Object-Oriented Analysis and Design

Ivor Page
University of Texas at Dallas.

1 Design Objectives

Let's summarize the rules of thumb for good object-oriented design. The programmer is concerned with writing class definitions, while at run time, it is appropriate to talk of objects and their interactions:

1. Each class should be self-contained, providing a complete service of some kind.
2. Each should have a simple and small interface couched in the language of the application domain. If this is so, users of the class are able to ignore the details of its internal structure and implementation and concentrate on the interface. We call this **abstraction**. See Fig. 3.
3. Classes should hide their data, we call this **encapsulation**, only providing a small number of controlled methods or functions in the interface for accessing that data. These factors taken together enable **separation of interface from implementation**. The actual designs of the algorithms employed inside the classes can be changed as better, more efficient algorithms are found. After a design is complete, changes in a class implementation must not affect its interface. The effect on current and future users is then only in terms of efficiency: their applications need not be recoded to take advantage of the new class implementation. The advantages of separation of interface from implementation include rapid prototyping and graceful refinement. See Fig. 3.
4. The majority of the work at run time should be within objects, not in between objects.
5. The complexity of the system should be evenly spread amongst the objects.
6. Classes should be written with **reuse** in mind. This requirement impacts the interfaces, and may cause us to spend more time on design that we otherwise would.
7. Systems of classes should be written to solve a wide range of problems from the domain of the actual task at hand. We do not set out to write a program to solve a single problem, but a system of classes that solves a wide range of problems of similar type. The code that tailors the system to solve the specific problem at hand should be localized to a small number of classes.
8. Our designs should be easy to extend. We should ask how easy it is to add new data types and functionality. For example, in a 3D graphics program that deals with a small set of graphical objects, how easy is it to add graphical objects? If the program doesn't implement shadows, how easy will it be to add them (assuming the math is not too complex)?
9. We should try to minimize the number of classes in a design. This requirement may be subordinate to those of keeping the class interfaces small and simple, and of minimizing the inter-object communication.

2 Advantages and Disadvantages of OOP

If we follow the current OOP methodologies, what benefits and pitfalls can we expect?

- Probably the most vitally important benefit to strive for is reuse. It is here that the financial gains will be the greatest. Being able to select fully proven classes from a library and plug them into a new application is likely to revolutionize the software business.
- Although we can expect an increase in analysis and design time, testing should become easier and less time consuming. Many logical errors will be found by the compiler because of the strongly typed nature of classes.
- If decomposition is successful, classes become so independent of each other that their implementation can be carried out by individuals who do not need to interact at all. Each part of the design is tightly specified in terms of its interface before implementation takes place.
- Some will argue that OOP languages are complex and take longer to learn. This is so if you compare C++ with C, for example. Assuming we can teach these new languages thoroughly, and teach software engineering discipline along with them, the gains in having a better educated programming community surely far outweigh the extra cost of education.
- Some say that programmers are less able to predict the execution time of their code when using OOP languages. This indicates lack of education. It is just as possible to estimate run time with C++ as with C. Smalltalk may present greater problems than C++ in speed estimation.
- Some say that their programs are larger and slower when OOP languages are used than when traditional procedural languages are used. There are good reasons why programs could be smaller at run time with OOP languages. Again, these are symptoms of lack of knowledge.
- Some say that OOP languages are moving targets. "Let's wait until there is a standard for C++." Why not assume that what we have now in C++ is unlikely to change radically, at least for a while, and try to take advantage of its benefits right now?

3 Analysis

3.1 Discovering the Classes

We start with the requirements specification, listing all the noun phrases. Our first example will be the solitaire card game.

Brief Specification:
There are multiple piles of cards, as shown below. The draw pile is face down unordered, while the discard pile is face up unordered. The game begins with pile p_1 *containing 6 cards,*

p_2 containing 5 cards, and so on, each P pile is unordered, face down. Each Q pile initially contains one card face up. The four S piles begin empty. The game is played in rounds. Each round begins with the player turning over a stack of three cards from the draw pile onto the discard pile, so that only the top card of the three is visible. If fewer than three cards are present on the draw pile, then all of them are turned over. A variety of subsequent actions can take place. A card can be moved from the discard pile to one of the Q piles, assuming each Q pile remains ordered, in decreasing order, with color alternating. A card can be moved from the discard pile, or from a Q pile, to a S pile, assuming each S pile remains ordered in increasing order, strictly starting at the Ace, with each S pile containing cards from just one suite. An entire Q pile can be moved to another Q pile assuming each Q pile adheres to the Q pile rule above. When a Q pile is empty, the top card of the corresponding P pile may be turned over, replacing the empty Q pile. Alternatively, if a there is a Q pile beginning with a King, or the top card of the discard pile is a King, it may be moved, replacing an empty Q pile. These card movements may continue until no further moves are possible, then the round is over.

The game is won when all the cards have been transferred to the four S piles. The game is lost when no movement of cards can be found during a sequence of rounds that cycles through the draw pile. At any time the player may resign.

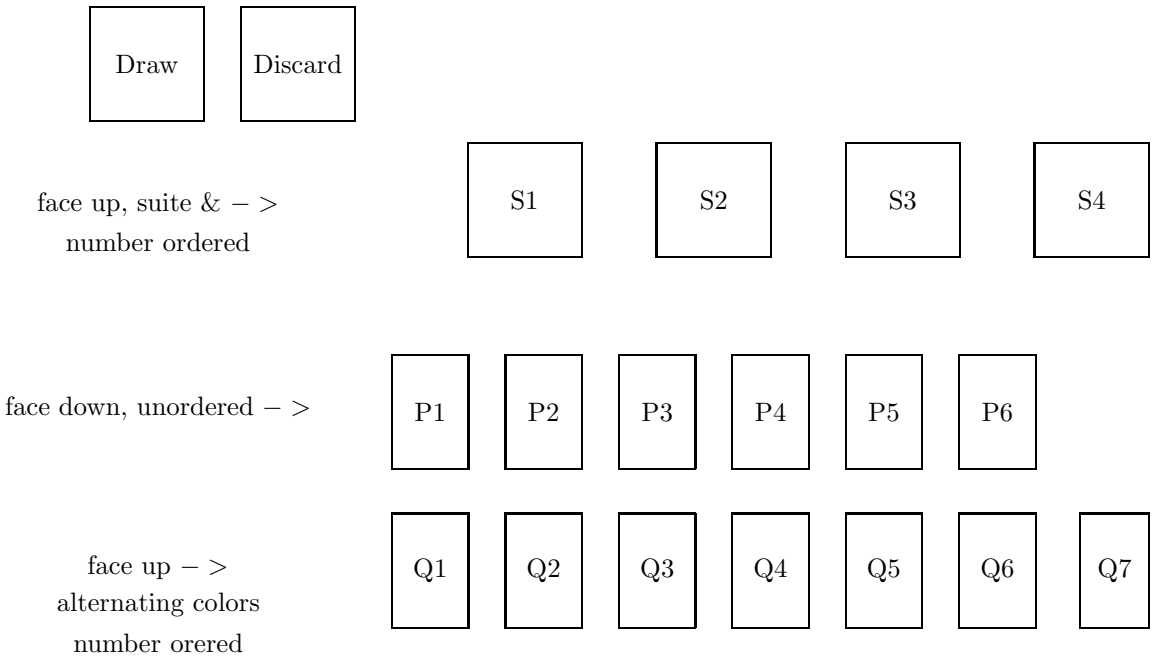

Figure 1: Card Game Layout

First pick out all the nouns and noun phrases appearing in the requirements spec: Assume we came up with the following:

Card	suite	color	number	face_up
face_down	Pile	Discard_pile	Draw_pile	S_pile
P_pile	Q_pile	Deck	User	Screen
Window	Player	Mouse	Red_card	Black_card
move_1_card	flip_3_cards	resign	Game_won	File_pile
Position	flip_card	King	Ace	
S_pile				

Now reduce this set according to simple rules:

- Remove vague names. If they are needed, we can reintroduce them later: File_pile, Position
- Resolve names with identical meanings: User = Player
- Remove names of things outside of the environment of the program: User, Screen, Mouse
- Remove names that are really attributes of other classes: Color, Number, Suite, Face_up, Face_down
- Remove names that are really functions or system inputs: Move_1_card, flip_3_cards, resign, flip_card, move_1_card
- Remove names that refer to states of the system. These may become attributes: Game_won
- Consider adjectival names carefully. Do they represent separate classes, or can the differences be managed by attributes? : There are several pile classes. Are they really different? We will assume for the moment that they are. But Red_card and Black_card are really just cards with different color attributes. Similarly King and Ace are really just cards with specific values for their number attribute.

This leaves:

Card	Pile	Discard_pile	Draw_pile	S_pile
P_pile	Q_pile	Deck	Window	

It is not clear at this stage that we have all the classes we need. We will discover missing classes later.

Now use 4x6 cards to describe each class that remains. If we cannot describe it, then maybe it shouldn't be a class. We have a subset of classes that have similar names: all the piles of cards. They differ so far only in their adjectival prefix. We need to consider if these classes really are different in some way. If not, redundant names can be removed. In our case, let's assume these piles are different. In fact, on closer examination, the Deck is another kind of pile. An obvious inheritance structure appears likely amongst these classes, where the kinds of piles all inherit a base class called Pile. This structure will be investigated later.

Here are some of the 4x6 cards:

Class:	Card: a playing card
Attributes:	
suite	enum of Hearts, Clubs, Diamonds, Spades
number	1 .. 13
face_up	Boolean
color	red or black
Member functions:	

Class:	Deck: 52 playing cards, all face down, random order
Attributes:	
deck_data	a data structure for holding Cards
Member functions:	

Class:	Q_pile: face up, alternating color, reducing order
Attributes:	
number_of_cards	gives how many Cards are in the pile
first_card	suggest a linked list structure
Member functions:	

As you can see, we have left the member functions blank for now.

After writing out all the classes with as much detail as we can, we should run scenarios from the game to see that our classes and their attributes are sufficient and are efficient. For example, do we want to have the attribute face_up within the Card class, or should it be an attribute of each of the card pile classes? In this game, all cards within one pile have the same value for this attribute, so the latter seems the most sensible. Perhaps there are card games, however, where piles have some cards that are face up and others that are face down. We will choose to make face_up an attribute of the card pile classes for now.

We also have color as an attribute of the Card class, but the color of a card can easily be derived from its suite, so color can be replaced by a function that computes the color. In fact it is only necessary to have a single integer, say value, $0 \leq \text{value} \leq 51$, to cover the suite, number, and color. The designer must choose the best arrangement of variables and functions to meet the needs of the project. If we use the single value idea, then we will need functions to compute color and suite. We will stick with two fields, number and suite.

In the Q_pile, we see the attribute first_card, which suggests a linked list structure for the pile. This is going too far at this stage. We should restrict our attention to the problem domain semantics and not worry about implementation details, so we remove this attribute for now.

We notice that there is no class that accepts the user's mouse clicks and interprets them. Clearly a user request, given as a sequence of mouse clicks, will result in a sequence of calls to the member functions of several objects at run time. We also notice that there is no obvious place to store the rules of the game. If we have functions such as flip_three_cards, then at least some of the rules will be distributed amongst the classes. We might elect to concentrate all the rules within one class in order to promote reuse of the rest of the game.

The remaining classes would form a **framework** for all similar card games; only the class containing the rules need change for different games.

We also note that, so far, no object is capable of declaring the game won or deadlocked. It may be that we will need an object that actually monitors the flow of the game, performing these checks and testing for a win or deadlock.

It might be worthwhile to combine the user interface class with the rules class, and the flow control class, since their functions are all inherently related to the user. For the moment, we will call this class the GUI class, for "graphical user interface." An alternative is to break out the rules functionality into a separate class, and send to it each user request. In many projects, the rules checker is implemented as a state machine, which holds a state transition table, and keeps the current system state. Each request is checked against the table to see if the corresponding transition is legal. In the card game, the state of the system is bound up in the states of all the card piles. It would be foolish to try to construct a complete state transition table for the entire game, since the number of possible states is extremely large.

However, we could still have a separate class for checking the rules. Each user request would be sent to this class, and it would send inquiry messages to the card piles implicated in the request to see if a legal move was being attempted. Object interactions for each user request would then become a two stage process, one to check for legality, and a second to implement the changes. Every pile object would need functions to answer both kinds of calls. We will keep to the one GUI object for now in the hope that in some cases a user request can be implemented with a single set of calls to the card piles.

A merger appears possible between Draw_pile and Discard_pile. There are some actions that take place between these two piles that can be hidden from the outside world. This merger simplifies the interface of the aggregate class. The remaining parts of the program need not know when the draw pile becomes empty and the discard pile is flipped over to make a new draw pile. There is a small snag here, however, since detection of a lost game is partly based on knowing when the draw pile is reestablished. Perhaps the flip_3_cards function can return a boolean to indicate that the entire draw/discard pile has been cycled without a single card being removed from the discard pile.

One further possible merger occurs between P_pile and Q_pile. If we expect more interaction between corresponding P and Q piles, than in between other piles, then the merger makes sense. Again, the interface to the aggregate of P_pile and Q_pile is simpler than the combination of the two separate interfaces.

Then we note that at run time there will be four S piles, six P piles and seven Q piles. How shall we address them? One possibility is to collect each of these three sets of objects into an array type container. We would then have S_pile[i], P_pile[i], and Q_pile[i]. This seems a satisfactory arrangement, somewhat along the lines of the problem domain, however, there is some confusion here because we wanted to aggregate corresponding pairs of P and Q piles. Let's make the container an array of Table piles. We'll call the container classes S_array, and Table_array. The S_array holds S piles, while the Table_array holds Table piles. Since the objects being held are different, we would use a template for the Array class, the parameter being the type of object to be held.

Good analysis and design comes from fleshing out all the choices available and selecting

the best combination we can find. Having arrays of S piles and Table piles seems to lack homogeneity. Perhaps the idea of having three array objects, one for each of the P piles, Q piles, and the S piles, is preferable. If so, the aggregation of P and Q piles is at best awkward and should be dropped. Since all three kinds of objects to be held in the arrays are kinds of piles (derived from the Pile class), there is no need for a template. Only one Array class is needed.

Given the desire to make the program framework cover as many card games as possible, we might need to consider making all card movements very trivial. For example, there could be just one basic card movement function that could move exactly one card from one pile to another. The flip_3_cards function would have to be written as a sequence of one card movements. The mergers of classes, as discussed above, might also have to be scrapped in the most generic version.

At this stage it may not be possible to estimate the cost of making the entire game completely generic. We would most likely continue the analysis phase keeping all options as open as possible, and make the decision later as to how generic we can afford to go. Decisions that increase the reuse potential of individual classes may run counter to those that make the entire program the most generic possible. A satisfactory compromise can usually be found, although it is important to document all the choices as they arise.

Here are the 4x6 cards with the changes:

Class:	Card: a playing card
Attributes:	
suite	enum of Hearts, Clubs, Diamonds, Spades
number	1 .. 13
Member functions:	

Class:	Deck: 52 playing cards, all face down, random order
Attributes:	
data	a data structure for holding Cards
Member functions:	

Class:	Draw_pile: face down, unordered
Attributes:	
position	position in window of top left corner of first card
number_of_cards	gives how many Cards are in the pile
data	a data structure for holding Cards
Member functions:	

Class:	Discard_pile: face up, unordered
Attributes:	
position	position in window of top left corner of first card
number_of_cards	gives how many Cards are in the pile
data	a data structure for holding Cards
Member functions:	

Class:	Source_pile: aggregation of draw and discard piles
Attributes:	
Member functions:	

Class:	S_pile: face up, single suite, increasing order
Attributes:	
position	position in window of top left corner of first card
suite	enum of Hearts, Clubs, Diamonds, Spades
number_of_cards	gives how many Cards are in the pile
data	a data structure for holding the cards
Member functions:	

Class:	S_array: container for 4 S_pile objects
Attributes:	
s	name of array holding the objects
how_many	number of S piles currently occupied
Member functions:	
operator[]	overload of [] for this container

Class:	P_pile: face down, unordered
Attributes:	
position	position in window of top left corner of first card
number_of_cards	gives how many Cards are in the pile
data	a data structure for holding the cards
Member functions:	

Class:	Q_pile: face up, alternating color, reducing order
Attributes:	
position	position in window of top left corner of first card
number_of_cards	gives how many Cards are in the pile
data	a data structure for holding the cards
Member functions:	

Class:	Table_pile: aggregation of p pile and q pile
Attributes:	
Member functions:	

Class:	Table_array: container for Table_pile objects
Attributes:	
t	name of array holding the objects
Member functions:	
operator[]	overload of [] for this container

Class:	GUI: graphical user interface, gets user inputs, validates them according to the game rules, checks for game over or deadlock
Attributes:	
state	enum of initial, game_in_progress, deadlock, game_over
Member functions:	

3.2 Architecture

Now that we have discovered the classes in our model, we might reflect on the architectural model that we have accidentally stumbled across. We have three kinds of classes, interface classes (GUI), control classes (combined into the GUI to check rules and control flow), and entity classes (those that support data structures closely related to the problem domain.) This division into three kinds of objects has been noted by many researchers. See Jacobson, Chapter 6. The classes which are usually the most stable are the entity classes (our pile classes). Designers will tend to agree about the designs of these classes. They closely model problem domain objects. The least stable classes relate to user interfaces.

The control structure tends to be the place where designers will differ the most. For example, in the card game, the rules checking functionality may be spread out amongst the classes in many different ways. We have chosen to lump all the rules in one place for ease of reuse. This is precisely the reasoning behind picking the three dimensions, interface, control, and entity. In any OOP a class can be considered as a point in this 3-space coordinate system. Each class will exhibit a certain amount of each of these traits. Some designers try to keep all their classes on just one of these axes:

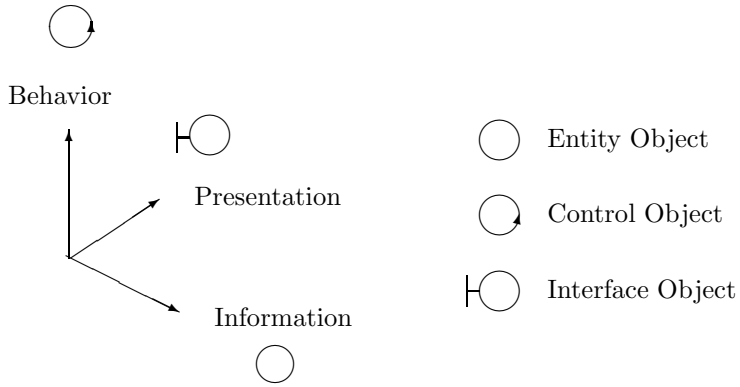

Figure 2: The Three Dimensions of Object Space

Jacobson suggests that designers should try to keep to these three basic types of classes, since they group the main types of behavior. All systems change. It is best to design systems such that changes are concentrated in one, hopefully small, space. By keeping all interface behavior in one set of classes, changes are localized to those classes, and since these classes only deal with interface behavior, there should be no complex web of code to untangle amongst code sections that are unrelated to interface matters. Experience with real projects suggest these three dimensions create systems which are easy to comprehend and simple to change. In order to conform to this model, our GUI class would have to be split into its interface and control components. This split may be worthwhile in order to evenly spread the amount of work amongst run time objects.

3.3 The System Interface

The next step is to discover all the system level operations (sysops): those operations or messages that come from the outside environment. In our case, we are mainly concerned about user inputs, although in a windows environment we would have to take care of *resize*, *repaint*, and many other messages. The list of sysops is usually inspired by the verbs appearing in the requirements specification.

Coleman advocates using 4x6 cards for what he calls the *interface schema:*

Operation	new_game
Description	empty all piles, make new deck and shuffle it, then deal.
Reads	state of game
Changes	all piles
Sends	prompt or cursor to user
Assumes	
Results	all piles left in new game situation

Operation	turn_3
Description	turn over top 3 (or all if less than three) from Draw to Discard.
Reads	Draw_pile
Changes	Draw and Discard piles
Sends	prompt, cursor, or error message if Draw is empty
Assumes	Draw pile is not empty
Results	top 3 (or all if less than three) from Draw are face up on top of Discard.

Operation	move_from_Discard
Description	take top from Discard and place on Q[i].
Reads	i, or infers it from mouse position, Discard, Q[i]
Changes	Discard and Q[i]
Sends	prompt, cursor, or error message if top of Discard does not go on Q[i]
Assumes	top of Discard is opposite color and 1 value lower than top of Q[i]
Results	top of Discard now face up on Q[i].

Operation	flip_Discard
Description	turn Discard face down and make it Draw, then empty Discard.
Reads	Draw, Discard
Changes	Draw and Discard
Sends	prompt, cursor, or error message if Draw isn't empty
Assumes	Draw is empty
Results	Discard is now empty and Draw is the inverted old Discard pile

Operation	move_to_S(i,j)
Description	Move top card of Q[i] to pile S[j]
Reads	i,j, or infers them from mouse positions, Q[i], S[j]
Changes	Q[i] and S[j] piles
Sends	prompt, cursor, or error message if top of Q[i] does not go on Suite[j]
Assumes	top of Q[i] is same suite and 1 value higher than top of S[j]
Results	old top of Q[i] is now face up on S[j].

Operation	move_Pile(i,j)
Description	entire Q pile Q[i] moved to the top of Q[j].
Reads	i,j, Q[i], Q[j]
Changes	Q[i] and Q[j] piles
Sends	prompt, cursor, or error message if bottom of Q[i] does not go on top of Q[j]
Assumes	bottom of Q[i] is opposite color and one less in value than top of Q[j], OR, Q[j] is empty
Results	Pile Q[i] has been moved to top of Q[j] and Q[i] is now empty

Operation	flip_P_Top(i)
Description	top of P[i] is flipped over and becomes Q[i].
Reads	i, P[i], Q[i]
Changes	P[i] and Q[i] piles
Sends	prompt, cursor, or error message if P[i] is empty or Q[i] is not empty
Assumes	P[i] is not empty and Q[i] is empty
Results	top of Pile Q[i] has become the only member of Q[i]

There are some assumptions missing. In most of the user events it is assumed that a game is under way. In the case of a new_game command, if a game is under way, we might ask if the user wishes to abandon the current game before trashing it.

The "Reads" field includes any input data contained in the sysop and any objects that must be read. The "Changes" field lists the objects that must be altered in some way to complete the sysop. These fields are useful in the design phase when deciding which objects are involved in completing a sysop.

Coleman suggests that the sysop descriptions should be expanded into what he calls life cycle models, or run time sequences of events. Each sysop leads to a sequence of calls to run time objects. The exact sequence often depends on the system state. He advocates a formal language for the life cycle models. It may be that pseudo code would be a better choice here, although it seems important to note that this part of the analysis should not be rushed, since bad choices here can lead to lots of unnecessary work later. Here are some of the sysops in Coleman's formal language:

turn_3 = Draw_pile_not_empty.
(get_top_of_Draw.flip_over.place_on_Discard)$^{3\ or\ all}$.

flip_Discard = Draw_pile_empty.
(get_top_of_Discard.flip_over.place_on_Draw)^{all}.

The period implies sequencing, x.y means do x, then y. There is also the alternation operator, x—y, meaning x or y, and the concurrency operator, x——y. The repetition operator, (actions)^n, shows how many times a certain sequence is to be repeated.

Notice that the life cycle model simply explains the sequences of actions that need to take place on the run time objects. It does not assign responsibilities to particular objects for carrying out these actions. This takes place during the design phase.

4 The Design Phase

Next we move into the design phase. The first consideration is the object interactions.

This involves identifying the subsets of objects needed to complete each sysop, or user request in the case of the solitaire game, and determining which object will be the *controller*, and which others will be *collaborators*.

Let's consider just a couple of the system operations. In the case of turn_3, only the

draw and discard piles are involved, but a check is necessary to determine if a game is under way. The same check is necessary in all but one of the user commands. It seems that the GUI object is the most likely choice for the controller here. After checking that a game is in progress, it simply sends a call to the Source_pile.

Consider the move_to_S(i,j) operation. The assumptions here require bringing data from 2 objects together in order to carry out the necessary checks. Shall we have the Q pile be the controller and the S pile be the collaborator, or the other way round? Why not give control to the GUI object? The idea is to make the object interactions somewhat like those of a client - server system. The controller object is a client, and the collaborators are servers. The servers supply relatively simple chunks of service (at least, they look simple in terms of the calls we make to them.)

For the Solitaire game, the desire for reuse is so great that we would likely assign control to the GUI object for all user commands. This object serves the role of supplying the user command and user display interface.

Yet another way of thinking about the interface is to consider the *actor/use case* model. An *actor* is not a person, but an external stimulus that the system must respond to. A single user can create multiple actors, one for each kind of user request. A *use case* is a course of events initiated by an actor specifying the interaction that takes place between the actor and the system. The complete collection of use cases forms the user interface specification. See Jacobson Chapter 7.

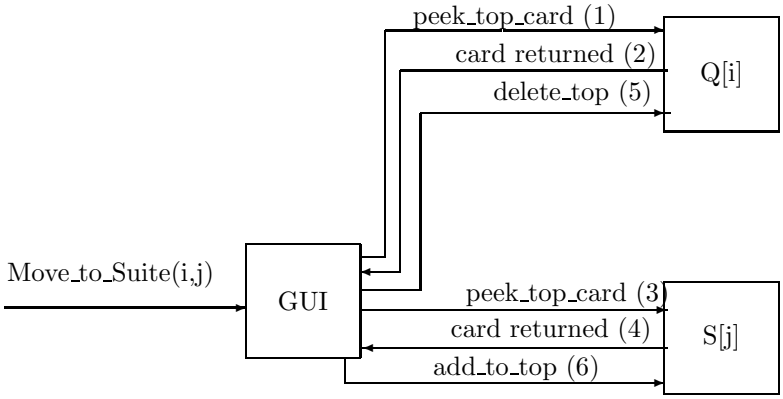

Figure 3: Object Interaction Diagram with Sequence Numbers

Clearly the interfaces between the objects need some careful work. At this point we can think about whether to take a copy of the 2 top cards into the GUI object with one call to each of the pile objects, or to get the necessary attributes of these cards, possibly with several calls (number and suite in each case). Another alternative is to have the 2 piles return pointers to the 2 top cards. Choices of whether to move pointers or objects should probably be left to the implementation phase, but alternatives should be noted whenever they arise. If the rules of the game were distributed amongst the objects, we could send the details of the top card of the Q pile to the S pile and ask if the card fits. If the reply was "yes," then the S pile would have already added the new card and the only remaining duty would be to send a delete_top_card message to the Q pile.

We would need an object interaction graph for each sysop. In this example there are few interactions that involve multiple objects in a complex way. The new_game sysop calls for action in every object, but it is a simple action, and there is little reason to worry about the ordering of these actions. We will see some other examples where the decisions are not so straightforward.

After finalizing the object interaction graphs, it should be possible to choose the class member functions. We will know the names of the functions, their arguments, and be able to specify the actions they perform, at least in terms of the problem domain. For the container classes, piles, and array type classes, at this point we would begin to consider the availability of classes from a class library. There is no point in designing yet another linked list class.

How can the GUI object declare that the game is won? We first have to decide whether to make the user move all cards individually to the S piles, or to have the system declare a win at the first opportunity and move the remaining cards from the Q piles to the S piles. In the former case, the GUI object could just count how many kings it had moved to the S piles. In the latter case, an algorithm for detecting a win would be written into the GUI object. This algorithm may imply some that extra functions be placed in the pile classes.

4.1 Inheritance

Finally we consider inheritance relationships. All the six card piles ARE piles, and so a highly general Pile class is clearly suggested. What do all the six piles have in common? They are just containers for sequences of cards. In each case, there can never be more than 52 cards in a pile. In some piles, there can never be more than 13, or some lower number. Inevitably we have the choice between a linked list and an array (although we should defer the decision until the implementation phase.) We should determine all the member functions (methods) of the classes in the design phase.

The objective in discovering inheritance relationships is to factor out the common features of the most derived classes and to move these up to parent classes. Opportunities for polymorphism depend on the problem domain. Here, a function, get_top_card() appears to be common to several classes. In this problem, this function is likely to be a member of the base class since it is needed by almost all the derived classes, and really has identical meaning in each case (it isn't needed in the S_pile class.)

The best way to arrive at the inheritance scheme is to consider all similar classes at the same (lowest) level, and then to factor out the commonality amongst both data attributes and the responsibilities (member functions) by building a tree from the bottom up. Say we have the following classes and attributes/responsibilities:

Class	Attributes	Responsibilities
A		v
B	w	v
C	w	v, x
D	w	v, y
E	z	v

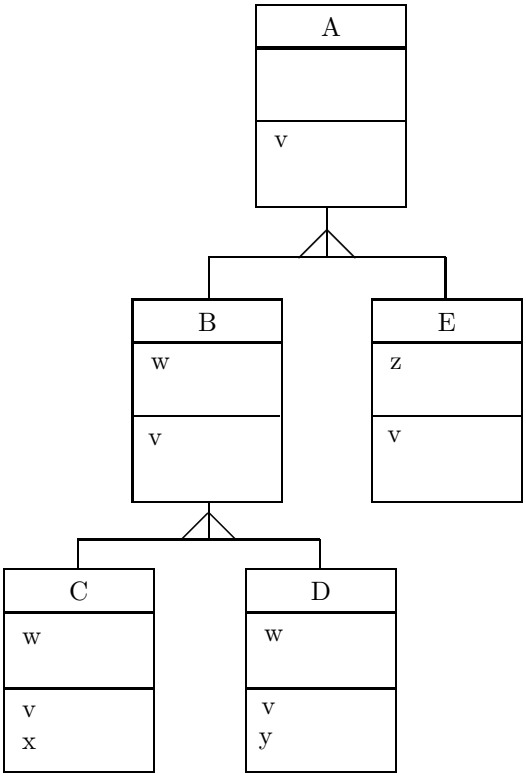

Figure 4: Inheritance Example

In this diagram, all the non-leaf nodes of the tree represent classes that are not needed. We will never make objects of their types. We call such classes *abstract* and can ensure in C++ that programs cannot make objects of their type.

Inheritance should nest properly:

Class	**Attributes**	**Responsibilities**
A		a, b
B		a', b, c
C		a, b, d

Assume class A is abstract. That is, we have no need of objects of this class.

Here the base class function "a" is overridden in class B and inherited in class C. It is better to move "a" out of class A, as follows:

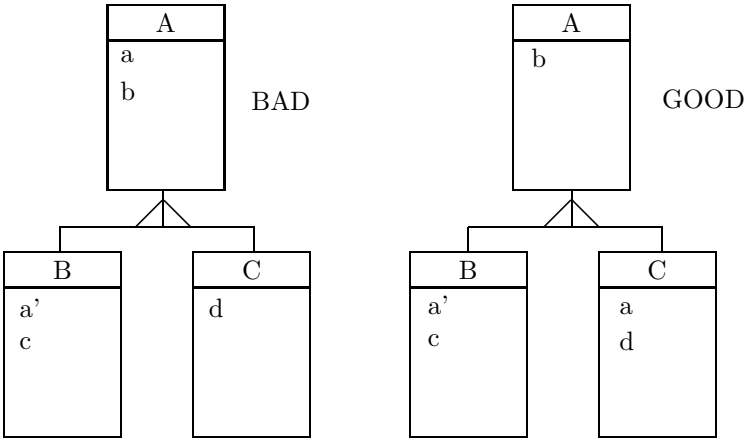

Figure 5: Good and Bad Inheritance Structures

It is also important that at each level of the inheritance tree, all classes at that level add the same kind of features to those at the previous level. Then, as each path is traced from the root to the leaves, the same kinds of developments take place level to level. This rule of thumb usually makes it very easy to add new classes. The appropriate level and position in the tree for the new class should be quite easy to find.

4.2 Invariants

Invariants are in a sense the rules of the operation of each object and of collections of objects, including, at the highest level, the entire program. We can write a large number for the solitaire game. For example, a card must not be in two places at once. There must always be exactly 52 (different) cards, each conforming to the rules for cards, i.e.
suite $\in$ {Hearts, Clubs, Diamonds, Spades}, $1 \leq$ number ≤ 13. Each kind of pile has its own rules, as enumerated earlier.

There may also be other kinds of predicates that could be checked. For example, certain functions have preconditions: a square root function requires that its argument's value be greater than zero, a matrix inversion function requires that the matrix is not singular, many graph theoretic operations require that the graph is acyclic. Which of these should we provide checks for? In the card game, we can ensure a certain invariant is met by making sure that each operation performed by the program follows a simple rule. If we create the Deck object correctly, and make sure that on every "movement" of a card, a deletion AND an addition takes place, and these operations are on the correct container objects, then we can neither create additional cards, nor drop cards.

In some cases, a check of the invariant will be at least as costly in time as the function to be performed (graph acyclicity for example), so it is important not to over-do these checks. However, some invariants lend themselves to simple predicates that can easily be programmed in as assertions. Most C and C++ compilers allow all assertions to be disabled when testing is complete. We should test everything that is easy to test, and document

those invariants that we chose not to test, just in case a simple test can be developed later.

5 Implementation

The actual process of implementation depends heavily on the choice of OOP language. We will briefly consider some general principles as related to C++.

When objects are on the move, i.e. passed as parameters to functions, it is usually faster to use "call by reference," which does not copy the values of the object's data onto the stack, but only places the address of the object on the stack. The general philosophy is, *make them once, keep them in one place until they aren't needed any more, then delete them.* In the card game, we would make the card objects once, placing them in the Deck object, then use pointers to the cards in the various data structures that hold cards (the other piles). The danger in this approach is that a bug could cause a card to be referred to by more than one container object (be present in more than one pile). Carefully placed assertions can check for this kind of error.

At this stage we discover that the Deck class has functionality that implies a different data structure from those of the other pile classes. The shuffle function is much simpler and faster to perform in an array than a linked list. The difference is so great as to force an array implementation for the Deck class. In the Deck class we do not need to add cards or add piles, and get_top_card is only needed during a deal sysop. The Deck class will hold all the cards throughout the game. Given the Deck class' interface, an array of pointers to cards seems appropriate.

Speed at run time is often critical and many believe that OOPs tend to be slower than procedural programs. There are some areas where speed can be lost in C++ if care isn't taken. For example, parameters ought to be passed by reference whenever possible. All short member functions should be *inline*, unless there are many independent calls to them. Care must be taken with inheritance structures when derived objects are frequently created and destroyed. Deep inheritance structures can lead to chaining of calls to constructors and destructors.

The way data structures are implemented depends on the above philosophy. Do data structures (containers) hold actual objects that are copied into and out of the structure, or do they hold pointers to them? The answer will mostly be to use pointers for reasons given above. In some cases, classes will declare constant or variable objects of some user defined classes. Since these member objects cannot be destroyed until the parent object itself is destroyed, it must be clear from the requirements specification that the lifetimes of the objects and their parent be identical.

There are some specific constraints in C++ (and all languages) that enforce a certain kind of implementation. For example, in C++, an array of objects of class X is created by (implicitly) calling X's constructor once for each element of the array. Unfortunately each element of the array is instantiated with identical initial values. If you want the elemental objects to begin life with different values, then it is necessary to use an array of pointers to X, rather than an array of X's. Then the program explicitly runs down the array of pointers making the elements as needed.

For many well known data structures, the programmer is advised to consider using a class library. Containers form the major part of such libraries and tend to conform to a certain style. It is common for such classes to come in two parts, the container, which has almost no public interface, and an iterator that the user declares to access the container's contents. This style enables multiple iterators to be declared for the same container. Data can be inserted, deleted, and searched via such iterators.

In the case of the card game, the Pile class could either be implemented as an array or a linked list. The largest Pile is the Deck, containing 52 pointers to cards. There are a total of twenty piles in the game and it doesn't seem too wasteful to have 20*52 pointers in total. The alternative is to use linked lists (for all but Deck). It need only be singly linked, and we only do insertions and deletions on one end of the list. If a linked list is chosen, should we include a "next" pointer within the card class, or should the container class handle all the pointers? If the latter, then the container will maintain a linked list of "link" objects, each link holding both a next pointer, pointing to the next link object, and a pointer to a card. A container class of this type will usually be implemented as a class to hold the objects, and another to act as an iterator. The user declares one iterator for each "pointer" needed into the data structure:

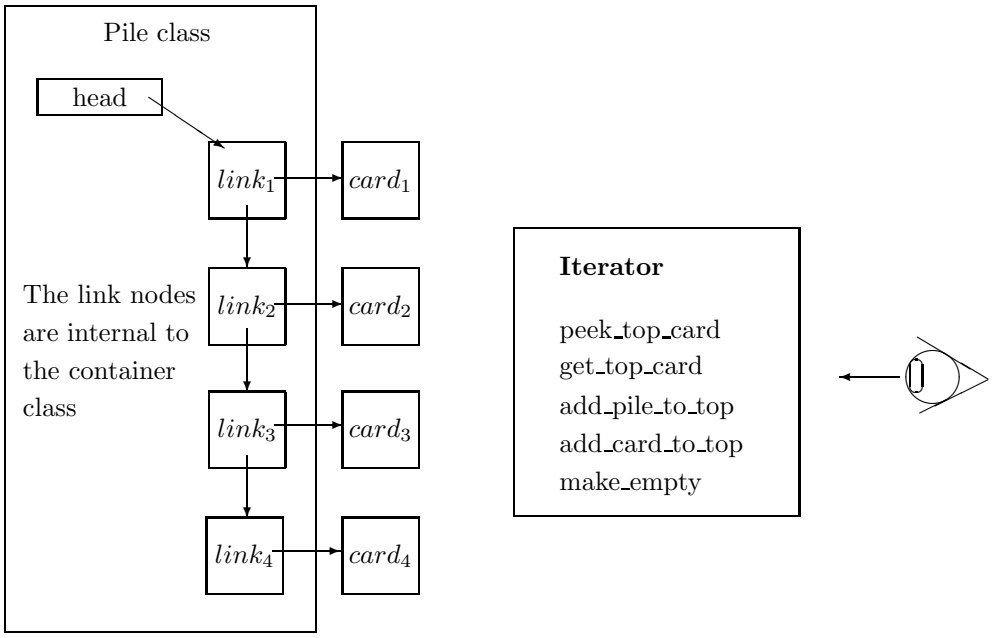

Figure 6: Linked List Container with Internal Links

Apart from memory space considerations, the advantage of the linked list over the array is in the speed of moving one Q pile onto another. With the list, it is only necessary to copy the head pointer of the list being moved into the tail of the receiving list.

We therefore end up with Deck being an array of pointers to cards, and the rest of the piles all inheriting a base class "Pile" which defines a linked list.

It is at this stage that reuse must be carefully considered. Each class, especially the containers, could be embellished in order to increase its reuse potential. For the linked list class, although our needs are quite trivial, we could add extra member functions to our class so that it becomes more attractive for reuse. The minimal requirement for reuse is that each class present a single coherent interface. We do not want to combine disperate operations that should be supported by two classes. In digital logic terms, an ALU combined with a large decoder would not be of use in very many projects. Most designers would need only one part of its functionality. In our case, the Pile class would have a very simple interface. If another project needs a similar class, it can inherit our Pile and add functions as necessary to build up the interface. We will say more about reuse in a later section.

6 Bibliography

1. Booch, G. *Object-Oriented Design with Applications.* Redwood City, Bengamin/Cummings.
2. Coleman, D. et.al. *Object-Oriented Development, The Fusion Method.* Prentice Hall.
3. Rumbaugh, J. et.al. *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ. Prentice Hall.
4. Wirfs Brock, R., et.al. *Designing Object-Oriented Software.* Englewood Cliffs, NJ. Prentice Hall.
5. Jacobson, I. et.al. *Object-Oriented Softwre Engineering.* Addison Wesley.