A Case Study In Object-Oriented Development:
Code Reuse For Two Computer Games

by

Roger E. Scott

Report submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

APPROVED:

Dr. Egyhazy, Chairman

Dr. Frakes

Dr. Haddad

September, 1992

Falls Church, Virginia
A Case in Object-Oriented Development: 
Code Reuse for Two Computer Games

by

Roger E. Scott

Committee Chairman: Csaba Egyhazy
Computer Science

(ABSTRACT)

A case study of the object-oriented development of two computer games using commercially available products was conducted. The games were constructed for use on Apple Macintosh computers using a C++ like programming language and an accompanying object-oriented class library.

Object-oriented techniques are compared with procedure oriented techniques, and benefits of object-oriented techniques for code reuse are introduced. The reuse of object-oriented code within a target domain of applications is discussed, with examples drawn from the reuse of specific functions between the two games.

Other reuse topics encountered in the development effort which are discussed: reuse of operating system routines, reuse of code provided by an object-oriented class library, and reuse of code to provide functions needed for a graphical user interface.
# Table of Contents

I. Introduction............................................................................................................. 1

II. Object-Oriented Approach Vs. Procedure Oriented Approach....................... 2

III. Code Reuse........................................................................................................... 7

- Code Reuse For A Graphical User Interface ..................................................... 8
- Code Reuse Within The ChessNet Computer Game ........................................ 9
- Reusable Code Created For This Project......................................................... 12
- Code Reuse And Domain Analysis .................................................................. 18
- Code Reuse For The Battleship Game ............................................................... 20

IV. Conclusions......................................................................................................... 22

References.................................................................................................................. 25

Appendices

- Appendix A - Object-Oriented Design Notation .............................................. 26
- Appendix B - Further Description Of The ChessNet Program ....................... 28
- Appendix C - Glossary ....................................................................................... 37
- Appendix D - Differences Between THINK C And C++ ................................. 38
- Appendix E - Source Code ............................................................................... 39
I. Introduction

Many years since its introduction, object-oriented programming has grown into a widely used technique. I had heard much about this method of organizing software development after reading articles about the subject and encountering object-oriented design methodology in a graduate level class; but prior to this project, had not had the opportunity to actually use it. The purpose of this project, therefore, is to design and develop an application using object-oriented techniques, and gain a first hand appreciation for any advantages or disadvantages of their use.

This case study consists chiefly of the development of an application used to play a game of Chess on Macintosh computers over a network. The application does not produce moves in the guise of a computer opponent, but instead the game (hereafter called ChessNet, which stands for chess on a network) provides an on-screen chessboard and pieces, with which the user can play an opponent running another copy of the game on a second Macintosh. The two machines are connected via an AppleTalk network.

A method was then devised to reuse the ChessNet design and code to develop a completely new application. This second application is a computer version of the game of Battleship\(^1\), which will also be a networked game played with two connected computers. The ChessNet application was developed using object-oriented design and coding techniques, and was implemented with the THINK C programming environment and its included Class Library. The development effort also required knowledge of Macintosh Toolbox routines (including QuickDraw drawing routines), and techniques for coding chess games.

This paper describes the design and construction of the ChessNet application. A preliminary non-object-oriented design is first contrasted with the object-oriented approach from the view of the application developer. Then the paper focuses on a frequently cited advantage of the object-oriented approach: ease of code reuse between applications. Meyer makes this claim: "Object-oriented design is the most promising technique now known for attaining the goals of extendibility and reusability."[1] Methods to reuse code using the object-oriented approach are introduced. In particular,

\(^{1}\) A popular board game, trademark of the Milton Bradley Company.
ways to reuse the design and code from the chess game to create the Battleship game are discussed. This second application involves two opponents taking turns attempting to sink each other's ships, placed strategically upon the playing board. This game is dissimilar enough from chess to be an interesting conversion, yet all of the objects in ChessNet will be reused. In addition to object-oriented reuse, this paper will also address reuse related to Macintosh operating system routines.

II. Object-Oriented Approach Vs. Procedure Oriented Approach

In order to show the nature of object-oriented design, it can be helpful to first look at an alternate type of design, an approach used by Booch [2]. Figures 1 - 3 show a simple design for a chess game, using the data flow notation of Gane and Sarson [3].

![Diagram of Alternate Dataflow Design](image)

*Figure 1*

With this approach, the design is first functionally decomposed into separate processes, which do some work on data moving in and out of the processes. Thus data flow diagrams show the movement of data through a program, represented here by rounded rectangular boxes for processes, lines with arrows which show the movement of
data, and open-ended rectangles which serve as data stores. The rectangle boxes in figure 1 represent devices external to the system. The data stores show a grouping of data, such as an array or a file located on disk, which exists for some meaningful duration during the execution of the program. This excludes passed parameters and local variables which exist only during the execution of a particular procedure. The arrows show movement of data usually passed as parameters when a procedure is called or when a procedure manipulates the contents of a data store directly.

Alternate Data Flow Design

Figure 2

This design begins with the "Game Controller" allocating and loading piece and square bitmaps into data store D1, so they will be available for drawing on the screen through the life of the program. Similarly the board configuration information in data store D2, inside the "Move Controller" process (figure 3) is allocated and initialized. The D2 data includes coordinates used for drawing the square on the screen and locating
mouse clicks, as well as the chess pieces stored in each square. The "Allocate & Initialize" process within the "Move Controller" sets the initial piece positions by calling the "Update Positions" process repeatedly. "Update Positions" stores the beginning move pieces in D2, and calls "Draw Board and Pieces" (back on Figure 2) to display the squares & pieces on the screen, using the D1 data store.

![Diagram of Move Controller Alternate Data Flow Design](Figure 3)

A move is initiated by the user clicking on a chess piece and dragging the cursor across the screen to another square. This mouse input is passed by the "Game Controller" process to the "Move Controller". Inside this process, "Change Cursor" alters the cursor to indicate that a piece is being moved. When the mouse button is released "Determine Move" calculates the "to" and "from" squares using the mouse input and the square coordinates in the D2 data store. "Is Move Legal" decides if the move is legal, using the piece positions also from D2. If the move is not legal nothing remains to be done (since the piece has not itself moved, just the cursor, nothing has to be undone) If the move is legal then it is sent in two directions: the first to "Update Positions" which updates the D2 data store with the move and calls "Draw Board and Pieces" to update the screen; the
second back to the "Game Controller" and through the "Network Controller" which sends the move across the network to the opponent's copy of the game. Lastly, a move coming in from the opponent is passed by the "Network Controller", through the "Game Controller" to the "Move Controller", where "Update Positions" again updates D2 and calls "Draw Board and Pieces" to update the screen.

The object-oriented design (OOD) for a chess game is introduced in the diagram of figure 4. Here the design is decomposed into objects, as opposed to processes. The objects contain methods, which are just processes in their own right, but more importantly, the objects also contain data structures. In figure 4, the top half of each object lists attributes or data stored in that object. The bottom half lists the methods related to that object. Relationships of one to one, one to many, or inheritance, are shown between the objects. A detailed description of this object oriented diagramming notation from Coad and Yourdon [4] is contained in appendix A. The piece and square bitmaps from the D1 data store in figure 2 are distributed between the "KindsOfSquares" and the "KindsOfPieces" objects. From the D2 data store in figure 3, coordinates placing a square in its window on the screen are moved to the "Squares" objects. Information relating which chess pieces are on which squares, again from the D2 data store, is also placed in the "Squares" object in the form of a "pointer" (currentPiecePtr) to a PiecesInUse object. If the pointer is null, then that square is empty. So in a sense, the data stores from figures 2 and 3 have been split apart and the processes have been wrapped around them to form the objects. Meyer states this in another way: "this law of inversion is the key to turning a functional decomposition into an object-oriented design: Reverse the viewpoint and attach the routines to the data structures. To programmers trained in functional approaches, this is as revolutionary as making the Sun orbit Earth."[5]
As a consequence of this re-orientation, there is more cohesion between procedures and data in the object-oriented design when compared with the data flow design. In ChessNet the data and the code that uses the data are associated explicitly in the object diagram as the attribute and methods of that object. In figures 2 and 3, the data stores exist as separate elements on the same level as the processes. Although the dataflow diagrams indicate now which processes access which data stores, it will be less obvious as the program is changed in a later maintenance stage, where within the code itself, it is less obvious what code goes with what data. Even without the OOD diagram, the code for ChessNet explicitly states the relationship between the attributes and methods of an object in a "header" file, where the object and its data are declared.

There is a benefit to having cohesion between procedures and data as Booch notes: "One side-effect of the functional decomposition is that all interesting data end up being global to the entire system, so that any change in representation tends to effect all subordinate modules. Alternately, in the object-oriented approach the effect of changing the representation of an object tends to be much more localized."[6] A change to data in an object will be more localized because it is only necessary to change the methods of that object’s class. Client code should only access that data through one of the methods of
that class, not directly. In the dataflow design, the procedures which we must modify due to changes in data representation may be more generally spread throughout the program. For example a change in the D2 data store may require changes in the following procedures in figure 3: "Determine Move", "Allocate Board Configuration", "Update Positions", and "IsMoveLegal". A change to the attributes in the Squares class in figure 4 would require changes only to the methods for that class. It is true that the number of methods requiring changes could equal the number of procedures, but at least the methods would all be confined to one place in the program.

There is a second benefit to grouping data with procedures: for the object-oriented approach it is more obvious where the data are allocated, initialized, and deallocated than in the data flow design. Though the first design has specific processes to perform these functions, when other data stores are added later they may be allocated and initialized elsewhere, as is left to the whims of the programmer. Object-oriented design imposes some organization by standardizing initialization. Each object should include an initializing method, which as standard practice would be called just after the object is created. Later users and maintainers of the program can count on the object-oriented program being organized in such a manner, and therefore can become familiar with the workings of the program more readily.

III. Code Reuse

While there is always a need for new applications to be created, whether driven by business needs or changing technology, the costs of developing new applications increases. This stems primarily from the personnel costs needed to design, code, test, and maintain new software. The reuse of existing designs and code is one way to reduce the expense of developing new applications.

Object-oriented programming offers advantages when reusing code developed by others. The first advantage involves the cohesion between code and data discussed in the previous section. It may be difficult to reuse code stored in libraries, in a non-object form, if this reuse requires major data structures to persist for significant periods of time through the execution of a program. The library would have to supply "create data" and "dispose of data" routines which put a burden upon the user to implement correctly. Alternately, "static" variables (available in languages such as C) could be used to save
the persistent data between calls to a function.[7] While use of static variables encapsulates data within a routine, they do not provide the flexibility of object-oriented techniques. Only one copy of a particular function and its static variables can be used (without duplicating the function and renaming the duplicate), whereas multiple instances of an object and its data can be easily created. Also data and code itself cannot be protected from being utilized by parts of the program which should not have access to it. With object-oriented programming, methods not meant to be used outside of an object can be kept "private", and thus protected from use in other parts of a program. Object-oriented classes also include initialize and dispose methods encapsulated along with the data, which are always called at the beginning and end of an object's existence. Thus there is no confusion as to when to initialize or dispose of data. A second advantage of object-oriented programming arises in the need to modify some of the reused code. "Inheritance" lets us replace portions of the provided code and data, while still reusing the bulk of it. Inheritance will be covered in more detail when discussing the code created for ChessNet.

**Code Reuse For A Graphical User Interface**

Another view of the difference between object-oriented development and functional decomposition, discussed earlier, focuses on the idea of 'state'. Booch says: "Because of the existence of state, objects are not input/output mappings as are procedures or functions. For this reason, we distinguish objects from mere processes which are input/output mappings."[8] The object does not just accept data as input, process it, and then output it, but holds onto it for some meaningful duration. Booch defines an object as "something that exists in time & space and may be affected by the activity of other objects. The state of an object denotes its value plus the objects denoted by this value. For example, thinking back to the multiple window system we discussed in the first section, the state of a window might include its size as well as the image displayed in the window (which is also an object)."

This idea of state relates to a graphical user interface (GUI). The types of capabilities provided by a GUI typically require state information to be saved for each instance of a certain part of the interface. For example, if a new window is added to our application we might need to save, in addition to information concerning the window's

---

2 i.e. in Booch's paper.
size and content, the kind of window (what it looks like and whether scrollable or not) and its current location on the screen. Its capturing of state information (and Booch’s example of a window system) suggests that the object-oriented approach would be especially useful in providing a means of reusing code for a GUI. This type of interface is a good candidate for reuse because, by its very nature, its use implies a standard way of doing things and standard methods should not have to be recreated. Use of a GUI suggests that certain portions of the application; using window frames to surround portions of the screen, presenting commands to the user in the form of pull-down items on a menu bar; or running a program by clicking on an icon, will all be done in standard ways. Since these identical functions are performed repeatedly by applications using the GUI, there is no reason to create new code to perform these functions when existing code can be reused.

Object-oriented technology has a special advantage when trying to reuse code to enable a developer to put features of a complex GUI interface into his application. Such interfaces require many data structures along with code which communicates between these structures. It would be difficult to supply these built in features to a developer without using object-oriented techniques, due to the large amount of entwined data and code associated with a GUI. If you want to add a new window using an class library, you need only create a new instance of a class, i.e. an object, and initialize it. Without a class library, you would have to call procedures to create data structures for the window, and then call other procedures to control the window. However, when using a class library where there is much interaction between objects, as with the THINK C class library, it can be difficult understanding how things actually work. The developer needs to determine to which objects some called methods belong, which can be difficult, especially with some classes inheriting methods four to five levels deep. (Also on the Macintosh, a called routine may not even be a method, but instead come from the Toolbox). These problems can be remedied with good documentation, which is always important, and tools to allow users to browse objects graphically.

**Code Reuse Within the ChessNet Computer Game**

We can separate the reuse of code within the ChessNet computer game into three distinct layers, and discuss each in turn. These layers of reuse, shown in figure 5, are as
follows: 1) system routines provided as part of the Macintosh operating system and described herein as the Toolbox; 2) pre-coded classes provided for the use of application programmers in the form of the THINK C Class Library; and 3) separate classes created as part of this project to provide reuse of specific functions between the ChessNet computer game and the Battleship computer game. A fourth type, the use of code resources, will be mentioned as an alternate form of reuse, related to number 1) above.

Reuse Layers

1) Macintosh O/S Toolbox Routines

2) THINK C Class Library

3) Code Created For Reuse Within Domain

Unique Code For ChessNet

ChessNet Code Layers

Figure 5

The first form of reuse in the ChessNet game is composed of routines which perform functions needed in the Macintosh environment. Known collectively as the Toolbox, portions of these are included in the read-only memory (ROM) of Macintosh computers. Though in ROM, a means is provided to update this code through subsequent releases of the Macintosh operating system. The Toolbox is a more conventional form of reuse in that it consists of a library of routines which require specific parameters, and return some value or perform some action, either internally or drawing to the screen. The subset of routines that draw to the screen is called QuickDraw. Many of these routines are called from within the methods in the THINK Class Library, but programmers may also use these routines directly as they see fit. However, it is sometimes difficult to determine the specific routine and its required parameters to use, for Apple documents them in a bulky series of manual that are extended (new volumes created instead of updating the old), as changes are made to the existing routines, or new ones added. Some form of on-line documentation, such as on CD-ROM, would definitely make using the Toolbox routines easier.
A related, but more minor, form of reuse in the Macintosh programming environment is the use of resources. These involve placing values into standard structures defined outside of the normal compilation process. The programmer uses a utility, such as ResEdit, to pull up a list of resources known to the Macintosh environment. He then defines a new instance of a needed structure, filling in specific values as required. The resource is added to the executable code. Within the program source code, the programmer calls other provided routines to load these structures, previously defined, into memory where they are actively used. For example, the programmer may define the structure for a window in ResEdit, filling in needed values such as window ID, type of window, initial size of the window, and initial location of the window on the screen. Within the program code, a Toolbox routine would be called to load this pre-defined window into memory, and cause it to appear on the screen at the designated location. These Resources provide a means of more easily reusing structures, such as menu items, windows, icons, and bitmaps, which are frequently needed when programming for a GUI environment. The reuse of these structures goes hand in hand with the reuse of code provided in the Macintosh Toolbox.

THINK C is a C language compiler for the Macintosh computer, which provides object extensions to the C language. These extensions make THINK C very similar to the C++ object-oriented language, although some differences remain between the two (see Appendix E). Basically, these extensions permit the following: the declaration of classes which contain attributes (data) and methods (functions); the linking of these into inheritance chains where a class will inherit the attributes and methods of those classes above it; the ability to create at run time instances of declared classes, or objects, whose attributes and methods may be accessed by other parts of a program through the use of pointers to those created objects. Along with the compiler, THINK C provides a class library which can be used at the programmers discretion. The THINK Class Library provides source code for a variety of classes which provide attributes, methods, and inheritance chains which users may link with their own code to build applications. Basically this library consists of objects which provide much of the functionality of the Macintosh interface. Figure 6 shows the THINK Class Library classes available for a programmers' use (not all of these classes were used in ChessNet).[8] The classes are shown in an hierarchical fashion (to show which classes are sub-classes of others) but the specific attributes and methods for each class are omitted. Typically, a majority of
these classes, not just one or two, would be used in an application, in order to provide the functionality of the Macintosh Interface.

Some examples of this functionality are: 1) the class "Switchboard" provides methods which comprise the event loop at the heart of every program operating within the Macintosh interface. This event loop chooses actions for each event received; whether originating from within the application or from outside such as those coming from another program or the user's mouse or keyboard input. 2) The combination of "Collaborator", "Bureaucrat", "View", and "Window" together provide the functionality for a window, while "Collaborator", "Bureaucrat", "View", and "Pane" together describe an area on the computer screen within that window. All of the classes have "Object" as a base class. As discussed earlier, the THINK Class library provides a good means of reusing code to provide the functions of a GUI such as the Macintosh interface. However, to accomplish this end, a majority of the classes must be used, not just a select few, because the classes have many interrelationships, and the inheritance extends up to nine levels deep. Most of the objects created from these classes send messages (i.e. call functions) to other objects from the THINK Class Library which are assumed to be present. This interaction between objects can at times become complex and its purpose is not always obvious. Therefore the use of a class library such as THINK C's does not come without a price. While it does give us a means of reusing code to perform the difficult yet standardized functions for an interface, time and study is required to understand the operation of the class library, especially where the library code interacts with new code written for an application.

Reusable Code Created For This Project

There are two types of code reuse provided for the Macintosh environment: the classes of the THINK Class Library used to perform the functions of a particular GUI; and the internal Toolbox routines called either directly, or indirectly through the code provided by the Class Library. An application programmer can use object-oriented code to add his own layer of reuse on top of these two layers. By planning carefully, new classes may be added which fulfill particular functions required in similar applications and thus may be reused across multiple applications. The area of functionality intended for reuse can be described as a "domain". These new classes may interact with the existing classes of the THINK Class Library or call Toolbox routines.
Figure 6  The THINK Class Library's class hierarchy
The ChessNet computer chess game can be broken down into three functional areas: 1) displaying a bitmapped graphic to some portion of the screen; 2) creating bitmapped screen objects which can be moved within a window with the mouse; and 3) sending move information across a network to the opponent’s copy of the computer game. These three functions, along with code specific to the chess game (such as that needed to determine legal piece moves), and the reused Toolbox and THINK Class Library code, is all that is needed for the ChessNet application. Code for the three functions above will be reused in the Battleship game. The description of this second game, along with an explanation of how these three functions are reused, while adding code unique to the Battleship game, will be described later. Figure 7 shows the classes used in ChessNet, and indicates which are from the class library (some are omitted), which were created for ChessNet but will be reused in Battleship, and which are unique to ChessNet. These differ slightly from the classes in figure 4, because classes were added to separate the reusable from the singular code. The ChessNet program is described further in appendix B.

The first functional area involves a class used to define an area on the screen, named "Square", which contains a pointer to the class used to store the bitmap for that square, "KindsOfSquares". To fulfill this function we only need code to create and initialize objects with the desired bitmaps and then draw the bitmaps to the desired location on the screen. The bitmaps must first be created in a drawing program and stored as resources, or data structures, which can be loaded into program memory during execution. As described earlier, the use of a resource editor to create data outside of the program compilation process, facilitates the reuse of these predefined structures. It is easier to deal with structures such as bitmaps in this manner rather than putting them into the program code directly. Code in the "Boards" class will create and initialize the desired number of "Squares" and "KindsOfSquares" objects, and will load the bitmaps into the "KindsOfSquares" objects. When a "Squares" object receives a draw command, it relays this message to its accompanying "KindsOfSquares" object, which actually draws the bitmap on the screen using a QuickDraw graphics routine.

The second functional area permits the user to click on and move a game piece around the window or game board. This also requires the loading of bitmaps as for the first function. Each piece must be created in a drawing program and stored as a resource which can be loaded into program memory during execution. Code in the "GameControl"
class will create and initialize the desired number of "PiecesInUse" and "KindsOfPieces" objects, and load the bitmaps into the "KindsOfPieces" objects where they are stored. The game pieces are set in their initial positions on the board by setting a pointer in the "Squares" objects to the appropriate "PiecesInUse" objects. When the user selects and moves a chess piece with the mouse, both new code and reused code from the THINK Class Library are combined to actually move the piece. The Class Library determines where the user has "clicked" on the screen, and which "Square" object needs to receive this notification. This "Square" creates a new object of the class "PieceMove", which controls the actual move, while receiving continual updates from the Class Library concerning the position of the mouse as it moves across the screen. The PieceMove object draws the game piece as it moves across the screen, while erasing the previous piece images and cleaning up the underlying squares to give the illusion of continuous movement.

The third functional area consists of simple network routines used to communicate between the two copies of the game on different computers. These routines access the Macintosh Toolbox, reusing several layers of code to actually send data across the network. Methods provided by the "GameControl" class perform two main steps. The first provides the ability to find and select the opponent's copy of the game and then link to it. The second step sends messages to the opponent's copy. These messages are composed in the form of "events", which are recognized as originating from the opponent's copy of the game. Attached to each event is a data structure, usually containing information about a move, which can be filled and processed as needed depending upon the implementation, i.e. whether used in the ChessNet or Battleship games.
The functions described above make up the portion of the code created for this project which is reusable within the chosen domain of "board games". Now the unique code must be added to the chess game to describe its particular set of rules. The inheritance feature of object-oriented programming can be used in this situation, to add unique code, while at the same time reusing code developed earlier. A sub-class can be added to an existing class such that the former inherits the methods and attributes of the
latter. To intercept method calls from elsewhere in the program, the sub-class can override methods in a super-class, by providing methods with duplicate names. Figure 8 shows an example where code is needed to support Networks A, B, and C for some hypothetical program.[10] With object-oriented programming, sub-classes for each new network can be added to provide code unique to one particular network, while the code common to all can be left in the super-class Network. Networks A and B may be able to use methods SendMove() and ReceiveMove(), and attribute moveData as they exist in the superclass Network. When we add Network C, which requires modified versions of these methods and the data structure, we can supply them in the sub-class Network C, where they will override the definitions in Network. When the user selects a network, an instance of the chosen network can be created, NetworkA, NetworkB, or NetworkC, which inherits methods and data from the superclass Network.

```
Object-Oriented Design

NETWORK

moveData

SendMove
ReceiveMove

NETWORK A

moveData

SendMove
ReceiveMove

NETWORK B

moveData

SendMove
ReceiveMove

NETWORK C

moveData

SendMove
ReceiveMove

Non-Object-Oriented Design

SendMove ( )
{            
    Case : Network A or Network B
    
    Case : Network C
    SendMoveForNetworkC ( );
}

ReceiveMove ( )
{            
    Case : Network A or Network B
    
    Case : Network C
    ReceiveMoveForNetworkC ( );
}
```

Figure 8

In a non-object-oriented program, some selection criteria is needed in each procedure that will have different code for various networks. Thus, both the SendMove() and ReceiveMove() procedures would need a case statement to switch between specialized code for Network C, and the general code for the other networks. The coding for "differences" ends up in separate places (each procedure) and is more difficult to keep track of. With inheritance, all the non-common code can be kept in one place: in overriding methods of the sub-class for the specific network; while the common code will be stored in the methods of the superclass. Another difference lies in that, in the
non-object-oriented program, the selecting between general and specific code may require the creation of additional similarly named procedures which perform the same function, but for a specific network. A case statement may call ReceiveMoveNetA(), ReceiveMoveNetB(), etc. This leads to a proliferation of procedure names. Within the object-oriented program names for methods which provide the same functionality may be reused. The method name ReceiveMove() can appear in the subclass Network C, where it overrides the inherited ReceiveMove() in the super-class Network. This is called 'overloading', and helps to reduce the proliferation of procedure names.

By using inheritance in the manner just described, the singular code required by the chess game can be added on top of the layers formed by the THINK Class Library and the reusable code developed for the domain of a "board game". Thus the "PieceMove2" in figure 7 is attached as a sub-class to the "PieceMove" class. Methods in "PieceMove2" override those in PieceMove, but instead of moving the piece these methods actually add the "rules" needed to determine legal moves in the game of chess. "PieceMove2" calls the methods within it's super-class, "PieceMove", to move the chess pieces. Likewise, "Network" is attached as a sub-class to the "GameControl" class. "Network" contains code that builds the data structure which is sent across the network. The specifics of this data structure will change from application to application. The parent "GameControl" class contains the generic method which actually sends the data across the network, but which does not need to change with different applications.

Code Reuse And Domain Analysis

The use of objects as "parts" for multiple applications in similar functional areas, such as in ChessNet, is an emerging concept for object-oriented development.[11] More emphasis will be put on developing each object to be flexible within its range of intended function. With this comes a de-emphasis on the development of the application as a whole. By concentrating on constructing a good set of re-usable, adaptable parts to be used by many applications, the development of the overall program may be simplified. The process is then reduced to picking out the proper objects for the job and putting them together. Software construction is envisioned as changing, along these lines, into an industrial enterprise, with applications being built in an assembly line mode where parts are put together. These ideas suggest that the important work will lie in
producing these parts, not in developing entire applications from scratch. Meyer puts this in the context of "top down" vs. "bottom up" approaches [12]:

Object-oriented design may be defined as a technique that, unlike classical (functional) design, bases the modular decomposition of a software system on the classes of objects the system manipulates, not on the functions the system performs. Classical approaches like functional top-down design (even, to a large extent, data flow analysis methods) require designers to first ask what the system does. Object-oriented design avoids such questions as long as possible, in fact until the system actually is run......

More importantly, top-down design goes against the key factor of software reusability because it promotes one-of-a-kind developments, rather than general-purpose, combinable software elements....

The bottom-up method promoted here does not mean that system design should start at the lowest possible level. What it implies is construction of systems by reusing and combining existing software.

Since the bottom-up approach de-emphasizes the overall system, more consideration must be given to its parts. A study of the intended domain of the objects to be reused will contribute to the effectiveness of each class as it is reused in some new application. If the planned reuse of each new class is not taken into account, it then increases the likelihood that the class is missing some needed attribute or method when used in another application. Kang comments: "The productivity and quality improvement from reusing components built for the purpose of reuse is much greater than that from components developed without reuse in mind. However, in order to build reusable components, the context in which the reusable components will be used must be understood and the reusable components must be designed to accommodate the contextual differences." [13] This study of planned reuse is called a domain analysis, which Prieto-Diaz defines as "a process by which information used in developing software systems is identified, captured, and organized with the purpose of making it reusable when creating new systems"[14] According to Kang, the outputs from a Domain Analysis do the following [15]:

- facilitate reuse of domain knowledge in systems development,
- define a context in which reusable components can be developed and the reusability of candidate components can be ascertained,
- provide a model for classifying, storing, and retrieving software components,
provide a framework for tooling and systems synthesis from the
 reusable components,
 allow large-grain reuse across products, and
 can be used to identify software assets.

No domain analysis has been attempted for this project, other than informally considering ways to reuse a few functional requirements from one application to another. However, the scope of this project is quite small with only three functional areas reused between two applications. Where more code is reused between a greater number of applications, a serious domain analysis effort would undoubtedly increase the likelihood of successful reuse.

**Code Reuse For The Battleship Game**

The game of Battleship starts with each player placing four ships on his own board. Neither player can see the other's ships. This is the preparation phase of the game, and will for the most part function the same way as the chess game: the user moves the pieces around the board with the mouse. After each player places his ships, they remain locked in their positions on the board. The second phase of the game then commences. Each captain takes turns indicating coordinates on the opponent's grid (actually one square on a board of 100 squares). If a ship occupies those coordinates it is identified as a "hit", otherwise it is a "miss". Each ship can accept from 2 to 4 "hits" (and thus overlays 2 to 4 squares) depending on the size of the vessel. A player loses and the game is over when all his ships are filled with "hits".

The domain classes from ChessNet can be reused for Battleship. The mechanism for moving the ships into position works the same as when the chess pieces are moved. One hundred squares must be allocated for the game of battleship whereas sixty-four were needed for the chess board. These squares will be initialized with a different size. Next different bitmaps for the squares and the ships will be required. These changes can be made by passing different parameters to the initializing methods. The complicated rules governing chess moves in "PieceMove2" can be replaced with simpler ones which
just restrict the ships to non-overlapping positions on the board. New code in "GameControl2" enables the player to signal to his opponent that his ships are in position and so is ready to start the second phase of the game.

**Battleship Class Hierarchy**

In the second phase, the game requires two views of the board. The first shows the player's own ships, and records any hits on them by his opponent. When it is his turn to fire, the player must flip to the other view of the board, where his hits and misses against his opponents vessels are recorded, but without showing the positions of the
vessels (their positions can be deduced as more hits are recorded which is the key for success in the game). The two views of the board may be accomplished reusing the ChessNet code by adding a new sub-class, "Squares2". "Squares" will store the first board view while "Squares2" holds the second. Instead of just two instances of "Kinds Of Squares", one black square and one white square, there will be three: an empty square, a "hit" square, and a "miss" square. In this second phase of the game, the opponent's shot, passed across the network, is actually just a square designator, and is recorded on the first board view as a miss in an empty square, or a hit on one of the ships. A shot is fired back by switching to the second board view, reviewing previous hits and misses against the opponent, and "clicking" on an empty square to return fire. The act of "clicking" on a target square and sending the "shot" across the network is handled by code in the "PieceMove2", "Network", and "GameControl" objects. An additional requirement of Battleship is that after a shot is fired, verification is needed to determine whether a hit was scored. This will be accomplished by storing the location of the enemies ships after the setup phase, using a "Squares2" attribute. These enemy ships would not be displayed to the user: their positions would be used internally to determine a hit or a miss.

IV. Conclusions

Functionality can be shared across applications using object-oriented methods in the following manner: place common code in reusable classes, while unique code for each application is placed in sub-classes. In ChessNet and Battleship, a layer of common classes representing functions needed by both are built upon two existing layers of code reuse. The first is a class library performing the functions of a graphical user interface. Secondly routines are provided for reuse as part of the operating system. On top of this three layer foundation is added the singular code which represents the "rules" of each game and make each game distinct from the other. The "rules" are attached in the form of sub-classes linked through inheritance to the classes composing the common elements of the two games. When some method in the reused classes is called, an overriding method of the same name may be called instead. This inherited method will perform some action according to the "rules" of the game and then call the overridden method to provide the common functionality as needed.
Some forethought is needed to replicate functionality across multiple applications. A study, called a Domain Analysis, is useful in this situation to ensure that code intended to be shared will prove useful when it is actually reused. This study determines not only what the code needs for reuse, but also where it will be reused, i.e. its domain. This needs to be defined in order for reuse to be consistently successful. The common code for ChessNet and Battleship could prove to be inadequate if used outside its intended domain: that of "board games". For example, the common code created for this case study would not prove sufficient for another more complex type of game such as a flight simulator.

When constructing a program which uses a graphical user interface, object oriented programming gives an advantage in productivity, especially for a developer unfamiliar with the interface. By creating objects using a class library designed for that interface, he immediately gains the methods and the data which provide its functionality. However, the interrelationships between objects from such a library can be complex. They may require some study, especially to understand the interactions between the class library and the application code. The developer will experience more independence in defining his own classes and objects the less frequently they communicate with those of the class library.

While there is no count for the code reused from the Toolbox operating system routines, the following is a breakdown of the approximate object sizes produced from source code for the other two layers of reuse, and the unique code for ChessNet:

<table>
<thead>
<tr>
<th>Game</th>
<th>Type of Code</th>
<th>Object size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChessNet &amp; Battleship</td>
<td>THINK Class Library for GUI</td>
<td>89,738</td>
</tr>
<tr>
<td>ChessNet &amp; Battleship</td>
<td>Reused code for common purposes</td>
<td>6948</td>
</tr>
<tr>
<td>ChessNet</td>
<td>Unique code for game &quot;rules&quot;</td>
<td>4778</td>
</tr>
</tbody>
</table>

These numbers show that an overwhelming majority of the code for these two applications has been reused in some form, or is available for reuse in another application. The object files from the THINK Class Library are very large because all the
methods for a class are linked in, even though many of the methods are not used. Thus use of the THINK Class Library will result in a program size of, at a minimum, approximately 90,000 bytes. Other factors affecting memory use are: 1) the hierarchical nature of classes results in deeply nested program calls which requires memory use in the form of calls and parameters being added to the "stack"; 2) multiple object instances (64 objects for the squares alone in ChessNet) may use more dynamic "heap" memory than procedure oriented programs which gather the data together in global data stores.

Despite these demands on memory usage, object-oriented techniques will grow in popularity as a means of reusing code in general, and for user interfaces in particular, especially as the cost of memory for small computers decreases. It use will also increase as more developers gain a working knowledge of these techniques. Though successful reuse of designs and code with object-oriented techniques requires much up front in terms of thought and planning, it should pay off in the long run with reduced development times and costs for successive applications.
References


Appendices

Appendix A - Object-Oriented Design Notation

The design of ChessNet was accomplished by loosely following a methodology set forth in the books: Object-Oriented Design, written by Peter Coad and Edward Yourdon. This design method uses the set of notation shown in Figure 10. Classes of objects are presented as rounded rectangular boxes divided three ways to show the class name on top, the attributes (or instance variables) in the middle section, and services on the bottom. (called methods by some - these are basically code procedures for the class) A box within a box represents a class and actual instances of that class, called Class-&-Object. A single box would represent just the class itself. This distinction is made because some classes, called "abstract", do not have instances of objects. Instead they occur in a hierarchy where the abstract class has sub-classes which inherit services and attributes. The sub-classes of abstract classes have object instances, but the abstract class, without any object instances of its own, exists only to provide services. The opposite, an object existing without a class, is meaningless and so not allowed.

The notation summary in Figure 10 shows several structures and connections. The Gen-Spec Structure is a relationship between generalized classes and more specialized, yet related classes. This Gen-Spec structure really represents an inheritance between classes, so the connecting line are drawn touching the inner-most box (if Class&Object), i.e. the class box. The Whole-Part structure is a relationship between objects, so the lines are drawn touching the outer-most, or object boxes. This structure shows that an object instance representing some "whole" is related to many other object instances of an entirely different class which represents some part of that whole. An instance connection is similar to the Whole-Part Structure in that it shows some relationship, perhaps one to many, between objects of different classes, yet the relationship has some meaning other than "part of some larger whole". A Message Connection is shown drawn between objects signifying that one object sends a message triggering a service in another object. Lastly, the large rectangle drawn around the entire page, with the numeral 1 in the corners indicates a "subject" area. Subject areas would be used to divide large, unwieldy projects into smaller more manageable partitions.
Figure 10: OOA/OOD notations

Subject or Design Component
(may be expanded or collapsed)

Note: In addition, Object State Diagrams and Service Charts may be used for specifying Services.
Appendix B - Further Description Of The ChessNet program.

Creating The Chess Pieces

Multiple steps are required in order to create and display the chess pieces on the screen in the ChessNet program. First, each piece was drawn using the MacPaint drawing program. Within this application each piece is not so much drawn as "marked". With MacPaint set at its maximum resolution the image is created by turning each pixel on or off. This allows the most control in creating the piece as it will appear as a bit map within ChessNet. As the magnification is lowered it appears as a smooth object, instead of a jagged construction of black and white blocks.

Next the image is copied and pasted into the "scrapbook", part of the operating system which facilitates moving data between applications. From there it is copied into yet another application, ResEdit, a resource editor for the MacIntosh, where the image is stored as a structure called a PICT. Resources are pre-defined structures for holding common elements of MacIntosh programs such as windows, menus, icons, and, in this case pictures. These are common throughout programs and using resources saves the developer from having to define the structures that will store each of these data items. Since they are pre-defined, he only needs to specify the contents.

Once compiled with the program code, the resources are available to be accessed by the code itself. In this case, the program uses a QuickDraw command to "draw" the image stored as a PICT into a simpler QuickDraw data structure called a BitMap. (These two structures are used because there is no BitMap resource to store the image when the program is not running) The image, now stored as a BitMap instance variable for an object, is available to be drawn and re-drawn on the screen while the program is executing. The actual process of drawing a BitMap onto the screen involves using a third type of QuickDraw data structure - the GrafPort. This structure is associated with a window on the screen and itself contains a BitMap along with other variables describing the drawing environment. To draw the chess piece within the window, the stored BitMap is copied to the BitMap contained within the GrafPort of that window, using a QuickDraw routine called CopyBits.
Coordinate Systems - Global, Window, & Local

In order to place the new chess pieces somewhere on the screen, we have to become acquainted with the coordinate systems used by the Macintosh Toolbox routines: global coordinates, window coordinates, and local coordinates. All three of these systems have vertical and horizontal axis, but unlike the common "x,y" geometric coordinates, these numbers run positively down the vertical axis away from the origin, as the horizontal axis runs positively to the right. The origin is usually placed in the upper left hand corner of the enclosure, whether that is the screen, a window, or some smaller part of the window. We do not deal with the global coordinates directly within ChessNet, which take in the entire area of the screen - or all of the "desktop". This is because after we create the window with a new instance of the "Window" class, initializing it with the window's starting position on the screen, the objects of the THINK C class library take care of updating the global coordinates to move the window around the screen. These objects also translate window and local coordinates into global coordinates.

The window coordinates measure the entire window, while the local coordinates measure panes, areas within the window, created using the THINK C class "Pane". Drawing the chess piece at a specific location within the window is not straightforward. Since the pieces are always drawn within the board squares, (moving pieces are described later) the pieces are placed on the screen using the coordinates of the squares. The "Squares" objects' window coordinates (its sub-class is "Pane") are set when the object is created and initialized, and saved as instance variables. Instead of passing window coordinates directly, another QuickDraw routine, SetOrigin, must be called just before CopyBits. We pass SetOrigin the Square's window coordinates, which it uses to convert the coordinates of the GrafPort from window to local coordinates (the window itself does not move). For example, if the square was set at (8, 8) within the window, the call to SetOrigin would change the GrafPort from window to local coordinates by setting the origin of the window within the GrafPort to (-8, -8). Subsequent calls to CopyBits, which always places the image at (0,0) within the window, are now inset to the proper location within the window because its origin has been changed. This procedure seems backwards from the more obvious method of just passing the window coordinates for where you want to draw to the drawing routine.
Initializing Objects

In order to use the objects at runtime within ChessNet they must be allocated, just like other data structures that are created dynamically. With THINK C, space for an object is created with the "new" function, similar to malloc. First a pointer to the object is declared, then "new" allocates space in memory for the object and returns a pointer to that object. Actually these pointers to objects are pointers to pointers and are called handles, but they will be referred to here as pointers. Handles are used so that the objects may be moved around in memory as needed by the operating system, without disturbing access to the object. Next, the initialization method must be called to fill the instance variables with starting values, most of which are passed as parameters to the method. If all objects are going to be created by other objects, obviously some set order must be devised, otherwise the objects could all be created from a procedure such as main. In ChessNet, main creates an instance of the THINK C "Application" class. This in turn creates the object "GameControl", which has as a superclass the THINK C "Document" class. "GameControl" is the central organizing object within ChessNet. It creates the main window (also an object), and sets the menu items. Next "GameControl" creates the main operational objects within ChessNet. The multiple "KindsOfPieces" and "PiecesInUse" are created and the initialize messages are sent to each new object. Then a "Boards" object is created. This object in turn creates and initializes the "KindsOfSquares" objects. Next "Boards" creates and initializes the "Squares" objects themselves, which require both the "KindsOfSquares" and the "PiecesInUse" objects to exist, so their pointers can be passed to the "Squares" where they are stored as attributes.

Creating Board, Initial Piece Setup, And Main Messages

After the main objects have been created and initialized, the square and the chess pieces in their beginning positions are displayed. First the chess board must display the starting chess pieces on their initial squares. The "Boards" object loops through the linked list of "Squares" objects, starting with the "Square" referenced by firstSqPtr. After creating and initializing the "Squares", "Boards" needs to assign the proper pieces to the proper squares. First the pointers to the "PiecesInUse" objects must be retrieved. "Boards" does this by repeatedly sending the GetPiecePtr message to "GameControl". This method retrieves the pointers of the "PiecesInUse" objects based upon an arranged numerical order. Next "Boards" assigns starting pieces to some of the "Squares" by sending them the SetPiece message. The Squares' SetPiece method places the "PiecesInUse" pointer into
the currentPiecePtr instance variable: if currentPiecePtr contains an address, then that square contains a chess piece; if currentPiecePtr is null then that square contains no chess piece.

Now the squares and pieces must be drawn. The "Boards" object again loops through the linked list of "Squares" objects, sending the Draw method to all of its by now properly initialized "Squares". Each "Square" object will in turn, in its Draw method, send Draw messages to its appropriate "KindsOfSquares" and "PiecesInUse" objects, if currentPiecePtr is not null. In turn, "PiecesInUse" passes the Draw message on to the proper "KindsOfPieces" object, depending upon what kind of chess piece it is. The actual drawing of the squares and the chess pieces on the screen is accomplished using the QuickDraw CopyBits routine, as described in the "Creating the Chess Pieces" section.

How Pieces Are Moved

The user moves a chess piece in ChessNet by moving the cursor with the Mouse to the piece being moved, pressing down and holding the Mouse button, moving the piece on the screen to the desired square, and then releasing the Mouse button. This simple action sets in motion a whole sequence of events within the ChessNet code. The squares and pieces involved in the move must be identified, the legality of the move within the rules of chess must be determined, the move must be stored or, if illegal, undone, and finally a successful move must be communicated to the opponent's copy of ChessNet.

The "Squares" objects have as a superclass the THINK C "Pane" class. This class describes an area within a window that can accept a mouse click, so by using "Pane" we can identify exactly where on the screen the mouse was clicked and then perform some action. The following describes the general mechanism by which the location of the mouse click is determined, and involves other classes provided by THINK C which communicate with one another. First of all the "Switchboard" class includes code which performs a typical "event loop" - constantly receiving events from the system, interpreting them and deciding where to pass them. Some of these events are ignored, while others are passed on to the specific code intended to handle that event. The "handler" would be included in other objects. For a mouse down event, the DoMouseDown method of "Switchboard" sends a DispatchClick message to the "Desktop" object, which determines which window the click was in and sends this same
message to the appropriate "Window" object. In turn "Window" has a list of all sub-views contained within the window. These sub-views are of the class "View" and have the sub-class "Pane", which in turn, for ChessNet, has the sub-class "Squares". So eventually the "Window" method DispatchClick finds the appropriate "Squares" object which will handle the mouse click, and sends it the DoClick message.

So when we click on a chess piece within a square, the DoClick message sent to a "Pane" object triggers the ChessNet code (as distinguished from the reused code provided by the THINK C class library) that actually moves the chess piece. We override the DoClick method provided in "Pane", by creating the "Squares" class, with its own DoClick method, as a sub-class of "Pane". This code creates and initializes a new object called "PieceMove", which is a sub-class of the THINK C class "MouseTask". We create this object to exist only for the life of the move, because a piece's movement is rather involved and we may wish to undo the move, so we need some place to store all the information that makes up the move. See Figure 11 for a description of the objects and messages involved in moving a piece.

After the "Squares" object (within its DoClick method) has created the "PieceMove" object, it sends the message TrackMouse to its enclosure. In this case the "Squares" enclosure is a window, so the message goes to the "Window" object. The TrackMouse method that is called is inherited from the "View" class, which "Window" is a sub-class of. As long as the user drags the chess piece across the screen, "View" will send a series of "tracking" messages to our "PieceMove" object. As parameters for these "tracking" messages, "View," sends the current point, the previous point, and the starting point at which the mouse was clicked. "View" knows which object to send the messages to because a pointer to "PieceMove" was included as a parameter in the call of TrackMouse. The tracking messages are BeginTracking, KeepTracking, and EndTracking. At the start of the mouse movement, "View" sends the BeginTracking message once to Piece Move. This method clears the currentPiecePtr of the square the piece is being moved from and updates the array of positions which is used to determined if the move was legal (this will be covered later). Next "View" sends the KeepTracking message to "PieceMove", and repeatedly sends this message as long as the mouse button is held down. This provides the mechanism by which the chess piece can be displayed as if it were a single object, constantly in motion across the screen. For each
Figure 11

ChessNet
OOD Diagram
KeepTracking message sent, this method draws the chess piece once on the screen, using the current point parameter to know where the cursor is on the screen and hence where to draw the piece. Drawing takes place by doing a QuickDraw SetOrigin call using the current point (which sets where the piece will be drawn) and sending the Draw message to the "PiecesInUse" object. In order to make the chess piece appear to move fluidly across the screen, the picture of the chess piece drawn in each previous call to KeepTracking must first be cleaned up. KeepTracking determines which squares the previous picture overlaid using the previous point parameter. KeepTracking sends the Draw message to these "Squares" objects, re-drawing the square and its contents, if any, which wipes clean the previous picture of the chess piece being moved - revealing the squares underneath.

When the user releases the mouse button as he drags the chess piece across the board, "View" sends the EndTracking message to "PieceMove". This method determines the "to square" which the piece is being moved to, again using the current point parameter. If the "to square" is a legitimate "Square" object, then its currentPiecePtr is set to the moved "PiecesInUse" object. If the "to square" is not legitimate, i.e. the mouse button is released with the mouse position off the board or outside the window, then the UnDo method is called. UnDo rolls back any effect of the move on the board state. Now the short life of the Piece Move object is almost at an end. It has one last job to do, before it is disposed of entirely. It must determine if the move just made was a legal move according to the rules of chess. If not, then UnDo is called, restoring the previous board state. If the move was legal, then nothing else is done. Either way, "PieceMove’s" function is fulfilled, and this object is disposed of. When the next move is made, a new instance of "PieceMove" will be created and initialized.
Figure 12 - Numbering system for square & pieces used to determine legal moves.

These following numbers represent the squares stored in an array. They are offset by +10 when referenced so they will fit into an array starting at 0. The middle numbers represent the actual squares on the board. The outer numbers represent border squares, which always contain -9 in the array item. When a move generates the -9 values, then the move is known to be off the board and is discarded.

-01 09 19 29 39 49 59 69 79 89 99 109
-02 08 18 28 38 48 58 68 78 88 98 108
-03 07 17 27 37 47 57 67 77 87 97 107
-04 06 16 26 36 46 56 66 76 86 96 106
-05 05 15 25 35 45 55 65 75 85 95 105
-06 04 14 24 34 44 54 64 74 84 94 104
-07 03 13 23 33 43 53 63 73 83 93 103
-08 02 12 22 32 42 52 62 72 82 92 102
-09 01 11 21 31 41 51 61 71 81 91 101
-10 00 10 20 30 40 50 60 70 80 90 100

The following numbers are stored in the array items referenced by the above values, and indicate the positions of the chess pieces on the board. White pieces are positive, black pieces are negative.

Empty Square 0
Pawn 1
Knight 2
Bishop 3
Rook, never moved 4
Rook, has moved 5
Queen 6
King, never moved 7
King, has moved 8
Border square -9
In ChessNet the current state of the chess board, i.e. where the pieces currently appear on the board, is stored in the currentPiecePtr for each "Squares" object. This pointer is either empty or points to the specific piece object that occupies that square. This scheme might be used in some manner to check for legal moves. However, the process of checking for legal moves can become quite complicated, what with the need to check not only the piece being moved but whether the king is left in check, a standard method used by existing chess games to generate legal moves has been used.[15] Under this scenario, an array is used with each array item representing a square numbered sequentially. Contained within each array item is a number representing a chess piece or an empty square. Using this method, chess moves can be generated simply by adding a value, or offset, to the array item number. For example 8 offset values can be added to the array item (i.e. "from" square) would give all the possible moves for a knight are: 8; -8; 12; -12; 19; -19; 21; and -21. These generated moves are called "pseudo moves", since they are not the finished list of legal moves for the knight. Some of the moves must be eliminated if they involve moving to a position occupied by your own piece, leaving your king in check, or moving off the board. Moving off the board is detected simply by using border squares which always contain the value -9. If a generated move results in a "move to" array item which contains the value -9, that move is immediately discarded. Figure 12 shows a complete listing of the values used to represent the pieces, and the board squares including the border (off the board) squares. Note that the values -10 through 109 are used so that the sequences 11.. 18; 21..28; 31.. 38; etc. will match up with the files on the chess board. These values must be offset by -10 to get the actual array items, which must start at 0.

All of the code checking for legal moves is unique to the chess game, so is stored in a sub-class of "PieceMove": "PieceMove2". The checking for legal moves starts after a "PieceMove" object has been created as described in the prior section. When the BeginTracking message is sent to "PieceMove" (actually "PieceMove2"), the array "arrayOfPositions", is filled with pieces information using the scheme just described. The program loops through the "Squares" object, checking the contents of each, and updating the arrayOfPositions accordingly. This array could be updated as each move is made, thus saving the time required to re-fill it as each move is made. However, the method used is simpler because the array can be stored in the "PieceMove2" object, which is created and destroyed with each move. The extra time required to re-fill the array each time does not add any noticeable delay while moving a piece.
Appendix C - Glossary

Class - A template for defining the methods and variables for a particular type of object. Methods, also called services, are procedures or functions associated with a class. Attributes are the data associated with a class, also called instance variables.

Class hierarchy - A tree structure representing the relationships among a set of classes. Lower classes in the hierarchy may inherit the methods and attributes of the classes above them.

Inheritance - A mechanism whereby classes can make use of the methods and variables defined in all classes above them on their branch of the class hierarchy.

Object - A run time instance of a class.

Overloading - The assignment of multiple meanings to the same method name, allowing a single message to perform different functions depending on which object receives.

QuickDraw - The subset of the Toolbox routines which implements drawing to the screen.

THINK C - A commercial programming environment for the Macintosh computer which supports the C language and object-oriented extensions to C.

THINK Class Library - A library of classes in the form of source code which may be compiled and linked into a program to provide support of the Macintosh user interface.

Toolbox - The software in the Macintosh ROM that implements the Operating System and the User Interface Toolbox. The User Interface Toolbox helps you implement the standard Macintosh user interface in your applications.
Appendix D- Differences Between THINK C And C++

The following is taken from the THINK C user's manual:

The syntax that THINK C uses for object is a subset of C++ as described in *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup (Addison-wesley). THINK C implements simple objects without multiple inheritance. THINK C does not implement any non-object extensions from C++ such as operator or function overloading.

**General Differences**

The main difference between THINK C and C++ is that, at its core, THINK C is an ANSI C compiler, and there are significant yet subtle differences between ANSI C and C++ besides the obvious ones. Keep in mind that C and C++ are completely different languages.

There are a few anomalous cases where something that works in THINK C might not work in C++, even if you've taken care not to use any of the THINK C extensions. Or there might be cases where a C++ program that restricts itself to the implementation of objects in THINK C will behave differently in THINK C. For guidance in these anomalous cases, look at section 18.2 "C++ and ANSI C" of *The Annotated C++ Reference Manual*. The scope rules of ANSI C and C++ are significantly different. Refer to section 3.1.2.1 "Scopes of Identifiers" of the ANSI C Standard (American National Standard for Information Systems--Programming Language--C ANSI X3.159-1989).
Appendix E - Source Code

/****
* Boards**.h
* Board **class for different types (probably views) of boards
* ****/
#include <CObject.h>
#include "Squares.h"
#include <CView.h>
#include "ChessNetDef.h"

class Boards : public CObject {
  public:
    struct Boards *nextBoardPtr; /* linked list of boards */
    struct Squares *arrayOfSqPtr[NUM_SQUARES-1]; /* array of pointers to all squares */
    struct KindsOfSquares *firstKindSqPtr;

  Construction/Destruction **/
  void IBoards(CView *anEnclosure, CBureaucrat *aSupervisor,
                Boards *nextPtrParm);
  void Dispose( void );
  void Rotate( void );
  void Draw( void ); /* re-draw all squares of board */

  /****
* Boards.c
* Board **class for different types (probably views) of boards
* ****/
#include "Boards.h"
#include <coops.h>
#include <stdlib.h>
#include <LongCoordinates.h>
#include "CChessNetDoc.h"

extern Network *gAGame; /* global ptr to A Game docurr */

/****
* This is your document's initialization method.
* If your document has its own instance variables, initialize them here.
* The least you need to do is invoke the default method.
* ****/

/* anEnclosure (usually window) gets passed as enclosure for Squares */
void Boards::IBoards( CView *anEnclosure, CBureaucrat *aSupervisor,
                      Boards *nextPtrParm )
{
  Rect unusedArea;
  match Draw() prototype - not used /*
  LongRect myLongRect; /* long rect needed for getframe call

  to CView */
Rect myRect; /* rect used to draw squares */

KindsOfSquares *arrayOfKindPtr[2]; /* pointers to KindsOfSquares */

struct squareData {
    short squareNumber;
    short ahEnc1;
    short sVEnc1;
    short color;
} arrayOSqData[] = {  
    11, 0, 315, BLACK,  
    12, 0, 270, WHITE,  
    13, 0, 228, BLACK,  
    14, 0, 180, WHITE,  
    15, 0, 135, BLACK,  
    16, 0, 90, WHITE,  
    17, 0, 45, BLACK,  
    18, 0, 0, WHITE,  
    21, 45, 315, WHITE,  
    22, 45, 270, BLACK,  
    23, 45, 225, WHITE,  
    24, 45, 180, BLACK,  
    25, 45, 135, WHITE,  
    26, 45, 90, BLACK,  
    27, 45, 45, WHITE,  
    28, 45, 0, BLACK,  
    31, 90, 315, BLACK,  
    32, 90, 270, WHITE,  
    33, 90, 225, BLACK,  
    34, 90, 180, WHITE,  
    35, 90, 135, BLACK,  
    36, 90, 90, WHITE,  
    37, 90, 45, BLACK,  
    38, 90, 0, WHITE,  
    41, 135, 315, WHITE,  
    42, 135, 270, BLACK,  
    43, 135, 225, WHITE,  
    44, 135, 180, BLACK,  
    45, 135, 135, WHITE,  
    46, 135, 90, BLACK,  
    47, 135, 45, WHITE,  
    48, 135, 0, BLACK,  
    51, 180, 315, BLACK,  
    52, 180, 270, WHITE,  
    53, 180, 225, BLACK,  
    54, 180, 180, WHITE,  
    55, 180, 135, BLACK,  
    56, 180, 90, WHITE,  
    57, 180, 45, BLACK,  
    58, 180, 0, WHITE,  
    61, 225, 315, WHITE,  
    62, 225, 270, BLACK,  
    63, 225, 225, WHITE,  
    64, 225, 180, BLACK,  
    65, 225, 135, WHITE,  
    66, 225, 90, BLACK,  
    67, 225, 45, WHITE,  
    68, 225, 0, BLACK,  
    71, 270, 315, BLACK,  
    72, 270, 270, WHITE,  
    73, 270, 225, BLACK,  
    74, 270, 180, WHITE,  
    75, 270, 135, BLACK,  
    76, 270, 90, WHITE,  
    77, 270, 45, BLACK,  
    78, 270, 0, WHITE,  
    81, 315, 315, WHITE,  
    82, 315, 270, BLACK,  
    83, 315, 225, WHITE,  
    84, 315, 180, BLACK,  
    85, 315, 135, WHITE,  
    86, 315, 90, BLACK,  
    87, 315, 45, WHITE,  
    88, 315, 0, BLACK  
};

/* counters for array indexes */
short i;
short j = 1;

/* store next board ptr */
this->nextBoardPtr = nextPtrParm;

/* create KindOfSquare (bitmaps) for black & white squares */
arrayOfKindPtr[1] = new(KindsOfSquares);
arrayOfKindPtr[1] -> KindsOfSquares(
/* Width of BitMap in pixels */
45,
/* Height of BitMap in pixels */
45,

/* Create offscreen port? */
TRUE,

"Black",

(KindsOfSquares *) NULL);

/* ptr to next kind in linked list */
arrayOfKindPtr[0] = new(KindsOfSquares);
arrayOfKindPtr[0]->IKindsOfSquares(
  /* Width of BitMap in pixels */
  45,
  /* Height of BitMap in pixels */
  45,
  /* Create offscreen port? */
  TRUE,
  "White",
)

arrayOfKindPtr[1];

/* ptr to next kind in linked list */
firstKindSqPtr = arrayOfKindPtr[0]; /* set first pointer for linked list */

/* draw bit maps for squares */
SetRect(&myRect, 0, 0, 45, 45);
arrayOfKindPtr[0]->BeginDrawing(); /* white square */
FillRect(&myRect, white);
arrayOfKindPtr[0]->EndDrawing();

arrayOfKindPtr[1]->BeginDrawing(); /* black square */
FillRect(&myRect, gray);
FrameRect(&myRect);
arrayOfKindPtr[1]->EndDrawing();

/* create and init 64 squares for this board */
for (i = 0; i < NUM_SQUARES ; i++)
  {  
    arrayOfSqrPtr[i] = new(Squares);
    (arrayOfSqrPtr[i])-ISquares(anEnclosure,"** for *** IPane */
    aSupervisor, /* CBureaucrat - for IPane */
    /* Width of square - IPane */
    45,
    /* Height of square - IPane */
    45,
    (arrayOfSqData[i]).aHencl,
    /* horz. pos.in enclosure IPane*/
    (arrayOfSqData[i]).aVEncl,
    /* vert. pos.in enclosure IPane*/
    sizFIXEDSTICKY, /* sizing option IPane */
    sizFIXEDSTICKY, /* sizing option IPane */
    (arrayOfSqData[i]).squareNumber,
    /* for ISquares */
    (arrayOfSqData[i]).color,
    /* BLACK/WHITE - for ISquares */
    (arrayOfSqData[i].color == WHITE) ? arrayOfKindPtr[0]
    : arrayOfKindPtr[1],
    /* pointer to proper b/w Bit Map */
    "TO:"
    /* board name - for ISquares */
  }
/* set pieces on starting squares
   pieces are numbered in order as they appear in PiecesInUse linked
   list
   white main pieces left to right, then black, then white pawns, then
   black,
   pass order to GetPiece() to get pointer */

/* start with white main pieces */
for( i = 0; i < NUM_SQUARES; i += 8 ) {
    arrayOfSqPtr[i]->SetPiece( gAGame->GetPiece( j++ ) );
}

/* Do black main pieces */
for( i = 7; i < NUM_SQUARES; i += 8 ) {
    arrayOfSqPtr[i]->SetPiece( gAGame->GetPiece( j++ ) );
}

/* Do white pawns */
for( i = 1; i < NUM_SQUARES; i += 8 ) {
    arrayOfSqPtr[i]->SetPiece( gAGame->GetPiece( j++ ) );
}

/* Do black pawns */
for( i = 6; i < NUM_SQUARES; i += 8 ) {
    arrayOfSqPtr[i]->SetPiece( gAGame->GetPiece( j++ ) );
}

/* allow clicks in each square */
/* draw each new square */
for( i = 0; i < NUM_SQUARES; i++) {
    (arrayOfSqPtr[i]->SetWantsClicks( TRUE );
    (arrayOfSqPtr[i]->Show();

    /************ NOT USING LONG COORDS - DEFAULT */

    /*
    (arrayOfSqPtr[i]->UseLongCoordinates( TRUE ); */
    ************

    (arrayOfSqPtr[i])->GetFrame( &myLongRect );
    (void) LongToQRect(&myLongRect, &myRect);
    (arrayOfSqPtr[i])->Prepare(); /* just added this */
    (arrayOfSqPtr[i])->Draw( &myRect );
}


#endif

/*******
 * Dispose
 *
 * This is your document's destruction method.
 * If you allocated memory in your initialization method
 * or opened temporary files, this is the place to release them.
 *   
 * Be sure to call the default method!
 */

void Boards::Dispose()
{
    short i;

    /* deallocate squares created in init */
    for (i = 0; i < NUM_SQUARES ; i++)
    {
        arrayOfSqPtr[i]->Dispose();
    }

    /* then delete the board */
    delete( this );
}

/*******
 * Rotate
 *
 * rotate board one quarter turn
 *
 */

void Boards::Rotate( void )
{
    short i;
    short h, v;
    Rect   myRect;

    /* reset coords */
    for( i = 0; i < NUM_SQUARES; i++)
    {
h = abs((arrayOfSqr[i] - hEnd - 315);
 v = abs((arrayOfSqr[i] - vEnd - 315);

 (arrayOfSqr[i] - Place( h, v, FALSE );

 /* then re-draw each square */
 for( i = 0; i < NUM_SQUARES; i++)
 {
     (arrayOfSqr[i] - Prepare();
     (void) LongToQDRect(&arrayOfSqr[i] - frame), &myRect);
     (arrayOfSqr[i]) - Draw( &myRect);
 }


*/


****

/* Draw
 *
 * draw all squares in board
 *
*****/

void Boards::Draw( void )
{
    short i;
    Rect myRect;

    /* re-draw each square making up board */
    for( i = 0; i < NUM_SQUARES; i++)
    {
        (void) LongToQDRect(&arrayOfSqr[i] - frame),
        &myRect);
        (arrayOfSqr[i] - Prepare();
        (arrayOfSqr[i] - Draw( &myRect);
    }

}

****

* CCChessNetApp.h

* Application class for a typical application.

*****/

#pragma once
/* Include this file only once */
#include <CAppliance.h>
#include "Switchboard2.h"

struct CCChessNetApp : CApplication {

public:
    /* instance variables */
    Switchboard2 *itsSwitchboard; /* override */

    /* methods */
    void ICChessNetApp(void);
    void SetUpFileParameters(void);

    void SetUpMenus(void);
    void UpdateMenus(void);

    void DoCommand(long theCommand);
    void Exit(void);

    void CreateDocument(void);
    void OpenDocument(SFReply *macSFReply);
    void MakeSwitchboard(void);

};

*****/

* CCChessNetApp.c

* Application methods for a typical application.

* Copyright © 1990 Symantec Corporation. All rights reserved.

*****/

#include "CCChessNetApp.h"
#include "CCChessNetDoc.h"
#include "ChessDef.h"
#include "Switchboard2.h"
#include "CCChessNetDoc.h"
#include "CBartender.h"

extern OSTYPE gSignature;

extern Network *gAGame; /* global ptr to A Game document */
extern CBartender *gBartender; /* Manages all menus */

#define kExtraMasters 4

#define kRainyDayFund 20480
#define kCriticalBalance 20480
#define kToolboxBalance 20480

void CCChessNetApp::CCChessNetApp(void)
{

    CApplication::CApplication(kExtraMasters, kRainyDayFund, kCriticalBalance, kToolboxBalance);

    /* The parameters to CApplication are the number of times to call MoreMasters, the total number of bytes of heap space to reserve for monitoring low memory situations, and the portion of the memory reserve to set aside for critical operations and toolbox calls. */

    Four (4) is a reasonable number of MoreMasters calls, but you should determine a good number for your application by observing the heap using Lightsbug, TMON, or Macbug. Set this parameter to zero, give your program a rigorous work-out, then look at the heap and count how many master pointer blocks have been allocated. Master pointer blocks are nonrelocatable and have a size of $100 (hex). You should call MoreMasters at least this many times -- add a few extra just to be safe. The purpose of all this preflighting is to prevent heap fragmentation. You don't want the Memory Manager to call MoreMasters and create a nonrelocatable block in the middle of your heap. By calling MoreMasters at the very beginning of the program, you ensure that these blocks are allocated in a group at the bottom of the heap.

    The memory reserve is a safeguard for handling low memory conditions and is used by the CMemory method in CApplication (check there for more comments). In general, your program should never request a memory block greater than this reserve size without explicitly checking in advance whether there is enough free memory to satisfy the request.

    */
}

void CCChessNetApp::SetUpFileParameters(void)
{
    /* SetUpFileParameters */
    /* In this routine, you specify the kinds of files your application opens. */
}

    void CCChessNetApp::SetUpFileParameters(void)
    {
        inherited::SetUpFileParameters(); /* Be sure to call the default method */
/*
** sfNumTypes is the number of file types
** your application knows about.
** sfFileTypes[] is an array of file types.
** You can define up to 4 file types in
** sfFileTypes[].
**
** sfNumTypes = 1;
** sfFileTypes[0] = 'TEXT';
/**

** Although it's not an instance variable,
** this method is a good place to set the
** gSignature global variable. Set this global
** to your application's signature. You'll use it
** to create a file (see CFile::CreateNew()).
**
** gSignature = ?????;    
}*/

void CChessNetApp::SetUpMenus()
{    
    inherited::SetUpMenus();    /* Superclass takes care of adding
        menus specified in a MBAR id = 1
        resource
    */

    /* Add your code for creating run-time menus here */
}

/**
** DoCommand
**
** Your application will probably handle its own commands.
** Remember, the command numbers from 1-1023 are reserved.
** The file Commands.h contains all the predefined TCL
** commands.
**
** Be sure to call the default method, so you can get
** the default behavior for standard commands.
**
***/
void CChessNetApp::DoCommand(long theCommand)
{
    switch (theCommand) {
        /* Your commands go here */

        case 1025:    /* rotate the board */
            gAGame->firstBoardPtr->Rotate();
            break;

        case 1026:    gAGame->ChooseColor();
        case 1027:    gAGame->GetOpponent();

            default: inherited::DoCommand(theCommand);
            break;
        }
    }

/**
** UpdateMenus
**
** Perform menu management tasks
**
***/
void CChessNetApp::UpdateMenus()
{
/* Enable standard commands */

/* Enable the commands handled by your Application class */

   gBartender->EnableCmd(1025);  /* Rotate board */
   gBartender->EnableCmd(1026);  /* Choose color */
   gBartender->EnableCmd(1027);  /* Link With Opponent */

/**
 * Exit
 * Chances are you won't need this method.
 * This is the last chance your application gets to clean up
 * things like temporary files before terminating.
 */

void CChessNetApp::Exit()
{
   /* your exit handler here */
}

/**
 * CreateDocument
 * The user chose New from the File menu.
 * In this method, you need to create a document and send it
 * a NewFile() message.
 */

void CChessNetApp::CreateDocument()
{

   /* CChessNetDoc *theDocument = NULL; */
   Network *theDocument = NULL; /* Network is subclass of CChessNetDoc */

   TRY
   {
      theDocument = new(Network);
   
   gAGame = theDocument;
   
   /* Send your document an initialization message. The first argument is the supervisor (the application). The second argument is TRUE if the document is printable */
   theDocument->ICHessNetDoc(this, TRUE);
   
   /* Send the document a NewFile() message.
      The document will open a window, and set up the heart of the application.
   */
   theDocument->NewFile();

   CATCH
   {
      /* This exception handler gets executed if a failure occurred anywhere within the scope of the TRY block above. Since this indicates that a new doc could not be created, we check if an object had been allocated and if it has, send it a Dispose message. The exception will propagate up to CSwitchboard's exception handler, which handles displaying an error alert. */

      if (theDocument) theDocument->Dispose();
   }

   ENDTRY;
}

/* ***
 * OpenDocument
 * The user chose Open... from the File menu.
 * In this method you need to create a document... */

and send it an OpenFile() message.

The macSFReply is a good SFReply record that contains
the name and vRefNum of the file the user chose to
open.

***/

void CChessNetApp::OpenDocument(SFReply *macSFReply)
{
    CChessNetDoc *theDocument = NULL;
    TRY
        
            theDocument = new(CChessNetDoc);

                
                    ///
                    /// Send your document an initialization
                    /// message. The first argument is the
                    /// supervisor (the application). The second
                    /// argument is TRUE if the document is printable.
                    ///
                
                theDocument->IChessNetDoc(this, TRUE);

                    
                        /// Send the document an OpenFile() message.
                        /// The document will open a window, open
                        /// the file specified in the macSFReply record,
                        /// and display it in its window.
                        
                
                theDocument->OpenFile(macSFReply);
        
    CATCH
        
            /*
                * This exception handler gets executed if a failure occurred
                * anywhere within the scope of the TRY block above. Since
                * this indicates that the document could not be opened, we
                * send it a Dispose message. The exception will propagate up to
                * CSwitchboard's exception handler, which handles displaying
                * an error alert.
            */

                if (theDocument) theDocument->Dispose();
        
    ENDTRY;
}
typedef short NetworkData[NUM_SQUARES-1];

class CCheessNetDoc : public CDocument {

public:

/* pointers to first objects in in linked lists */
  PiecesInUse   *firstPiecePtr;
  KindsOfPieces *firstKindPtr;
  Boards        *firstBoardPtr;

/* color of chess pieces for this player */
  short color;

/* Is it this player's turn to move? */
  Boolean myTurn;

/* pointer to pawn piece which just made double square move - eligible for en passant capture */
  PiecesInUse   *enPassantPtr;

/* used to link with opponent */
  TargetID      toTargetID;
  PortInfoRec   myPortInfo;
  AEDrec        *targetAddress;

Construction/Destruction **/

/* Altered by TCL Demo Weaver 1.0 (2/21/90) */

void ICheessNetDoc(CApplication *aSupervisor, Boolean printable);
void Dispose( void );

void DoCommand( long theCommand );

void UpdateMenus( void ); /* Altered by TCL Demo Weaver 1.0 (2/21/90) */

void NewFile( void );
void BuildWindow( Handle theData ); /*

Filing **/

PiecesInUse   *GetPiece( short pieceNumParm );
short GetPieceOrder( PiecesInUse *piecePtr );
KindsOfPieces *GetKindsOfPieces( short kindNumParm );

void LoadPiece( void );
void ChooseColor( void );
void GetOpponent( void );

void SendMoveToOpponent( NetworkData pieceNumbersPa );

}/

class Network : public CCheessNetDoc {

public:

Construction/Destruction **/

virtual void INetwork( void );

virtual void Dispose( void ); /*

Filing **/

virtual void ReceiveMove( NetworkData *pieceNumbers );

}
virtual void SendMove( void );

extern CApplication *gApplication; /* The application */
extern CBartender *gBartender;   /* The menu handling object */
extern CDecorator *gDecorator;   /* Window dressing object */
extern CDesktop *gDesktop;       /* The enclosure for all windo */
extern CBureaucrat *gGopher;     /* The current boss in the cha
extern OSType gSignature;       /* The application's signature
extern CError *gError;          /* The global error handler */

/* used by CChessNetDoc methods */
void CenterPic( PicHandle thePicture, Rect *myRectPtr );
void GetNSetDialogItems( short turnOnItem, short loopVar );

/* Document methods for a typical application.
Copyright © 1990 Symantec Corporation. All rights reserved.
*/

#include "CChessNetDoc.h"
#include <Global.h>
#include <Commands.h>
#include <CApplication.h>
#include <CBartender.h>
#include <CDataFile.h>
#include <CDecorator.h>
#include <CDesktop.h>
#include <CError.h>
#include <CPanorama.h>
#include <CScrollPane.h>

/* I added this */
#include <TBUtilities.h>
#include <CBitMap.h>
#include "ChessNetDef.h"
#include <CWindow.h>

#include <LongCoordinates.h>
#include <string.h>

#define WINCChessNet 500 /* Resource ID for WIND

template */

extern DialogPtr gDialogPtr; /* global dialog ptr */
extern Network *gAGame;       /* global ptr to A Game document */

/* Altered by TCL Demo Weaver 1.0 (2/21/90) */

void CChessNetDoc::ICHessNetDocCApplication *aSupervisor, Boolean printab{
    CDocument::IDocument(aSupervisor, printab);
    color = WHITE;
    myTurn = TRUE;
}

/* Dispose
This is your document's destruction method.
If you allocated memory in your initialization method
or opened temporary files, this is the place to release them.
Be sure to call the default method!
*/
void CChessNetDoc::Dispose()
{
    *loopBPtr = storeBPtr;
    *loopKPtr = storeKPtr;
    *loopPPtr = storePPtr;

    /* dispose of any objects for Boards, KindsOfPieces, & PiecesInUse */
    loopBPtr = firstBoardPtr;
    while (loopBPtr != NULL) {
        storeBPtr = loopBPtr->nextBoardPtr;
        loopBPtr->Dispose();
        loopBPtr = storeBPtr;
    }

    loopKPtr = firstKindPtr;
    while (loopKPtr != NULL) {
        storeKPtr = loopKPtr->nextKindPtr;
        loopKPtr->Dispose();
        loopKPtr = storeKPtr;
    }

    loopPPtr = firstPiecePtr;
    while (loopPPtr != NULL) {
        storePPtr = loopPPtr->nextPiecePtr;
        loopPPtr->Dispose();
        loopPPtr = storePPtr;
    }

    inherited::Dispose();
}

/**
 * DoCommand
 * *
 * This is the heart of your document.
 * In this method, you handle all the commands your document deals with.
 * *
 * Be sure to call the default method to handle the standard document commands: cmdClose, cmdSave, cmdSaveAs, cmdRevert, cmdPageSetup, cmdPrint, and cmdUndo. To change the way these
 * commands are handled, override the appropriate methods instead of handling them here.
 * *
 * ***/

void CChessNetDoc::DoCommand(long theCommand)
{
    switch (theCommand) {
    /* your document commands here */
    }
    default: inherited::DoCommand(theCommand);
    break;
}

/* Altered by TCL Demo Weaver 1.0 (2/21/90) */

/**
 * UpdateMenus
 * *
 * In this method you can enable menu commands that apply when your document is active.
 * *
 * Be sure to call the inherited method to get the default behavior.
 * The inherited method enables these commands: cmdClose, cmdSaveAs, cmdSave, cmdRevert, cmdPageSetup, cmdPrint, cmdUndo.
 * *
 * ***/

void CChessNetDoc::UpdateMenus()
{
    inherited::UpdateMenus();
    /* Enable your menu commands here (enable each one with a call to gBarbiener->EnableCmd(command_number)). */
}

/**/
* NewFile

When the user chooses New from the File menu, the CreateDocument() method in your Application class will send a newly created document this message. This method needs to create a new window, ready to work on a new document.

Since this method and the OpenFile() method share the code for creating the window, you should use an auxiliary window-building method.

*/

void CChessNetDoc::NewFile()
{

  /* Altered by TCL Demo Weaver 1.0 (2/21/90) */

  Str255 wTitle = "ChessNet"; /* Window title string. */

  /**
   ** BuildWindow() is the method that
   ** does the work of creating a window.
   **
   ** Its parameter should be the data that
   ** you want to display in the window.
   ** Since this is a new window, there's nothing
   ** to display.
   **
   **
   ** BuildWindow(NULL);
   **
   iteWindow->SetTitle( wTitle );
   **
   ** Send the window a Select() message to make
   ** it the active window.
   **
   iteWindow->Select();

  */

  /**
   ** BuildWindow
   **
   ** This is the auxiliary window-building method that the
   ** NewFile() and OpenFile() methods use to create a window.
   **
   ** In this implementation, the argument is the data to display.
   **
   **
   **
   void CChessNetDoc::BuildWindow (Handle theData)
   {
   
   Boards *boardPtr;
   /* used to create Boards object */

   Rect setSize;
   short i;

   /**
   ** First create the window and initialize
   ** it. The first argument is the resource ID
   ** of the window. The second argument specifies
   ** whether the window is a floating window.
   ** The third argument is the window's enclosure; it
   ** should always be gDesktop. The last argument is
   ** the window's supervisor in the Chain of Command;
   ** it should always be the Document object.
   **
   **
   ** iteWindow = new(CWindow);
   iteWindow->IWindow(WINDChessNet, FALSE, gDesktop, this);
   **
   ** After you create the window, you can use the
   ** SetSizeRect() message to set the window's maximum
   ** and minimum size. Be sure to set the max & min
   ** BEFORE you send a PlaceNowWindow() message to the
   ** decorator.
   **
   ** The default minimum is 100 by 100 pixels. The
   ** default maximum is the bounds of GrayRgn() (The
   ** entire display area on all screens.)
   */

}
** We'll use the defaults.

```cpp
setSize.top  = 360;
setSize.left = 360;
setSize.bottom = 360;
setSize.right = 360;
itsWindow->SetSizeRect(&setSize);

/* create & load KindsOfPieces & PiecesInUse */
LoadPieces();

/* create board & squares */
boardPtr = new (Boards);

firstBoardPtr = boardPtr; // set pointer to first object in linked list */

boardPtr->IBoards(itsWindow, this, (Boards*)NULL);

```
return( thisKindPtr );
}

void CChessNetDoc::LoadPieces( void )
{
    Rect drawRect;
    LongRect longDrawRect;
    short i, j;
    PiecesInUse *piecePtr, *storePiecePtr;
    *arrayOfPiecPtr[32];
    KindsOfPieces *arrayOfKindPtr[12];
    KindsOfPieces *aKindPtr;
    PicHandle arrayOfPicHandles[12];

    char *arrayOfPieceNames[] = {
        "WhitePawn \0",  "BlackPawn \0",
        "WhiteKing \0",  "BlackKing \0",
        "WhiteRook \0",  "BlackRook \0",
        "WhiteBishop \0", "BlackBishop \0",
        "WhiteKnight \0", "BlackKnight \0",
        "WhiteQueen \0",  "BlackQueen \0"
    };

    short arrayOfPieceNum[] = { 1, 0, 0, 7, 0, 0, -7, 0,
        /* WhitePawn */ -1, /* BlackPawn */
        /* WhiteKing */ 7, /* BlackKing */
        /* WhiteRook */ -7, /* BlackRook */
        /* WhiteBishop */ 2, /* BlackBishop */
        /* WhiteKnight */ /* BlackKnight */
    };

    /* BlackHorse */
    /* WhiteBishop */
    /* BlackBishop */
    /* WhiteCastle */
    /* BlackCastle */
    /* WhiteQueen */
    /* BlackQueen */
};

/******************************* create KindOfPieces ********************

/* create last KindOfPieces, then loop through others, from last to first */
arrayOfKindPtr[0] = new( KindsOfPieces );

arrayOfKindPtr[0]->KindsOfPieces( 45,
    /* Width of BitMap in pixels */
    45, /* Height of BitMap in pixels */
    TRUE, /* Create offscreen port? */
    /* Kind of piece */

    arrayOfPieceNames[0], (KindsOfPieces *) NULL,
    /* ptr to next kind in linked list */

    arrayOfPieceNum[0]); /* kind of piece by number */

for( i = 1; i < 12; i++ )
{
    arrayOfKindPtr[i] = new( KindsOfPieces );
}
arrayOfKindPtr[i]->IKindsOfPieces
/* Width of Bitmap in pixels*/
45,
/* Height of Bitmap in pixels*/
45,
/* Create offscreen port?*/
TRUE,
/* */
arrayOfPieceNames[i], /* kind of piece */
arrayOfKindPtr[i-1],
/* ptr to next kind in linked list */
arrayOfPieceNum[i]; /* kind of piece by number */
}

firstKindPtr = arrayOfKindPtr[11]; /* set first pointer for linked list */

for (i = 0; i < 32; i++) {
    /* load Pic for KindOfPiece */
    arrayOfPicHandles[i] = GetPicture( j );

    /* get original rect using GetBounds then center the picture -
     * this changes drawRect to centered coords */
    arrayOfKindPtr[i]->GetBounds( &longDrawRect);
    (void)LongToQDRect( &longDrawRect, &drawRect);
    CenterPic( arrayOfPicHandles[i], &drawRect );

    /* create mask region for KindOfPiece */
    arrayOfKindPtr[i]->maskRegion = NewRgn();
    OpenRgn();
    FrameRect( &drawRect);
    CloseRgn(arrayOfKindPtr[i-1]->maskRegion);

    /* load KindOfPiece with drawing of piece
     * also include frame around piece drawing */
    arrayOfKindPtr[i]->BeginDrawing();

    DrawPicture( arrayOfPicHandles[i], &drawRect );
    FrameRgn( arrayOfKindPtr[i]->maskRegion );
    arrayOfKindPtr[i]->EndDrawing();
}

/*************************************** create PiecesInUse *************/

/* create 32 PiecesInUse - linked list - start at end work backwards */
/* NOTE: loading black pawns and white pawns as PiecesInUse are crea
 * load other pieces individually below */

storePiecePtr = NULL;
for (i = 0; i < 32; i++) {
    piecePtr = new( PiecesInUse );
    piecePtr->PIPiecesInUse( 45,
    /* height */
    45,
    /* width */
    0,
    /* aHEncl */
    0,
    /* aVEncl */
    0,
    ((i < 8) ? arrayOKindPtr[i-1->maskRegion][i], /* KindOfPieces
     *itsKindPtr*/
    ((i < 8) ? BLACK:WHITE),
    /* short colorParm */
    storePiecePtr;
    /* PiecesInUse *nextPtrParm */
    storePiecePtr = piecePtr;
    arrayOfPiecePtr[i] = piecePtr;
}

/* set ptr to beginning of linked list */
firstPiecePtr = piecePtr;

/*************************************************************/
/* now load other PiecesInUse with non-pawn KindOfPieces */
void CChessNetDoc::ChooseColor( void )
{
    int dialogDone = FALSE;
    int itemHit = 1;
    int colorSelection;

    /* init dialog manager */
    InitDialogs( NIL_POINTER );

    /* initialize dialog */
    gDialogPtr = GetNewDialog( 401, NIL_POINTER, gAGame->mWindow
    /* GetNewDialog(short dialogID,void *dStorage,WindowPtr behind) */
    /* default selected item to queen */
    GetNSetDlgItemItem( 1, 3 );

    /* show dialog */
    ShowWindow( gDialogPtr );

    while ( dialogDone == FALSE )
    {
        /* save previous selection 'cuz at end will always point to save button */
        colorSelection = itemHit;

        ModalDialog( NIL_POINTER, &itemHit );

        switch ( itemHit )
        {
        case 1:

    /* now walk through all pieces again - setting the PiecesInUse pieceNumber to the appropriate KindOfPiece pieceNumber */
    for( i = 0; i < 32; ++i )
    {
       aKindPtr = arrayOfPiecePtr[i]->itKindPtr,
       arrayOfPiecePtr[i]->pieceNumber = aKindPtr->pieceNumber;

/*       black pieces .......*/
/* black castles */
arrayOfPiecePtr[16]->itKindPtr = arrayOfKindPtr[9];
arrayOfPiecePtr[23]->itKindPtr = arrayOfKindPtr[9];

/* black horses */
arrayOfPiecePtr[17]->itKindPtr = arrayOfKindPtr[5];
arrayOfPiecePtr[22]->itKindPtr = arrayOfKindPtr[5];

/* black bishops */
arrayOfPiecePtr[18]->itKindPtr = arrayOfKindPtr[7];
arrayOfPiecePtr[21]->itKindPtr = arrayOfKindPtr[7];

/* black queen */
arrayOfPiecePtr[20]->itKindPtr = arrayOfKindPtr[11];

/* black king */
arrayOfPiecePtr[19]->itKindPtr = arrayOfKindPtr[3];

/*       white pieces .......*/
/* white castles */
arrayOfPiecePtr[24]->itKindPtr = arrayOfKindPtr[8];
arrayOfPiecePtr[31]->itKindPtr = arrayOfKindPtr[8];

/* white horses */
arrayOfPiecePtr[25]->itKindPtr = arrayOfKindPtr[4];
arrayOfPiecePtr[30]->itKindPtr = arrayOfKindPtr[4];

/* white bishops */
arrayOfPiecePtr[26]->itKindPtr = arrayOfKindPtr[6];
arrayOfPiecePtr[29]->itKindPtr = arrayOfKindPtr[6];

/* white queen */
arrayOfPiecePtr[28]->itKindPtr = arrayOfKindPtr[10];

/* white king */
arrayOfPiecePtr[27]->itKindPtr = arrayOfKindPtr[2];

}
GetNSetDialogItems(1, 3);
break;

case 2:
GetNSetDialogItems(2, 3);

/* black */
break;

case 3:
HideWindow(gDialogPtr);
/* save button */
dialogDone = TRUE;
break;
}

switch (colorSelection) {
  case 1: color = WHITE;
    myTurn = TRUE;
    break;
  case 2: color = BLACK;
    myTurn = FALSE;
    break;
}

DisposeDialog(gDialogPtr);

waitForDialog(50000, gDialogPtr);

myErr = PCCBrowser("pLink with opponent's copy of game",
  "pChessNet",
  FALSE,
  &toTargetID.location,
  &myPortInfo,
  NULL,
  "p");

if (myErr == noErr) {
  toTargetID.name = myPortInfo.name;
  myErr = AECreateDesc(typeTargetID,
    &targetHandle
    sizeof( toTargetID )
    targetAddress);

  if (myErr != noErr) {
    /* SendMoves */
    sendPieceNumbersForSquaresAcrossNetworkToOpponent
    ***/
void CChessNetDoc::SendMoveToOpponent(NetworkData pieceNumbersParm)
{
  OSErr myErr;
  unsigned long dataLength = sizeof(NetworkData);
  Handle dataHandle;
  EventRecord myEvent;
  char *number = "0xGAME";
  char *extra;
  long result;
}
myEvent.what = kHighLevelEvent;
myEvent.message = strtol( number, &extra, 0);
myEvent.where = UNUSED /*

dataHandle = NewHandle( sizeof( NetworkData ) );
(void *) memcpy( *dataHandle, pieceNumbersParm, sizeof( NetworkData )
);

/*
myErr = PostHighLevelEvent( &myEvent,
toTargetID, sessionID,
&dataHandle ),
receiverIDIsTargetID );
/*
myErr = PostHighLevelEvent( &myEvent,
toTargetID, sessionID,
/* unused */
pieceNumbersParm ),
receiverIDIsTargetID );
DisposeHandle( dataHandle );
}

FFFFFFFFFFFFFF non-Method functions FFFFFFFFFFFFFFFF

void CenterPic( PicHandle thePicture, Rect *myRectPtr )
{ /* windRect not for a window in this case */
Rect windRect, pictureRect;

windRect = *myRectPtr;
pictureRect = (*(thePicture)).picFrame;
myRectPtr->top = (windRect.bottom - windRect.top -
(pictureRect.bottom - pictureRect.top))/2 + windRect.top;
myRectPtr->bottom = myRectPtr->top +
(pictureRect.bottom - pictureRect.top);
myRectPtr->left = (windRect.right - windRect.left -
(pictureRect.right - pictureRect.left))/2 + windRect.left;
myRectPtr->right = myRectPtr->left + (pictureRect.right -
pictureRect.left);
}

void GetNSelDialogItems( short turnOnItem, short loopVar )
{ int i;
int itemType;
Rect itemRect;
Handle itemHandle;

/* set in order of defines, such as: QUEEN_RADIO, ROOK_RADIO,
BISHOP_RADIO,
KNIGHT_RADIO */

for (i = 1; i < loopVar; i++)
{ GetDlgItem( gDialogPtr, i, &itemType, &itemHandle, &itemRect.
SetCtlValue( itemHandle, ( i == turnOnItem ) ? ON : OFF );
}

//******************** Interface for the KindsOfPieces Class

//******************** Interface for its

#include "ChessNetDef.h"
class KindsOfPieces : public CBitMap {
    /* Class
    Declaration */

    public:

    /* Instance
    Variables */
    char kind[10];
    RgnHandle maskRegion;
    struct KindsOfPieces *nextKindPtr;

    short pieceNumber;

    /* number representation of kind of piece
    1 - pawn
    2 - horse
    3 - bishop
    4 - rook that has not moved
    5 - rook that has moved
    6 - queen
    7 - king that has not moved
    8 - king that has moved

    Negatives indicate black pieces */

    /* Instance
    Methods */

    Contraction/Destruction */
    void KindsOfPieces(short width, short height, Boolean
    makePort, char *piece,
    *nextPtrParm, short pieceNumParm);

    void Dispose(void);

    void Draw( Rect *area );
    void Erase( void );

    Accessing */
    void SetRegion( Rect *area );

    };

    /*************************************************************************************************************/
    KindsOfPieces.c

    SUPERCLASS = CBitMap

    ******************************************************************************
    #include "KindsOfPieces.h"
    #include "Global.h"
    #include <string.h>
    #include <Quickdraw.h>
    #include <LongCoordinates.h>
    /** CONSTRUCTION/DESTRUCTION METHODS ****/

    /***************************************************************************************
    Initialize a KindsOfPieces object

    *******************************************************************************/
char *pieceParm, /* kind of piece */
KindsOfPieces *nextPtrParm, /* next kind in linked list */
short pieceNumParm; /* number representation of kind of piece */
{
    IBitMap( width, height, makePort);
    nextKindPtr = nextPtrParm;
    strcpy( kind, pieceParm );
    pieceNumber = pieceNumParm;
}

//*****************************************************************************/
// Dispose ( OVERRIDE )
// Dispose of a KindsOfPieces
******************************************************************************/
void KindsOfPieces::Dispose()
{
    inherited::Dispose(); /* Pass message on to superclass */
}

//*****************************************************************************/
// Draw a BitMap
******************************************************************************/
void KindsOfPieces::Draw( Rect *area ) /* parm now is unused */
{
    LongRect longRectArea;
    RectRect(area, &macBitMap->bounds, area);
    (void) QDToLongRect(area, &longRectArea);
    inherited::CopyFrom(&longRectArea, &longRectArea, maskRegion);
}

//*****************************************************************************/
// Erase region ( to erase previously drawn BitMap )
******************************************************************************/
void KindsOfPieces::Erase( void )
{
    EraseRgn( maskRegion );
}

//*****************************************************************************/
// ACCESSING METHODS /***/
******************************************************************************/
void KindsOfPieces::SetRegion( Rect *area )
{
    (**maskRegion).rgnBBox = *area;
}

//*****************************************************************************/
// Interface for the KindsOfSquares Class
******************************************************************************/
#pragma once /* Include this file only once */
#include <CBitMap.h> /* Interface for its superclass */
#include "ChessNetDef.h"

class KindsOfSquares : public CBitMap {
public:

};
Variables **/

    char             kind[10];
    RgnHandle        maskRegion;

struct KindsOfSquares  *nextKindPtr;

/** Instance

Methods **/

  /**
  Construction/Destruction **/
  void          IKindsOfSquares(short width, short height, Boolean
                     *nextPtrParm);
  void          Dispose(void);
  void          Draw( Rect *area );
  void          Erase( void );

  Accessing **/

    void          SetRegion( Rect *area);

/****** CONSTRUCTION/DESTRUCTION METHODS ****/

/*****************************/

KindsOfSquares.c

The KindsOfSquares

               SUPERCLASS = CBitMap

/*****************************/

#include "KindsOfSquares.h"
#include "Global.h"
#include <string.h>
#include <Quickdraw.h>
#include <LongCoordinates.h>

void KindsOfSquares::Dispose()
{
    inherited::Dispose();            /* Pass message on to
    superclass                */
} /* Dispose of a KindsOfSquares

/*****************************/

Draw

    Draw a BitMap
#pragma once /* Include this file only once */
#include "CChessNetDoc.h"
#include "ChessNetDef.h"
#include <CObject.h>
#include <PPCToolBox.h>
#include <AppleEvents.h>
#include <EPPC.h>

typedef short NetworkData [NUM_SQUARES-1];

class Network : public CChessNetDoc {

public:

Construction/Destruction /**/
virtual void INetwork( void );
virtual void Dispose( void );

Filing /**/
virtual void ReceiveMove( NetworkData *pieceNumbers );
virtual void SendMove ( void );
};

/****
* Network.h
*
* Functions needed to communicate with other copy of game
*
****/
#include "CChessNetDoc.h"
#include <PPCToolBox.h>
#include <AppleEvents.h>
#include <copts.h>
#include <stdlib.h>
#include <types.h>

void Network::INetwork( void )
{
    
    /***
     * Dispose
     *
    ***/

    void Network::Dispose( void )
    {
    }

    void Network::ReceiveMove( NetworkData *pieceNumbers )
    {
        short i;

        /* reset all pieces to squares as received from opponent */
        for( i = 0; i < NUM_SQUARES; i++ )
        {
            firstBoardPtr->arrayOfSqrPtr[i]->SetPiece( GetPiece( *pieceNumbers[i] ) );
        }

        /* redraw each square by drawing board */
        firstBoardPtr->Draw();
    }

    void Network::SendMove( void )
    {
        short i, order;
        NetworkData pieceNumbers;

        /* get & save all piece order numbers so as to send to opponent */
        for( i = 0; i < NUM_SQUARES; i++ )
        {
            order = GetPieceOrder( firstBoardPtr->arrayOfSqrPtr[i]->currentPiecePtr );
            pieceNumbers[i] = order;
        }

        /* collected piece order numbers - now send over network */
        CChessNetDoc::SendMoveToOpponent( pieceNumbers );
    }

    /***************************************************************************/
    CPieceMove.h

    Interface for the PieceMove Class

    /***************************************************************************/

#pragma once
    /* Include this file only once */

#include <CMouseTask.h>
    /* interface for superclass */
#include "Squares.h"
#include "PiecesInUse.h"
#include "ChessNetDef.h"

class PieceMove : public CMouseTask
    /* Class Declaration */

public:
    /***************************************************************************/

    Variables **/

    struct Squares
    *
        *arrayOrigSqrPtr[NUM_SQUARES_PER_MOVE];
    /***************************************************************************/

    /* store squares to undo move */
    struct PiecesInUse
    *
        *arrayOrigPiecePtr[NUM_SQUARES_PER_MOVE];
    /***************************************************************************/
/* store pieces to undo move */
short arrayOrigPieceNum[NUM_SQUARES_PER_MOVE];

/* store piece numbers to undo num */
short origSqArrayIndex;

/* store which piece or square you're pointing at */
short arrayOfPositions[120];  /* stores positions of pieces on board,
  10 x 12 grid - w/ border spaces to detect out of bounds moves */

struct KindsOfPieces  { *thisKindPtr; /* KindsOfPieces ptr for piece being moved */
  struct PiecesInUse  { *thisPiecePtr; /* PiecesInUse ptr for piece being moved */
    struct Squares  { *thisSqPtr; /* Squares ptr for square moving from */
      Point offsetPoint;  /* original click point within pane - stored
distance from origin in pane coords */
    } firstKeepTracking;  /* first call of KeepTracking */
    Boolean hasMoved;  /* Boolean passed to piece hasMoved if not undone */
  } enPassantPtr; /* save gAGame->enPassantPtr for Undo */
  Boolean goodMove;
  }

Methods **/
/** Instance

Constriction/Destruction **/

void IPieceMove( short aNameIndex, Squares *arraySqPtrParm[NUM_SQUARES_PER_MOVE], PiecesInUse *arrayPiecePtrParm[NUM_SQUARES_PER_MOVE], short indexParm,
                  Point offsetParm );

void Dispose(void);
/**

Accessing **/

void BeginTracking( LongPt *startPt );
void KeepTracking( LongPt *currPt, LongPt *prevPt, LongPt *startF
void EndTracking( LongPt *currPt, LongPt *prevPt, LongPt *startPt
void Undo();

);//

/* used by PieceMove methods */

void CleanupSquares(Point offsetPoint, Point *prevPt, Rect pieceRect );

*******************************************************************************
PieceMove.c
The PieceMove Class
SUPERCLASS = CMouseTask
*******************************************************************************
#include "PieceMove.h"
#include <CMouseTask.h>
#include "KindsOfPieces.h"
#include "Boards.h"
#include "PiecesInUse.h"
#include "KindsOfPieces.h"
#include "Squares.h"
#include "ChessNetDef.h"
#include <stdlib.h>
#include <LongCoordinates.h>

/* need for def. of GetNSTDialogItems - not a member */

extern Network *gAGame; /* global ptr to A Game document */
extern DialogPtr gDialogPtr; /* global dialog ptr */

/**** CONSTRUCTION/DESTRUCTION METHODS ****/

/IPieceMove

Initialize a PieceMove object

*******************************************************************************/

void PieceMove::IPieceMove( short aNameIndex, 
*Squares arrayOrigSqPtrParm[],
*PiecesInUse arrayOrigPiecePtrParm[],
indexParm, 
*offsetPointParm )
{
    IMouseTask( aNameIndex );
    for( i = 0; i < indexParm && i < NUM_SQUARES_PER_MOVE; i++ )
    {
        arrayOrigSqPtrParm[i] = arrayOrigSqPtrParm[i];
        arrayOrigPiecePtrParm[i] = arrayOrigPiecePtrParm[i];
        arrayOrigPieceNum[i] = arrayOrigPieceNum[i];
    }
    /* save index - will start at this value & decrement if you have to Undo move */
    origSqArrayIndex = i - 1;
    /* offsetPoint used so that piece starts off from starting place 
    no matter where first clicked when within piece */
    offsetPoint = offsetPointParm;
    /* save Kind ptr for first piece ptr and square ptr in array 
    this is always the piece & square being moved!
    thisKindPtr = arrayOrigPiecePtr[0]->itsKindPtr;
    thisPiecePtr = arrayOrigPiecePtr[0];
    thisSqPtr = arrayOrigSqPtr[0];
    firstKeepTracking = TRUE;
}

*******************************************************************************/

/**************************** FINISH CONSTRUCTION/DESTRUCTION METHODS ******/

/**************************** FINISH CONSTRUCTION/DESTRUCTION METHODS ******/

/*** ACCESSING METHODS ****/

DISPOSE

Dispose (OVERRIDE)

*******************************************************************************/

void PieceMove::Dispose()
{
    inherited::Dispose();
}

*******************************************************************************/
BeginTracking (OVERRIDE if needed)

BeginTracking a PieceMove (don't need to override if you don't change startPt)

void PieceMove::BeginTracking( LongPt *unusedPt )
{
  Rect area;

  /* erase piece being moved from starting square */
  thisSqPtr->currentPiecePtr = (PiecesInUse *) NULL;
  thisSqPtr->Prepare();

  (void) LongToQDRect( &(thisSqPtr->frame), &area);
  thisSqPtr->Draw( &area );
}

KeepTracking (OVERRIDE)

KeepTracking a PieceMove

void PieceMove::KeepTracking( LongPt *currLongPt, LongPt *prevLongPt, LongPt *startLongPt )
{
  long hLocation;
  long vLocation;
  short i, j;
  short vOffset, hOffset, hEncl, vEncl;
  Rect pieceRect;
  Point pt1, pt2, pt3;
  Point *currPt, *prevPt, *startPt = &pt3;

  /* cPreparedView is static variable - provides pointer to a CView object which is fast to pointer to CPane class so can access FrameToWind method */

  thisSqPtr->FrameToWind( currLongPt, currPt);
  thisSqPtr->FrameToWind( prevLongPt, prevPt);
  thisSqPtr->FrameToWind( startLongPt, startPt );

  /*
  (((CPane*)CView::cPreparedView)->FrameToWind( currLongPt, currPt);
  (((CPane*)CView::cPreparedView)->FrameToWind( prevLongPt, prevPt);
  (((CPane*)CView::cPreparedView)->FrameToWind( startLongPt, startPt ) */

  /* if piece has moved - then draw a new piece */
  if ( (currPt->h != prevPt->h) || (currPt->v != prevPt->v) )
    {
      if (firstKeepTracking)
      {
        /* first time called*/
        (void) LongToQDRect( &(thisSqPtr->frame), &pieceRec
        /*thisSqPtr->GetFrame( &pieceRec );*/
        /*thisSqPtr->FrameToWindR( &(thisSqPtr->frame),
        &pieceRec );*/

        /* update - redraw underlying squares - for previous piece position */
        CleanupSquares( offsetPoint, prevPt, pieceRect );

        if (firstKeepTracking)
        {
          firstKeepTracking = FALSE;
        }
        /* end - if not firstKeepTracking */

        /* draw piece in movement
         *******************************/

        /* COMPENSATE FOR OFFSET BETWEEN CORNER AND CLICK POINT */
        /* offsetPoint holds original offset between click point and corner rect in pane coords. Need to add it here so that piece starts off from where it was when you move it, not with corner of rect at click point */
Frame and Origin is the

hOriginCurr = - currPt->h + offsetPoint.h; // Origin maps between
/*
vOriginCurr = - currPt->v + offsetPoint.v; // Window coords.
/ *
*/
/* top left corner of the window */
/* expressed in Frame coords */

hOriginPrev = - prevPt->h + offsetPoint.h; // Origin maps between
/ *
vOriginPrev = - prevPt->v + offsetPoint.v; // Window coords.
/ *
/* top left corner of the window */
/* expressed in Frame coords */

/* set origin to draw at */
SetOrigin((short) hOriginCurr, (short) vOriginCurr);

/* get piece rect from the mask region for piece */
/* RectRgn( thisKindPtr->maskRegion, &pieceRect ); */
(void) LongToQDRect( &(thisSqPtr->frame), &pieceRect);
/*thisSqPtr->GetFrame( &pieceRect );*/
/*thisSqPtr->FrameToWindR( &(thisSqPtr->frame), &pieceRect );*/

/* draw moving piece */
thisKindPtr->Draw( &pieceRect );

/* end - if piece point has moved - redraw it */
}

EndTracking (OVERRIDE)

EndTracking a PieceMove

PieceMove::EndTracking( LongPt *currLongPt, LongPt *prevLongPt,

LongPt *startLongPt )
{
    Boards *boardPtr;
    Rect pieceRect;
    short fromSq, fromPiece, toSq;
    Point pt1, pt2, pt3;
    Point *currPt = &pt1, *prevPt = &pt2, *startPt = &pt3;

    /* cPreparedView is static variable - provides pointer to a CView
     object which is cast to pointer to CPane class so can
     access FrameToWind method */

    /* set attribute - used in subclass code */
    goodMove = FALSE;

    thisSqPtr->FrameToWind( currLongPt, currPt);
    thisSqPtr->FrameToWind( prevLongPt, prevPt);
    thisSqPtr->FrameToWind( startLongPt, startPt);

    /*
((CPane*)CView::cPreparedView)->FrameToWind( currLongPt, currPt);
((CPane*)CView::cPreparedView)->FrameToWind( prevLongPt, prevPt);
((CPane*)CView::cPreparedView)->FrameToWind( startLongPt, startPt)
*/

    boardPtr = gAGame->firstBoardPtr;

    (void) LongToQDRect( &(thisSqPtr->frame), &pieceRect);
    /*thisSqPtr->GetFrame( &pieceRect );*/
    /*thisSqPtr->FrameToWindR( &(thisSqPtr->frame), &pieceRect );*/

    /* update - redraw underlying squares -
     for previous piece position */

    CleanupSquares(offsetPoint, currPt, pieceRect);

    /* determine move TO square & set current piece for that square */
    for (i = 0; i < NUM_SQUARES && found == FALSE; i++)
    /* check to see if square contains point */
    if (boardPtr->arrayOfSqPtr[i]->Contains( *currPt ) )
/* save To square & its prior piece in case of undo */
/* increment array index - decrement this in Undo */
origSqArrayIndex++;
arrayOrigSquares[origSqArrayIndex] = boardPtr->arrayOfSquares[i];
/* store square to undo move */
arrayOrigPiecePtr[origSqArrayIndex] = boardPtr->arrayOfSquares[i]->currentPiecePtr;
/* store piece to undo move */
/* set current piece for MOVE TO square */
boardPtr->arrayOfSquares[i]->setPiece(thisPiecePtr);
(void) LongToQDRect( &boardPtr->arrayOfSquares[i]->frame, &pieceRect);
/* get frame for piece */
boardPtr->arrayOfSquares[i]->prepareOn();
boardPtr->arrayOfSquares[i]->draw( &pieceRect );
found = TRUE; /* stop! */
}
goodMove = found;

void PieceMove::undo()
{
    Rect pieceRect;

    /* restore & redraw square and pieces to state before last move */
    /* start at last value for origSqArrayIndex - then decrement */
    for( ; origSqArrayIndex >= 0; origSqArrayIndex-- )
    {
        /* reset pieceNumber if needed - for just moved king and rook */
        if( arrayOrigPieceNum[origSqArrayIndex] != (PiecesInUse *) NULL )
        {
            arrayOrigPiecePtr[origSqArrayIndex]->pieceNumber =
            arrayOrigPiecePtr[origSqArrayIndex]->pieceNumber;
        }
    }
    /* reset old piece for each square & draw it */
    arrayOrigPiecePtr[origSqArrayIndex]->currentPiecePtr =
    arrayOrigPiecePtr[origSqArrayIndex];
    arrayOrigSquares[origSqArrayIndex]->prepareOff();
    arrayOrigSquares[origSqArrayIndex]->draw( &pieceRect );
}
/* restore state of gGame->enPassantPtr using PieceMove::enPassantPtr */
gGame->enPassantPtr = this->enPassantPtr;

void CleanupSquares(Point offsetPoint, Point *updatePtr, Rect pieceRect )
{
    Rect updateRect, shortUpdateRect, shortFrameRect;

    /*---------------------------*/
    Undo (OVERRIDE - in CTask)
    /*---------------------------*/
    Undo a previous move for this PieceMove
    */
    non-method function */
/* convert update rect from window to frame (but long) coords however, not actually using long coords so already map to QD */
updateSquares[NUM_SQUARES + 1];
short i, j;
longRect = longFrameRect, longUpdateRect;

updateRect.left = updatePt->x - offsetPoint.x;
updateRect.top = updatePt->y - offsetPoint.y;
updateRect.right = updateRect.left + pieceRect.right; /* frame always at 0,0 */
updateRect.bottom = updateRect.top + pieceRect.bottom;

/* check to see if each square contains any of 4 points of update area */
for (i = 0, j = 1; i < NUM_SQUARES; i++)
    boardPtr->arrayOfSquares[i] = GetFrame( &longFrameRect );

/* convert update rect from window to frame (but long) coords however, not actually using long coords so already map to QD */
boardPtr->arrayOfSquares[i] = WindToFrameRect( &updateRect,
&longUpdateRect);

/* now convert from long to short */
(void) LongToQDRect( &longUpdateRect, &shortUpdateRect);

(void) LongToQDRect( &longFrameRect, &shortFrameRect);
SetRect( &shortFrameRect, &shortUpdateRect, &shortFrameRect );

if (! EmptyRect( &shortFrameRect ) )
    updateSquares[++j] = boardPtr->arrayOfSquares[i];

while (j >= 0) {
    /* updateSquares[j]->Prepare(); */
    /* convert update rect from window to frame (but long) coords however, not actually using long coords so already map to QD */
    updateSquares[j]->WindToFrameRect( &updateRect,
&longUpdateRect);

    /* now convert from long to short */
    (void) LongToQDRect( &longUpdateRect, &shortUpdateRect);

    /* draw only update area - converted from window to frame */
    updateSquares[j]->ForceNextPrepare(); /* in case my setorigi
    messed up */

    preparedViewScheme /* */
    updateSquares[j]->Prepare();
    updateSquares[j --]->Draw( &shortUpdateRect);
}

/******************* Interface for the PieceMove2 Class

***************************************************************************/

/* include <CMouseTask.h> */
#include "Squares.h"
#include "PieceInUse.h"
#include "ChessNetDef.h"
#include "PieceMove.h"

class PieceMove2: public PieceMove { /* Class Declaration */

public:
Variables **/

Methods **/

Construction/Destruction **/

```c
void IPieceMove2( short aNameIndex,
    Squares *arraySqrPtrParm[NUM_SQUARES_PER_MOVE],
    PiecesInUse *arrayPiecePtrParm[NUM_SQUARES_PER_MOVE],
    short indexParm, 
    Point offsetPointParm );

void Dispose(void);
```

Accessing **/

```c
void BeginTracking( LongPt *startPt );

void EndTracking( LongPt *currPt, LongPt *prevPt, LongPt *startPt );
```

/** Methods needed to check for legal moves **/

```c
Boolean IsMoveLegal( short fromSq, short fromPiece, short toSq );

Boolean TryToCastle( short *pseudoMoves, short *counter,
    short fromSq, short fromPiece, short toSq );

Boolean EnPassant( short fromSq, short fromPiece, short toSq );

Boolean IsYourKingInCheck( short *pseudoMoves, short *counter,
    short fromSq, short fromPiece, short toSq );

Boolean IsSquareUnderAttack( short *pseudoMoves, short *counter,
    short fromSq, short fromPiece, short toSq );
```

```c
void GeneratePsuedoMoves( short *pseudoMoves, short *counter,
    short fromSq, short fromPiece );

void GenerateRankNFileMoves( short *pseudoMoves, short *counter,
    short fromSq, short fromPiece );

void GenerateDiagonalMoves( short *pseudoMoves, short *counter,
    short fromSq, short fromPiece );

void GeneratePawnMoves( short *pseudoMoves, short *counter,
    short fromSq, short fromPiece );

void GenerateKnightMoves( short *pseudoMoves, short *counter,
    short fromSq, short fromPiece );

void GenerateKingMoves( short *pseudoMoves, short *counter,
    short fromSq, short fromPiece );

void CheckASquare( short i, short *pseudoMoves, short *counter,
    short fromSq, short fromPiece );

void FillArrayPositions( void );

void PromotePawn( short pieceColor, Squares *toSquarePtr );
```

PieceMove2.c

The `PieceMove2` class

SUPERCLASS = PieceMove
#include "PieceMove2.h"
#include <CMouseTask.h>
#include "KindsOfPieces.h"
#include "Boards.h"
#include "PiecesInUse.h"
#include "KindsOfPieces.h"
#include "Squares.h"
#include "ChessNetDef.h"
#include <stdlib.h>
#include <LongCoordinates.h>
#include "PieceMove.h"

#include "CCheeseNetDoc.h"  /* need for def. of GetNSetDialogItems - not a
member */

extern Network *gAGame;  /* global ptr to A Game document */
extern DialogPtr *gDialogPtr; /* global dialog ptr */
extern DialogPtr gDialogPtr; /* global dialog ptr */

/*** CONSTRUCTION/DESTRUCTION METHODS ***/

PieceMove

Initialize a PieceMove object

*******************************************************************************/

void PieceMove2::PieceMove2( short aNameIndex,
*arraySqPtrParm[ ],
*arrayPiecePtrParm[ ],
indexParm,
offsetPointParm )
{
    iPieceMove( aNameIndex,
arraySqPtrParm,
arrayPiecePtrParm,
indexParm,
offsetPointParm );
}

/*******************************************************************************/

Dispose (OVERRIDE)

Dispose of a PieceMove

*******************************************************************************/

void PieceMove2::Dispose()
{
    inherited::Dispose();
}

/*******************************************************************************/

/*** ACCESSING METHODS ***/

BeginTracking (OVERRIDE if needed)

BeginTracking a PieceMove (don't need to override if you don't
change startPt)

*******************************************************************************/

void PieceMove2::BeginTracking( LongPt *unusedPt )
{
    Rect area;
    /* save piece positions in array to determine legal moves later */
    FillArrayPositions();
    PieceMove::BeginTracking( unusedPt );
}

*******************************************************************************/

EndTracking (OVERRIDE)
EndTracking a PieceMove

void PieceMove2::EndTracking(LongPt *currLongPt, LongPt *prevLongPt,
                             LongPt
                             *startLongPt)
{
    short i, j;
    Boards *boardPtr;
    Rect pieceRect;
    short fromSq, fromPiece, toSq;
    Point p1, pt2, pt3;
    Point *currPt = &pt1, *prevPt = &pt2, *startPt = &pt3;

    /* cPreparedView is static variable - provides pointer to a CView
       object which is cast to pointer to CPane class so can
       access FrameToWind method
    */

    thisSQPtr->FrameToWind(currLongPt, currPt);
    thisSQPtr->FrameToWind(prevLongPt, prevPt);
    thisSQPtr->FrameToWind(startLongPt, startPt);

    PieceMove::EndTracking(currLongPt, prevLongPt, startLongPt);

    if (isgoodMove) {
        Undo();
    } else { /* ending square was found so check for legal move */
        /* arrayPositions filled in BeginTracking */

        if (FALSE == IsMoveLegal(thisQPtr->squareNumber, /* from square */
                                  thisQPtr->pieceNumber, /* from piece */
                                  arrayOrigSQPtr[origSQArrayIndex]->squareNumber))
            /* check for pawn double square move - if found set
             enPassantPtr in
             AGame object - used to allow legal enPassant
             */
            /* but first save current state of enPassant in case of U
capture */
            /* to square

            legal move undo move */
            /* if
             */
            if (abs(thisQPtr->pieceNumber) == 41 &&
                abs(thisQPtr->pieceNumber) == 7)
                thisQPtr->hasMoved = TRUE;

            /* set special values for moved king or rook
             */

            /* rook changes from 4 to 5, king changes from
             */

            switch (thisQPtr->pieceNumber) {
                case (4): thisQPtr->pieceNumber;
                case (4): thisQPtr->pieceNumber;
                break;
                case (7): thisQPtr->pieceNumber;
                break;
                break;
                break;
            }
        else
            /* no ending
             */
    }
}

/* move is legal, so set hasMoved flag -
   meaningful only for king & rook */

*/
this->enPassantPtr = gAGame->enPassantPtr;
/* move is legal and other work done so send across network */
gAGame->SendMove();

if (!abs(thiskKindPtr->pieceNumber) == 1
  /* is pawn */
  &&
  !abs(arrayOrigSqrPtr[origSqArrayIndex]->squareNumber)
  /* double move */
  
  *thisSqrPtr->squareNumber) == 2))
  gAGame->enPassantPtr = thisPiecePtr;
  /* otherwise blank out enPassantPtr - must perform on immediate after double pawn move */
  else
  gAGame->enPassantPtr = (PiecesInUse *) NULL;

/* PAWN PROMOTION */

/* check for pawn */
if (!abs(thisPiecePtr->pieceNumber) == 1) {
  /* array item 1 should be second square involved - */
  switch (arrayOrigSqrPtr[1]->squareNumber)
  {
    case 68; case 78:
    case 88: /* on top squares? */
    PromotePawn(WHITE,
      fromSqr, fromPiece, toSqr);
    break;
    case 61; case 71:
    case 81: /* on bottom squares? */
    PromotePawn(BLACK,
      fromSqr, fromPiece, toSqr);
    break;
    
    /* generate all possible moves for from Piece at fromSqr */
    GeneratePseudoMoves(&pseudoMoves[0], &counter, fromSqr,
      fromPiece);
    /* set end of array marker to 110 - not a valid square */
    pseudoMoves[ counter ] = 110;
    /* check to see if toSqr is in list of possible moves - if it is, */
    legal move - unless the move leaves your king in check */
    for (i = 0; returnValue == FALSE && pseudoMoves[i] != 110 &&
      NUM_POSSIBLE_MOVES; i++) {
      if (toSqr == pseudoMoves[i])
        returnValue = TRUE;
    }
    
    Boolean PieceMove2::IsMoveLegal( short fromSqr, short fromPiece, short toSqr )
    { Boolean returnValue = FALSE;
      short
      i, counter = 0;
      /* check for castle move - test for unmoved king & proper to square */
      if (!abs(fromPiece) == 7 && (toSqr == fromSqr + 29 || toSqr == fromSqr - 21)
        
        returnValue = TryToCastle( &pseudoMoves[0], &counter, 
      fromSqr, fromPiece, toSqr);
    } else { /* other than castle move */
      
      /* generate all possible moves for from Piece at fromSqr */
      GeneratePseudoMoves(&pseudoMoves[0], &counter, fromSqr,
        fromPiece);
      /* set end of array marker to 110 - not a valid square */
      pseudoMoves[ counter ] = 110;
      /* check to see if toSqr is in list of possible moves - if it is, */
      legal move - unless the move leaves your king in check */
      for (i = 0; returnValue == FALSE && pseudoMoves[i] != 110 &&
        NUM_POSSIBLE_MOVES; i++) {
        if (toSqr == pseudoMoves[i])
          returnValue = TRUE;
      }
    }
}
passant move
/* if move above was illegal, check for possibility of pawn en
passant occurs, set
returnValue to TRUE and check for king in check as
follows */
if ( returnValue == FALSE )
   returnValue = EnPassant( fromSq, fromPiece, toSq );
/* if move above was legal, so far, then determine if your own king
has been
left in check, which would then be an illegal move */
if ( returnValue == TRUE )
   returnValue = IsYourKingInCheck( &pseudoMoves[0],
   &counter,
   fromSq, fromPiece, toSq );
/* end other than castle move */
}
return( returnValue );

Boolean PieceMove2::TryToCastle( short *pseudoMoves, short *counter,
   short fromSq, short fromPiece, short toSq )
{
   Boolean returnValue = TRUE;
   short i;
   short *thisPiece, *thisSq, *rookSq, *newRookSq;
   Boards *boardPtr = gAGame->firstBoardPtr;
   PiecesInUse *aPiecePtr;
   Squares *aSquarePtr1, *aSquarePtr2;
   Rect drawRect;
/* remember! already tested for unmoved king & proper to square */
switch ( toSq )
   { case ( 31 ): rookSq = 11; newRookSq = 41;
      break;
   case ( 38 ): rookSq = 18; newRookSq = 48;
      break;
   case ( 71 ): rookSq = 81; newRookSq = 61;
      break;
   case ( 78 ): rookSq = 88; newRookSq = 58;
      break;
   /* check for rook in proper square and never moved */
   /* already tested for unmoved king so don't worry about rook color */
   /* 4 = unmoved rook */
   if( abs( arrayOfPositions[ rookSq + BOARD_OFFSET ] ) == 4 )
      returnValue = FALSE; /* if not rook in corner or alre:
moved */
else
   /* make sure intervening squares empty and
   not castling through check */
   /* left */
   if( toSq == 31 || toSq == 38 )
      for( i = fromSq; i > 20 && returnValue == TRUE; i -= 1 )
         if( arrayOfPositions[ i + BOARD_OFFSET ] ==
         11 i == fromSq)
            /* check squares, starting with king squ:
that you are castling through, up to, not including rook square, to see
empty, except for king square */
            if( arrayOfPositions[ i + BOARD_OFFSET ] ==
            /* check to see if king is:
   check, or intervening squares in check, 'cuz you can't castle
   through check */
   IsSquareUnderAttack( pseudoMoves, counter, i, fromPiece )
      returnValue =
      else
      /* castle right */
      returnValue = FALSE;
   }
   returnValue = FALSE;
   }
   }
   returnValue =
   else
   /* castle right */
for( i = fromSq; i < 80 && returnValue == TRUE; i += 10 )
{
    /* check squares, starting with king square, 
        that you are castling through, 
        up to, not including rook square, to see if 
        empty, except for king square */
    if( arrayOfPositions[ i + BOARD_OFFSET ] == 0 
        && i == fromSq )
        /* check to see if king in 
           through check */
        returnValue = 
            IsSquareUnderAttack( pseudoMoves, counter, i, fromPiece );
    else
        returnValue = FALSE;
}
/* end rook is corner & unmoved */
if( returnValue == TRUE )
{
    for( i = 0; i < NUM_SQUARES ; i++ )
    {
        /* walk through all squares looking for rookSquare to 
           blank out and newrooksquare to put rook in */
        if( boardPtr->arrayOfSqPtr[i]->squareNumber == rookSq )
        {
            /* save this piece ptr. so it can be re-assigned */
            aPiecePtr = boardPtr->arrayOfSqPtr[i]->
                >currentPiecePtr;
            /* save square ptr so it can be drawn later */
            aSquarePtr1 = boardPtr->arrayOfSqPtr[i];
            /* save moved from rook square & piece for later undo */
            origSqArrayIndex++;
            arrayOrigSqPtr[origSqArrayIndex] = aSquareF;
            /* store square to undo move */
            arrayOrigPiecePtr[origSqArrayIndex] = aPiece
            /* store piece to undo move */
            arrayOrigPieceNum[origSqArrayIndex] = 
                aPiecePtr->pieceNumber;
        }
        newRookSq = 
            (PiecesInUse *) NULL;
    } else if( boardPtr->arrayOfSqPtr[i]->squareNumber == 
        outside of loop */
        aSquarePtr2 = boardPtr->arrayOfSqPtr[i];
        /* save moved to rook square & blank piece for 
           later undo */
        origSqArrayIndex++;
        arrayOrigSqPtr[origSqArrayIndex] = aSquareF;
        /* store square to undo move */
        arrayOrigPiecePtr[origSqArrayIndex] =
            (PiecesInUse *) NULL;
        /* store piece to undo move */
    }
}
/* now assign rook piece to new square - in effect moving it */
squarePtr->currentPiecePtr = aPiecePtr;

/* re-draw both new & old rook squares */
squarePtr1->Prepare();
/* prepare, ie. set origin */

(void) LongToQDRect(&aSquarePtr1->frame, &drawRect);

aSquarePtr1->Draw( &drawRect );
/* draw old square */

aSquarePtr2->Prepare();
/* prepare, ie. set origin */

(void) LongToQDRect(&aSquarePtr2->frame, &drawRect);

aSquarePtr2->Draw( &drawRect );
/* draw new square */

/* still need return value so move is not undone later */
return( returnValue );

Boolean PieceMove2::EnPassant( short fromSq, short fromPiece, short toSq )
{
    short leftMove, rightMove, behindSq, oppositePiece;
    returnValue = FALSE;
    leftMove, rightMove = behindSq, oppositePiece,
    short;  
    BoardPtr = gGame->firstBoardPtr;
    KindOfPieces *aKindPtr;
    PiecesInUse *aPiecePtr, *aPiecePtr2;
    Rect drawRect;

    /* was pawn moved? and was there a double pawn move made last move? */
    if ( abs(byMoveFromPiece) == 1 && ( gGame->enPassantPtr != (PiecesInUse *)
        NULL ) )
    {
        /* set values depending on black or white piece */
        if (fromPiece > 0)
            leftMove = -9;  /* white pawn */
        rightMove = 11;
        /* black pawn */
        behindSq = -1;
        oppositePiece = -1;
        }
    else
        leftMove = 9;
        rightMove = -11;
        behindSq = 1;
        oppositePiece = 1;

    /* was 1 square diagonal move made? */
    if ( toSq - fromSq == leftMove || toSq - fromSq == rightMove )
    {
        aPiecePtr = gGame->enPassantPtr;
        aKindPtr = aPiecePtr->itKindPtr;

        /* was the double moved pawn of opposite color? */
        if ( aKindPtr->pieceNumber == oppositePiece )
        {
            /* find the piece immediately behind moved piece */
            for( i = 0; i < NUM_SQUARES; i++ )
            {
                if ( behindSq == ( boardPtr->
                    arrayOfSqrPtr[i]->squareNumber - toSq )
                    &&
                    ( boardPtr->arrayOfSqrPtr[i-1-
                    >currentPiecePtr ==
                    gGame->enPassantPtr )
                {  /* does this square contain en passant ptr pawn? */
                    }
                
                /* then carryout en passant */
                capture */

                /* save square & piece for und */
                /* increment array index - decrement this in Undo worl */
                origSqArrayIndex++;
aPiecePtr2 = boardPtr->arrayOfSqPtr[i]->currentPiecePtr;

arrayOrigSqPtr[origSqArrayIndex] = boardPtr->arrayOfSqPtr[i];

/* store square to undo move */
arrayOrigPiecePtr[origSqArrayIndex] = boardPtr->arrayOfSqPtr[i]->currentPiecePtr;

/* store piece to undo move */
arrayOrigPieceNum[origSqArrayIndex] = aPiecePtr2->pieceNumber;

/* store piece number for undo */

if (has been captured) /* now blank piece ptr out - pawn */
    boardPtr->arrayOfSqPtr[i] = (PiecesInUse *) NULL;

/* now draw the empty square */
boardPtr->arrayOfSqPtr[i] = (void) LongToQDRect(&boardPtr->arrayOfSqPtr[i]->frame, &drawRect);

/* draw new square */

/* set return value */
returnValue = TRUE;

/* prepare, i.e. set origin */
Prepare();

/* find your king square - if king is not piece being moved */
if (abs(fromPiece) == 7 || abs(fromPiece) == 8)
    kingSq = toSq;
/* piece being moved is your king, so don't have to find */
Boolean PieceMove2::IsSquareUnderAttack( short *pseudoMoves, short *counter
short fromSq, short fromPiece)
{    Boolean returnVa = TRUE;    /* not called unless returnVa:
    short i;
    short thisPiece, thisSq;
    Boards *boardPtr = gAGame->firstBoardPtr;
    PiecesInUse *aPiecePtr;
    KindsOfPieces *aKindPtr;
    /* re-use pseudoMoves array so initialize counter variable */
    *counter = 0;
    for (i = 0; i < NUM_SQUARES ; i++)
    {
        /* if square has a piece */
        if (boardPtr->arrayOfSqPtr[i]->currentPiecePtr !=
            (PiecesInUse *) NULL)
        {
            aPiecePtr = boardPtr->arrayOfSqPtr[i]->
            >currentPiecePtr;
            aKindPtr = aPiecePtr->itsKindPtr;
            thisPiece = aKindPtr->pieceNumber;
            if ((fromPiece < 0 & & thisPiece < 0) || /* check
                (fromPiece > 0 & & thisPiece > 0)
                
            )
            {
                /* if this is your king save the square
                number for later use */
                if (abs(thisPiece) == 7 || abs(thisPiece)
                    == 8)
                {
                    kingSq = boardPtr->
                    >arrayOfSqPtr[i]->squareNumber;
                }
            }
        }
    }
    /* else have to find king */
    /* now using king square, check moves of opponent to see if king left in
    check - not legal */
    returnVa = IsSquareUnderAttack( pseudoMoves, counter, kingSq,
    fromPiece);
    return( returnVa );
}
/* check to see if fromSq is in list of possible moves of opposing pieces - if it is, */
then fromSq IS under attack ( if king occupies it - would be in check */
for (i = 0; returnValue == TRUE && pseudoMoves[i] != 110 && i <
NUM_POSSIBLE_MOVES; i++)
{
    if (fromSq == pseudoMoves[i])
        returnValue = FALSE; /* if square under attack return false */
}
return( returnValue );

void PieceMove2::GeneratePsuedoMoves( short *pseudoMoves, short *counter,
short fromSq, short fromPiece )
{
    switch (fromPiece)
    {
    case 1: case -1: GeneratePawnMoves( pseudoMoves, counter,
fromSq, fromPiece );
        break;
    case 2: case -2: GenerateKnightMoves( pseudoMoves, counter,
fromSq, fromPiece );
        break;
    case 3: case -3: GenerateDiagonalMoves( pseudoMoves, counter,
fromSq, fromPiece );
        break;
    case 4: case -4: GenerateRankNFileMoves( pseudoMoves, counter,
fromSq, fromPiece );
        break;
    case 5: case -5: GenerateRankNFileMoves( pseudoMoves, counter,
fromSq, fromPiece );
        break;
    case 6: case -6: GenerateDiagonalMoves( pseudoMoves, counter,
fromSq, fromPiece );
        break;
    } GenerateRankNFileMoves( pseudoMoves, counter, fromSq, fromPiece );
    break;

    case 7: case -7: GenerateKingMoves( pseudoMoves, counter,
fromSq, fromPiece);
        break;
    case 8: case -8: GenerateKingMoves( pseudoMoves, counter,
fromSq, fromPiece);
        break;
    }
}

void PieceMove2::GenerateRankNFileMoves( short *pseudoMoves, short *counter,
short fromSq, short fromPiece )
{
    short i, j, offset;
    for( j=0; j<4; j++)
    {
        switch ( j )
        {
        case 0 : offset = 1;
            break;
        case 1 : offset = -1;
            break;
        case 2 : offset = 10;
            break;
        case 3 : offset = -10;
            break;
        }
        i = fromSq + BOARD_OFFSET;
        do {
            i += offset;
            CheckASquare( i, pseudoMoves, counter, fromPiece);
        } while( arrayOfPositions(i) == 0 );
    }

void PieceMove2::GenerateDiagonalMoves( short *pseudoMoves, short *counter,
short fromSq, short fromPiece )
{
    short i, j, offset;
    for( j=0; j<4; j++)
    {
        switch ( j )
        {
        case 0 : offset = 1;
            break;
        case 1 : offset = 1;
            break;
        case 2 : offset = 10;
            break;
        case 3 : offset = -10;
            break;
        }
        i = fromSq + BOARD_OFFSET;
        do {
            i += offset;
            CheckASquare( i, pseudoMoves, counter, fromPiece);
        } while( arrayOfPositions(i) == 0 );
    }
for( j=0; j<4; j++ )
{
    switch( j )
    {
        case 0 : offset = 9;
            break;
        case 1 : offset = 11;
            break;
        case 2 : offset = -9;
            break;
        case 3 : offset = -11;
            break;
    }
    i = fromSq + BOARD_OFFSET;
    do {
        i += offset;
        CheckASquare( i, pseudoMoves, counter, fromPiece);
        while( arrayOfPositions[i] == 0 );
    } while( i < STARTSQUARES );
}

void PieceMove2::GeneratePawnMoves( short *pseudoMoves, short *counter,
short fromSq, short fromPiece)
{
    Boolean whitePawn = TRUE, onStartSq = FALSE;
    /* set white or black pawn */
    if( fromPiece == -1 )
        whitePawn = FALSE;
    /* determine if pawn on one of starting squares */
    if( whitePawn )
    {
        for( i = 12; i < 33 && onStartSq == FALSE; i += 10 )
            /* check for white pawn on start squares */
            if( fromSq == i )
            {
                onStartSq = TRUE;
            }
    } else
    {
        for( i = 17; i < 88 && onStartSq == FALSE; i += 10 )
            /* check for black pawn on start squares */
            if( fromSq == i )
                onStartSq = TRUE;
    }

    /* white pawn on start square */
    if( whitePawn == TRUE && onStartSq == TRUE )
    {
        /* loop twice to add 1 & 2 square pawn moves if empty */
        for( i = fromSq + BOARD_OFFSET + 1; arrayOfPositions[i] == 0
            &&
            i < (fromSq + BOARD_OFFSET + 3); i++ )
        {
            pseudoMoves[(*counter)++] = i - BOARD_OFFSET;
            /* store empty square */
        }
    }
    /* black pawn on start square */
    else if( whitePawn == TRUE )
    {
        /* loop twice to add 1 & 2 square pawn moves if empty */
        for( i = fromSq + BOARD_OFFSET - 3; arrayOfPositions[i] == 0
            &&
            i > (fromSq + BOARD_OFFSET - 1); i-- )
        {
            pseudoMoves[(*counter)++] = i - BOARD_OFFSET;
            /* store empty square */
        }
    }
    /* white pawn - 1 square move */
    else if( whitePawn == TRUE )
    {
        if( arrayOfPositions[fromSq + BOARD_OFFSET + 1] == 0 )
            pseudoMoves[(*counter)++] = fromSq + 1;
            /* at empty square */
    }
    /* black pawn - 1 square move */
    else
    {
        if( arrayOfPositions[fromSq + BOARD_OFFSET - 1] == 0 )
            pseudoMoves[(*counter)++] = fromSq - 1;
            /* at empty square */
    }
}
/* add diagonal attack squares */
if (whitePawn == TRUE) {
    /* white right attack */
    if (arrayOfPositions[fromSq + BOARD_OFFSET + 11] < 0 &&
        arrayOfPositions[fromSq + BOARD_OFFSET + 11] != -9 )
        /* check for border square */
        pseudoMoves[(*counter)++] = fromSq + 11; /* store captured square */

    /* white left attack */
    if (arrayOfPositions[fromSq + BOARD_OFFSET - 9] < 0 &&
        arrayOfPositions[fromSq + BOARD_OFFSET - 9] != -9 )
        /* check for border square */
        pseudoMoves[(*counter)++] = fromSq - 9; /* store captured square */
} else {
    /* black right attack */
    if (arrayOfPositions[fromSq + BOARD_OFFSET - 11] > 0 )
        /* pos. so no border squares */
        pseudoMoves[(*counter)++] = fromSq - 11; /* store captured square */

    /* black left attack */
    if (arrayOfPositions[fromSq + BOARD_OFFSET + 9] > 0 )
        /* pos. so no border squares */
        pseudoMoves[(*counter)++] = fromSq + 9; /* store captured square */
}

void PieceMove2::TranslateMove(short *pseudoMoves, short *counter, short fromSq, short fromPiece) {
    short i, j, offset;
    for (j=0; j<8; j++) {
        switch (j) {
        case 0 : offset = 8; break;
        case 1 : offset = 19; break;
        case 2 : offset = 12; break;
        case 3 : offset = 21; break;
        case 4 : offset = -19; break;
        case 5 : offset = -8; break;
        case 6 : offset = -12; break;
        case 7 : offset = -21; break;
        }
        i = fromSq + BOARD_OFFSET + offset;
        CheckASquare(i, pseudoMoves, counter, fromPiece);
    }
}

void PieceMove2::GenerateKingMoves(short *pseudoMoves, short *counter, short fromSq, short fromPiece) {
    short i, j, offset;
    for (j=0; j<8; j++) {
        switch (j) {
        case 0 : offset = -11; break;
        case 1 : offset = -10; break;
        case 2 : offset = -9; break;
        case 3 : offset = -1; break;
        case 4 : offset = 1; break;
        }
        i = fromSq + BOARD_OFFSET + offset;
        CheckASquare(i, pseudoMoves, counter, fromPiece);
    }
}

void PieceMove2::GenerateKnightMoves(short *pseudoMoves, short *counter, short fromSq, short fromPiece) {
    short i, j, offset;
    for (j=0; j<8; j++)
        { /* knight jumps */
            switch (j) {
            case 0 : offset = -11; break;
            case 1 : offset = 19; break;
            case 2 : offset = 12; break;
            case 3 : offset = 1; break;
            case 4 : offset = 8; break;
            case 5 : offset = 12; break;
            case 6 : offset = -11; break;
            case 7 : offset = -1; break;
            }
            i = fromSq + BOARD_OFFSET + offset;
            CheckASquare(i, pseudoMoves, counter, fromPiece);
        }
}
case 5 : offset = 9;  
break;

case 6 : offset = 10;  
break;

case 7 : offset = 11;  
break;

i = fromSq + BOARD_OFFSET + offset;  
CheckASquare( i , pseudoMoves, counter, fromPiece);
}

void PieceMove2::CheckASquare( short i, short *pseudoMoves, short *counter, 
short fromPiece)
{
    if( arrayOfPositions[i] == -9)  
        /* out of bounds value - do nothing */
    else if( arrayOfPositions[i] == 0)  
        pseudoMoves[(*counter)++] = i - BOARD_OFFSET;  
        /* store empty square */
    else if((fromPiece > 0 && arrayOfPositions[i] < 0) ||  
            (fromPiece < 0 && arrayOfPositions[i] > 0))  
        pseudoMoves[(*counter)++] = i - BOARD_OFFSET;  
        /* store square w/ captured piece 
        otherwise square has your own piece so 
        do nothing
        */

    

void PieceMove2::FillArrayPositions( void )
{
    short i,j,k, m;
    PiecesInUse *aPiecePtr;

    /************* first fill in arrayOfPositions *************/

    for( i = 0; i < 20; i++)  
        /* se left out of bounds squares */
        arrayOfPositions[ i ] = -9;

    m = 0;
    /* counter for array of sq. ptra */

    for( j= 0; j < 8; j++)  
        /* set bottom out of bounds square */
        arrayOfPositions[ j ] = -9;

    for( k = 0; k < 8; k++)  
        /* set actual squares and pieces */
        aPiecePtr = gAGame->firstBoardPtr->arrayOfSqPtr[m++]->currentPiecePtr;

    if( aPiecePtr == (PiecesInUse *) NULL )  
        arrayOfPositions[i++] = 0;
    else {
        arrayOfPositions[i++] = aPiecePtr->pieceNum;

        arrayOfPositions[i++] = -9;
        /* set top out of bounds square */
    }

    for( i = 100; i < 120; i++)  
        /* se right out of bounds squares */
        arrayOfPositions[ i ] = -9;

}
void PieceMove2::PromotePawn( short pieceColor, Squares *toSquarePtr )
{
    int dialogDone = FALSE;
    int itemHit = QUEEN_RADIO;
    int pieceSelection;
    short kindPieceNum;
    KindsOfPieces *aKindPtr;
    PiecesInUse *newPiecePtr, *storePiecePtr;
    Rect drawRect;

    enum piece_numbers
    {
        WHITE_PAWN = 1,
        BLACK_PAWN,
        WHITE_KING,
        BLACK_KING,
        WHITE_HORSE,
        BLACK_HORSE,
        WHITE_BISHOP,
        BLACK_BISHOP,
        WHITE_CASTLE,
        BLACK_CASTLE,
        WHITE_QUEEN,
        BLACK_QUEEN,
        -6
    };

    /* GetNewDialog(short dialogID,void *dStorage,WindowPtr behind)*/
    /* default selected item to queen */
    GetNSetDialogItems( QUEEN_RADIO, 5 );

    /* show dialog */
    ShowWindow( gDialogPtr );

    while ( dialogDone == FALSE )
    {
        /* save previous selection 'cuz at end will always point to save
                         button */
        pieceSelection = itemHit;

        /* ModalDialog(NIL_POINTER, &itemHit); */
        ModalDialog( ModalFilterProcPtr ) NULL, &itemHit );

        switch ( itemHit )
        {
            case QUEEN_RADIO:
                GetNSetDialogItems( QUEEN_RADIO, 5 );
                break;
            case ROOK_RADIO:
                GetNSetDialogItems( ROOK_RADIO, 5 );
                break;
            case BISHOP_RADIO:
                GetNSetDialogItems( BISHOP_RADIO, 5 );
                break;
            case KNIGHT_RADIO:
                GetNSetDialogItems( KNIGHT_RADIO, 5 );
                break;
            case SAVE_BUTTON:
                HideWindow( gDialogPtr );
                dialogDone = TRUE;
                break;
        }
    }
}
switch (pieceSelection) {
    case QUEEN_RADIO:  
        if (pieceColor == WHITE)
            kindPieceNum = WHITE_QUEEN;
        else
            kindPieceNum = BLACK_QUEEN;
        break;
    case ROOK_RADIO:  
        if (pieceColor == WHITE)
            kindPieceNum = WHITE_CASTLE;
        else
            kindPieceNum = BLACK_CASTLE;
        break;
    case BISHOP_RADIO:  
        if (pieceColor == WHITE)
            kindPieceNum = WHITE_BISHOP;
        else
            kindPieceNum = BLACK_BISHOP;
        break;
    case KNIGHT_RADIO:  
        if (pieceColor == WHITE)
            kindPieceNum = WHITE_HORSE;
        else
            kindPieceNum = BLACK_HORSE;
        break;
}

/* done with dialog - got selection - now change pawn to new piece */

/* find correct kind ptr for kind of piece you want to add */
aKindPtr = gAGame->GetKindsOfPieces (kindPieceNum);

/* create a new PiecesInUse */
storePiecePtr = gAGame->firstPiecePtr;
newPiecePtr = new (PiecesInUse);
newPiecePtr->PiecesInUse( 45, 45, /* width */ 0, /* height */
    /* aHEncl */ 0, /* aVEncl */ aKindPtr,
    /* CBitmap */ aBitmap /* */
    );

PieceInUse.h

Interface for the PiecesInUse Class

persona
#include <CObject.h>
#include <CBitmap.h>

class PiecesInUse : public CObject { /* Class Declaration */
public:

Variables **/

char short /* color of piece */
struct PiecesInUse piece in linked list */
Rect struct KindsOfPieces *itsKindPtr;
Boolean short number representation of kind of piece

1 - pawn
2 - horse
3 - bishop
4 - rook that has not moved
5 - rook that has moved
6 - queen
7 - king that has not moved
8 - king that has moved

negatives indicate black pieces Negatives indicate black pieces

Methods **/

Constraction/Destruction **/
void IPiecesInUse( short aWidth,

KindsOfPieces *aKindPtrParm,
colorParm,
*nextPtrParm);
void Dispose(void);

Accessing **/
void Draw( Rect *area ); /* parm not i

; /*permission list*********************************************************/

PiecesInUse.c

The PiecesInUse Class

SUPERCLASS = CBitMapPane

*****************************************************************************

#include "PiecesInUse.h"
#include "KindsOfPieces.h"

/**** CONSTRUCTION/DESTRUCTION METHODS ****/

*****************************************************************************

/IPiecesInUse

Initialize a PiecesInUse object

*****************************************************************************
void PiecesInUse::PiecessInUse(
    short aWidth,
    short aHeight,
    short aHEnc,
    short aVEnc,
    KindsOfPieces *aKindPtrParm,
    short colorParm,
    PiecesInUse *nextPtrParm)
{
    color = colorParm;
    nextPiecePtr = nextPtrParm;
    itsKindPtr = aKindPtrParm;

    this->pieceNumber = itsKindPtr->pieceNumber;

    frame.left = aHEnc;
    frame.top = aVEnc;
    frame.right = frame.left + aWidth;
    frame.bottom = frame.top + aHeight;

    hasMoved = FALSE;
}

/**********************************************
Dispose (OVERRIDE)
Dispose of a PiecesInUse
***********************************************/

void PiecesInUse::Dispose()
{
    delete( this );
}

/**********************************************
****ACCESSING METHODS****/

void PiecesInUse::Draw( Rect *area )
{
    itsKindPtr->Draw( area );
}

****
* Squares.h
*
* Squares **class for all squares on board
*
****/

#pragma once
/* Include this file only once */
#include <CPane.h>
#include "PiecesInUse.h"
#include "KindsOfSquares.h"

class Squares : public CPane {

public:

    char squareName[8];
    short color;
    /*
    BLACK/WHITE */
    PiecesInUse *currentPiecePtr; /* current piece in squ

    char boardName[8];
    KindsOfSquares *itsKindPtr;
    short squareNumber;

    /*

Construction/Destruction ***/

void ISquares( CView *anEnclosure,
              *aSupervisor,
              aWidth,
              aHeight,
short
short
SizingOption
SizingOption
short
short
KindsOfSquares
char

*SKindPtrParm,
*boardNameParm

void DisPose( void );
void Draw( Rect *area );
void SetPiece( PieceInUse *piecePtrParm );

void DoClick( Point hitPt, short modifierKeys, long when );

void Squares::ISquares( CView *anEnclosure, CBureaucrat *aSupervisor, short *
short short
short short short short
short short short short
char

*SKindPtrParm,
*boardNameParm

{ (void) CPane::IPane( anEnclosure, aSupervisor, aWidth, 
```cpp
void Square::SetPiece(PiecesInUse *pieceParm)
{
    this->currentPiecePtr = pieceParm;
}

void Square::DoClick()
{
    overrides DoClick in CView
}

void Square::Draw(Rect *area)
{ /* overridden */
    /* draw square itself */
    itaKindPtr->Draw(area);
    /* draw piece if square contains a piece */
    if (currentPiecePtr != NULL) {
        currentPiecePtr->Draw(area);
    }
}

/**
 * Dispose
 *
 * This is your destruction method.
 * If you allocated memory in your initialization method
 * or opened temporary files, this is the place to release them.
 * 
 * Be sure to call the default method!
 */

void Square::Dispose()
{
    inherited::Dispose();
}

/**
 * Draw
 *
 * Draw this square
 */
```
thisKindsOfPieces = currentPiecePtr->itsKindPtr;

if (currentPiecePtr == NULL ||
    (!PtInRgn(hitPt, thisKindsOfPieces->maskRegion)))
{
    /* if square has no current piece or point is not in mask region */
    /* ----- DO NOTHING! ----- ignore mouse click */
}
else {
    //**********try to move piece**********/

    /* initialize arrays for square and pieces being moved */
    for (i = 0; i < NUM_SQUARES_PER_MOVE; i++) {
        arraySqPtrParm[i] = NULL;
        arrayPiecePtrParm[i] = NULL;
    }

    /* create new PieceMove object */
    pieceMovePtr = new PieceMove2;

    /* pass square(s) & piece(s) being moved to new PieceMove */
    /* for MOVING FROM ONE SQUARE ONLY */
    arraySqPtrParm[0] = this;
    arrayPiecePtrParm[0] = currentPiecePtr;
    pieceMovePtr->iPieceMove2(UNDO_MOVE_STR,
        arraySqPtrParm,
        arrayPiecePtrParm,
        /* array index */
        1,
        /* hitPt */
        hitPt);

    /* send Track Mouse to window - it will send BeginTracking
       KeepTracking
       EndTracking
       to PieceMove (a CMouseTask) */
}

    /* get interior size of window */
    (void) Prepare(); /* just added this */
    gAGame->itsWindow->GetInterior(&pInRect);
    (void) EnclToFrameR(&pInRect);
    /*gAGame->itsWindow->TrackMouse( pieceMovePtr, hitPt, &pInRect );*/
    (void) QDToFrame( hitPt, &longHitPt);
    TrackMouse( pieceMovePtr, &longHitPt, &pInRect );

    /* tell supervisor of itsWindow that PieceMove task is complete */
    gAGame->Notify( pieceMovePtr );

    /* end if square has current piece and point is in mask region accept m-
      click*/
}
/* end DoClick */

******************************************************************************

Switchboard2.h

added to override DoHighLevelEvent

******************************************************************************

#pragma once /* Include this file only once */
#include "CSwitchboard.h"

class Switchboard2 : public CSwitchboard
{
public:
    /* Class Declaration */

    /* Instance Variables **/

    /* Instance Methods **/
    virtual void ISwitchboard2(void);
    /* Initialization */
void Switchboard2::DoHighLevelEvent (const EventRecord *theEvent)
{
    OSPerr err;
    char *number = "0xGAME";
    char *extra;
    long result;
    TargetID *myTarget;
    unsigned long *myRefCon;
    Handler dataHandle;
    unsigned long dataLength = sizeof (NetworkData);
    dataHandle = NewHandle( sizeof (NetworkData) );

    /* convert string to long int - store in result - use for comparison */
    result = strtol(number, &extra, 0);

    if (theEvent->message == result) {
        (void) AcceptHighLevelEvent(myTarget,
                                   myRefCon,
                                   &dataHandle);
    } else {

    }
}

Switchboard2::DoHighLevelEvent

Handle a high level events. The TCL assumes all high level events are AppleEvents. If you use high level events that don't follow the Apple Event Interprocess Messaging Protocol you must override DoHighLevelEvent and handle those events yourself. Be sure to call inherited::DoHighLevelEvent so standard AppleEvents get handled.

Switchboard2.c

used to implement receiving high level event from opponent

#include "Switchboard2.h"
#include "Global.h"
#include "TBUtilities.h"
#include "CApplication.h"
#include "CDesktop.h"
#include "CBartender.h"
#include "CWindow.h"
#include "CSwitchboard.h"
#include "Commands.h"
#include "Constants.h"
#include "CAppearance.h"
#include "AppleEvents.h"
#include <stdio.h>
#include "CChessNetDoc.h" /* includes typedef short NetworkData [NUM_SQUARES-1]; */
extern Network *gAGame; /* global ptr to A Game document */

Iinitialize a Switchboard object.
Switchboard::DoHighLevelEvent(theEvent);
}

DisposeHandle( dataHandle );

****
/* ChessNet.c
 */
/* A ChessNet main file for writing programs with the
 * THINK Class Library
 */
/* Copyright © 1990 Symantec Corporation. All rights reserved.
 */
****/
#pragma once
#include "ChessNetApp.h"
#include "ChessNetDoc.h"
#include "ChessNetGlobalDecl.h"

extern QApplication *gApplication;

int main()
{
    gApplication = new(ChessNetApp);
    (ChessNetApp *)&gApplication->mApp
    gApplication->exec();
}

/* Header for declarations used throughout ChessNet application
 */
***/
#pragma once /* Include this file only once */
#define WHITE 0
#define BLACK 1
#define NUM_SQUARES 64

#define TRUE 1
#define FALSE 0
#define UNDO_MOVE_STR 1
#define NUM_SQUARES_PER_MOVE 4
#define NUM_POSSIBLE_MOVES 100
#define BOARD_OFFSET 10

/* defines for Pawn Promotion dialog */
#define PAWN_DIALOG 400
#define NIL_POINTER 0L
#define QUEEN_RADIO 1
#define ROOK_RADIO 2
#define BISHOP_RADIO 3
#define KNIGHT_RADIO 4
#define SAVE_BUTTON 5
#define MOVE_TO_FRONT -1L
#define ON 1
#define OFF 0

****
/* ChessNetGlobalDecl.h
 */
/* Global variables DECLared
 */
***/
#pragma once /* Include this file only once */
#include "ChessNetDoc.h" /* global ptr to A Game document */
#include "ChessNetDoc.h" /* global ptr to A Game document */
Network gAGame; /* global ptr to A Game document */
DialogPtr gDialogPtr; /* global dialog ptr */