# Why OOM is Better than Traditional M

*by Erik Zoltán*

## Why You May Not Believe This Article

It is often difficult to explain to people why Object-Oriented M (OOM) is "better" than traditional M programming: it seems as though the moment you start presenting examples, the most knowledgeable people start to raise objections to them. This is because there is a temptation to present *rather simple examples*. But that is exactly the wrong approach, as the following idealized graph makes clear.

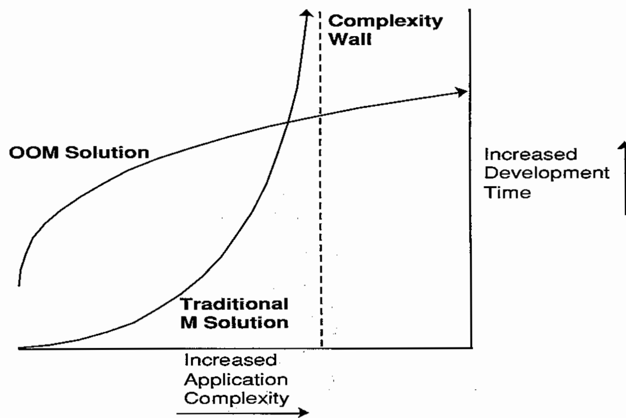


Fig. 1. Comparison of OOM and Traditional M Programming

This graph does not represent the results of a particular study or project, but it does illustrate well-documented relationships; similar graphs can be found in the OO literature. If the claims made in the OO literature are true, then they have the following implications:

- simpler examples may actually take longer to develop using OOM than with traditional M programming.
- the benefits of OOM should be most clearly seen with more complex applications that evolve over time.
- traditional programming techniques will encounter a "complexity wall," making it harder and harder to add new features.
- OOM designs, by contrast, will eventually approach linear development time, meaning that new features will not be much harder to add.

This article will argue that OOM possesses advantages over traditional M programming for larger-scale development projects. It will do this by contrasting a traditional, "procedural" M application that has encountered the complexity wall, with an OOM solution that avoids the wall. This OOM solution was designed with EsiObjects 2.0, an object-oriented M development environment by ESI Technology Corporation, that now conforms to the forthcoming ANSI M OO binding. A card game application is used because it is simple enough to explain in a short article, relatively non-technical, yet offers adequate scope to illustrate the complexity wall. Much more complex M applications, though they might provide better illustrations, would take longer to adequately explain.

## Poker Game: Initial Design

### *Traditional Solution*

Imagine an application that plays 5-card draw poker. How might one code such an application in M? There is an array of available cards (the deck) and one array for each hand; optionally, discards might be stored in another array. The computer always deals and acts as one of the players. It needs to know how to determine the winner and how to make decisions regarding its own discards. Betting also needs to be supported.

Determining the winner is an important problem. The procedural solution uses an extrinsic function to assign a numeric "weight" to a hand. This function looks for each kind of hand in a predefined order: straights and

```
WEIGHT(HAND) ; Return Hand's Weight
 N CARD,SUIT,X,CL,SL,FLUSH,STR,HIGH,MATCH
 Q:$D(HAND)<10 ""
 F X=1:1:5 S CARD=HAND(X) D
 . S SUIT=$E(CARD,$L(CARD))
 . S $E(CARD,$L(CARD))=""
 . S CARD=$S(CARD="J":11,CARD="Q":12,
   CARD="K":13,CARD="A":14,1:CARD)-1
 . S CL(CARD)=$G(CL(CARD))+1
 . S SL(SUIT)=$G(SL(SUIT))+1
 S FLUSH=$O(SL($O(SL(""))))=""
 S CARD=$O(CL("")),HIGH=$O(CL(""),-1)
 F X=1:1:4 S STR=$D(CL(CARD+X)) Q:'STR
 I STR,FLUSH Q 700+HIGH
 Q:FLUSH 400+HIGH
 Q:STR 300+HIGH
 S CARD="" F X=1:1:5 S MATCH(X)=0
 F  S CARD=$O(CL(CARD)) Q:CARD=""  S X=CL(
   CARD),MATCH(X)=MATCH(X)+1
 S (X,HIGH)=0,CARD=""
 F  S CARD=$O(CL(CARD)) Q:CARD=""  D
 . S:CL(CARD)>X X=CL(CARD),HIGH=CARD
 Q:MATCH(4) 600+HIGH
 I MATCH(3),MATCH(2) Q 500+HIGH
 Q:MATCH(3) 200+HIGH
 Q:MATCH(2)=2 100+HIGH
 Q:MATCH(2) 50+HIGH
 Q HIGH
```

Fig. 2. $$WEIGHT Function

flushes, 4-of-a-kind, full house, 3-of-a-kind, two pairs, one pair or finally, nothing at all. The player whose hand has the highest weight is the winner and collects the pot for that hand. For example, a hand in which there is a full house receives a weight between 501 and 513, depending on the value of the tripled cards. But a straight flush receives a value between 705 and 713, so it clearly beats a full house. This function, invoked as $$WEIGHT(.HAND), is illustrated in Fig. 2.

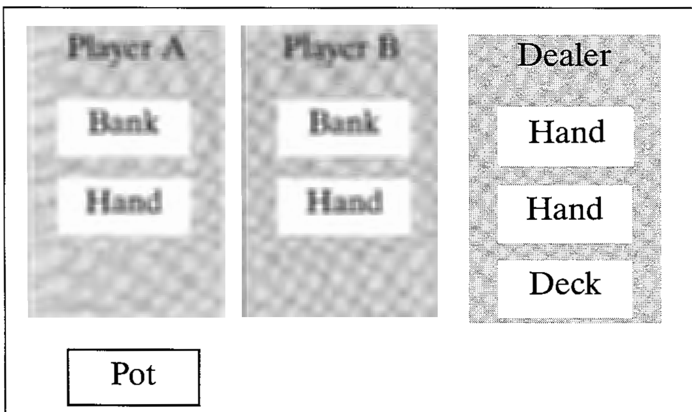


Fig. 3 Components of CardGame Object

## OOM Solution

The initial OOM solution requires significantly more work. Classes are created to match the elements of a card game. There is a CardGame object containing a Pot (the money that has been bet), a collection of Player objects, and one Dealer.

Because of a principle called *encapsulation*, objects cannot directly modify the internal states of other objects. Instead, they communicate by sending *messages* back and forth. Thus, Players request additional cards from the Dealer that obtains them from its Deck. The Deck is a discrete object encapsulating (containing) its cards, just as the Dealer encapsulates the Deck. It would be improper for the Dealer to directly modify the Hand of each Player, since Hand is an internal component of the Player object. The Dealer must send a message requesting direct access to a Player's Hand; the Player returns a reference to the Hand object. Similarly, the Dealer cannot directly modify the Hand: it sends the Hand a message telling it which card(s) to add. The Hand then implements this request, or it could refuse to do so—it might reject the attempt to add a sixth card, for example.

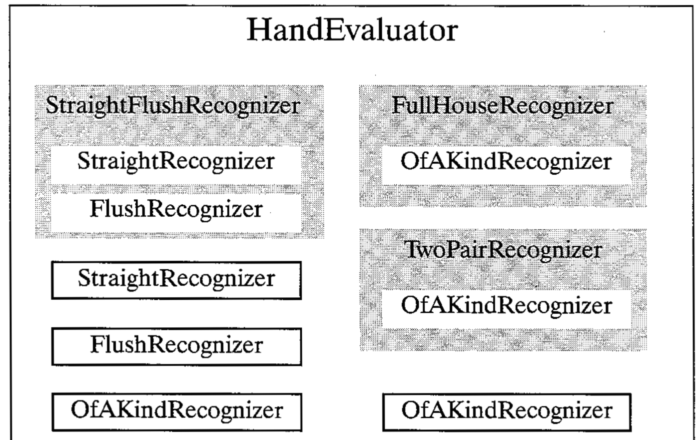


Fig. 4. HandEvaluator Object

To determine a winner, the OOM solution employs a special-purpose HandEvaluator object containing a series of Recognizers. The StraightFlushRecognizer specializes in recognizing straight flushes and encapsulates its own StraightRecognizer and FlushRecognizer. The OfAKindRecognizer is used for two, three, or four of a kind, and so on.

Actually, an alternative OOM solution is simpler in its initial state. Instead of the Recognizer classes, it would implement a HandEvaluator similar to the $$WEIGHT function used by the procedural solution. It might still need to divide this functionality up into Recognizer classes later as reusability becomes more of an issue. This is an example of "graceful evolution."

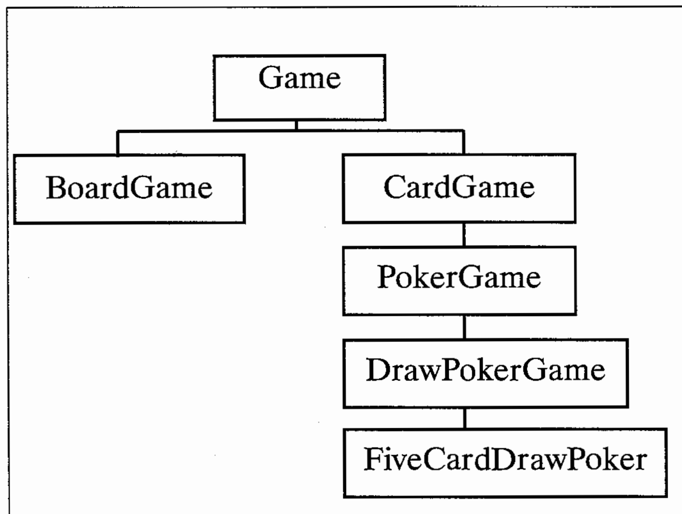


Fig. 5. CardGame Class Hierarchy

Finally, this OOM solution anticipates the development of future related applications by creating a class, CardGame, to store all the attributes and behavior common to all card games; PokerGame, to hold everything common to all kinds of poker, and so on. The class FiveCardDrawPokerGame inherits most of its capabilities from superclasses—the class itself just specifies the fact that there are five cards.

By now it should be obvious why there's a "design bulge" in the early stages of object-oriented development. Next we'll see why this early investment is time well spent.

## Complexity Wall: Unexpected Enhancements

How would these two designs fare if we tried to add the following features?

### *Five or Seven Cards*

Varying the number of cards is the least worrisome of the changes. In the procedural version, the hand recognition logic is generalized by using a FOR loop instead of a more direct approach. The OOM version creates a new class called SevenCardDrawPoker, specifying seven cards instead of five.

### *Jacks are Wild*

This certainly throws a wrench into the works in either case. It is much harder to recognize a flush because a jack-of-diamonds could be used as a spade to make the flush work. It also raises the possibility of new winning hands such as five-of-a-kind. A good solution takes into account the possibility that other cards might one day be wild. The OOM version benefits from built-in reusability. For example, modifying the OfAKind-Recognizer would certainly have an impact on FullHouseRecognizer and TwoPairRecognizer.

### *Other Poker Variants*

Most procedural implementations will have real difficulties incorporating different variants of Poker depending on how the code was originally written. In my experience, most programmers will simply clone code such as the $$WEIGHT function and customize it for each different card game, thereby increasing future maintenance burdens. Particularly industrious programmers might try to abstract redundant functionality out into reusable functions and subroutines. However, even then it would not be surprising to find the same functionality appearing repeatedly in different ways throughout the system.

An exotic form of poker is much easier in an OOM solution. A new class, HouseRulesPokerGame, is created and positioned appropriately in the class hierarchy (see Fig. 6). It is then necessary to promote and demote certain capabilities of other classes: aspects of DrawPoker that apply to all poker games might need to be promoted to PokerGame, while aspects of PokerGame that do not apply to HouseRulesPokerGame might need to be demoted to DrawPokerGame. Promotion and demotion are usually just a matter of moving things higher and lower while generalizing or specializing the way things are handled.

What would be different about HouseRules-PokerGame? It might be a matter of changing the number of cards, the betting rules, the dealing/discarding procedures, or something about wild cards. It

might be a matter of adding and removing Recognizers within the HandEvaluator. In general, the changes will not be too problematic because this is exactly the kind of thing OOM is designed for.

## *Playing Blackjack or Go Fish*

This enhancement brings us very close to the complexity wall. The procedural poker game does not lend itself to being reused for other card games in which the rules and setup of the game are totally different. In most cases, much of the functionality for these games would have to be rewritten from scratch.

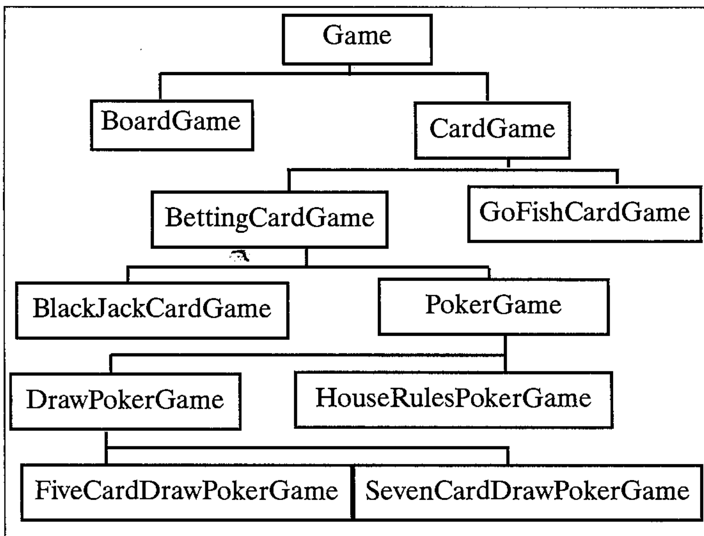


Fig. 6. CardGame Class Hierarchy

The OOM solution requires two enhancements:

1. Add the classes for GoFishCardGame and one for BlackJackCardGame. Note that we have also added BettingCardGame. Certain capabilities are demoted from CardGame, others promoted from PokerGame. Such class surgery is a snap in EsiObjects, because it's all point-and-click.

2. Implement the changes for the new games. Both games require new Recognizers, their own HandEvaluators, and new procedures for dealing and playing. But common aspects are inherited from superclasses, so the OOM programmer only needs to worry about the differences between each game. No complexity wall is encountered.

# Summary

This example aptly illustrates the differences between the two kinds of M. The traditional application is initially easier to create, but starts to fall apart when adding features and reusing code in new contexts. The object-oriented solution takes longer to get off the ground, but sails smoothly past the complexity wall. Many organizations are now taking a long, hard look at OOM because it offers these major advantages:

**Encapsulation:** Each object is entirely responsible for its own state and behavior; thus, if there's a bug in the way cards are dealt, it can only reside in the Dealer or Deck objects.

**Polymorphism:** Related objects implement the same behavior; in many cases, exactly the same code may be used to interact with a BlackJackCardGame and a HouseRulesPokerGame.

**Inheritance:** Code that is common to all forms of poker can be stored in the class PokerGame, where it is inherited by all subclasses. Anything at PokerGame, that is not appropriate for any of its subclasses, may instead be overridden (or deemed "private") at a lower level.

Encapsulation enforces ***modularity*** and reduces problematic dependencies between objects. Polymorphism and inheritance enforce ***reusability***. Of course, traditional programming systems do not prevent one from writing modular, independent, and reusable code, but the features of OOM serve to extend such coding efforts well beyond the Complexity Wall. ***M***

---

Erik Zoltán is a freelance consultant who has been programming, writing and teaching in the M community for the last 6 years. He has also done extensive work with the EsiObjects OOM programming system for ESI Technology Corp.

---