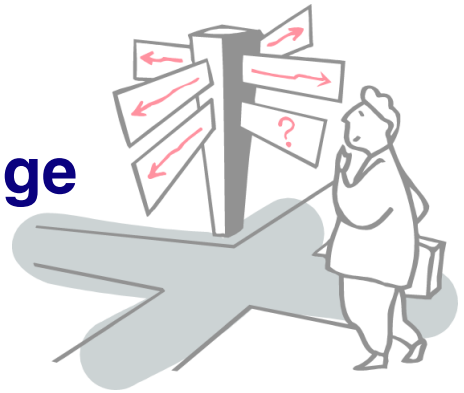
2. Object-Oriented Design Principles

Roadmap

- Motivation: stability in the face of change
- Model domain objects
- Model responsibilities
- Separate interface and implementation
- Delegate responsibility
- Let the code talk
- Recognize Code Smells

Roadmap



- **Motivation: stability in the face of change**
- Model domain objects
- Model responsibilities
- Separate interface and implementation
- Delegate responsibility
- Let the code talk
- Recognize Code Smells

Motivation

The law of continuing change:
A large program that is used undergoes continuing change or becomes progressively less useful. The change process continues until it is judged more cost-effective to replace the system with a recreated version.

— Lehman and Belady, 1985

What should design optimize?

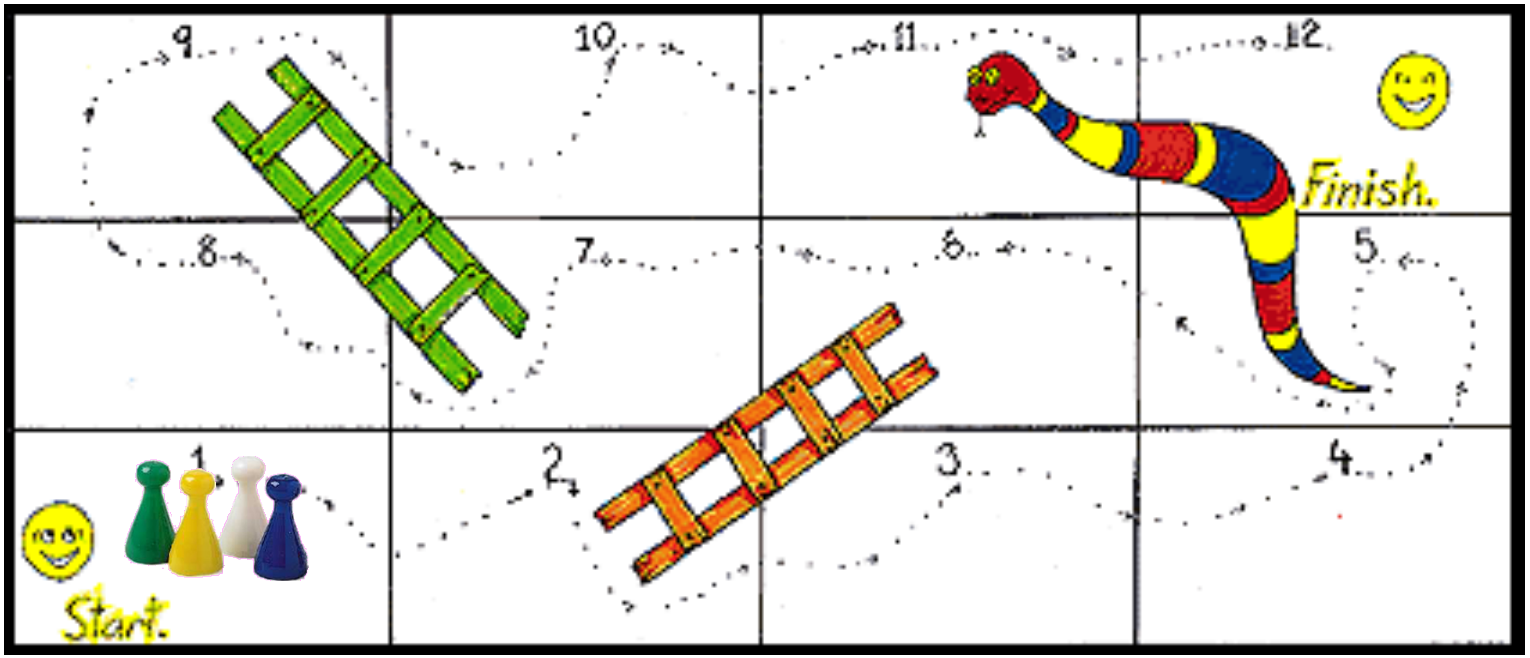
*Enable small, incremental changes by designing software around **stable abstractions** and **interchangeable parts**.*

How do we find the “right” design?

Object-oriented design is an *iterative and exploratory process*

Don’t worry if your initial design is ugly.
If you apply the OO design principles consistently, your final design will be beautiful!

Running Example: Snakes and Ladders



http://en.wikipedia.org/wiki/Snakes_and_ladders

Game rules

- **Players**
  - Snakes and Ladders is played by **two to four players**, each with her own token to move around the board.
- **Moving**
  - Players **roll a die** or spin a spinner, then **move the designated number of spaces**, between one and six. Once they land on a space, they have to perform any action designated by the space.
- **Ladders**
  - If the space a player lands on is at the bottom of a ladder, he should **climb the ladder**, which brings him to a space higher on the board.
- **Snakes**
  - If the space a player lands on is at the top of a snake, she **must slide down to the bottom** of it, landing on a space closer to the beginning.
- **Winning**
  - The winner is the player **who gets to the last space on the board first**, whether by landing on it from a roll, or by reaching it with a ladder.

http://www.ehow.com/facts_5163203_snakes-amp-ladders-rules.html

Variations

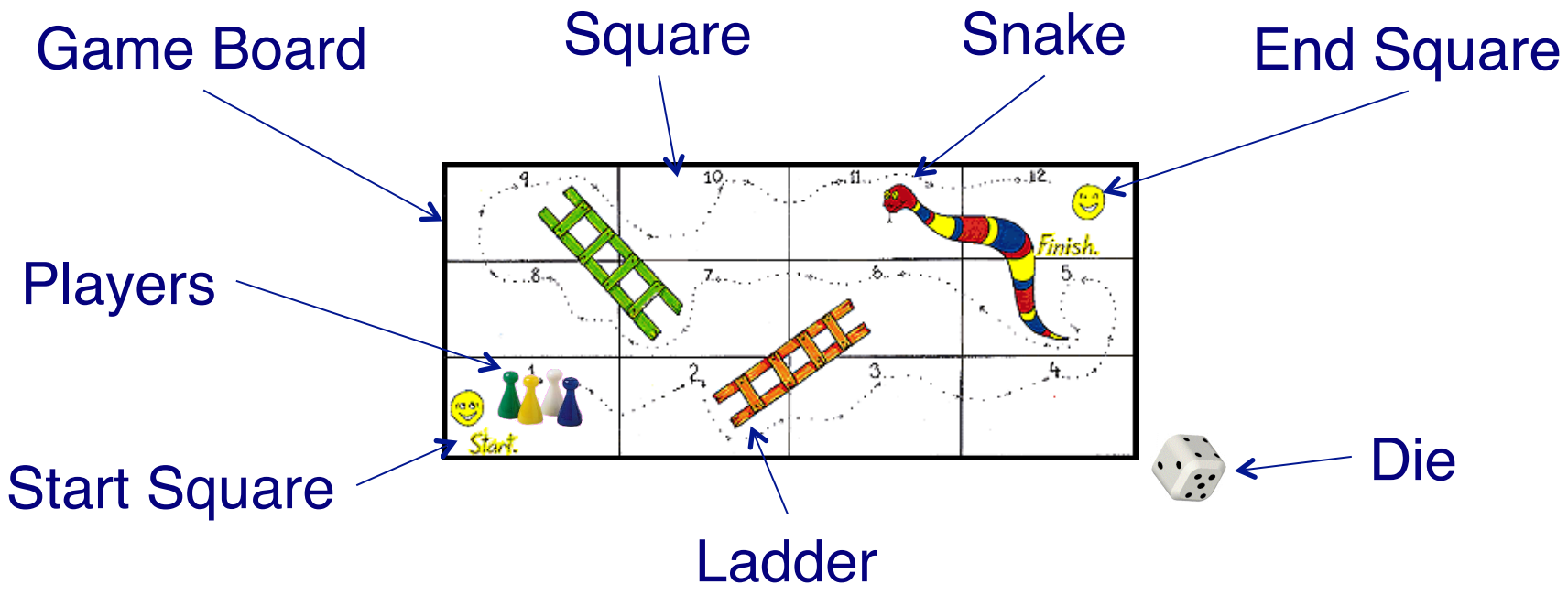
- A player who lands on an occupied square must go back to the start square.
- If you roll a number higher than the number of squares needs to reach the last square, you must continue moving backwards.
- …

Roadmap

- Motivation: stability in the face of change
- **Model domain objects**
- Model responsibilities
- Separate interface and implementation
- Delegate responsibility
- Let the code talk
- Recognize Code Smells

Programming is modeling

Model domain objects



What about roll, action, winner … ?

Everything is an object

Every domain concept that *plays a role* in the application and *assumes a responsibility* is a potential object in the software design

"Winner" is just a state of a player — it has no responsibility of its own.

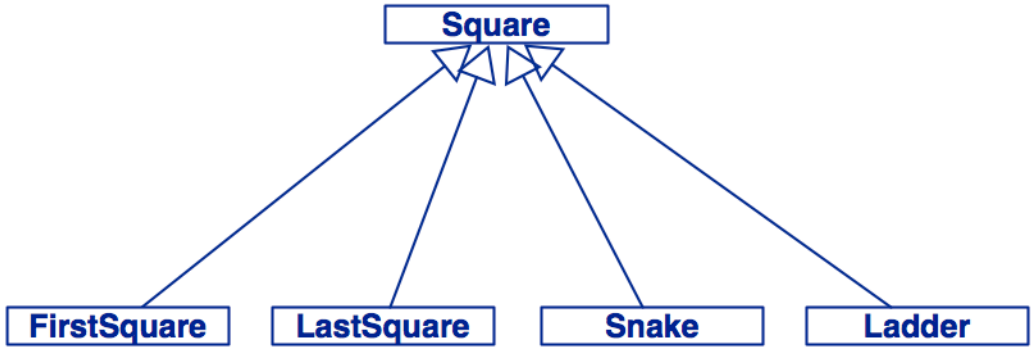
Computation is simulation

*"Instead of a bit-grinding processor … plundering data structures, we have a universe of **well-behaved objects that courteously ask each other** to carry out their various desires."*

— Ingalls 1981

Model specialization

The first square *is a* kind of square, so model it as such



Is a snake a kind of reverse ladder?

Roadmap

- Motivation: stability in the face of change
- Model domain objects
- **Model responsibilities**
- Separate interface and implementation
- Delegate responsibility
- Let the code talk
- Recognize Code Smells

Responsibility-Driven Design

Well-designed objects have clear responsibilities

Drive design by asking:

- What *actions* is this object responsible for?
- What *information* does this object share?

Responsibility-driven design ... minimizes the rework required for major design changes.
— Wirfs-Brock, 1989

Snakes and Ladders responsibilities

Player
- keeps track of where it is
- moves over squares of the board

Game
- keeps track of the game state

Square
- keeps track of any player on it

Die
- provides a random number from 1 to 6

First Square
- can hold multiple players

Last Square
- knows it is the winning square

Snake
- sends a player back to an earlier square

Ladder
- sends a player ahead to a later square

The Single Responsibility Principle

An object should have no more than one key responsibility.

If an object has several, unrelated responsibilities, then you are missing objects in your design!

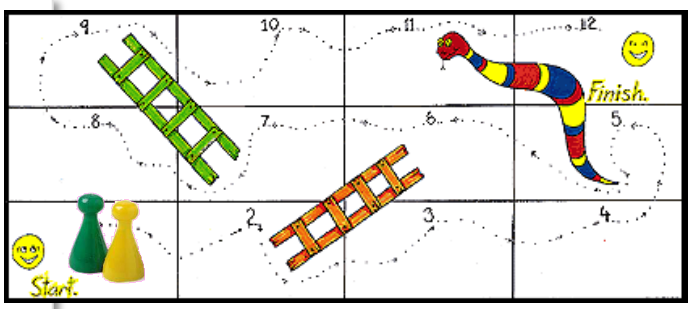
The different kinds of squares have separate responsibilities, so they must belong to separate classes!

http://en.wikipedia.org/wiki/Single_responsibility_principle

Top-down decomposition

Use concrete scenarios to drive interface design

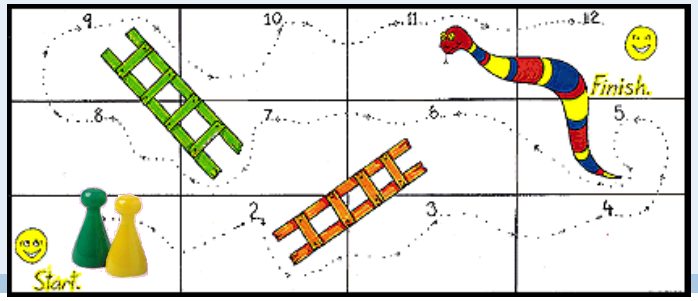
```
jack = new Player("Jack");
jill = new Player("Jill");
Player[] args = { jack, jill };
Game game = new Game(12, args);
game.setSquareToLadder(2, 4);
game.setSquareToLadder(7, 2);
game.setSquareToSnake(11, -6);
assertTrue(game.notOver());
assertTrue(game.firstSquare().isOccupied());
assertEquals(1, jack.position());
assertEquals(1, jill.position());
assertEquals(jack, game.currentPlayer());
```

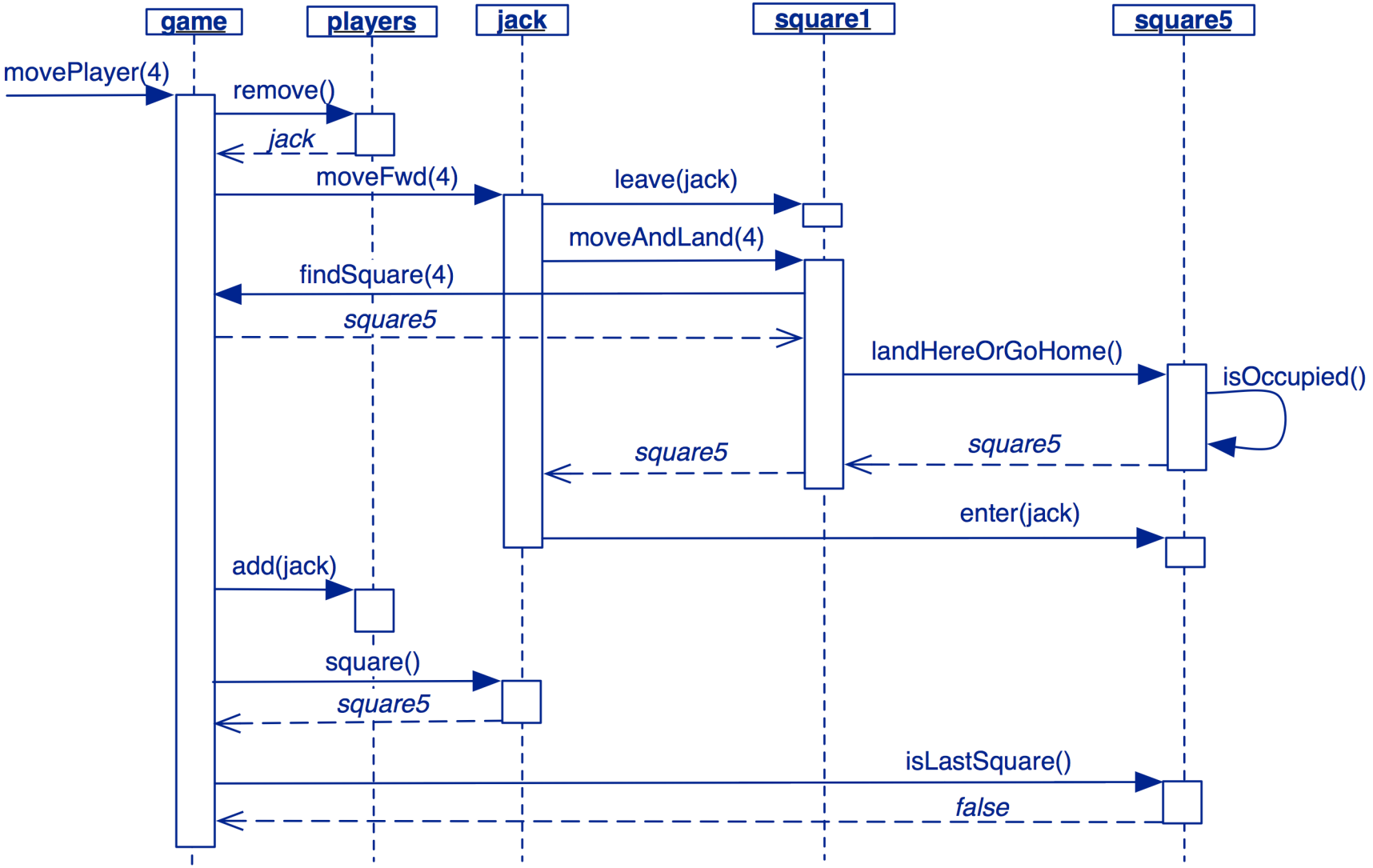



```
game.movePlayer(4);
assertTrue(game.notOver());
assertEquals(5, jack.position());
assertEquals(1, jill.position());
assertEquals(jill, game.currentPlayer());
```

http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design

Jack makes a move





Roadmap

- Motivation: stability in the face of change
- Model domain objects
- Model responsibilities
- **Separate interface and implementation**
- Delegate responsibility
- Let the code talk
- Recognize Code Smells

Separate interface and implementation

Information hiding: a component should provide *all* and *only* the information that the user needs to effectively use it.

Information hiding protects *both* the provider and the client from changes in the implementation.

http://en.wikipedia.org/wiki/Information_hiding

Abstraction, Information Hiding and Encapsulation

Abstraction = *elimination of inessential detail*

Information hiding = *providing only the information a client needs to know*

Encapsulation = *bundling operations to access related data as a data abstraction*

*In object-oriented languages we can implement **data abstractions** as classes.*

Encapsulate state

```
public class Game {
  private List<ISquare> squares;
  private int size;
  private Queue<Player> players;
  private Player winner;
  ...
}
```

```
public class Player {
  private String name;
  private ISquare square;
  ...
}
```

```
public class Square implements ISquare {
  protected int position;
  protected Game game;
  private Player player;
  ...
}
```

Don't let anyone else play with you.
— Joseph Pelrine

Keep behaviour close to state

```
public class Square implements ISquare {
  private Player player;

  public boolean isOccupied() {
    return player != null;
  }

  public void enter(Player player) {
    this.player = player;
  }

  public void leave(Player _) {
    this.player = null;
  }
  ...
}
```

Program to an interface, not an implementation

Depend on interfaces, not concrete classes

```
public interface ISquare {
  public int position();
  public ISquare moveAndLand(int moves);
  public boolean isFirstSquare();
  public boolean isLastSquare();
  public void enter(Player player);
  public void leave(Player player);
  public boolean isOccupied();
  public ISquare landHereOrGoHome();
}
```

```
public class Player {
  private ISquare square;
  public void moveForward(int moves) {
    square.leave(this);
    square = square.moveAndLand(moves);
    square.enter(this);
  }...
}
```

Players do not need to know all the different kinds of squares …

Aside: Messages and methods

Objects *send messages* to one another; they don't "call methods"

```
public class Square implements ISquare {
  private Player player;

  public void enter(Player player) {
    this.player = player;
  }
  ...
}
```

```
public class FirstSquare extends Square {
  private List<Player> players;

  public void enter(Player player) {
    players.add(player);
  }
  ...
}
```

Clients should not care what kind of square they occupy

The Open-Closed Principle

Make software entities open for extension but closed for modifications.

```
public class Square implements ISquare {
 public ISquare moveAndLand(int moves) {
  return game.findSquare(position, moves).landHereOrGoHome();
 }
 public ISquare landHereOrGoHome() {
  return this.isOccupied() ? game.firstSquare() : this ;
 }
 ...
}
```

```
public class Ladder extends Square {
  public ISquare landHereOrGoHome() {
    return this.destination().landHereOrGoHome();
  }
  protected ISquare destination() {
    return game.getSquare(position+transport);
  }
  ...
}
```

http://en.wikipedia.org/wiki/Open/closed_principle

Why are data abstractions important?

Communication — Declarative Programming

- Data abstractions …
  - State what a client *needs to know*, and no more!
  - State *what you want to do*, not how to do it!
  - Directly *model your problem domain*

Software Quality and Evolution

- Data abstractions …
  - Decompose a system into *manageable parts*
  - Protect clients from *changes* in implementation
  - Encapsulate client/server *contracts*
  - Can *extend their interfaces* without affecting clients
  - Allow new *implementations to be added* transparently to a system

Roadmap

- Motivation: stability in the face of change
- Model domain objects
- Model responsibilities
- Separate interface and implementation
- **Delegate responsibility**
- Let the code talk
- Recognize Code Smells

Delegate responsibility

"Don't do anything you can push off to someone else."
— Joseph Pelrine

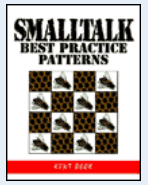
```
public class Player {
  public void moveForward(int moves) {
    square.leave(this);
    square = square.moveAndLand(moves);
    square.enter(this);
  }
  ...
}
```

Responsibility implies non-interference.
— Timothy Budd

```
public class Square implements ISquare {
  public ISquare moveAndLand(int moves) {
    return game.findSquare(position, moves)
                              .landHereOrGoHome();
  }
  ...
}
```

```
public class Game {
  public ISquare findSquare(...) {
    ...
    return this.getSquare(target);
  }
  ...
}
```

Lots of short methods



Once and only once

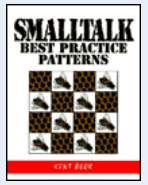
"In a program written with good style, everything is said once and only once."

Lots of little pieces

"Good code invariably has small methods and small objects. Only by factoring the system into many small pieces of state and function can you hope to satisfy the 'once and only once' rule."

http://c2.com/cgi/wiki?LotsOfShortMethods

Composed Method



Divide your program into methods that perform one identifiable task.

- Keep all of the operations in a method at the *same level of abstraction*.
- This will naturally result in programs with *many small methods, each a few lines long*.

Maintain a consistent level of abstraction …

```
public class Game {
  public void play(Die die) {
    System.out.println("Initial state: " + this);
    while (this.notOver()) {
      int roll = die.roll();
      System.out.println(this.currentPlayer()
        + " rolls " + roll + ":  " + this);
      this.movePlayer(roll);
    }
    System.out.println("Final state:   " + this);
    System.out.println(this.winner() + " wins!");
  }
  ...
```

… to obtain many small methods

```
public boolean notOver() {
  return winner == null;
}
```

```
public Player currentPlayer() {
  return players.peek();
}
```

```
public void movePlayer(int roll) {
  Player currentPlayer = players.remove(); // from front of queue
  currentPlayer.moveForward(roll);
  players.add(currentPlayer);              // to back of the queue
  if (currentPlayer.wins()) {
    winner = currentPlayer;
  }
}
```

```
public Player winner() {
  return winner;
}
```

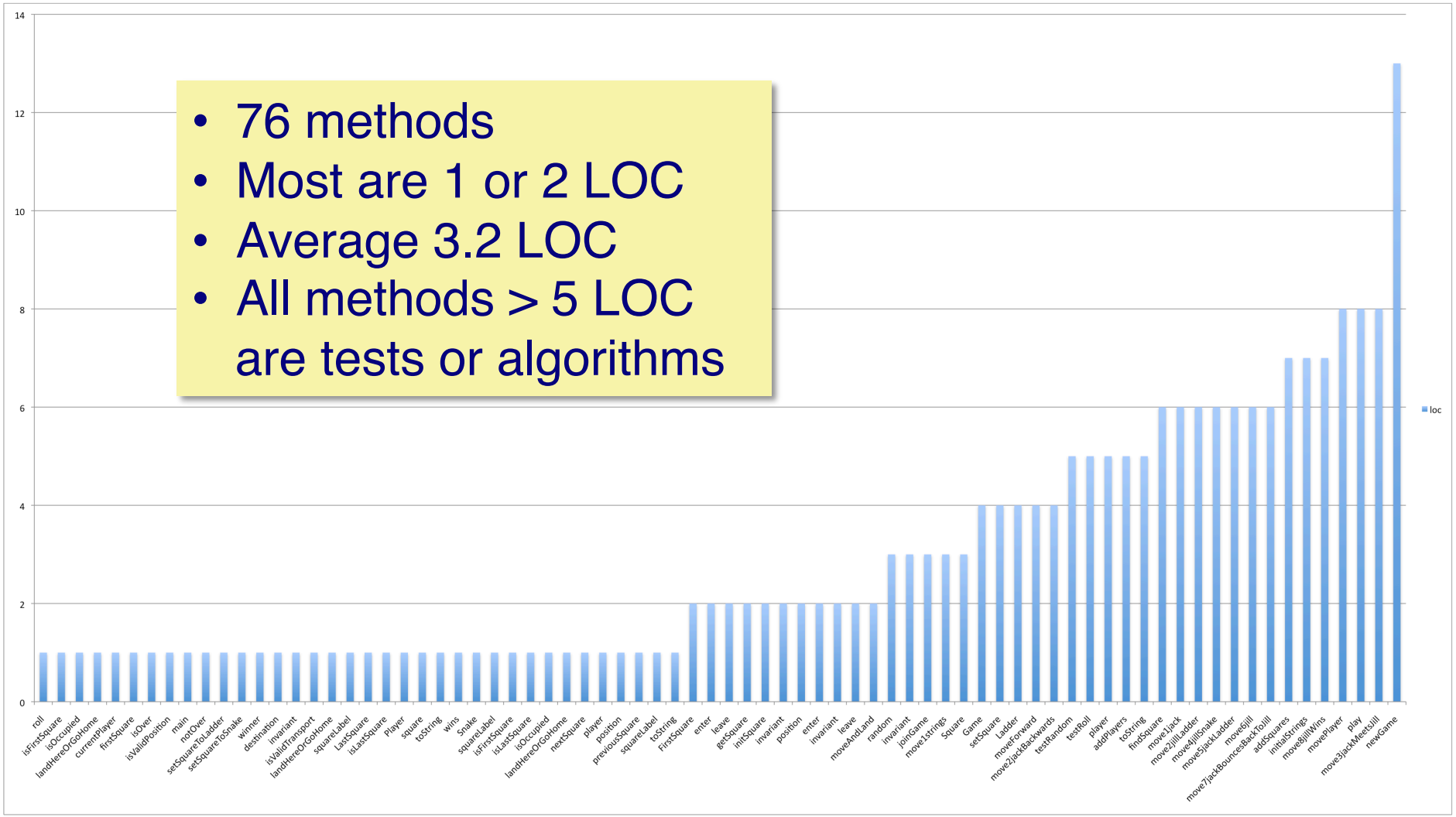
… and simple classes

```
public class Die {
  static final int MIN = 1;
  static final int MAX = 6;

  public int roll() {
    return this.random(MIN,MAX);
  }

  public int random(int min, int max) {
    int result = (int) (min + Math.floor((max-min) * Math.random()));
    return result;
  }
}
```

Snakes and Ladders methods



- 76 methods
- Most are 1 or 2 LOC
- Average 3.2 LOC
- All methods > 5 LOC are tests or algorithms

Design by Contract = Don't accept anybody else's garbage!

```
public class Game {
  public void movePlayer(int roll) {
    assert roll>=1 && roll<=6;
    ...
  }
  ...
}
```

```
public class Player {
  public void moveForward(int moves) {
    assert moves > 0;
    ...
  }
  ...
}
```

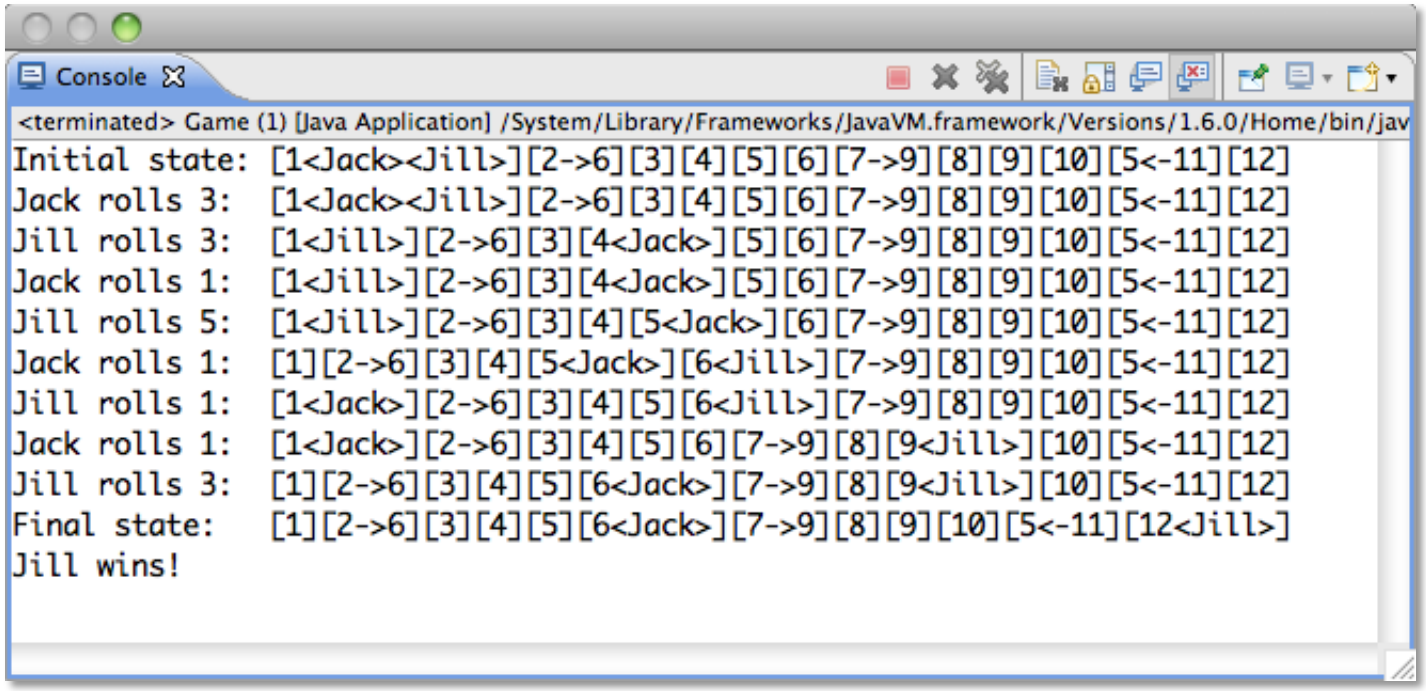
```
public class Square implements ISquare {
  public ISquare moveAndLand(int moves) {
    assert moves >= 0;
    ...
  }
  ...
}
```

More on this in the following lecture

http://en.wikipedia.org/wiki/Design_by_contract

Demo

```
public static void main(String args[]) {
  (new SimpleGameTest()).newGame().play(new Die());
}
```



Roadmap

- Motivation: stability in the face of change
- Model domain objects
- Model responsibilities
- Separate interface and implementation
- Delegate responsibility
- **Let the code talk**
- Recognize Code Smells

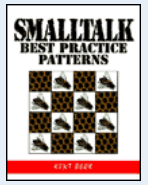
Program declaratively

Name objects and methods so that code documents itself

```
public class Player {
  public void joinGame(Game game) {
    square = game.getSquare(1);
    ((FirstSquare) square).players().add(this);
  }
  ...
}
```

```
public class Player {
  public void joinGame(Game game) {
    square = game.firstSquare();
    square.enter(this);
  }
  ...
}
```

Role Suggesting Instance Variable Name

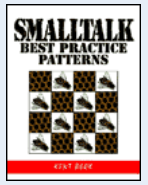


Name instance variables for the role they play in the computation.

```
public class Game {
  private List<ISquare> squares;
  private int size;
  private Queue<Player> players;
  private Player winner;
  ...
}
```

Make the name plural if the variable will hold a collection.

Intention Revealing Method Name




```
public class Player {
  public void moveForward(int moves) {
    ...
    square.enter(this);
  }
  ...
}
```

*Name methods after **what** they accomplish, not how.*

```
public class Square implements ISquare {
  private Player player;
  public void enter(Player player) {
    this.player = player;
  }
  ...
}
```

```
public class FirstSquare extends Square {
  private List<Player> players;
  public void enter(Player player) {
    players.add(player);
  }
  ...
}
```

Roadmap

- Motivation: stability in the face of change
- Model domain objects
- Model responsibilities
- Separate interface and implementation
- Delegate responsibility
- Let the code talk
- **Recognize Code Smells**

The Law of Demeter: "Do not talk to strangers"

Don't send messages to objects returned from other message sends

```
public void movePlayer(int roll) {
  ...
  if (currentPlayer.square().isLastSquare()) {
    winner = currentPlayer;
  }
}
```

```
public void movePlayer(int roll) {
  ...
  if (currentPlayer.wins()) {
    winner = currentPlayer;
  }
}
```

Tell, don't ask

en.wikipedia.org/wiki/Law_Of_Demeter

Be sensitive to Code Smells

- **Duplicated Code**
  - Missing inheritance or delegation
- **Long Method**
  - Inadequate decomposition
- **Large Class / God Class**
  - Too many responsibilities
- **Long Parameter List**
  - Object is missing
- **Feature Envy**
  - Method needing too much information from another object
- **Data Classes**
  - Only accessors

http://en.wikipedia.org/wiki/Code_smell

Conclusions and outlook

- **Use responsibility-driven design** to stabilize domain concepts
- **Delegate responsibility** to achieve simple, flexible designs

- *Specify contracts* to protect your data abstractions
  - Design by Contract lecture
- *Express your assumptions* as tests to tell what works and doesn't
  - Testing Framework lecture
- *Develop iteratively and incrementally* to allow design to emerge
  - Iterative Development lecture
- *Encode specialization hierarchies* using inheritance
  - Inheritance lecture

What you should know!

- *Why does software change?*
- *Why should software model domain concepts?*
- *What is responsibility-driven design?*
- *How do scenarios help us to design interfaces?*
- *What is the difference between abstraction, encapsulation and information hiding?*
- *Can you explain the Open-Closed principle?*
- *How can delegation help you write declarative code?*
- *How should you name methods and instance variables?*

Can you answer these questions?

- *How do you identify responsibilities?*
- *How can we use inheritance to model the relationship between Snakes and Ladders?*
- *How can we tell if an object has too many responsibilities?*
- *Is top-down design better than bottom-up design?*
- *Why should methods be short?*
- *How does the Law of Demeter help you to write flexible software?*
- *Why do “God classes” and Data classes often occur together?*

License

http://creativecommons.org/licenses/by-sa/3.0/