1 A Challenge – Thinking Parallel

This chapter introduces the subject by

- defining concurrency and parallelism
- discussing a motivating example based on a multi-player on-line game
- introducing the main concepts of processes, channels, alternatives and timers

Designing and building concurrent and parallel systems is easy, provided the appropriate language structures and methods are used. In fact it can be argued that designing concurrent systems is easier than designing sequential systems because they more closely follow the initial thought processes when a problem is posed. However, there is a widely held opinion in the wider computing community that concurrent programming is hard and should be avoided at all costs. This is undoubtedly true if you use any of the existing thread models to build concurrent systems. That is like using assembler to create a complex user interface accessing a database system. We need a higher level abstraction that allows the programmer to think concurrently and avoid the nitty-gritty of thread based programming. It is very hard to convert ideas into artefacts if you do not have the appropriate language capability. For too long programmers have been trying to design and build concurrent and parallel systems with inappropriate and dangerous language structures. The aim of this book is to demonstrate that a few simple language structures and one design pattern are all that is required to build parallel systems that are correct and about which we can reason. The language structures and design pattern are supported by a wealth of formal verification that allows the engineer to have confidence in their designs, without having to fully understand all the formal underpinning.

1.1 Concurrency and Parallelism

Concurrent means a system built from a set of components that execute on a single processor. The components interact with each other in some managed and controlled way to ensure that these interactions do not have unwanted side effects. These components or processes are said to be interleaved on the processor because only one process can execute at one time and the available processing resource is shared among the processes. At any one time more than one of the processes could be executed and the system designer should not make any assumptions as to which process will be allocated the processor. Parallel means that more than one processor is used to execute the processes and these communicate over some form of communication infrastructure. This could be a multi-core processor using memory or a distributed system running a TCP/IP network Within a parallel system it is likely that some of the processors will run some processes concurrently.

1.2 Why Parallel?

There are two reasons for wanting to think parallel. The obvious one enables the programmer to take advantage of modern multi-core processors and networks of workstations. This is generally aimed at obtaining better performance in solving a problem.

The less obvious reason is that, surprisingly, in many cases, it is easier to design a concurrent solution to a problem. The design process seems more natural as it involves just processing elements and the flow of data between these processing elements. The design process is essentially one of creating data flow diagrams. We shall discover that there is just one fundamental parallel design pattern used in the creation of these data flow diagrams or process network diagrams. A further advantage of this design process is that all the processing of data is contained within a processing element and thus the behaviour of it can be observed simply by looking at the definition of the processing element itself. Once the behaviour of a processing element is known it then becomes much easier to connect many of these together with a compositional style of network construction because data processing only takes place within a processing element. This then has the concomitant benefit on the design of data structures, because they become much simpler. They can be defined in a single class, with the required access and manipulation methods and treated as an abstract data type. There is no need to construct data objects that are 'thread safe' because any instance of a data object can only be accessed by a single processing element at any one time.

A further advantage of the approach is that the definition of the processing elements is independent of whether it will be executed concurrently, in a single core of a multi-core processor or in a distributed system. The way in which a processing element is invoked will change but not the definition.

This book focusses on the design of concurrent and parallel systems rather than the performance of the resulting solutions. Without good design and the ability to think parallel it is difficult to exploit the performance advantage that multiple processing elements give.

1.3 A Multi-player Game Scenario

The Pairs Game is a simple game played by families to improve children's spatial awareness and memory. A pack of cards is spread out face down in a regular pattern. A player turns over one card, and then a second. If the cards match in number and colour then the player retains that pair of cards. If the cards do not match then the player replaces the cards face down and another player can choose a pair of cards. Obviously as more cards are turned over the players 'learn' where pairs are located so they can increase the number of pairs they retain. The winner is the player with the most pairs.

For this implementation we shall assume that each player has their own workstation, connected to a network, through which they can interact with the game. In addition there is another workstation that has the ability to create new games and also to manage the interactions between the players. One aspect of the children's version is that it also teaches them to take turns and the patience to wait without giving away the location of a pair. An initial representation of the game architecture is given in Figure 1-1, with three players.

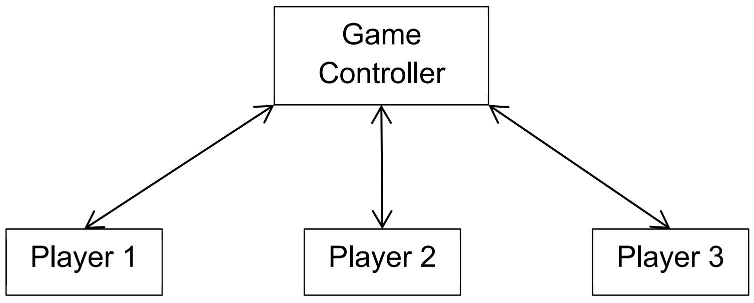


Figure 1-1 Initial Game Architecture

We assume that messages, or data, are going to be passed between each of the Players and the Game Controller but there is no direct interaction between the players. The design process then becomes one of specifying the data and messages that are passed between the processing elements, after which they can be specified and implemented. Each processing element can be tested in isolation once we known the data and messages it is to receive, send and manipulate.

1.3.1 Game Design

The fundamental design underpinning the architecture can be summarised as follows:

Each Player has to enrol with the Game Controller, at which point the Game Controller will send the Player the state of the current game, in which they can now participate.

In this version Players can only see the cards they have turned over. They cannot observe the cards turned over by other Players. That challenge is left to the final chapter of the book.

A Player is presented with a graphical representation of the game showing the cards that have yet to be claimed and also the names of the other Players and the number of pairs they have claimed since joining the game.

A Player chooses two cards by pressing the mouse over each of the two cards they wants to reveal. Players can reveal cards in their own time independently of the other players.

If the cards match, the Player implicitly makes a Claim for that pair, otherwise an interface button is enabled to allow the player to replace the cards and then select another pair of cards.

The Claim may not be successful even if it were a pair because another Player may have claimed that pair previously. After each Claim, successful or not, the state of the game the player can see is updated to the current state. The game state includes both the cards that are yet to be matched and also the number of pairs each player has successfully claimed.

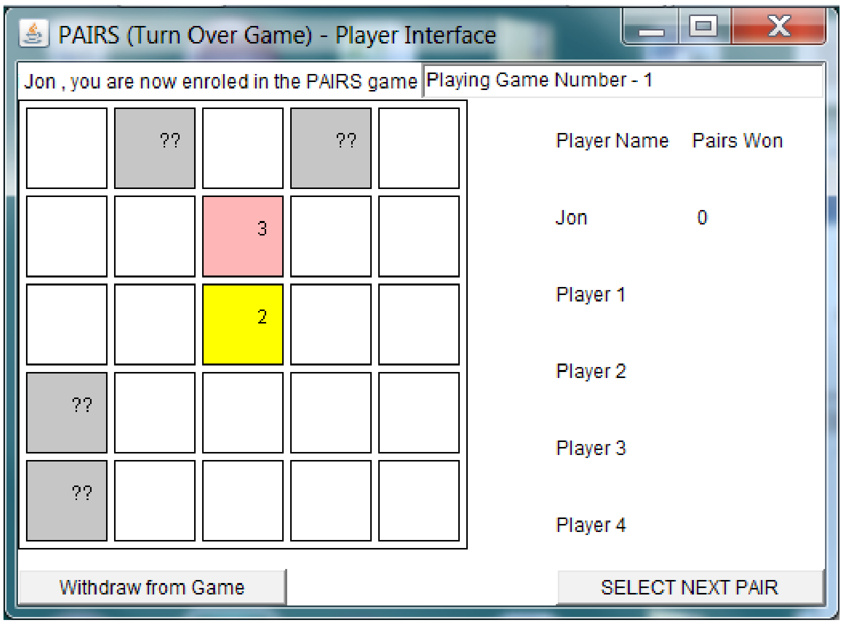
When all the cards in the current game have been claimed, the Game Controller will create a new game, and send it to each of the enrolled Players.

A Player can withdraw from the game at any time.

The main area of contention in this design occurs when more than one Player claims the same pair. The Game Controller has to be able to deal with this situation, informing the Players which of them were successful.

1.3.2 Player Interface

The Player Interface is shown in Figure 1-2. It is shown in a state where a Player named Jon has enrolled in the game. The player has revealed two cards, which do not match and thus the 'SELECT NEXT PAIR' button has been enabled. Once that button is pressed the revealed cards will revert to the 'greyed out' state and the player will be able to reveal more cards.



Jon , you are now enroled in the PAIRS game | Playing Game Number - 1

Player Name	Pairs Won
Jon	0
Player 1	
Player 2	
Player 3	
Player 4	

Figure 1-2 The Player Interface

1.3.3 Messages

Messages can be sent in both directions, from the Player to the Game Controller and vice-versa.

1.3.3.1 From Player to Game Controller

Enrol Player – sends enrolment data, once the player has input required data

Withdraw Player – sends a request to withdraw the player from the game

Get Game Details – a request for the controller to send the current Game Details

Claim Pair – contains details of a matched pair of cards that the Player is claiming

1.3.3.2 From Controller to Player

Enrol Details – details to update the names of the players

Game Details – the current state of the game including available cards and player scores

1.3.4 Player Internal Structure

The Player contains a user interface that comprises two parts. The first, concerns the creation of the graphical output and the second deals with events such as button and mouse presses and textual input. The design process means we can deal with these separately because, as we shall see in Chapter 11, each element of a user interface has its own process based architecture which means that such components can be integrated more easily into a parallel system design. Thus when a button is pressed it communicates a message that contains the current text associated with the button. The text in a button can be changed by sending a text string to the button process. Mouse events are dealt with in a similar manner in that a mouse event process communicates the mouse events so they can be processed by another process.

A first design for the Player might look something like that shown in Figure 1-3.

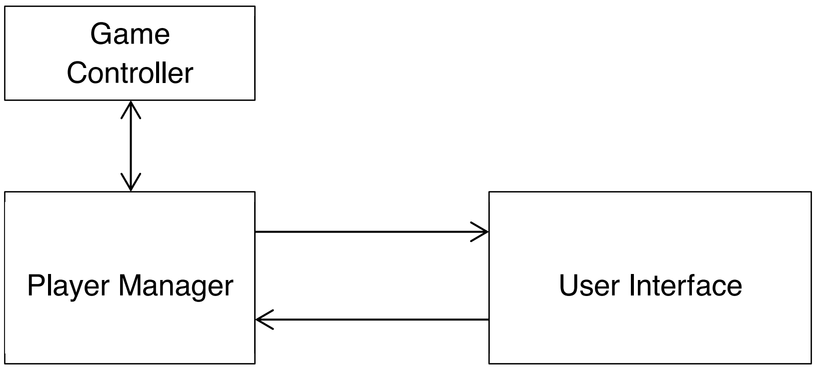


Figure 1-3 A First Player Design

In this design we recognise the fact that there are two primary Player components the Player Manager and a User interface and that we send messages between these components.

The Player Manager deals with interactions between the Player and the Game Controller, while at the same time dealing with interactions from the User Interface. This is a reasonable design until we consider the effect of Mouse Events on the Player Manager. Mouse Events happen in bursts as the mouse is moved or pressed and further the number of events generated is large, most of which are not significant to the operation of the system. We are only interested in mouse presses that occur in a greyed out square. This in the next stage in the design; mouse events are separated from the other interactions with the User Interface, such as Label and Button processing and the update of the paintable Canvas that holds the representation of the cards.

Figure 1-4 shows the next design, where a Mouse Event Buffer has been introduced. This is sent all mouse events but then filters these so that only mouse presses from the card canvas are retained and then passed to the Player Manager.

This is an improvement but still does not provide a completely satisfactory solution. The Player Manager really only wants to deal with mouse presses that are contained within a square that is currently greyed out. This processing can be placed in a separate process. The Player Manager then simply has to determine whether a pair of cards matches, deal with update of the interface and possibly claiming a matched pair of cards.

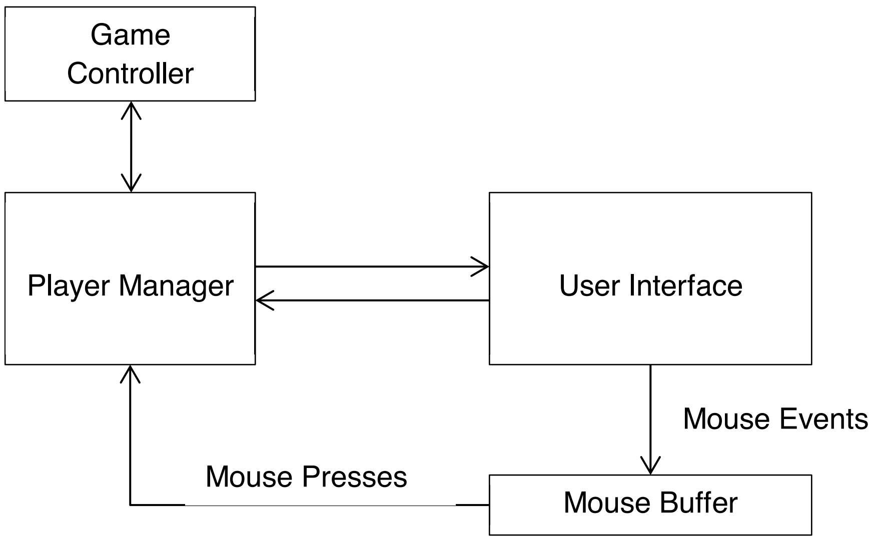


Figure 1-4 Separation of Mouse Events

The revised design is shown in Figure 1-5. The Matcher simply determines whether the mouse press is within a square that is currently greyed out.

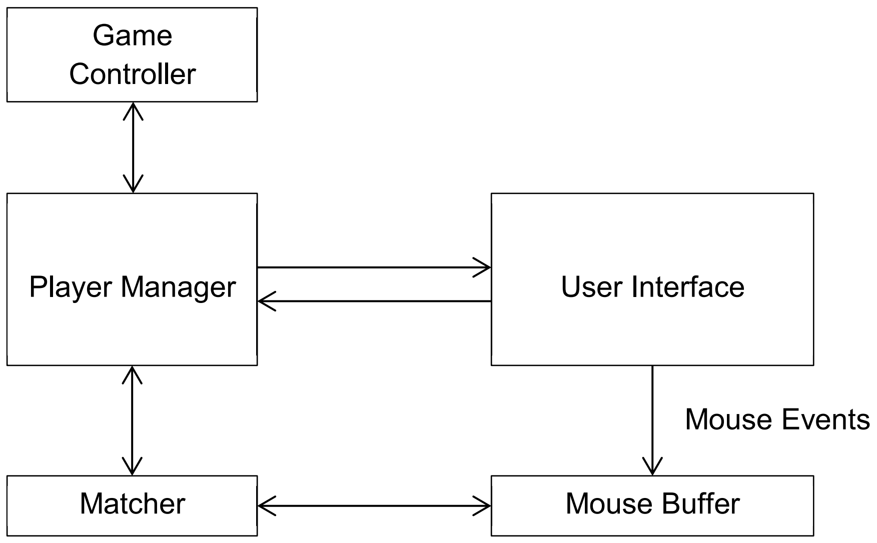


Figure 1-5 Inclusion of Matcher Processing

A further refinement has also been introduced as indicated by the bi-directional connection between the Player Manager and the Matcher and the Matcher and the Mouse Buffer. The Player Manager will know when it needs another point either to be the first or second in a pair. Thus it can ask the Matcher for a valid point, which it then can input and process. Similarly the Matcher can only deal with a press at specific points in the algorithm so it requests a mouse press event from the Mouse Buffer which it then inputs and processes. This refinement, may at first sight appear to complicate the design, however, as we shall see in Chapter 8, this implements a specific design pattern that we shall use throughout the remainder of the book. This so-called client-server design is fundamental to the design of parallel systems that are deadlock and livelock free.

1.3.5 Player Manager Communications

Initially, the only action the User Interface permits is to enable the user to input their name and also to type in the IP-address of the Game Controller. The latter enables the dynamic connection of the Player to the Game Controller and the former enables the Player to enrol in the game. The Game Controller responds with the list of players currently enrolled in the game with the new player's name added to the list and the number of claimed pairs set to zero.

Once enrolled, the Player can withdraw from the game at any time. The problem is the time when this withdrawal occurs is not known. Thus the Player Manager has to be able to deal with it at any time. This is referred to as non-deterministic behaviour in that it is known when an event can occur but not when. The language architecture includes a specific language structure that captures this type of behaviour and is known as a non-deterministic choice. It is fundamental to the design of parallel systems and will be described further in Chapter 3.

The rest of the behaviour is relatively straightforward, once we are aware of the possibility of a withdraw event is non-deterministic and can build that into the design. The Player Manager can request the co-ordinates of a card from the Matcher. The Matcher will respond as and when it has a valid card. Once the Player Manager has two valid cards it can check to see if they correspond. If they do it can send a Claim message to the Game Controller, which will respond with the updated game state including the number of pairs each player has achieved.

If the pair of cards does not correspond, then the Player Manager can enable the 'SELECT NEXT PAIR' button on the User Interface. Only when that button is pressed by the player will the cards return to the greyed out state and the player will be able to select another pair of cards.

1.3.6 Summary

The above description has glossed over a number of important ideas, which will be explored in more detail later on in the book. It has, however, introduced the basic fundamentals that concern the design and implementation of parallel systems. This is that we pass messages between processing elements hence the concept of Communicating Process Architectures. The designer has to manage these interactions and that we have a very small number of design principles that need to be followed to build functional, correct and maintainable parallel systems.

1.4 The Basic Concepts

The fundamental concepts that we shall be dealing with, when designing and thinking parallel are:

Process,
Channel,
Timer,
Alternative.

In comparison to other concurrent and parallel based approaches, the list is very small but that is because we are dealing with higher-level concepts and abstractions. Therefore it is much easier for the programmer to both build concurrent and parallel systems and also to reason about their behaviour. One of the key aspects of this style of parallel system design is that processes can be composed into larger networks of processes with a predictable overall behaviour.

1.4.1 Process

A process, in its simplest form, defines a sequence of instructions that are to be carried out. Typically, a process will communicate with one or more processes using a channel to transfer data from one to the other. In this way a network of processes collectively provide a solution to some problem. Processes have only one method, `run()`, which is used to invoke the process. A list of process instances is passed to an instance of a `PAR` object which, when `run`, causes the parallel execution of all the processes in the list (Kerridge, et al., 2005). A `PAR` object only terminates when all the processes in the list have themselves terminated.

A process encapsulates the data upon which it operates. Such data can only be communicated to or from another process and hence all data is private to a process. Although a process definition is contained within a `Class` definition, there are no explicit methods by which any property or attribute of the process can be accessed.

A network of processes can be invoked concurrently on the same processor, in which case the processor is said to interleave the operations of the processes. The processor can actually only execute one process at a time and thus the processor resource is shared amongst the concurrent processes.

A network of processes can be created that runs on many processors connected by some form of communication mechanism, such as a TCP/IP based network. In this case the processes on the different processors can genuinely execute at the same time and thus are said to run in parallel. In this case some of the processors may invoke more than one process and so an individual processor may have some processes running concurrently but the complete system is running in parallel. The definition of a process remains the same regardless of whether it is executed concurrently or in parallel. Furthermore the designer does not have to be aware, when the process is defined, whether it will execute concurrently or in parallel.

A network of processes can be run in parallel on a multi-core processor in such a way that the processes are executed on different cores. We can thus exploit multi-core processors directly by the use of a process based programming environment. The exploitation of multi-core processors will result in those processes running on the same core executing concurrently and those on different cores in parallel.

Throughout the rest of this book we shall refer to a network of parallel processes without specifically stating whether the system is running concurrently or in parallel. Only when absolutely necessary will this be differentiated.

1.4.2 Channel

A channel is the means by which a process communicates with another process. A channel is a one-way, point-to-point, unbuffered connection between two processes. One process `write`s to the channel and the other `read`s from the channel. Channels are used to transfer data from the outputting (writing) process to the inputting (reading) process. If we need to pass data between two processes in both directions then we have to supply two channels, one in each direction. Channels synchronise the processes to pass data from one to the other. Whichever process attempts to communicate first waits, idle, using no processor resource until the other process is ready to communicate. The second process attempting to communicate will discover this situation, undertake the data transfer and then both processes will continue in parallel, or concurrently if they were both executed on the same processor. It does not matter whether the inputting or outputting process attempts to communicate first the behaviour is symmetrical. At no point in a channel communication interaction does one process cycle round a loop determining whether the other process is ready to communicate. The implementation uses neither polling nor busy-wait-loops and thus does not incur any processor overhead.

This describes the fundamental channel communication mechanism; however, within the parallel environment it is possible to create channels that have many reader and / or writer processes connected to them. In this case the semantics are a little more complex but in the final analysis the communication behaves as if it were a one-to-one communication.

When passing data between processes over a channel some care is needed because, in the Java and Groovy environment, this will be achieved by passing an object reference if both processes are executing concurrently on the same processor. In order that neither of the processes can interfere with the behaviour of each other we have to ensure that a process does not modify an object once it has been communicated. This can be most easily achieved by always defining a new instance of the object which the receiving process can safely modify.

If the communication is between processes on different processors this requirement is relaxed because the underlying system has to make a copy of the data object in any case. An object reference has no validity when sent to another processor. Such a data object has to implement the `Serializable` interface.

If the processes are running on a multi-core processor then they should be treated as processes running concurrently on the same processor because such processes can share the same caches and thus processes will be able to access the same object reference.

1.4.3 Timers

A key aspect of the real world is that many systems rely on some aspect of time, either absolute or relative. Timers are a fundamental component of a parallel programming environment together with a set of operations. Time is derived from the processor's system clock and has millisecond accuracy. Operations permit the time to be read as an absolute value. For example, processes can be made to go idle for some defined, sleep, period. Alarms can be set, for some future time, and detected so that actions can be scheduled. A process that calls the `sleep()` method or is waiting for an alarm is idle and consumes no processor resource until it resumes execution.

1.4.4 Alternatives

The real world in which we interact is non-deterministic, which means that the specific ordering of external events and communications cannot be predefined in all cases. The programming environment therefore has to reflect this situation and permit the programmer to capture such behaviour. The alternative captures this behaviour and permits selection between one or more input communications, timer alarms and other synchronisation capabilities. The events over which the alternative makes its selection are referred to as guards. If one of the guards is ready then that one is chosen and its associated process carried out. If none of the guards are ready then the alternative waits, doing nothing, consuming no processor resource until one is ready. If more than one is ready, it chooses one of the ready guards according to some selection criterion. The ability to select a ready guard is a crucial requirement of any parallel programming environment that is going to model the non-deterministic real world.

1.5 Summary

This brief chapter has defined the terms we are going to use during the rest of the book. From these basic concepts we are going to build many example concurrent and parallel systems simply by constructing networks of processes, connected by channels, each contributing, in part, to the solution of a problem. Whether the network of processes is run in parallel over a network, in a multi-core processor, or concurrently on a single processor has no bearing upon the design of the system. In some systems, the use of multiple processors may be determined by the nature of the external system and environment to which the computer system is connected.