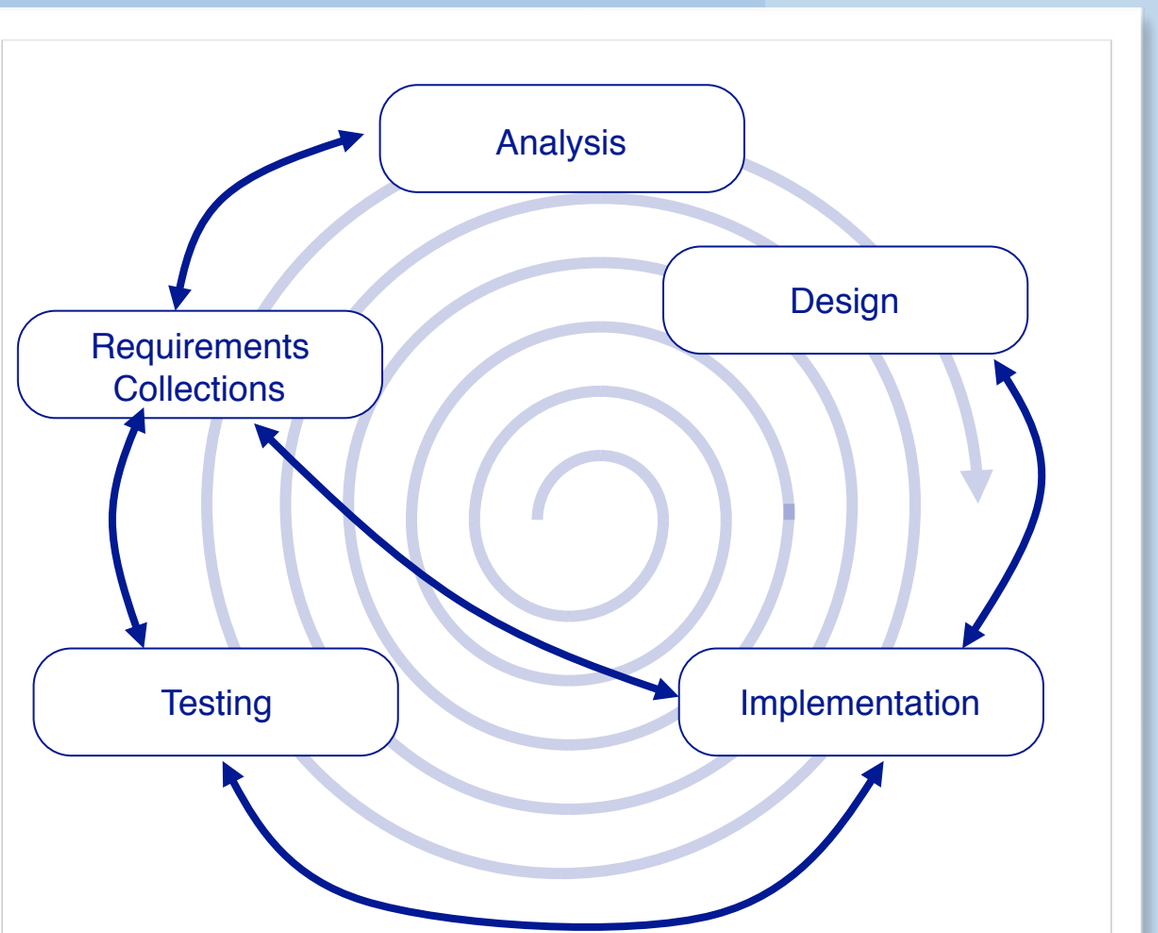
UNIVERSITÄT
BERN

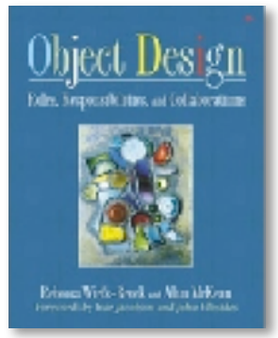
6. Iterative Development

Oscar Nierstrasz



Iterative Development

Sources



- Rebecca Wirfs-Brock, Alan McKean, *Object Design — Roles, Responsibilities and Collaborations*, Addison-Wesley, 2003.
- Kent Beck, *Extreme Programming Explained — Embrace Change*, Addison-Wesley, 1999.

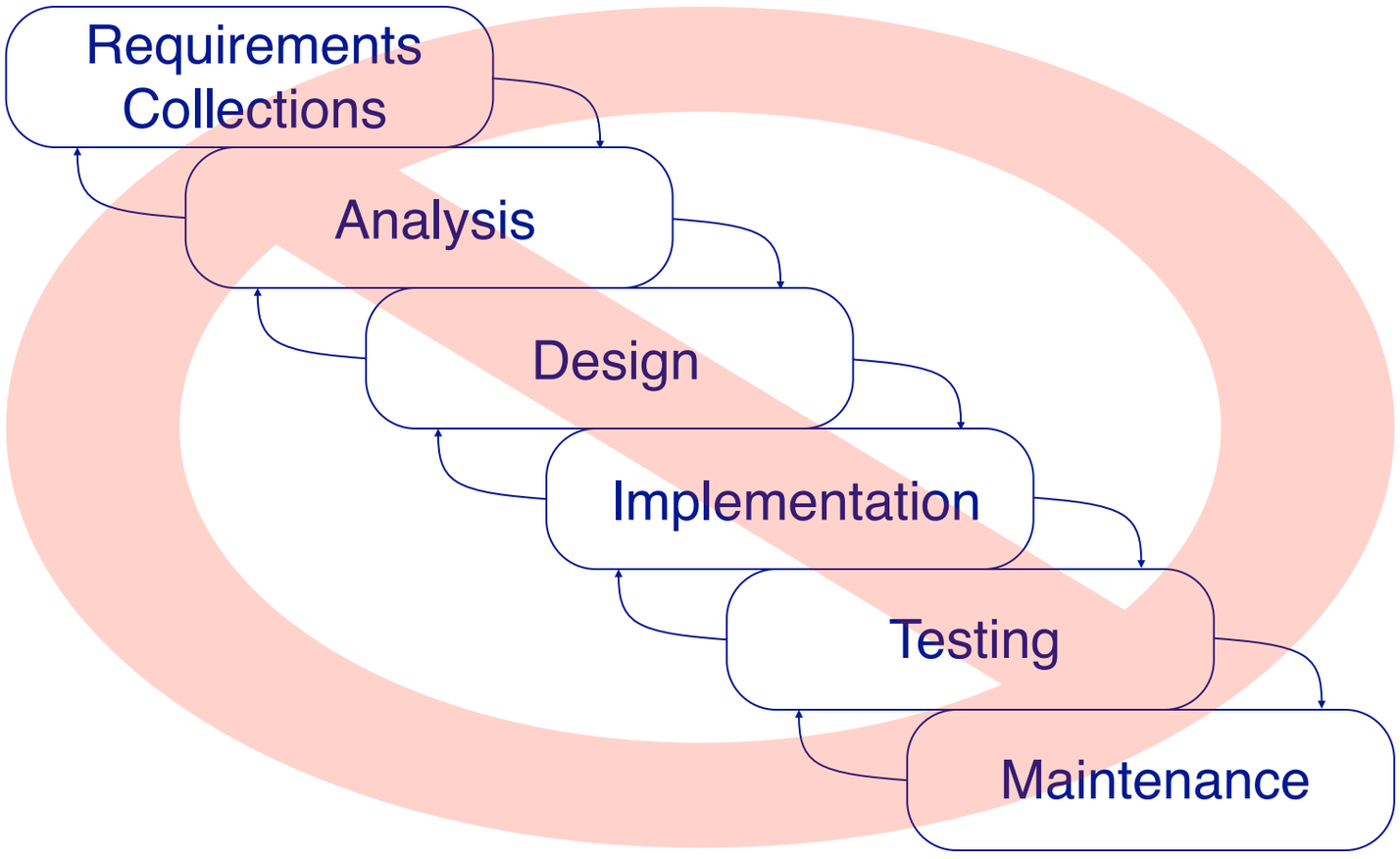
Roadmap

- The iterative software lifecycle
- Responsibility-driven design
- TicTacToe example
  - —Identifying objects
  - —Scenarios
  - —Test-first development
  - —Printing object state
  - —Testing scenarios
  - —Representing responsibilities as contracts

Roadmap

- **The iterative software lifecycle**
- Responsibility-driven design
- TicTacToe example
  - Identifying objects
  - Scenarios
  - Test-first development
  - Printing object state
  - Testing scenarios
  - Representing responsibilities as contracts

The Classical Software Lifecycle



The classical software lifecycle models the software development as a step-by-step "waterfall" between the various development phases.

The waterfall model is unrealistic for many reasons, especially:

- requirements must be "frozen" too early in the life-cycle
- requirements are validated too late

The waterfall model of software development was first described by Winston Royce in 1970 as an example of a flawed process:

http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

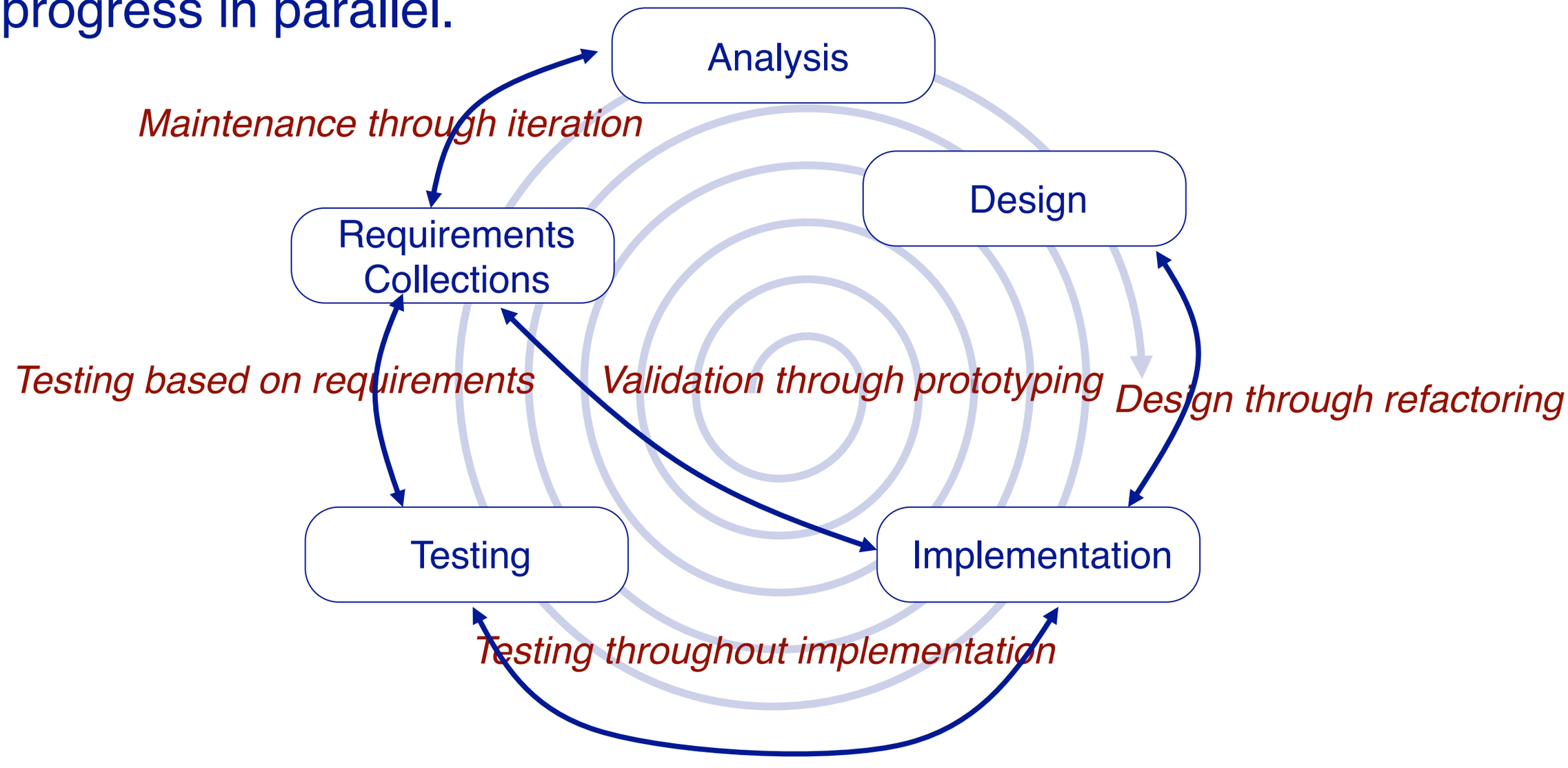
In this approach, each phase is carried out strictly sequentially, with deliverables (documents) being passed from one phase to the next. The deliverables constitute the “water” falling downhill.

The key problems are (i) all requirements must be completely known in advance, and (ii) only at the end of the process is there any working software that can be validating. In essence waterfall process maximizes risk and minimizes value.

Curiously many organizations base their development process on the waterfall model, even though it is known not to work in practice.

Iterative Development

In practice, development is *always iterative*, and all software phases progress in parallel.



✎ *If the waterfall model is pure fiction, why is it still the standard software process?*

In an iterative development process, you start with a minimal set of requirements, carry out a feasibility study, implement a first prototype, and then evaluate it. You then proceed iteratively and incrementally, adding new requirements, extending the system, and delivering a new iteration.

In practice, even organizations that claim to be applying a waterfall model are actually applying an iterative process.

Roadmap

- The iterative software lifecycle
- **Responsibility-driven design**
- TicTacToe example
  - —Identifying objects
  - —Scenarios
  - —Test-first development
  - —Printing object state
  - —Testing scenarios
  - —Representing responsibilities as contracts

What is Responsibility-Driven Design?

Responsibility-Driven Design is

- a method for deriving a software design in terms of *collaborating* objects
- by asking what *responsibilities* must be fulfilled to meet the requirements,
- and assigning them to the appropriate *objects* (i.e., that can carry them out).

RDD starts by identifying potential domain objects. A good candidate for an object in your design has clear *responsibilities*.

Responsibilities consist of (i) what the objects *knows*, and (ii) what it *can do*. In the final design, what an object knows will be represented in its state (instance variables), and what it can do will end up as services (methods). In the initial design, however, we only identify these two aspects in very general terms (e.g., a stack manages a LIFO collection of objects, and lets clients push and pop elements).

In a good design, all objects have clear responsibilities.

How to assign responsibility?

Pelrine's Laws:

- ✔ *"Don't do anything you can push off to someone else."*
- ✔ *"Don't let anyone else play with you."*

RDD leads to fundamentally different designs than those obtained by functional decomposition or data-driven design.

Class responsibilities tend to be more stable over time than functionality or representation.

In RDD, things should happen close to objects with those responsibilities. If an object is responsible for some knowledge, then it should also be responsible for services related to that knowledge. Pelrine's Laws make clear that you should not violate responsibilities in a system.

Roadmap

- The iterative software lifecycle
- Responsibility-driven design
- **TicTacToe example**
  - Identifying objects
  - Scenarios
  - Test-first development
  - Printing object state
  - Testing scenarios
  - Representing responsibilities as contracts

Example: Tic Tac Toe

Requirements:

> *"A simple game in which one player marks down only crosses and another only ciphers [zeroes], each alternating in filling in marks in any of the nine compartments of a figure formed by two vertical lines crossed by two horizontal lines, the winner being the first to fill in three of his marks in any row or diagonal."*
>
> — Random House Dictionary

We should design a program that implements the rules of Tic Tac Toe.

Setting Scope

Questions:

- Should we support other games?
- Should there be a graphical UI?
- Should games run on a network? Through a browser?
- Can games be saved and restored?

A monolithic paper design is bound to be wrong!

An iterative development strategy:

- limit initial scope to the *minimal requirements* that are interesting
- *grow the system* by adding features and test cases
- let the *design emerge by refactoring* roles and responsibilities

✎ How much functionality should you deliver in the first version of a system?

✔ *Select the minimal requirements that provide value to the client.*

These principles are also formalized in XP and other “agile development” methods. Start with a small set of initial requirements that bring value, implement those in a short development cycles, then select new requirements and iterate.

Never start with a “big design” that attempts to take into account all the features that you might need later. Chances are very high that those requirements will change either their nature or their priority before you get to them.

Roadmap

- The iterative software lifecycle
- Responsibility-driven design
- TicTacToe example
  - **Identifying objects**
  - Scenarios
  - Test-first development
  - Printing object state
  - Testing scenarios
  - Representing responsibilities as contracts

Tic Tac Toe Objects

Some objects can be identified from the requirements:

Objects	***Responsibilities***
Game	Maintain game rules
Player	Make moves Mediate user interaction
Compartment	Record marks
Figure (State)	Maintain game state

Entities with clear responsibilities are more likely to end up as objects in our design.

We can identify all these domain objects from the “requirements document” (the definition of the game). Potential objects for our design are those that have some clear responsibilities.

Note that in a good design, no object has too many responsibilities. Clearly we could assign all the responsibilities listed to a single object, e.g, the game, but that would lead to a very centralized, procedural design.

Tic Tac Toe Objects ...

Others can be eliminated:

Non-Objects	***Justification***
Crosses, ciphers	Same as Marks
Marks	Value of Compartment
Vertical lines	Display of State
Horizontal lines	ditto
Winner	State of Player
Row	View of State
Diagonal	ditto

We can also identify domain objects from our requirements document that do not have any identifiable responsibilities. Such objects may be eliminated for various reasons, for example, some are just *different names* for objects we have already seen (crosses = marks), or because they just describe a *state* of an object we already have (winner is a state of a player).

Missing Objects

Now we check if there are unassigned responsibilities:

> Who starts the Game?
> Who is responsible for displaying the Game state?
> How do Players know when the Game is over?

Let us introduce a *Driver* that supervises the Game.

✎ How can you tell if there are objects missing in your design?

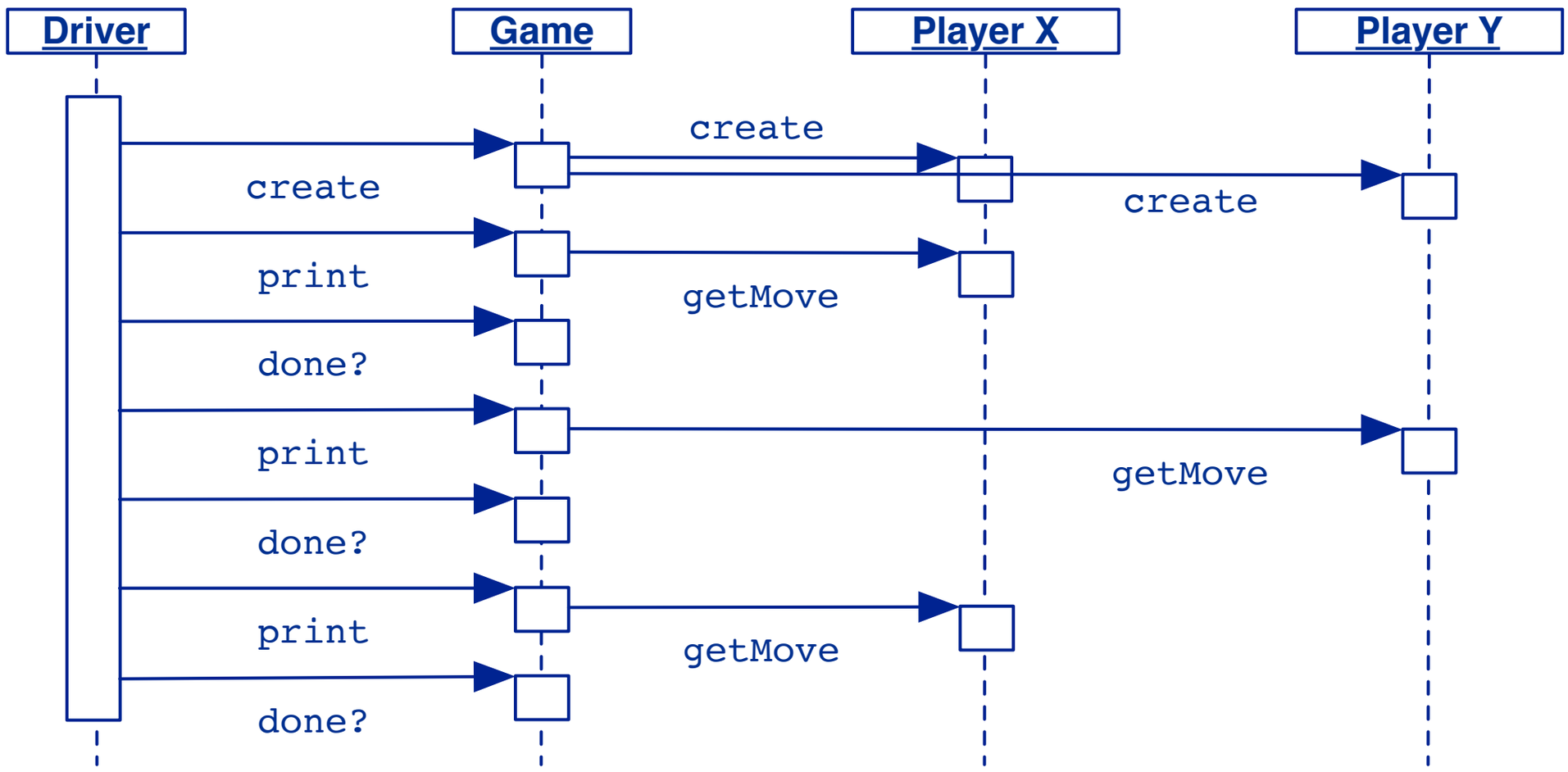
✔ *When there are responsibilities left unassigned.*

Roadmap

- The iterative software lifecycle
- Responsibility-driven design
- TicTacToe example
  - Identifying objects
  - **Scenarios**
  - Test-first development
  - Printing object state
  - Testing scenarios
  - Representing responsibilities as contracts

Scenarios

A scenario describes a typical sequence of interactions:



Are there other equally valid scenarios for this problem?

Here we have another UML sequence diagram. As before, read the diagram from top to bottom: First the driver creates a game object, which in turn creates two player objects.

Afterwards we will revisit this scenario and redesign it when we start to consider new requirements. For the time being, it is “good enough” to start our initial design.

Version 0 — skeleton

Our first version does very little!

```
class GameDriver {
   static public void main(String args[]) {
      TicTacToe game = new TicTacToe();
      do { System.out.print(game); }
      while(game.notOver());
   }
public class TicTacToe {
   public boolean notOver() { return false; }
   public String toString() { return("TicTacToe\n");}
}
```

✎ How do you iteratively "grow" a program?

✔ *Always have a running version of your program.*

We will iteratively build up our TicTacToe game, adding new features with each iteration. After each iteration we will have a functioning game that does something new.

Although this first version is not really useful, we show it to underline the point that every iteration should produce a “complete” running program.

Roadmap

- The iterative software lifecycle
- Responsibility-driven design
- TicTacToe example
  - Identifying objects
  - Scenarios
  - **Test-first development**
  - Printing object state
  - Testing scenarios
  - Representing responsibilities as contracts

Version 1 — game state

- We will use chess notation to access the game state
  - —Columns ‘a’ through ‘c’
  - —Rows ‘1’ through ‘3’

✎ *How do we decide on the right interface?*

✔ *First write some tests!*

Test-first development

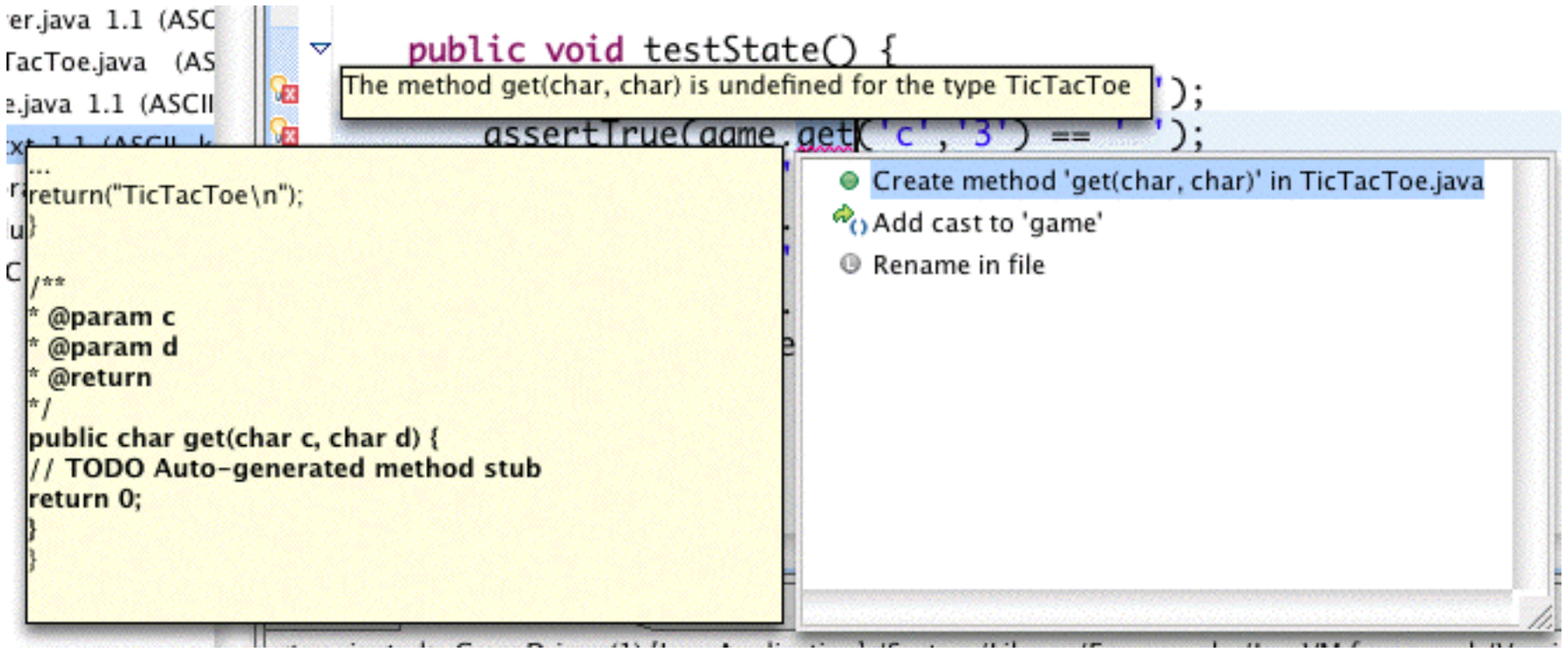
```
public class TicTacToeTest {
   protected TicTacToe game;

   @Before public void setUp() {
      game = new TicTacToe();
   }

   @Test public void testState() {
      assertEquals(game.get('a','1'), ' ');
      assertEquals(game.get('c','3'), ' ');
      game.set('c','3','X');
      assertEquals(game.get('c','3'), 'X');
      game.set('c','3',' ');
      assertEquals(game.get('c','3'), ' ');
      assertFalse(game.inRange('d','4'));
   }
}
```

Here we express that the game board should initially contain just blank squares. We need a way to *set* squares of the board and to *get* their current values. We also need a way to check whether a particular coordinate on the board is valid (in range).

The test express how we want to be able to perform these simple actions with the board. In other words, *designing the test helps us design the interface of the code we are testing*.

Note that we are also testing boundary conditions by setting and getting values at the (literal) boundary of the board.

Generating methods



Test-first programming can drive the development of the class interface …

As before, we can exploit the quick fix feature of Eclipse to generate the methods we need to make the test run.

Roadmap

- The iterative software lifecycle
- Responsibility-driven design
- TicTacToe example
  - Identifying objects
  - Scenarios
  - Test-first development
  - **Printing object state**
  - Testing scenarios
  - Representing responsibilities as contracts

Representing game state

```
public class TicTacToe {
   protected char[][] gameState;
   public TicTacToe() {
      gameState = new char[3][3];
      for (char col='a'; col <='c'; col++)
         for (char row='1'; row<='3'; row++)
            this.set(col,row,' ');
   }
...
```

Checking pre-conditions

set() and get() translate from chess notation to array indices.

```
public void set(char col, char row, char mark) {
   assert(inRange(col, row));    // precondition
   gameState[col-'a'][row-'1'] = mark;
}
public char get(char col, char row) {
   assert(inRange(col, row));
   return gameState[col-'a'][row-'1'];
}
public boolean inRange(char col, char row) {
   return (('a'<=col) && (col<='c')
            && ('1'<=row) && (row<='3'));
}
```

Printing the State

By re-implementing `TicTacToe.toString()`, we can view the state of the game:

```
3     |     |
   ---+---+---
2     |     |
   ---+---+---
1     |     |
      a     b     c
```

✎ How do you make an object printable?

✔ *Override `Object.toString()`*

NB: By overriding the `toString()` method, you will also obtain more useful feedback within the debugger, as the built-in object inspector will use this method to display the state of objects.

TicTacToe.toString()

Use a `StringBuilder` (not a `String`) to build up the representation:

```
public String toString() {
   StringBuffer rep = new StringBuffer();
   for (char row='3'; row>='1'; row--) {
      rep.append(row);
      rep.append("   ");
      for (char col='a'; col <='c'; col++) { ... }
      ...
   }
   rep.append("   a   b   c\n");
   return(rep.toString());
}
```

The String class in Java is *immutable*, that is, instances once created cannot change their state. Immutable classes are a common design idiom to improve readability and runtime efficiency. They are also inherently *thread-safe*, that is, they can safely be shared by concurrent threads, since no thread can modify their values.

Typically immutable objects support operators to create new instances from combinations of existing ones. For example, in Java the `+` operator will create a new string from two existing ones.

Building up a complex string through repeated use of this operator is not very efficient, however, which is why the `StringBuffer` class is available for this purpose.

Roadmap

- The iterative software lifecycle
- Responsibility-driven design
- TicTacToe example
  - Identifying objects
  - Scenarios
  - Test-first development
  - Printing object state
  - **Testing scenarios**
  - Representing responsibilities as contracts

Version 2 — adding game logic

We will:

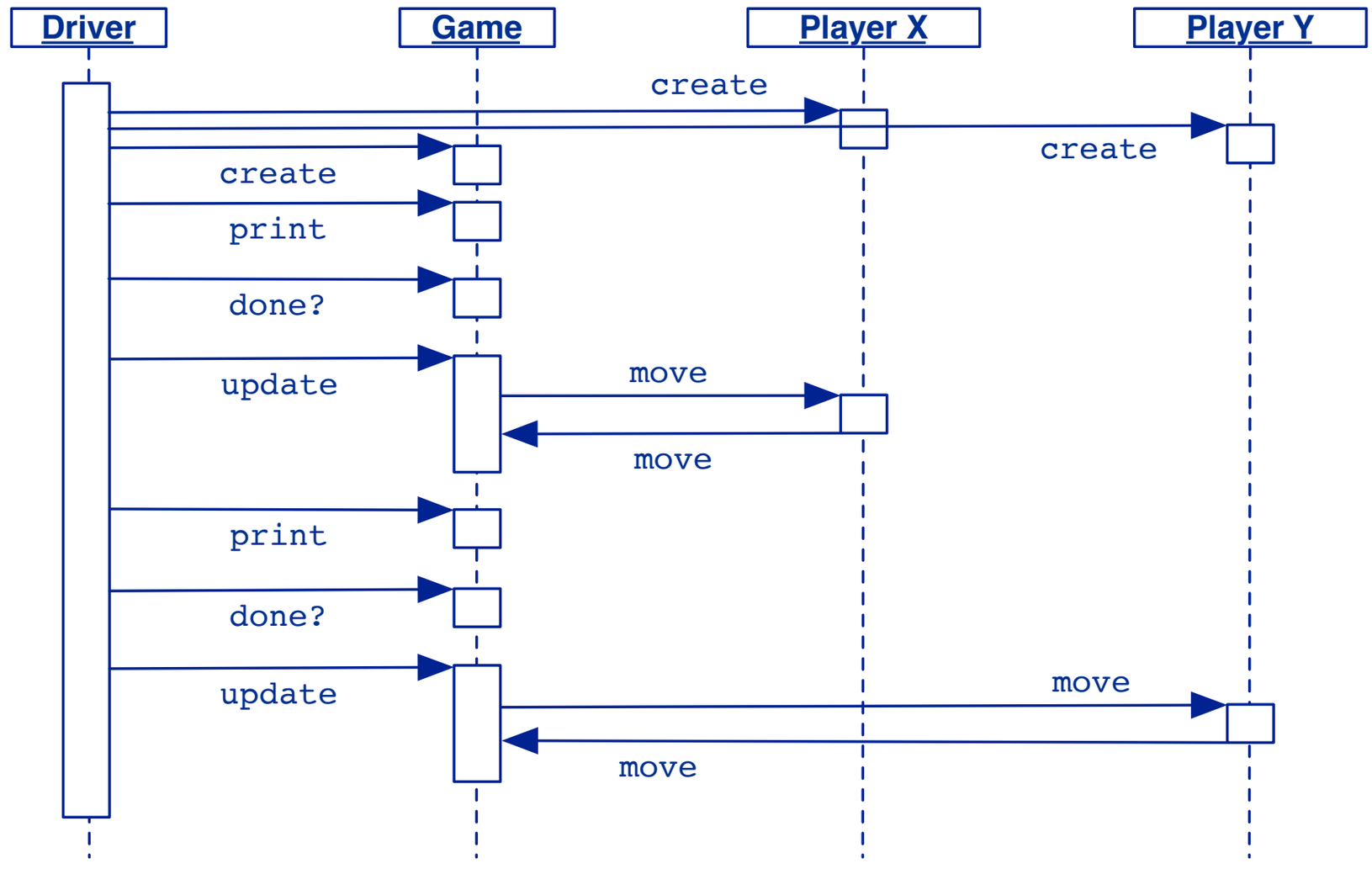
- Add test scenarios
- Add Player class
- Add methods to make moves, test for winning

Refining the interactions

We will want both real and test Players, *so the Driver should create them.*

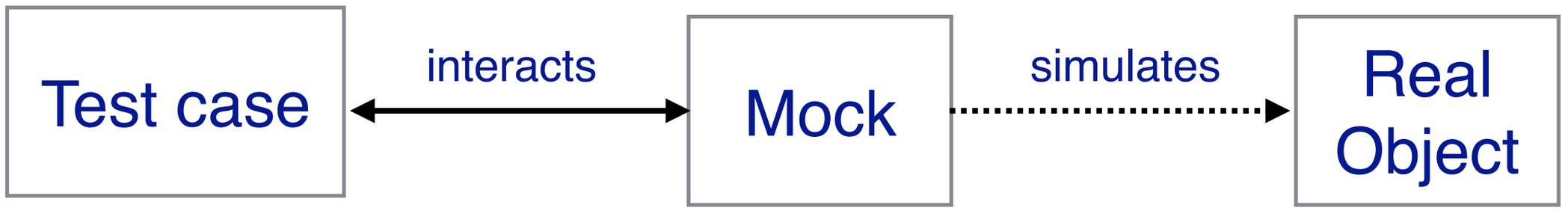
Updating the Game and printing it *should be separate operations.*

The Game should ask the Player to make a move, and *then the Player will attempt to do so.*



Mock Objects

A *mock object* simulates the behaviour of a real object in a controlled way to support automated testing.



To simulate a scripted game, we can use a “mock” Player that plays a *fixed sequence* of moves.

A mock object implements the interface of a real object and replaces it in the context of a test case to simulate a deterministic scenario in a cheap and controlled way. A mock object can simulate a database, a user, a random environment, etc.

NB: Martin Fowler distinguishes between mock objects, stubs, and other kinds of test objects that double for a real object:

http://martinfowler.com/articles/mocksArentStubs.html

Testing scenarios

Our test scenarios will play and test *scripted* games

```
@Test public void testXWinDiagonal() {
   checkGame("a1\nb2\nc3\n", "b1\nc1\n", "X", 4);
}
// more tests …

public void checkGame(String Xmoves, String Omoves,
      String winner, int squaresLeft) {
   Player X = new Player('X', Xmoves); // a scripted player
   Player O = new Player('O', Omoves);
   TicTacToe game = new TicTacToe(X, O);
   GameDriver.playGame(game);
   assertEquals(winner, game.winner().name());
   assertEquals(squaresLeft, game.squaresLeft());
}
```

The `checkGame()` method will run a scripted game, with moves for players X and Y. The scripts each consist of a string containing multiple lines, one for each move of a given player.

The mock Players are instantiated, and the game is played to the end. The test method checks if the expected player wins, and if the the expected number of squares are left unoccupied.

Note how the mock Players are injected into the game object: at no point is the game aware that the players are not real objects.

Running the test cases

```
   |   |
---+---+---
   |   |
---+---+---
   |   |
   a   b   c
Player X moves: X at a1
   |   |
---+---+---
   |   |
---+---+---
 X |   |
   a   b   c
...
```

```
Player O moves: O at c1
   |   |
---+---+---
   | X |
---+---+---
 X | O | O
   a   b   c
Player X moves: X at c3
   |   | X
---+---+---
   | X |
---+---+---
 X | O | O
   a   b   c
game over!
```

Although it may be amusing to see the output of the test game, tests are supposed to be *silent*, and only report success or failure. Later we shall see how to suppress the test output.

The Player

We use *different constructors* to make real or test Players:

```
public class Player {
    protected final char mark;
    protected final BufferedReader in;
```

A real player reads from the standard input stream:

```
    public Player(char mark) {
        this(mark, new BufferedReader(
                new InputStreamReader(System.in)
        ));
    }
```

This constructor just calls another one ...

...

Note the call to `this(…)` within the constructor. This is a feature peculiar to Java that allows one constructor to call another without invoking `new`.

In the code of the `Player`, we want to be able to read a line of input at a time, either from a script, or from the standard terminal input object, `System.in`.

To read a line at a time, it turns out we need a `BufferedReader`, but `System.in` does not support this interface. As it turns out, we can *wrap* `System.in` as an `InputStreamReader`, and then use that to instantiate the `BufferedReader` that we need.

This is a good example of the Adapter design pattern.

Player constructors ...

But a Player can be constructed that reads its moves from any input buffer:

```
protected Player(char initMark, BufferedReader initIn) {
        mark = initMark;
        in = initIn;
}
```

This constructor is not intended to be called directly.

...

Constructors are normally public, since they are intended to be used by clients to instantiate objects. The use of a protected constructor is unusual, but very useful. In this case we avoid writing the same code twice, and can share this code between the other public constructors. By declaring the constructor protected, we prevent external clients from using it to directly instantiate objects.

Player constructors ...

A test Player gets its input from a String buffer:

```
public Player(char mark, String moves) {
   this(mark, new BufferedReader(
              new StringReader(moves)
   ));
}
```

The default constructor returns a dummy Player representing “nobody”

```
public Player() { this(' '); }
```

The fact that:

```
new Player(‘ ‘);
```

creates a player representing “nobody” is perhaps somewhat obscure. How could you use inheritance to make the intent more clear?

Roadmap

- The iterative software lifecycle
- Responsibility-driven design
- TicTacToe example
  - Identifying objects
  - Scenarios
  - Test-first development
  - Printing object state
  - Testing scenarios
  - **Representing responsibilities as contracts**

Tic Tac Toe Contracts

Explicit invariants:

- turn (current player) is either X or O
- X and O swap turns (turn never equals previous turn)
- game state is 3×3 array marked X, O or blank
- winner is X or O iff winner has three in a row

Implicit invariants:

- initially winner is nobody; initially it is the turn of X
- game is over when all squares are occupied, or there is a winner
- a player cannot mark a square that is already marked

Contracts:

- the current player may make a move, if the invariants are respected

Encoding the contract

We must introduce state variables to implement the contracts

```
public class TicTacToe {
   static final int X = 0;                          // constants
   static final int O = 1;
   protected char[][] gameState;
   protected Player winner = new Player();          // = nobody
   protected Player[] player;
   protected int turn = X;                          // initial turn
   protected int squaresLeft = 9;
...
```

Supporting test Players

The Game no longer instantiates the Players, but accepts them as constructor arguments:

```
public TicTacToe(Player playerX, Player playerO)
{   // ...
    player = new Player[2];
    player[X] = playerX;
    player[O] = playerO;
}
```

Invariants

These conditions may seem obvious, which is exactly why they should be checked ...

```
protected boolean invariant() {
    return (turn == X || turn == O)
        && ( this.notOver()
           || this.winner() == player[X]
           || this.winner() == player[O]
           || this.winner().isNobody())
        && (squaresLeft < 9                  // else, initially:
           || turn == X && this.winner().isNobody());
}
```

Assertions and tests often tell us what methods should be implemented, and whether they should be public or protected.

Delegating Responsibilities

When Driver updates the Game, the Game just asks the Player to make a move:

```
public void update() throws IOException {
    player[turn].move(this);
}
```

Note that the Driver may not do this directly!

...

Delegating Responsibilities ...

The Player, in turn, calls the Game's move() method:

```
public void move(char col, char row, char mark) {
    assert(notOver());
    assert(inRange(col, row));
    assert(get(col, row) == ' ');
    System.out.println(mark + " at " + col + row);
    this.set(col, row, mark);
    this.squaresLeft--;
    this.swapTurn();
    this.checkWinner();
    assert(invariant());
}
```

Note how the the use of Design by Contract simplifies our code. We do not have to implement any special logic here to handle possible errors, but we simply assert our pre-conditions and then execute straight-line code.

Since this code modifies the game's state, we also check that our invariant holds at the end.

Small Methods

Introduce methods that make the *intent* of your code clear.

```
public boolean notOver() {
    return this.winner().isNobody()
                    && this.squaresLeft() > 0;
}
protected void swapTurn() {
    turn = (turn == X) ? O : X;
}
```

Well-named variables and methods typically eliminate the need for explanatory comments!

Accessor Methods

Accessor methods protect clients from changes in implementation:

```
public Player winner() {
   return this.winner;
}
public int squaresLeft() {
   return this.squaresLeft;
}
```

✎ When should instance variables be public?

✔ *Almost never! Declare public accessor methods instead.*

getters and setters in Java

Accessors in Java are known as “getters” and “setters”.

- Accessors for a variable `x` should normally be called `getx()` and `setx()`

Frameworks such as EJB depend on this convention!

Code Smells — TicTacToe.checkWinner()

✎ *Duplicated code stinks!*
How can we clean it up?

```
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;
}
}
```

This code is long and repetitive. Here we look for a winning row, column or diagonal, each time checking three squares in sequence.

Right now it is not obvious how to simplify the repetitive code. Later, when we generalize the game, we will see that we can solve the problem in a much more elegant way …

GameDriver

In order to run test games, we separated Player instantiation from Game playing:

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

✎ *How can we make test scenarios play silently?*

What you should know!

- *What is Iterative Development, and how does it differ from the Waterfall model?*
- *How can identifying responsibilities help you to design objects?*
- *Where did the Driver come from, if it wasn't in our requirements?*
- *Why is Winner not a likely class in our TicTacToe design?*
- *Why should we evaluate assertions if they are all supposed to be true anyway?*
- *What is the point of having methods that are only one or two lines long?*

Can you answer these questions?

- ✎ *Why should you expect requirements to change?*
- ✎ *In our design, why is it the Game and not the Driver that prompts a Player to move?*
- ✎ *When and where should we evaluate the TicTacToe invariant?*
- ✎ *What other tests should we put in our TestDriver?*
- ✎ *How does the Java compiler know which version of an overloaded method or constructor should be called?*