Contents

Chapter 1

Introduction

This paper deals with the game of Kriegspiel, and the development of an interface, called JKrieg, to play it on a computer. It contains an overview of what Kriegspiel is, and why it is deeply different from chess; then, it goes into detailing how the program works, why it works the way it does, and what tools and techniques were used to make it work that way. This paper is structured along the metaphorical path taken by its author when he was presented with the challenge of investigating this topic and creating such an application.

For the above reasons, the first section is devoted to explaining the core concepts and rules of Kriegspiel. The aims and requirements for the interface are then listed, followed by some theoretical results behind the interface itself. Chapter 4 deals with the design and implementation stages of the project, examining the structure and making of JKrieg in detail. In Chapter 5, the attention shifts to the final testing step. Finally, the two appendixes contain additional material, not strictly related to a single stage of the project: the former is an abridged postmortem for the JKrieg application, giving some personal insight on the development process as a whole. The latter describes the making of the custom chess diagrams featured throughout this paper, and how they were obtained using the LaTeX typesetting language.

The author would like to thank all the people who helped bring this work to reality.

Chapter 2

Kriegspiel

Kriegspiel is a variant of the game of chess. It mainly differs from orthodox chess in that players can only see their own pieces, and must refer to a neutral entity – the **umpire**, or referee – to acquire additional, but always incomplete, information. A far more detailed discussion on Kriegspiel is given in [2].

2.1 History of Kriegspiel

Kriegspiel was invented in the late XIX century in England, by Michael Henry Temple. Its creator drew inspiration from the highly popular war games played throughout Europe, and especially in Germany and Prussia (Kriegspiel means **war game** in German). These games, variants of which still enjoy a large number of followers nowadays, were realistic war simulations played on a tabletop environment representing a real battlefield, with miniature toy soldiers standing for units and regiments of different types, each with different features and carried weaponry. These games were extremely popular among high-ranked military officers, generals and even Emperors themselves, because they were believed to improve and refine one's tactical skills.

One of the most widely played war games implemented the so-called "fog of war", that is, it prevented a player from seeing enemy units not in visual contact with any friendly brigade. This was accomplished by playing on three different tables, one for each player plus one for the umpire. The umpire would record every movement and action on his table, notifying players about enemy sightings and resolving combat out of military experience or numerical tables.

Temple adapted this concept to the game of chess, creating a variant that was to be played on three chessboards, and deciding what pieces of information the umpire should reveal to the players. As with other variants, and for the very reason that variants are not orthodox, several versions of Kriegspiel were introduced and played as the game became more and more widespread. While all versions agreed on that players should never interact with each other and always refer to the umpire, their differences lay in the way information is passed from and to the umpire. For example, one version requires players to declare a sequence of move attempts; the umpire then executes the first legal move in the list.

In this paper we will make use of the Kriegspiel rules currently being enforced on the **Internet Chess Club**. This set of rules was designed to allow for fast and intuitive gameplay over the Internet.

2.2 Rules of Kriegspiel

Every rule in orthodox chess also applies to Kriegspiel, with two exceptions: the game will *not* result in a draw because of a position being repeated three times, nor will a draw occur due to the lack of captures or pawn actions for 50 moves in a row. Aside from these exceptions, the major innovation of Kriegspiel over orthodox chess is that **players cannot see their opponent's pieces**, and they have to make guesses concerning their location.

Given the incomplete-information nature of Kriegspiel, a new entity is introduced to act as a medium between the players: the umpire, or referee. Only the umpire has access to complete information, and it performs all communication with players (players cannot directly interact with each other in any way). Before each player's turn, the umpire transmits a message containing additional information (note that, although the message mainly concerns the current player, both players actually receive the message). This information includes:

- Number of pawn tries for the current player. This is defined as the

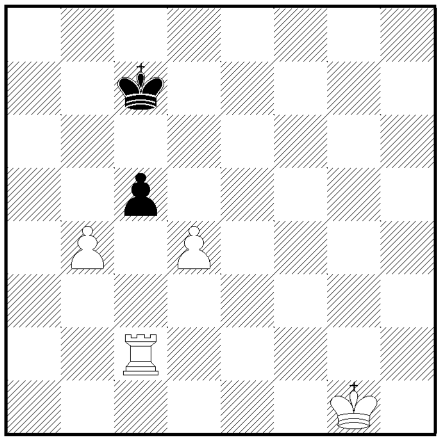


Figure 2.1 Pawn tries.

"number of legal capturing moves using pawns". For example, in **Figure 2.1** White has two pawn tries (even if on the same enemy piece), whereas Black has none (captures are not legal because they would leave the King in check). Pawn tries include possible en passant captures.

- A capture happened as a result of the last move. The message specifies where the capture took place and whether the captured piece was a pawn or another piece (in this case, the actual piece is not revealed).
- The current player's King is in check. The message also reveals the check type, which can be "rank", "file", "short/long diagonal" (from the King's point of view), or "knight".

In addition, if a player attempts an illegal move, the umpire will notify that player only that he or she should make another move instead, without revealing the reason making that move illegal. The opponent will receive no notification.

2.3 The challenge of Kriegspiel

Under the computer scientist's point of view, Kriegspiel offers a completely different challenge than the traditional game of chess. This is true in regard to both artificial intelligence algorythms and interfaces. As of now, despite the game's growing popularity, neither highly skilled artificial players nor specific interfaces exist for Kriegspiel.

The main reason behind this lack of working applications is that Kriegspiel, while borrowing most of its rules from chess (arguably the game to which programmers and computer scientists have been devoting the highest attention), is actually an entirely different game requiring completely new techniques. Artificial players capable of master-level Kriegspiel performance would share very few similarities with their orthodox chess counterparts: only piece mechanics would remain the same. In fact, virtually every artificial chess player is based upon the concept of the **minimax** algorhtym, which consists in creating a tree in which every node represents a move, and associating a convenience factor to each. Hence, finding the best possible move reduces to scanning the tree searching alternatively for the best and worst choices (representing the program and its opponent's move, respectively). This concept is thoroughly explained in [1]. It is easily seen in [2] that this approach cannot be applied to Kriegspiel, for the simple reason that a tree of known moves cannot be constructed.

A similar problem arises when it comes to designing interfaces. By definition, an interface provides information. But in the case of Kriegspiel, such information is necessarily incomplete, and the application must provide intuitive means to display probabilistic data, guesses. In addition, as will be shown in the next chapter, there is a significant amount of additional information hidden inside the seemingly barebones messages sent by the umpire. A capable interface must figure out as much data as possible, as this process might spell the difference between victory and defeat, all other things being equal.

Chapter 3

Requirements & Research

This chapter deals with the goals and expectations that JKrieg was to meet. It describes its planned features in detail, with a particular stress upon providing an accurate representation for hidden information, without specifying how these features were actually implemented, which is the focus of a later chapter. Planning out layouts and features proved to be a non-trivial task, because of the afore-mentioned lack of existing specific interfaces for Kriegspiel. With no models to follow or criticize, every step was a step in a new direction, and a pioneering work in some ways.

3.1 Requirements

- As a minimum requirement, the program must provide a fully-functional Kriegspiel interface for online play on the **Internet Chess Club**. Because this service is text-based, it must provide means to send text commands and receive messages from the server.

- The program must render the virtual chessboard **in graphics mode and in a intuitive fashion**. This includes a wise choice of piece shapes and board colors.

- The program must keep track of elapsed time and disconnections.

- The program must feature intuitive gameplay mechanisms, most notably **drag'n'drop** to perform moves.

	Piece	Check	King
White	○	⊕	♔
Black	●	⨁	♚

Table 3.1 Kriegspiel guess tokens, as implemented in JKrieg.

- The program must provide **graphical representation for each and every message from the umpire**. This includes recognizing pawn tries, captures, checks, and making guesses on the location of enemy pieces. The program should extract as much information as possible from the umpire, possibly making use of **a new, intuitive symbology**.
- The program must offer some kind of help to the player during the end-game stage, when only a few pieces are left.
- The program must keep a count of captured enemy pieces.

3.2 Symbology

The first decision that was taken immediately after reviewing the core requirements, was that, with JKrieg having to somehow display probabilistic data inside a chessboard, a new notation was in order. The interface must be able to inform the player that a piece might be there, or it might not. It must also warn about incoming checks and where they originate from. The ideal notation had to be simple and originate from a constant, easily recognizable visual pattern. In the end, the **guess tokens** in **Table 3.1** were chosen, and called **Piece**, **Check** and **King** token, respectively.

As it is easily seen, **guess tokens are all circular in shape**. This not only does set them apart from the regular pieces of chess, none of which resembles a circle, it also hints at their analogous meaning – probability. Also, *guess tokens appear at various degrees of transparency on the chessboard.* The more opaque the token, the higher chance there is that an enemy piece actually be in that case.[1]

[1]This feature was implemented in JKrieg; however, for printing reasons, transparency

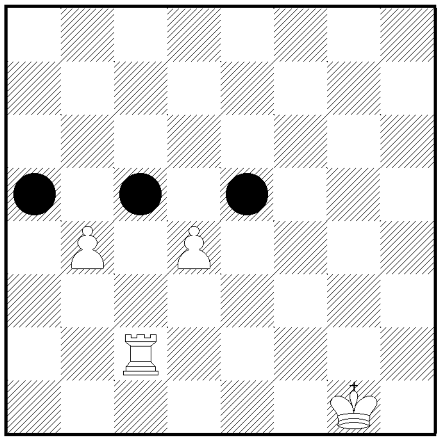


Figure 3.1 The umpire says: "2 pawn tries."

- The **Piece token**, carrying no other distinguishing sign, simply informs that a generic enemy piece might be hiding there. It is activated in response to pawn tries and captures: in the former case, all the possible target cases are highlighted; in the latter, a fully opaque token is put on the case where the capture takes place, meaning that the presence of an enemy piece there is now certain.

  Let us demonstrate the usage of Piece tokens in regards to pawn tries. **Figure 3.1** depicts what JKrieg would show to the White player, if the actual piece disposition was the one portrayed in the last chapter (**Figure 2.1**). This diagram, as well as the following ones, portray what JKrieg will have to display to the player, knowing the position of his or her pieces as well as the latest umpire message.

  The tokens appear wherever a pawn capture might take place.

- The **Check token** warns a player that his or her King is in check, and the offending piece might be found on the selected case. For example, in **Figure 3.2**, the White King finds himself threatened by a Knight. Later in this chapter, it will be shown that it is sometimes possible to narrow down the choices and highlight a lower number of cases.

will not be shown in figures throughout this paper.

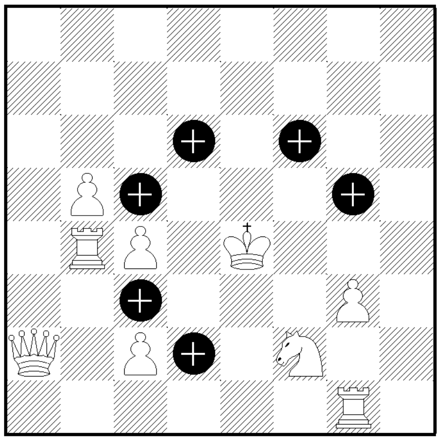


Figure 3.2 The umpire says: "Knight Check"

In order to facilitate chess players in using the interface, the Check token is identified by a '+' sign, which is universally known as the symbol for check positions in all literature and game transcriptions.

- The **King token** is the counterpart to the Check token; when a player puts the enemy King in check, these tokens list the locations where it might be hiding. As it is described later, placing this kind of token raises the most problems. **Figure 3.3** depicts a possible disposition for the Black pieces, causing the check in **Figure 3.2**.

Now that the building blocks for the interface have been introduced and explained, the following sections of the chapter will be devoted to showing how they can be used to provide non-trivial, immediate information to the user. JKrieg supports the following results with a few small exceptions that will be explicitly pointed out, and the implementation of these theorectical properties took the better part of the time resources devoted to the project. In fact, the complexity of the situations originating from the seemingly laconic messages of the umpire was somewhat underestimated at the beginning of the development cycle; while the umpire can only generate a handful of messages (eight including the five check types, plus the notification of illegal moves), they can assume different hidden meanings and their combina-

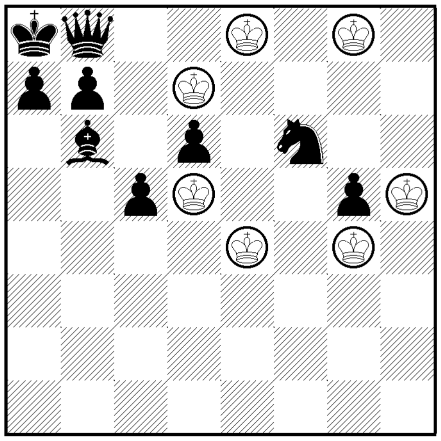


Figure 3.3 Black plays ...Nh7-f6, the umpire says: "Knight Check"

tions make for additional interesting cases. This partly explains why strong artificial players for this discipline have not been programmed yet.

3.3 The Piece Token

As stated above, this token makes its appearance whenever pawn tries or enemy captures take place. Pawn tries are vital tools in finding out the disposition of opponent pieces, which makes pawns extremely precious in Kriegspiel: they are the only piece able to gather information on their surroundings without attempting to move.

The trick of putting Piece tokens wherever pawn captures might happen is a trivial, but effective one. It provides immediate information, and the player can often guess, if not the exact capturing move, at least the most likely area on the chessboard. *This method is partially inaccurate in the case of en-passant captures, since the location of the captured piece does not actually coincide with that of the token, but for the purposes of Kriegspiel, this is rather unimportant.*

It would be possible to highlight **opponent pawn tries** as well. Each time the opponent receives a pawn try notification, the program would then place a token on every legal case which might host the pawn. However, this would probably clobber the chessboard with pawn tokens, so it was decided not to implement this function. JKrieg will still warn a player textually whenever his or her opponent gets pawn tries.

The other instance of Piece token insertion happens upon opponent capture. A token with a 100% likelyhood (fully opaque) replaces the captured piece. This token will then fade out with each subsequent move, representing the odds of the piece still being on the same case reducing over time. Since the opponent is aware that the location of the offending piece is now well-known, he or she will tend to move it again if possible, in order to return it to the shadows. Because of this tendency, this type of token fades out quickly, its opacity being halved on every opponent move. When a fixed threshold is crossed, or a friendly piece "explores" the case, the token is removed from play.

Pawn tries, or lack thereof, can also interact with the other two tokens to help narrow down choices. These techniques are described in the following sections.

3.4 The Check Token

The Check token highlights the possible locations of the piece attacking the player's King. Its basic behavior is quite trivial: the program simply stamps tokens starting from the cases adjacent to the King until a friendly piece or an edge of the chessboard are met. As it is easily seen, Knight checks work in a slightly different way, and only compatible empty cases get touched in this case.

However, if one remembers the rules of chess, there is much more information which can be extracted from a check notice.

- **If a capture took place on the last move...** Here, there are two basic cases. The former happens when the capturing piece lands into a case and directly threatens the player's King. The latter happens when the capturing piece uncovers another piece behind itself, clearing its 'line of sight' towards the player's King. These two cases seem to be easily recognizable from one another; it suffices to **determine whether the case in which the capture took place is compatible**

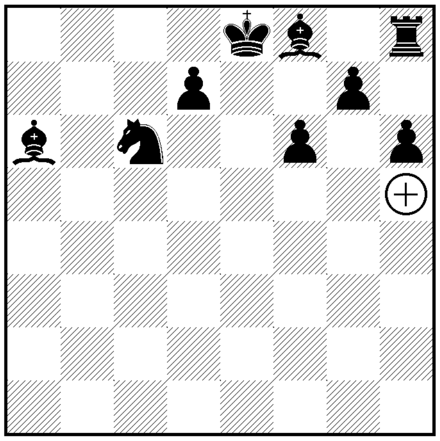
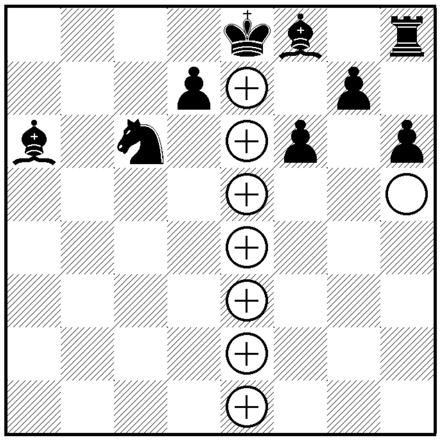



Figure 3.4 In both diagrams, the umpire announces a capture in h5. In the left diagram, the umpire says "short diagonal check". In the right diagram, the umpire says "file check".

with the check type being revealed by the umpire. In other words, if the player is told his or her King is under File check and the capture did not happen in the same file as that of the King, then that piece has nothing to do with the check. On the other hand, if the capture actually took place in the right File, then the offending piece must be responsible for the check, and only the target square must be lit with the Check token, as exemplified in **Figure 3.4**.

- **If pawn tries are possible, then the attacking piece, and only it, can be captured.** As stated in the rules of chess, there are three ways a player can react to a check: by moving the King out of enemy range, by moving a piece to cover the King, or by capturing the offending piece. This means that, if pawn tries can be performed, the attacking piece can be the only target. This narrows down the possible locations to those that are compatible with the current check type *and* a possible target of a pawn try. Most of the time, only one or two cases will meet these requirements, and the location of the enemy piece can be accurately described.

An example of this technique is given in **Figure 3.5**.

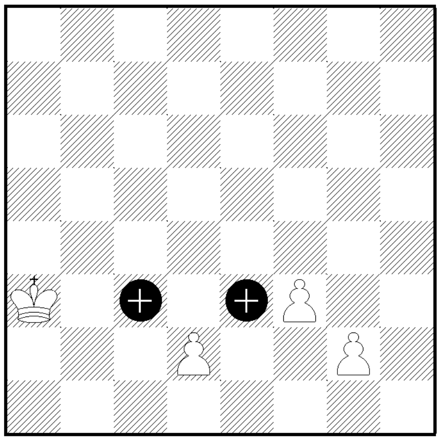


Figure 3.5 The umpire says ”Rank check, 1 pawn try”

- As an additional rule, **if pawn tries are not possible, the attacking piece cannot be in a capturing position for any pawn, unless that pawn is protecting the King from another piece.** The wording of the previous sentence is actually more complicated than the rule itself: it simply means that, if no pawn tries are announced by the referee, and the player's King is in check, in general the attacking piece cannot be attacked by the player's pawns *unless* that pawn is being blocked by another piece. Because a picture is worth a thousand words, **Figure 3.6** explains this fact intuitively, showing that, in Kriegspiel, nearly every rule has its exceptions. The red pawn might be blocked by the possible presence of a Bishop or a Queen on the King's diagonal. On the other hand, the blue pawn has no constraints binding it to its King, and as such the lack of pawn tries attests that the piece giving check cannot be found in c5.

This particular case has not been implemented in the current release of JKrieg, as its benefits did not justify the time resources needed for its coding. It is being explained here to convey the subtlety and complexity of the information that one could elaborate basing themselves on umpire messages.

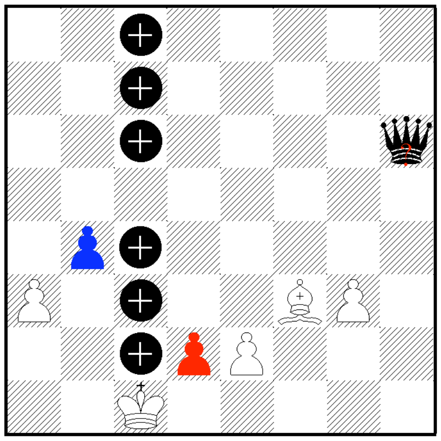


Figure 3.6 The umpire says "File Check"

3.5 The King Token

The King token is complementary to the Check token; while the latter shows up whenever the player's King is in check, the former makes its appearance if the **opponent's** King is in check. As mentioned earlier in this chapter, the King token offered a harder challenge to the interface developer, because the check notifications sent by the umpire are King-centered, that is, they describe the type of check basing on the location of the attacked King, which is, of course, generally unknown to the opponent and their playing interface as well. This raises a number of problems, most of which occur when **diagonals** are involved.

- The first problem consists of **determining what allied piece is attacking the opponent's King**. Note that this problem is in many ways specular to the one pointed out in the last section, when dealing with captures (**Figure 3.4**). Only, this time the dilemma does not limit itself to the 'capture+check' combination; it is a constant presence that the interface has to deal with.

- The basic approach to this issue is analogous to the one adopted in the last section; that is, the program **checks whether the last moved piece is compatible with the umpire's information.** In other

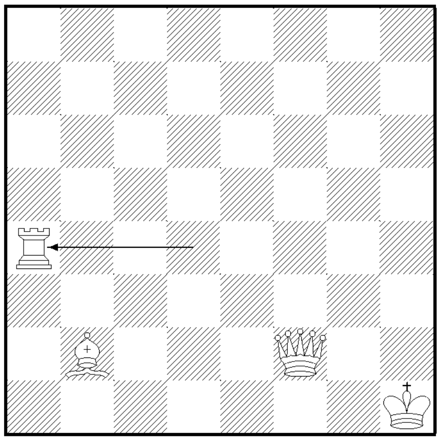


Figure 3.7 Following Rd4-a4 . . . the umpire says "Long diagonal check". Which one is guilty, the Bishop or the Queen?

words, if the player moves a Rook and the umpire notifies a long diagonal check, then there is no way a Rook could have caused such a check. There must be another allied piece capable of a long diagonal check, previously hidden by the Rook.

Unfortunately, what if the Rook was hiding **two** Bishops on two different diagonals? **In general, it is not always possible to determine what piece is causing a check.** See **Figure 3.7** for a visual example. In cases such as this, the interface will have to select all applicable pieces and consider all the possible choices.

- While the previous statement proves that the placement of King tokens is significantly more challenging than that of Check tokens, it is often possible to rule out several location through relatively simple algorithms. First and foremost, any **diagonal check** can be either short or long, and if a location belongs to the wrong diagonal, it has to be discarded. Additionally, if the attacking piece is the one which has just been moved (or so believes the program), **the diagonal along which it has moved is to be ruled out**, since the diagonal was already under friendly control. It should be noted that **this does not apply if a capture took place after the last move**, since the King might

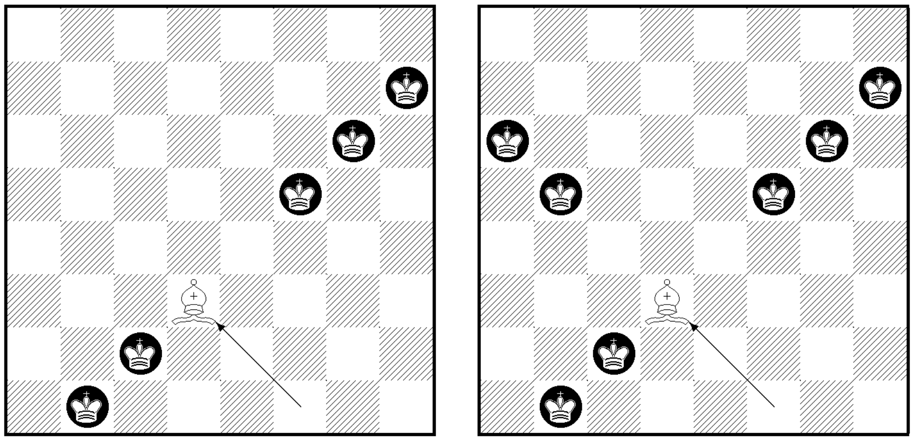


Figure 3.8 In both diagrams, White plays Bf1-d3 and the umpire says "Long diagonal check". In the diagram to the right, the umpire also announces a capture in d3.

be found beyond the captured piece. In this case, only the half diagonal between the original position and the new position of the attacking piece is to be excluded, as shown in **Figure 3.8**.

- A very peculiar case occurs whenever the umpire announces a **double check**. This happens if a King is threatened by two pieces at the same time, and leads to an interesting, and possibly unexpected, result. Under these circumstances, **it is often possible to pin-point the exact position of the enemy King**. The reason is quite simple: the two sets of King tokens generated by the two pieces (one of which might actually be the union of two different pieces, because of the undecidibility shown in **Figure 3.7**) have to be **intersected**, which will usually narrow the choice down to one or two cases at most. For instance, in **Figure 3.9**, the umpire announces two ongoing checks at the same time, thus mercilessly revealing the location of the opponent's King.

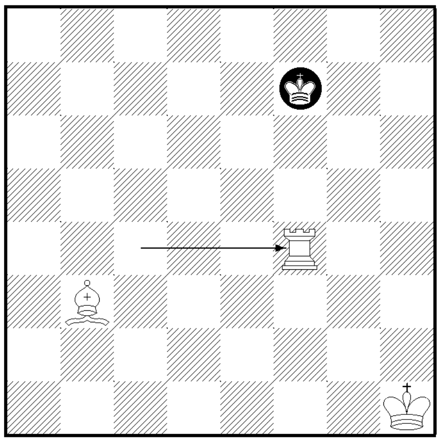


Figure 3.9 White plays Rc4-f4 . . . , the umpire says "File check, Long diagonal check"

3.6 The Zone of Control

The guess tokens discussed so far may be a powerful tool in discovering and presenting more information than meets the eye, but the help they provide is mainly a **short-term** one. For example, knowing that the enemy King might be in a small selection of places across the chessboard is surely useful, but on the next turn, it will usually be gone, with little or no clues concerning its new location. Thus, if the tokens do support player **tactics**, some tool is still needed to support player **strategy**, that is, offer suggestions about the overall arrangement of one's pieces.

This is especially true during the endgame, when few pieces remain in play and knowing which areas of the chessboard are supposedly under the player's control is even more important, and the knowledge more accurate due to the lesser number of uncertainties. For these reasons, it was chosen to implement a system to visually indicate a player's current **Zone of Control**, providing a tool to intuitively spot both strenghts and weaknesses in the current structure of one's strategy.

Conceptually speaking, the Zone of Control is a very simple feature to add; each case on the chessboard **can change color** according to its status, and it will turn red if no allied piece is controlling it (keeping in mind that a piece does not control the case it is on). Of course, the term 'control' may be somewhat misleading, as there is no guarantee that a 'controlled' place of chessboard be actually reachable by the player's pieces – an enemy piece may be blocking the path. However, red squares do guarantee that the location is truly unprotected.

A visual example of a Zone of Control display is given in **Figure 3.10**. Some relevant information is immediately perceivable by the end user, who can make use of it to strenghten their defense and decide an attack plan. For example, not only is the Rook in a8 lacking any form of protection, but its whole diagonal is seriously underdefended, and should the pawn in d5 be captured, the player would be facing a potentially very dangerous attack. The same goes for the f6 pawn, unprotected and in an excellent position from which to threaten the major pieces of the Black player (especially with a Knight). Of course, generally the opponent will not be aware of such an opportunity, but having two undefended pawns out of five is ill-advised in Kriegspiel, where pawns are much more important than in regular chess, due to their probing capabilities in the form of pawn tries.

During the endgame, the Zone of Control also helps checkmating the enemy King. When only the King remains in play (which can be easily figured, as JKrieg will mantain counters for captured pieces), this method becomes very accurate, and the red squares inform the user about the possible hideouts of the runaway monarch, making it easier to track it down once the first check has been performed and the player begins to have more clues concerning its starting place. It also helps to prevent the almost defeated opponent from seeking and obtaining a stalemate, which would turn a victory into a draw; that is, the King must have at least one escape route (red case) until the time comes to deliver the killing checkmate blow.

From the above discussion, it is readily seen that the accuracy of the Zone of Control view improves as the game progresses. This means that its value is near to worthless during the initial stages of a game, and since JKrieg could not easily understand the right timing for activating it, it was decided to let the player manually choose to switch it on and off through a checkbox, so as to ensure maximum flexibility.

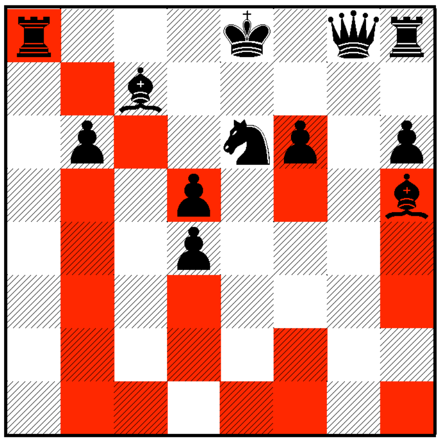


Figure 3.10 An example of Zone of Control display.

Chapter 4

Design & Implementation

This chapter deals with the actual implementation of JKrieg in the Java programming language, starting with some abstract planning, then to delve into more specific technical details concerning the final classes. The choice of Java came extremely natural to the developer, because the object-oriented paradigm lent itself very well to the problem to be faced, and the wide range of platform-independent factory classes that interact with the underlying system, hiding its complexity, would certainly simplify the development process. Moreover, Java's only real drawback, its lack of performance compared with other programming languages, did not really matter in this particular project, as visual interfaces do not require much processing power, and the advantages far outweighed the disadvantages.

4.1 Program Structure

Using Java as a programming language made it natural to implement Krieg as a modular, layered application. The first step taken during the design stage was to decide what and how many abstraction levels would be present, and how they would communicate with each other. Eventually, the three-level structure depicted in **Figure 4.1** was chosen, each layer solving a specific set of problems, and providing higher level information to the upper floors. These levels were implemented through the Java concept of "interface".

A Java interface can be thought of as a contract, an arbitrary number of related methods. In order to implement an interface, a class must specify each and every method declared in the contract - the details of what these method actually do is left to the implementing class. Under many aspects, interfaces behave like normal class definitions: for example, it is possible to have a method requiring an interface as a parameter. The method will then accept any object whose class implements the requested interface. This way, interfaces can become powerful tools for the purpose of producing extendable, mantainable and clean code: as any number of classes can implement the same interface, it is possible to code several specific, self-contained application behaviors while leaving the rest of the program unaffected by them. One of the golden rules of programming, according to which a code section should only be able to see what it strictly needs to perform its function, is then closer to being satisfied.

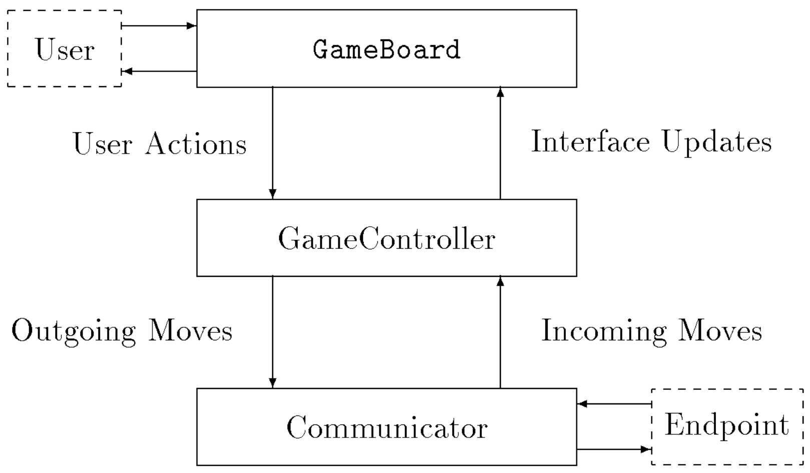


Figure 4.1 Structure of JKrieg.

This theory was therefore applied to Krieg, starting from its lowest level. The basic question here was: "Where will the program get its data from, and where will it send data to?". While Krieg was intended for online play with the Internet Chess Club, it was clear that a large part of the code would not need changes in order to make Krieg run under other protocols, or against a computer player. Therefore, the lowest layer was called **Communicator** (or Driver). The Communicator entity is charged with the task of starting games, and trasmitting information to and from the other endpoint. Of course, how this is performed would vary according to what the other endpoint is: for example, a Socket will be used in case of a remote server, or some form of interapplication communication, like shared files, when against an artificial player.

At this stage of design, the application is able to think in terms of moves and games - it does not need to know where this information comes from or goes to. The next important point follows another seemingly naive, but terribly serious question: "What game are we playing?". As the name itself states, Krieg was conceived in order to play Kriegspiel; but again, there are dozens of other chess variants, and of course chess itself; as different as they can be, they still share the majority of the rules. Rather than limiting the application to just one game, a new layer was added, called **GameController**. Another interface, a Controller handles a different set of rules: for example, Krieg includes two classes implementing it, OrthodoxChessController and KriegspielController, whose names are quite self-explanatory. With relatively little effort, one could code more implementing classes to support both changes in rules, like Progressive Chess, and changes in piece appearance itself, like Heraldic Chess (a variant involving piece colors).

A GameController's primary task is to perform and receive moves. Moves are generally received from the board, and delivered to the Communicator for processing. The Controller can execute legality checks, or simply ignore them altogether and leave the checks to the remote server (which is what most interfaces actually do in order to ensure maximum versatility).

The other interesting feature a GameController owns, is its being able to exercise control over the gaming board. Apart from changing piece positions and pictures, which is a rather obvious power, the Controller creates and mantains (if needed) a Java JPanel. This class is part of the Java Swing package, described in the "Tools & Techonologies" section. A JPanel basically groups a set of interface elements of any kind supported by Swing (even customized), ranging from classical buttons and checkboxes to any flavor of lists, hierarchic views, progress bars and popup menus. The panel is simply attached to the gaming board, and the GameController is responsible for reacting to events happening inside the panel, and accordingly changing the information it displays. The idea behind this choice is that the GameController will use the JPanel to show its specific information to the player. For example, both OrthodoxChessController and KriegspielController include two Timer components to keep track of elapsed time; in addition, the Kriegspiel panel provides more information about captured pieces and so on.

On top of a GameController sits the interface itself under the form of a **GameBoard** object. This object extends a **JFrame**, that is, it is a very specialized window that can be attached to a GameController. A GameBoard is an essentially *passive* object, making no decisions on its own and instead relying on the underlying GameController to perform the needed computations and update the interface accordingly. When the player attempts to move a piece on the chessboard, the GameBoard merely sends a message to the GameController, indicating the move being requested, and then completely forgets about it. It is the GameController that updates the board if needbe, whereas the GameBoard object only owns the code that actually draws the pieces on the window.

4.2 The Communicator Level

The Communicator interface makes up the lowest level in the JKrieg hierarchy. As the name implies, a class implementing this interface is able to communicate with another entity, called an **endpoint**. Because of the wonders of object-oriented programming, the upper levels will not need to make any assumptions concerning the nature of this endpoint, be it local or remote, client-server or peer-to-peer structured, human or artificial. Depending on the actual endpoint, the communication protocol and techniques will vary. In this section we will deal with Socket data transmission for play over the Internet Chess Club, but virtually anything is possible so long as the Communicator interface is respected, that is, if all the required methods are implemented.

One of the main aims of the Communicator layer is to consume incoming move notifications, in turn producing standardized diagrams which can be consumed by the higher level, *de facto* hiding the nature of the remote endpoint to the overlying GameController. This is accomplished through the **VirtualBoard** class, which was imported from the JChessboard package and edited to better fit the application's purposes. A VirtualBoard object provides an internal, uniform representation of the pieces on a chessboard at any given time. It also provides a few additional useful features, like telling whether a given case is being controlled by a player.

VirtualBoard objects are the medium through which incoming moves are shipped to the GameController. Instead of just notifying about the move, the Communicator sends the updated chessboard. The goal here is to ensure maximum scalability and expandability: there are a few chess variants where the concept itself of ”move” is a very peculiar one. Kriegspiel is one example of it, as the player technically receives no moves at all, but there are more situations where this issue shows up. For instance, in **Progressive Chess**, the first player performs a move, then the opponent makes two, then three, and so forth. So long as a few rules are followed (for example, only the last move in a sequence may give check), any array of moves can be played. Here, only transmitting the whole chessboard would save the day without introducing unnecessary complications.

However, VirtualBoard objects alone may not be sufficient to describe the situation on the gaming board. There might be additional information that cannot be directly stored within such objects, such as remaning time, or umpire messages in the case of Kriegspiel. This is why a Communicator will also construct an instance of the **MoveMiscellanea** class as a support to the VirtualBoard itself. This class is structured around the **dictionary** paradigm; that is, it stores a vector of **tag-value** pairs, thus associating a tag object, which is always a **String**, to a value object, which can belong to any class. For example, an acceptable value for the ”White Clock” tag would be ”2:23”. It is possible to search the list for a given tag, or retrieve them in sequence within a for loop.

The most crucial methods that need to be implemented are listed in **Table 4.1**.

As it is readily seen, some of these methods are declared as **public**, whereas others are **protected**. Access specifiers are a key feature of most object-oriented programming languages, as they make it possible to enforce the mechanics of encapsulation, which is the technique of hiding unnecessary details from external parts of the program, in order to reduce the occurrence of potential conflicts between different sections of code and better structure the software. In particular, methods declared as **public** may be always invoked from every location; on the other hand, **protected** methods may

Accessor	Ret. Type	Name	Parameters
public	**void**	`initCommunication`	()
public	**void**	`setController`	(**GameController** gc)
public	**boolean**	`isGameSupported`	(**GameSpecs** gameInfo)
public	**GameID**	`beginGame`	(**GameSpecs** gameInfo)
public	**void**	`endGame`	(**boolean** finished)
public	**void**	`writeMove`	(**Move** attempt)
protected	**void**	`doIOLoop`	()
protected	**String**	`readMessage`	()
protected	**void**	`writeMessage`	(**String** message)
protected	**void**	`parseMessage`	(**String** message)

Table 4.1 Listing of the most important methods in the Communicator interface.

only be called from inside the class which defines them, or any class inheriting from it (this case could have been prevented by declaring the method as **private**).

Thus, a Communicator acts like the proverbial black box, exchanging information with the upper layers through a very limited selection of public methods, and performing its inner workings via protected methods. Here is a brief summary of the purpose of the aforementioned methods in the Communicator interface.

- `initCommunication`. This self-explanatory method does all required initialization setup, instantiating needed classes and filling records and variables for later use. This method is expected to be invoked prior to any other action involving the Communicator.

- `setController`. This method accepts a **GameController** as its only parameter, that is, any object belonging to a class implementing the omonymous interface. The Communicator writes down the information in a variable; it will be required if the object is to do anything at all with the data it receives from the remote endpoint. The GameController is the upper level in the hierarchy, and it will be notified whenever something happens demanding its attention (which usually coincides with a new move taking place).

- `isGameSupported`. This method takes a **GameSpecs** object, returning a **boolean**. A GameSpecs is merely an information wrapper, containing the name of the game to be played, and other optional data, such as desired timing and time increment. The Communicator is supposed to return **true** if the chosen game with the chosen options can be played over the current protocol, **false** otherwise. For example, Kriegspiel may be played over the Internet Chess Club, but Heraldic Chess may not.

- `beginGame`. Again, this method accepts a **GameSpecs** object, and starts a game following the incoming specifications. It returns a **GameID** object containing more detailed infomation about the new game, or **null** if an error occurred. The information within can be compared with the seven basic tags of the **PGN file format**.

- `endGame`. This method stops the current game. It requires a **boolean** parameter, indicating whether the game has come to a natural end or is being aborted by the upper levels for some other reason; this detail is necessary to decide if and which messages to deliver to the endpoint.

- `writeMove`. This method is the key to outgoing communication with the endpoint, and is expected to be called by the overlying GameController. It takes a **Move** object as a parameter, which obviusly encodes the move being attempted. This class was imported by the GNU **JChessboard** and seamlessly integrated into JKrieg; basically, it contains the source and destination locations for the moved pieces, with some further flags reserved for special cases, such as pawn promotions.

- `doIOLoop`. This protected method implements the heart of the input/output operations. It performs all IO activity undefinitely, which is why it should never be called by the program main thread. In fact, the Communicator implementation for the Internet Chess Club also implements the **Runnable** interface, as it will be shown.

- `readMessage`. This method reads the next message coming from the endpoint (hanging the current thread if none is available, though this is left to the specific implementation), and then returns it a **String** object.

- `writeMessage`. This method sends a message to the endpoint; of course, the program must have made sure the message itself makes sense according to the protocol being used. This method may be synchronous or asynchronous; the choice is left to the specific implementation.

- `parseMessage`. This method, ideally called after `readMessage`, accepts a **String** and takes appropriate action, including notifying the attached GameController if needed.

4.2.1 The ICCDriver class

The **ICCDriver** class is the Communicator implementation allowing play over the Internet Chess Club. After a brief overview of the ICC's history and features, further details shall be given concerning the inner working of this class.

The ICC is the oldest of Internet Chess Servers, dating back to the eighties, when a small community of chess players foresaw the dramatic impact that the Net could have on the game of chess. Today, it is by far the largest chess community to be found online, often symultaneously hosting over two thousand players. Its code was completely rewritten in 1992, still mantaining full backwards compatibility, to improve in efficiency and compensate for the ever growing number of players. A much more in-depth discussion of the ICC is given in [3].

Basically, the ICC is a server that allows players to log on and find opponents to challenge. It supports player rating using the **ELO rating system**, which lets a player keep track of his or her performance in time. Aside from orthodox chess, over twenty variants can be played, and it is also possible to watch and review other people's games. On top of that, it offers the full capabilities of an IRC-like chat system. Most of these features are only available to registered members who pay a monthly fee, though login as a guest is allowed, thus letting anyone play an unrated game.

The Internet Chess Club makes use of a human-readable protocol. In fact, it is perfectly possible to log on and play games via a telnet client without the need for a graphical interface, even though the user experience is bound to be lacking and incomplete. Of course, the programmers introduced a mechanism to make data transmission a more program-friendly process; this is accomplished through the concept of **datagram**. In order to have the server send datagrams instead of the usual stream of messages (which is much harder for a program to parse), the following command has to be issued.

`set level 1`[1]

After the server has been instructed to do so, it will start to format its messages in a different fashion. The structure of a datagram is as follows.

[`command-number   player   message`]

Here, `command-number` is an opcode representing the incoming message type, `player` indicates the username of the player the message come from, and `message` contains the actual message text. This way, the program can figure when a message ends, what it is meant to do, and who sent it in the first place. There is a number of available opcodes, though JKrieg only handles a handful of these, simply trashing and ignoring the rest, as they are of little relevance to the game itself. **Table 4.2** shows a selection of command opcodes implemented in JKrieg, together with their meaning.

The message body contains all the relevant information. When the datagram transmits a move (more precisely, a **ply** or **half-move**), the body is a ASCII representation of a chessboard, with letters such as K and P standing for the various pieces. Uppercase letters represent White pieces, whereas lowercase letters are Black pieces. The datagram also contains time information, showing the clock for both players.

There is a method, `produceBoardFromText`, that transforms the ASCII-based chessboard into an equivalent VirtualBoard, ready to be consumed by the higher levels. This method, as well as other methods in the ICCDriver class, makes use of a support class, **RegexpChecker**, which implements a very simple parser for a subset of regular expressions. The **RegexpChecker**

[1]Level 2 datagrams are available on ICC, but their purposes and usage go beyond the scope of this paper.

Opcode	Meaning
2	**Move.** Represents a chessboard update during a game.
19	**Disconnection.** Opponent has disconnected from the game.
101	**Tell.** Privately sends a message to a single user.
108	**Forfeit.** User forfeits due to lack of time.
110	**Say.** Sends a message to everyone in channel.
121	**Issuing.** Match request sent to selected opponent.
123	**Accepted.** User has accepted match request.
207	**Draw.** Game has ended in a draw.
208	**Resign.** User resigns and concedes the game.

Table 4.2 A few ICC opcodes, as implemented in JKrieg.

never needs to be instantiated, because all of its methods are declared as **static**.

The ICCDriver receives datagrams from the Internet Chess Club via the **Socket** class inside the `doIOLoop` method, required by the interface contract. The socket returns bytes as they become available from the network connection, sleeping while the connection is idle (a useful feature of Java, preventing the program from polling the socket and wasting precious processor cycles). These bytes are then stored in a temporary buffer until a complete datagram can be extracted from it. The datagram is delivered to the `parseMessage` method, which will take appropriate action and usually invoke the driver's GameController.

In addition to Communicator, ICCDriver also implements the `Runnable` interface. This basically means that ICCDriver can be **run** as a separate thread, and it will just execute the `doIOLoop` method (which is basically the typical infinite input/output loop) until the thread itself is killed by the user quitting the application.

4.2.2 The ClientConsole class

As the Internet Chess Club features a text-based protocol with a very large number of different commands, and it would have been impossible to provide a graphical interface for all of these, it was decided to have a **console**, resembling the shell of a command line operating system, in which the player can issue commands and receive text feedback, whereas the graphical chessboard is only launched when games actually begin. The **ClientConsole** class provides such functionality. It is basically a **JFrame** containing a scrolling pane for incoming messages, and a text box for typing commands.

While seemingly simple, the ClientConsole class was written as a separate class instead of being incorporated into ICCDriver because of a few trickier passages involving the Java way of managing application events, specifically key press events. As a particular behavior was desired, that is, having the console send a message whenever the Return key is pressed and the command line has the focus (because executing a command by pressing the Return key is the expected behavior of any command line environment), it was necessary to define a custom **Keymap** event and attach it to the text box component. This element of complexity suggested that these operations should be shifted over to a separate class.

This also allowed to make a Client as general-purpose as possible. Its only constructor takes a **ActionListener** object as the only parameter. This represents the object that will be notified whenever the user types a command and presses the Return key – in JKrieg this object is the ICCDriver itself, since it also implements the `ActionListener` interface, but the console can notify anything conforming to said interface.

It is possible to send data to the console by means of the `write` command, which will cause the String parameter to be appended to the current console text. Again, it should be noted that, although this method will be invoked from the ICCDriver class (with the output string of datagrams obtained through the open Socket with the ICC server), a ClientConsole has no associations with it and can be reused by any class which might be interested in its functionality, including other Communicator implementations.

4.3 The GameController Level

The GameController interface defines the class specification for playing a particular game or chess variant, no matter how it is being played. It is the middle level between the graphical chessboard and the lower Communicator, and it exchanges messages and data with both. It holds nearly absolute power over the user interface, and it can replace ample sections of the interface itself to fit the particular purposes of the type of game being played. In the case of complex chess variants like Kriegspiel, it is by far the most challenging layer to program out of the three.

The idea behind this level is that each implementing class should be able to handle the rules and user interface for exactly one game. For instance, JKrieg comes with two working GameController implementations, namely **OrthodoxChessController** and **KriegspielController**. When the underlying Communicator receives confirmation that a game has started, it usually instantiates a controller of the correct class and memorizes it in a variable, though this is not a required behavior and the creation might occur elsewhere (for example, in the main function).

The above concept is a very powerful one, and allows to play the same game over any medium supporting it, **so long as the rules are identical**. The only problem with it arises when two systems enforce slightly different rules, yet actually referring to the same game. Typically, chess variants are not set in stone; as there is no official authority regulating their usage, and they are not nearly as old as the game of chess, they often come in several flavors that differ from one another in minor, but nonetheless important ways. As mentioned in Chapter 2, there are half a dozen different rulesets for playing Kriegspiel, and the one implemented by the KriegspielController class follows the rules enforced on the Internet Chess Club. This means that, if another Internet Chess Server decided to add Kriegspiel to their list of featured games, but conforming to the original ruleset by Michael Henry Temple, the KriegspielController class as it is now would not be able to play it correctly. Another class would have to be written, or, more conveniently, the existing class would have to be extended to provide support for the different rules, perhaps with an optional parameter in its constructor. Another approach would involve coding a **child class**, inheriting basic behaviors from KriegspielController and specifying different courses of action whenever the rulesets differ.

With the aforementioned *caveat* in mind, it is now possible to introduce the method listing for the GameController interface. This is shown in **Table 4.3**. It is to be noted that most of these methods are actually simple accessors

Accessor	Ret. Type	Name	Parameters
public	**String**	`getGameName`	()
public	**GameBoard**	`getBoard`	()
public	**JPanel**	`createCustomPanel`	()
public	**JPanel**	`getCustomPanel`	()
public	**void**	`setCommunicator`	(**Communicator** c)
public	**Communicator**	`getCommunicator`	()
public	**VirtualBoard**	`getVirtualBoard`	()
public	**boolean**	`allowedMove`	(**Move** m)
public	**void**	`receiveMove`	(**VirtualBoard** vb, **MoveMiscellanea** mm)
public	**boolean**	`doMove`	(**Move** m)

Table 4.3 Method listing for the GameController interface.

to important variables, and as such, the implementing class only has to define a handful of truly essential functions.

- `getGameName`. This method simply returns the name of the variant handled by the implementation. It might be used by a tool which lists all the available games incorporated in the interface, or interrogated by a Communicator to decide which implementation to instantiate to deal with a new game notification.

- `getBoard`. This method returns the **GameBoard** object associated with the current game. This class is the focus of a later section and represents the highest level of the JKrieg user interface, actually displaying chessboard data on the screen.

- `createCustomPanel`. This method creates, sets up and returns a **JPanel** object. JPanels are part of the Java Swing package, and they serve to group related interface elements in a window. With this method, the Controller builds an additional panel containing an arbitrary of interface elements pertaining to the game being played. Typically, this panel will be automatically attached to the overlying GameBoard and displayed to the right side of the chessboard. The GameController will be responsible for updating and mantaining the panels created with this method.

- `getCustomPanel`. This method just returns the JPanel created with the aforementioned `createCustomPanel` function.

- `setCommunicator`. This method sets the Communicator to be used for outgoing move notifications.

- `getCommunicator`. This method returns the Communicator currently used for outgoing move notifications.

- `getVirtualBoard`. This method returns the **VirtualBoard** object associated with the current game. As mentioned earlier in this chapter, these object keep the status of piece positions at any given time (as known to the user).

- `allowedMove`. This method returns **false** if the move being attempted by the player is guaranteed to be illegal, **true** otherwise. It is perfectly possible to have this method always return **true**, though some checking would avoid interrogating the remote endpoint in vain when the move is clearly illegal (for example, caused by a slipping hand). It should be noted that, in the case of Kriegspiel, the inherently incomplete nature of the game makes the problem impossible to solve, and moves declared as legal by the program may be denied by the server.

- `receiveMove`. The main method in the GameController interface, it accepts a **VirtualBoard** object describing the new state of the chessboard, plus a list of additional information passed via a **MoveMiscellanea** object. The code has to react appropriately and update the interface, both the graphical board and the attached JPanel.

- `doMove`. This method, usually invoked by the **GameBoard** that the user interacts with, orders the program to execute a move. This usually involves formatting the move itself and sending it down to the Communicator for actual delivery.

4.3.1 The KriegspielController class

The **KriegspielController** class is one of the two GameController implementations bundled with JKrieg, the other being **OrthodoxChessController**. The latter, as the name suggests, allows to play regular games of chess, but it is little more than a simple move wrapper, and because of this it was chosen not to document it here in the present paper. KriegspielController, on the other hand, is a complex class, easily the largest and trickiest file in the JKrieg directory. Kriegspiel is not a difficult game to program *per se*; in fact, the Controller coded to handle orthodox chess would have been able to play games of Kriegspiel as well, and with only minor changes. Of course, this would have been accomplished by simply copying and pasting the umpire's messages from the lower Communicator level into a textbox or a similar interface element – which is how many existing programs allow to play Kriegspiel at the time of this writing. But the aims for JKrieg were much higher than this; the application was to help the player by integrating his or her knowledge and the incoming messages in a seamless way, hence the theoretical work developed and discussed throughout **Chapter 3**.

The `KriegspielController` class witnesses to the potential flexibility of the three-layer architecture chosen for JKrieg. It literally shapes up the overlying generic visual chessboard through a very limited set of methods, and yet the changes are self-contained within a single class. Of course, the whole JKrieg package was created with Kriegspiel in mind, since this was the original goal for this application; however, the only places where Kriegspiel is mentioned are here and in the **ICCDriver** class, wherein the program formats incoming umpire messages into a MoveMiscellanea object so that they can be understood by the present class.

The first problem that was faced during the development of this class involved creating an internal representation format for information of probabilistic nature. In other words, it was necessary to encode the entities introduced in Chapter 3 as **guess tokens** into the class. This implied that a matrix of values, corresponding to the 64 cases on the chessboard, should be created and mantained by the program. These values **keep track of which tokens are where, and their relative intensity** (as JKrieg makes use of transparency to convey the probability of a guess being correct). Thus, in addition to the VirtualBoard the controller is expected to mantain, the class also makes use of additional support variables which are updated as move notifications are received by the Communicator below, and in turn the actual chessboard will be ordered to update what is shown to the user.

It follows from the reasons above that the `receiveMove` method is where most of the action takes place. The method has to take a VirtualBoard and a MoveMiscellanea object in order to respect the interface contract, though the former is a completely superfluous parameter, as it will only contain the allied pieces (or show that one is missing if a capture has happened, which could be learnt from the umpire messages as well). The actual source of information is the MoveMiscellanea, storing incoming messages in a well-known protocol. Of course, in order to provide Kriegspiel functionality (at least with the author's implementation of it), the Communicator has to encode the information as KriegspielController expects it to be, that is, using the same tags and associating values correctly. However, as each game is based on a very limited set of rules, the protocols are invariably very simple to code and keep uniform.

Table 4.4 shows all the legal tags that the KriegspielController class understands. Passing different tags will not cause errors, but they will be ignored by the program.

These tags summarize all the information needed to correctly play the Kriegspiel ruleset as enforced on the ICC. As it is easily seen, there are two redundant tags, **Check** and **Double Check**; their function is to allow the program to perform common routines, independent of the actual check type, or warn about the presence of two simultaneous checks, which changes the mode the algorithm works in.

What the `receiveMove` method does is scan the MoveMiscellanea for the various tags by using a very simple `for` cycle, reacting to each entry in the dictionary as they are parsed. The knowledge base and guesses are updated accordingly, and finally the VirtualBoard status is changed to reflect the new computations. Of course, this is easier said than done, and the class itself features over 1,200 lines of code. Out of these, the `receiveMove` method, which dispatches the various tags to the appropriate places and deals with particular cases such as illegal moves, is about 160 lines long. It should be noted that over 60% of the code making up the class is devoted to handling

Tag	Value
"White Clock"	A String such as "0:32"
"Black Clock"	A String such as "0:32"
"Outcome"	A String ("1-0", "0-1" or "1/2-1/2")
"Pawn Tries"	An Integer object (number of tries)
"Piece Captured"	Ignored (location is known)
"Pawn Captured"	Ignored (location is known)
"Check"	Ignored (must be sent with a check type tag)
"Double Check"	Ignored (must be sent with check type tags)
"Rank check"	Ignored
"File check"	Ignored
"Long diagonal check"	Ignored
"Short diagonal check"	Ignored
"Knight check"	Ignored

Table 4.4 Listing of the MoveMiscellanea tags understood by KriegspielController.

checks (that is, the Check and King tokens), as these are by far the most complex situations.

Each check type is treated separately through four methods, respectively called `handleRankCheck`, `handleFileCheck`, `handleDiagonalCheck` and `handleKnightCheck`, each tackling a different type. Inside these methods, after an introductory part of variable initialization, the code is split up into two sections by an **if** statement, separating the opponent's checks from the player's. This is performed here because there is a notable number of shared calculations between the two cases, and this choice avoids hundreds of lines of redundant code.

4.4 The GameBoard Level

The **GameBoard** class, which extends the standard **JFrame** class coming with the Swing package, provides the actual graphical representation of a chessboard with which the user can interact. Unlike the underlying levels, consisting of interfaces, GameBoard is a full-fledged class; this choice was motivated by the fact that the GameBoard is not meant to be replaced by different implementations. In fact, one of the primary aims of an interface is provide the user with a consistent, familiar visual pattern, thus suggesting that the same basic chessboard should be used, no matter what the Controller or the Communicator objects are. However, it is still possible to take advantage of Java's inheritance mechanism, and generate a specific child class, should it be necessary to the application.

When it came to deciding what the board would look like, a traditional, conservative approach was chosen, involving a classic two-dimensional grid. The user would expect the interface to adhere to a consolidated standard like drag'n'drop, a feature fully supported by Swing objects (see [5]). While drag'n'drop would have been possible with a three-dimensional board, such a solution would have proven both harder technically and potentially less usable and intuitive, especially to handicap bringers. Finally, a three-dimensional view may make sense in regard to orthodox chess, but variants like Kriegspiel like 2D images more, because transparency and special effects can be effectively used to provide more complete information.

Table 4.5 lists the more significant methods making up this class, excluding accessors.

- `GameBoard`. This is the only constructor specified for the class. It accepts a number of parameters, including GameSpecs and GameID indicating information concerning the new game (mostly used for setting the window title to show the players' names and the game number, if applicable). It also requires a GameController implementation to be passed to it, so that move notifications can be delivered to it. During the initialization stage, the GameController is also asked to build and return its **custom panel**, so that it can be attached to the window and displayed to the right side of the chessboard. The remaining parameters are useful to let the program know if the player is using the Black pieces (in that case, the board will not activate drag'n'drop interaction until the next move).

- `updateBoard`. This method tells the object to update the status of all the tiles on the board, according to an internal VirtualBoard object, typically immediately after changing it through a `setBoard` call.

- `setBoard`. This method is usually invoked by the GameController to change the internal representation of the chessboard after a new move has been received by the Communicator. This does not immediately alter the display to reflect the changes; the `updateBoard` has to be called for this to happen.

- `getTile`. This method is little more than an accessor, returning a GameTile object corresponding to the rank and file coordinates that it takes as input. This method was added to the class because it is sometimes necessary for the Controller to perform direct, custom action upon a specific tile of the chessboard. For example, the Zone of Control feature discussed in Section 3.6 requires the GameController to invoke a method to set the color of a GameTile to red.

- `handleMove`. This method reacts to a drag'n'drop operation from the user. Its implementation first asks the GameController for legality of the attempted move, and in the case of an affirmative response (which may also mean that the answer is unknown), it dispatches the move to the GameController itself by invoking its `doMove` method.

In addition, the GameBoard overrides a method of its parent class JFrame, `processWindowEvent`. This method reacts to different windows event, such as the open window being activated or deactivated (when the user clicks on another window, for example), resized or closed. The new implementation invokes the overriden method, but adds a supplemental behavior: if the user closes the window, it warns the lower levels that, if a game is in progress, it should be aborted. This way the remote endpoint will be warned that the user has left the game, and the player will be able to start a new game if they so desire.

4.4.1 The GameTile Class

A GameTile object represents a single cell of the chessboard. This class inherits basic behaviors from the Swing **JComponent** class, thus making a GameTile conceptually *on par* with any other interface element, such as checkboxes, text fields or buttons. Every object contains all the information

Accessor	Ret. Type	Name	Parameters
public	**GameBoard**	`GameBoard`	(**GameController** gc, **GameID** id, **GameSpecs** specs, **String** plName, **boolean** interact, **boolean** white)
public	**void**	`updateBoard`	()
public	**void**	`setBoard`	(**VirtualBoard** vb)
public	**GameTile**	`getTile`	(**int** file, **int** rank)
public	**boolean**	`handleMove`	(**int** fromRank, **int** fromFile, **int** toRank, **int** toFile)

Table 4.5 Listing of the most important methods in the GameBoard class.

it needs to operate in stand-alone mode, including square color, pieces, and special images. The main advantage of this method is that drag'n'drop nearly came to Krieg for free, as the behavior is embedded into JComponents. As an added benefit, a small twist in the two "for" loops that arrange their placement is everything the program needs to swap the board when the user plays the black pieces (the Components are just placed at different coordinates).

The most noteworthy feature of a GameTile consists of supporting seamless drag'n'drop from other GameTiles. This was accomplished by having the class implement a number of interfaces, namely **DragGestureListener**, **DragSourceListener**, **DropTargetListener** and **Transferable**. The main reason for having to implement four interfaces lies in the fact that a GameTile can be both a source and a target of a drag'n'drop action, since pieces are dragged across the whole chessboard.

The main method in the DragGestureListener interface is `dragGestureRecognized`. This method is invoked with a **DragGestureEvent** object as a parameter. The implementation has to call the `startDrag` method on that object if it deems the drag operation legal; in that case, it has to specify the cursor icon and drag'n'drop avatar image for the current drag. The latter is a simply the image of the piece being moved, whereas the cursor icon will not change. This being said, the method makes sure that the piece being dragged belongs to the player and it is the player's turn to move before authorizing the drag. `startDrag` also requires that the information to be exchanged via the drag be passed as a Transferable parameter. This is why GameTile also implements the Transferable interface, and it passes a reference to itself as drag information.

Conforming to the Transferable interface means that the object can release its information in a well-defined set of **data flavors**. Examples of data flavors include plain text, RTF formatted text, HTML, or user-defined formats. In this case, GameTile only supports a single data flavor, "Tile". A Transferable transmits the information it contains through the `getTransferData` method, which takes the desired data flavor as a parameter and returns the data in the specified flavor as an Object, if applicable. For example, a GameTile, when asked to transfer its data in the "Tile" flavor, will return a **Point** object containing its coordinates on the chessboard.

The DragSourceListener interface requires the class to define five methods. Although GameTile implements them as dummy functions that really do nothing, these serve as placeholders, should one wish to incorporate additional visual feedback to the drag – for example, by highlighting a legal target square when the dragged piece passes over it.

The DropTargetListener represents the other side of the drag, that is, the drop. The most important method here is `drop`. The implementation first checks if the dragged item is applicable (that is, dragging text onto a chessboard will not yield any result) by making sure that the incoming Transferable supports the "Tile" data flavor. Then, it asks it to transfer its "Tile" data, and decodes the Point object, obtaining the coordinates of the location from which the drag originated. Since the GameTile knows its own location on the chessboard, the program now knows both source and target of the attempted move, and the GameBoard can invoke the `handleMove` method and have the Controller deal with it.

4.4.2 Images & Transparency

Custom imaging is another integrated feature of the GameTile class, and, while specifically designed for Kriegspiel, it is potentially utilizable by several other variants. Basically, the GameBoard loads picture data at startup, but it is possible to specify different images to be displayed on the GameTile through the `addCustomImage` method. This method comes in two flavors: a simpler one that only takes an **Image** object and centers it in the component's area, and a more complex one that also requires a **char** parameter representing the desired transparency for the image.

Transparency is achieved through the **FilteredImageSource** class, which allows a programmer to create **filters** affecting an Image object. There are various types of filters, and the transparency filter needed for this method (which achieved guess token transparency in Kriegspiel, thus associating opacity to probability) belongs to the simplest type; that is, where each filtered pixel only depends on the corresponding source pixel.

This being said, the **ImageTransparencyFilter** class, extending the more general **RGBImageFilter** class, was coded to solve the problem. The class implements a constructor, taking the desired transparency rate as parameter, as well as a single method, `filterRGB`. Given two coordinates and the color value of the source pixel, expressed as an **int**, this method is expected to return the filtered value. Changing transparency translates to nothing more than changing the pixel's **alpha value** to the desired quantity, expressed in 255ths.

Chapter 5

Tests & Conclusions

The program was developed in such a way as to allow the programmer to perform test runs as soon as possible. Actually, the first test took place a few days after the writing of the first line of code: since the development process started with the coding of the Communicator layer and its ICCDriver implementation, the first tests were executed under the text-based console bundled with this class, and this process discovered a number of communication bugs having to do with the intricacies of the Internet Chess Club protocol.

At a later stage, it was possible to employ the graphical interface itself to play test games. The OrthodoxChessController class, while little more than a dummy class for playing regular chess, was nonetheless an important tool in that it allowed to make sure that the general implant of the program and the visual interface was correct, before descending into the complexity of the specific Kriegspiel features. Here, the main testing tool constisted of either running two copies of the program on the same machine, and play a game between them, or running a copy of JKrieg together with a different interface, in order to find possible issues deriving from having two programs interact with each other, even though with the ICC server in the middle. This being said, the above stage did not reveal excessively troublesome bugs and errors to correct, although the ICC human-readable protocol for representing a chessboard required a good deal of effort to translate correctly.

5.1 Kriegspiel Tests

Testing the functionality of the KriegspielController class, however, raised many more hidden problems. The first tests of this series still involved self-playing using two running copies of the program, and brought out a number of bugs, such as misinterpretation of some special cases, wrongly initialized variables, and divisions by zero when attempting to calculate correct opacity for some guess tokens. These problems were solved separately, but a fact that was not realized until later in the development process was that **the testing procedure itself was flawed**. In other words, playing simulated games against the programmer himself only checked the correctness of those features that the programmer thought about checking. The tests focused on the same game patterns, trying to reproduce a number of situations, but failing to capture them all.

The mistake manifested itself in all of its seriousness when the second series of tests utilized an artificial Kriegspiel player residing on the Internet Chess Club, **Krieg**.

5.1.1 Krieg

Krieg is a computer player of Kriegspiel over the Internet Chess Club. It was mentioned in Chapter 2 that, because information is incomplete in Kriegspiel, in constrast with the most powerful AI agents liking complete information in order to use brute-force algorhythms and compute as many moves as possible, there are not very strong artificial Kriegspiel players. However, Krieg is strong enough, and it becomes much stronger, to the point of utter invincibility, as time constraints get tighter (for example, 3-minute games without increment).

Of course, there is more to Krieg than meets the eye. To put it simply, Krieg cheats. Being a registered user of the Internet Chess Club, the program has access to a number of restricted features, including watching rated games as a spectator. The trick is as simple as it is effective: during the game, Krieg also registers itself as a watcher of its own game, and therefore can see the complete information that only the umpire is normally entitled to know. This means that Krieg knows where its opponent's pieces are located and, in fact, Krieg is just another chess program with a module on top of it that converts Kriegspiel into regular chess. Just another chess program, and a weak one as well, since a human player can beat it anyway when time is less of a problem.

No matter its strenght or weakness, Krieg is a very useful testing tools. It was by playing with it that the developer found out that the theory behind the guess tokens was missing a few crucial points. A mistake had been committed earlier in the development that would be paid for dearly later on. For one, the programmer had so far been following the wrong assumption that if both a capture and a check happen at the same time, then the capturing piece is the one causing the check. As shown in **Figure 3.4** on page 15, this is dramatically wrong, as it fails to take into account 'discovery checks', but the programmer stuck to it until Krieg proved it wrong in a real-world game.

This also proved that a more thorough theoretical work should have been carried out at the beginning of the design step. What is now shown in Chapter 3 is the result of a first effort, then corrected and integrated with the new information drawn from the testing stage on the field.

5.2 Conclusions

JKrieg was the developer's first attempt at creating a serious, real-world application following a set of external specifications, and as such resulted in a mixture of success and failure. Most of the mistakes could have been easily avoided and derived from his lack of experience and proper software engineering theory. However, despite the problems of a troubled creative process, JKrieg is a working application that meets its requirements, and even more importantly, it can be upgraded and extended to provide further functionality. For one, the Kriegspiel controller could be enhanced to reveal even more information than it currently shows; right now, all the calculations are performed basing on the last chessboard status only. But it would certainly be possible to improve the effectiveness of JKrieg **by taking into account the previous states**, so as to rule out possible piece positions.

Another area of enhancement might lie in how the probability of guess tokens is actually computed. As it stands right now, all King and Check tokens are always displayed with the same transparency level, which is not always realistic. Algorithms could be devised to recognize and promote more likely areas, and demote places that are less easily reachable by enemy pieces.

Appendix A

The JKrieg Postmortem

Introduction

This appendix contains the **postmortem** document for the JKrieg application. A postmortem is an article describing the overall development process of a software for future reference and information, as well as to summarize the lessons learnt by the developers during the various stages of the work. It is usually followed by a brief data sheet containing vital details about the project, the developing team, the platforms it works on and notable technologies employed.

Normally, the introduction part of a postmortem describes the infant stages of the life of a project – when and why it was designed, its intended target and most relevant competitors, how the developer team was assembled and what hierarchies and structures were used to coordinate the work. JKrieg is not a commercial project but an academic one and was born on the effort of a single developer from April to October 2003. Its aim was to create the first truly specific Kriegspiel user interface, for play on the Internet Chess Club.

What went right

The choice of the Java programming language was most certainly the best decision made during the development process. Not only did Java provide the object-oriented environment that allowed JKrieg to be portable, encapsulated and extensible, it also offered a number of vital functions "out of the box" through the Swing package and other fundamental classes included in the basic release. Features such as drag'n'drop, transparency, sockets, timers and textboxes nearly came for free and required only minor customization work to adjust them to fit the application's needs. In the long run, this saved a lot of work and made it possible to satisfy the software requirements within the expected deadline. The three-layer architecture also made the task simpler and allowed the developer to handle problems separately, thus minimizing the complexity of a program spanning over about 30 classes and 7,000 lines of code.

The idea of using tokens to help a player better visualize information on the chessboard was a successful one. These entities are intuitive enough to be useful, and yet not intrusive enough to be a distraction to the player. One of the most important merits of this system is that the player does not have to shift his or her eyes from the chessboard to take a look at the incoming messages from the umpire; this way, the attention span (which can be shortened and weakened by such simple actions as scrolling a window on the screen) is better preserved and this may result in the player being more attentive and concentrated, especially during fast-paced games on the Internet.

What went wrong

The theoretical work about guess tokens (described in Chapter 3) was performed at different stages of the development cycle, and this proved to be the single most significant pitfall. As the intricacies of the situations possibly deriving from the tokens were not entirely considered during the design stage, the complexity of the resulting system was severely underestimated until later in the implementation stage, when the first series of tests reported the presence of conceptual flaws in the algorythms that placed the tokens on the screen. These algorhythms sometimes made mistakes basing on wrong (and even naively so) assumptions, This basically forced the developer to rewrite a few sections of code from scratch, wasting valuable time resources, and only the encapsulation praised in the section above allowed to save the day and limit the losses down to an acceptable degree.

This was a harsh lesson learnt from the project, and stressed the absolute importance of a thorough, even paranoid design stage, as the cost of correcting a mistake increases **exponentially** with every step down the path of development.

Data Sheet

Project name JKrieg

Timeline April to October 2003, or 3 months-man

License GNU Public License

Platforms Any platform running the Java2 Virtual Machine (1.3.1 or later)

Project size 27 class files, 7487 lines of code, approx. 250,000 characters

Technologies Java 2, most importantly the Swing graphics package

JKrieg File Listing

```
ClientConsole.java
Communicator.java
GameBoard.java
GameController.java
GameHistory.java
GameID.java
GameMove.java
GameSpecs.java
GameTile.java
GameTileRowIndicator.java
HistoryPanel.java
ICCDriver.java
ICCGameID.java
ICCLoginDialog.java
ImageTransparencyFilter.java
Krieg.java
KriegspielController.java
LoginDelegate.java
Move.java
```

```
MoveMiscellanea.java
OrthodoxChessController.java
PGNViewerController.java
PiecePosition.java
RegexpChecker.java
SoundManager.java
TimerComponent.java
VirtualBoard.java
```

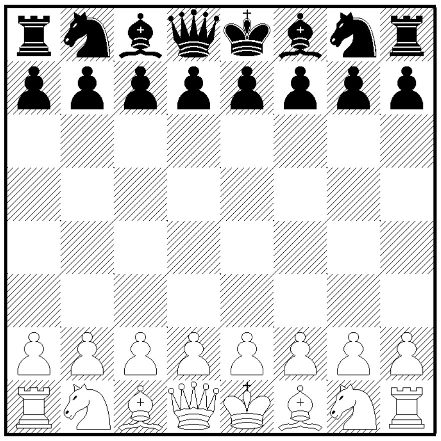
Appendix B

Chess Diagrams

This paper was written in the LaTeX typesetting language, and the chess diagrams featured throughout it were obtained by utilizing a custom combination of two specialist packages, **chess** and **skak**. These packages allow to easily create diagrams from several well-known notations, and even keep track of the progress of a game, move by move, making it possible to print the current state of the chessboard at any given time. All in all, they are excellent choices for typesetting chess games, with skak, the younger package of the two, offering a wider range of options and a more intuitive set of functions and macros.

However, the chess diagrams employed in this paper were much more challenging to make than a standard game diagram, for obvious reasons such as the presence of non-existant symbols like the Guess Tokens and a plethora of more or less strange additions, including colored pieces, arrows and letters. Producing diagrams with such a level of customization would have been nearly impossible, if not for the inherent power of the TeX language. In fact, in order to accomplish these results, the author had to tamper around with macro files and find more than a few tricky solutions to seemingly unsolvable problems.

The key concept behind the diagrams is that each tile of the chessboard grid is just another **glyph** of a font. In other words, the White King on a black field is not any different from the letter 'a' in the rest of this document; it just belongs to another, specialist font. This font is, obviously, a **monospace** one, that is, all glyphs are equally wide, so that they can be perfectly aligned in columns. Looking at it this way, the following chessboard:



was obtained by means of the following command:

```
\myshowboard
 {rmblkans}
 {opopopop}
 {0Z0Z0Z0Z}
 {Z0Z0Z0Z0}
 {0Z0Z0Z0Z}
 {Z0Z0Z0Z0}
 {POPOPOPO}
 {SNAQJBMR}
```

where `myshowboard` is a user-defined macro which changes the font to the skak one and performs other tasks such as drawing borders around the chessboard. Each character is mapped to the corresponding image just like in any other font, White pieces being uppercase, and different letter being used to separate white fields from black fields (for example, the White King on a white field is represented by the letter K, and by the letter J when it is on a black field).

However, this alone does not explain how it is possible to heavily customize a board by adding characters not included in the font definition. This is accomplished through the use of an incredibly versatile LaTeX command, the `picture` environment. An environment is basically a group of statements framed by a pair of `begin` and `end` commands, and the picture environment is used to produce pictures of any size, starting from elementary building blocks: characters, lines, circles, and so on. The best thing about pictures produced this way is that they can be put nearly anywhere on a page, even in a line of text (for example, they can be used to draw a new character accent not existing in the standard release); which means that they can even replace an arbitrary number of squares on the chessboard. This technique, as well as several others, is introduced and detailedly exposed in [4].

Thus, what was done was to insert custom-made pictures in place of skak characters whenever something unusual had to be displayed. For instance, six macros were created to display the three guess tokens on both white and black backgrounds, and invoked whenever the need arose. The integration with the rest of the chessboard is seamless so long as the picture is declared as having the same size as the other characters (that is, 20x20 points). The Piece token is a mere circle, the Check token is the Piece one with a '+' char in Roman font in the middle, and King token adds a 15-point version of the King glyph from the skak font, so that it can fit in the circle.

Adding color was perhaps the hardest, and yet most intriguing undertaking. While glyphs can be easily colored via the `textcolor` command (remembering that boards are nothing more than text displayed using an uncommon font), drawing the red background of **Figure 3.10** on page 22 proved to be an uphill battle, as the command of choice for this operation, `colorbox` is affected by a documented bug, making the colored box a few points too large (a problem which has to do with the package 'patching' a driver born to work in black & white mode), which is unacceptable in a case such as this, where size is of the utmost importance, and the overgrown box would surpass the required 20 points, thus laying waste to the overall setup.

The solution was quick and in a way lacked elegance, but it got the job done with the required accuracy and a single line of code. The `multiput` command, when used inside the `picture` environment, allows to repeat another shape over and over a given number of times, each time translating it by a fixed amount of space. This being said, it sufficed to use this command to clone a single, vertical red line running from bottom to top, and have it repeated enough times to fill up the area. It should be noted that, in a larger paper with more occurrences of this pattern, it would be possible to store this command in a **save box** so as to prevent it from being processed each time.

Bibliography

[1] Paolo Ciancarini. *I Giocatori Artificiali.* Mursia, 1992.

[2] Paolo Ciancarini. *La Scacchiera Invisibile: Introduzione al Kriegspiel.* To be published, 2003.

[3] Marco Collareda. Scacchi e computer. programmi, algoritmi, interfacce grafiche e internet chess server, 2002.

[4] Antoni Diller. *LaTeX Line By Line.* John Wiley & Sons, 1993.

[5] Sun Microsystem Inc. *Java Look and Feel Design Guidelines.* Available for download at http://java.sun.com/products/jlf/ed2/book/, second edition, 2001.