

**Username:** Pralay Patoria **Book:** Professional C++, Second Edition. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC 107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

## DESIGNING A CHESS PROGRAM

This section introduces a systematic approach to designing a C++ program in the context of a simple chess game application. In order to provide a complete example, some of the steps refer to concepts covered in later chapters. You should read this example now, in order to obtain an overview of the design process, but you might also consider rereading it after you have finished later chapters.

### Requirements


Before embarking on the design, it is important to possess clear requirements for the program's functionality and efficiency. Ideally, these requirements would be documented in the form of a requirements specification. The requirements for the chess program would contain the following types of specifications, although in more detail and number:

- The program will support the standard rules of chess.
- The program will support two human players. The program will not provide an artificially intelligent computer player.
- The program will provide a text-based interface:
  - The program will render the game board and pieces in plain text.
  - Players will express their moves by entering numbers representing locations on the chessboard.

The requirements ensure that you design your program so that it performs as its users expect.

### Design Steps

You should take a systematic approach to designing your program, working from the general to the specific. The following steps do not always apply to all programs, but they provide a general guideline. Your design should include diagrams and tables as appropriate. This example includes sample diagrams and tables. Feel free to follow the format used here or to invent your own.



*There is no "right" way to draw software design diagrams as long as they are clear and meaningful to yourself and your colleagues.*

### Divide the Program into Subsystems

Your first step is to divide your program into its general functional subsystems and to specify the interfaces and interactions between the subsystems. At this point, you should not worry about specifics of data structures and algorithms, or even classes. You are trying only to obtain a general feel for the various parts of the program and their interactions. You can list the subsystems in a table that expresses the high-level behaviors or functionality of the subsystem, the interfaces exported from the subsystem to other subsystems, and the interfaces consumed, or used, by this subsystem on other subsystems. The best and recommended design for this chess game is to have a clear separation between storing the data and displaying the data by using the *Model-View-Controller* (MVC) paradigm discussed in Chapter 28. That way, you can easily switch between having a text-based interface and a graphical user interface. A table for the chess game subsystems could look like this:

SUBSYSTEM NAME	INSTANCES	FUNCTIONALITY	INTERFACES EXPORTED	INTERFACES CONSUMED
GamePlay	1	Starts game Controls game flow Controls drawing Declares winner Ends game	Game Over	Take Turn (on Player) Draw (on ChessBoardView)
ChessBoard	1	Stores chess pieces Checks for ties and checkmates	Get Piece At Set Piece At	Game Over (on Gameplay)
ChessBoardView	1	Draws the associated ChessBoard	Draw	Draw (on ChessPieceView)
ChessPiece	32	Moves itself Checks for legal moves	Move Check Move	Get Piece At (on ChessBoard) Set Piece At (on ChessBoard)
ChessPieceView	32	Draws the associated ChessPiece	Draw	None
Player	2	Interacts with user: prompts user for move, obtains user's move Moves pieces	Take Turn	Get Piece At (on ChessBoard) Move (on ChessPiece) Check Move (on ChessPiece)
ErrorLogger	1	Writes error messages to log file	Log Error	None

As this table shows, the functional subsystems of this chess game include a GamePlay subsystem, a ChessBoard and ChessBoardView, 32 ChessPieces and ChessPieceViews, two Players, and one ErrorLogger. However, that is not the only reasonable approach. In software design, as in programming itself, there are often many different ways to accomplish the same goal. Not all ways are equal: Some are certainly better than others. However, there are often several equally valid methods.

A good division into subsystems separates the program into its basic functional parts. For example, a Player is a subsystem distinct from the ChessBoard, ChessPieces, or Gameplay. It wouldn't make sense to lump the players into the Gameplay subsystem because they are logically separate subsystems. Other choices might not be as obvious.

In this MVC design, the ChessBoard and ChessPiece subsystems are part of the Model. The ChessBoardView and ChessPieceView are part of the View and the Player is part of the Controller.

Because it is often difficult to visualize subsystem relationships from tables, it is usually helpful to show the subsystems of a program in a diagram where arrows represent calls from one subsystem to another.

### Choose Threading Models

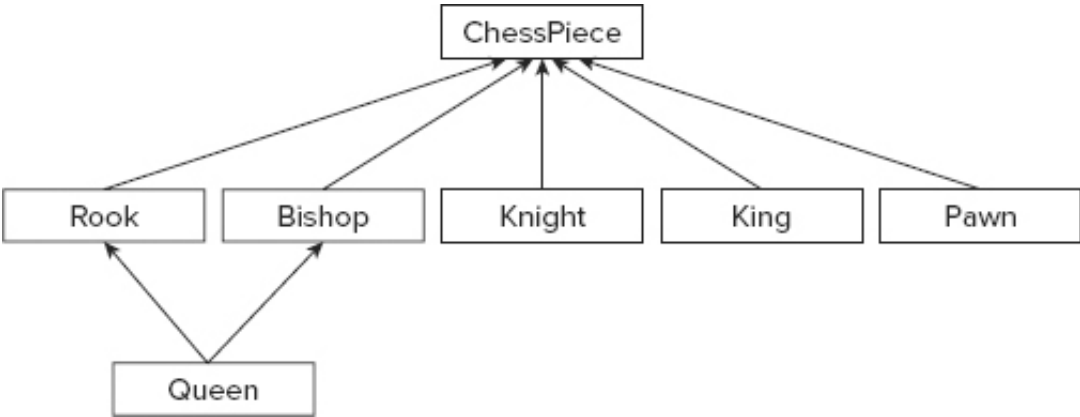
In this step, you choose the number of threads in your program and specify their interactions. In multithreaded designs, you should try to avoid shared data as much as possible because it will make your designs simpler and safer. If you cannot avoid shared data, you should specify locking requirements. If you are unfamiliar with multithreaded programs, or your platform does not support multithreading, then you should make your programs single-threaded. However, if your program has several distinct tasks, each of which should work in parallel, it might be a good candidate for multiple threads. For example, graphical user interface applications often have one thread performing the main application work and another thread waiting for the user to press buttons or select menu items. Multithreaded programming is covered in Chapter 22.

The chess program needs only one thread to control the game flow.

### Specify Class Hierarchies for Each Subsystem

In this step, you determine the class hierarchies that you intend to write in your program. The chess program needs a class hierarchy, to represent the chess pieces. The hierarchy could work as shown in [Figure 2-1](#).

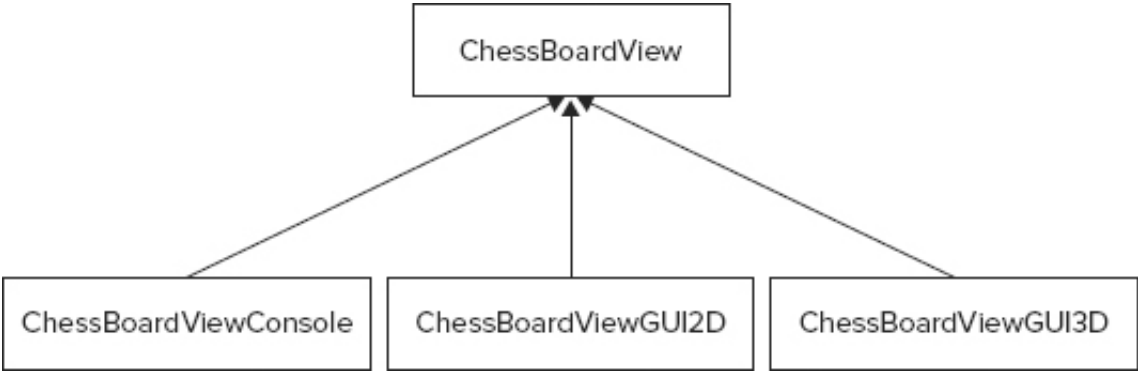
**FIGURE 2-1**



In this hierarchy, a generic `ChessPiece` class serves as the abstract superclass. The hierarchy uses multiple inheritance to show that the queen piece is a combination of a rook and a bishop. A similar hierarchy is required for the `ChessPieceView` class.

Another class hierarchy can be used for the `ChessBoardView` class to make it possible to have a text-based interface or a graphical user interface for the game. [Figure 2-2](#) shows an example hierarchy that allows the chess board to be displayed as text on a console or as a 2D or 3D rendering.

FIGURE 2-2



A similar hierarchy is required for the `Player` controller and for the individual classes of the `ChessPieceView` hierarchy.

Chapter 3 explains the details of designing classes and class hierarchies.

**Specify Classes, Data Structures, Algorithms, and Patterns for Each Subsystem**

In this step, you consider a greater level of detail, and specify the particulars of each subsystem, including the specific classes that you write for each subsystem. It may well turn out that you model each subsystem itself as a class. This information can again be summarized in a table:

SUBSYSTEM	CLASSES	DATA STRUCTURES	ALGORITHMS	PATTERNS
GamePlay	GamePlay class	GamePlay object includes one ChessBoard object and two Player objects.	Gives each player a turn to play	None
ChessBoard	ChessBoard class	ChessBoard object stores a two-dimensional representation of 32 ChessPieces.	Checks for win or tie after each move	None

ChessBoardView	ChessBoardView abstract superclass Concrete subclasses ChessBoardViewConsole, ChessBoardViewGUI2D ...	Stores information on how to draw a chess board	Draws a chess board	Observer
ChessPiece	ChessPiece abstract superclass Rook, Bishop, Knight, King, Pawn, and Queen subclasses	Each piece stores its location on the chess board.	Piece checks for legal move by querying chess board for pieces at various locations.	None
ChessPieceView	ChessPieceView abstract superclass Subclasses RookView, BishopView ... and concrete subclasses RookViewConsole, RookViewGUI2D ...	Stores information on how to draw a chess piece	Draws a chess piece	Observer
Player	Player abstract superclass Concrete subclasses PlayerConsole, PlayerGUI2D, PlayerGUI3D ...	Two player objects (black and white)	Take Turn algorithm: Prompt user for move, check if move is legal, and move piece.	Mediator
ErrorLogger	One ErrorLogger class	A queue of messages to log	Buffers messages and writes them to a log file	Singleton pattern to ensure only one ErrorLogger object

This section of the design document would normally present the actual interfaces for each class, but this example will forgo that level of detail.

Designing classes and choosing data structures, algorithms, and patterns can be tricky. You should always keep in mind the rules of abstraction and reuse discussed earlier in this chapter. For abstraction, the key is to consider the interface and the implementation separately. First, specify the interface from the perspective of the user. Decide *what* you want the component to do. Then decide *how* the component will do it by choosing data structures and algorithms. For reuse, familiarize yourself with standard data structures, algorithms, and patterns. Also, make sure you are aware of the standard library code in C++, as well as any proprietary code available in your workplace.

### Specify Error Handling for Each Subsystem

In this design step, you delineate the error handling in each subsystem. The error handling should include both system errors, such as memory allocation failures, and user errors, such as invalid entries. You should specify whether each subsystem uses exceptions. You can again summarize this information in a table:

SUBSYSTEM	HANDLING SYSTEM ERRORS	HANDLING USER ERRORS
GamePlay	Logs an error with the ErrorLogger, shows a message to the user and gracefully shuts down the program if unable to allocate memory for ChessBoard or Players	Not applicable (no direct user interface)
ChessBoard	Logs an error with the ErrorLogger and throws an exception if unable to allocate memory	Not applicable (no direct user interface)
ChessPieceView	Logs an error with the ErrorLogger and throws an exception if something goes wrong during rendering	Not applicable (no direct user interface)
Player	Logs an error with the ErrorLogger and throws an exception if unable to allocate memory	Sanity-checks user move entry to ensure that it is not off the board; prompts user for another entry. Checks each move legality before moving the piece; if illegal, prompts user for another move.
ErrorLogger	Attempts to log an error, informs user, and gracefully shuts down the program if unable to allocate memory	Not applicable (no direct user interface)

The general rule for error handling is to handle everything. Think hard about all possible error conditions. If you forget one possibility, it will show up as a bug in your program! Don't treat anything as an "unexpected" error. Expect all possibilities: memory allocation failures, invalid user entries, disk failures, and network failures, to name a few. However, as the table for the chess game shows, you should handle user errors differently from internal errors. For example, a user entering an invalid move should not cause your chess program to terminate.

Chapter 10 discusses error handling in more depth.