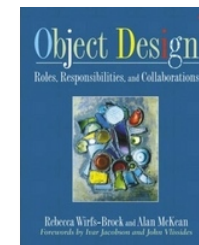


7. Inheritance and Refactoring

Inheritance and Refactoring

Source

- > Wirfs-Brock & McKean, *Object Design — Roles, Responsibilities and Collaborations*, 2003.



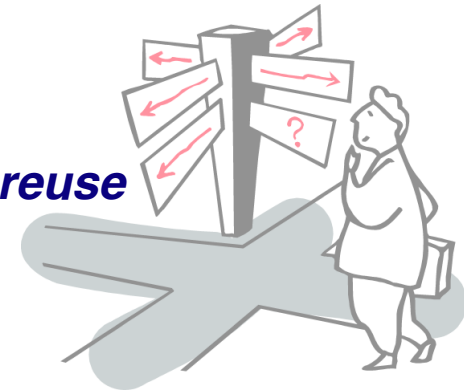
Roadmap

- > Uses of inheritance
 - conceptual hierarchy, polymorphism and code reuse
- > TicTacToe and Gomoku
 - interfaces and abstract classes
- > Iterative development
 - Quiet testing
- > Refactoring
 - iterative strategies for improving design
- > Top-down decomposition
 - decomposing algorithms to reduce complexity



Roadmap

- > **Uses of inheritance**
 - *conceptual hierarchy, polymorphism and code reuse*
- > TicTacToe and Gomoku
 - interfaces and abstract classes
- > Iterative development
 - Quiet testing
- > Refactoring
 - iterative strategies for improving design
- > Top-down decomposition
 - decomposing algorithms to reduce complexity



What is Inheritance?

Inheritance in object-oriented programming languages is a mechanism to:

- *derive new subclasses* from existing classes
- where subclasses *inherit all the features* from their parent(s)
- and may *selectively override* the implementation of some features.

Inheritance mechanisms

OO languages realize inheritance in different ways:

<i>self</i>	<i>dynamically access</i> subclass methods
<i>super</i>	<i>statically access</i> overridden, inherited methods
<i>multiple inheritance</i>	inherit features from <i>multiple superclasses</i>
<i>abstract classes</i>	<i>partially defined classes</i> (to inherit from only)
<i>mixins</i>	build classes from <i>partial sets of features</i>
<i>interfaces</i>	<i>specify</i> method argument and return types
<i>subtyping</i>	guarantees that subclass instances can be <i>substituted</i> for their parents

The Board Game

Tic Tac Toe is a pretty dull game, but there are many other interesting games that can be played by *two players with a board and two colours of markers*.

Example: Go-moku

“A Japanese game played on a go board with players alternating and attempting to be first to place five counters in a row.”

— Random House

We would like to implement a program that can be used to play several *different kinds of games using the same game-playing abstractions* (starting with TicTacToe and Go-moku).

Inheritance is used for three orthogonal, but related purposes

Conceptual hierarchy (domain modeling):

- > Go-moku *is-a kind of* Board Game; Tic Tac Toe is-a kind of Board Game

Polymorphism (design):

- > Instances of Gomoku and TicTacToe can be *uniformly manipulated* as instances of BoardGame by a client program

Software reuse (implementation):

- > Gomoku and TicTacToe reuse the BoardGame *interface*
- > Gomoku and TicTacToe reuse and extend the BoardGame *representation* and the *implementations* of its operations

Roadmap

- > Uses of inheritance
 - conceptual hierarchy, polymorphism and code reuse
- > **TicTacToe and Gomoku**
 - *interfaces and abstract classes*
- > Iterative development
 - Quiet testing
- > Refactoring
 - iterative strategies for improving design
- > Top-down decomposition
 - decomposing algorithms to reduce complexity



Class Diagrams

The TicTacToe class currently looks like this:

Key	
-	private feature
#	protected feature
+	public feature
<u>create()</u>	static feature
<i>checkWinner()</i>	abstract feature

TicTacToe
-gameState : char [3][3] -winner: Player -turn : Player -player : Player[2] -squaresLeft : int
+ <u>create</u> (Player, Player) +update() +move(char, char, char) +winner() : Player +notOver() : boolean +squaresLeft() : int -set(char, char, char) -get(char, char) : char -swapTurn() -checkWinner() -inRange(char col, char row) : boolean

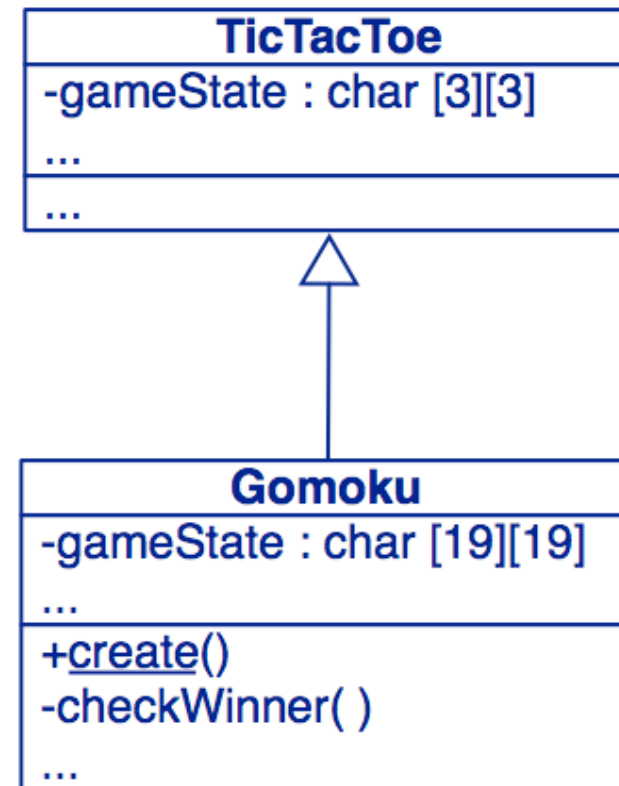
A bad idea ...

Why not simply use inheritance for incremental modification?

Exploiting inheritance for code reuse without refactoring tends to lead to:

- > *duplicated code* (similar, but not reusable methods)
- > *conceptually unclear design* (arbitrary relationships between classes)

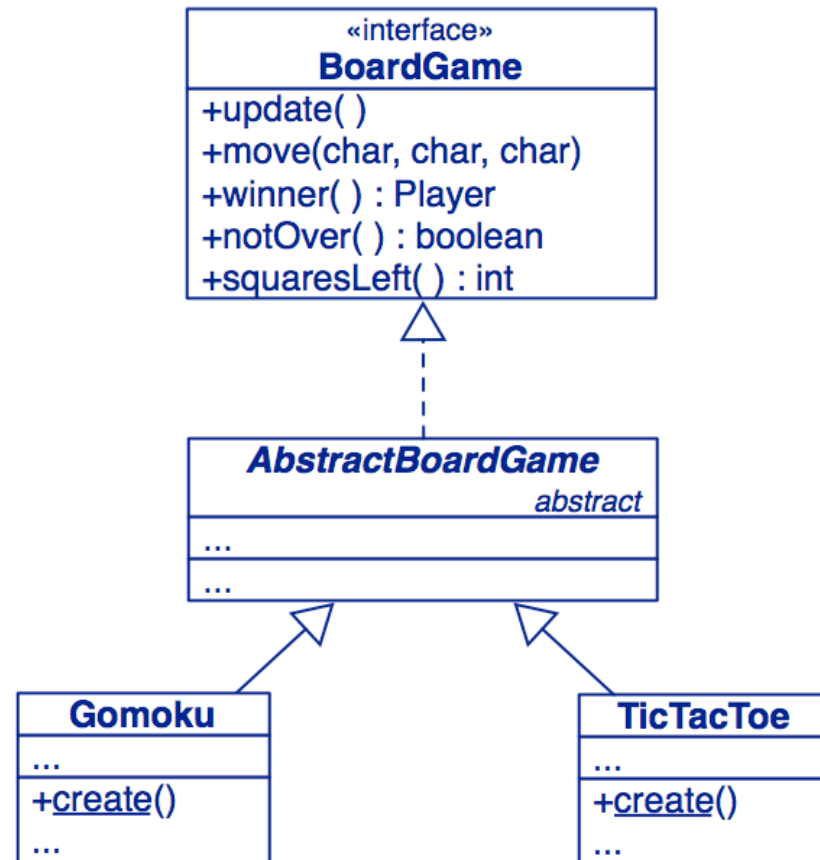
Gomoku is not a kind of TicTacToe



Class Hierarchy

Both Go-moku and Tic Tac Toe are *kinds of Board games* (IS-A).

We would like to define a *common interface*, and factor the common functionality into a *shared parent class*.



Behaviour that is not shared will be implemented by the subclasses.

Roadmap

- > Uses of inheritance
 - conceptual hierarchy, polymorphism and code reuse
- > TicTacToe and Gomoku
 - interfaces and abstract classes
- > **Iterative development**
 - Quiet testing
- > Refactoring
 - iterative strategies for improving design
- > Top-down decomposition
 - decomposing algorithms to reduce complexity



Iterative development strategy

We need to find out which TicTacToe functionality will:

- already work for both TicTacToe and Gomoku
- need to be adapted for Gomoku
- can be generalized to work for both

Example: set() and get() will not work for a 19×19 board!

Rather than attempting a “big bang” redesign, we will iteratively redesign our game:

- introduce a BoardGame interface that TicTacToe implements
- move all TicTacToe implementation to an AbstractBoardGame parent
- fix, refactor or make abstract the non-generic features
- introduce Gomoku as a concrete subclass of AbstractBoardGame

After each iteration we run our regression tests to make sure nothing is broken!

 When should you run your (regression) tests?

✓ *After every change to the system.*

Version 3 (add interface)

We specify the interface both subclasses should implement:

```
public interface BoardGame {  
    public void update() throws IOException;  
    public void move(char col, char row, char mark);  
    public Player currentPlayer();    // NB: new method  
    public Player winner();  
    public boolean notOver();  
    public int squaresLeft();  
}
```

Initially we focus only on *abstracting* from the current TicTacToe implementation

Speaking to an Interface

Clients of TicTacToe and Gomoku should only depend on the BoardGame *interface*:

```
public class GameDriver {  
    public static void main(String args[]) {  
        Player X = new Player('X');  
        Player O = new Player('O');  
        playGame(new TicTacToe(X, O));  
    }  
  
    public static void playGame(BoardGame game) {  
        ...  
    }  
}
```

Speak to an interface, not an implementation.

Roadmap

- > Uses of inheritance
 - conceptual hierarchy, polymorphism and code reuse
- > TicTacToe and Gomoku
 - interfaces and abstract classes
- > Iterative development
 - ***Quiet testing***
- > Refactoring
 - iterative strategies for improving design
- > Top-down decomposition
 - decomposing algorithms to reduce complexity



Quiet Testing

Our current TestDriver prints the state of the game after each move, making it hard to tell when a test has failed.

Tests should be silent unless an error has occurred!

```
public static void playGame(BoardGame game, boolean verbose) {  
    ...  
    if (verbose) {  
        System.out.println();  
        System.out.println(game);  
    }  
    ...  
}
```

NB: we must shift all responsibility for printing to playGame().

Quiet Testing (2)

A more flexible approach is to let the *client* supply the `PrintStream`:

```
public static void playGame(BoardGame game, PrintStream out) {  
    try {  
        do { // all printing must move here ...  
            out.println();  
            out.println(game);  
            out.print("Player "  
                + game.currentPlayer().mark() + " moves: ");  
            ...  
        } while (true);  
    } catch (Exception e) {  
        // ...  
    }  
}
```

The TestDriver can simply send the output to a Null stream:

```
playGame(game, System.out);  
playGame(game, new PrintStream(new NullOutputStream()));
```

NullOutputStream

A Null Object implements an interface with null methods:

```
public class NullOutputStream extends OutputStream {  
    public NullOutputStream() { super(); }  
  
    // Null implementation of inherited abstract method  
    public void write(int b) throws IOException { }  
}
```

Null Objects are useful for eliminating flags and switches.

TicTacToe adaptations

In order to pass responsibility for printing to the GameDriver, a BoardGame must provide a method to *export the current Player*:

```
public class TicTacToe implements BoardGame {  
    ...  
    public Player currentPlayer() {  
        return player[turn];  
    }  
}
```

Now we run our regression tests and (after fixing any bugs) continue.

Version 4 — add abstract class

AbstractBoardGame will provide common variables and methods for TicTacToe and Gomoku.

```
public abstract class AbstractBoardGame implements BoardGame {  
    static final int X = 0;  
    static final int O = 1;  
    ...  
}
```

In a first step we include the entire TicTacToe implementation ...

 When should a class be declared abstract?

✓ *Declare a class abstract if it is intended to be subclassed, but not instantiated.*

Roadmap

- > Uses of inheritance
 - conceptual hierarchy, polymorphism and code reuse
- > TicTacToe and Gomoku
 - interfaces and abstract classes
- > Iterative development
 - Quiet testing
- > **Refactoring**
 - *iterative strategies for improving design*
- > Top-down decomposition
 - decomposing algorithms to reduce complexity



Refactoring

Refactoring is a process of moving methods and instance variables from one class to another to improve the design, specifically to:

- reassign responsibilities
- eliminate duplicated code
- reduce coupling: interaction between classes
- increase cohesion: interaction within classes

Refactoring strategies

We have adopted one possible refactoring strategy, first moving everything except the constructor from `TicTacToe` to `AbstractBoardGame`, and changing all private features to protected. `TicTacToe` inherits everything:

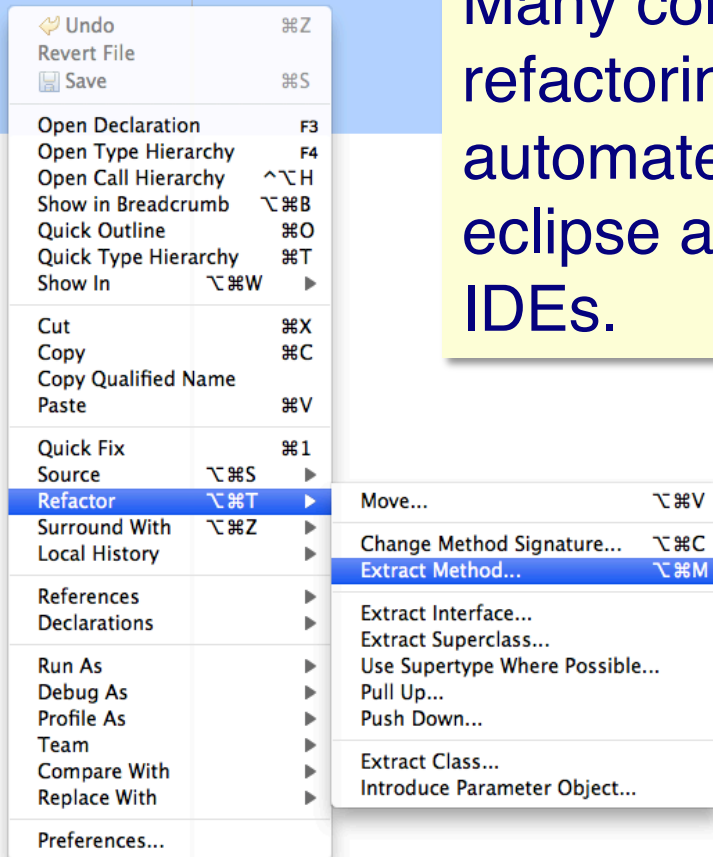
```
public class TicTacToe extends AbstractBoardGame {  
    public TicTacToe(Player playerX, Player playerO)  
    {  
        super(playerX, playerO);  
    }  
}
```

We could equally have started with an empty `AbstractBoardGame` and gradually moved shared code there.

Refactoring support in Eclipse

```
protected void checkWinner()
{
    char player;
    for (char row='3'; row>='1'; row--) {
        player = this.get('a',row);
        if (player == this.get('b',row)
            && player == this.get('c',row)) {
            this.setWinner(player);
            return;
        }
    }
    for (char col='a'; col <='c'; col++) {
        player = this.get(col, '1');
        if (player == this.get(col, '2')
            && player == this.get(col, '3')) {
            this.setWinner(player);
            return;
        }
    }
    player = this.get('b', '2');
    if (player == this.get('a', '1')
        && player == this.get('c', '3')) {
        this.setWinner(player);
        return;
    }
    if (player == this.get('a', '3')
        && player == this.get('c', '1')) {
        this.setWinner(player);
        return;
    }
}

/**
 * Look up which player is the winner, and set winner
 * accordingly. Hm. Maybe we should store Players
 * instead of chars in our array!
 */
protected void setWinner(char aPlayer) {
    if (aPlayer == ' ')
        return;
    if (aPlayer == player[X].mark())
        winner = player[X];
    else
        winner = player[0];
}
```



Many common refactorings are automated by eclipse and other IDEs.

Version 5 — refactoring

Now we must check which parts of `AbstractBoardGame` are *generic*, which must be *repaired*, and which must be *deferred* to its subclasses:

- > the number of rows and columns and the winning score may vary
 - introduce instance variables and an `init()` method
 - rewrite `toString()`, `invariant()`, and `inRange()`
- > `set()` and `get()` are inappropriate for a 19×19 board
 - index directly by integers
 - fix `move()` to take `String` argument (e.g., “f17”)
 - add methods to parse string into integer coordinates
- > `getWinner()` and `toString()` must be generalized

AbstractBoardGame

We introduce an abstract `init()` method for arbitrary sized boards:

```
public abstract class AbstractBoardGame ... {  
    protected abstract void init();  
    ...  
}
```

And call it from the constructors of our subclasses:

```
public class TicTacToe extends AbstractBoardGame {  
    ...  
    protected void init() {  
        rows = 3;  
        cols = 3;  
        winningScore = 3;  
    }  
    ...  
}
```

Or: introduce a constructor for `AbstractBoardGame`!

BoardGame

Most of the changes in `AbstractBoardGame` are to protected methods.

The only public (interface) method to change is `move ()`:

```
public interface BoardGame {  
    ...  
    public void move(String coord, char mark);  
    ...  
}
```

Player

The Player's move() method can now be radically simplified:

```
public void move(BoardGame game) throws IOException {  
    String line;  
    line = in.readLine();  
    if (line == null) {  
        throw new IOException("end of input");  
    }  
    game.move(line, this.mark());  
}
```

 *How can we make the Player responsible for checking if the move is valid?*

Roadmap

- > Uses of inheritance
 - conceptual hierarchy, polymorphism and code reuse
- > TicTacToe and Gomoku
 - interfaces and abstract classes
- > Iterative development
 - Quiet testing
- > Refactoring
 - iterative strategies for improving design
- > **Top-down decomposition**
 - *decomposing algorithms to reduce complexity*



Version 6 — Gomoku

The final steps are:

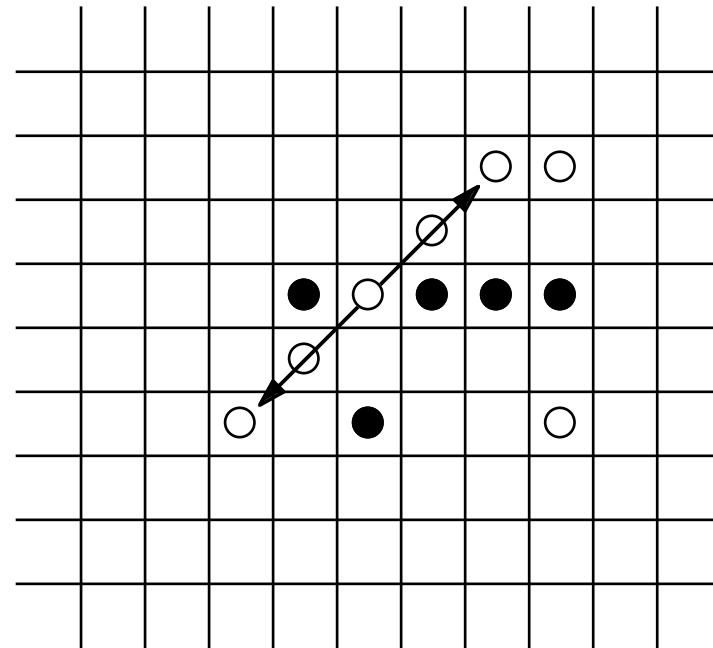
- > rewrite `checkWinner()`
- > introduce Gomoku
 - modify `TestDriver` to run tests for both `TicTacToe` and Gomoku
 - print game state whenever a test fails
- > modify `GameDriver` to query user for either `TicTacToe` or Gomoku

Keeping Score

The Go board is *too large to search exhaustively* for a winning Go-moku score.

We know that *a winning sequence must include the last square marked*. So, it suffices to search in all four directions *starting from that square* to see if we find 5 in a row.

📎 *Whose responsibility is it to search?*



A new responsibility ...

Maintaining the state of the board and searching for a winning run seem to be *unrelated responsibilities*. So let's introduce a new object (a Runner) to run and count a Player's pieces.

```
protected void checkWinner(int col, int row)... {
    char player = this.get(col,row);
    Runner runner = new Runner(this, col, row);
    // check vertically
    if (runner.run(0,1) >= this.winningScore)
        { this.setWinner(player); return; }
    // check horizontally
    if (runner.run(1,0) >= this.winningScore)
        { this.setWinner(player); return; }
    ...
}
```

The Runner

The Runner must know its game, its home (start) position, and its current position:

```
public class Runner {  
    BoardGame game;  
    int homeCol, homeRow;           // Home col and row  
    int col=0, row=0;              // Current col & row  
  
    public Runner(BoardGame myGame, int myCol, int myRow) {  
        game = myGame;  
        homeCol = myCol;  
        homeRow = myRow;  
    }  
    ...  
}
```

Top-down decomposition

Implement algorithms abstractly, introducing helper methods for each abstract step, as you decompose:

```
public int run(int dcol, int drow)
    throws AssertionError {
    int score = 1;
    this.goHome() ;
    score += this.forwardRun(dcol, drow);
    this.goHome();
    score += this.reverseRun(dcol, drow);
    return score;
}
```

Well-chosen names eliminate the need for most comments!

Recursion

Many algorithms are more naturally expressed with recursion than iteration.

Recursively move forward as long as we are in a run. Return the length of the run:

```
private int forwardRun(int dcol, int drow) {  
    this.move(dcol, drow);  
    if (this.samePlayer())  
        return 1 + this.forwardRun(dcol, drow);  
    else  
        return 0;  
}
```

More helper methods

Helper methods keep the main algorithm *clear and uncluttered*, and are mostly *trivial to implement*.

```
private int reverseRun(int dcol, int drow) ... {  
    return this.forwardRun(-dcol, -drow);  
}  
  
private void goHome() {  
    col= homeCol;  
    row = homeRow;  
}
```

 *How would you implement `move()` and `samePlayer()`?*

BoardGame

The Runner now needs access to the `get()` and `inRange()` methods so we make them *public*:

```
public interface BoardGame {  
    ...  
    public char get(int col, int row);  
    public boolean inRange(int col, int row);  
    ...  
}
```

 Which methods should be public?

✓ *Only publicize methods that clients will really need, and will not break encapsulation.*

Gomoku

Gomoku is similar to TicTacToe, except it is played on a 19x19 Go board, and the winner must get 5 in a row.

```
public class Gomoku extends AbstractBoardGame {  
    public Gomoku(Player playerX, Player playerO) {  
        super(playerX, playerO);  
    }  
    protected void init() {  
        rows = 19;  
        cols = 19;  
        winningScore = 5;  
    }  
}
```

In the end, Gomoku and TicTacToe could inherit *everything* (except their constructor) from AbstractGameBoard!

Abstract test framework

```
public abstract class AbstractBoardGameTest extends TestCase {
    protected BoardGame game;

    public AbstractBoardGameTest (String name) { super(name); }

    public void checkGame(String Xmoves, String Omoves,
                          String winner, int squaresLeft) {
        Player X = new Player('X', Xmoves);
        Player O = new Player('O', Omoves);
        game = makeGame(X,O);
        GameDriver.playGame(game, new PrintStream(new NullOutputStream()));
        assertEquals(game.winner().name(), winner);

        assertEquals(game.squaresLeft(), squaresLeft);
    }
    abstract protected BoardGame makeGame(Player X, Player O) ;
    ...
}
```

Gomoku tests ...







Subclasses specialize the factory method for instantiating the game

```
public class GomokuTest extends AbstractBoardGameTest {
    ...






    public void testXWinsDiagonal() {
        checkGame("\naa\n"           // nonsense input
            + "f6\ng5\ne7\nd8\nc9\n",
            "b2\nh4\nc3\nd4\n",
            "X", (19*19-9));
    }

    protected BoardGame makeGame(Player X, Player O) {
        return new Gomoku(X, O);
    }
}
```

What you should know!

-  *How does polymorphism help in writing generic code?*
-  *When should features be declared protected rather than public or private?*
-  *How do abstract classes help to achieve code reuse?*
-  *What is refactoring? Why should you do it in small steps?*
-  *How do interfaces support polymorphism?*
-  *Why should tests be silent?*

Can you answer these questions?

-  *What would change if we didn't declare `AbstractBoardGame` to be abstract?*
-  *How does an interface (in Java) differ from a class whose methods are all abstract?*
-  *Can you write generic `toString()` and `invariant()` methods for `AbstractBoardGame`?*
-  *Is `TicTacToe` a special case of `Gomoku`, or the other way around?*
-  *How would you reorganize the class hierarchy so that you could run `Gomoku` with boards of different sizes?*

License

<http://creativecommons.org/licenses/by-sa/2.5/>



You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:



Attribution. You must attribute the work in the manner specified by the author or licensor.



Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.