs that are being met by the other person saying No. Similarly, when one says No to a request, it can help the other person to say what needs we are meeting by saying No. Nonviolent communication is quite particular in suggesting that people only do things that bring them joy, and if somebody feels that they will enjoy life more by saying No to a request, then we can enjoy hearing their No by understanding the needs that are being met by this answer. But we also may need to find other people to help us meet our needs.

Conflicts

Nonviolent communication suggests that our needs are never in conflict, but it may be that our strategies are. For example, if one person wants to party whilst another wants to sleep, a conflict may arise. Similarly, if two people want to be heard and both insist on talking with the expectation that the other person be quiet and listens, a conflict will arise unless at least one

person is able to understand that the other needs to be heard and understood before they will be able to listen. At such a moment, it can help if both understand that they themselves need to be heard and understood, but that they don't necessarily need to be heard and understood right then. It can help to provide oneself with the self-empathy to understand that one needs to be heard. And being grateful for understanding this about oneself can be sufficient to provide the fuel to try to understand what the other person is feeling and needing, until they feel they have been heard and understood. When the other person has received gratitude for sharing what they wanted to be heard, they may feel inclined to meet ones need for empathy. A request can then be made for that person to say if they have any more that they would like to be heard, and then another request can be made to hear what one has to say. In a difficult or challenging situation, or when you are tired, it can take a little bit longer to understand what you are feeling and needing, and then it can be useful to say to the other person that you will return, but right now you need go away and do some work on yourself. This can avoid the feeling of needing to flee and absenting oneself in a way that leads to further conflict or breakdown in a relationship.

Nonviolent communication provides an approach by which conflicts can be resolved and avoided. The approach involves firstly one person and then the other explaining how they are feeling and what they need until the other person can explain it back to them in a way that brings the satisfaction of meeting their need to be heard and understood. From the position where each understands what the other person actual needs, requests can be made that meet those needs. There is no need for anybody to compromise by accepting that their needs won't be met.

Whether or not there is a conflict, it's always worth remembering to try to understand what is alive in the other person. Giving empathy can be hard work, and it's not always easy to understand how somebody is feeling and what they are needing in any given moment. But, as Marshall Rosenberg used to say, if it's worth doing, it's worth doing badly. One strategy is to try to guess once, and if you didn't guess well, then ask the other person to tell you how they are feeling and what they are needing, and perhaps offer to say what you heard, or better just be with them in their pain, or until they feel heard. You don't need to fix anything there and then. Just

understanding how somebody is feeling and what they are needing goes a long way.

Event-oriented development

As we have seen, the person-centred approach is a moment-by-moment approach, and nonviolent communication encourages "present requests" which are things that somebody can agree to there and then. However, as Whitehead wrote, "No actual entity can rise beyond what the actual world as a datum from its standpoint — its actual world — allows it to be."

Whereas we are conditioned by the conditions established in the past, development works to condition the conditions that will condition future actual occasions of experience. The conditions are actual occasions of the past. The process of any actual occasion is conditioned by its subjective aim. A subjective aim is a proposition prehended with the subjective form of purpose. And so by adjusting the proposition and taking them up as a purpose, we can gradually and eventually make things better for ourselves and others.

Why for others, and not just ourselves? Why for ourselves, and not just for others? The answer can be found in understanding the purpose of a life as reaching for a greater intensity of feeling and depth of satisfaction. Whitehead wrote:

> *"Morality of outlook is inseparably conjoined with generality of outlook. The antithesis between the general good and the individual interest can be abolished only when the individual is such that its interest is the general good, thus exemplifying the loss of the minor intensities in order to find them again with finer composition in a wider sweep of interest."*

Development is a matter of describing propositions that others feel able to agree with, so they appropriate them as a purpose. And so development will necessarily involve fashioning a description of what we might do together, proposing such descriptions as potential agreements, seeking objections and improvements to the description, reaching agreement about what will be

done, undertaking what was agreed, and then checking that what was agreed to be undertaken was accomplished. We can see that sometimes people may not have the capability to do particular things, but, with help and assistance and support, people can learn how to do new things. Agreements can be formed to coach and train people in particular activities, and when capabilities exist, these abilities can be used to produce desirable outcomes. The how-to genre is already extensive. There are already many approaches to the development of software and organisations and mental health and relationships and environmental responsibility. What has been missing, it seems to me, is a decisive unity of feeling that the world is built up of events.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y