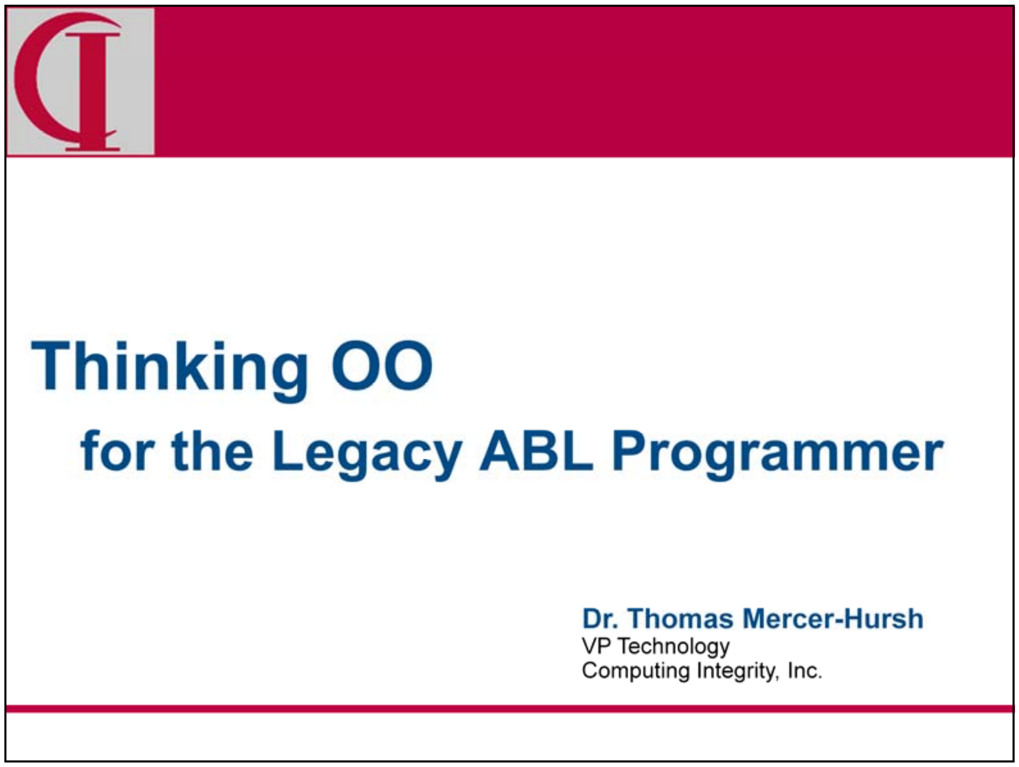
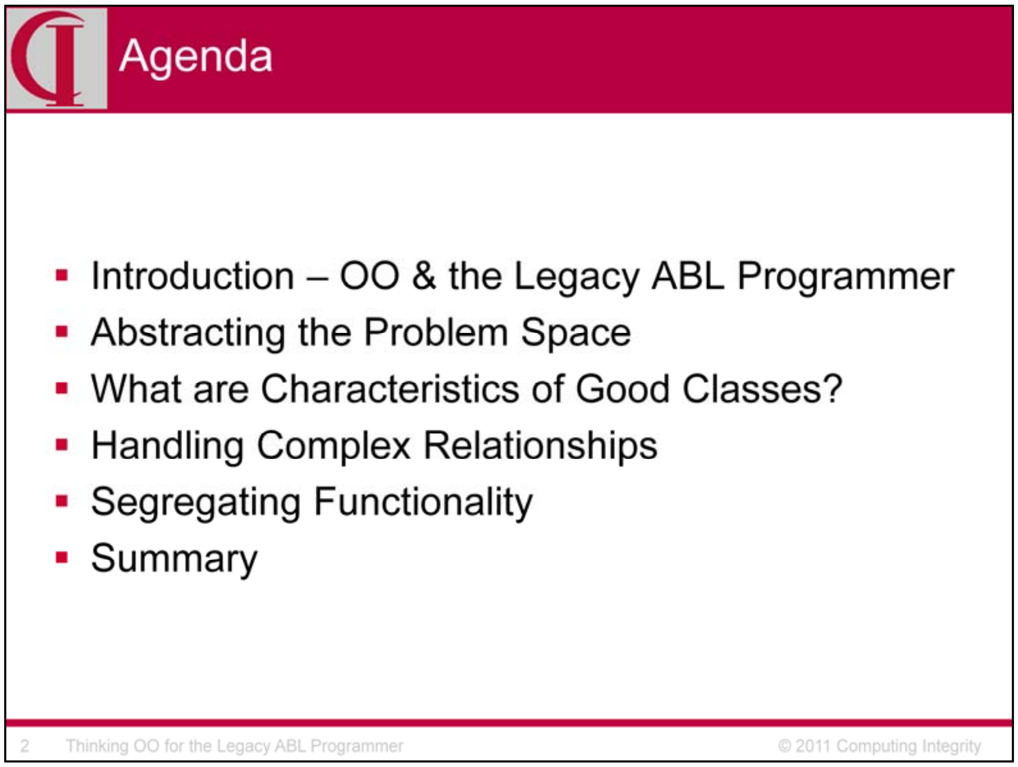


Thinking OO for the Legacy ABL Programmer

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

Let me begin by introducing myself. I began working with Progress in 1984 and I have been a Progress Application Partner since 1986. 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. Object Orientation is widely accepted as a preferred paradigm for developing complex applications by much of the programming world and now that OO features are now in ABL, I and others have been exploring the benefits of using OO in ABL.

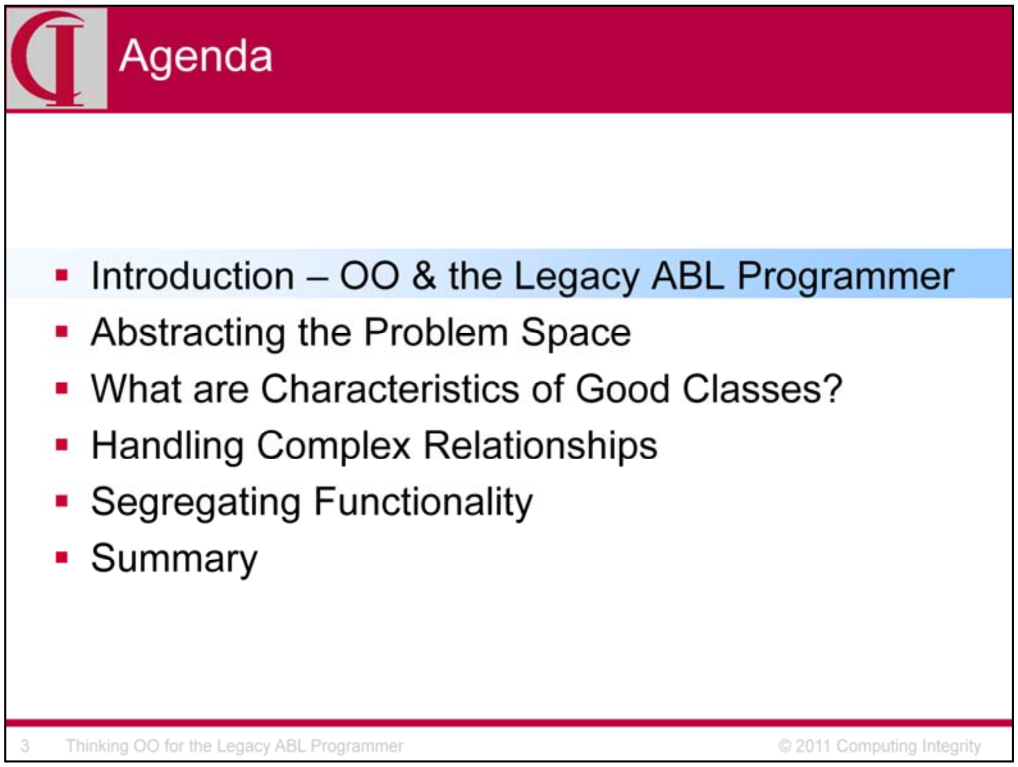


Agenda

- Introduction – OO & the Legacy ABL Programmer
- Abstracting the Problem Space
- What are Characteristics of Good Classes?
- Handling Complex Relationships
- Segregating Functionality
- Summary

2 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

So, here’s our agenda for today. First we will set the stage about why this presentation exists, then we will compare the building blocks of legacy ABL applications to OO applications, then we will look a bit about how OO thinks about the problem space vs the computational space, and then we will look a bit at how OO applications are structured.

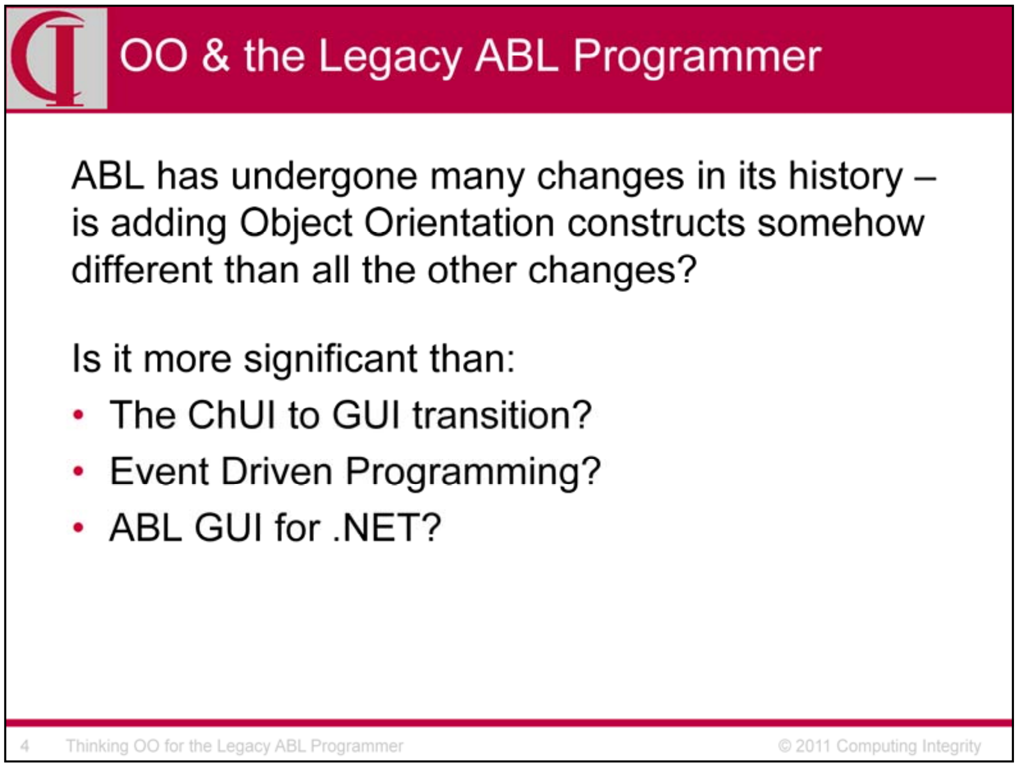


Agenda

- Introduction – OO & the Legacy ABL Programmer
- Abstracting the Problem Space
- What are Characteristics of Good Classes?
- Handling Complex Relationships
- Segregating Functionality
- Summary

3 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

First, why did I create this presentation?



OO & the Legacy ABL Programmer

ABL has undergone many changes in its history – is adding Object Orientation constructs somehow different than all the other changes?

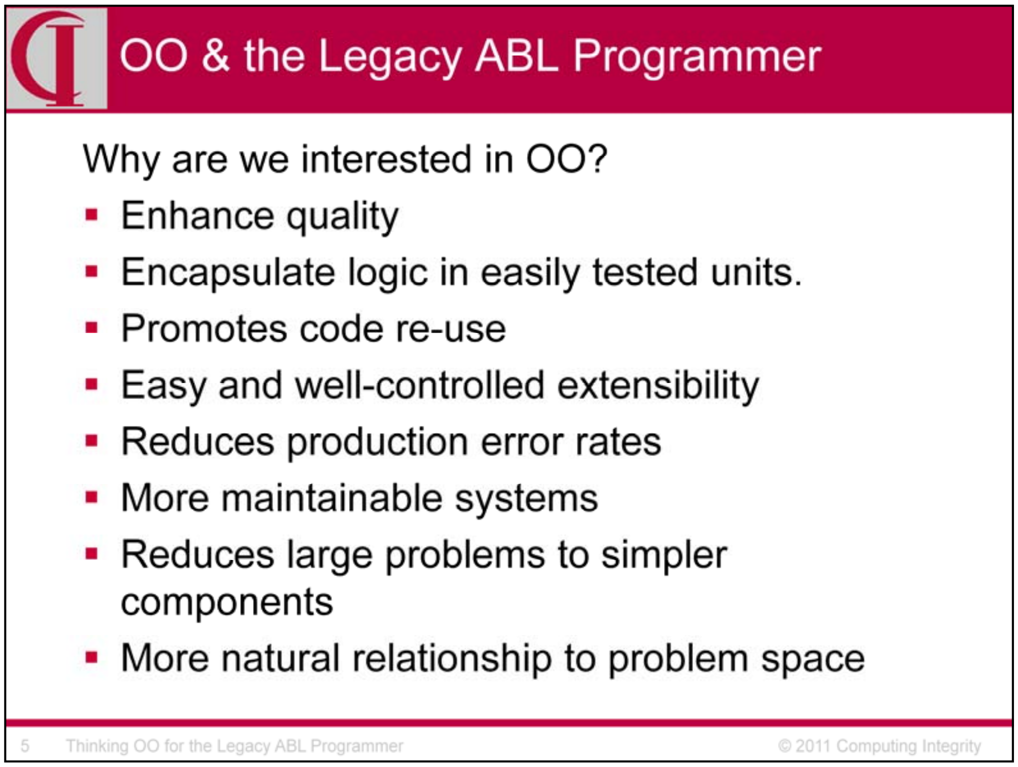
Is it more significant than:

- The ChUI to GUI transition?
- Event Driven Programming?
- ABL GUI for .NET?

4 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Those of you who have been working in ABL for a while know that the language has evolved quite a bit over the years. Some of those transitions have been pretty dramatic, changing not only what we were capable of doing, but the very structure of our programs. For example, the V6 to V7 transition which not only brought us the first blush of the richness of GUI user interfaces, it also started us on the conversion from procedural to event driven programming. To be sure, that was a very big transition … one that many applications still haven't made or have made only incompletely. And, of course, the more recent addition of ABL GUI for .NET not only made for a much, much richer user interface, but the programming for that interface was quite different than what we were used to for even traditional ABL GUI interfaces with these complex, external components to manage.

But, while PSC themselves has emphasized the continuity and compatibility of the OO extensions to ABL … rightly so, perhaps, and certainly in an effort not to scare us … I suggest that really making the transition to OO programming involves a bigger shift in mindset than any of these historical transitions. Yes, one can use OO constructs in legacy ABL programs without really taking on the OO mindset, but if one does, one won't fully benefit from the benefits which OO programming can provide.



OO & the Legacy ABL Programmer

Why are we interested in OO?

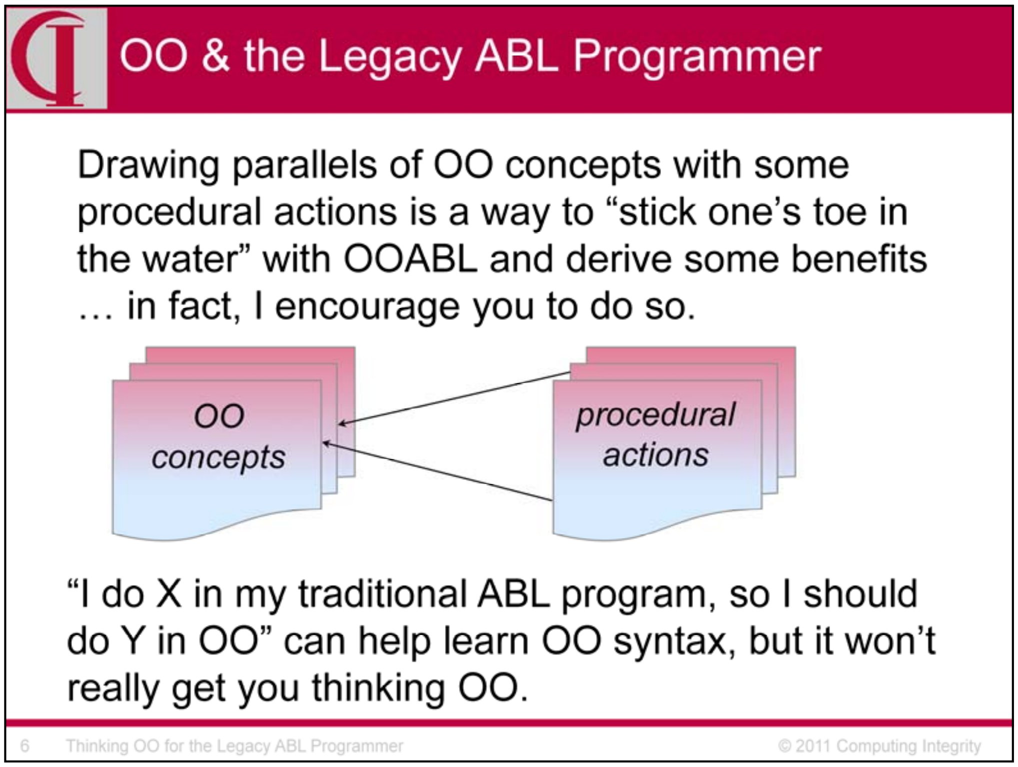
- Enhance quality
- Encapsulate logic in easily tested units.
- Promotes code re-use
- Easy and well-controlled extensibility
- Reduces production error rates
- More maintainable systems
- Reduces large problems to simpler components
- More natural relationship to problem space

5 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

It is believed that the OO paradigm improves quality by

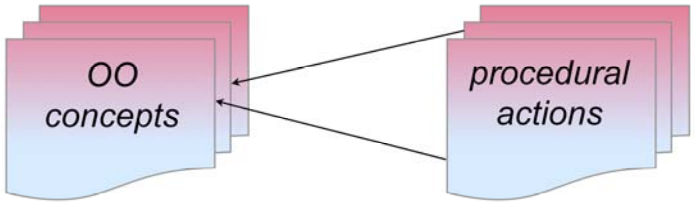
- encapsulating logic in easily tested units,
- promoting code re-use,
- providing a mechanism for easy and well-controlled extensibility,
- reducing production error rates by catching errors at compilation,
- creating a more maintainable system,
- reducing large problems to simpler components, and
- providing a more natural relationship between code and the real world.

I realize that sounds like a pretty extraordinary set of expectations, but it is all achievable if one follows the best OO practice.



OO & the Legacy ABL Programmer

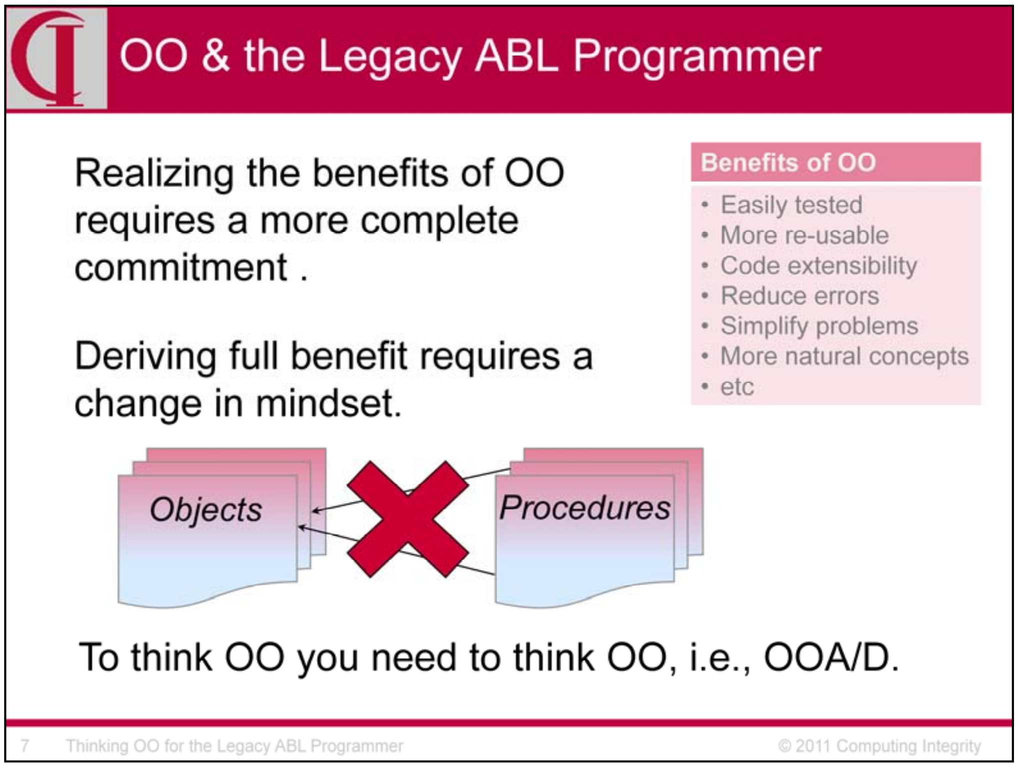
Drawing parallels of OO concepts with some procedural actions is a way to “stick one’s toe in the water” with OOABL and derive some benefits … in fact, I encourage you to do so.



“I do X in my traditional ABL program, so I should do Y in OO” can help learn OO syntax, but it won’t really get you thinking OO.

I want to be very clear that I am not discouraging anyone from using OO in a limited way in their application if that is all that can be justified or managed right now. It can help and provide some very nice solutions. Moreover, it can encourage people to do more of it.

But, one shouldn’t think that just because one is using a little OO here and there and is becoming comfortable with OO syntax, that one has achieved an OO mindset. It is that mindset, more than the syntax, which leads to the benefits which I have described.



OO & the Legacy ABL Programmer

Realizing the benefits of OO requires a more complete commitment .

Deriving full benefit requires a change in mindset.

Benefits of OO

- Easily tested
- More re-usable
- Code extensibility
- Reduce errors
- Simplify problems
- More natural concepts
- etc

Objects
Procedures

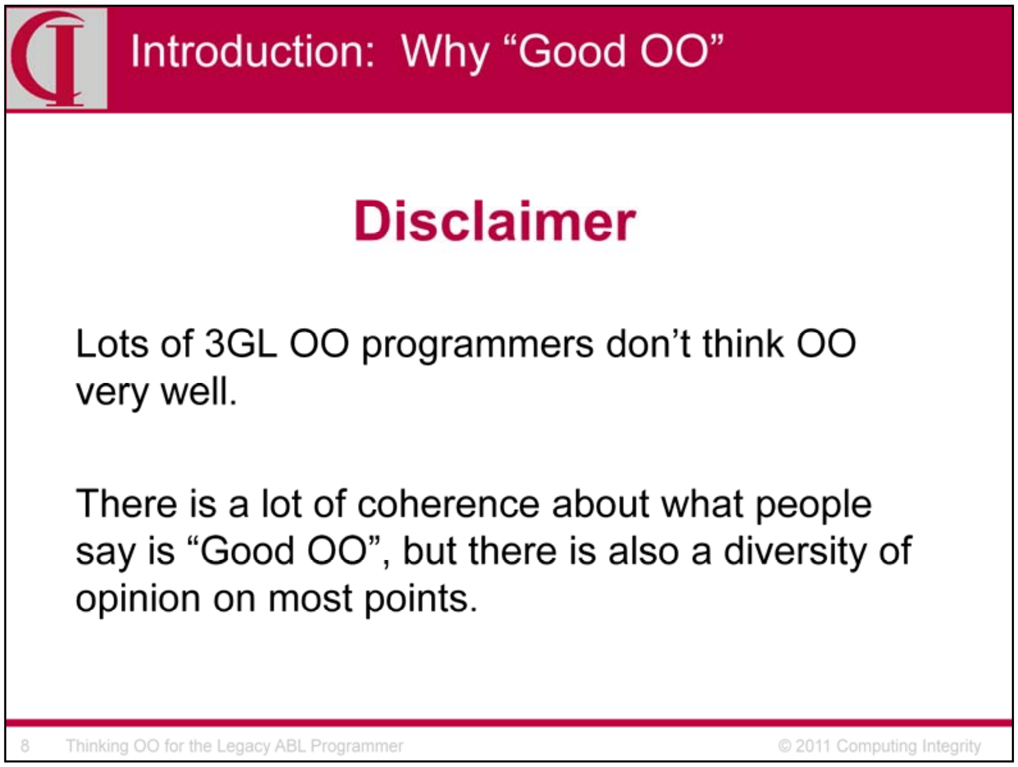
To think OO you need to think OO, i.e., OOA/D.

7 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

In manuals and other presentations, there are often comparisons provided in which a programming technique used in legacy ABL programs is compared to a OO programming technique. That can be helpful for learning syntax and recognizing when to use particular constructs, but it doesn't change the way you think about the analysis and design of programs.

Thinking OO really means understanding OOA/D, Object-Oriented Analysis and Design. It is a way of thinking that starts with the problem space, decomposes it into coherent units, connects those units, and from this structure builds a computational space which mirrors the problem space. What that means may not be very clear right now, but hopefully it will be a little more clear in an hour. It is a very different way to think about your programs than you have been used to in traditional ABL … or perhaps other languages in which you have written.

One of the ironies of modern computing is that most university programs teach people programming in OO languages, but they don't teach them OOA/D. Consequently, there is an awful lot of bad OO code in 3GL OO programs and systems. Yes, OO syntax and structure may help provide a better structure, at least some of the time, but without a thorough OO mindset, good results are almost accidental.



Introduction: Why "Good OO"

Disclaimer

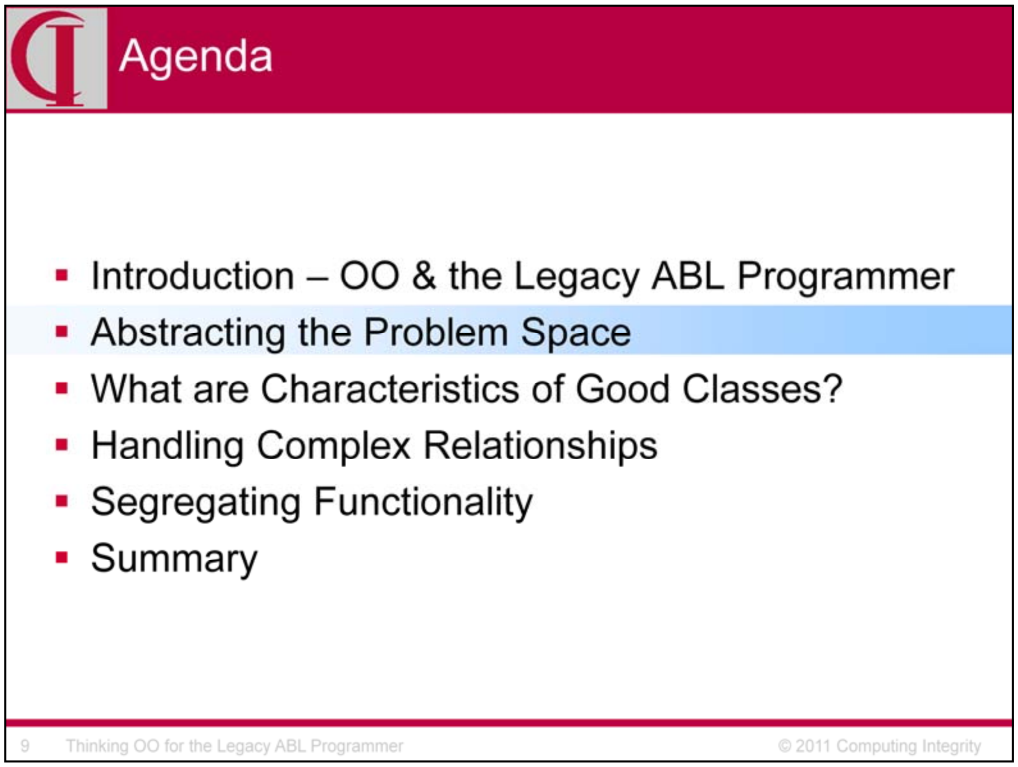
Lots of 3GL OO programmers don't think OO very well.

There is a lot of coherence about what people say is "Good OO", but there is also a diversity of opinion on most points.

8 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Before going any further, let me note that while I talk as if "Good OO" were one set of rules about which everyone agreed, in practice there is a diversity of opinion on a great many, if not most points. In some cases, this diversity is even great enough to include polar opposite views. Nevertheless, there is a lot of coherence in opinion, especially if one chooses one's mentors carefully, so I think it is justified to consider "Good OO" as a meaningful reference.

Let me also note that there is a lot of variation in how well these rules are observed. There are many people aware of these rules who somehow "forget" to apply the rule to their own work, particularly if they are experienced. There is some tendency to think that rules are for new people who don't know any better, but that those who have a track record don't need to keep checking their work.

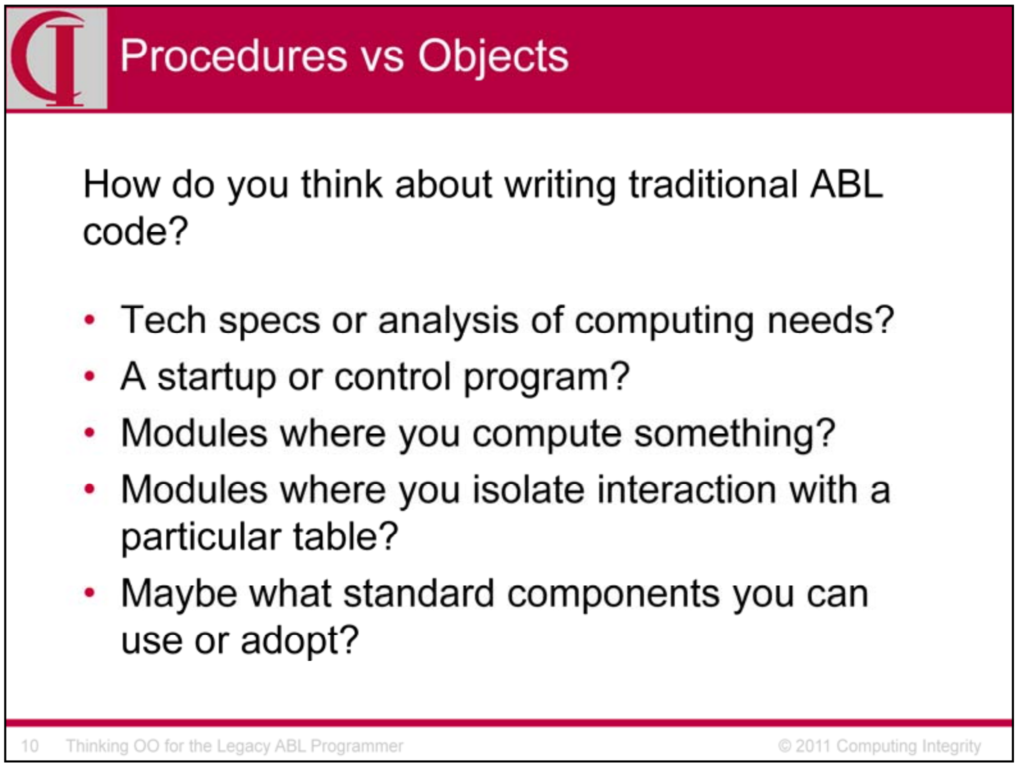


Agenda

- Introduction – OO & the Legacy ABL Programmer
- Abstracting the Problem Space
- What are Characteristics of Good Classes?
- Handling Complex Relationships
- Segregating Functionality
- Summary

9 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Let’s start our exploration of OO thinking by looking at the building blocks of our applications.


Procedures vs Objects

How do you think about writing traditional ABL code?

- Tech specs or analysis of computing needs?
- A startup or control program?
- Modules where you compute something?
- Modules where you isolate interaction with a particular table?
- Maybe what standard components you can use or adopt?

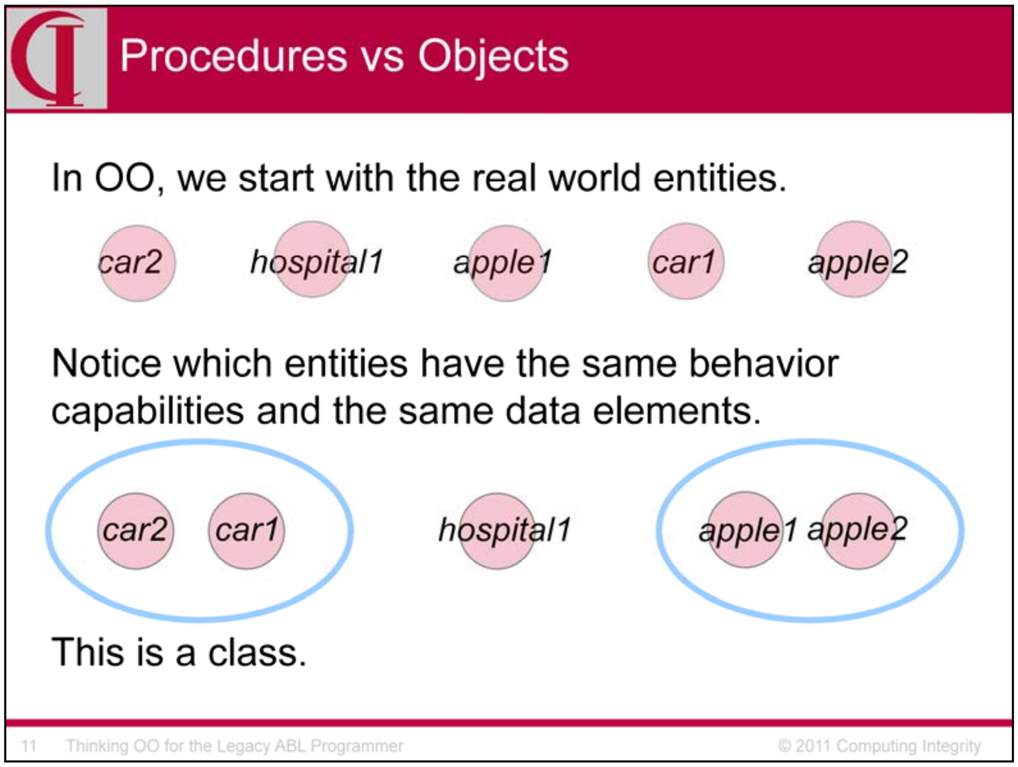
10 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

What do you think about when you set about writing some new ABL function in a legacy environment. Probably, you start either with some technical specifications prepared by someone else or you have a vague request from a user. If the user request, the first thing you will do is probably try to turn the user request into a technical specification, i.e., in terms of what tables you need information from, which ones you will post to, what algorithms are involved, etc.

Then, you might start at the beginning, i.e., defining the top level program which you will execute to run the program. Some of you will just start writing. Others might create a general structure first and then start creating modules to fill in that structure. You might think about a particular computation that you need to do and write a module for that. Perhaps you have learned good habits about isolating interaction with each table in its own internal procedure or program to minimize buffer scope and transaction size. Maybe you have some library components you can use.

All of this sounds perfectly sensible … but, it is all about computation. At the earliest possible opportunity one has translated the problem as seen by the user into a computational solution and all of the rest of the thinking is about the structure of that computational solution.

But, hey, one is writing a program … what else would one do except think about the computational solution?



Procedures vs Objects

In OO, we start with the real world entities.

car2
hospital1
apple1
car1
apple2

Notice which entities have the same behavior capabilities and the same data elements.

car2
car1
hospital1
apple1
apple2

This is a class.

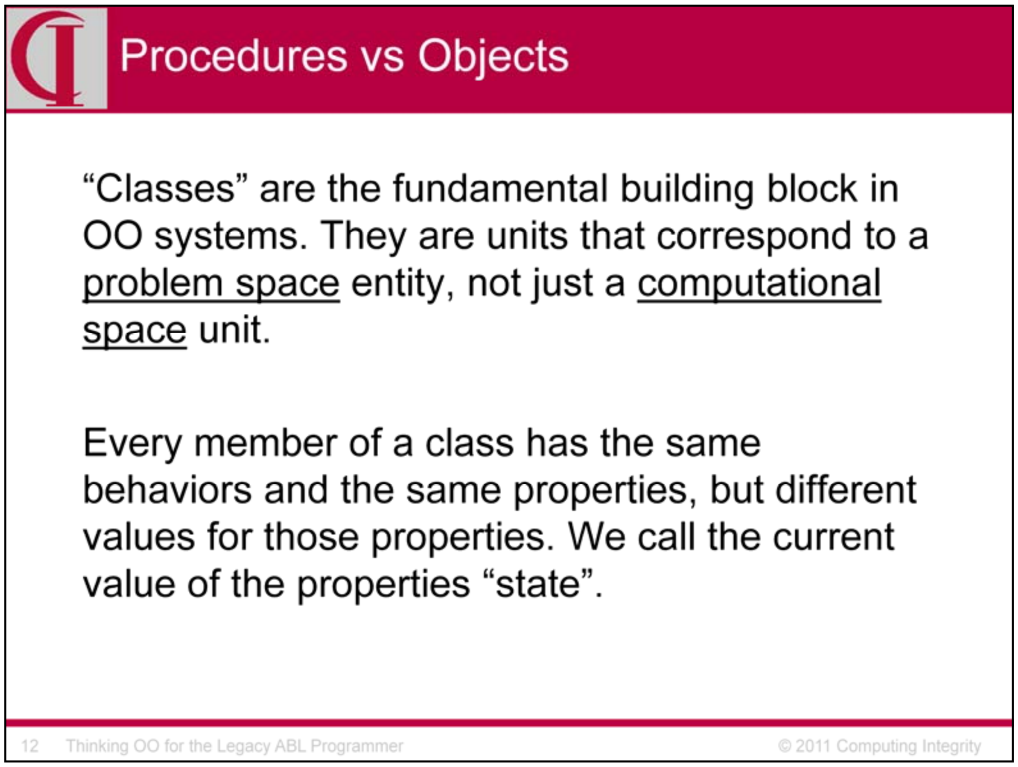
So, if the steps one would take in writing traditional ABL are sensible … what does one do if one is doing OO … *good* OO.

Let's step back a bit here from what one might do tasked with creating a single new function because working on an single function assumes that there is a whole lot of background already in place. Of course, in the case of adding to a legacy application, where there is a lot of code, but little analysis or model behind that code, there may be little context and one might need to start one's analysis from scratch if one is making a big enough change.

In looking at a problem area, the first thing one is going to do is to identify the "things" in that area. "Things" are the entities which are mentioned in the problem description and the requirements. If the problem area is order entry, the things are orders, all the possible orders that might flow through the system. It the problem area is automotive sales then the things are cars and customers and salespeople.

Some of those things or entities have the same behavior capabilities and the same properties or data elements, i.e., one entity differs from another entity in the values of their properties, but not in which properties they have. E.g., one might have Joe's Eats, Frank's Fine Food, and Chez Joanie. All have a name and some characteristics relevant to the problem type like an address or a telephone number.

That is a class.



Procedures vs Objects

"Classes" are the fundamental building block in OO systems. They are units that correspond to a <u>problem space</u> entity, not just a <u>computational space</u> unit.

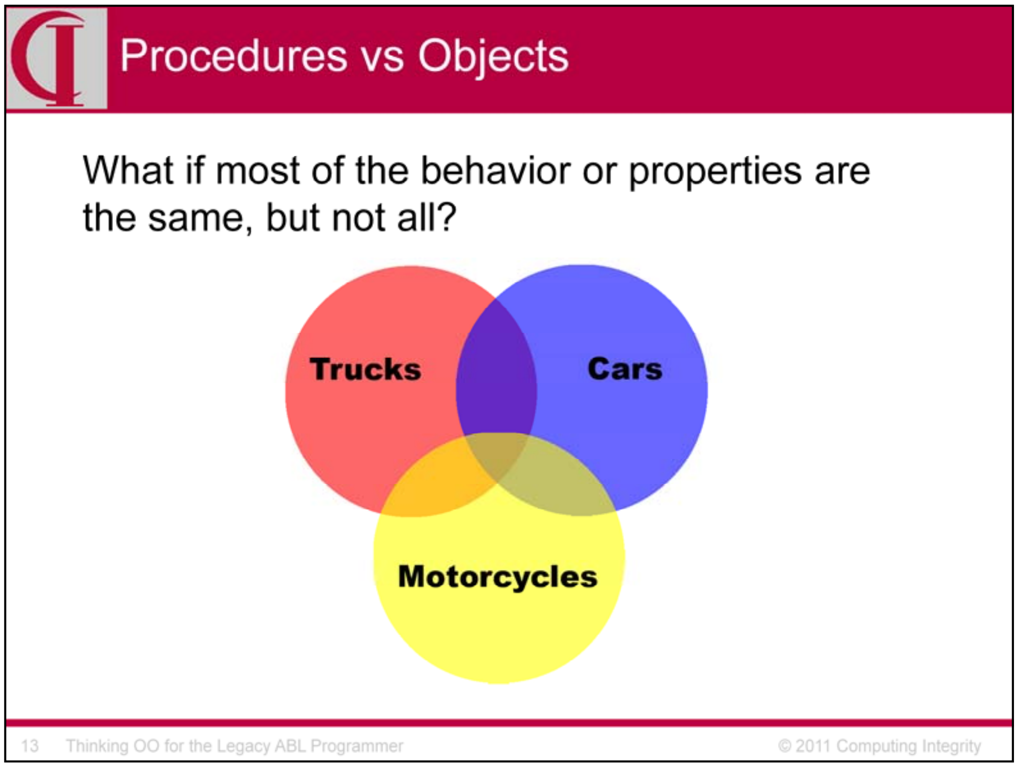
Every member of a class has the same behaviors and the same properties, but different values for those properties. We call the current value of the properties "state".

12 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

The fundamental building block in OO systems is a unit that corresponds directly to a problem space entity, not just a computational space unit.

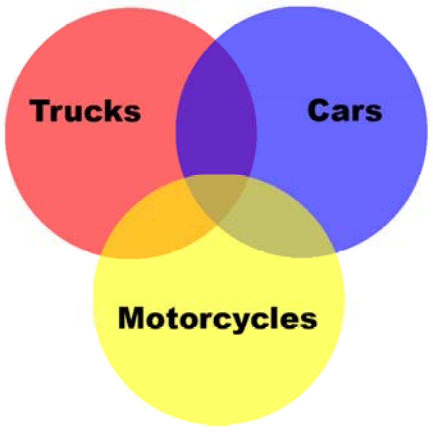
Every member of a class has the same behaviors and the same properties, but different values for those properties.

We call the current value of the properties "state".



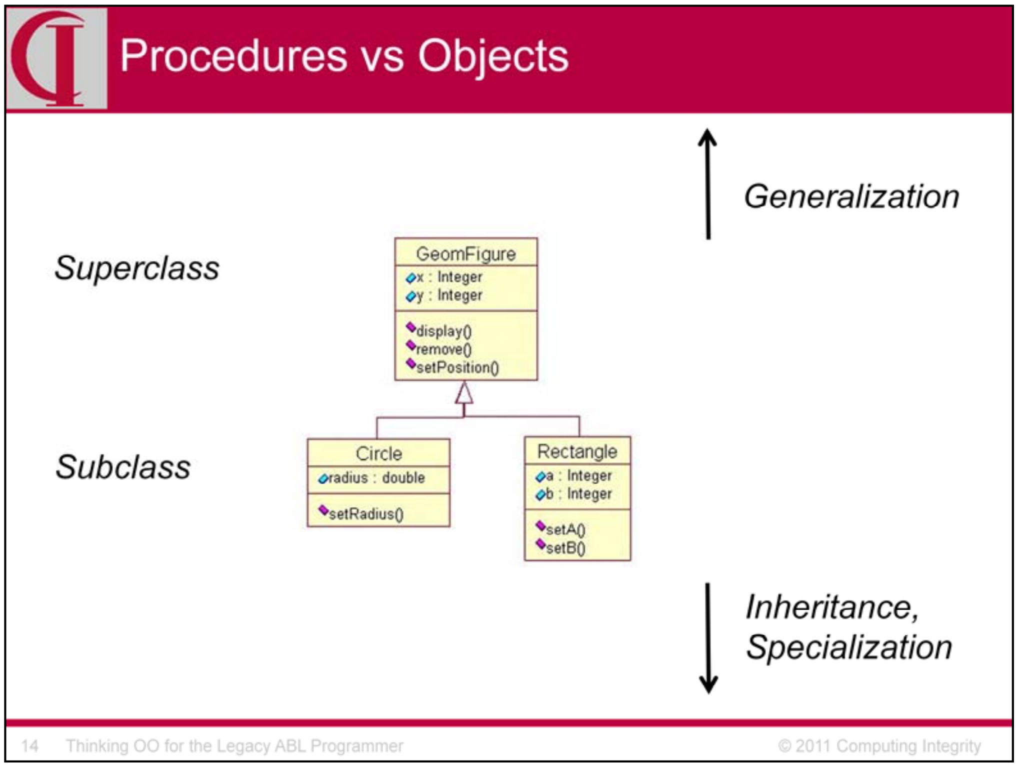
Procedures vs Objects

What if most of the behavior or properties are the same, but not all?



13 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

There are times when one will look at a collection of entities and find nice consistent classes, but ones which seem to overlap. I.e., all trucks might share the same behavior and properties and all cars might share the same behavior and properties, but some of the behavior and properties of trucks and cars might overlap. Note that this overlap or lack of overlap will depend entirely on the domain. In some domains, all of the behavior and properties of cars and trucks might be identical. In others, they might not overlap at all. But, in some, they can overlap, but only partially. Thus, here we have some properties and/or behavior which are shared by all trucks, cars, and motorcycles, some unique to trucks some unique to cars, and some unique to motorcycles.

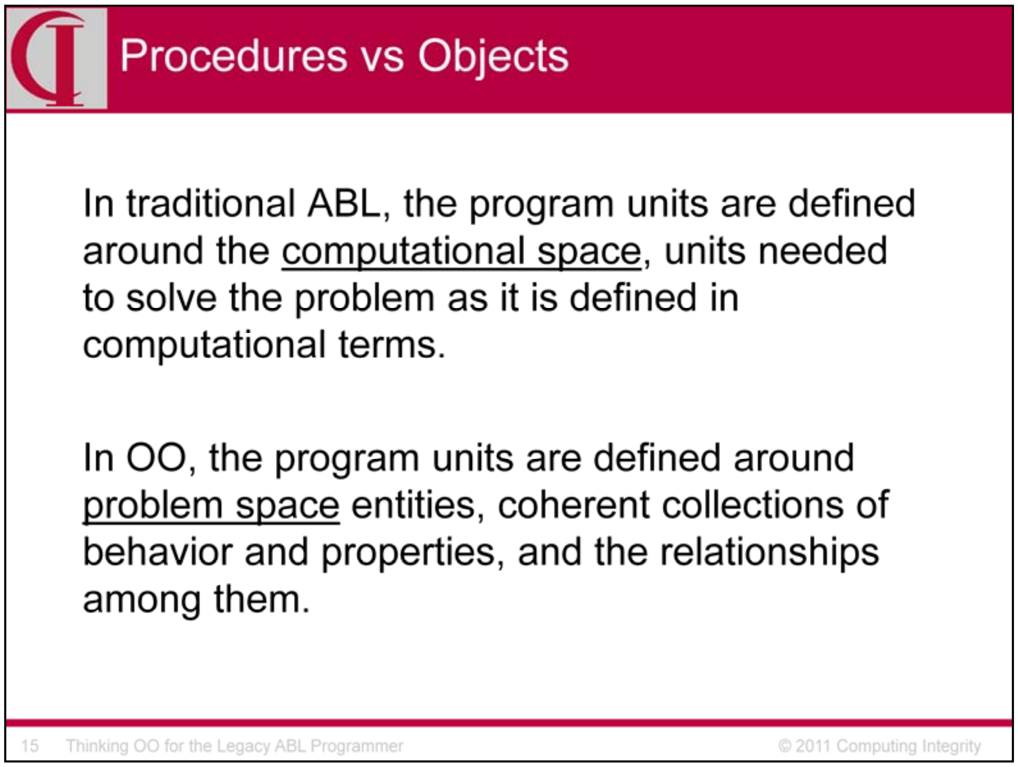


Procedures vs Objects

14 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Let's take a simple case in which we have identified a class of circles that have position, x and y, and radius and a class of rectangles which have position, x and y, and size a and b. Each property or property pair has a corresponding "setter" method and both classes have the behavior of display and remove. We could just have two separate classes here which happen to have some properties and behavior in common, but this runs counter to the desire in OO to have everything be in one place only. So, we define what is called a Superclass, here called GeomFigure and put the shared properties and behavior in it. Then, the two classes we originally identified "inherit" from that Superclass and become what we call Subclasses. Each Subclass has all of the properties and behavior of the Superclass plus its own properties and behavior, but none of the properties and behavior of other Subclasses.

We generally use the word Inheritance to describe this relationship from the top down and the world Generalization to describe it from the bottom up. Of course, this means that Generalization fits people doing good OOA//D and Inheritance tends to go with people who think they know how things ought to work. ☺ One also uses the term Specialization when considering the tree from the top down.



Procedures vs Objects

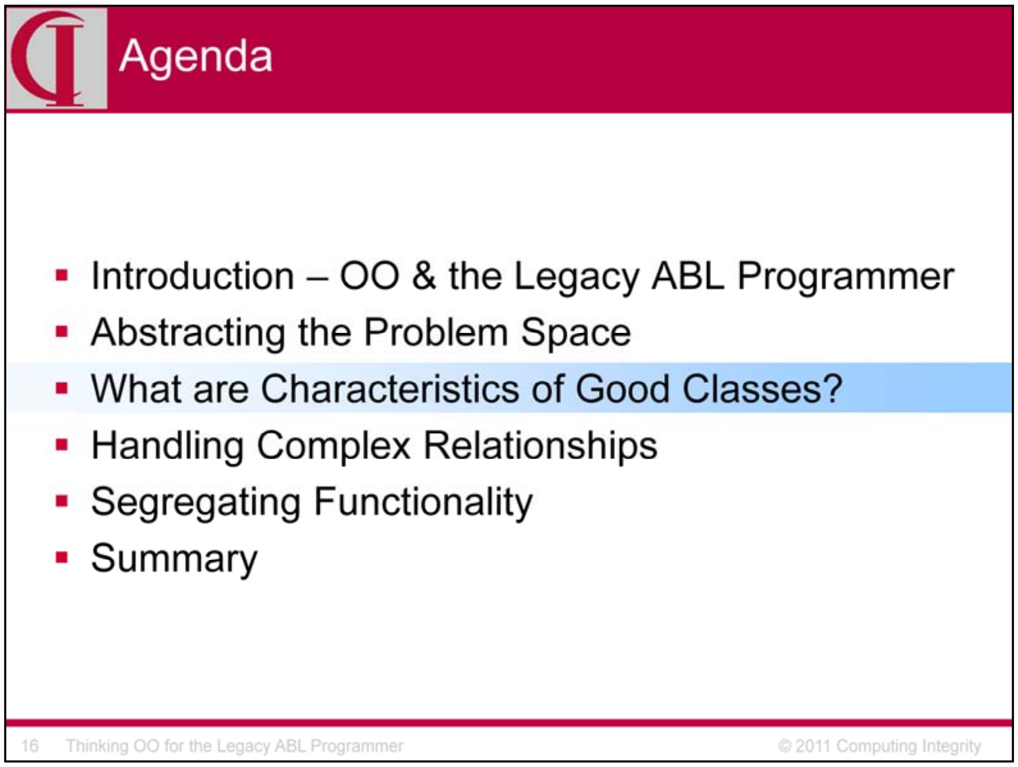
In traditional ABL, the program units are defined around the computational space, units needed to solve the problem as it is defined in computational terms.

In OO, the program units are defined around problem space entities, coherent collections of behavior and properties, and the relationships among them.

15 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

In traditional ABL, the program units are defined around the computational space, units needed to solve the problem as it is defined in computational terms.

In OO, the program units are defined around problem space entities, coherent collections of behavior and properties, and the relationships among them.

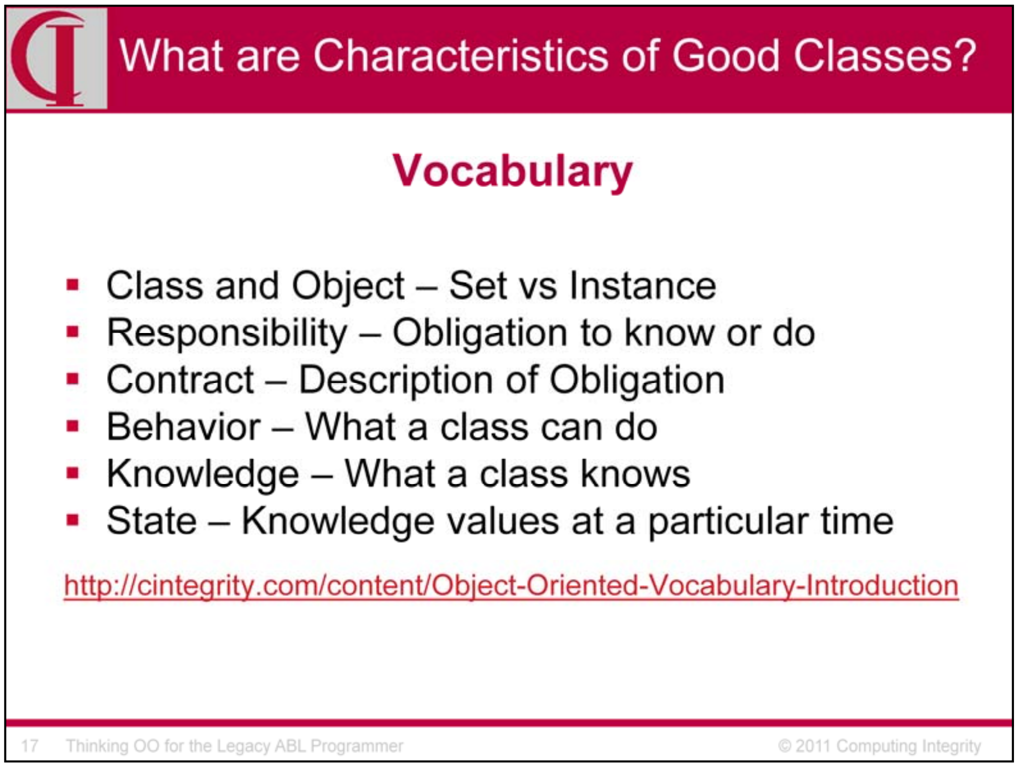


Agenda

- Introduction – OO & the Legacy ABL Programmer
- Abstracting the Problem Space
- What are Characteristics of Good Classes?
- Handling Complex Relationships
- Segregating Functionality
- Summary

16 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Let's now look at some of the characteristics of good classes so that we better understand what kind of units we are dealing with.



What are Characteristics of Good Classes?

Vocabulary

- Class and Object – Set vs Instance
- Responsibility – Obligation to know or do
- Contract – Description of Obligation
- Behavior – What a class can do
- Knowledge – What a class knows
- State – Knowledge values at a particular time

http://cintegrity.com/content/Object-Oriented-Vocabulary-Introduction

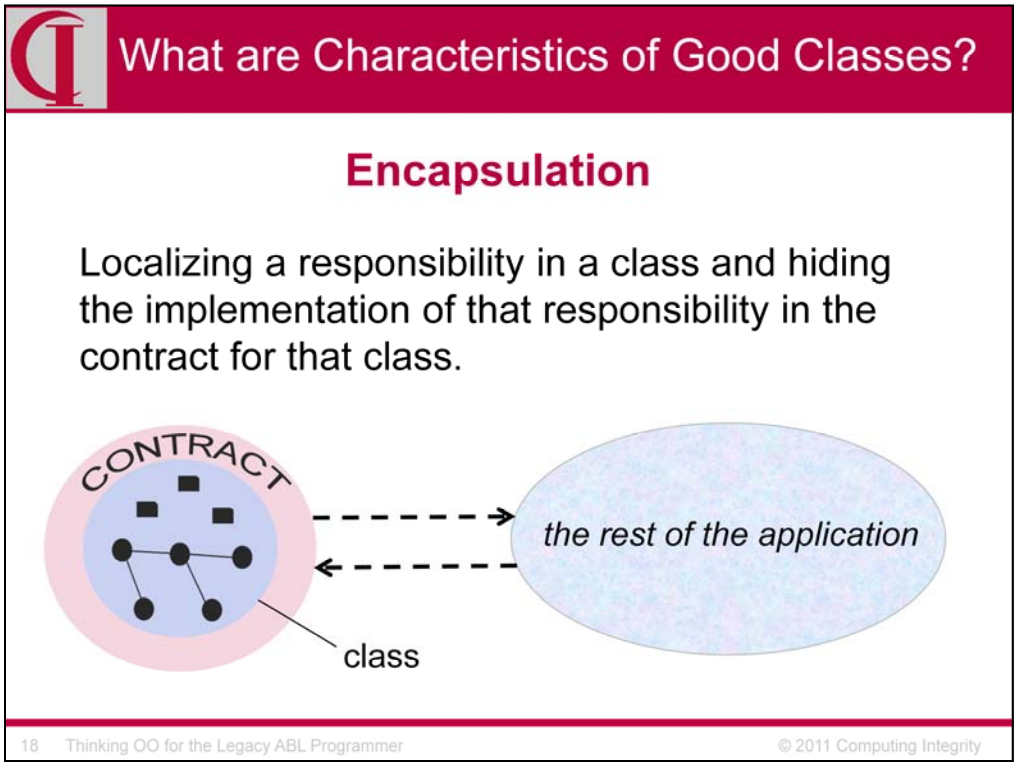
17 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

In the discussion which follow, I will be using a couple of familiar words that have special meanings in OO, so perhaps we should review these first. There is a vocabulary paper on my website which covers these and other terms.

First, there is the difference between class and object. A class is the name for a set of entities which have common behavior and elements of knowledge. An object is a specific instance of a class and thus has the state of a single entity that is a member of a class. Thus, an object corresponds directly to one entity in the problem space.

Responsibility is a common word in OO thinking and indicates a somewhat different way of thinking about code than we are used to in traditional ABL. A responsibility is an obligation to know or do something. The contract describes this obligation. In traditional ABL we often think in terms of the implementation, i.e., we know that a particular set of code does a certain thing. In OO, though, we very strongly want to think only of what we are expecting that unit of code to know or do, not at all how it does it. The contract should be the minimal required for it to perform that function.

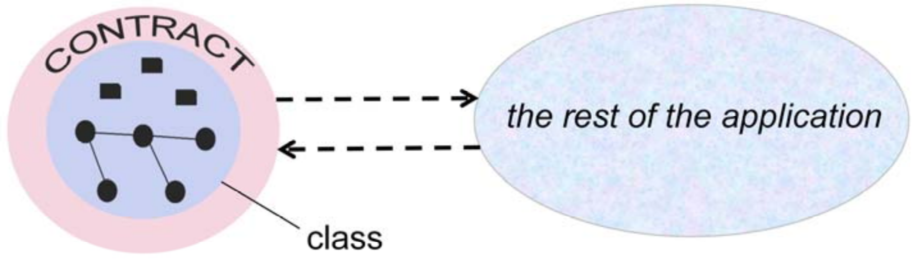
We tend to talk in OO about knowledge and behavior instead of data and code. The choice of both words reflect our focus on the responsibility for members of the class to do something, i.e., know something or exhibit certain behavior, rather than on how it is that they accomplish this. Often, knowledge will correspond to a simple variable in an object, but it may be computed or derived without us knowing any different from outside. One often distinguishes between knowledge – something that one counts on all members of a class knowing – and state – the particular value which a piece of knowledge has at a particular time in a particular object.



What are Characteristics of Good Classes?

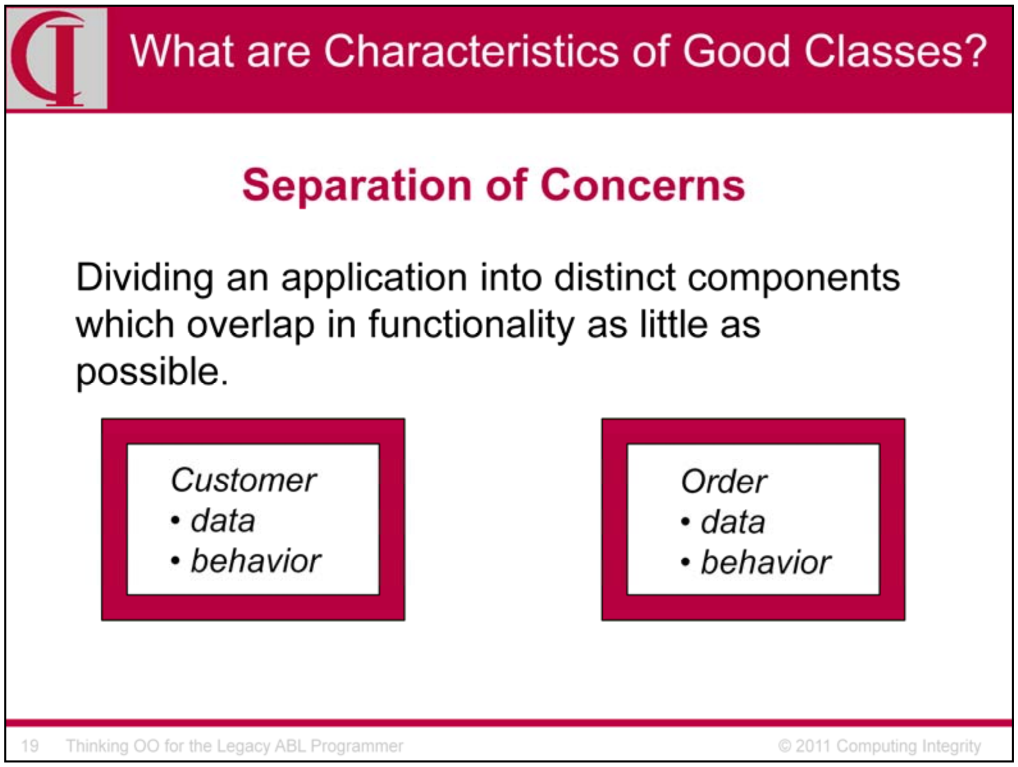
Encapsulation

Localizing a responsibility in a class and hiding the implementation of that responsibility in the contract for that class.



Encapsulation is an idea basic to OO thinking. Essentially, encapsulation means that a class, i.e., a program unit, should implement a single responsibility and it should hide the implementation of that responsibility in the contract for the class, i.e., expose only what needs to be exposed to fulfill the contract. A contract is simply the specification of the responsibility which the class agrees to fulfill.

The class should include all of the knowledge and behavior needed to implement the responsibility, but only expose what is necessary for interaction. That which is exposed is the contract for the class.



What are Characteristics of Good Classes?

Separation of Concerns

Dividing an application into distinct components which overlap in functionality as little as possible.

Customer
• data
• behavior
Order
• data
• behavior

Separation of Concerns, means dividing an application into components so that components overlap in functionality as little as possible. I.e., each piece of code has one concern or one responsibility and that responsibility is shared with no other code. This may seem like an obvious and simple idea, but it is central to the OO way of thinking.

I'm sure that some of you will be asking yourself, "doesn't everyone do this?" But, ask yourself …

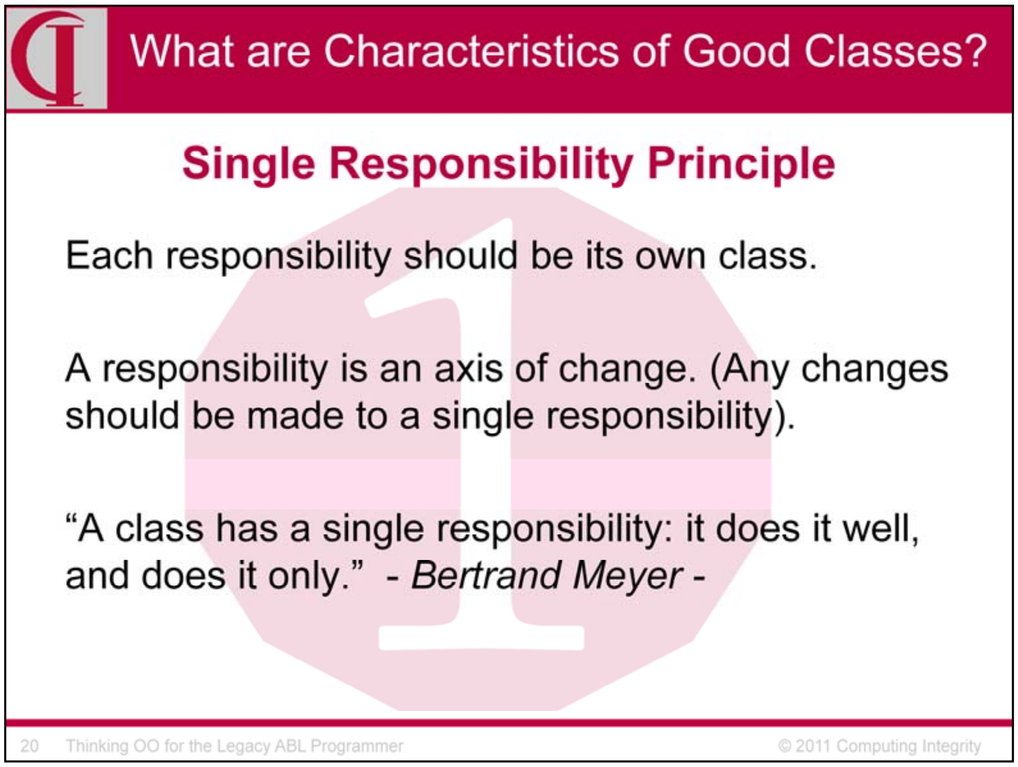
• How many different places is the customer table accessed in your application?

• If you wanted to see all of the logic which is ever applied to the data in the customer table, how many different places would you have to look?

• Do you even know where all those places are?

In most legacy ABL applications, functionality is spread broadly instead of being clustered in one place. Thus, not only is the functionality for a single entity spread, but the functionality in any given program unit is often a mixture of functionality for multiple entities. E.g., order entry code will contain some of the logic for dealing with Customers and Items. So we have the Order code is mixed with Customer and Item code and we have Customer code is spread around in many different modules, including order entry..

Separation of Concerns is thus very complementary to Encapsulation.



What are Characteristics of Good Classes?

Single Responsibility Principle

Each responsibility should be its own class.

A responsibility is an axis of change. (Any changes should be made to a single responsibility).

"A class has a single responsibility: it does it well, and does it only." - *Bertrand Meyer* -

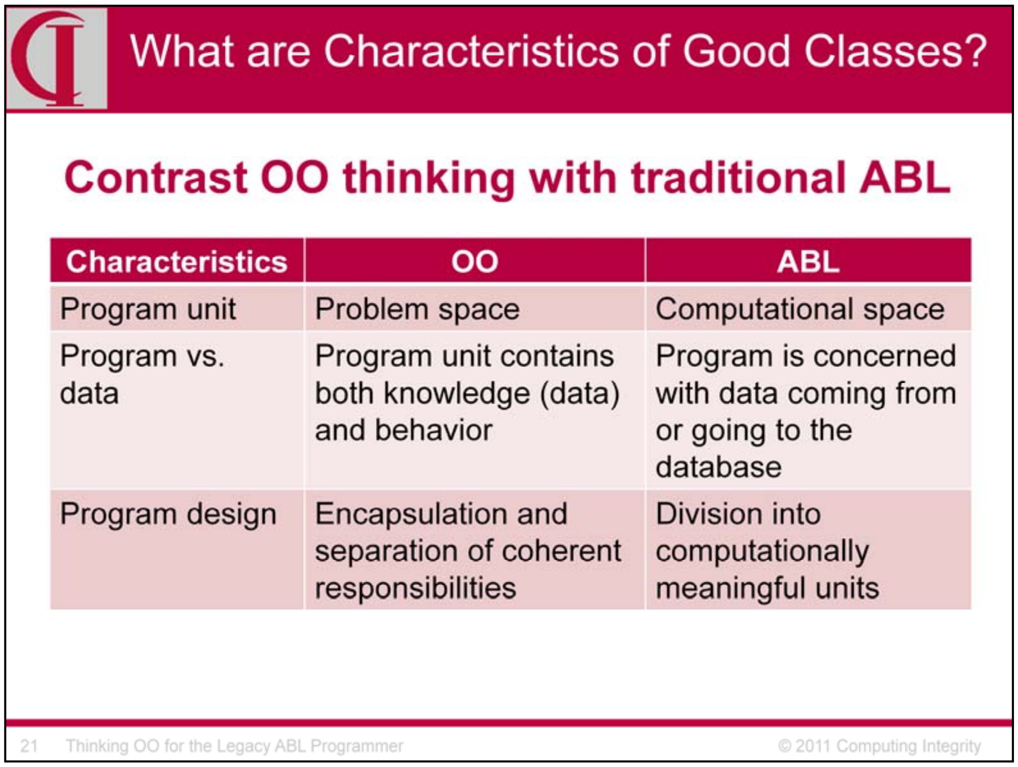
20 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

The Single Responsibility Principle can be seen as the complement of Separation of Concerns, i.e., that each responsibility should be its own class. A responsibility is an axis of change, so if there is a change in one responsibility, it should impact only one class except in special circumstances such a change in the public contract of a class.

This is a principle which can apply equally well to non-OO code. Sometimes, particularly with large, complex responsibilities, one will find all the code for a particular function in one place, referenced where it is needed. To do anything else would seem foolish. But, it is quite common for fairly large amounts of only slightly different code to be copied here and there in the application and it is very typical in traditional ABL applications for small bits of essentially identical code to appear numerous places in an application.

Take, for example, an application where customer addresses have a 5 digit zip code. A new need arises in one part of the application for a 9 digit zip. With an OO structure, we have one class responsible for storage which has to be modified to accommodate the additional data and one CustomerAddress class which handles providing that data to other components. If we preserve the contract which supplies the 5 digit zip and add a new contract which adds the additional four digits, then absolutely every place in the application that used to use only the 5 digit zip can continue to do so unchanged while the one component which needs the additional information can be modified to access the new contract. Absolutely minimal change with complete confidence and complete independence on how we actually store this information.

Compare that to having to review every access to customer address. Yes, we can minimize rework by storing the two parts separately, but then we are allowing usage to dictate storage. With the separation, storage can be whatever we choose because the contract is preserved.



What are Characteristics of Good Classes?

Contrast OO thinking with traditional ABL

Characteristics	OO	ABL
Program unit	Problem space	Computational space
Program vs. data	Program unit contains both knowledge (data) and behavior	Program is concerned with data coming from or going to the database
Program design	Encapsulation and separation of coherent responsibilities	Division into computationally meaningful units

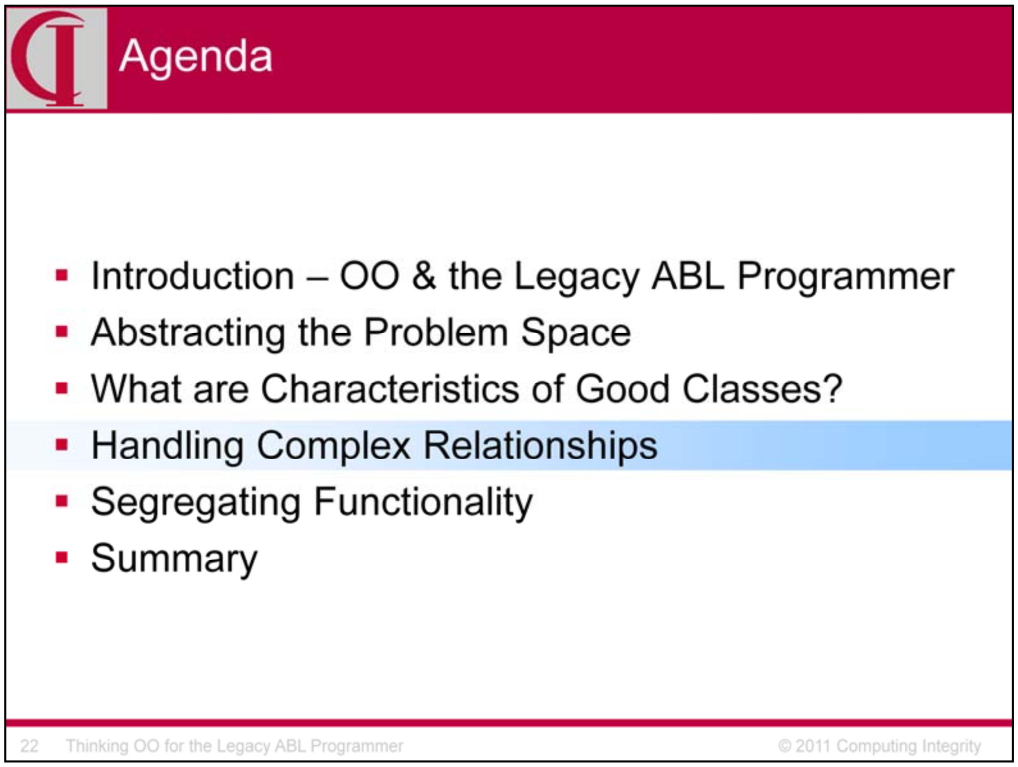
21 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Contrasting with traditional ABL:

Program unit corresponds to problem space unit, not a computational unit.

Program unit contains both knowledge (data) and behavior instead of thinking about data coming from or going to the database.

There is a strong emphasis on encapsulation and separation of coherent responsibilities rather than division into computationally meaningful units.

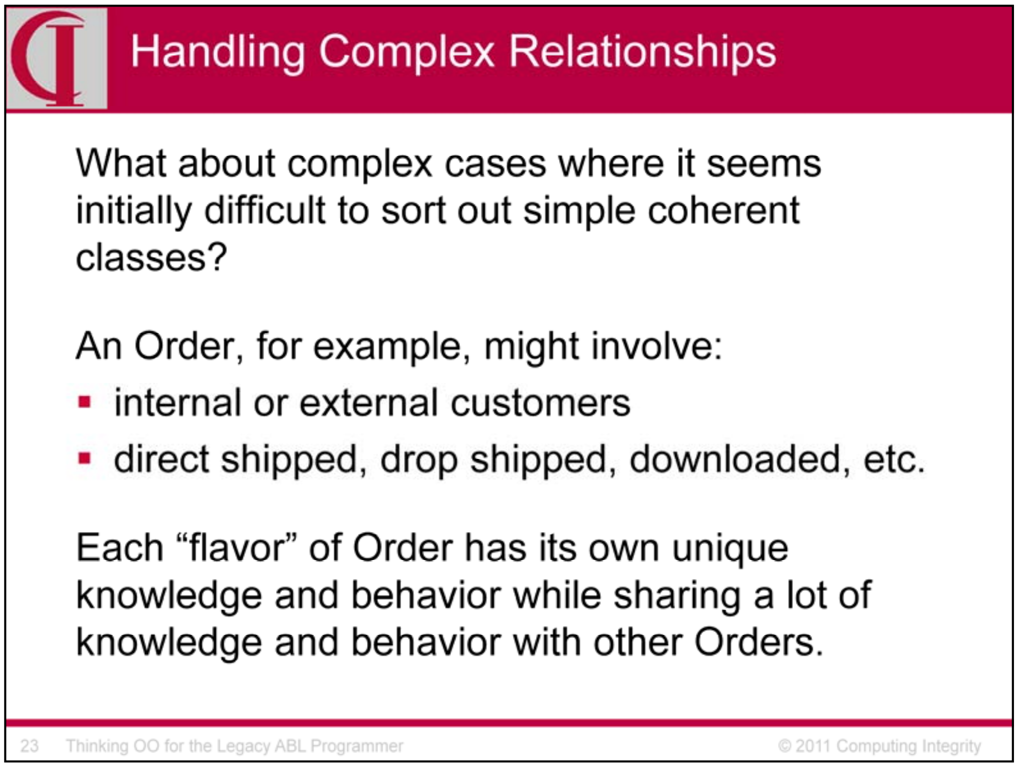


Agenda

- Introduction – OO & the Legacy ABL Programmer
- Abstracting the Problem Space
- What are Characteristics of Good Classes?
- Handling Complex Relationships
- Segregating Functionality
- Summary

22 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Having talked a bit about individual units of programming in the OO mindset, let’s look a bit at how they fit together.



Handling Complex Relationships

What about complex cases where it seems initially difficult to sort out simple coherent classes?

An Order, for example, might involve:

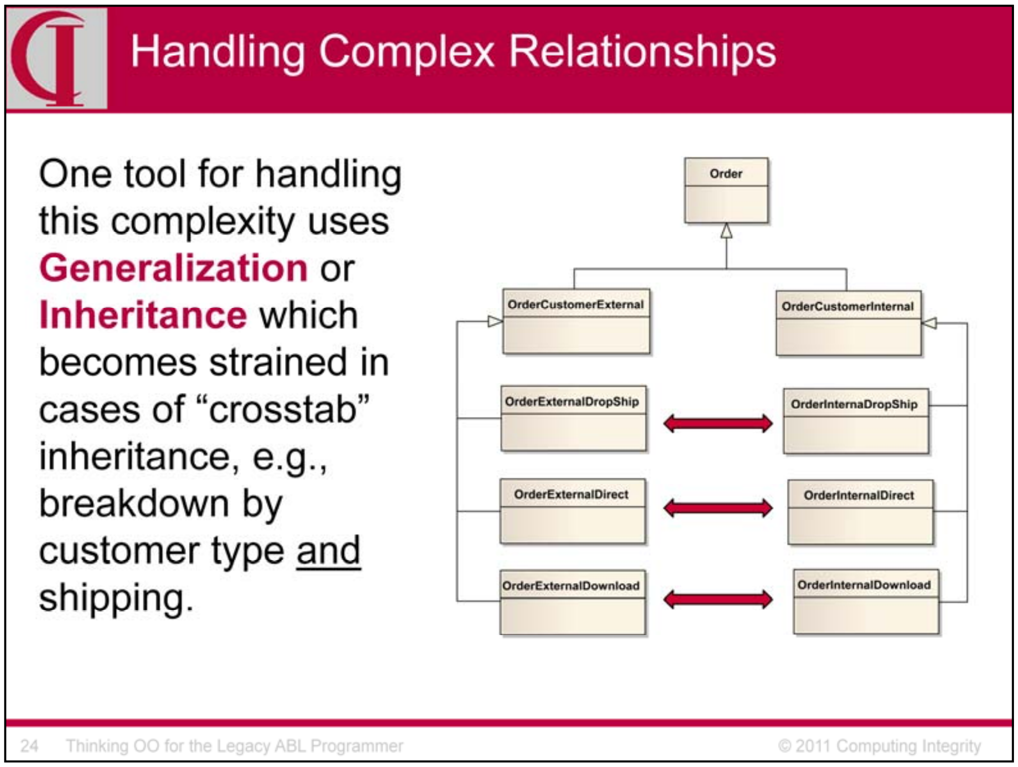
- internal or external customers
- direct shipped, drop shipped, downloaded, etc.

Each “flavor” of Order has its own unique knowledge and behavior while sharing a lot of knowledge and behavior with other Orders.

23 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

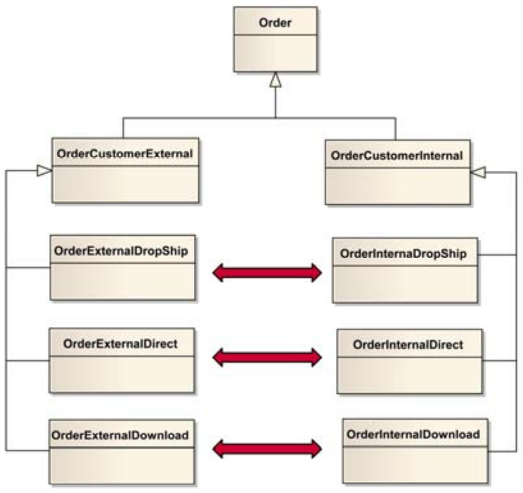
What about complex cases where it seems initially difficult to sort out simple coherent classes?

An Order, for example, might be for internal or external customers, might be direct shipped, drop shipped, downloaded, over the counter, among many other “flavors”. Each flavor has its own unique knowledge and behavior while sharing a lot of knowledge and behaviors with other Orders.



Handling Complex Relationships

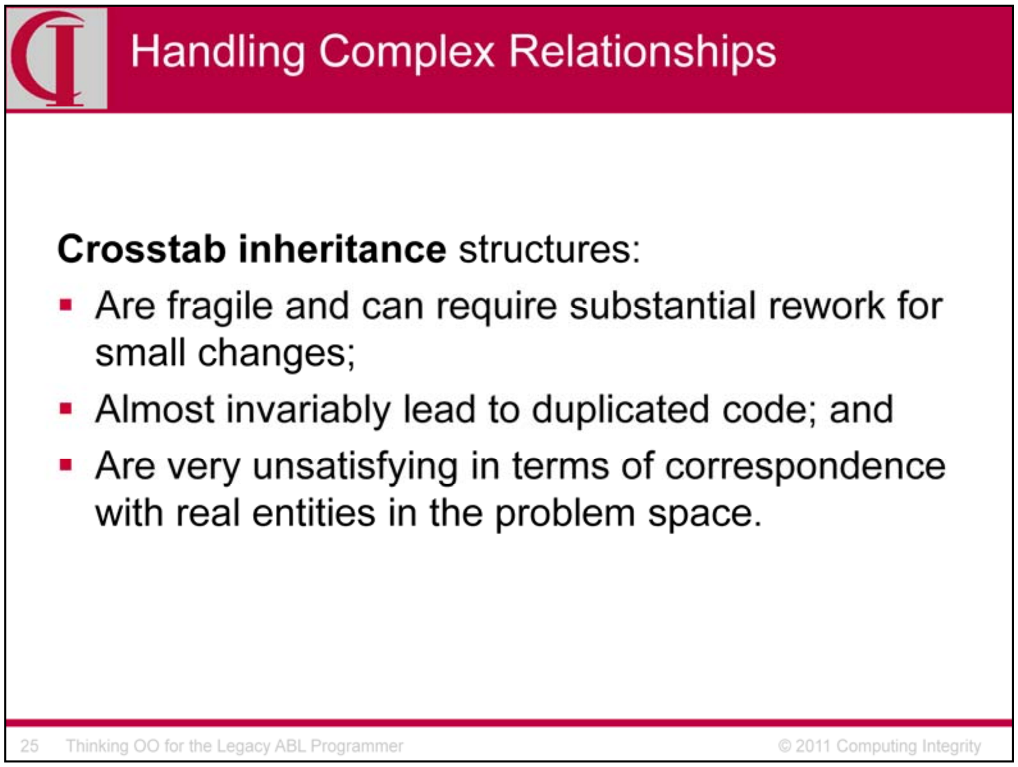
One tool for handling this complexity uses **Generalization** or **Inheritance** which becomes strained in cases of "crosstab" inheritance, e.g., breakdown by customer type and shipping.



24 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

We talked briefly about Generalization as one way to handle cases where we could identify multiple classes with some shared and some unique knowledge and behaviors. For many situations, that is just the right thing.

But, for cases like the Order example, we have a problem. Thinking from the top down for a moment, we could breakdown Orders into those for External Customers and those for Internal Customers, each with some unique knowledge and behavior. Or, we could break down Orders by shipping method – drop ship, direct ship, download, etc. But, in reality Orders include both of these breakdowns. Thus, we might have a drop ship order for an internal customer or a drop ship order for an external customer or a direct ship order for an internal order, etc. This is comparable to a table with customer type on one side and shipping method on the other side and the possible classes are all the cells. There is, of course, no reason this has to be limited to two dimensions either, so how do we handle a situation like this?



Handling Complex Relationships

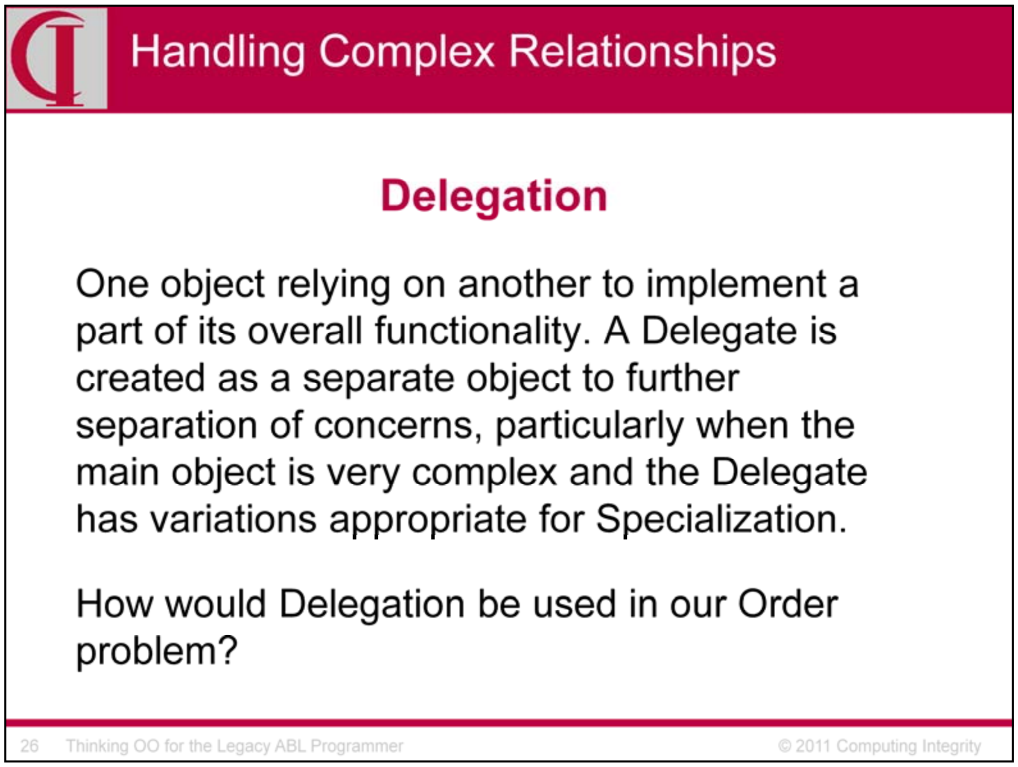
Crosstab inheritance structures:

- Are fragile and can require substantial rework for small changes;
- Almost invariably lead to duplicated code; and
- Are very unsatisfying in terms of correspondence with real entities in the problem space.

25 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

One can create actual crosstab inheritance structures but they:

- Are fragile and can require substantial rework for small changes;
- Almost invariably lead to duplicated code; and
- Are very unsatisfying in terms of correspondence with real entities in the problem space.



Handling Complex Relationships

Delegation

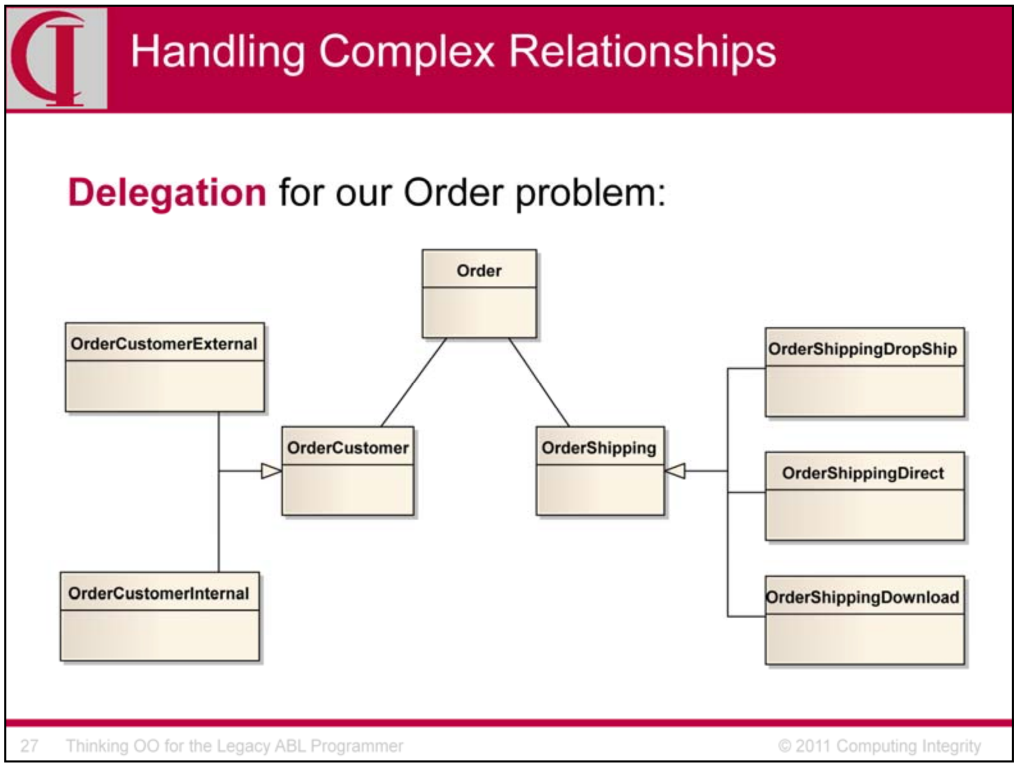
One object relying on another to implement a part of its overall functionality. A Delegate is created as a separate object to further separation of concerns, particularly when the main object is very complex and the Delegate has variations appropriate for Specialization.

How would Delegation be used in our Order problem?

26 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

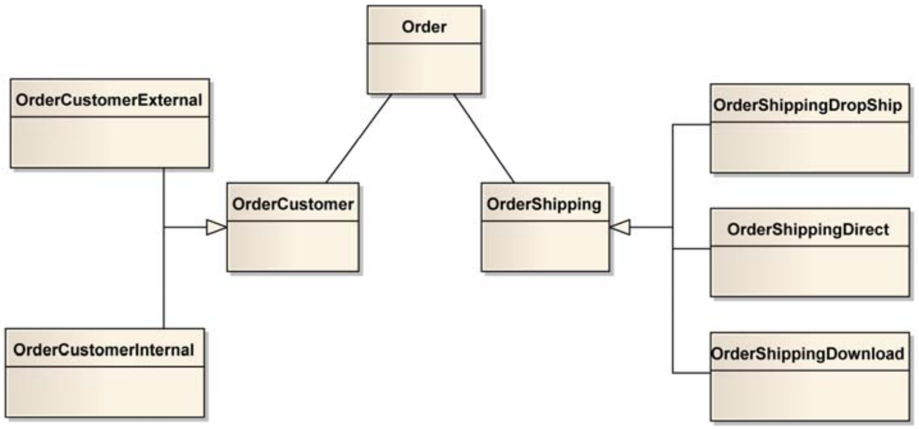
So, what do we do instead? The answer lies in another OO principle, Delegation.

Delegation is one object relying on another to implement a part of its overall functionality. A Delegate is created as a separate object to further separation of concerns, particularly when the main object is very complex and the Delegate has variations appropriate for Specialization. Once separated, the Delegate and the original Object are peers, each with their own separate sphere of Responsibility. It is important that this separation exist in the problem space, i.e., the Delegate should be an intrinsic decomposition recognizable in the problem space, not merely an arbitrary cluster of knowledge and behaviors. The container object, i.e., the one that has Delegated some of its behavior, may expose Methods by which that Delegated behavior can be accessed so that Clients of that object need not be aware of the Delegate.



Handling Complex Relationships

Delegation for our Order problem:



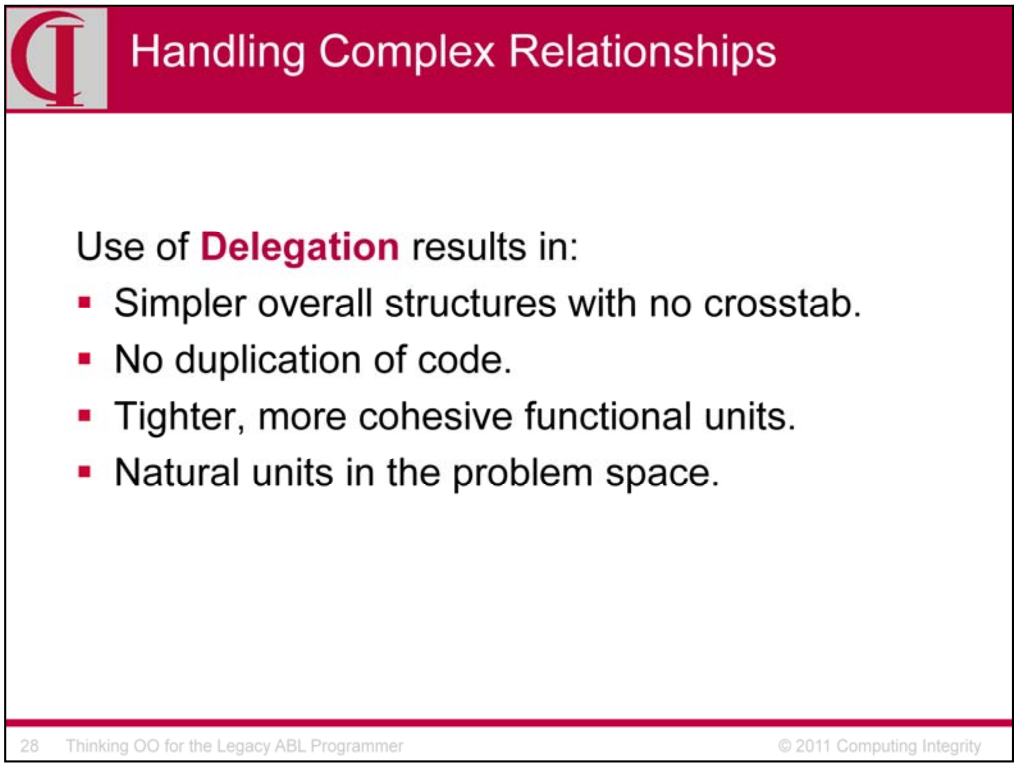
27 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

How would Delegation be used in our Order problem?

First, we might define an Order Customer superclass with Internal and External subclasses and move all responsibilities related to that area out of the Order.

Second, we might define an Order Shipping class with Drop Ship, Direct Ship, Download, etc. subclasses and move that responsibility out.

The result is that we have one Order Class, One OrderCustomer superclass with two or more subclasses, and one OrderShipping class with multiple subclasses and we have eliminated any problem with crosstab structure.



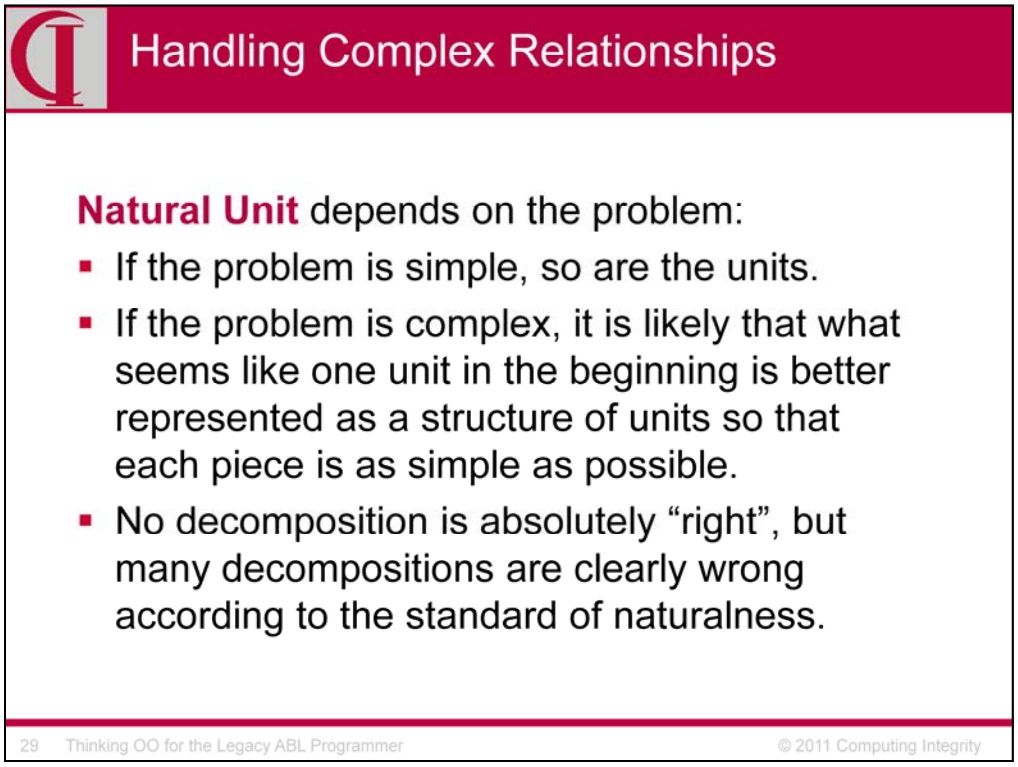
Handling Complex Relationships

Use of **Delegation** results in:

- Simpler overall structures with no crosstab.
- No duplication of code.
- Tighter, more cohesive functional units.
- Natural units in the problem space.

Use of Delegation results in:

- Simpler overall structures with no crosstab.
- No duplication of code.
- Tighter, more cohesive functional units.
- Natural units in the problem space.



Handling Complex Relationships

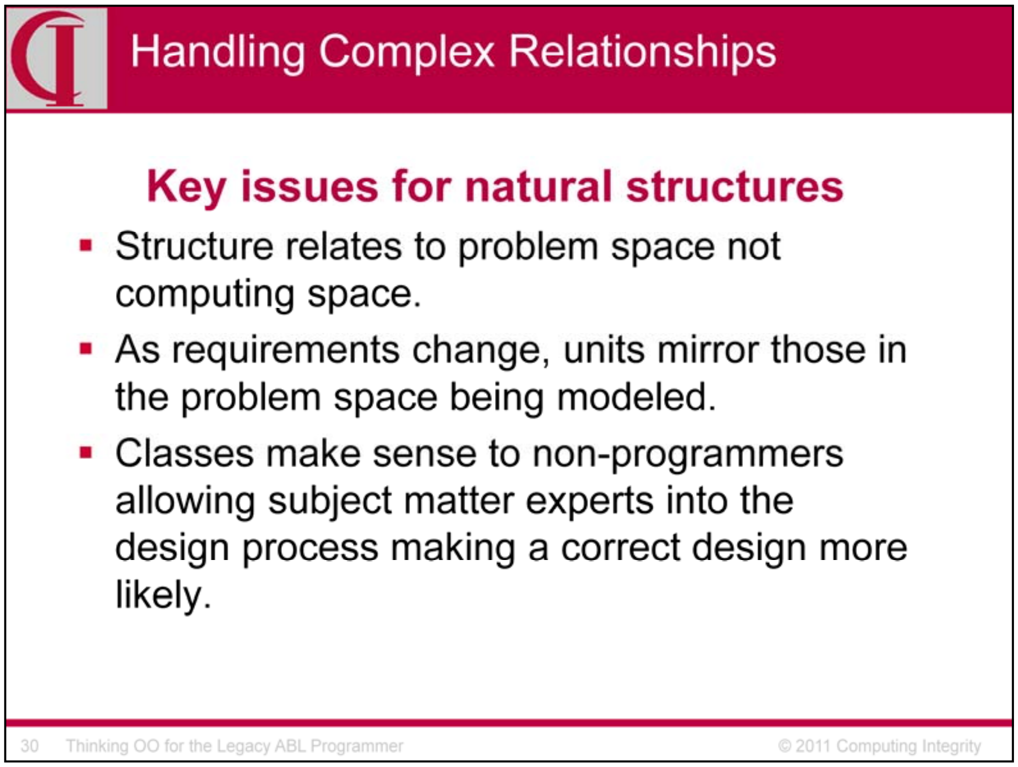
Natural Unit depends on the problem:

- If the problem is simple, so are the units.
- If the problem is complex, it is likely that what seems like one unit in the beginning is better represented as a structure of units so that each piece is as simple as possible.
- No decomposition is absolutely “right”, but many decompositions are clearly wrong according to the standard of naturalness.

29 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Previously we mentioned distilling a problem into “Natural Units”. “Natural Unit” depends on the problem:

- If the problem is simple, so are the units.
- If the problem is complex, it is likely that what seems like one unit in the beginning is better represented as a structure of units so that each piece is as simple as possible.
- No decomposition is absolutely “right”, but many decompositions are clearly wrong according to the standard of naturalness.



Handling Complex Relationships

Key issues for natural structures

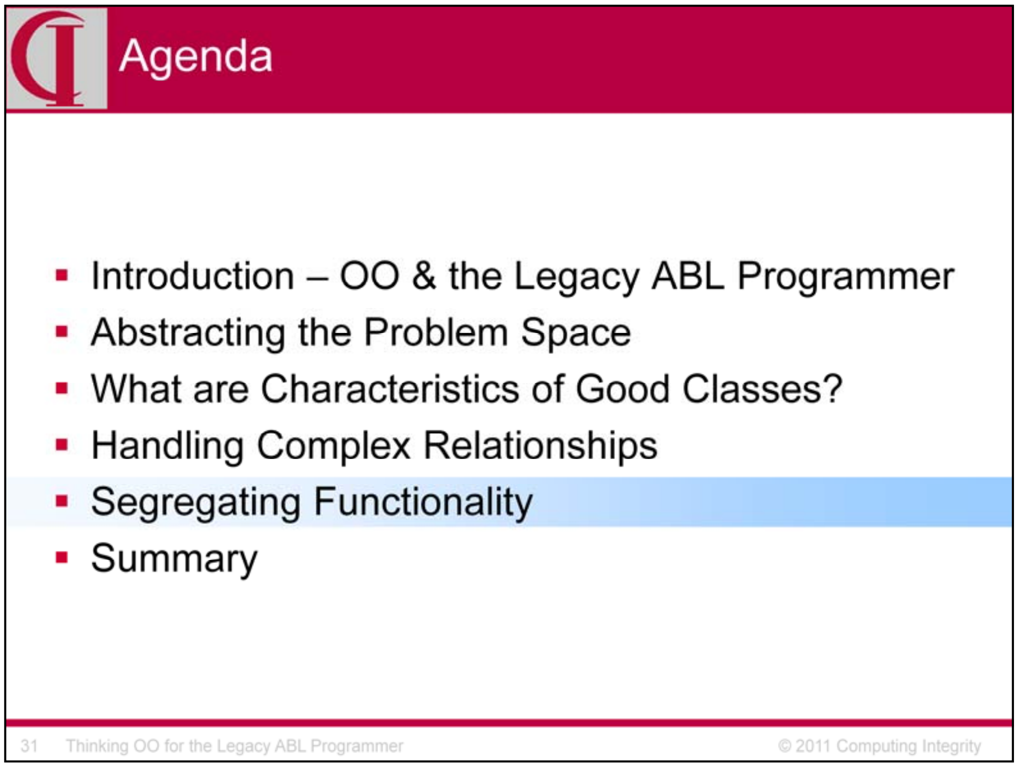
- Structure relates to problem space not computing space.
- As requirements change, units mirror those in the problem space being modeled.
- Classes make sense to non-programmers allowing subject matter experts into the design process making a correct design more likely.

30 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Key issues for here is that objects are structured according to the **natural structures** of the problem space, not according to the needs of the computing space.

As requirements change, everything has its place and the interaction of the system is easy to understand because the units mirror those in the problem space being modeled.

Moreover, since classes correspond to natural units in the problem space, many non-programmers can relate to and discuss OO specifications. This allows a closer involvement of subject matter experts in the design process, enhancing the likelihood of arriving at a correct design.

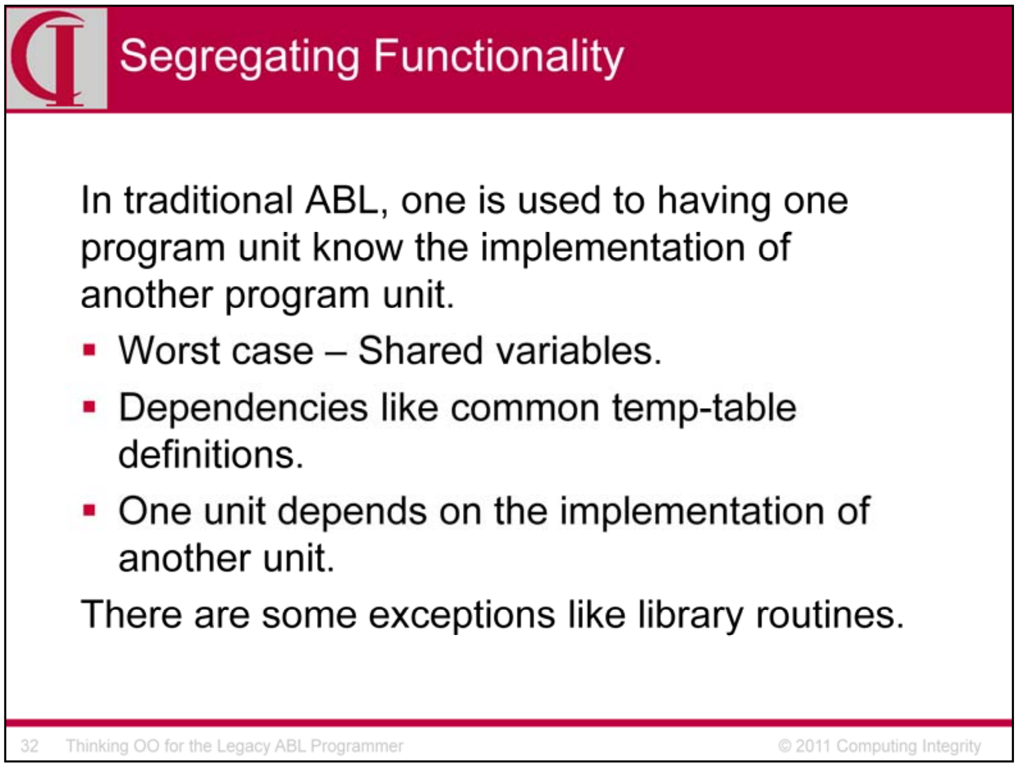


Agenda

- Introduction – OO & the Legacy ABL Programmer
- Abstracting the Problem Space
- What are Characteristics of Good Classes?
- Handling Complex Relationships
- Segregating Functionality
- Summary

31 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Having some idea of what classes should look like, let's consider some principles of how they should interact.



Segregating Functionality

In traditional ABL, one is used to having one program unit know the implementation of another program unit.

- Worst case – Shared variables.
- Dependencies like common temp-table definitions.
- One unit depends on the implementation of another unit.

There are some exceptions like library routines.

32 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

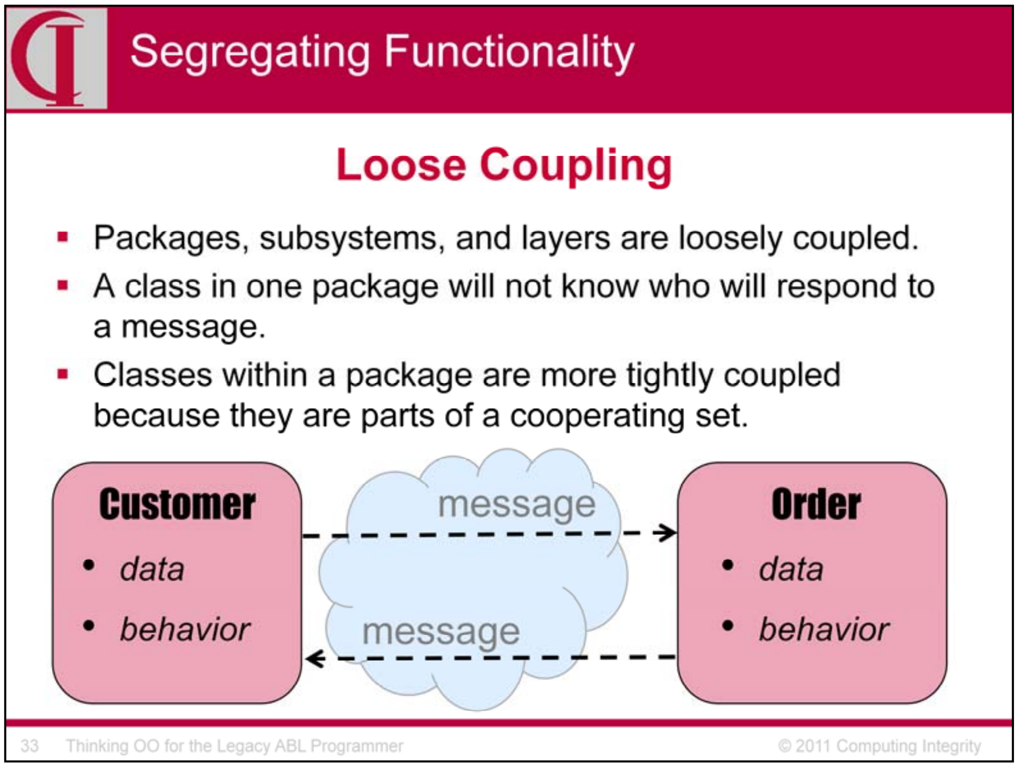
In traditional ABL, one is used to having one program unit know the implementation of another program unit.

Worst case – Shared variables.

Dependencies like common temp-table definitions.

One unit depends on the implementation of another unit.

There are some exceptions like library routines.



Segregating Functionality

Loose Coupling

- Packages, subsystems, and layers are loosely coupled.
- A class in one package will not know who will respond to a message.
- Classes within a package are more tightly coupled because they are parts of a cooperating set.

Customer
• data
• behavior
message
message
Order
• data
• behavior

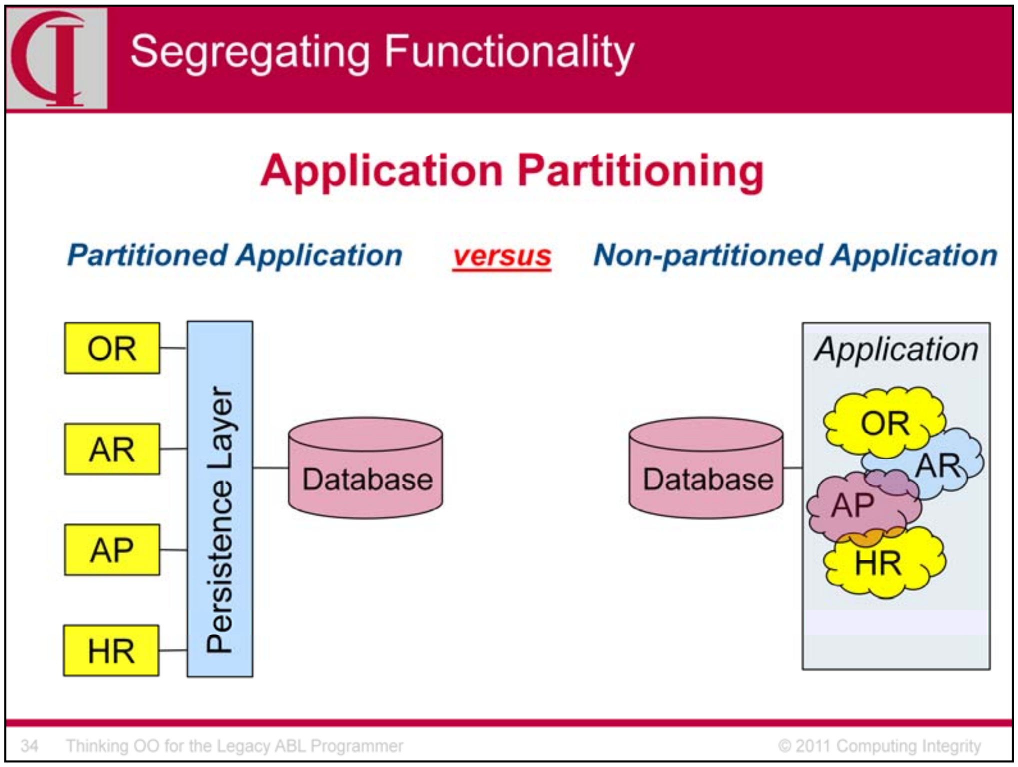
33 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

A consequence of connecting classes by the minimal contract necessary to provide for their interaction is that the farther apart two classes are, the less likely it is that they will even know about a class's complete contract, much less any particulars about its implementation. Thus, classes which are in different packages, subsystems, or layers will know very little about each other, even possibly not knowing which class in a package will even respond to a particular message. Thus, packages, subsystems, and layers are typically loosely coupled.

By comparison, since classes within a package are parts of a single cooperating set, they are likely to be aware of larger parts of the contract of other classes within the same package and are thus more tightly coupled. There are a number of specific principles which echo this general idea in the whitepaper on my web site.

Thus, one might have a printing subsystem composed of multiple cooperating classes which know about each other's contracts because they are mutually interdependent, but another package using the printing subsystem will be unaware of this structure and merely send print request messages to the printing subsystem.

This is a principle which could be equally relevant to traditional ABL, but is rarely observed.



Segregating Functionality

Application Partitioning

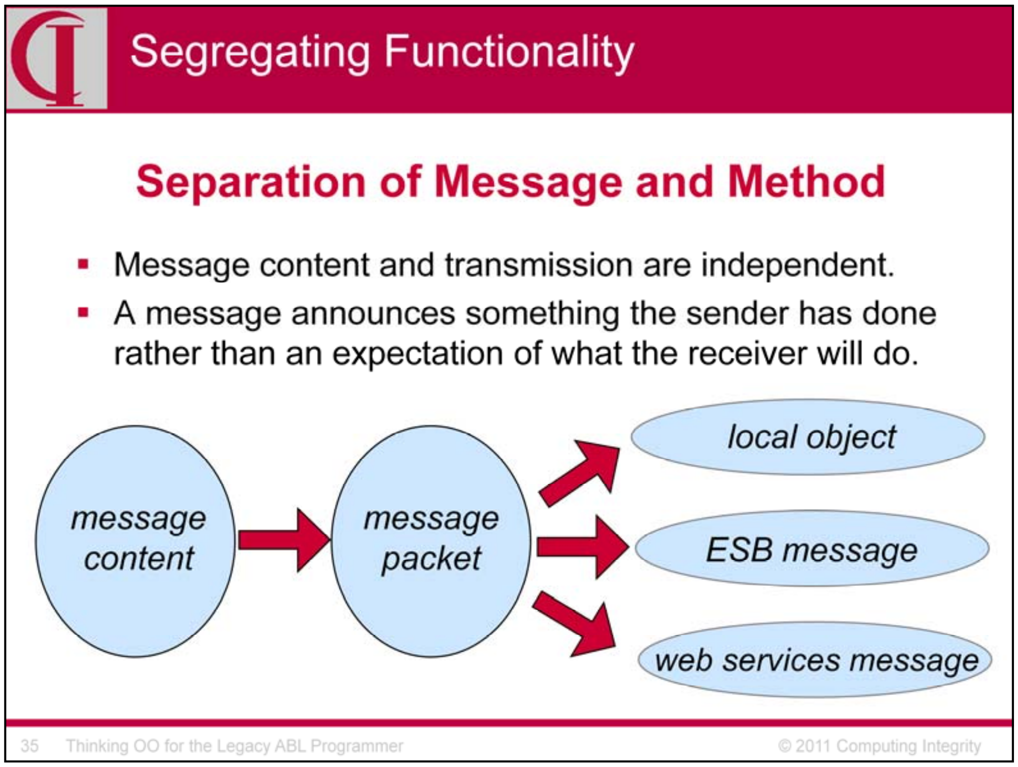
Partitioned Application ***versus*** *Non-partitioned Application*

34 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Just as one should separate responsibilities cleanly between classes, the whole application should be divided into clearly partitioned subsystems. Each subsystem should be cohesive within the subsystem and separate between subsystems in much the same way that classes are, but at a higher level.

Normally, one thinks of Separation of Concerns in a very local way, i.e., one class relative to another class, but the same principle applies at the package and application level. In particular, one should start in considering an application or suite of applications by dividing responsibilities among subsystems such that each subsystem is cohesive, i.e., represents a single high level area of responsibility, and responsibilities do not bleed across subsystems, i.e., they are cleanly separated.

While perhaps more natural to OO, this is a principle which can be applied to design in any language.



Segregating Functionality

Separation of Message and Method

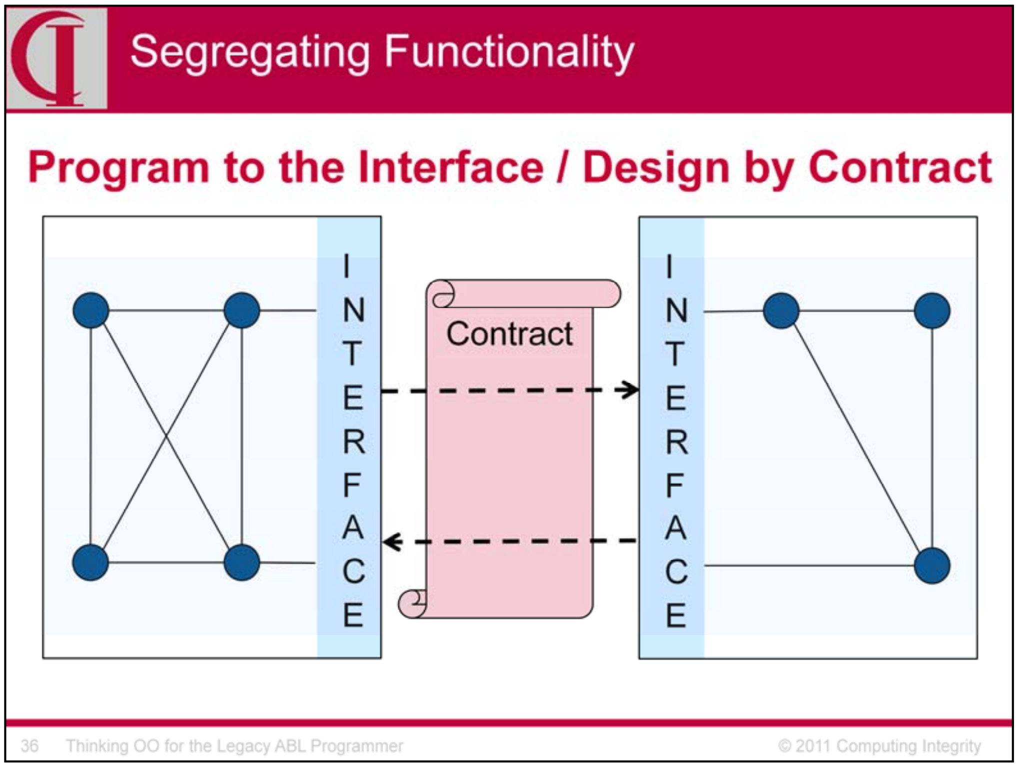
- Message content and transmission are independent.
- A message announces something the sender has done rather than an expectation of what the receiver will do.

message content
message packet
local object
ESB message
web services message

35 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

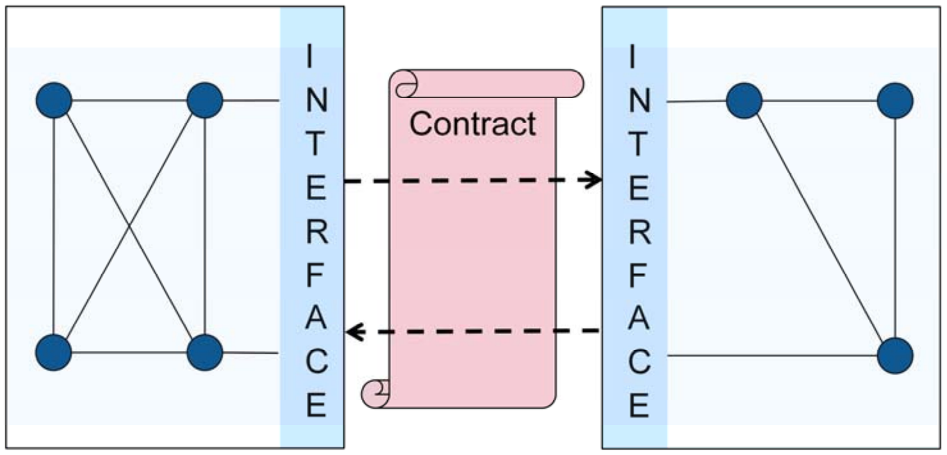
Perhaps one of the more important shifts of orientation between traditional ABL thinking and OO thinking is the move from “Do This” to “I’m Done Doing This”. “Do This” creates a dependency of the sender on the receiver which frequently cascades to create hierarchical structures of dependency which are fragile, difficult to understand, and hard to modify. By having the sender announce “I’m Done” to interested parties, no dependency is created beyond that of simply sending and receiving the message about the event. Everything else happens within the object, i.e., a single area of responsibility for which that object is the implementation. The resulting systems are far more stable and more easily modified because the scope of any change is limited to a single responsibility contained within a single class. They are also easier to understand because each class is a single responsibility which can be looked at in isolation.

There is nothing to prevent applying this principle to traditional ABL and it is, in fact, characteristic of publish and subscribe mechanisms, either locally or across the bus.



Segregating Functionality

Program to the Interface / Design by Contract

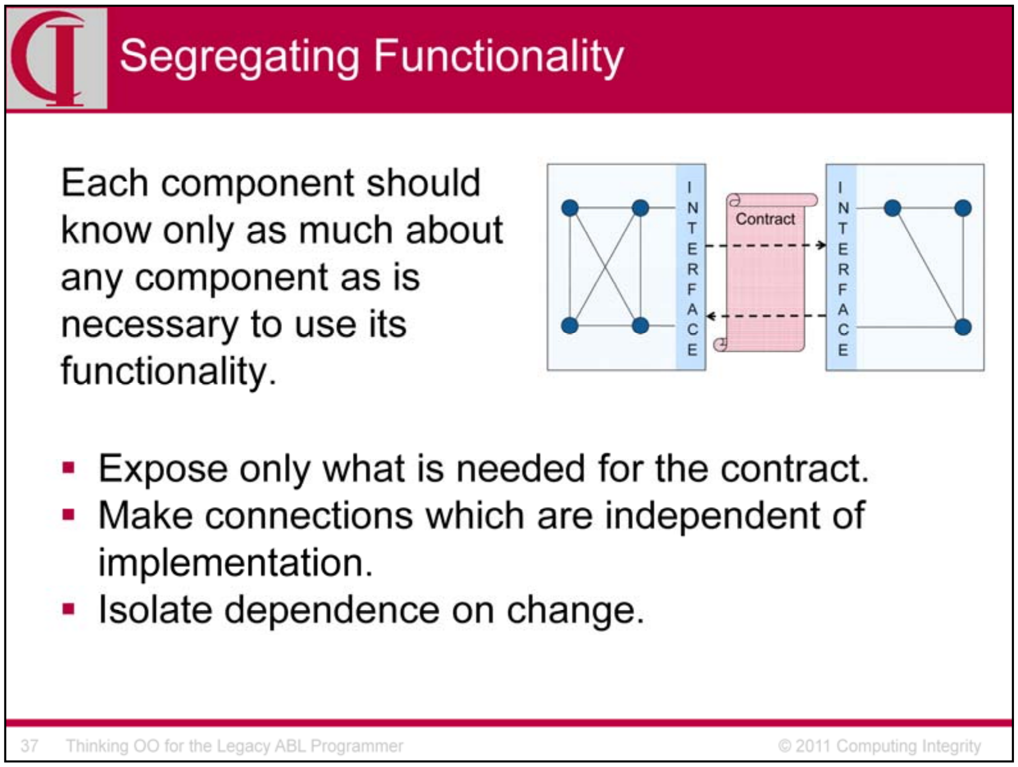


In OO programming, it is common to define what are called interfaces. An interface is an abstract definition of a particular contract which is then implemented by multiple classes. For example, one might decide that there were certain behaviors common to all wheeled vehicles and define an interface with methods corresponding to those behaviors. This interface could then be implemented by a number of different wheeled vehicle classes. In programming, whenever one needs only methods in the shared contract, one can reference any of the classes which implement this interface by referring to the interface instead of the specific class. By programming to the interface, one programs to the contract, not the implementation. Multiple implementations can all respond to the same contract. Any change or addition in the implementation requires no change in the consumer.

Those of you familiar with inheritance will recognize a similar purpose. Inheritance is for when there is actual shared behavior and knowledge while interfaces are for where there is just shared contract, but the implementations are different. Both techniques have their place – Inheritance when there is a specific super and sub set relationship and interfaces when there in only a shared contract, which might apply across very different classes.

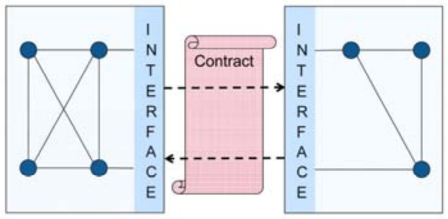
For designing, this is equivalent to the notion of Design by Contract, i.e., designing according to the contracts which classes present without regard to their implementation.

Traditional ABL does not have interfaces per se, but the idea of designing and programming to a defined contract is still applicable. Any time one programs with active knowledge and use of the implementation of another component, one is violating this principle of programming to the contract.



Segregating Functionality

Each component should know only as much about any component as is necessary to use its functionality.



- Expose only what is needed for the contract.
- Make connections which are independent of implementation.
- Isolate dependence on change.

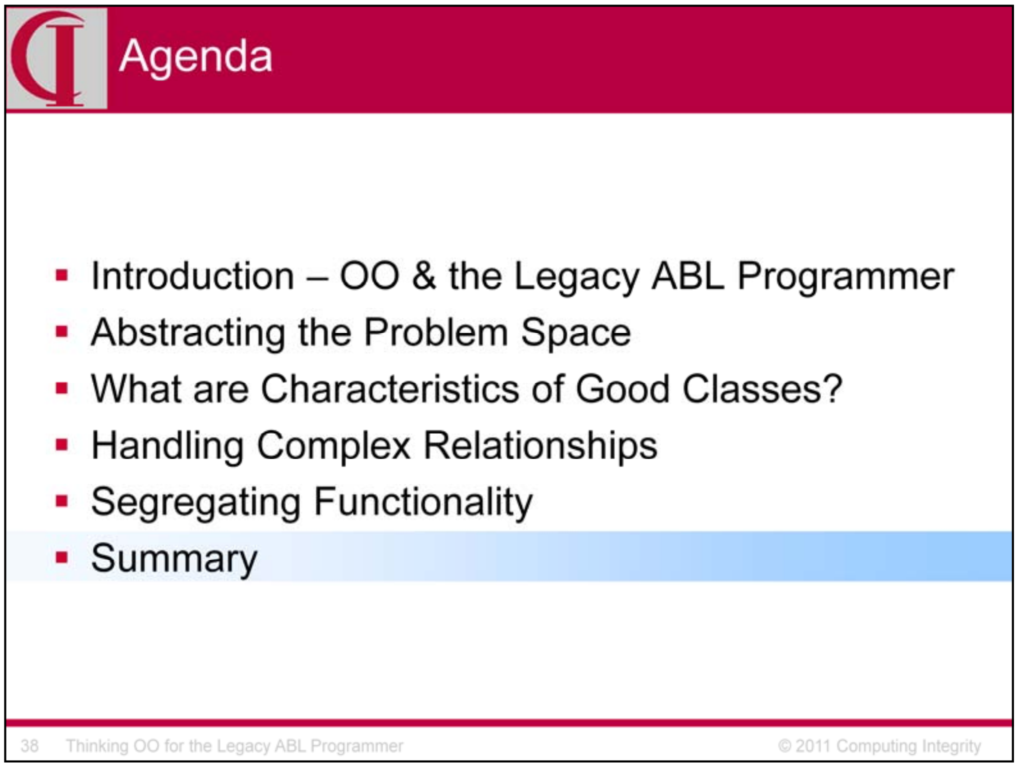
37 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

The overriding principle here is that each class or component or subsystem should know only as much about any other class, component, or subsystem as is necessary to use its functionality.

Expose only what is needed for the contract.

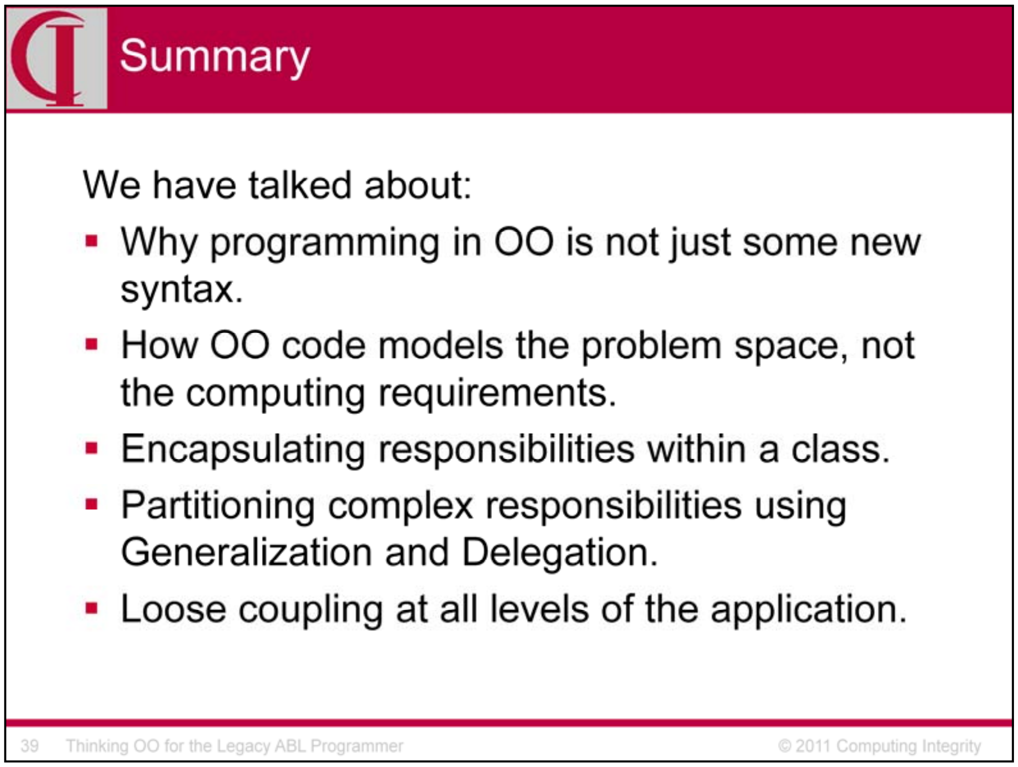
Make connections which are independent of implementation

Isolate dependence on change.



Agenda

- Introduction – OO & the Legacy ABL Programmer
- Abstracting the Problem Space
- What are Characteristics of Good Classes?
- Handling Complex Relationships
- Segregating Functionality
- Summary



Summary

We have talked about:

- Why programming in OO is not just some new syntax.
- How OO code models the problem space, not the computing requirements.
- Encapsulating responsibilities within a class.
- Partitioning complex responsibilities using Generalization and Delegation.
- Loose coupling at all levels of the application.

39 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

We have talked about:

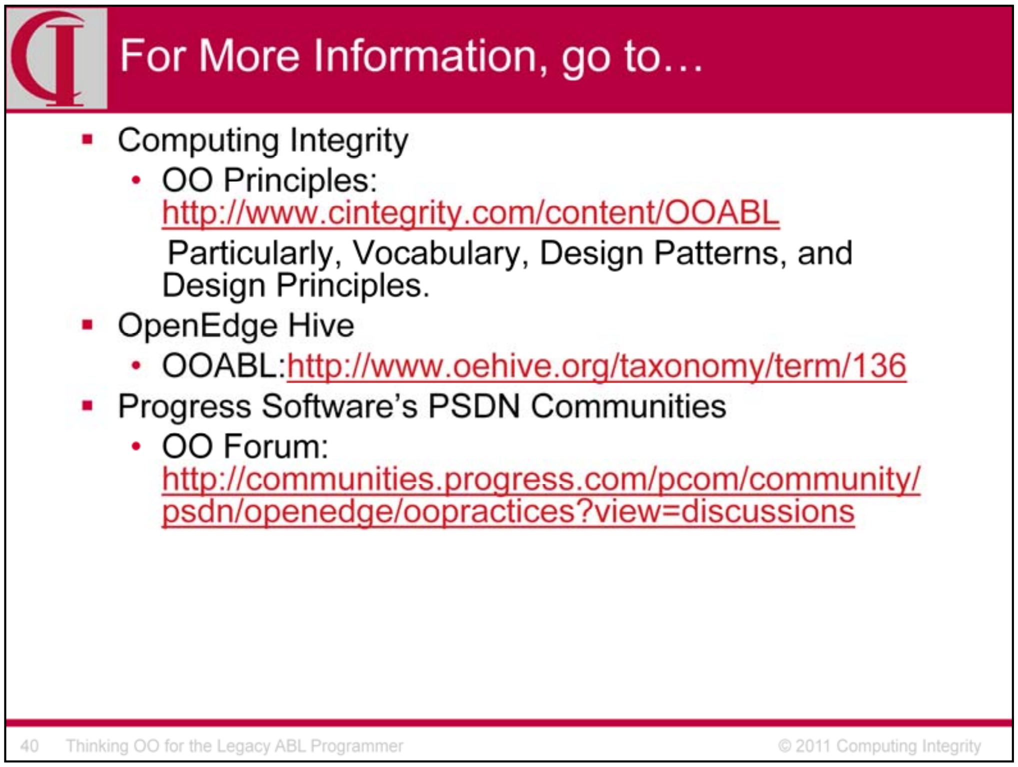
Why programming in OO is not just some new syntax.

How OO code models the problem space, not the computing requirements.

Encapsulating responsibilities within a class.

Partitioning complex responsibilities using Generalization and Delegation.

Loose coupling at all levels of the application.

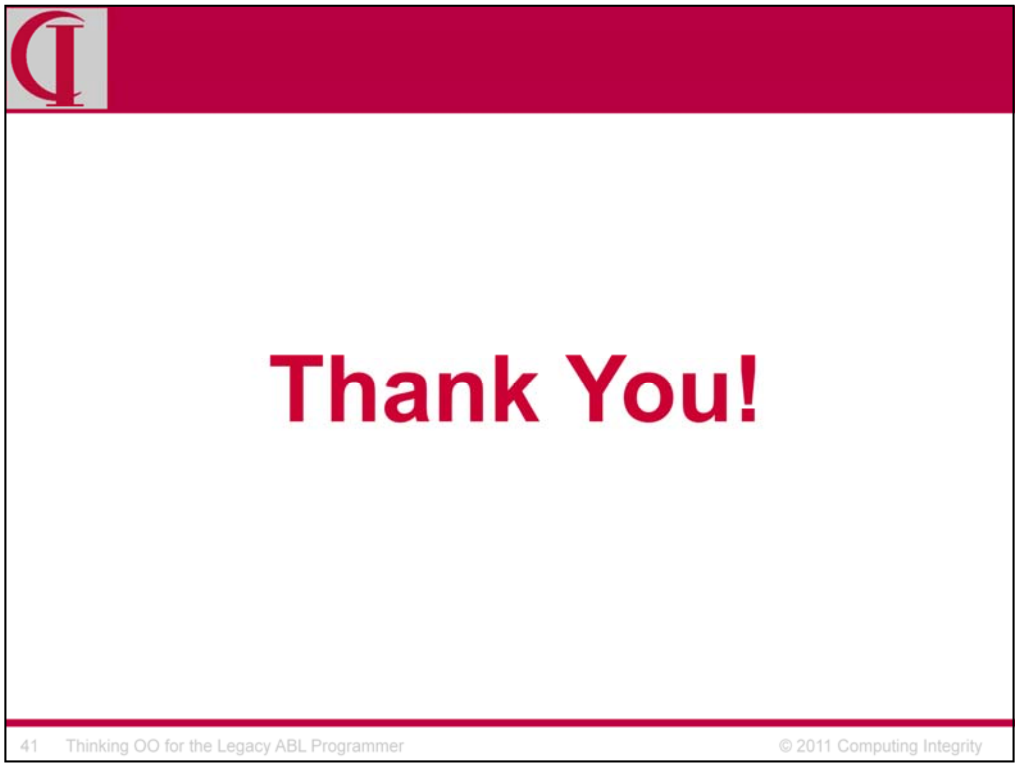


For More Information, go to…

- Computing Integrity
  - OO Principles: http://www.cintegrity.com/content/OOABL
    Particularly, Vocabulary, Design Patterns, and Design Principles.
- OpenEdge Hive
  - OOABL:http://www.oehive.org/taxonomy/term/136
- Progress Software's PSDN Communities
  - OO Forum: http://communities.progress.com/pcom/community/psdn/openedge/oopractices?view=discussions

40 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

Here are some links for more information. Generally, look on OE Hive under OOABL and look at the articles section of our website.



Thank You!

Thank you.

Questions?

For more information:

http://www.cintegrity.com

thomas@cintegrity.com

510-233-5400

42 Thinking OO for the Legacy ABL Programmer © 2011 Computing Integrity

And now for questions.