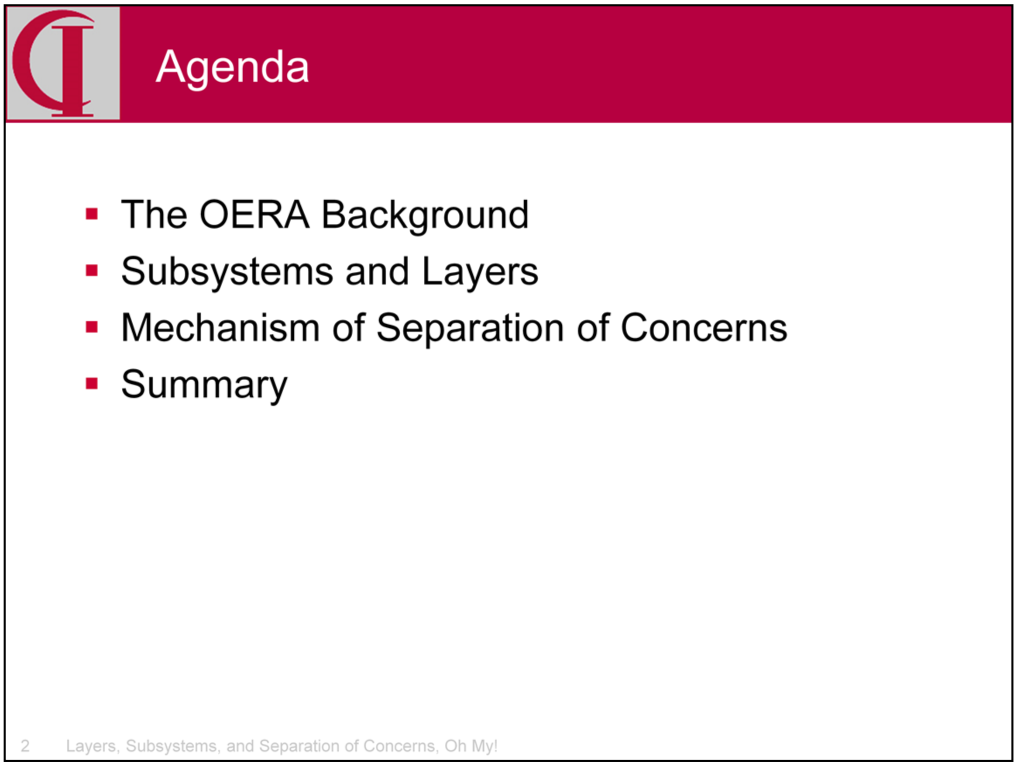
Layers,
Subsystems, and
Separation of Concerns,
Oh My!

Dr. Thomas Mercer-Hursh
VP Technology
Computing Integrity, Inc.

Let me begin by introducing myself. I have been a Progress Application Partner since 1986 and for many years I was the architect and chief developer for our ERP application. In recent years I have refocused on the problems of transforming and modernizing legacy ABL applications.



Here is our agenda for today. First I am going to talk a little about OERA and what it means. Then talk about Subsystems in relationship to that meaning. And, then look at some mechanisms.

Agenda

- The OERA Background
- Subsystems and Layers
- Mechanism of Separation of Concerns
- Summary

First, let's talk a bit about OERA and what it really means.

The OERA Background

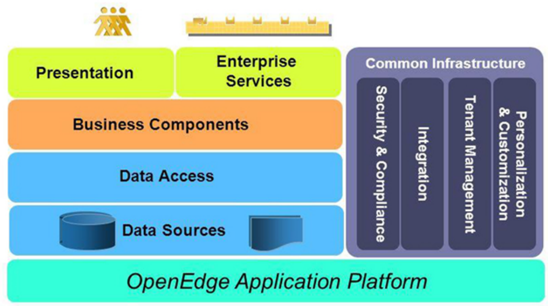
OpenEdge Reference Architecture

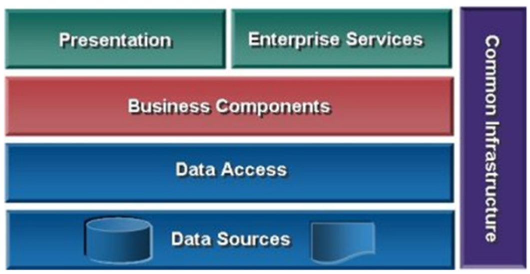
- Introduced idea of constructing an application in layers.
- Each layer had a single uniform purpose.
- One “layer” might have more than one subject matter.

I am sure that you have all seen one or more versions of a diagram of OERA like this. This diagram introduced the OpenEdge community to the idea of constructing an application in layers where each layer had a single uniform purpose, although it might have more than one subject matter.

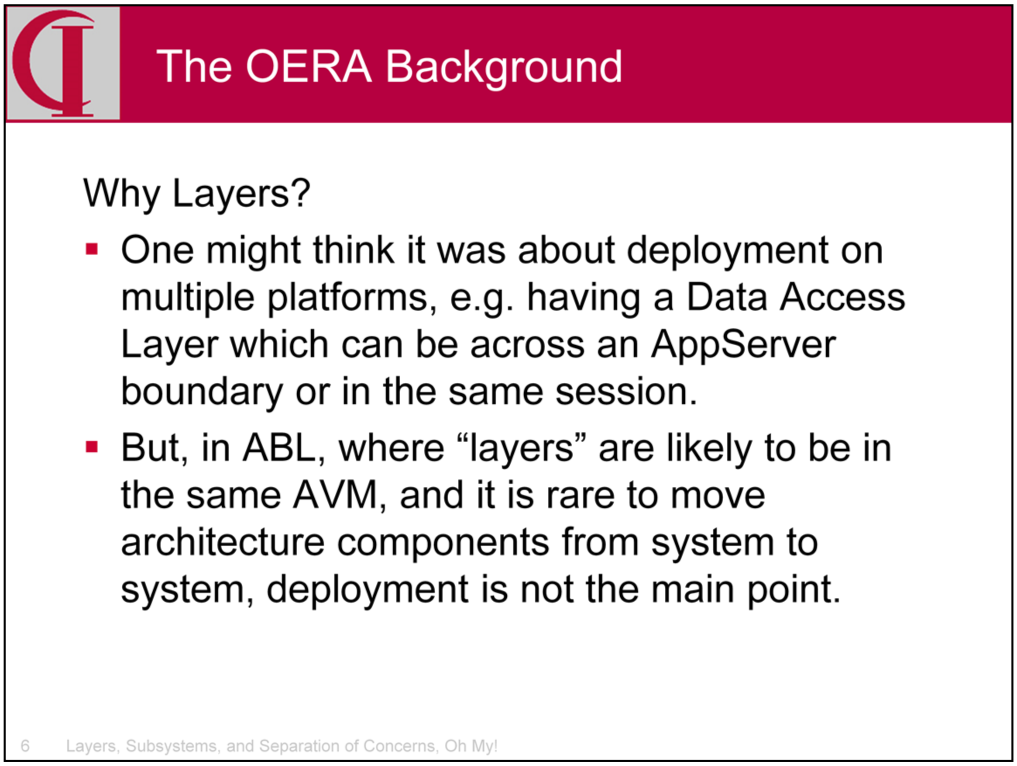
The OERA Background

- Diagram has come in a number of flavors.
- All have this clear sense of partitioning by type of service.
- Each may emphasize a particular area.

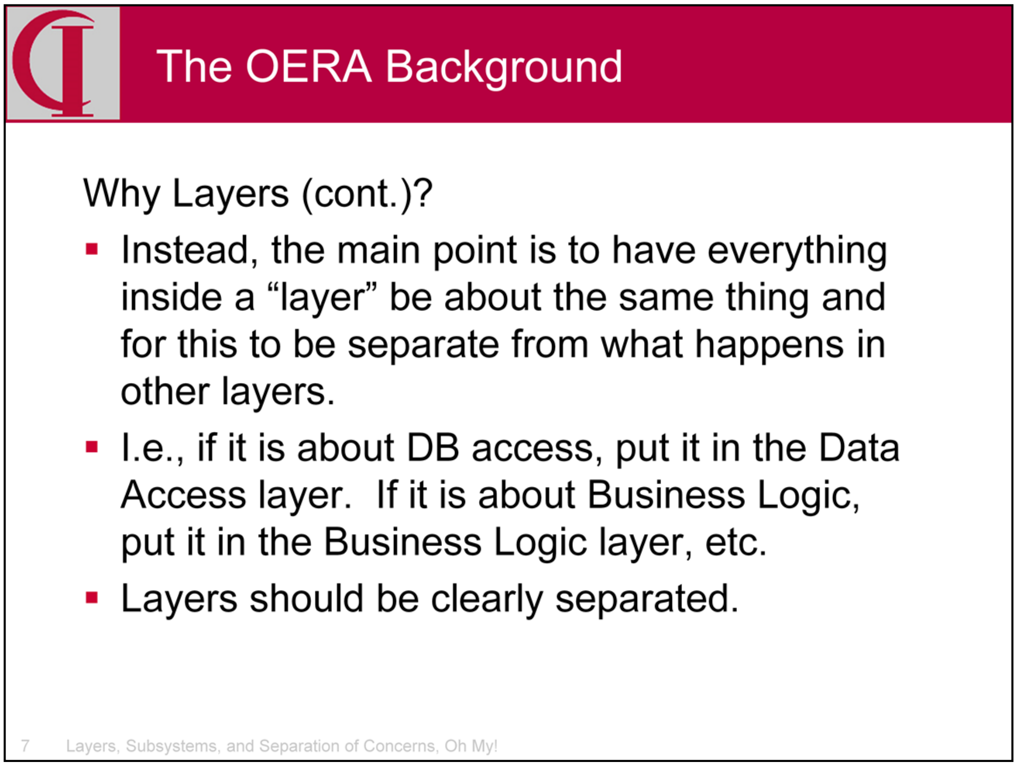




The diagram has appeared in a number of different flavors, providing different emphasis. The one in the lower left was an early and simple example, which nevertheless recognized that layers might have infrastructure that sat off to the side of all layers. The one to the upper right breaks this down into more categories and brings emphasis to a common platform. But, all have a clear sense of partitioning based on the type of service provided.



So, what is the point of layers? Some people might be tempted to think that it was about deployment, e.g., what one might put on either side of an AppServer boundary. Well, it can have that implication in some cases, but it is very common in ABL for multiple layers, possibly even all of them, to reside in a single AVM. It is also rare to change the location of components after the original architectural decisions. So deployment is not the main point.



For me, the main point of writing an application in layers is that everything in any given layer is about the same kind of thing and for that to be as separate as possible from other layers. So, if there is a class that is about interacting with the database, it belongs in the data access layer. Whereas, if the class is about business logic, it belongs in the business logic layer. These two should be as separate as possible. Unlike the Big Ball of Mud (BBOM) designs which were characteristic of legacy ABL in which UI, logic, and data access were all mushed together in one program.

The OERA Background

Why separate?

- Separation provides insulation from change – if something changes about how the Customer is stored, the business logic doesn't need to be impacted because the data access takes care of that.
- Separation localizes change – a change in how stored only needs modification in the component responsible for change.
- Most true when classes correspond to "problem space" entities.

But, what is the virtue of separation you might ask? Separation localizes change and insulates other components from change. If one makes a change in how some part of the Customer is stored, e.g., putting the Address in a separate table, that change only impacts the data access component and needs not impact the business logic or UI components in any way. This is most true when there is a natural correspondence between classes in the application and entities in the problem space since changes in the program are likely correspond to changes in our model of the problem space.

In this talk I will be using the term "problem space" to refer to the real world entities and behavior which the application is trying to model. This is not necessarily the real world in all its possible complexity, but our view of it that we have defined as the problem we are trying to solve.

The OERA Background

Separation of Concerns

- Divide an application into distinct components that overlap in functionality as little as possible.
- A “concern” is a focus of responsibility and the associated attributes and behavior.
- Encapsulation is the embodiment of Separation of Concerns.

In formal terms, this leads us to one of the central tenets of object-oriented programming – separation of concerns. This means that an application should be divided into components, any one component is an expression of a specific “responsibility”, some coherent combination of necessarily related attributes and behavior. Also, there should be as little overlap in functionality between one component and the next. If the responsibility of a particular component is managing the data access for Customers, then nowhere else should there be data access for Customers and the only thing in that component should be the functionality required to access customer data. Each class should be about one thing and should contain all of the attributes and behavior related to that thing. Each class should be strongly separated from other classes. This concept is what is meant by the term Encapsulation.

Agenda

- The OERA Background
- Subsystems and Layers
- Mechanism of Separation of Concerns
- Summary

So, with this background, let’s see what this notion of Separation of Concerns does to our perception of layers.

Subsystems and Layers

Separation of Concerns and Layers

- A layer provides separation by function, but not by subject matter.
- Layer separation is important for all the reasons one wants separation between different areas of responsibility … deployment, isolation of change, technology changes, etc.
- But, layer separation isn't the only separation we want.

The Separation of Concerns one achieves with layers is a separation by function type. This is certainly valuable for providing flexability in deployment, isolation of change to a single component, and freedom to make changes in technology. But, it is not the only separation we want in our application.

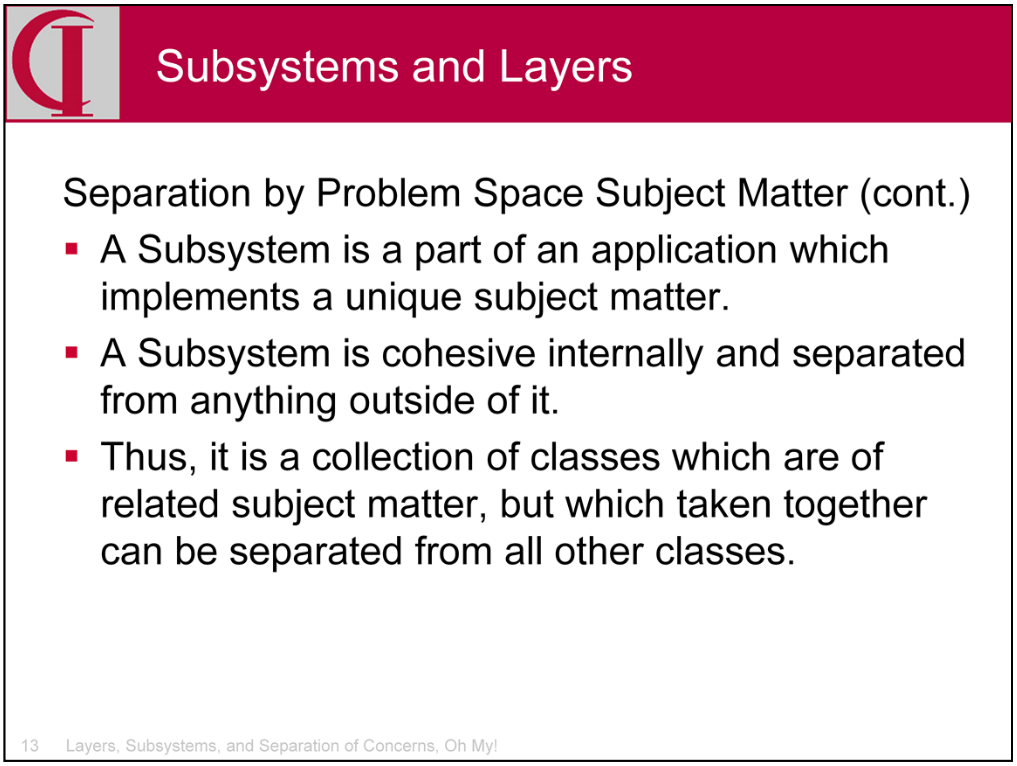
Subsystems and Layers

Separation by Problem Space Subject Matter

- Having the structure of the application mirror the structure of the problem space is a key characteristic of object-oriented design.
- The core of this mirroring is having a Class which corresponds to a problem space entity.
- When Classes interact strongly, it may be convenient to organize them into a Package for namespace convenience, but this may or may not correspond to a clear problem space unit.

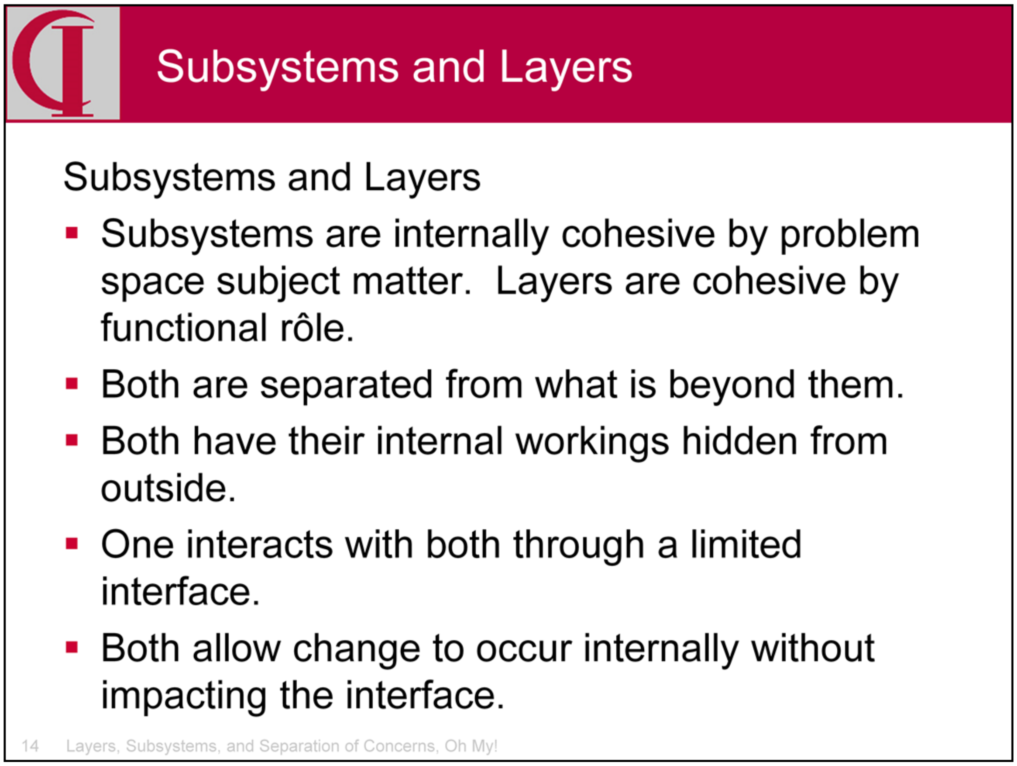
The other separation we want is by Subject Matter where the divisions of Subject matter are those we find in the problem space, not in our code. In Object-Oriented Analysis and Design, the core principle is that a class in the design will correspond to a specific entity in the problem space. Often, we will organize closely interacting classes into Packages. Ideally, this Package will also correspond to a natural set of entities in the problem space, but this may not always be true.

Consider example



At a higher level of grouping we have Subsystems. A good Subsystem will always implement a unique body of Subject Matter which can be easily identified. It will be internally cohesive, i.e., all the classes it contains will be about closely related subject matters. It will be cleanly separated from anything outside the system.

Consider example



Subsystems and layers are similar in purpose, but are different in that layers are about functional roles in the architecture and subsystems are about subject matter subdivisions. Both are cohesive internally. Both are separated externally. Both hide their inner workings from the outside. Both should be provided with a simple interface and all interaction should be through that interface. And, both allow one to change the interior in implementation or technology without impacting the interface.

Consider example

Subsystems and Layers

Subsystems and Layers

- Subsystems are identified by Application Partitioning, logical decomposition of the problem space into cohesive units.
- Layers are identified by functional architectural relationships.
- Subsystems are unique to the application.
- Layers are fixed by the current architecture.

One identifies Subsystems through a process called Application Partitioning, i.e., the logical decomposition of the problem space into cohesive units. Application Partitioning is often a combination of top down and bottom up processing -- identifying broad categories from the top, identifying classes to go with individual problem space entities from the bottom, and resolving the grouping into Subsystems as the two meet in the middle. Layers tend to be defined by the architectural structure and are independent of the specific subject matter of the application. Subsystems, of course, are unique to the application. Related applications may have some similar Subsystems, but the content of each will be defined by the specifics of the application.

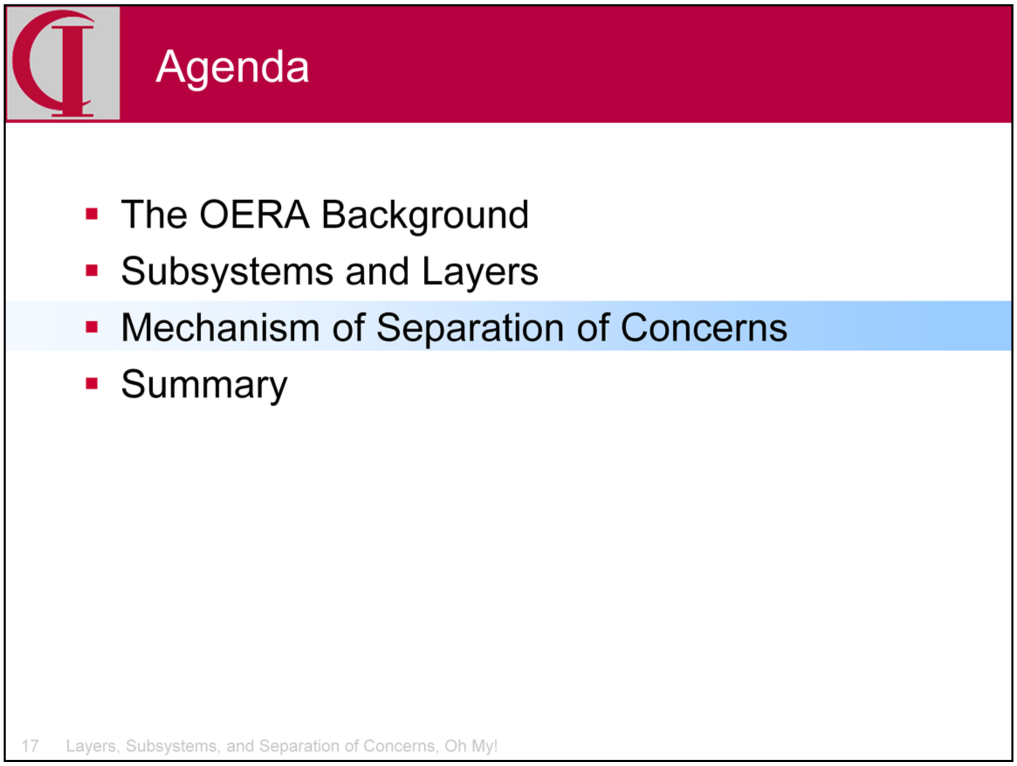
Consider example

Subsystems and Layers

Identifying Subsystems

- Focus on natural units in the problem space.
- Look for areas of logical cohesion which can be separated from other areas.
- Look for a unified responsibility.
- Look for natural boundaries where a simple interface can communicate what needs to be exchanged.
- Look for behavior which is functionally isolated, not dependent on other subsystems.

In identifying Subsystems, keep your attention on the problem space, not the computing space. Look for natural units. Look for entities which are strongly related to each other. Look for entities which work together to accomplish some larger purpose. Look for boundaries of possible separation, especially those where there may be some natural interface such as a document or message which controls the interaction.



Now let's look at some of the mechanisms by which we help support Separation of Concerns.

Mechanisms of Separation of Concerns

Having decided to construct our application divided into layers by functionality and subsystems by subject matter and having established that both should be internally cohesive and separate from others, how do we get these pieces to interact?

Understand that the separation is key to long term maintenance, allowing us to change one component without changing others, whether in response to changes in problem space requirements or functional architecture issues.

So, having decided that separation of concerns is important for both layers and subsystems, what mechanisms are we going to use to implement this separation and still allow the components to work together.

Remember that separation is essential if we want to be able to change individual components without changing everything they are connected to. We can need to make changes because of changes in problem space requirements or because we change our ideas about architecture … and we will make such changes … and it is separation that will allow us to make them individually on the affected component and not have to fiddle with everything that component is connected to.

Mechanisms of Separation of Concerns

The first requirement is to get the design right.

Get it right and there will be a natural, obvious flow between components.

Get it wrong and you will be constantly tinkering to make it work.

If you find yourself tinkering, revisit the design.

Everything starts with the right design. Break down the problem space into the right subsystems and you will have a simple, natural flow of simple messages between them. Break it down incorrectly and you will find yourself adjusting left and right to try to make things work. If you find yourself doing that, it is time to revisit the design.

Mechanisms of Separation of Concerns

What do we want the interaction to look like?

- Messages, not calls, i.e., data not execution.
- Events, not commands.
- Communication is by published interface.
- No knowledge of implementation in the other component.

So, what do these simple, natural interactions look like? We want the interactions to be messages, not calls, i.e., A is passing some information to B and it is up to B to decide what it needs to do with it, not A commanding B to do something. Typically, these messages are about events, i.e., that something new has happened. Each subsystem should have a published interface and all interaction should be via that interface. We should never be reaching down into another subsystem to use a component independent of the interface because that implies that we have knowledge of how the other subsystem is implemented.

Mechanisms of Separation of Concerns

Messages and Events

- “This happened”, rather than “Do this”.
  - “This happened” is only about oneself.
  - “Do this” implies knowledge of the other component’s capabilities.
- Events do not presuppose what the other component will do with the information.
- Obviously, some messages are requests, but they should not presuppose how the information requested will be provided.

This focus on messages and events is summarized by “This happened”, rather than “Do this”. “This happened” is a subsystem reporting about something that happened and it is letting others know in case they are interested. “Do this” is one subsystem telling another subsystem what to do and often implies knowing not only what the other subsystem can do, but how it is going to do it. Events don’t presuppose knowing what the other subsystem will do with the information. Obviously, some messages are going to look a lot like requests, but they certainly shouldn’t presume how the other subsystem will do its job or they will be coupled.

Note that connections between layers will often not have quite the same degree of separation. A business logic layer is going to ask a data access layer for a particular set of information. It still should have no presumption about how the data access layer will do that including whether the data is even local.

Mechanisms of Separation of Concerns

How does one send a "Message"?

- Mechanism depends on deployment:
  - If remote, requires a message passing mechanism.
  - If local, can be as simple as a method call.
- The communication is with an interface, not with the members of the component directly.
- A message contains only data.

So, what do I mean when I say "message"? How do I send one?

How depends on deployment. If local to the AVM, it can be as simple as a method call. If the other subsystem is remote, then we need some kind of message passing system.

Remember that the communication is always interface to interface, not reaching directly in to the classes within.

And, a message contains only data, not behavior.

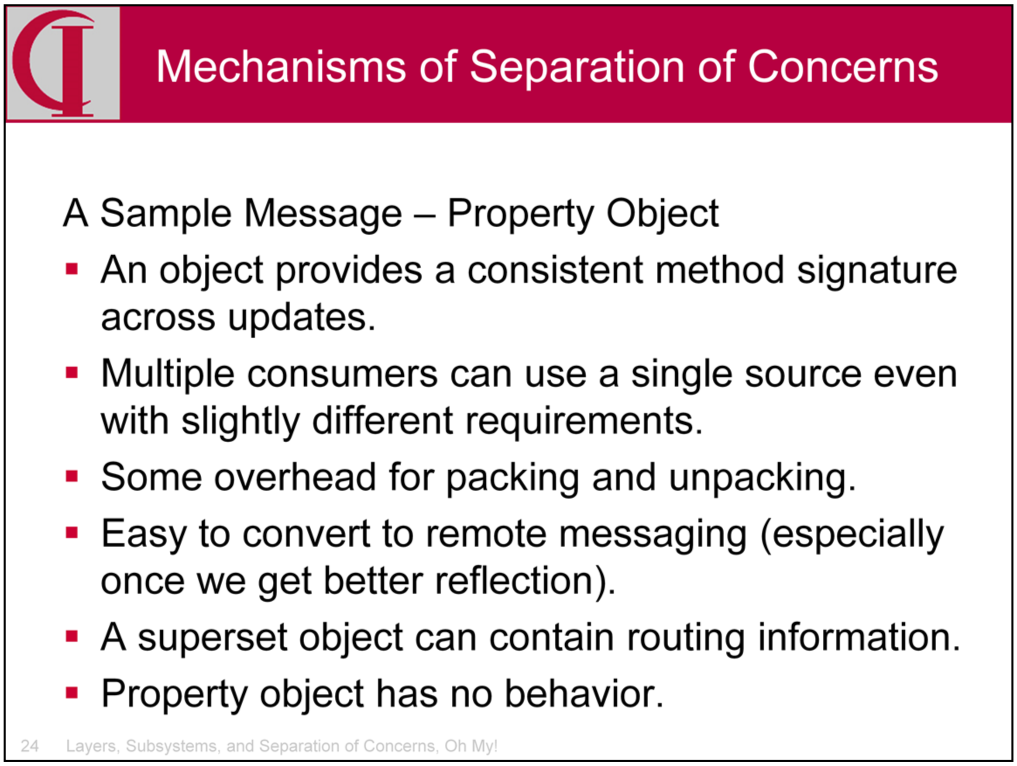
Mechanisms of Separation of Concerns

A Sample Message – Property Object

```
class CustomerMsg:
  define public property CustNum as integer no-undo get. set.
  define public property Name as character no-undo get. set.
  define public property Address as character no-undo get. set.
  define public property Address2 as character no-undo get. set.
  define public property City as character no-undo get. set.
  define public property State as character no-undo get. set.
  define public property Country as character no-undo get. set.
  …
end class.
```


Here is an example of what we mean by a property object – a class which has properties and nothing else. We can put data in, transfer the object, and take data out and that is all.

Note, property objects are the exception to the rule of “you make it; you destroy it” because it is the recipient who knows when they are done with the object.



Using property objects provide some advantages and disadvantages over simple parameters. The method signature is simple and remains unchanged when the contents of the property object change. It is possible for multiple consumers to use the same property object, even though they may not need even value within it. One wants to be careful to not overdo this, of course. There is some overhead for packing and unpacking. It is easy to convert property objects to remote messaging and will be even easier when we get better reflection. If we make property objects a child of a superset object, the superset object can contain message type and routing information which is accessible to an intermediary without having to be aware of the implementation details of the specific object.

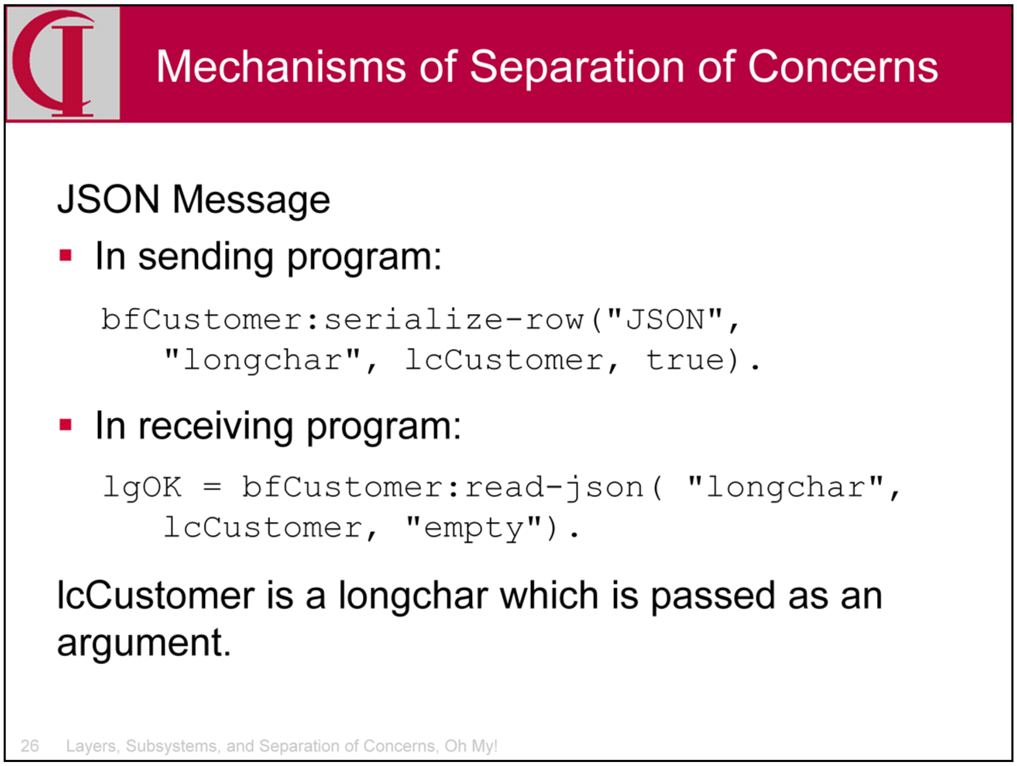
Again, property objects contain no behavior … unless we decide to provide a way for them to serialize themselves before PSC gets around to doing this for us.

Mechanisms of Separation of Concerns

JSON Message

- An alternative to a property object is to convert the information to a JSON string and simply pass the string as a parameter.
- Again, consistent interface despite changing data.
- Again, no dependence on implementation of either side.
- Particularly convenient for data in temp-tables.

Another approach is to use a JSON message. This could be XML, but these days JSON seems more interesting because it is more broadly useful and less verbose. One gets a JSON message simply by converting the individual data elements to a JSON string. Again, we have a consistent message signature because it is just a string. Because it is just data, there is no dependence on either side for implementation and JSON strings can be consumed by many technologies. It is particularly convenient for data in temp-tables.



To illustrate how simple this can be, if we have a buffer with Customer data in the sending program, serializing this to a JSON string is literally a one line instruction. Likewise, reading the JSON string back into a buffer on the other end is also one line. If we want to parse the fields in the JSON string independently, of course it takes a few more instructions.

Mechanisms of Separation of Concerns

Knowing too much

- Making direct calls to the methods of another object implies knowledge of that object. This is OK within a package of related objects, but undesirable between subsystems where one is trying to keep things loosely coupled.
- The solution is to pass messages through an interface which then takes responsibility for routing to the correct internal component.

One of the goals of Separation of Concerns is to avoid one subsystem knowing too much about the internal implementation of another subsystem because that creates dependencies. If, for example, if one subsystem makes direct calls into the methods of internal classes of another subsystem then one has created a dependency where one can't change the internal implementation of that subsystem without breaking the other subsystem which depends on it. Within a subsystem we can accept this dependency, although we still strive for encapsulation, but not between subsystems.

The solution is for all interaction with a subsystem to be through an interface which handles all messages in and out. This interface can route the message to the appropriate internal class based on the type of message. How is this managed?

Mechanisms of Separation of Concerns

Given a parent class for all messages:

```
class com.cintegrity.SM.MessageBase inherits Incident:
  define public property targetDomain as character no-undo get. set.
  define public property targetService as character no-undo get. set.

  constructor public MessageBase(
      in_targetDomain as character,
      in_targetService as character,
      in_type as character
      ):
    super(in_type).
    targetDomain = in_targetDomain.
    targetService = in_targetService.
  end constructor.
end class.
```

All messages inherit from MessageBase. All messages set a domain, requested service, and a type (in parent). A general dispatcher routes the message to the interface for the subsystem according to the domain and that interface examines the service to act accordingly.

So, let's look at how we might implement this approach. We would start by defining a parent class for all messages. This has two properties, targetDomain and targetService. These are set in the constructor. The superclass Incident contains the definition for type which is not an immediate part of this discussion.

All messages will inherent from this superclass. In creating all message objects, we will set these three values.

We can then send the message to a general purpose dispatcher who can interrogate the destination domain by inquiring of the superclass. It can then route the message to the subsystem appropriate for that domain. This way, the sender doesn't even need to know the name of the other subsystem and doesn't have to interact with it directly.

Mechanisms of Separation of Concerns

Sample code:

```
method public void DispatchMessage(
    incoming_message as MessageBase
    ):
  ...
  define variable obInMsgPerformCreditReview as class
      AR_PerformCreditReviewMessage no-undo.
  define variable obInMsgReceiveCustomer as
      AR_ReceiveCustomerMessage no-undo.
```


Then, in the interface of the receiving domain, we have a method to receive the message, which again takes it in as the superclass. It will then define variables for each of the possible subclasses and …

Mechanisms of Separation of Concerns

Sample code:

```
  case incoming_message:targetService:
  ...
    when "PerformCreditReview"
      then do:
        obInMsgPerformCreditReview = cast(incoming_message,
            AR_PerformCreditReviewMessage).
        PerformCreditReview(obInMsgPerformCreditReview:min_balance,
             obInMsgPerformCreditReview:min_credit_limit,
       obInMsgPerformCreditReview:nearness_pct).
      end.
    when "ReceiveCustomer"
      then do:
        obInMsgReceiveCustomer = cast(incoming_message,
            AR_ReceiveCustomerMessage).
        ReceiveCustomer(obInMsgReceiveCustomer:customer).
      end.
    ...
  end case.
end method.
```


Select on the service in the Message base to identify the type of action required. It then casts the superclass into the specific message type and calls a small method with that object that implements the desired action by calls to the classes of the subsystem as needed.

Mechanisms of Separation of Concerns

Dispatcher

- When a subsystem is requested, the Dispatcher checks to see if it is already started. If not, it starts it and adds its interface to the available services table.
- When a message is received, the message is checked for domain and the domain looked up in the subsystem table.
- The message is then sent to the interface for that subsystem.

Let me say a bit more about the Dispatcher I mentioned.

Mechanisms of Separation of Concerns

Dependency Injection

- If A needs to interact with B, but B is not subordinate to A, then how does A get a link to B?
- One answer is for the factory which creates A to also create B and supply the link to A.
- The factory is then responsible for the life cycle of both A and B and can manage them independently.

For an example and discussion see:
https://community.progress.com/community_groups/openedge_architecture/w/openedgecloudarcade/998.oeri-dependency-injection-container-injectabl.aspx

Mechanisms of Separation of Concerns

Messaging Services

- Another way to provide real isolation of subsystems is to communicate through a messaging service such as Sonic or Stomp.
- Messages will almost naturally conform to the desired standard because of the limits of what can be sent in a message.
- Opens up broad notifications to anyone interested or possible multiple services to process a given message type.
- The service takes the rôle of the Dispatcher.

Mechanisms of Separation of Concerns

AppServer Considerations

- Since the client end of a AppServer connection can be almost any technology, it might seem desirable to use a neutral message format such as a JSON string.
- But, AppServer does handle conversions to all target technologies, so anything but classes can be sent to all technologies.
- However, the principles of isolating subsystems and layers still apply.

Agenda

- The OERA Background
- Subsystems and Layers
- Mechanism of Separation of Concerns
- Summary

In Summary

- OERA layers are primarily for Separation of Concern, not Deployment.
- Subsystems provide Separation of Concern by Subject Matter.
- Good Subsystems depend on good Application Partitioning.
- Communication between Subsystems should be by Events, not commands.
- Property Objects and JSON strings provide simple consistent message signatures.

In Summary

- Interface to Subsystems should be through a simple Interface which hides the internal classes and methods.
- A Dispatcher can be used to further remove dependencies
- Other techniques include Dependency Injection and Messaging Services.

For More Information, go to…

- Books
  - H.S. Lahman: *Model-Based Development: Applications* from Addison-Wesley (ISBN 978-0321774071).
  - Stephen J. Mellor & Marc J. Balcer: *Executable UML: A Foundation for Model-Driven Architecture* from Addison-Wesley (ISBN 0-201-74804-0)
  - Leon Starr: *Executable UML: How to Build Class Models* from Prentice Hall (ISBN 0-13-067479-6)

Contact me at thomas@cintegrity.com.

Related presentations at http:///www.cintegrity.com

Here are some links for more information.

Questions?

Thank You

Dr. Thomas Mercer-Hursh
thomas@cintegrity.com