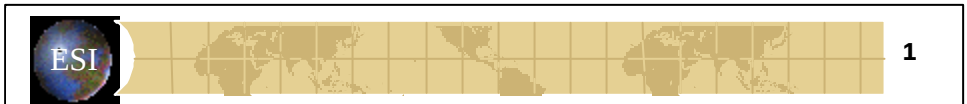
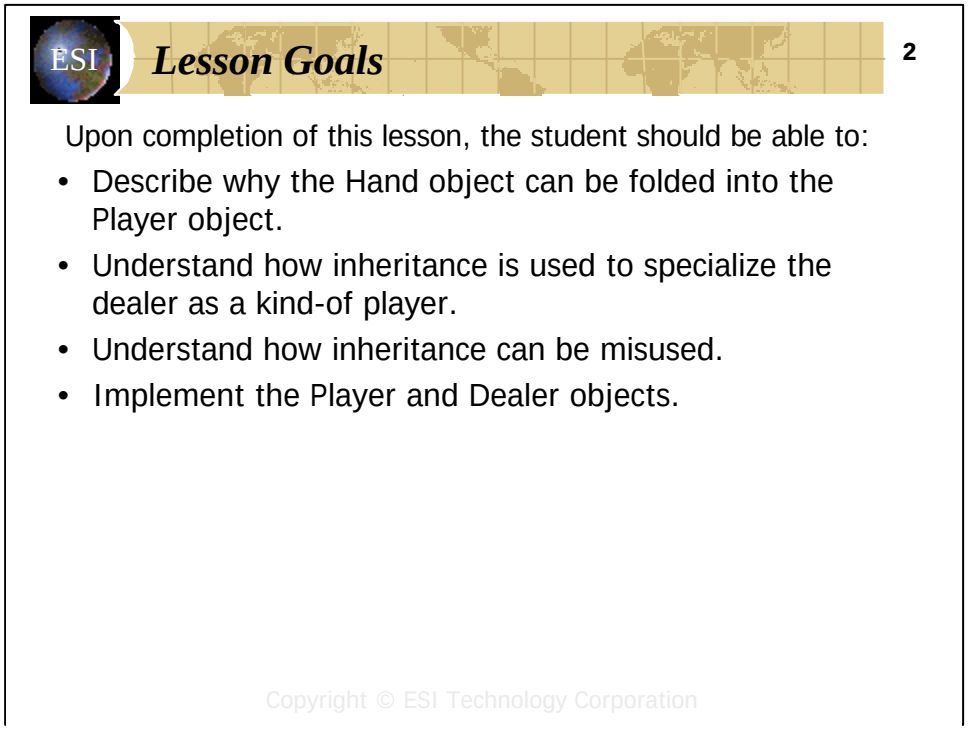


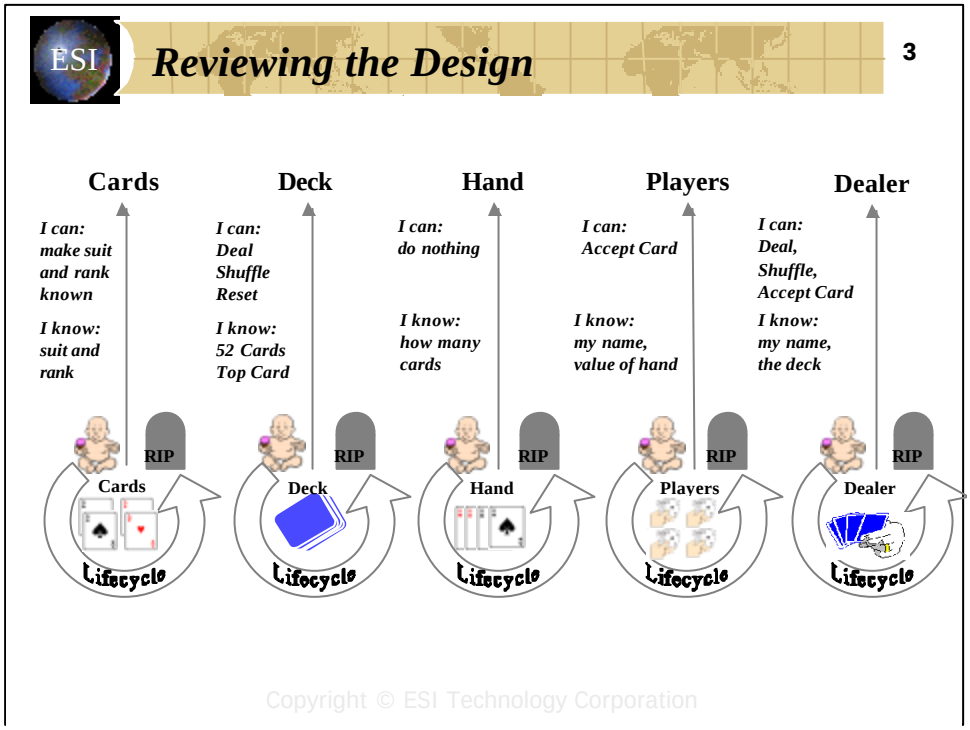
Object Oriented Concepts:

Using Inheritance and Specialization

This lesson will build upon the concepts we covered in previous lessons. In this lesson we implement the Player and Dealer classes. Implementing these classes forces us to look at specialization and how to use (and possibly misuse) inheritance. We will also explore the extensibility of our design and show how it will get us into trouble if we were to evolve it into a more flexible, extensible design.



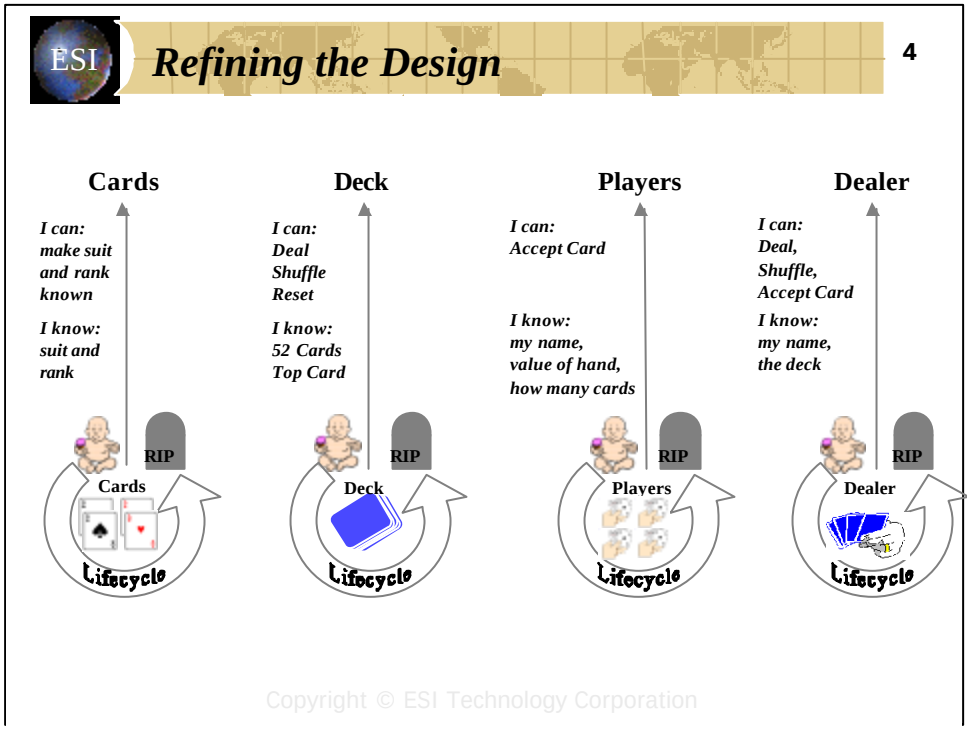
Read and understand the objectives of this lesson.



You made plans to meet her again. You engage in some small talk while you pull out your notes from the last lesson. “Did you get the Deck class coded? Did you have any problems?” you ask. “Yes I got it coded and tested. It was rather simple to do!” she said.

“Great,” you say, “Lets move on to the next topic, that is, refining our design by taking a look at the Hand and Players classes.” Your student studies the ‘can do’ and ‘knows’ diagram carefully. “What do you see and how can we refine the design?” you ask.

“Hmm, the first thing I notice is that the Hand class does not have any behavior defined on it. That seems to say something.” “And what else?”, you ask. “It only knows ‘how many cards’ which seems to overlap with what the Player’s class know. If a player knows the ‘value of the hand’, why shouldn’t the player also know ‘how many cards’ are in the hand? It would appear that we could move the knowledge of ‘how many cards’ into the Player class and delete the Hand class.”



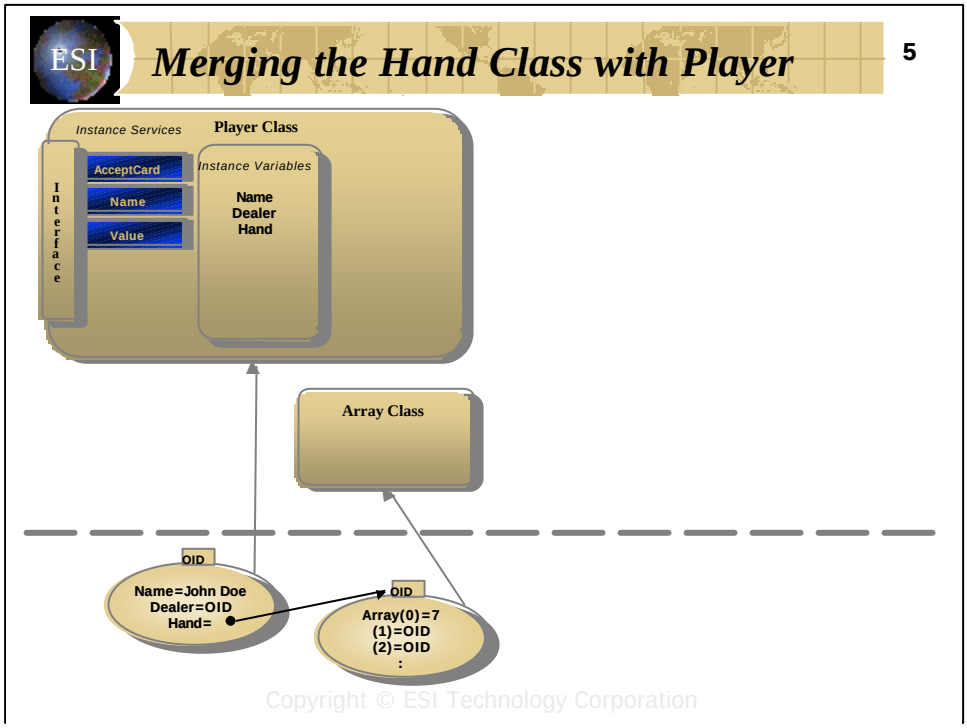
"Yes, this is possible. What are the consequences?" you ask as you move the 'knows how many cards' to the Player class and erase the Hand class from the design document.

"Well, clearly it simplifies the design. On the other hand, it introduces some inflexibility by combining two abstractions of the game," she replied.

"Exactly!" you reply, "However, in the context of this system, combining abstractions is not a bad thing since the system has a very limited purpose. Under most circumstances where extensibility is a basic requirement of a system, it is always important not to paint yourself into the proverbial corner. When building a system, it is always important to make sure that in future iterations a design can be extended without a total rewrite. It's important to put lots of time into analysis and design, making sure that you have generalized the design. In that way, you can easily extend the system. This clearly lets you perform shorter implementation iterations, getting new functionality out to your customers more often. Object Orientation gives you the edge you need to accomplish extensibility goals."

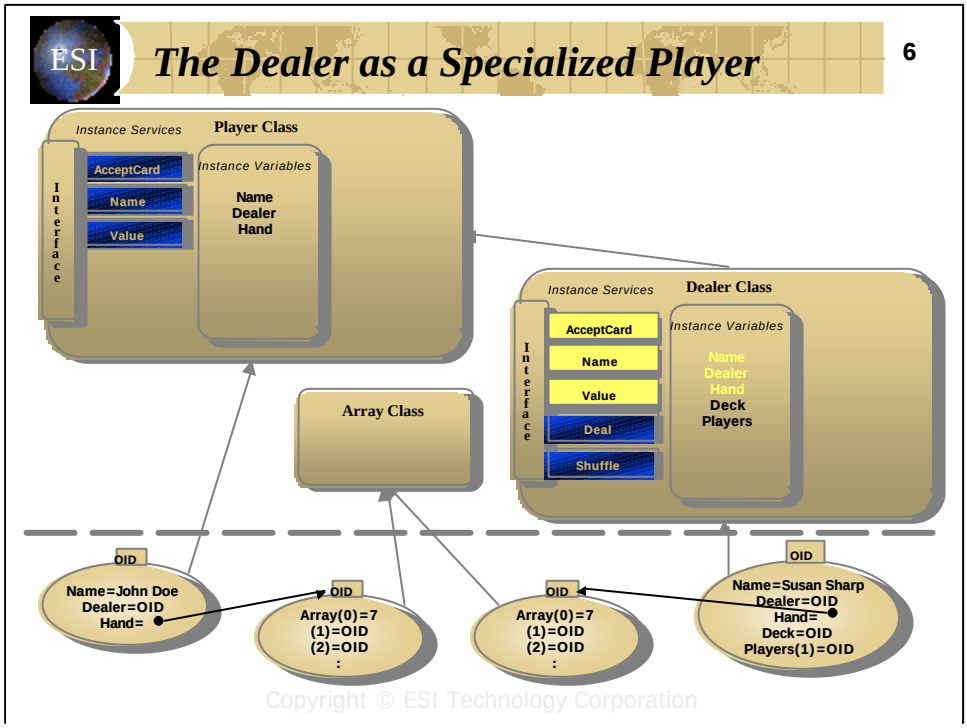
"In other words, if I were building an information system for the casino, I would want to do a thorough analysis of our existing operations and at the same time, understand where management expects to take the business. I would then design a system to accommodate our operations now and in the future," she said.

"Funny you should say that because, next, we are going to look at the concept of **specialization** and **inheritance**. We will show how powerful the concept is when used correctly, but how it can result in an inflexible system if used incorrectly. So, lets move on!".



"Let's review the Player class," you say as you pull out the design document. "Up until now the Player class contained the instance variables Name and Dealer. Now that we've merged the class Hand with Player, we will have to add an instance variable to hold a pointer to an Array object. This is our first attempt at reuse. Remember that most OO development environments contain reusable classes. Most have a set of classes called Collections. An Array is a Collection class. We don't have time to talk about Collections in detail, however, if most development environments provide these objects, they will certainly have documentation that tells you how to use them. When you model the Player class, refer to the documentation provided by your tool's vendor."

"So, why did you chose an Array object to hold a hand?" she asked. "Because, it's ordered. That is, it gives you control over the ordering of its entries and later on we will have a need for that capability," you reply. "Ok…," she said indifferently.

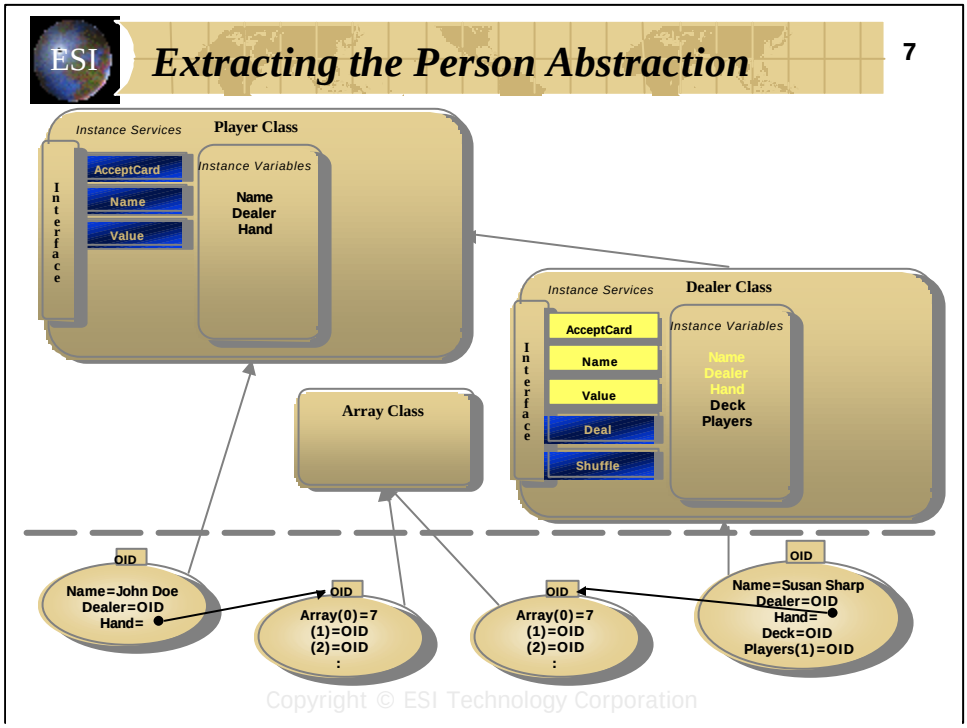


"Back in Chapter 3 we discussed the concepts of abstraction and how it gave you a tool to separate meta information from real world scenarios. Also, how we used the classification system to model these abstract levels and how we could link them together in 'kind-of' relationships. We discovered how these linkages are used by the compiler and the runtime system to implement the concept of inheritance as well as promotion and overriding capabilities." you state. "Yes, I remember," she said.

"Take a look at this drawing. Notice that the Dealer class is linked to the Player class in a 'kind-of' relationship. It becomes a subclass to Player which is its superclass. Dealer inherits certain variables and services from the Player class. What are they?" you ask. "The AcceptCard method and the Name and Value properties as well as the Name, Dealer and Hand instance variables." she replied. You take out your yellow marker and mark them as inherited.

"Notice how, through the concept of inheritance, the Dealer object becomes more specialized than the Player. It displays more behavior than the Player," you remark. "Yes, inheritance is cool! I can see where it is the secret to reusability as well as to extensibility. If we properly separate abstractions from our real world objects, we can use the features of OO to build a system that can be extended without being rewritten. It should be highly reusable as well," she said.

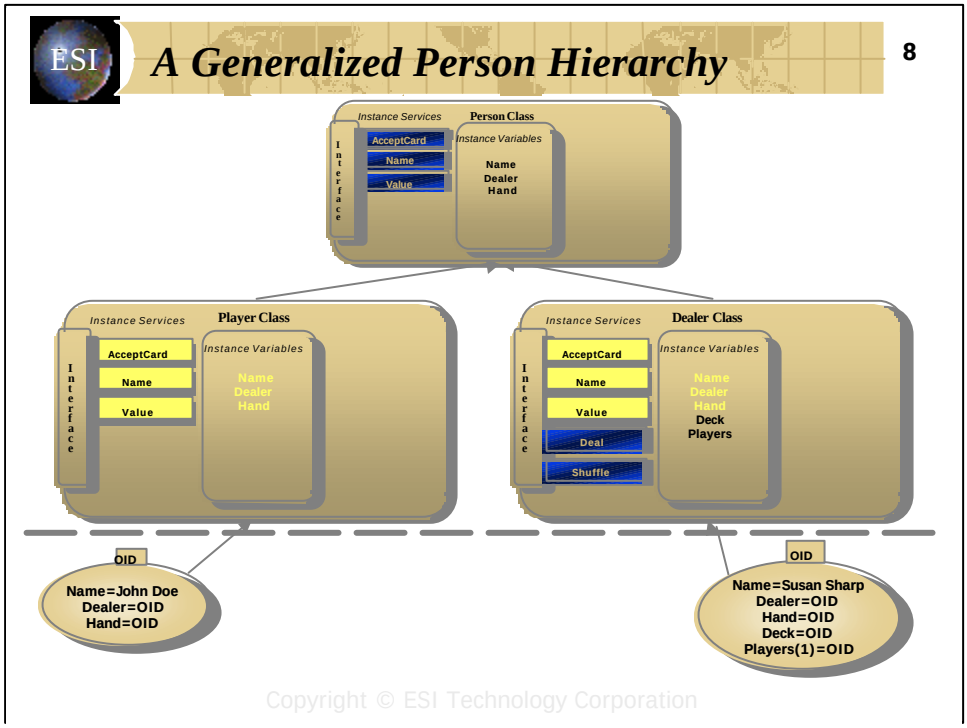
"You're correct! Extracting different abstractions is the secret to building an extensible system that results in reusability," you reply. "Now, let's see if we can extend our design."



“Take a look at the design so far, in particular the Player and Dealer classes. What do they have in common?” you ask.

“Well, these classes model human beings or persons. The Players are persons and the Dealer is a person,” she said. “Also, not to digress, but one thing has always bothered me about the Player and Dealer class hierarchy. Since Dealer inherits from Player, the Dealer class is the most specialized and therefore concrete. Right? If that is true, then the Player is more general and shouldn’t it be an abstract class? If so, didn’t you say that abstract classes could not have instances?”

“Great observation!,” you reply, “They are persons. To answer your concern about abstract and concrete classes, you are correct. Generally a class that is abstract cannot have instances. However, in our design we have declared both of them concrete, therefore capable of bearing instances. The Dealer is just a little more specialized than the Player. Just because one class is more general than another that inherits from it, this does not mean it cannot bear instances. So, how would you evolve this design to ‘fix’ what you see as two problems?”

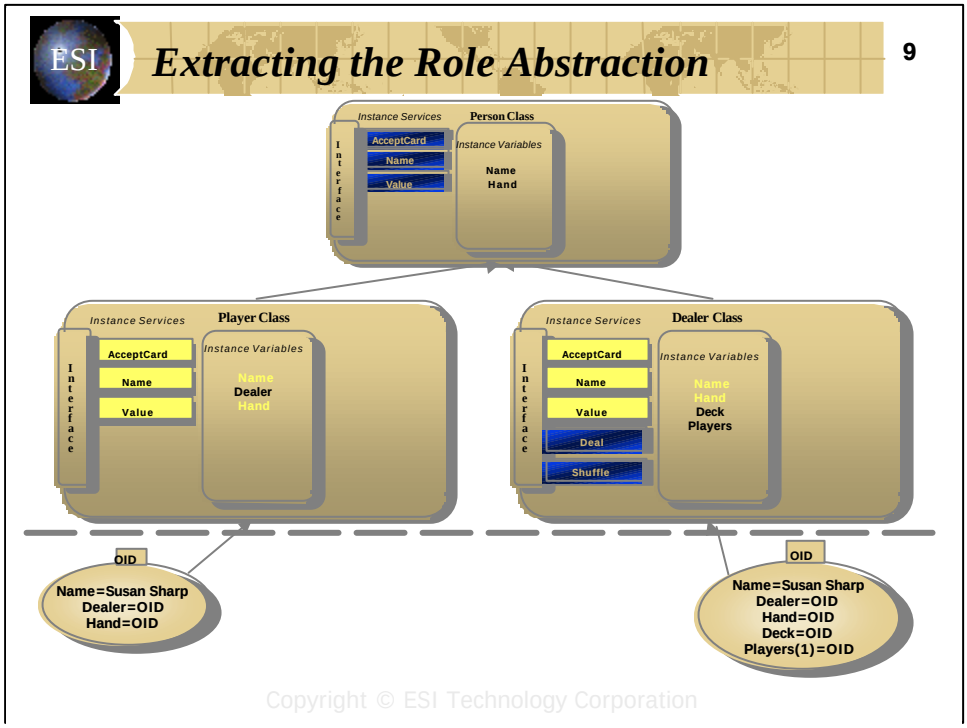


“I would extract the person abstraction and create a Person superclass to the Player and Dealer classes,” she explains as she draws a new class diagram. “Then, I would promote all the Player’s instance variable declarations, method and properties to the Person class since the Dealer must continue to inherit these components. I would then delete them from the Player class so that they would be inherited from the Person class as well.”

“Wow,” you say, “You’re catching on fast. But notice how you were able to generalize the design on the fly without disturbing the instances. They are still instances of the classes they were instantiated from and do not know that the abstraction changed above them. No database conversion here! How do you feel about this design?”

“I really think this is generalized, I like it better than the original design.Inheritance really is cool. It gives you a tool to model abstractions with,” she said.

“You’re right again, inheritance is a cool tool,” you explain, “However, I don’t want to burst your bubble but there is one little problem left to solve. Unfortunately this problem has some side effects that are not pleasant. Let’s take the design one step further.”

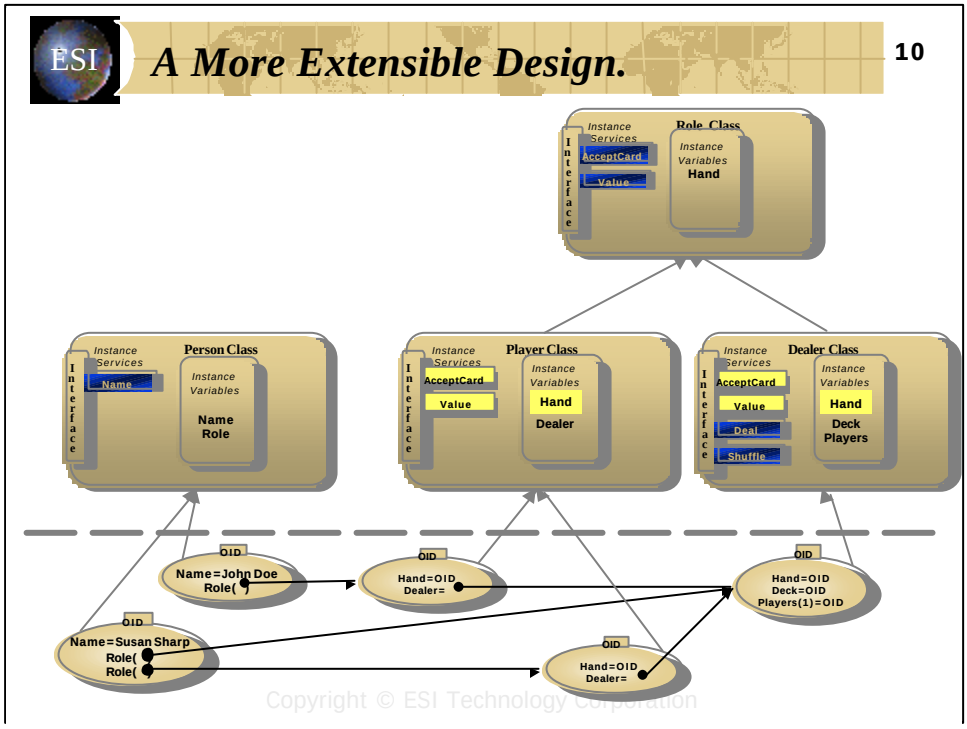


To illustrate the problem, you ask her, “In your off hours, do you sometimes relax by returning to the casino to participate as a player?”.

“Yes, of course,” she replies.

“Ok, lets make an assumption here. Assume that all Player and Dealer objects are stored in the computer persistently. That is, they exist on the computer until someone explicitly deletes them. This is typical in a database system. So, as a Dealer, you would exist in the computer as Susan Sharp the Dealer object and, because you also play the game in your spare time, you would exist in the computer as Susan Sharp the Player object. Here is the problem. This is what we call the duplicate record problem. Without getting into the details, let me tell you that it can cause problems. However, the real question is, how do you fix it?” you ask.

“Hmm! Just by looking at the problem, it appears that we did not finish the job of extracting abstractions. Yes, Player and Dealer objects are Person types, however, Player and Dealer are also roles that people play. Give me a couple of minutes, I think I can redesign this!”, she said confidently.



She takes out a blank piece of paper an starts to scribble furiously. When she finishes, she explains the design. “The simplest way to do this is to first change the class named Person to Role. Person is an abstract class and, in this case, should not have any instances and it is already linked to the Player and Dealer subclasses. Now delete the Name instance variable definition and Name property from the Role class. Next create a new Person class and declare a variable Name and its property Name. Additionally declare a new instance variable called Role. This instance can be an M array or you can reuse a Set collection. In any event, the collection will hold the Role objects the person could play.”

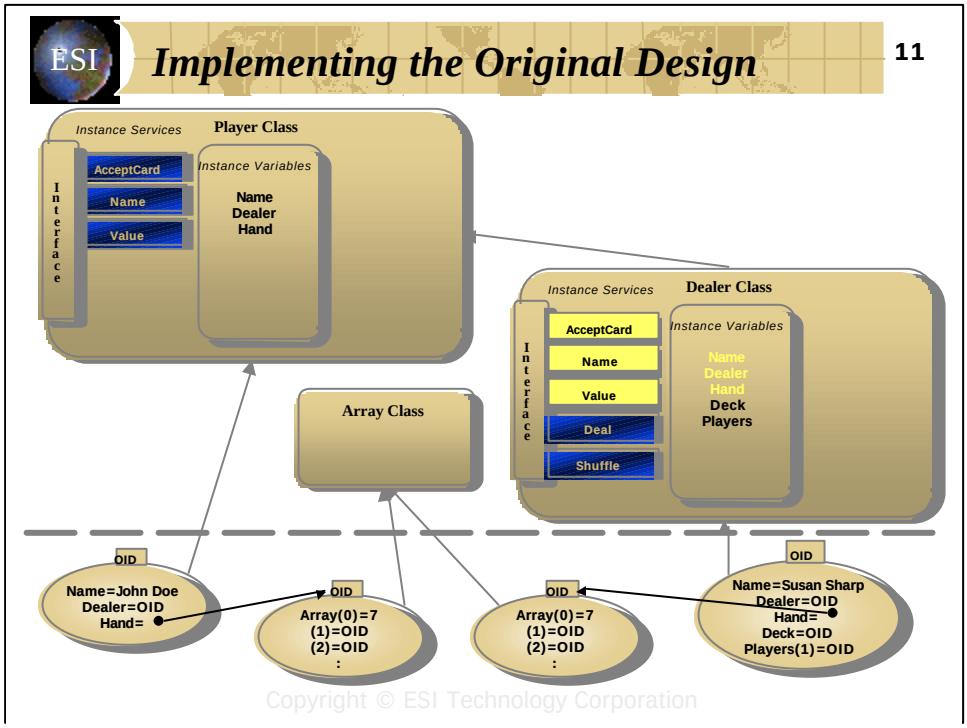
“That’s great,” you say, “Notice how much more flexible the new design is over the original design? It is now much more extensible than the original design. Tell me, what would happen if you were to implement and use the first design and then be required to switch to this design?”.

“You would have to run a database conversion I guess?” she replied.

“Exactly,” you say, “This is why it is important to separate abstractions. This leads to flexibility and is the cornerstone to extensibility and reusability.”

She looks a little puzzled like something is bothering her. You ask, what’s the problem? She says, “With the flexibility, the design got more complex. Won’t it be a little harder to support and won’t is be slower to execute?”.

“Ah, this is generally the trade off. But the answer is simple. Is it better to paint yourself into a corner or add a little complexity and gain reusability and extensibility?” you ask.



"What are we going to do?", she asked. "Well," you reply, "Since I know for a fact that my company will never extend this design, we are going to continue with the original design. It is adequate for the requirements of this system, that is a simple Poker game. Besides, I only have another day here!"

"So what do I do next?" she asked. You then hand her a list of instructions that will guide her through the programming task of implementing the original design.

1) First design and code the Player class. Create state variables: Hand (an array for the cards), Name (player's name). Now add the operations: AcceptCard (method), Reset (method), Value (property) which represents the value of the player's Hand and Name (property).

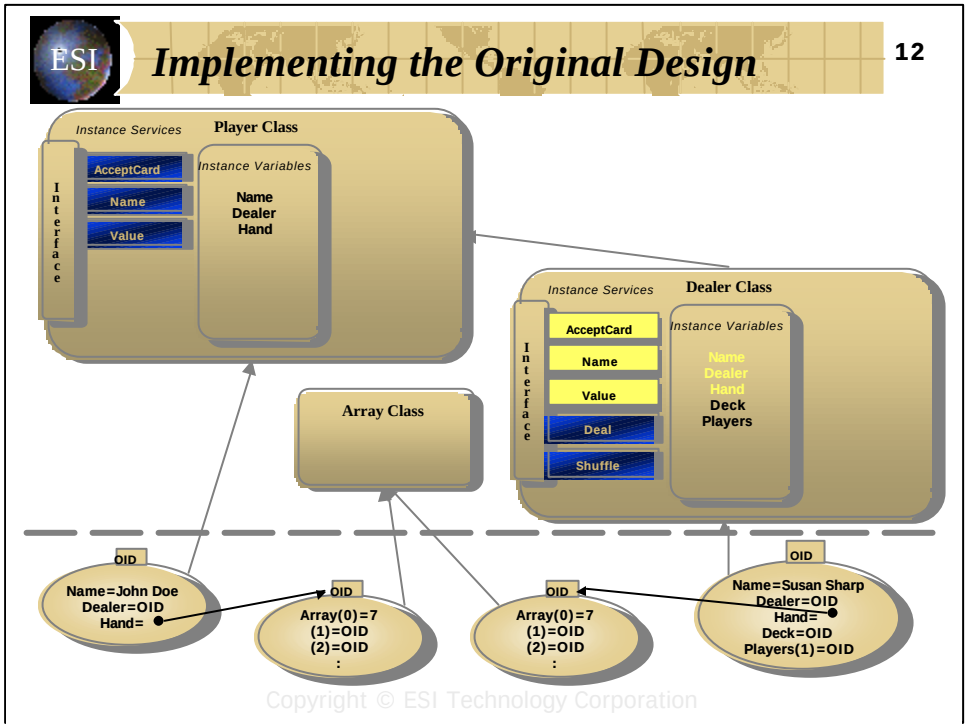
2) If you use an Array collection for the Hand and if your development environment supports automatic instantiation and binding to instance variables, then use it. In EsiObjects™ the Hand variable would be declared as Initialized and bound to a Base$Array object.

3) Now code the AcceptCard. It should simply take in a Card object and add it to its Hand array.

4) Code the Name as a get/set property.

5) The Value property should be left for now since this is the complex portion of the game – determining value. So leave it for now.

6) Please note: When the Player is destroyed the Hand array is cleaned up (since it's a Child object) but any Cards in the Array are NOT destroyed. Collections do not destroy their contents and since the Cards are outside the scope of the Player, they are not affected.



7) Reset should simply clear out the Player's Hand array.

8) Test the Player object and AcceptCard method etc.

9) Now design and code the Dealer class. Since the Dealer is a specialized Player, the Dealer should be subclassed to Player. Note the reuse – a Dealer is automatically a Player.

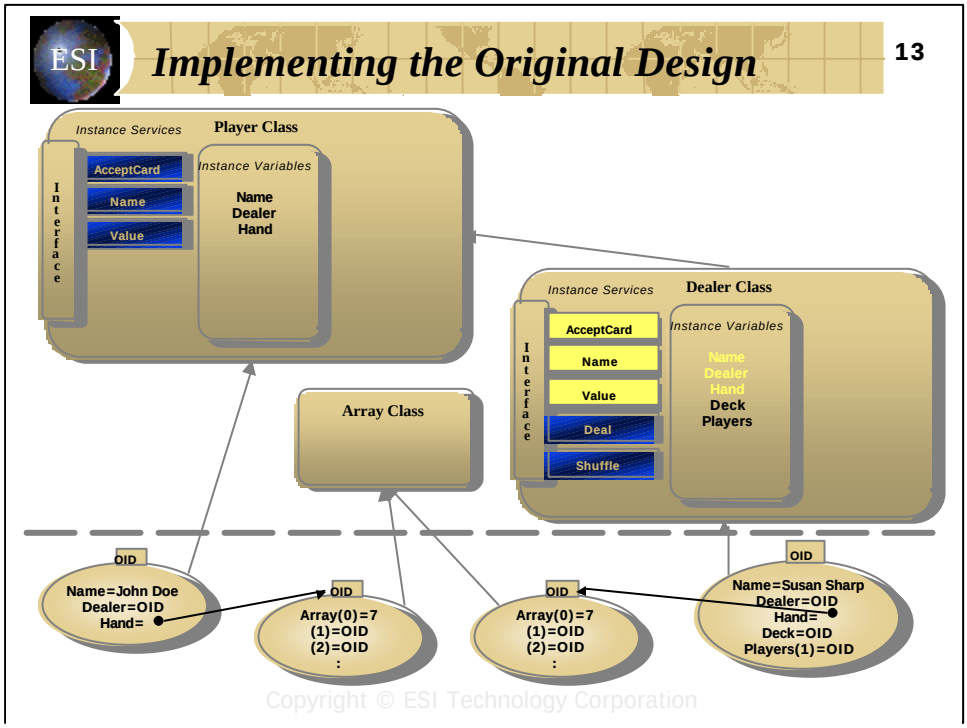
10) Here is the state of a Dealer: Deck (a Deck object), Players (an array of Players.) Here are the operations: Setup (intialize game), AddPlayer, Shuffle, Deal, Reset.

11) Variable Deck is initialized to a Deck object.

12) Players can be a static variable (it will be a simple internal M array.)

13) The main complexity here is to figure out how best to populate the Player array. AddPlayer should accept a Player object and add it to the Player array. The Dealer should also add itself to the Player array. This could be done with a constructor method (which would need to be overridden.) The Dealer should be placed at the end of the Player array since the Player array will be used for dealing – thus the order of the deal should have the dealer as the last one to get a card. Any number of possible solutions can be used.

14) Since the AddPlayer method will accept a Player object you should validate the input to make sure the input is a valid Player object. Your development system should have a function to validate the input parameter as an object of Player type.



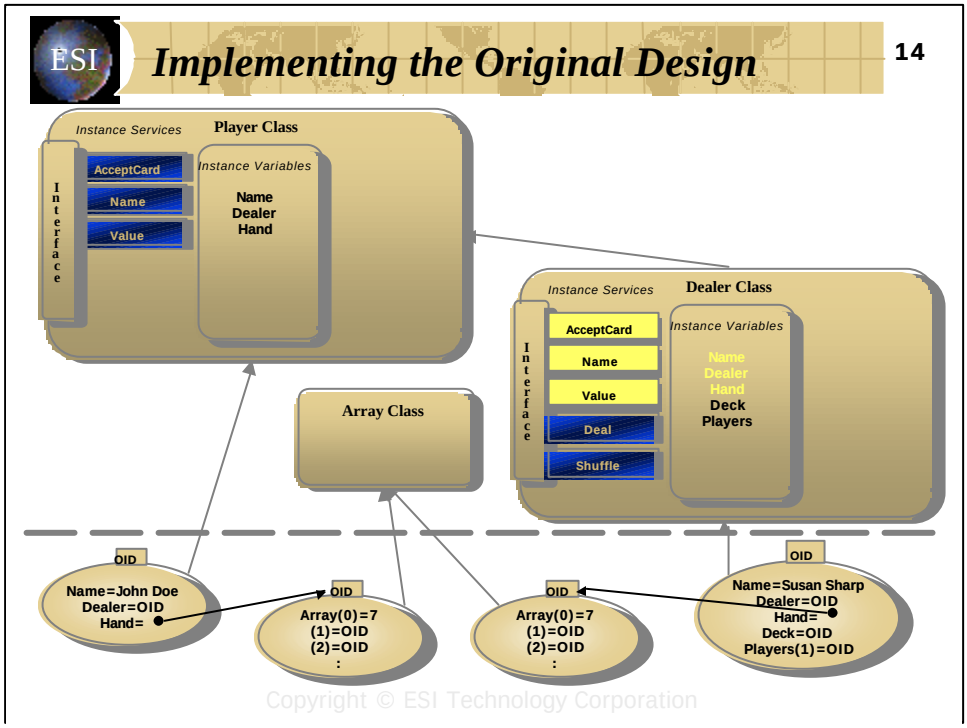
15) Now code Shuffle. It simply calls Shuffle on the Deck object.

16) Deal accepts a number on input which is the number of Cards to deal (default to 5). If the number of Cards to deal exceeds the number of cards available to deal to all players, the Deal method could return –1 or something. The dealer should pass a card from the Deck to each player in the same way a real dealer deals cards. The first card to the first player, the second card to the second player, etc. The Dealer is last in the deal order. Repeat for the number of cards specified to deal.

17) Override the Reset method in Player. If your implementation has DO $SUPER capabilities, use it to do the work of Reset in the Player class. In addition, the Reset in Dealer should call Reset in the Deck. (Shuffle could also be called, unless you wish to keep Reset and Shuffle as two distinct operations)

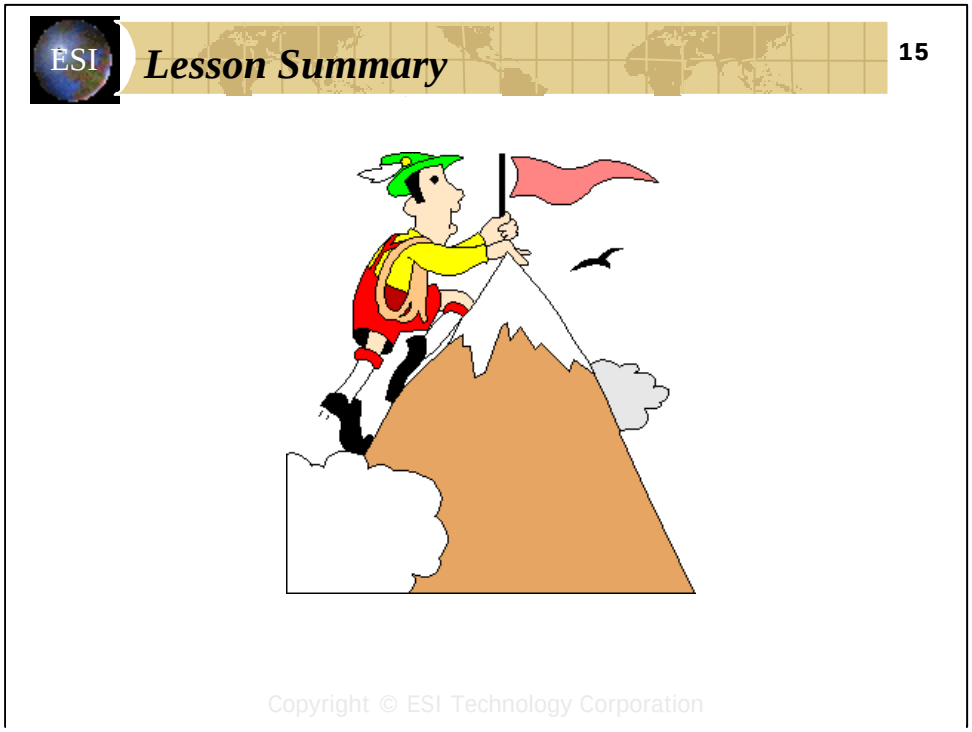
18) Using your implementation's means of creating objects interactively (the Xecute Shell in EsiObjects), create Dealer and Player objects. Add Player objects to the Dealer. If you have an Object Browser, use it to inspect the internals of the the Dealer and Player objects. Does the Player array in Dealer look complete and correct?

19) Try the Shuffle and Deal methods. Again, if you have an Object Browser, use it to check Player's hands etc. Try Reset too. Make sure that shuffling happens between each iteration of a "game", whether it's done in Reset or as a separate method call to Shuffle directly.



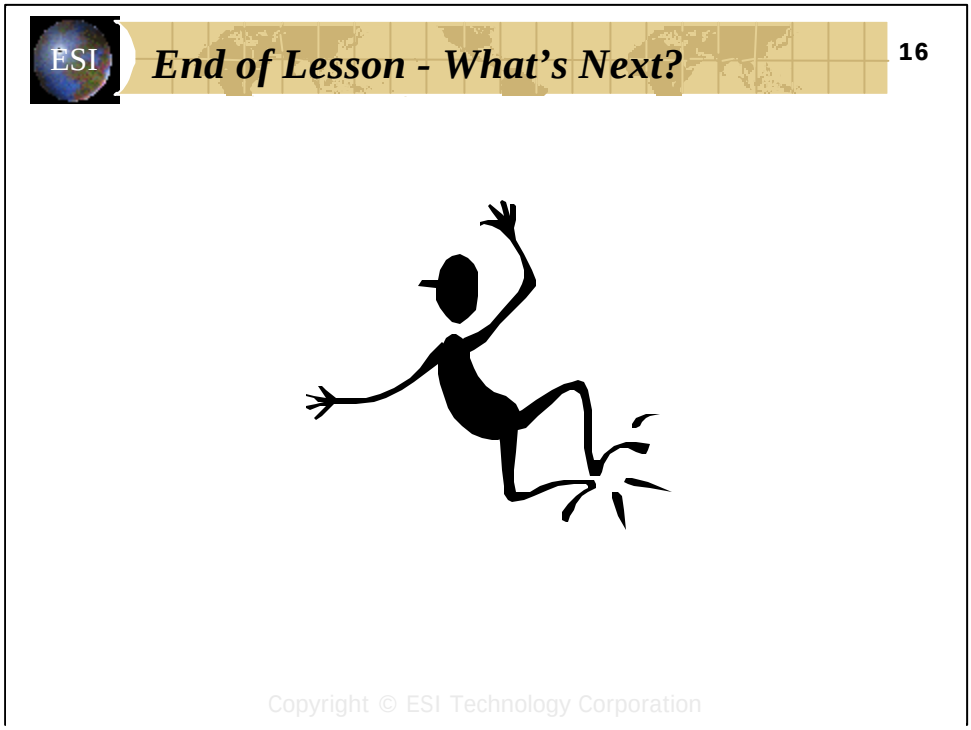
At this point you should have a basic shell of a game except for the critical part – determining a winner. We will leave this for the next lesson when we will introduce the concept of a Hand Analyzer. It will introduce you to the concept of **event processing** which is extremely complimentary to object orientation.

We have not provided for the destruction of the Dealer or Player objects. The Deck and Card objects and other array objects are all cleaned up when the Dealer or Player are destroyed. Usually the application running the game will be responsible for creating and destroying the Dealer and Player objects. No DESTROY methods are needed.



This lesson tried to show how a class can be collapsed into another class when it is a logical extension but does not have any individual behavior. We also covered how subclassing and inheritance can be used to great advantage, but also showed how it can create problems if the design is incorrectly applied. We then covered the instructions for adding the Player and Dealer classes and their services.

Except for some embellishment, the model side of the game is complete. Feel free to expand upon the game.



The next lesson (Lesson 6) will continue with the next iteration of the Poker Game. This lesson will explore adding a HandAnalyzer object to the system. This is an object that knows how to analyze a poker hand, forming the needed functionality to determine a winner. The new concept covered in this lesson will be **Events** and how to use them to broadcast information such as a winning hand.