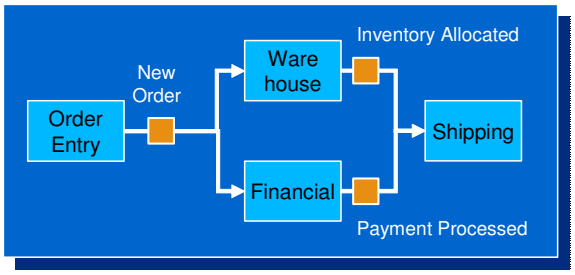
Programming Without a Call Stack – Event-driven Architectures

Gregor Hohpe
www.eaipatterns.com

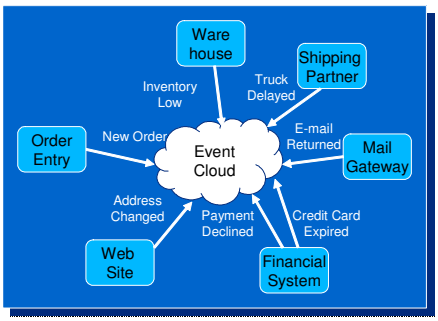
Most computer systems are built on a command-and-control scheme: one method calls another method and instructs it to perform some action or to retrieve some required information. But often the real world works differently. A company receives a new order; a web server receives a request for a Web page, the right front wheel of my car locks up. In neither case did the system (order processing, web server, anti-lock brake control) schedule or request the action. Instead the *event* occurred based on external action or activity, caused either by the physical world or another, connected computer system. Could we change the architecture of our system to relinquish control and instead respond to events as they arrive? What would such a system look like?

Events Everywhere

The real world is full of events. The alarm goes off; the phone rings; the "gas low" warning light in the car comes on. Many computer systems, especially embedded systems, are designed to respond to events. The engine control computer in your car receives an event every time the crankshaft is at the zero position and starts the timer for another round of ignitions. As of now, many of the systems that function based on external events live in a rather small universe, most of them even invisible to the user. However, as computer systems become more and more interconnected they start to publish and receive an increasing number of events. An order management system may receive orders from a Web site or an order entry application and notify other systems of the new order. Systems interested in new orders might be the financial system, which will see whether the order is backed with a credit line or a valid credit card to charge, and the warehouse, which verifies that inventory to fulfill the order is present. Each of these systems might then publish another event to any interested party. The shipping system in turn might wait for both an Inventory Allocated and Payment Processed message and in response prepare the goods for shipment. This event-based style of interaction is notably different from the traditional command-and-control style that would have the warehouse ask for the inventory status, wait for an answer, and then ask the financial system to process the payment. Next, the order management system would wait for a positive answer and lastly instruct the shipping system to send the goods.



But the event story does not end here. The warehouse might detect that inventory is low and let other systems, for example the procurement system, know. When the customer's credit card is about to expire we might to be alerted so we can send an e-mail to the customer requesting a new card number. The list of interesting events goes on and on. David Luckham [POE] coined the term "Event Cloud" to describe the interchange of many events between multiple systems.



Event-driven Architectures

So what defines the step from simply exchanging information through events to a full-fledged event-driven architecture (EDA)? EDAs exhibit the following set of key characteristics:

- *Broadcast Communications*. Participating systems broadcast events to any interested party. More than one party can listen to the event and process it.
- *Timeliness*. Systems publish events as they occur instead of storing them locally and waiting for the processing cycle, such as a nightly batch cycle.
- *Asynchrony*. The publishing system does not wait for the receiving system(s) to process the event(s).
- *Fine Grained Events*. Applications tend to publish individual events as opposed to a single aggregated event. (The further apart the communicating parties are, the more may physical limitations limit how fine grained the events can afford to be)

EDA Key Characteristics

- Broadcast communications
- Timeliness
- Asynchrony
- Fine Grained
- Ontology
- Complex Event Processing

- *Ontology*. The overall system defines a nomenclature to classify events, typically in some form of hierarchy. Receiving systems can often express interest in individual events or categories of events.
- *Complex Events Processing*: The system understands and monitors the relationships between events, for example event aggregation (a pattern of events implies a higher-level event) or causality (one event is caused by another).

Event-driven architectures (EDA) tend to exhibit an aura of simple elegance. Because these systems are modeled after real world events the resulting system model is usually very expressive. These desirable benefits have already motivated some EAI (Enterprise Application Integration) vendors to proclaim that EDAs are the next step in the evolution beyond Service-oriented Architectures (SOAs).

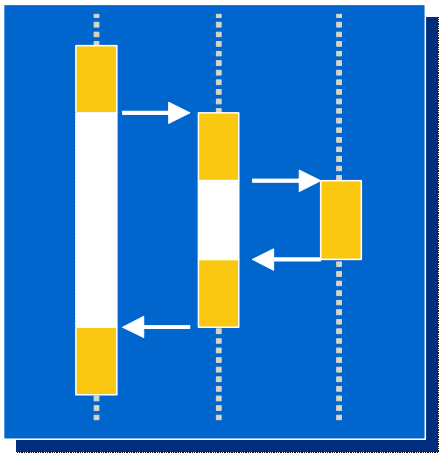
Good Bye, Call Stack

However, the simple elegance of EDAs can be deceiving. Designing such a system correctly can actually be more challenging than it may initially appear. As we saw above,

one of the key properties of event-based systems is the simplified interaction between components that is restricted to the exchange of events. Therefore, in order to understand the design implications of an EDA we should not look at what an EDA introduces, but should begin by examining what an EDA is taking away. An event-based architecture takes away what must be one of the most pervasive and underappreciated constructs in programming – the call stack.

Call stack based interaction allows one method to invoke another, wait for the results, and then continue with the next instruction. This behavior can be summarized as three main features: coordination, continuation and context. Coordination provides for synchronized execution, i.e. the calling method waits for the results of the called method before it continues. The continuation aspect ensures that after the called method completes the execution continues with the statement following the method call. Lastly, the call stack holds local variables as part of the execution context: once the method invocation completes the caller's entire context is restored.

The interaction between components in an EDA does not provide any of these functions – the interaction is limited to one component publishing an event that can be received (usually with a delay) by one or more other components. There is no inherent coordination, continuation or context preservation. Why would one want to eliminate these tremendously useful features that every developer has come to appreciate? The answer lies in the fact that the convenience of the call stack comes at the price of assumptions. While assumptions per se are not necessarily a bad thing, it is important to make them explicit so one can evaluate whether the desired execution environment matches the assumptions or not. So let's have a quick look at the key assumptions that accompany the ever-present call stack.



First, a call stack is primarily useful in environments where one thing happens after another. The fact that a single return address is pushed onto the stack implies a single path of execution where the caller's execution does not continue until the called method completes. The distinct advantage is that the called method does not have to have worry about synchronization or concurrency issues. Because the call stack prefers a single line of execution it implicitly assumes that method invocations and executions are fast compared to the execution of the primary code. This makes it practical for the caller to wait for the callee's results before it continues processing. If invocations are slow or carry a large overhead this assumption could become a liability. For example, this is the very reason Web services-based architectures are moving away from an RPC-based to a message-based communication style.

Call Stack Assumptions

- One thing happens at a time
- We know what should happen in what order
- We know who can provide a needed function
- Execution happens in a single virtual machine

More subtle but equally important is the assumption that the systems knows what should happen in what order. Because one method calls another method directly the calling method has to have a pretty clear idea of what it wants to happen next.

Not only is the caller assumed to know what is supposed to happen next but the caller also has to be aware which method can provide the desired functionality. It might seem odd at first to separate knowing what to do from knowing what method to invoke. After all, methods are (or at least should be) named after the function they accomplish. Still, having one method call another method directly means tying the execution of a specific piece of functionality to the invocation of a specific method. Often, this direct linkage has to be established at compile time and is not easily changed afterwards. In many cases this linkage is not a problem. It seems perfectly acceptable for me to call `customer.setName()` to set a customer's name. If need be, polymorphism can provide for a level of indirection between caller and executor so that a subclass of `customer` can sneak in a different implementation of `setName`.

Last but not least, the call stack flourishes in an environment where caller and callee share the same memory space. This allows compiler and linker to insert direct references to methods and keep method call overhead small. Also, a single memory, single processor environment is inherently geared towards sequential execution, which is again matches nicely with the call stack mentality.

Focus on Interaction

A call stack defines a specific interaction style between components, one that is equally popular and well understood. Because a call stack is assumed to be the standard mode of interaction most object-oriented design tends to focus on the structural aspects of the solution over the aspects related to interaction. This is generally appropriate for most object-oriented systems that exist in a single memory space and are under the control of a single development team, i.e. systems that fulfill the basic assumptions required by a call stack.

Traditional object-oriented design does not ignore interaction altogether. Some of the classic design patterns presented in [GOF] concern themselves with the way objects interact. For example, the *Mediator* "encapsulates how a set of objects interact" while the *Observer* "notifies all dependent objects of a state change". Both patterns elevate the interaction between objects from their shadow life to become first class players in the object model. These patterns give us a hint that looking at the way components interact can be more interesting than might at first appear.

In distributed systems the cost of interaction goes up significantly and the significance of interaction suddenly increases dramatically. At the same time structural aspects can move into the background as distributed systems often do not provide for rich structural mechanisms such as inheritance, polymorphism and the like. For example, this shift of attention from structure to interaction is at the heart of many of the debates on service-oriented computing. Service-oriented architectures have rather simple composition rules but pay close attention to loosely coupled interaction between systems.

To Couple or Not to Couple

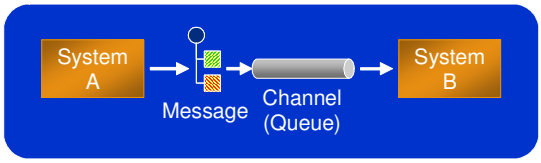
Service-oriented architectures have brought the notion of coupling into the forefront of our minds. Coupling is a measure of the dependency between two communicating entities. The more assumptions the entities make about one another the more tightly coupled they are. For example, if the communicating entities use a technology specific communication format they are more tightly coupled than entities that communicate over a technology-neutral format. Loose coupling is desired in situations that require independent variability, for example because the communicating entities are under control of different organizations. Looser coupling and therefore fewer assumptions leave more room for variation.

The way components interact also impact coupling between entities. The more rules and assumptions the interaction protocol prescribes, the more coupling between the components is introduced. Simpler interaction rules imply less coupling because fewer constraints are imposed on the participating entities.

Two primary strategies can help reduce the coupling that results from the interaction between components:

1) Insert a level of indirection
2) Simplify the rules of interaction

It has been postulated that in the field of computer science any problem can be solved simply by adding an additional more level of indirection. Of course, the problem of coupling is not immune to this approach. If we want to avoid one component to interact with another component without being directly linked to that component we can insert a component in the middle to isolate the two. In the object-oriented world this is exactly what the *Mediator* pattern [GOF] does: one object calls the mediator, who in turn figures out which other object to call. This approach improves reuse between objects because the interaction between them is extracted into a separate element, which can be configured or changed without having to touch the original components. The same approach lays the foundation of message-oriented architectures [EIP]. Instead of communicating directly, components send messages across event channels.

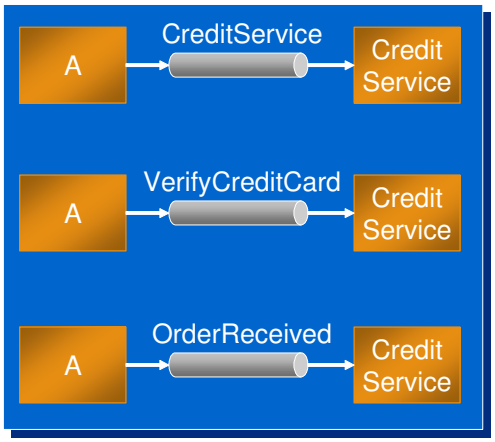


The second aspect of coupling focuses on the rules of the interaction. A call-stack oriented interaction has fairly strict rules: one method calls the other and waits for the results of the invocation. Subsequently, execution always continues where it left off. One way to reduce coupling between the interacting parties is to simplify the rules of the interaction. If we remove the continuation and coordination aspects of the interaction, all that is left is the fact that one component sends data to another component. We would be hard pressed to define a form of communication that is even simpler while still being worthy of the name interaction.

What is in a Name?

The channel-based interaction introduces two new elements, a channel and a message. Despite their simplicity these new elements open up options and force new decisions. A deceivingly simple question is "how should the channel be named?" When one method called another directly there was no intermediate element and therefore no decision to make.

A very simple approach assigns each component its own channel. For example, a component that deals with credit card validations could be called the `CreditService` and react to messages sent on a channel named `CreditService`. If any component needs something related to credit it could send a message to that channel. While the channel gives us a level of indirection at the implementation level (we could replace one credit service implementation with another without anyone noticing) the semantics of the interaction are not much decoupled. The caller still has to know which component provides the functionality it requires, much like it did in the call stack scenario.



To reduce the dependency on a specific service we could name the channel analogous to a method name. For example, if the service provided an operation that can verify a credit card supplied by a customer we might simply name the channel `VerifyCreditCard`. This does increase the level of abstraction somewhat because the caller no longer has to know which component is able to service this type of request. Service-oriented computing generally follows this approach.

Despite the introduction of the channel the semantics of the interaction still smell like a call stack. One component sends a request ("check this credit card") and expects a response ("card good" or "card bad"). But we have not yet exhausted the creative possibilities of the channel semantics. The above examples assume that the component knows that a credit card has to be verified. Can we lift this burden from the "caller" altogether so that the components are truly decoupled? We can take the decoupling one step further by changing the channel name (and the associated semantics) to `OrderReceived`. This simple change in name signifies a significant shift in responsibility. The message on the channel no longer represents an instruction but an *event*, a notification that something happened. We also no longer assume that there is a single recipient for the event. Again, the assumptions between the communicating parties have been reduced. As a result, EDA is often considered to be more loosely coupled than SOA.

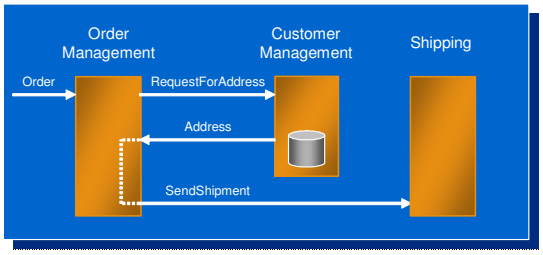
Shifting Responsibilities

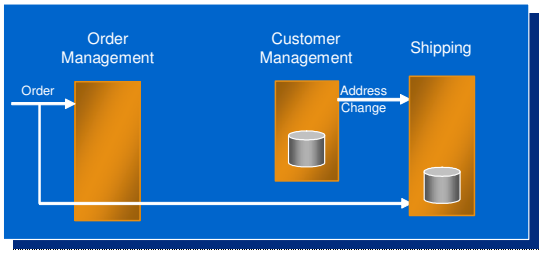
Communicating through events as opposed to commands indicates a subtle but important shift of responsibility. It allows components to be decoupled to the extent that the "caller" is no longer aware of what function is executed next nor which component is executing it.

Another equally important shift of responsibility between caller and callee is that of keeping state.

In a system that is based on queries and commands state is usually kept in one application that is considered the "master" for the data. When another application needs to reference that data it sends a query to the owning application and waits for the response before it continues processing. For example, when the order management system needs to fulfill an order it queries the customer management systems for the customer's address so it can tell the shipping application to send the shipment to that address.

Event-driven systems work differently, almost to the inverse. Systems so not query systems for information but instead keep their own copy of the required data and listen to updates as they occur. In our example this would mean that the shipping system keeps its own copy of the customer's address so when an order arrives it can use that address to label the shipment without having to query the customer management system. While replicating data this way might seem dangerous it also has advantages. The customer management system simply broadcasts changes to the data without having to know who all keeps a copy. Because the customer management is never queried for address data it never becomes a bottleneck even as the system grows and the demands for addresses multiply.





The principle behind the shift in responsibility is once again tied to the concept of coupling. In a loosely coupled interaction a source of data should not be required to keep state at the convenience of its communication partners. By shifting the burden of keeping state to the consumer the component is can be completely oblivious to the needs of the data consumers – the key ingredient into loose coupling. The shift away from the query-response pattern of interaction means that many components have to act as event Aggregators [EIP]: they listen to events from multiple sources, keep the relevant state and combine information from multiple events into new events. For example, the shipping system effectively combines address change events and order events into request for shipment to a specific address.

Complex Events

An EDA can offer more benefits than loose coupling and independent variability. A system where components interact only through events makes it easy to track the all interaction and analyze them. A whole new discipline has emerged around the analysis of event sequences and the understanding of event hierarchies. For example, a rapid series of similar request events to a Web server might mean that the server is under a distributed denial of service attack. The fact that this sequence of request events occurred is in itself a meaningful event that should be published into the event cloud. This type of event hierarchy is the subject of Complex Event Processing or CEP [POE].

Instant Replay

Event-based systems can exhibit another enormous benefit. If all interaction in a system occurs through events one can recreate the system state from scratch simply by replaying all events. Financial systems typically fall into this category. An account not only keeps its current state (i.e. the balance) but also maintains the history of all events that affected the account (i.e. deposits, withdrawals). Martin Fowler calls this approach *Event Sourcing* [EAA].

Event sourcing application exhibit two valuable properties. First, the system state can be recreated even if the state of an individual component was lost. By replaying all events to the components each component can sequentially recover the state it was in without resorting to other persistence mechanisms. More interesting even is the ability to replay the events but with changes. In a sense, we can rewrite history by inserting changes into a past stream of events and then replay the revised event stream. This feature can be invaluable in real-life scenarios. For example, a customer who orders a certain amount of goods over the year may qualify for a year-end rebate. If a customer's orders just exceed the limit but the customer returns an item in the new year he should not get the rebate. Traditional systems implement specific logic that checks whether a return moves the customer below the threshold and debits the customer with the rebate they originally received. An event sourced system can solve this situation more elegantly. A returned item voids the original "purchase" event. Subsequently we can replay the revised series of purchase events, skipping the voided event. The associated business logic will compute the customer's final account balance, not including the rebate. We can then compare the revised scenario with the original and compute the adjustment without having to understand the original business logic (i.e., when a rebate is paid). One can easily see that for complex business rules event replay can be an invaluable feature.

Composition

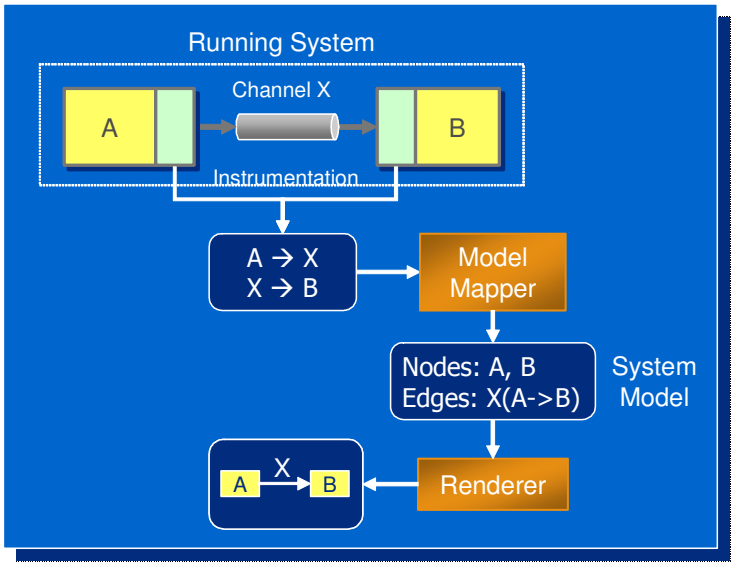
So where is the catch? EDAs apparently exhibit a series of desirable properties. But the flexibility that the loose coupling affords usually comes at a price. This price amounts to less build-time validation and the fact that a highly composable system allows us to compose it in many ways that do not make a lot of sense or do not do what we had in mind. For example, an event source and a listener might accidentally be configured for a different type of event or channel, potentially the result of a trivial typo in the name. As a result the event sink will not receive any events at all. However, we do not find out until we start the system and even then it can be difficult to determine the actual source of the problem. Is the listener listening to the wrong event? Is the sender sending the wrong event? Is the event source publishing any events at all? Is the event channel interrupted? The configurability can suddenly turn into a debugging liability. This is exactly what Martin Fowler warns us of when he describes "the architect's dream, the developer's nightmare".

With variability comes uncertainty. If the system architecture allows the individual components to evolve, tomorrow's system may look different than yesterday's system. It is therefore imperative to create tools that help with configuration and analysis. The composition of individual components into a coherent, event-driven application should be viewed as an additional layer of the overall system architecture. This layer should be

taken as seriously as the core code layers. All too often is this type of composition information (e.g. the names of published or subscribed event channels) hidden away in cryptic configuration files that are scattered across machines. Instead, one should define a domain language specifically for the composition layer. This language could include validation rules that flag valid configurations. For example, a configuration which prescribes an event subscription that does not match any publication may be considered invalid. Likewise, circular references in the event graph may be undesirable and should be detected at design time.

Visualization

In a highly distributed and loosely coupled system, determining the actual system state alone can be challenging because they continue to evolve continuously. To make matters worse, critical information is often spread across many machines. In these situations it can be invaluable to generate system model from the running system. This can be accomplished by instrumenting the running system with sensors, which track the sending and receiving of messages. The sensors forward the harvested information to a central location that maps it onto an abstracted system model, for example a directed graph.



Such a graph can then be run through a graph rendering algorithm such as AT&T GraphViz [VIZ]. The result is a human-readable, accurate model of the systems structure. Such a model and diagram can be invaluable for debugging and analysis.

Summary

Event-based systems can offer an interesting alternative to traditional command-and-control system design. EDAs enable loosely coupled, highly composable systems that often provide a close mapping to real-life events. Using events consistently as the interaction mechanism between components enables techniques such as event replay, which can be very difficult to accomplish in traditional designs. However, all these benefits come at a price. Systems that pass up the well-known tenets of a call stack in favor of loosely structured interaction are inherently more difficult to design and debug. Therefore, one should employ management and visualizations tools to create a system that is dynamic but not chaotic.

References

[CSP] *Communicating Sequential Processes*, C.A.R. Hoare, 1985, Prentice Hall, on-line at http://www.usingcsp.com

[EAA] *Further Patterns of Enterprise Application Architecture*, Martin Fowler, http://www.martinfowler.com/dev/eaa

[GOF] *Design Patterns*, Gamma et al., 1995, Addison-Wesley

[EIP] *Enterprise Integration Patterns*, Hohpe, Woolf, 2003, Addison-Wesley, www.eaipatterns.com

[POE] *The Power of Events*, Luckham, 2002, Addison-Wesley, www.complexevents.com

[VIZ] GraphViz Graph Visualization Software, http://www.graphviz.org/

About the Author

Gregor Hohpe is a software architect with Google, Inc. Gregor is a widely recognized thought leader on asynchronous messaging and service-oriented architectures. He co-authored the seminal book "Enterprise Integration Patterns" and has been working extensively with the Microsoft Patterns & Practices group. Gregor is a MVP Solution Architect and speaks regularly at technical conferences around the world. Find out more about his work at www.eaipatterns.com.