Event Sourcing

Capture all changes to an application state as a sequence of events.

Martin Fowler

12 December 2005

This is part of the Further Enterprise Application Architecture development writing that I was doing in the mid 2000's. Sadly too many other things have claimed my attention since, so I haven't had time to work on them further, nor do I see much time in the foreseeable future. As such this material is very much in draft form and I won't be doing any corrections or updates until I'm able to find time to work on it again.

We can query an application's state to find out the current state of the world, and this answers many questions. However there are times when we don't just want to see where we are, we also want to know how we got there.

Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes.

How it Works

The fundamental idea of Event Sourcing is that of ensuring every change to the state of an application is captured in an event object, and that these event objects are themselves stored in the sequence they were applied for the same lifetime as the application state itself.

Let's consider a simple example to do with shipping notifications. In this example we have many ships on the high seas, and we need to know where they are. A simple way to do this is to have a tracking application with methods to allow us to tell when a ship arrives or leaves at a port.

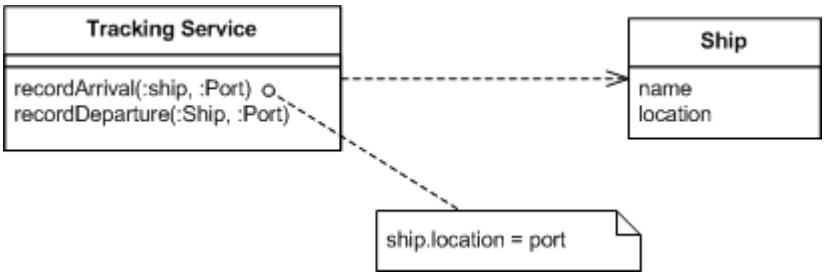

Figure 1: A simple interface for tracking shipping movements.

In this case when the service is called, it finds the relevant ship and updates its location. The ship objects record the current known state of the ships.

Introducing Event Sourcing adds a step to this process. Now the service creates an event object to record the change and processes it to update the ship.

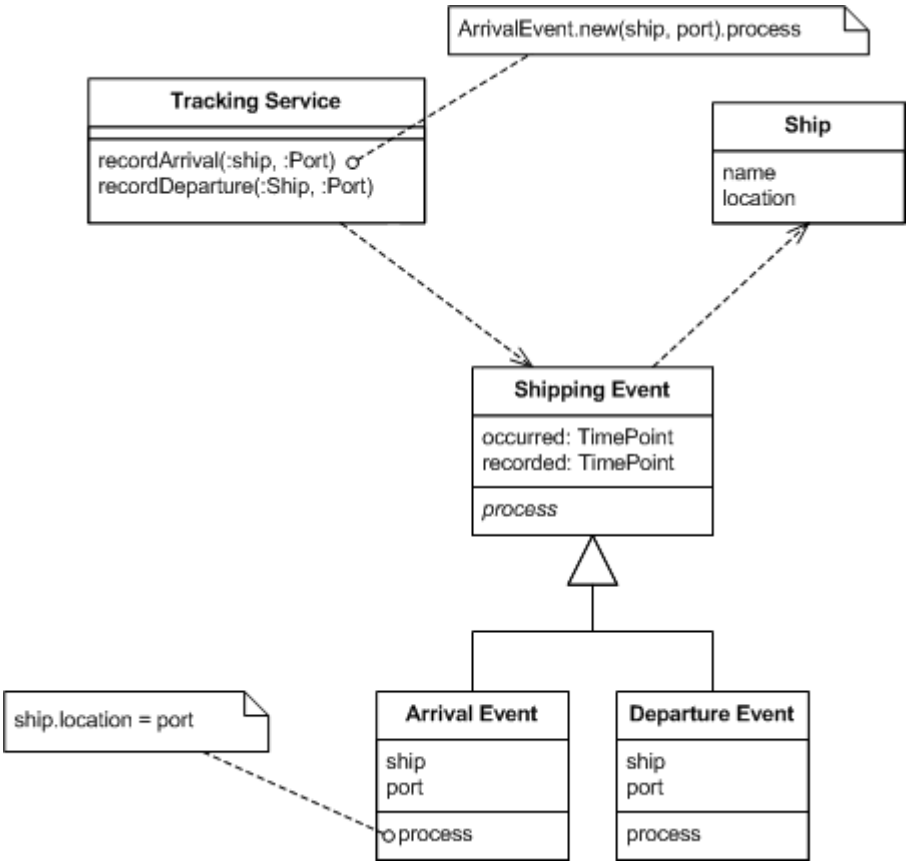

Figure 2: Using an event to capture the change.

Looking at just the processing, this is just an unnecessary level of indirection. The interesting difference is when we look at what persists in the application after a few changes. Let's imagine some simple changes:

- The Ship 'King Roy' departs San Francisco
- The Ship 'Prince Trevor' arrives at Los Angeles
- The Ship 'King Roy' arrives in Hong Kong

With the basic service, we see just the final state captured by the ship objects. I'll refer to this as the application state.

Figure 3: State after a few movements tracked by simple tracker.

With Event Sourcing we also capture each event. If we are using a persistent store the events will be persisted just the same as the ship objects are. I find it useful to say that we are persisting two different things an application state and an event log.

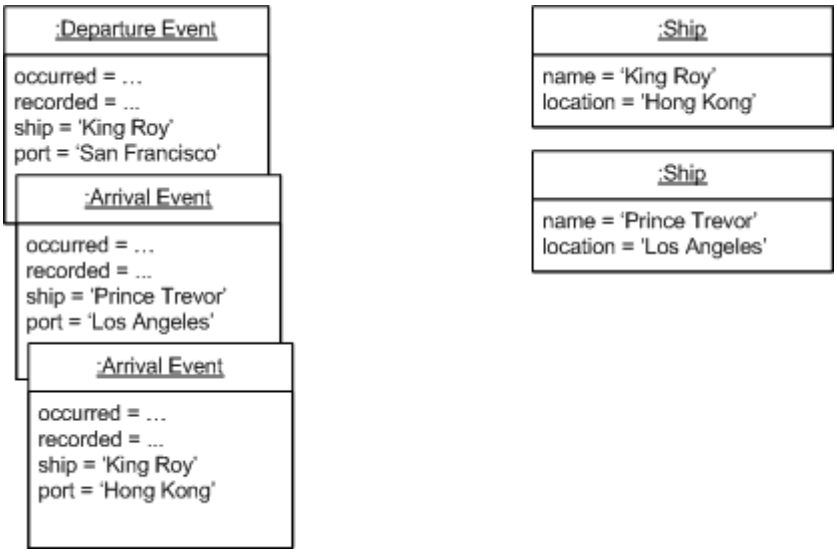

Figure 4: State after a few movements tracked by event sourced tracker.

The most obvious thing we've gained by using Event Sourcing is that we now have a log of all the changes. Not just can we see where each ship is, we can see where it's been. However this is a small gain. We could also do this by keeping a history of past ports in the ship object, or by writing to a log file whenever a ship moves. Both of these can give us an adequate history.

The key to Event Sourcing is that we guarantee that all changes to the domain objects are initiated by the event objects. This leads to a number of facilities that can be built on top of the event log:

- Complete Rebuild: We can discard the application state completely and rebuild it by re-running the events from the event log on an empty application.
- Temporal Query: We can determine the application state at any point in time. Notionally we do this by starting with a blank state and rerunning the events up to a particular time or event. We can take this further by considering multiple time-lines (analogous to branching in a version control system).
- Event Replay: If we find a past event was incorrect, we can compute the consequences by reversing it and later events and then replaying the new event and later events. (Or indeed by throwing away the application state and

replaying all events with the correct event in sequence.) The same technique can handle events received in the wrong sequence - a common problem with systems that communicate with asynchronous messaging.

A common example of an application that uses Event Sourcing is a version control system. Such a system uses temporal queries quite often. Subversion uses complete rebuilds whenever you use dump and restore to move stuff between repository files. I'm not aware of any that do event replay since they are not particularly interested in that information. Enterprise applications that use Event Sourcing are rarer, but I have seen a few applications (or parts of applications) that use it.

Application State Storage

The simplest way to think of using Event Sourcing is to calculate a requested application state by starting from a blank application state and then applying the events to reach the desired state. It's equally simple to see why this is a slow process, particularly if there are many events.

In many applications it's more common to request recent application states, if so a faster alternative is to store the current application state and if someone wants the special features that Event Sourcing offers then that additional capability is built on top.

Application states can be stored either in memory or on disk. Since an application state is purely derivable from the event log, you can cache it anywhere you like. A system in use during a working day could be started at the beginning of the day from an overnight snapshot and hold the current application state in memory. Should it crash it replays the events from the overnight store. At the end of the working day a new snapshot can be made of the state. New snapshots can be made at any time in parallel without bringing down the running application.

The official system of record can either be the event logs or the current application state. If the current application state is held in a database, then the event logs may only be there for audit and special processing. Alternatively the event logs can be the official record and databases can be built from them whenever needed.

Structuring the Event Handler Logic

There are a number of choices about where to put the logic for handling events. The primary choice is whether to put the logic in Transaction Scripts or Domain Model. As usual Transaction Scripts are better for simple logic and a Domain Model is better when things get more complicated.

In general I have noticed a tendency to use Transaction Scripts with applications that drive changes through events or commands. Indeed some people believe that this is a necessary way of structuring systems that are driven this way. This is, however, an illusion.

A good way to think of this is that there are two responsibilities involved. *Processing domain logic* is the business logic that manipulates the application. *Processing selection logic* is the logic that chooses which chunk of processing domain logic should run depending on the incoming event. You can combine these together, essentially this is the Transaction Script approach, but you can also separate them by putting the processing selection logic in the event processing system, and it calls a method in the domain model that contains the processing domain logic.

Once you've made that decision, the next is whether to put the processing selection logic in the event object itself, or have a separate event processor object. The problem with the processor is that it necessarily runs different logic depending on the type of event, which is the kind of type switch that is abhorrent to any good OOer. All things being equal you want the processing selection logic in the event itself, since that's the thing that varies with the type of event.

Of course all things aren't always equal. One case where having a separate processor can make sense is when the event object is a DTO which is serialized and de-serialized by some automatic means that prohibits putting code into the event. In this case you need to find selection logic for the event. My inclination would be to avoid this if at all possible, if you can't then treat the DTO as an hidden data holder for the event and still treat the event as a regular polymorphic object. In this case it's worth doing something moderately clever to match the serialized event DTOs to the actual events using configuration files or (better) naming conventions.

If there's no need to reverse events, then then it's easy to make a Domain Model ignorant of the event log. Reversing logic makes this more tricky since the Domain Model needs to store and retrieve the prior state, which makes it much more handy for the Domain Model to be aware of the event log.

Reversing Events

As well as events playing themselves forwards, it's also often useful for them to be able to reverse themselves.

Reversal is the most straightforward when the event is cast in the form of a difference. An example of this would be "add $10 to Martin's account" as opposed to "set Martin's account to $110". In the former case I can reverse by just subtracting $10, but in the latter case I don't have enough information to recreate the past value of the account.

If the input events don't follow the difference approach, then the event should ensure it stores everything needed for reversal during processing. You can do this by storing the previous values on any value that is changed, or by calculating and storing differences on the event.

This requirement to store has a significant consequence when the processing logic is inside a domain model, since the domain model may alter its internal state in ways which shouldn't be visible to the event object's processing. In this

case it's best to design the domain model to be aware of events and to be able to use them in order to store prior values.

It's worth remembering that all the capabilities of reversing events can be done instead by reverting to a past snapshot and replaying the event stream. As a result reversal is never absolutely needed for functionality. However it may make a big difference to efficiency since you may often be in a position where reversing a few events is much more efficient than using forward play on a lot of events.

External Updates

One of the tricky elements to Event Sourcing is how to deal with external systems that don't follow this approach (and most don't). You get problems when you are sending modifier messages to external systems and when you are receiving queries from other systems.

Many of the advantages of Event Sourcing stem from the ability to replay events at will, but if these events cause update messages to be sent to external systems, then things will go wrong because those external systems don't know the difference between real processing and replays.

To handle this you'll need to wrap any external systems with a Gateway. This in itself isn't too onerous since it's a thoroughly good idea in any case. The gateway has to be a bit more sophisticated so it can deal with any replay processing that the Event Sourcing system is doing.

For rebuilds and temporal queries it's usually sufficient for the gateways to be able to be disabled during the replay processing. You want to do this in a way that's invisible to the domain logic. If the domain logic calls PaymentGateway.send it should do so whether or not you are in replay mode. The gateway should handle that distinction by having a reference to the event processor and checking the whether it's in replay mode before passing the external call off to the outside world.

External updates get more complicated if you are using Retroactive Event see the discussion there for gory details.

Another tactic that you might see with external systems is buffering the external notifications by time. It may be that we don't need to make the external notification right away, instead we only need to do it at the end of the month. In this case we can reprocess more freely until that time appears. We can deal with this either by having gateways that store external messages till the release date, or triggering the external messages through a notification domain event rather than doing the notification immediately.

External Queries

The primary problem with external queries is that the data that they return has an effect on the results on handling an event. If I ask for an exchange rate on December 5th and replay that event on December 20th, I will need the exchange rate on Dec 5 not the later one.

It may be that the external system can give me past data by asking for a value on a date. If it can, and we trust it to be reliable, then we can use that to ensure consistent replay. It also may be that we are using Event Collaboration, in which case all we have to ensure we retain the history of changes.

If we can't use those simple plans then we have to do something a bit more involved. One approach is to design the gateway to the external system so that it remembers the responses to its queries and uses them during replay. To be complete this means that the response to every external query needs to be remembered. If the external data changes slowly it may be reasonable to only remember changes when values change.

External Interaction

Both queries and updates to external systems cause a lot of complication with Event Sourcing. You get the worst of both with interactions that involve both. Such an interaction might be a an external call that both returns a result (a query) but also causes a state change to the external system, such as submitting an order for delivery that return delivery information on that order.

Code Changes

So this discussion has made the assumption that the application processing the events stays the same. Clearly that's not going to be the case. Events handle changes to data, what about changes to code?

We can think as three broad kinds of code changes here: new features, defect fixes, and temporal logic.

New features essentially add new capabilities to the system but don't invalidate things that happened before. These can be added pretty freely at any time. If you want to take advantage of the new features with old events you can just reprocess the events and the new results pop up.

When reprocessing with new features you'll usually want the external gateways turned off, which is the normal case. The exception is when the new features involve these gateways. Even then you may not want to notify for past events, if you do you'll need to put some special handling in for the first reprocess of the old events. It'll be kludgey, but you'll only have to do it once.

Bug fixes occur when you look at past processing and realize it was incorrect. For internal stuff this is really easy to fix, all you need to do is make the fix and reprocess the events. Your application state is now fixed to what it should have been. For many situations this is really rather nice.

Again external gateways bring the complexity. Essentially the gateways need to track the difference between what happened with the bug, and what happens without it. The idea is similar to what needs to happen with Retroactive Events. Indeed if there's a lot of reprocessing to consider it would be worth actually using the Retroactive Event mechanism to replace an event with itself, although to do

that you'll need to ensure the event can correctly reverse the buggy event as well as the correct one.

The third case is where the logic itself changes over time, a rule along the lines of "charge $10 before November 18 and $15 afterwords". This kind of stuff needs to actually go into the domain model itself. The domain model should be able to run events at any time with the correct rules for the event processing. You can do this with conditional logic, but this will get messy if you have much temporal logic. The better route is to hook strategy objects into a Temporal Property: something like `chargingRules.get(aDate).process(anEvent)`. Take a look at Agreement Dispatcher for this kind of style.

There is potentially an overlap between dealing with bugs and temporal logic when old events need to be processed using the the buggy code. This may lead into bi-temporal behavior: "reverse this event according to the rules for Aug 1 that we had on Oct 1 and replace it according to the rules for Aug 1 that we have now". Clearly this stuff can get very messy, don't go down this path unless you really need to.

Some of these issues can be handled by putting the code in the data. Using Adaptive Object Models that figure out the processing using configurations of objects is one way to do this. Another might be embed scripts into your data using some directly executable language that doesn't require compilation - embedding JRuby into a Java app for example. Of course the danger here is keeping under proper configuration control. I would be inclined to do it by ensuring any change to the processing scripts was handled in the same way as any other update - through an event. (Although by now I'm certainly drifting away from observation to speculation.)

Events and Accounts

I've seen some particularly strong examples of Event Sourcing (and consequent patterns) in the context of accounting systems. The two have a very good synergy between them, both in their requirements (audit is very important for accounting systems) and in their implementation. A key factor here is that you can arrange things so that all the accounting consequences of a Domain Event are the creation of Accounting Entrys and link these entries to the original event. This gives you a very good basis for tracking changes, reversal, and the like. In particular it simplified the various adjustment techniques.

Indeed one way of thinking about accounts is that the Accounting Entrys are a log of all of the events that change the value of an Account, so that an Account is itself an example of Event Sourcing.

When to Use It

Packaging up every change to an application as an event is an interface style that not everyone is comfortable with, and many find to be awkward. As a result it's not a natural choice and to use it means that you expect to get some form of return.

One obvious form of return is that it's easy to serialize the events to make an Audit Log.Such an audit trail is useful for audit, no shocks there, but also has other usages. I chatted with someone who got their online accounts into an awkward state and phoned in for help. He was impressed that the helper was able to tell him exactly what he did and thus was able to figure out how to fix it. To provide such a capability means exposing the audit trail to the support group so they can walk through a user's interaction. While Event Sourcing is a good way of doing this, you could also do this with more regular logging mechanisms, and that way not have to deal with the odd interface.

Another use for this kind of complete Audit Log is to help with debugging. Of course, it's old hat to use log files for debugging production problems. But Event Sourcing can go further, allowing you to create a test environment and replay the events into the test environment to see exactly what happend, with the ability to stop, rewind, and replay just like you can when executing tests in a debugger. This can be particularly valuable to do parallel testing before putting an upgrade into production. You can replay the actual events in your test system and test that you get the answers you expect.

Event Sourcing is the foundation for Parallel Models or Retroactive Events. If you want to use either of those patterns you will need to use Event Sourcing first. Indeed this goes to the extent that it's very hard to retrofit these patterns onto a system that wasn't built with Event Sourcing. Thus if you think there's a reasonable chance that the system will need these patterns later it's wise to build Event Sourcing now. This does seem to be one of those cases where it isn't wise to leave this decision to later refactoring.

Event Sourcing also raises some possibilities for your overall architecture, particularly if you are looking for something that is very scalable. There is a fair amount of interest in 'event-driven architecture' these days. This term covers a fair range of ideas, but most of centers around systems communicating through event messages. Such systems can operate in a very loosely coupled parallel style which provides excellent horizontal scalability and resilience to systems failure.

An example of this would be a system with lots of readers and a few writers. Using Event Sourcing this could be delivered as a cluster of systems with in-memory databases, kept up to date with each other through a stream of events. If updates are needed, they can be routed to a single master system (or a tighter cluster of servers around a single database or message queue) which applies the updates to the system of record and then broadcasts the resulting events to the wider cluster of readers. Even when the system of record is the application state in a database this could be a very appealing structure. If the system of record is the event log, there is are plenty of options for very high performance since the event log is a purely additive structure that requires minimal locking.

Such an architecture isn't flawless, of course. The reader systems are liable to be out of sync with the master (and each other) due to differences in timing with event propagation. However this broad style of architecture is used and I've heard almost entirely favorable comment about it.

Using event streams like this also allows new applications to be added easily by tapping into the event streams and populating their own models, which don't need to be the same for all systems. It's an approach that fits in very well with a messaging approach to integration.

Example: Tracking Ships (C#)

Here is a very simple example of Event Sourcing to get the basic idea over. For this example I'm deliberately being ultra-simple to act as a starting point - then I'll use further examples to explore some of the more complicated issues.

The domain model is a simple one of ships that carry cargo and move between ports.

There are four kinds of events that affect the model:

- Arrival: ship arrives at a port
- Departure: ship leaves a port
- Load: cargo is loaded on a ship
- Unload: cargo is unloaded from a ship

Let's take a simple example of moving a ship around.

class Tester...

```
  Ship kr;
  Port sfo, la, yyv;
  Cargo refact;
  EventProcessor eProc;

  [SetUp]
  public void SetUp() {
    eProc = new EventProcessor();
    refact = new Cargo ("Refactoring");
    kr = new Ship("King Roy");
    sfo = new Port("San Francisco", Country.US);
    la = new Port("Los Angeles", Country.US);
    yyv = new Port("Vancouver", Country.CANADA) ;
  }
[Test]
public void ArrivalSetsShipsLocation() {
  ArrivalEvent ev = new ArrivalEvent(new DateTime(2005,11,1), sfo, kr);
  eProc.Process(ev);
  Assert.AreEqual(sfo, kr.Port);
}
[Test]
public void DeparturePutsShipOutToSea() {
  eProc.Process(new ArrivalEvent(new DateTime(2005,10,1), la, kr));
  eProc.Process(new ArrivalEvent(new DateTime(2005,11,1), sfo, kr));
```

```
  eProc.Process(new DepartureEvent(new DateTime(2005,11,1), sfo, kr));
  Assert.AreEqual(Port.AT_SEA, kr.Port);
}
```

To make these tests work we just need the arrival and departure events. The event processor is very simple.

class EventProcessor...

```
  IList log = new ArrayList();
  public void Process(DomainEvent e) {
    e.Process();
    log.Add(e);
  }
```

Each event has a process method.

class DomainEvent...

```
  DateTime _recorded, _occurred;
  internal DomainEvent (DateTime occurred) {
    this._occurred = occurred;
    this._recorded = DateTime.Now;
  }
  abstract internal void Process();
```

The arrival event simply captures the data and has a process method that simply forwards the event to an appropriate domain object.

class DepartureEvent...

```
  Port _port;
  Ship _ship;
  internal Port Port  {get { return _port; }}
  internal Ship Ship  {get { return _ship; }}
  internal DepartureEvent(DateTime time, Port port, Ship ship) : base (time)  {
    this._port = port;
    this._ship = ship;
  }
  internal override void Process() {
    Ship.HandleDeparture(this);
  }
```

So here the event just does the processing selection logic. The processing domain logic is done by the ship.

class Ship...

```
  public Port Port;
  public void HandleDeparture(DepartureEvent ev) {
    Port = Port.AT_SEA;
  }
```

A departure event just sets the Ship's port to a Special Case. You'll notice I'm passing the event into the domain object. There is a choice here about whether the event should pass in just the data the domain object needs for its processing, or pass in the event itself. By passing the event the event doesn't need to know exactly what data the domain logic needs. If the event gains additional data later,

there's no need to update signatures. The downside of passing the event is that the domain logic is now aware of the event itself.

That simple test just shows how the basic event processing works. Now I'll show a little domain logic to see how that would work. Our ships are transporting books to satisfy my fantasies of having an entire shipping line to move my books around the world. As I'm sure you know, it's very dangerous to send books through Canada, since it's this runs the risk of the books being contaminated with lots of "eh". I've seen books where nearly every sentence has a eh (longer sentences can get two or three).

So my perfect book shipping system would be able to detect if a cargo had passed through Canada.

class Tester...

```
[Test]
public void VisitingCanadaMarksCargo() {
  eProc.Process(new LoadEvent(new DateTime(2005,11,1), refact, kr));
  eProc.Process(new ArrivalEvent(new DateTime(2005,11,2), yyv, kr));
  eProc.Process(new DepartureEvent(new DateTime(2005,11,3), yyv, kr ));
  eProc.Process(new ArrivalEvent(new DateTime(2005,11,4), sfo, kr));
  eProc.Process(new UnloadEvent(new DateTime(2005,11,5), refact, kr));
  Assert.IsTrue(refact.HasBeenInCanada);
}
```

Since the cargo may be moved between ships and offloaded, it's the cargo that has the responsibility of knowing whether it's been exposed to these northern dangers. Fortunately the risk only occurs when actually in a port, just being in the waters is safe. So our arrival event has to keep track of this.

class ArrivalEvent...

```
Port _port;
Ship _ship;
internal ArrivalEvent (DateTime occurred, Port port, Ship ship) : base (occurred) {
  this._port = port;
  this._ship = ship;
}
internal Port Port {get {return _port;}}
internal Ship Ship {get{return _ship;}}

internal override void Process() {
  Ship.HandleArrival(this);
}
```

Again the handler is the Ship object.

class Ship...

```
IList cargo;
public void HandleArrival (ArrivalEvent ev) {
  Port = ev.Port;
  foreach (Cargo c in cargo) c.HandleArrival(ev);
}
```

The ship isn't responsible for tracking Canadian visits so it passes the arrival notification on to the cargo.

class Cargo...

```
public bool HasBeenInCanada = false;
public void HandleArrival(ArrivalEvent ev) {
  if (Country.CANADA == ev.Port.Country)
    HasBeenInCanada = true;
}
```

Consider the alternative of where the event process method holds the domain logic - it would need to have a lot of knowledge of the domain model as the domain logic gets more complex. In this approach the domain objects pass the event around to relevant objects so they can handle it to do what they need in response.

Example: Updating an External System (C#)

One of the great features of Event Sourcing is that you can reprocess events as much as you like. However if processing events should cause interactions with external systems, then this is a Bad Thing. The best that could happen is they'll tired of all your Spam events.

An easy way to help deal with this is to ensure that your system calls external systems through gateways that can be configured to ensure that no messages go out unless you are processing an event 'for real'.

I'll illustrate this with a simple example using ships and ports (no cargo this time). Let's say that whenever a ship enters a port it must notify the local customs authority. We can make this happen in the event processing domain logic.

class Port...

```
public void HandleArrival (ArrivalEvent ev) {
  ev.Ship.Port = this;
  Registry.CustomsNotificationGateway.Notify(ev.Occurred, ev.Ship, ev.Port);
}
```

Notice that this code just invokes the notification on the gateway object, it doesn't care whether this is a real processing or some kind of replay. The general principle here is that the domain logic should never care about the context of the running of the events.

It's the gateway's responsibility to figure out whether to actually send the message on or not. Since this case is pretty simple, it does this simply by having a link to the event processor and checking to see if the processor is active.

class CustomsEventGateway...

```
EventProcessor processor;
public void Notify (DateTime arrivalDate, Ship ship, Port port) {
  if (processor.isActive)
    SendToCustoms(BuildArrivalMessage(arrivalDate, ship, port));
}
```

The event processor simply makes itself active when doing regular processing.

class EventProcessor...

```
  public void Process(DomainEvent e) {
    isActive = true;
    e.Process();
    isActive = false;
    log.Add(e);
  }
```

Although this case is very simple, the fundamental principles are the same. Gateways decide whether to send an external message, not the domain logic. The gateways decide this based on information they gather about the context of the processing. In this case a simple boolean state from the processor is enough.

Example: Reversing an Event (C#)

Here we'll take the shipping example and see how to reverse the events. The critical thing we need for reversal is to ensure that we can accurately calculate the prior state of any object that has changed state due to the event.

A good place to store this prior data is on the event itself, something that works quite well with the example's approach of passing the event around the domain objects. Since the domain objects have the event to hand, they can easily store information on the event for them.

The load event makes a simple example. The event carries the following source data.

class LoadEvent...

```
  int _shipCode;
  string _cargoCode;
  internal LoadEvent(DateTime occurred, string cargo, int ship) : base(occurred){
    this._shipCode = ship;
    this._cargoCode = cargo;
  }
  internal Ship Ship {get { return Ship.Find(_shipCode); }}
  internal Cargo Cargo {get { return Cargo.Find(_cargoCode); }}
```

The processing is handed off to the cargo object, which needs to store the port that was the prior location of the cargo.

class LoadEvent...

```
  internal override void Process() {
    Cargo.HandleLoad(this);
  }
internal Port priorPort;
```

class Cargo...

```
  internal void HandleLoad(LoadEvent ev) {
    ev.priorPort = _port;
    _port = null;
    _ship = ev.Ship;
```

```
    _ship.HandleLoad(ev);
  }
```

To reverse the event we add a reverse method that mirrors the process method, calling a reversal method on the domain object.

class LoadEvent...

```
  internal override void Reverse() {
    Cargo.ReverseLoad(this);
  }
```

class Cargo...

```
  public void ReverseLoad(LoadEvent ev) {
    _ship.ReverseLoad(ev);
    _ship = null;
    _port = ev.priorPort;
  }
```

In this case the event takes on a bunch of mutable prior data which is part of its processing data. In cases like this, simple fields suffice. Other cases can require a more sophisticated data structure. When the cargo handles an arrival event, it keeps track of whether it's been in Canada eh. It can do this with a simple boolean field. To store the prior value on the event needs more than a simple field since many cargoes can be affected by an arrival. So in this case I use a map indexed by the cargo.

class Cargo...

```
  public void HandleArrival(ArrivalEvent ev) {
    ev.priorCargoInCanada[this] = _hasBeenInCanada;
    if ("CA" == ev.Port.Country)
      _hasBeenInCanada = true;
  }
  private bool _hasBeenInCanada = false;
  public bool HasBeenInCanada {get { return _hasBeenInCanada;}}
```

class ArrivalEvent...

```
  internal Port priorPort;
  internal IDictionary priorCargoInCanada = new Hashtable();
```

Then to reverse:

class Cargo...

```
  public void ReverseArrival(ArrivalEvent ev) {
    _hasBeenInCanada = (bool) ev.priorCargoInCanada[this];
  }
```

This example nicely illustrates how the source data on the event and the error handling affects how we do reversal. For the load event we need to store the port that the cargo was at when it was loaded. If the event included this on its source data we wouldn't have to do this. A bit of extra source data removes the need to add prior data. This doesn't work everywhere: the arrival event can't source the prior Canadian state of its cargoes.

What's deemed correct processing can also make a difference. In this system we have both arrival events and departure events for ships. Assuming all works correctly we should always get these interleaved. Thus a ship could reverse an arrival event by setting its port field to `Port.OUT_TO_SEA`. What happens when we get two arrival events in succession? To reverse this we need to store the prior port on the ship. Our alternative would be to declare a second arrival without a departure as an error, we don't need to do this storage.

Example: External Query (C#)

External queries are awkward even for basic Event Sourcing because if you want to rebuild the application state you need to do this using external query responses that were made in the past.

Let's imagine a case where a ship has to determine the value of its cargoes when it enters a port. This valuation is done by an external service. If a ship enters a port on November 3rd and I process the event immediately, I'll get the value of that cargo as at November 3rd. If I rebuild my application state on December 5th I'll want the same value of the cargo, even if its value has changed in the meantime.

For this example I'll show one way of handling this by keeping a record of external queries and using them to provide values when reprocessing an event. In many ways it's a similar approach to that you use with an external update - turn the interaction into events at the boundary of the system and use the record of events to remember what happened.

Using a similar style to event handling that I'm using elsewhere in these examples, we pass the event to the domain objects to handle. In this case the cargo initiates the call to the external system and saves the value. (We'll assume we're going to actually do something useful with this value, but that's not relevant to this example.)

class Cargo...

```
public void HandleArrival(ArrivalEvent ev) {
  declaredValue = Registry.PricingGateway.GetPrice(this);
}
private Money declaredValue;
```

As is my habit, I encapsulate all external system access through a Gateway that I control. The basic gateway will just translate the call into whatever is needed for the external interaction. To support replaying events I just wrap this with a class that logs these queries.

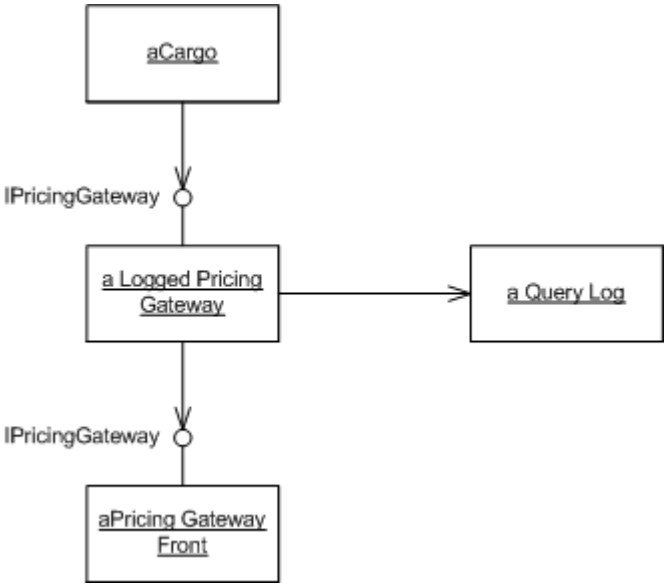

Figure 6: Wrapping the gateway with a logging gateway to support proper rebuilding from events.

The logging gateway checks each call to see if it has an old request that matches this one, if not it makes a new request.

class LoggedPricingGateway...

```
  public Money GetPrice(Cargo cargo) {
    GetPriceRequest oldReq = oldRequest(cargo);
    if (null != oldReq) return (Money) oldReq.Result;
    else return newRequest(cargo);
  }
```

If the request is a new one, it turns the request into an event object, invokes the external query, and stores it in the log.

class LoggedPricingGateway...

```
  private Money newRequest(Cargo cargo) {
    GetPriceRequest request = new GetPriceRequest(cargo);
    request.Result = gateway.GetPrice(cargo);
    log.Store(request);
    return (Money) request.Result;
  }
private class GetPriceRequest : QueryEvent {
  private Cargo cargo;
  public GetPriceRequest(Cargo cargo) : base() {
    this.cargo = cargo;
  }
```

class QueryEvent...

```
  DomainEvent _eventBeingProcessed;
  Object result;
  public QueryEvent() {
    _eventBeingProcessed = Registry.EventProcessor.CurrentEvent;
  }
  public object Result {
    get { return result; }
    set { result = value; }
  }
  public DomainEvent EventBeingProcessed {
```

```
    get { return _eventBeingProcessed; }
  }
}
```

So to find an old request, it searches its log.

class LoggedPricingGateway...

```
  private GetPriceRequest oldRequest(Cargo cargo) {
    IList candidates =
      log.FindBy(EventProcessor.CurrentEvent, typeof (GetPriceRequest));
    foreach (GetPriceRequest request in candidates) {
      if (request.Cargo.RegistrationCode == cargo.RegistrationCode)
        return request;
    }
    return null;
  }
```

The query log is generic, so we can issue a query to get a few items by using the Domain Event that was processed and the type of request. This gives us a small set that needs further checking in a gateway-specific manner.

The log of requests will need to be persisted in the same way as the log of Domain Events is persisted as it's needed to rebuild the application state.