Conquer medieval kingdoms with CodeRuler

Stretch your Java programming skills with this new animated graphical simulator

Level: Introductory

Sing Li (westmakaha@yahoo.com)
Author, Wrox Press
29 Jun 2004

Guard your castle! Claim your land! Command your knights to joust valiantly and defeat their foes. Capture the enemy's position and seize its land while dodging its menacing knights. If writing mudane Java code is giving you the blues lately, maybe it's time to turn your medieval fantasies into reality. You can rule your own kingdom while refining your Java programming skills and mastering the Eclipse development environment all at the same time. It's all in a hard day's work for a supreme CodeRuler. Simulation-gaming enthusiast Sing Li puts you on the fast track to ultimate kingdom domination.

Born of the 2004 ACM International Collegiate Programming Competition (see Resources), CodeRuler is IBM alphaWorks' newest fantasy gaming simulator challenge. The game has a simple premise: You are the imperial ruler of your very own medieval kingdom. Your peasants and knights depend on your brilliant strategic thinking, agile adaptability, and superior Java programming skill to survive, increase, and prosper. Your objective as a player is to write Java code that simulates this ruler. The gaming simulator pits your ruler against up to six opponents' rulers (or the included sample rulers) and determines the winner.

This article guides you along the shortest path to ruling your own medieval kingdom. It reveals the game's environment, describes the rules, discusses general strategies, and provides two complete working ruler entries that you can put to use (or modify) immediately.

The simulation environment

CodeRuler is a graphical, animated simulation gaming environment. As a medieval ruler, you must battle other rulers for land and dominance. Your kingdom consists of:

- Peasants who can claim and work the land.
- Knights who can fight battles and capture other rulers' peasants, knights, or castles.
- A castle that can create more knights and peasants. The more land you have, the faster it creates them.

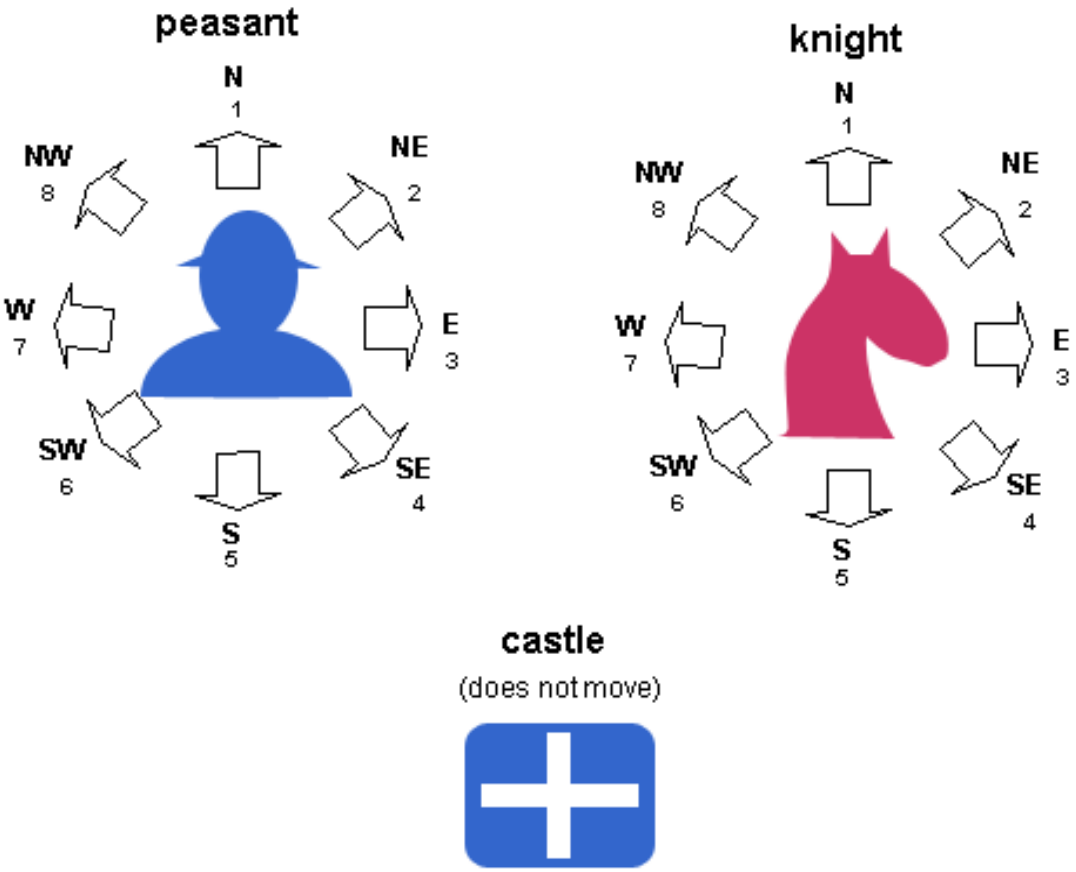
The graphical gaming world

The game is played out in a two-dimensional world represented by a map of the kingdom. (The background landscape sketch merely acts as wallpaper; it doesn't affect game play or change as the game progresses.) Figure 1 illustrates a CodeRuler game in progress.

Figure 1. CodeRuler in action

Figure 1 shows two competing rulers at work. The ruler -- the mastermind behind the strategic movements of the game pieces -- doesn't appear in the game world. The game pieces (peasants, knights, and castles) are the colored dots moving within the simulated world. Figure 2 illustrates the pieces' shapes and their possible movement directions.

Figure 2. Movement pattern for CodeRuler game pieces

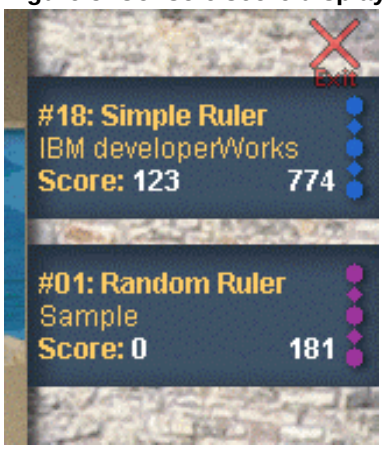


As you can see from Figure 2, knights and peasants use the same movement pattern. They can move a single square in any one of the eight directions for each turn. Each direction has an associated number, which you use in Java coding. Each number also has a predefined constant (such as NW) that you use in your code.

The console score display

You can see the status console on the right side of Figure 1. The names of the currently playing rulers, and the organizations to which they belong, appear at the top of the console. The two numbers are the ruler's current score (left) and the number of land squares that the peasants have claimed. Figure 3 shows an example score display.

Figure 3. Console score display

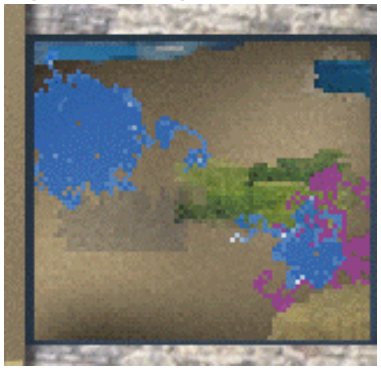


In Figure 3, ruler number #18 is called Simple Ruler from IBM developerWorks. The ruler's current score is 123, and the ruler has captured 774 squares of land for the kingdom. You can abort the simulation at any time by clicking on the red X on the top right.

At-a-glance land occupation display

You can see a small version of the world in the middle of the status console of Figure 1. The image, reproduced in Figure 4, lets you see each ruler's current occupation of land at a glance. You can easily see in Figure 4 that the blue-colored ruler has claimed significantly more land than the magenta-colored ruler.

Figure 4. At-a-glance land occupation display



The simulation clock

At the bottom of the status console in Figure 1 is a clock. Figure 5 shows a closeup.

Figure 5. The CodeRuler clock

A sun travels around the clock's dial. The match is over after the sun has traveled one complete cycle. Each tick of the clock is a turn for the simulator. As a ruler, you determine the moves your pieces make during each turn.

Rules of combat

Each ruler has initial control over:

- 10 peasants
- 10 knights
- 1 castle

During the course of the game, you want to:

- Use your peasants to claim as much land as possible (and keep it claimed).
- Use your knights to capture as many of your opponent's peasants as possible, stopping them from claiming land.
- Use your knights to combat and capture as many of your opponent's knights as possible. This weakens your opponent's defensive capability.
- Use your knights to attempt capture of the other ruler's castle. Castles are factories for knights and peasants; you cannot create more peasants or knights without them. With multiple castles, you gain the ability to create peasants or knights at a faster rate than your opponents. See the sidebar Creating new peasants and castles, for creation rates.

- Strategically prevent the capture of your castle.

Capturing game pieces

Only a knight can capture the opponent's peasants, castles, or knights. It can capture peasants and castles simply by moving into their squares. To capture an opposing knight, you must first bring its strength value down to 0. Each knight starts with a strength value of 100 and loses a random strength value between 15 and 30 for each attempted capture by an opponent. The knight performing a successful capture gains 20 strength units.

The scoring scheme

To win a match, you must be the ruler who has the highest score at the end of the match. Note that the winner might or might not be the ruler with the most land claimed. Table 1 provides the game's scoring scheme.

Creating new peasants and castles
The rate of creation of peasants or knights by a castle depends on the number of land squares that you own:

Land you own	Number of turns to create one peasant or knight
124 or fewer	No creation
125	14
250	12
500	10
1,000	8
2,000	6
More than 4,000	4

Table 1. Scoring scheme for captures

If you...	You get
Capture a peasant	4 points
Capture a knight	6 points
Capture a castle	15 points

At the end of the match, your remaining pieces, captured castles, and claimed land all add to your score, as shown in Table 2.

Table 2. Scoring scheme for remaining pieces

Piece remaining	Score
Peasant	1 point
Knight	2 points
Castle	25 points
Land	1 point per 10 squares

Game details

Each player writes Java code that simulates a ruler. The gaming simulator matches your ruler against other rulers and determines the winner. In your code, you must orchestrate the movement of your peasants, knights, and castle(s). A set of API provides information on your pieces and those of other competing rulers. Using this API, you can write code that implements offensive, defensive, or even adaptive strategies.

Components of the game

The CodeRuler game requires the Eclipse IDE for writing, debugging, and testing your ruler's code. (See Eclipse: The integrated kingdom development environment, later in this article.)

Meet the mastermind behind CodeRuler
For a glimpse under the hood of the CodeRuler engine and some ideas for advanced strategies, read this revealing behind-the-scenes interview with CodeRuler's creator, Tim deBoer.

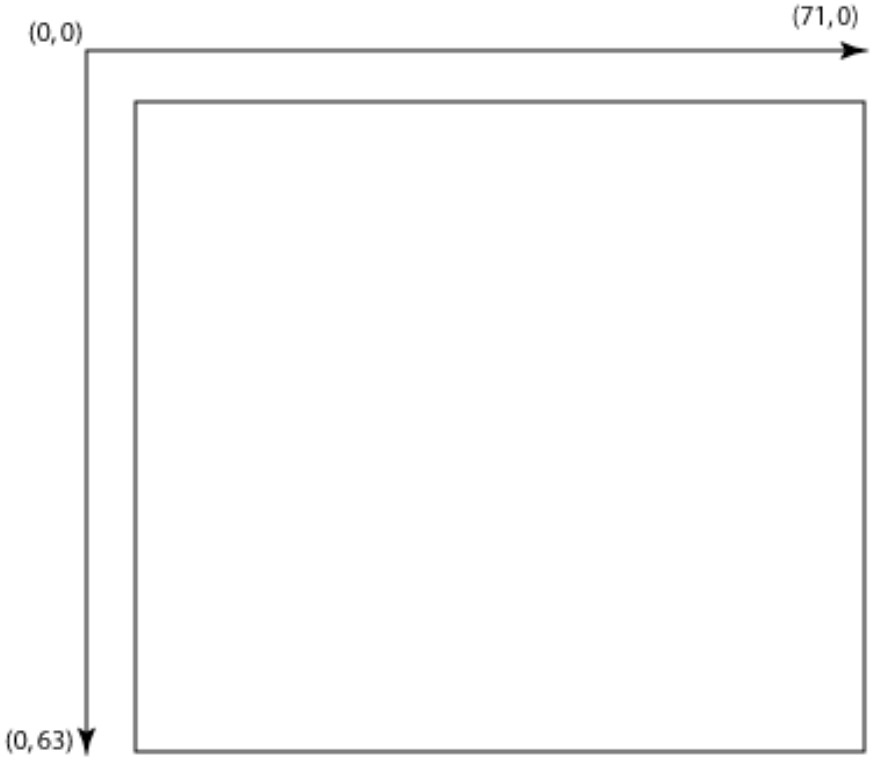
CodeRuler includes:

- An interactive gaming simulator, as a plug-in to the Eclipse IDE.
- The documentation of the API you can use to code your ruler.
- A set of movement, capture, and scoring rules.
- A local arena for running your ruler against a set of sample rulers.
- A networking mechanism for submitting your ruler to compete in public tournaments, or for setting up your own tournaments.

Simulated world coordinate system

The game is staged in a simulated world consisting of 4,608 squares -- 72 squares wide by 64 squares high. The squares are numbered according to an (*x*, *y*) coordinates system. The *x* axis extends from left to right and the *y* axis from top to bottom. Figure 6 shows the layout of the CodeRuler world. Position (0,0) is at the top-left corner.

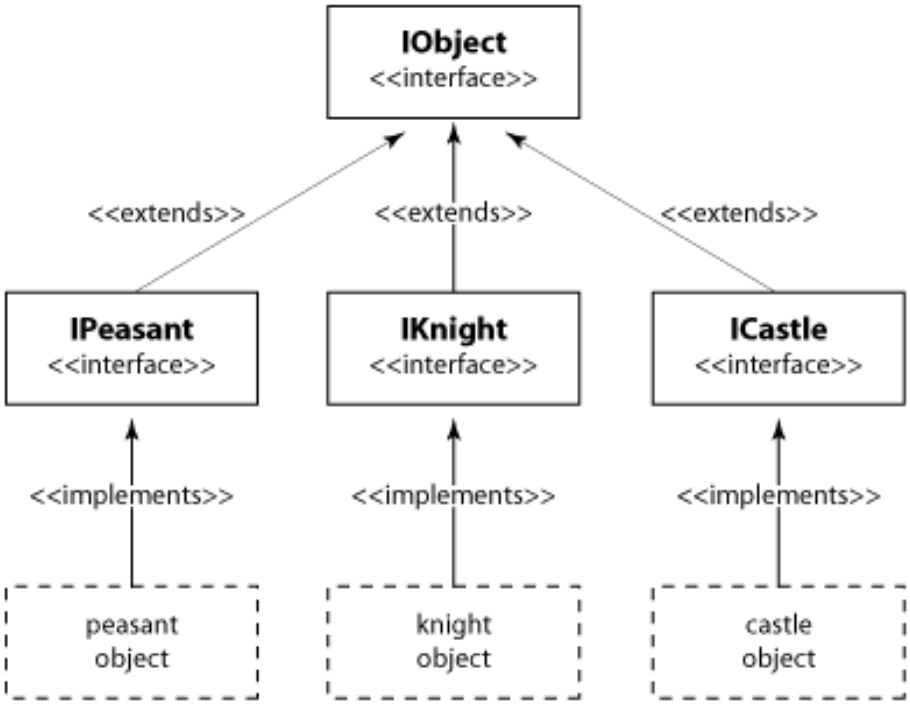
Figure 6. The CodeRuler world coordinates system



CodeRuler API and inheritance hierachy

Before you can command the game pieces, you need to understand the CodeRuler API. The API is highly object oriented and has an explicit inheritance hierachy. An understanding of the hierachy is crucial to effective CodeRuler coding. Figure 7 shows the inheritance hierarchy.

Figure 7. CodeRuler inheritance hierachy



The inheritance tree in Figure 7 is based on Java interfaces. Each game piece must implement its associated interface: peasant must implement `IPeasant`, knights must implement `IKnight`, and so on. However, you never need to code any of these classes, because the CodeRuler simulator uses built-in implementations. As a ruler, you need to use the API provided by the interfaces only to get information about the game pieces.

The IObject interface
The `IObject` interface is the super interface to all pieces on the playing field. Each piece implements `IObject` indirectly. `IObject` factors out the common behaviors of all pieces:

- `getRuler()`: The ruler the piece belongs to
- `getX(), getY()`: The piece's current position
- `isAlive()`: Whether the piece is alive (that is, not captured)
- `getId()`: A unique ID (across all pieces -- knights, peasants, and castles)

The `IObject` super interface also has two convenience methods. These methods can be quite handy in your strategy design and help you avoid the need to employ complex trigonometric math:

- `getDirectionTo()` calculates the closest direction to a specified point on the board.
- `getDistanceTo()` calculates the distance to a specified point on the board.

The IPeasant interface
The `IPeasant` interface adds no new behavior to the `IObject` interface. You can move peasants with the Ruler's `move()` method, which changes their positions. You use a peasant to claim land. A peasant's automatic behavior is to claim any land position that it moves over. An opposing knight can capture a peasant in a single move; no calculation of strength is involved.

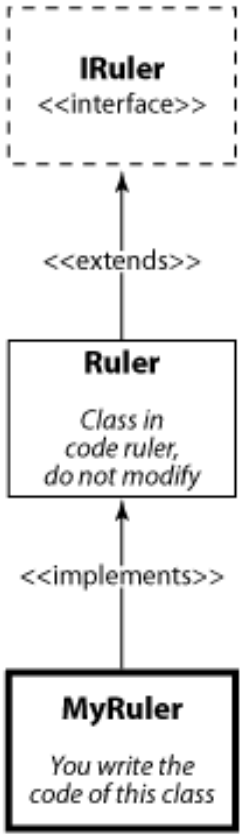
The ICastle interface
The `ICastle` interface, like the `IPeasant` interface, adds no new behavior to the `IObject` interface. A castle's automatic behavior is to create more peasants or knights. The speed of creation depends on the amount of land you own. See the sidebar Creating new peasants and castles, for the creation rate.

The IKnight interface
The `IKnight` interface adds one method, called `getStrength()`, to the `IObject` interface. A knight is captured when its strength is reduced to zero. You can use the `IKnight` interface's `getStrength()` method in your strategy to avoid the loss of knights. See Capturing game pieces earlier in this article for a discussion of a knight's strength calculation.

The interface hierarchy in Figure 7 represents the game pieces that move around the world during the simulation. The ruler, however, is not a game piece and does not move in the simulated world. The `IRuler` interface specifies the ruler's behavior.

The IRuler interface
The `IRuler` interface has no need to -- and does not -- inherit from `IObject`. Figure 8 shows the `IRuler` interface's inheritance hierarchy.

Figure 8. Inheritance hierachy of the IRuler interface



The `IRuler` interface specifies the generic behaviors that all rulers implement. They include informational methods that are useful for the implementation of your strategy:

- `getPeasants()` gets a list of all peasants under this ruler.
- `getKnights()` gets a list of all knights under this ruler.
- `getCastles()` gets a list of all castles under this ruler.
- `getLandCount()` gets a count of all the land squares that this ruler owns.
- `getPoints()` gets the current number of points this ruler has earned.
- `getRulerName()` gets the ruler's name.
- `getSchoolName()` gets the organization name of the ruler's creator.

The Ruler and MyRuler classes
To enforce game-rule-specific behaviors, and to help you implement the `IRuler` interface, CodeRuler supplies the `Ruler` class as shown in Figure 8. This class provides default implementations for most of the `IRuler` methods. You write the content of the `MyRuler` class, which must inherit from the `Ruler` class. You need not, and should never, modify the `Ruler` class.

`Ruler` provides several vital action methods that you should use in your implementation of `MyRuler`:

- `move()` moves pieces around the world.
- `capture()` attempts to capture an opponent's game pieces.

`Ruler` also implements several methods that can change the generation mode of your castle(s). By default, your castle(s) manufacture peasants continuously. However, you use these methods to tell it or them to manufacture knights instead:

- `createKnights()` tells the castle to manufacture knights.
- `createPeasants()` tells the castle to manufacture peasants.

Last but not least, one of the reasons for the existence of the `Ruler` class is to define additional abstract methods that you must implement in your own `MyRuler` class. The simulation engine calls these methods during its execution.

The only code you must write is the code that implements the methods listed in Table 3:

The simulator's workflow
From the perspective of a CodeRuler player, the simulator has the following workflow:

1. Places your initial game pieces at a randomly selected kingdom location in the simulated world
2. Calls your implementation of the `initialize()` method
3. Calls your `orderSubjects()` method on every turn

Your `getRulerName()` and `getSchoolName()` methods should not contain strategy code. The simulator can call them at any time.

Table 3. Methods in all MyRuler implementations

Method	Description
`getSchoolName()`	Returns a string of 25 or fewer characters that identify your group or organization. (CodeRuler was originally designed for competition among colleges.) This will be used during the game to identify your ruler. In Figure 1, for example, Simple Ruler has the school name IBM developerWorks.
`getRulerName()`	Returns a string of 25 or fewer characters that uniquely identify your ruler. For example, one of the rulers in Figure 1 is named Simple Ruler.
`initialize()`	The system calls this method when you first place your ruler into the game. Perform any initialization you need here. You are limited to one second of initialization. The amount of work your machine can perform during initialization will vary with the CPU speed and the Java VM in use, but one second is sufficient for most code-initialization tasks. Do not attempt any work that might depend on slow input/output.
`orderSubjects()`	This is the core of a `CodeRuler`. The system calls this method once every turn. You need to exercise your strategy and tell your pieces what to do within this method.

Eclipse: The integrated kingdom development environment

You need to download and install the Eclipse IDE (version 2.1 or later) in order to run the CodeRuler simulation environment (see Resources). CodeRuler integrates into the Eclipse IDE as a plug-in, thereby leveraging Eclipse's developer-friendly features.

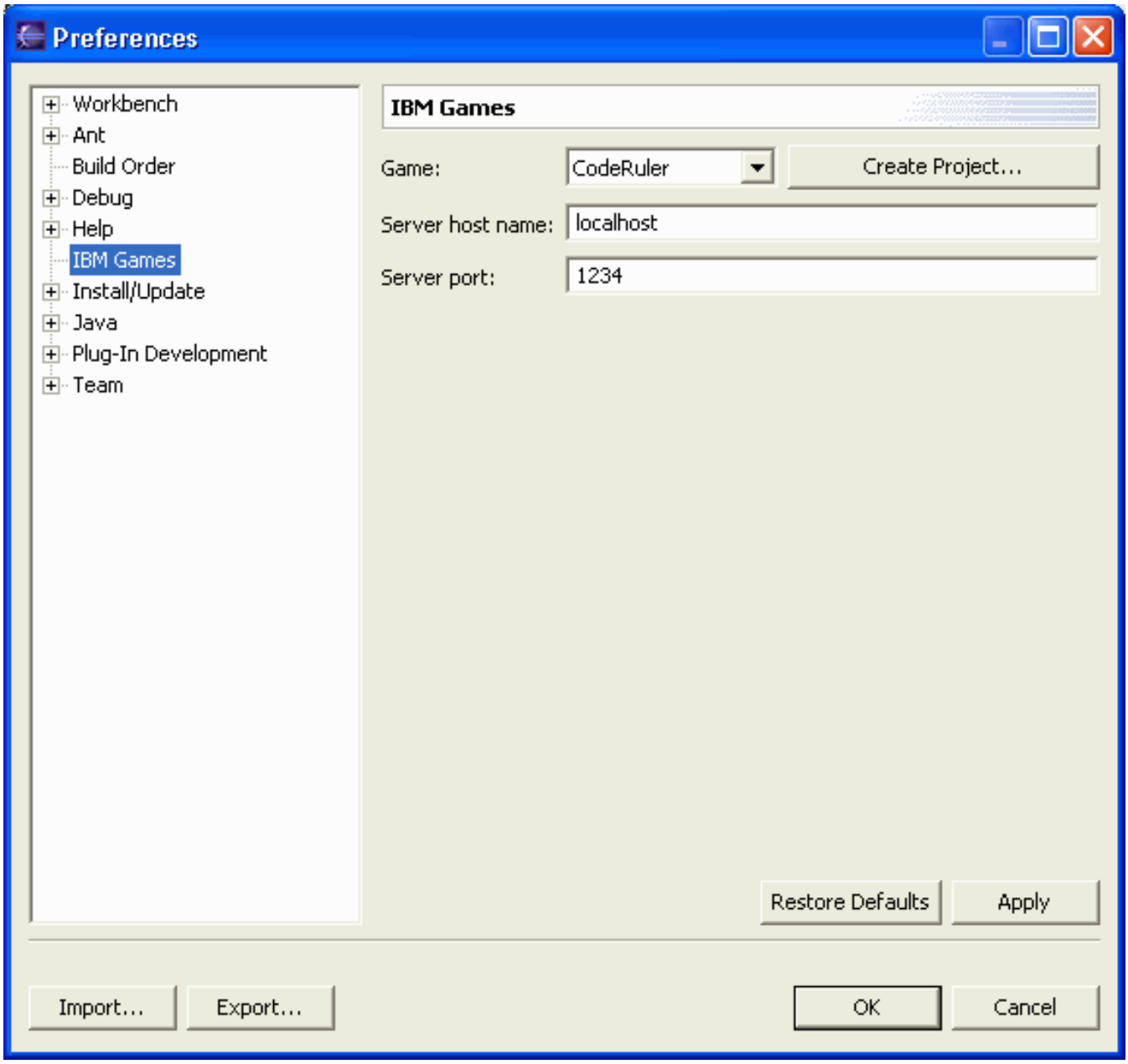
Installing Eclipse and CodeRuler

To install Eclipse, unarchive the distribution into a directory and run the eclipse executable (eclipse on *nix, and eclipse.exe on Win32 systems). You need JDK/JRE 1.4.2 or later installed. (Version 1.4.2 is highly recommended because CodeRuler is developed and tested under this VM version.) After you install Eclipse, download the CodeRuler engine (see Resources). To install CodeRuler, unarchive the CodeRuler distribution into the *<eclipse installation directory>*/plugins directory. This should create a com.ibm.games directory under the plugins directory. Start or restart Eclipse, and the CodeRuler plug-in will load. You're now ready to give CodeRuler a try.

Creating your CodeRuler project

You need to create a new project in Eclipse to use CodeRuler. From the main menu, select Windows|Preferences. A dialog box pops up, as shown in Figure 9.

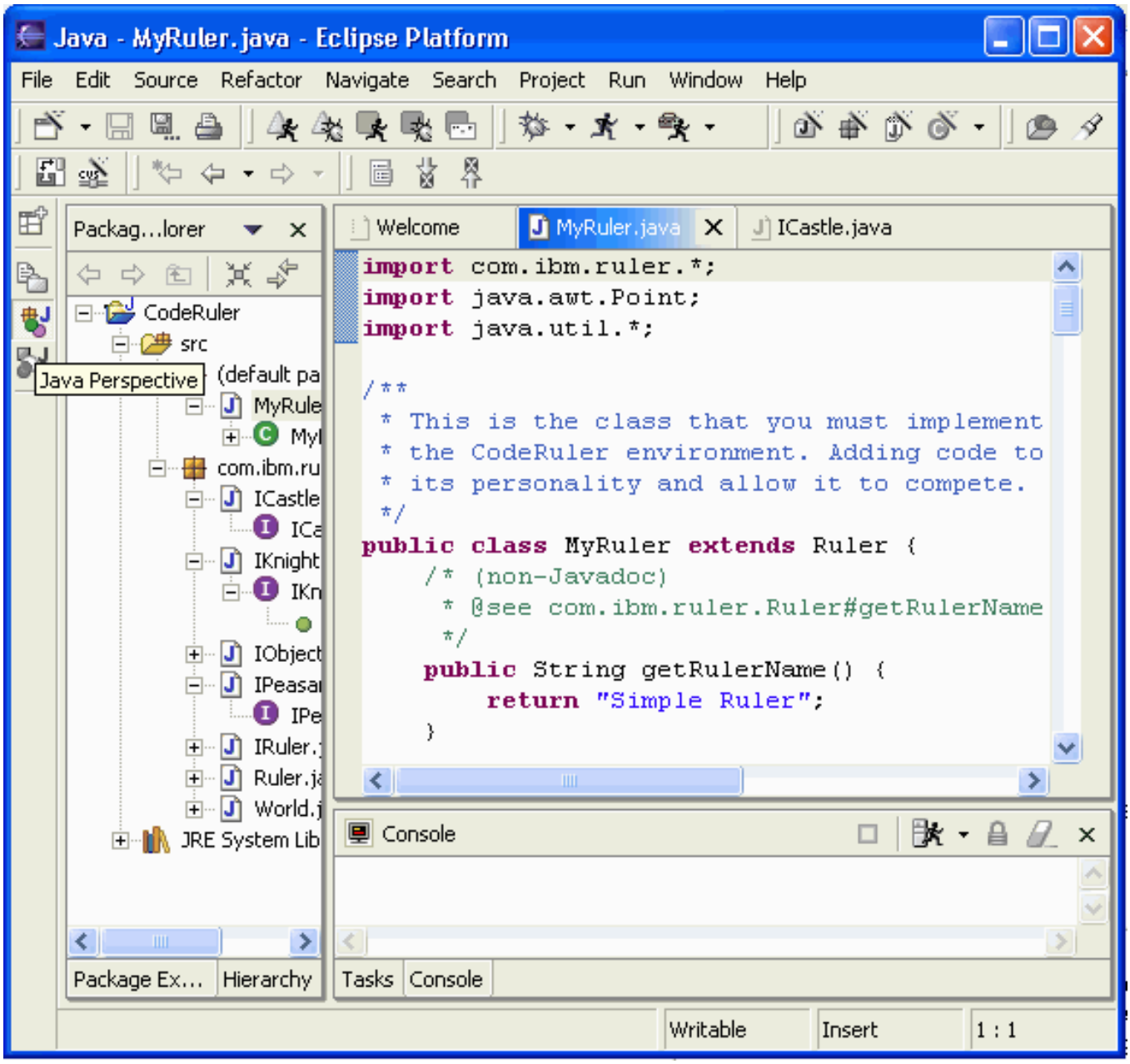
Figure 9. Creating a new CodeRuler project



Select IBM Games in the list on the left, as depicted in Figure 9. Next, select CodeRuler from the Game list on the right. Finally, click on **OK** to create a new CodeRuler project from the template. You are now ready to code your own CodeRuler.

On the tabs bar on the left hand side of the IDE, click on the Java Perspective tab. Figure 10 identifies this tab.

Figure 10. Selecting the Java perspective in Eclipse



Expanding the src node and the default package reveals the MyRuler.java node, as in Figure 10. The source code editor opens the file for editing when you double-click on the MyRuler.java node. This is where you must place your code.

Coding your first ruler

The first ruler you'll create is simple. It moves all the peasants randomly. Listing 1 shows the code for this ruler, with the added code highlighted in boldface.

Listing 1. Simple Ruler implementation of orderSubjects()

```
import java.util.Random;
...
protected Random rand = new Random();

public String getRulerName() {
   return "Simple Ruler";
   }
public String getSchoolName() {
   return "IBM developerWorks";
   }

public void orderSubjects(int lastMoveTime) {
   IPeasant[] peasants = getPeasants();
   for (int i = 0; i < peasants.length; i++) {
     move(peasants[i], rand.nextInt(8) + 1);
   }
}
```

The importance of being timely
When you code your ruler, be aware of the timing constraint that you are under. For the `initialize()` method, you are limited to one second, which should be plenty of time for any non-input/output code initialization. You are preempted if you take more than one second and might suffer from partial initialization. For each turn of the game, the `orderSubjects()` method limits you to half of a second. An incoming parameter of the `orderSubjects()` call lets you know how much time you used in the last turn. If you exceed the time limit, you are disqualified for the rest of the match.

The code in Listing 1 uses `java.util.Random` to generate a random number between 1 and 8. This number determines the direction that a peasant moves. Note the use of the `Ruler` class's `getPeasants()` method to obtain an array of all peasants, and the use of the `move()` method to move the peasants.

Moving the peasants randomly allows them to claim land. But because this ruler doesn't try to capture anything, your code doesn't need to move the knights.

Battling in your first match

To try out your first match, first save the newly edited ruler by either clicking on the save button on the toolbar or selecting **File >Save** from the menu. A save also compiles your code. Correct any typing or syntax errors before proceeding further.

You'll notice five iconic buttons, shown in Figure 11, that are CodeRuler specific.

Figure 11. Integrated CodeRuler buttons in the Eclipse toolbar

Table 4 explains the functions of the buttons in Figure 11, moving from left to right.

The "do not try" list for CodeRuler
A clever strategy is important to winning the game, but CodeRuler discourages tricky use of Java language features to hijack the gaming engine or win by other devious means. Your code *should not*:

- Define constuctors
- Use initialization blocks
- Create threads
- Create processes
- Write to files
- Use JDBC
- Use Swing or AWT to create GUI components
- Access the network or other similar system features
- Use reflection and introspection to discover and walk simulator internals

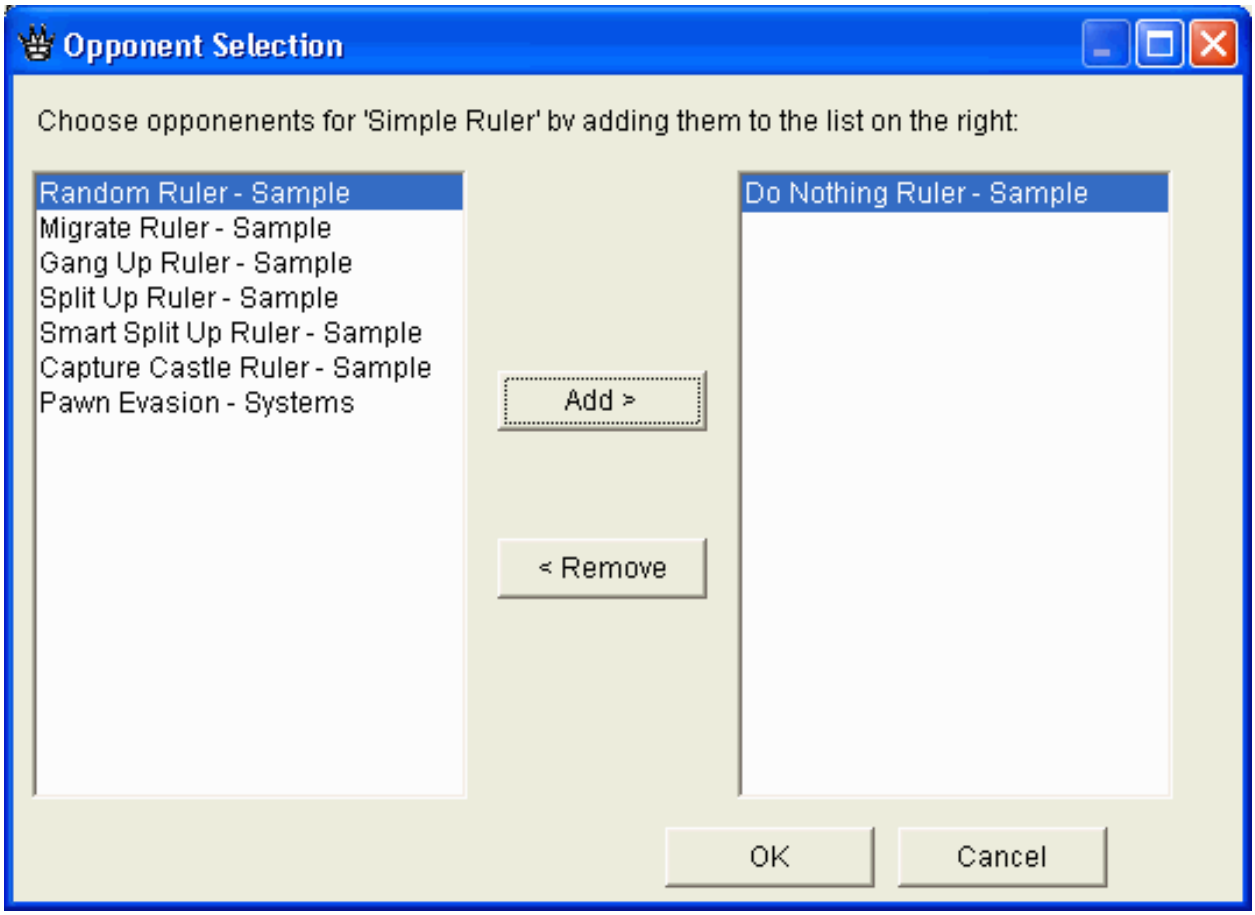
In all public matches and tournaments, players who use such hacker tactics will be disqualified from competition. A custom Java SecurityManager will catch most of these attempts.

Table 4. Functions of the CodeRuler buttons

Button	Description
Run against samples	Use this button to test your ruler against your choice of sample ruler(s).
Debug against samples	Use this button to test your ruler against your choice of sample ruler(s). Run your ruler in debug mode, stopping at any breakpoint you have set.
Run against other teams	After you have submitted your code, rulers from other teams will have been downloaded. Use this button to test your ruler against those from the other teams.
Debug against other teams	Run your ruler in the IDE's debug mode while testing your ruler against those from other teams.
Submit code	Submit your ruler. This also downloads an encrypted bundle of all the rulers submitted by other teams thus far.

Your first ruler will run against only the sample rulers in your initial experimentation. This means you'll use only the first button, the one highlighted in Figure 11. When you click on this button, CodeRuler starts and loads your ruler. You're given a chance to select your opponent(s), as shown in Figure 12.

Figure 12. Selecting your opponent for a match



Try adding one Do Nothing Ruler. Start the match and observe how your peasants randomly move around and claim the land. You should win this match easily.

Next, try the Random Ruler. This ruler behaves almost identically to yours. The average land occupation is about equal.

If you try any of the other sample rulers, the Simple Ruler you created will likely lose. Most of the other sample rulers attempt to capture your pieces agressively. It's time add an offensive edge to your Simple Ruler.

Creating an offensive ruler

Listing 2 shows the code for the modified ruler, with the added code highlighted.

Listing 2. Modified Simple Ruler implementation to capture opponents aggressively

```
import com.ibm.ruler.*;
import java.awt.Point;
import java.util.Random;
import java.util.Vector;

public class MyRuler extends Ruler {
        public String getRulerName() {
          return "Simple Ruler";
        }

        public String getSchoolName() {
          return "IBM developerWorks";
        }

        public void initialize() {
        }
    protected Random rand = new Random();
    protected Vector enemies = new Vector();

    public void orderSubjects(int lastMoveTime) {

           IPeasant[] peasants = getPeasants();
               IKnight[] knights = getKnights();
           for (int i = 0; i < peasants.length; i++) {
                 move(peasants[i], rand.nextInt(8) + 1);
               }
```

```
          enemies.clear();
              IPeasant[] otherPeasants = World.getOtherPeasants();
              IKnight[] otherKnights = World.getOtherKnights();
              ICastle[] otherCastles = World.getOtherCastles();

              for (int i=0; i<otherPeasants.length; i++) {
                 enemies.add(otherPeasants[i]);

              }
              for (int i=0; i<otherKnights.length; i++ ){
                 enemies.add(otherKnights[i]);
              }
              for (int i=0; i<otherCastles.length; i++) {
                 enemies.add(otherCastles[i]);
              }

          int size = knights.length;
              for (int i = 0; i < size; i++) {
              IKnight curKnight = knights[i];
                if (!enemies.isEmpty()) {
                  IObject curEnemy = (IObject) enemies.remove(0);
                   moveAndCapture(curKnight, curEnemy);
                }
                else
                   break;
               } // of outter for

          }



  public void moveAndCapture(IKnight knight, IObject enemy) {
        if ((enemy == null) || !enemy.isAlive())
          return;
         // find the next position in the direction of the enemy
        int dir = knight.getDirectionTo(enemy.getX(), enemy.getY());
        Point np = World.getPositionAfterMove(knight.getX(), knight.getY(), dir);

      if (np == null)
        return;
      if ((np.x == knight.getX()) && (np.y == knight.getY())) {
          move(knight, rand.nextInt(8) + 1);
                return;
       }

        // capture anything that is in our way
        IObject obj = World.getObjectAt(np.x, np.y);
        if ((obj != null)  && (obj.getRuler()!= this))
          capture(knight, dir);
        else
          move(knight, dir);
    }
}
```

You'll recognize the green highlighted code in Listing 2.

The versatile World object
Before you embark on extensive ruler coding, make sure you spend some time studying the documentation for the `World` object. This object contains many static methods that you'll find useful for implementing your strategy. For example, you can use `getLandOwner()` to find out who owns a square of land, use `getObjectAt()` to identify the game piece that's in a specific location, and use `getOtherPeasants()`, `getOtherKnights()`, `getOtherCastles()`, and `getOtherRulers()` to discover your enemies.

The red highlighted code sets up a `Vector` consisting of all the enemy game pieces that are alive in the simulation world. Note the use of the `World` object to get this information (that is, `World.getOtherPeasants()`).

The blue highlighted code loops through all of your knights and makes each of them move toward a living opponent's game piece. It also captures any enemy pieces that might be in place. It uses the `moveAndCapture()` method to move and capture.

The `moveAndCapture()` method moves a specified knight toward a specified enemy piece. It uses the `World` object's `getPositionAfterMove()` method to determine if the knight is stuck, and makes a random move if it is. It also uses the `World` object's `getObjectAt()` method to test and capture any enemy pieces that might be in its way.

Try this new `Simple Ruler` against some of the sample rulers. You'll see that it fares quite well against many of them. Of course, there's plenty of room for improvement. As an exercise, you can try modifying the code to:

- Command your castle(s) to generate knights when the number of knights get low.
- Assign targets to your knights more efficently.
- Program your peasants to claim land more efficiently.

- Program your peasants to evade attempted capture.
- Switch to a defensive survival strategy when the number of peasants and knights is low.

Conclusion

It's your choice: from the simplest heuristic-based robotic ruler to the most sophisticated statistical gaming theory model driven commanders, CodeRulers span all possibilities. Just as in the real world, the most sophisticated strategy and intricate coding don't always guarantee sure winners. In fact, some of the champion rulers deploy the most simple, yet elegant, guerrilla tactics. If strategic design and Java development is your blood, you owe it to yourself to give CodeRuler a spin.

Resources

- Learn more about the ACM International Collegiate Programming Contest, the world's oldest, largest, and most prestigious programming contest.
- Download the latest version of the CodeRuler simulation engine and associated documentation from IBM alphaWorks.
- For the latest version of the Eclipse IDE, documentation, mailing lists, and community news, visit eclipse.org.
- Check out another classic simulation game powered by the same simulation engine as CodeRuler. Code Rally puts you in the hot seat at a race car rally!
- For another highly popular battle simulation game from alphaWorks, try Robocode. The active international community of Robocode gamers is sure to give you a challenge.
- Learn more about Robocode from Sing Li's articles "Rock 'em, sock 'em Robocode!" (*developerWorks*, January 2002) and "Rock 'em, sock 'em Robocode: Round 2" (*developerWorks*, May 2002).
- Visit the Developer Bookstore for a comprehensive listing of technical books, including hundreds of Java-related titles.
- You'll find hundreds of articles about every aspect of Java programming in the *developerWorks* Java technology zone.
- Interested in test driving IBM products without the typical high-cost entry point or short-term evaluation license? The developerWorks Subscription provides a low-cost, 12-month, single-user license for WebSphere®, DB2®, Lotus®, Rational®, and Tivoli® products -- including the Eclipse-based WebSphere Studio IDE -- to develop, test, evaluate, and demonstrate your applications.

About the author

Sing Li is the author of *Professional Apache Tomcat 5*, *Pro JSP, Third Edition*, *Early Adopter JXTA*, *Professional Jini*, and numerous other books with Wrox Press. He is a regular contributor to technical magazines and an active evangelist of the P2P evolution. Sing is a consultant and freelance writer and can be reached at westmakaha@yahoo.com.

What do you think of this document?

○ Killer! (5) ○ Good stuff (4) ○ So-so; not bad (3) ○ Needs work (2) ○ Lame! (1)

Comments?

Submit feedback

Conquer medieval kingdoms with CodeRuler

A behind-the-scenes interview with CodeRuler's creator

Taking a short break from my busy mediaeval conquests, I caught up with Tim deBoer, the creator of CodeRuler, to ask him a few questions. Tim works in the IBM WebSphere Tools group.

developerWorks: Why CodeRuler?

deBoer: I've always enjoyed playing computer games, and as a programmer, I enjoy creating new games just as much.

dW: Is CodeRuler a formal project? How big was the development team?

deBoer: CodeRuler was developed specifically for the ACM-ICPC and is the result of refining ideas with the ACM-ICPC systems teams for several years. In particular, John Clevenger (co-designer) and Sam Ashoo have been a great help.

CodeRuler was designed and developed by a very small team to keep it a surprise. To say that going from this small team to over 200 of the brightest computer science students in the world is a good stress test is a vast understatement. Even after hours of combing over the game, the API, and documentation, we always receive questions we hadn't thought of, and the game is used in ways we could never have predicted.

dW: Other than the ACM-ICPC, where else have CodeRuler tournaments or contests been held?

deBoer: CodeRuler has been used in several contests, including a programming contest at the University of California at San Diego and the USA Computing Olympiad. We hope it will continue to be used by gamers and teachers in the Java community, and I look forward to seeing some great CodeRulers.

dW: Tell me about some of the design thoughts behind CodeRuler.

deBoer: The game was designed so that one single good strategy should not be enough to win. It's a combination of using your peasants, knights, and even the castle in a coordinated and well-planned way. In most cases, the team that wins will have thought out separate strategies for their peasants and knights, and figured out how to implement them so that they don't interfere with or duplicate each other's work. For instance, a simple implementation for peasants claiming land could cause the peasants to walk over each other's paths a lot instead of claiming new land. And don't forget the little details, like having your peasants avoid enemy knights so that they don't get captured!

dW: Have you encountered any unexpectedly innovative strategies lately?

deBoer: Out of the 68 teams that competed in the ACM-ICPC Java Challenge, the winning Leilu Tarawa team from the University of Calgary came up with the most original solution.

Each team is given points when its knights win battles against peasants, knights, and castles. What they realized is that castles don't fight back or run away, and they can't be destroyed, so they are a "renewable" point resource. Their strategy was to let other teams conquer their castle, and then take back the castle by conquering it themselves. They would lose out on building up their armies and winning other battles, but using this technique they could gain a lot of points during each match. This would also give the other team that was trying to conquer their castle a lot of points. But evened out over a number of matches, and as long as multiple teams tried to take over the castle, they would maintain the lead.

dW: Some of our advanced readers are interested in the simulation engine behind CodeRuler. Can you tell us something about the technology?

deBoer: The CodeRuler simulator engine is based on the same engine used in CodeRally. It allows the simulation to be run in several different modes -- displaying the simulation at the same time as the UI for development, or running a tournament and saving it to disk for later playback. It also has several modes to run test tournaments for debugging on the contestants' machines, a spectator display that plays random matches from the submissions, and the full tournament, which plays multiple matches and rounds to determine the winners.

The simulator is a state-based game; it calls each team in turn to determine their next move. Then it calculates the next state by walking through each "character" in the game and moving each one in turn, and then displays the result on screen.

dW: How did you ensure fair game play in the design of CodeRuler?

deBoer: CodeRuler allows multiple teams to play against each other in real-time competition. To keep the game engine secure, several things need to be ensured:

- No team should be able to directly access or affect the code from another team. (All interaction is through the game engine.)
- When code from other teams is copied onto the local machine for testing, it should be encrypted so that there is no chance of the local team using the source or compiled code.
- Badly written code from one of the teams should never be able to crash the game.
- No team should receive more CPU time than another team.

Writing the game using Java is a great advantage here, since the Java language natively supports many of these things. The following parts of the Java language were used:

- The Java language supports having each team's code in its own namespace, so it can't be directly accessed by another team. When references are passed via the CodeRuler API, teams are always given a proxy to the other team instead of a direct reference.
- When code from other teams is copied onto the local machine, it is already encrypted. A special ClassLoader was created that loads classes directly from the encrypted files and runs them from memory so that the decrypted code is never visible.
- Because we provided our own SecurityManager, each team is prevented from creating threads, ending the game, and so on.
- Running each team at the same time could cause one team to hog the CPU, and letting them run on the main thread could be dangerous if they have an infinite loop that stops the game from continuing. To solve this, a thread is created to launch each team's code in turn, and the main thread monitors and stops the thread if it continues past a set timeout.

dW: Any last words for budding CodeRulers?

deBoer: May the best code win!

Return to article.