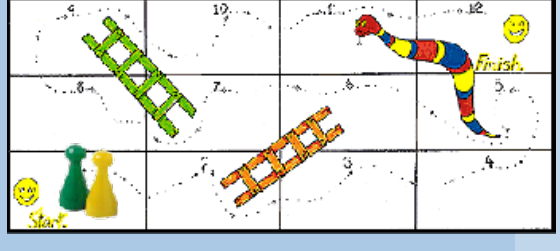
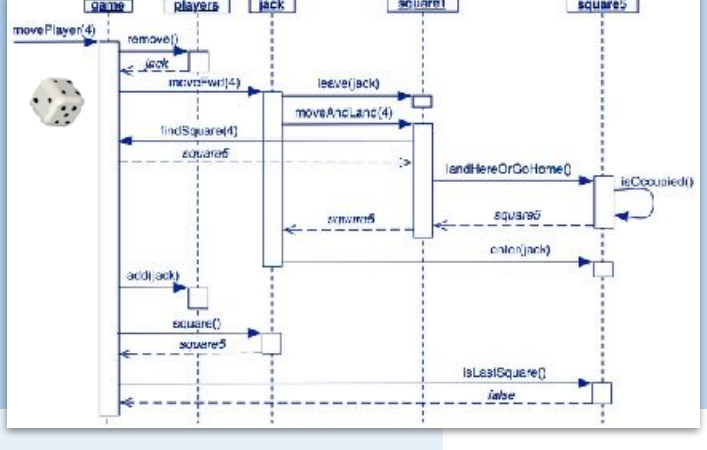
UNIVERSITÄT BERN

2. Object-Oriented Design Principles

Oscar Nierstrasz





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

Lehmann and Belady studied the evolution of industrial software systems in the early 80s and formulated several “laws of software evolution”. Even though software technology has changed, these laws are generally considered to hold true today.

Systems that are used in a real environment are always asked to do more because business changes.

Nowadays, more effort is spent in “maintenance” (i.e. after deployment) than in initial development.

Manny Lehman and Les Belady. Program Evolution: Processes of Software Change, p. 538, London Academic Press, London, 1985.

What should design optimize?

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

We want changes to be easy to make — small, incremental changes should be well-localized, easy to understand and verify.

Consequently we need to build software around stable abstractions.

Data (implementation) tends to change, so what do we do?

Instead we should focus on *domain concepts* and *responsibilities*. This is the key idea behind object-oriented design.

How do we find the “right” design?

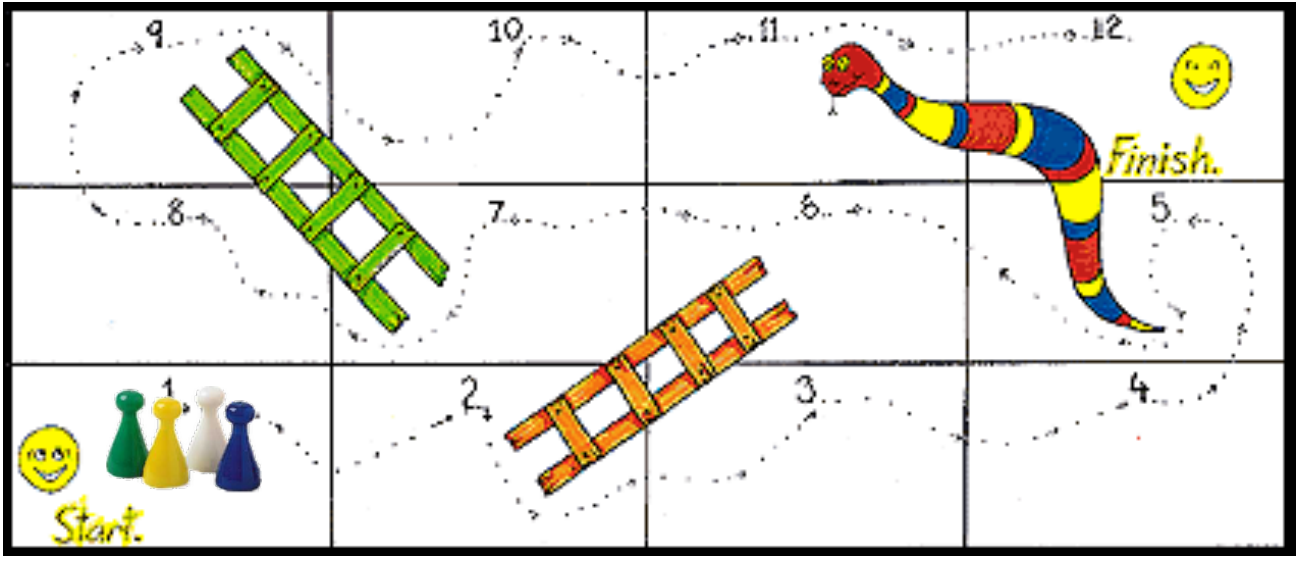
Object-oriented design is an *iterative and exploratory process*

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

The examples that we show in this lecture are the end result of a first iteration. The design might change radically if we take new requirements into account.

We will explore these issues in depth in later lectures.

Running Example: Snakes and Ladders



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

Snakes and Ladders is a rather simple game suitable for teaching children how to apply rules. It is dull for adults because there is absolutely no strategy involved, but this makes it easy to implement as a computer program!

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.
- …

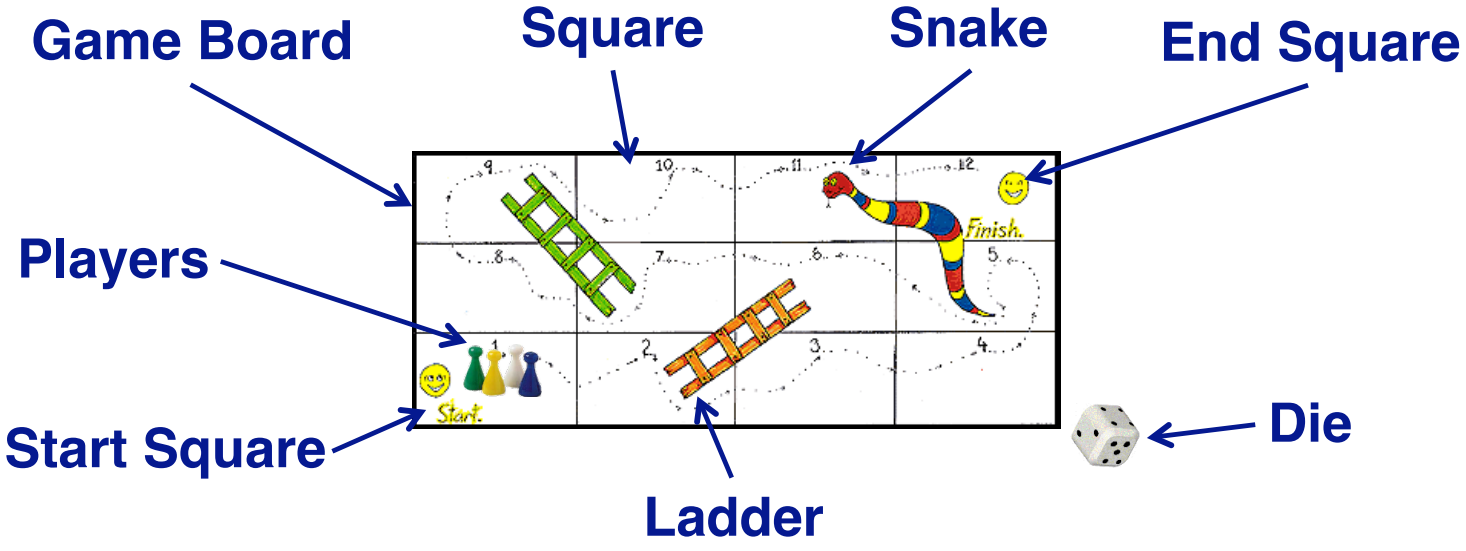
Roadmap

- 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

In oo programming we go from a high leve domain to low level doamin. Saying that in low level maybe the board is an array etc.

Model domain objects



What about roll, action, winner … ?

Object-orientation lets you simulate a virtual world, so model the domain objects as you want them to be. Note that we do not have to model the world as it is, but as we want to think about it! In our virtual world, objects can be active and take responsibility for their actions, unlike passive objects in the real world.

We have a game board, rules, players, a die, and various kinds of squares. Watch out for synonyms – space/square, player/token …

Are there other domain objects we have missed?

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.

Don’t distinguish between first and second class objects: everything that has a responsibility and plays a role should be an object.

Sure, Squares are part of the board, but they are objects too.

“Everything is an object” — Goldberg and Robson, Smalltalk 80: the Language and its Implementation, 1983

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

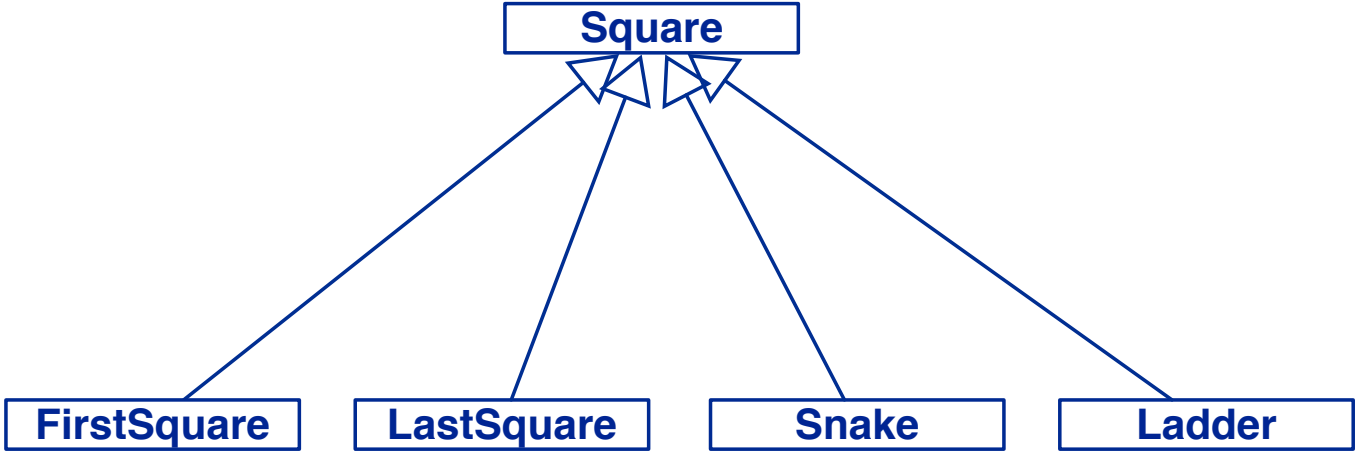
Daniel H. H. Ingalls, Design Principles Behind Smalltalk, Byte Magazine, August 1981.

Alan Kay (the inventor of Smalltalk) argued that computing power should be used to simulate a virtual world that enhances the user's interactive experience.

Alan C. Kay. Microelectronics and the Personal Computer. In Scientific American 3(237) p. 230—240, 1977

Model specialization

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



Is a snake a kind of reverse ladder?

Model IS-A relationships, e.g., a Snake *is-a* Square, the First Square *is-a* Square. There are different kinds of Squares, so model them as a hierarchy with a common interface.

Are the first and last squares really special kinds of squares, or not? Are there other special squares?

Is a snake a kind of ladder, or vice versa? Are they both special versions of something more general?

More about this in the lecture on inheritance …

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*

Data-driven approaches are bad for encapsulation because they focus too quickly on implementation (representation) of objects rather than their interface.

Instead, focus on the responsibilities of an object: what data and tasks is it responsible for?

Responsibility = what you know and maintain

This will lead you to focus on the *interface* of an object rather than its *representation*.

Identifying responsibilities will help you to discover missing objects, and it will also tell you whether an object is needed in your design or not.

Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. Designing Object-Oriented Software, Prentice-Hall, 1990.

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

Robert (“Uncle Bob”) Martin has written extensively on OO design principles:

http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Martin equates responsibility with change of state:

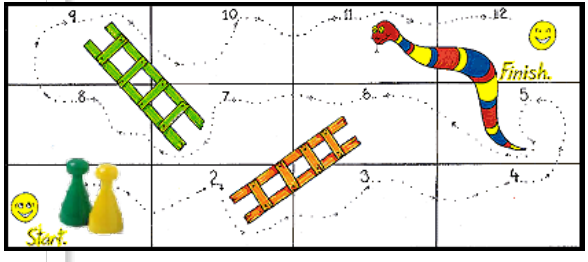
> “There should never be more than one reason for a class to change.”

Multiple responsibilities impact cohesion — an object that does too many things is overly complex.

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());
```



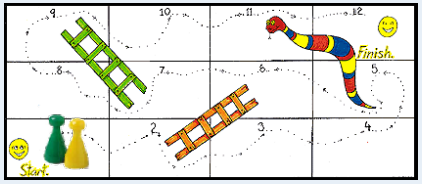

```
game.movePlayer(4);
assertTrue(game.notOver());
assertEquals(5, jack.position());
assertEquals(1, jill.position());
assertEquals(jill, game.currentPlayer());
```

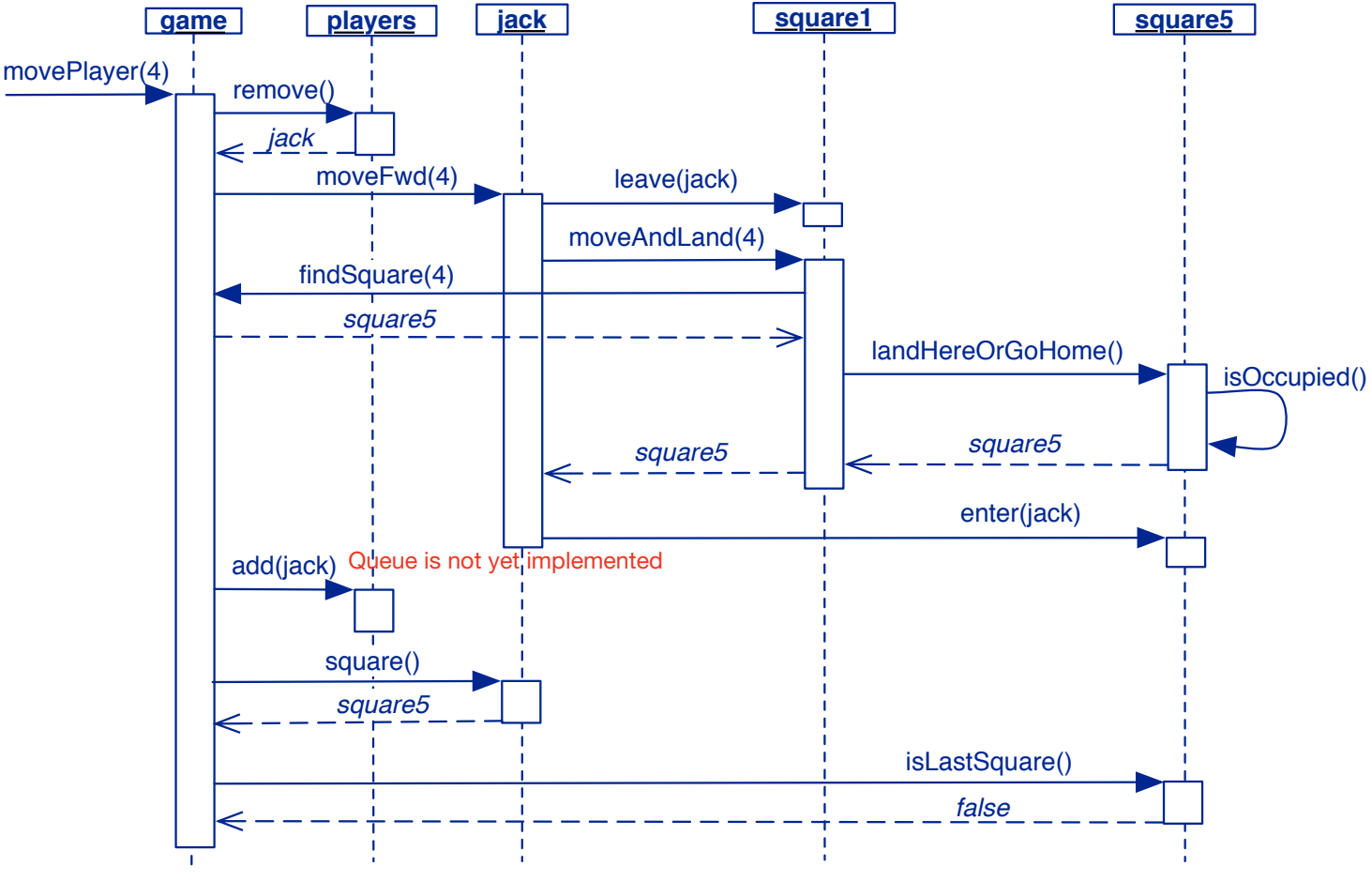
http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design

Test-driven development works this way. By writing tests first, you are forced into deciding what interfaces your objects must support.

In this case we are writing a test for a full scenario, not a unit test. More on this in the Unit Testing lecture …

Jack makes a move





This is a UML sequence diagram tracing a concrete scenario of the first move of a game:

The game has a queue of players to keep track of whose turn it is.

Jack moves forward by leaving his current square, then asking that square to find the square 4 positions further to land on.

The current square asks the game board which square that is.

It then asks that square (the fifth one) if it is safe to land there.

Since it isn’t, Jack can occupy this square.

The game puts Jack in the back of the queue, and checks if he has landed on the last square.

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

Clients should depend only on an interface, so changes to the implementation do not affect them.

Conversely, the provider is free to change the implementation if it knows that clients will not be affected.

Information hiding is important for many reasons — it is a version of the “need to know” principle.

Separate development of components is also enabled if dependencies are restricted to interfaces.

David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. In CACM 15(12) p. 1053—1058, December 1972.

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.*

See also:

Edward V. Berard, "Abstraction, Encapsulation, and Information Hiding" in Essays On Object-Oriented Software Engineering, 1993.

See:

William Cook, "On Understanding Data Abstraction, Revisited", OOPSLA 2009

for an interesting discussion on the distinction between data abstractions and abstract data types.

Encapsulate state

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

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

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

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

As a rule, the state of an object should only be accessible to the object itself. To ensure that instances of subclasses can also access their state, instance variables should normally be declared as `protected`. (If they are declared private, then subclass instances won't be able to accesss their own state!)

Avoid `public` or package scope for state as these will violate encapsulation and make your design more fragile.

leave the modifier out. (Default)

Changes to public state will impact all clients who make use of it.

Keep behaviour close to state

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

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

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

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

Behaviour should be associated with the objects responsible for the associated state.

Avoid defining behaviour for which another object is responsible.

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 {
 protected 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 …

Avoid depending on concrete classes. Your code should not care which implementation is underneath.

If you have *more than one implementation* of a given interface, consider defining it either as a *Java interface* or an *abstract class*.

Java *interfaces* have the advantage that they are *independent of the class hierarchy*, but they require all implementations to implement all methods. If you change the interface this will impact all clients.

Abstract classes are tied to the hierarchy, but you can provide a *default implementation*.

See the Introduction to *Gamma, et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.*

Aside: Messages and methods

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

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

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

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

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

Clients should not care what kind of square they occupy

The word “method” was introduced in Smalltalk as a metaphor – You tell objects to do something by *sending them a message*; the object then *chooses the method* it will use to handle that message.

If you talk about “calling a method” you are breaking the metaphor – it makes no sense in English to “call someone’s method”!

The point is that objects should encapsulate their behavior; you speak to their interface by sending a message. The method is internal – you should not know it.

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

Design classes and packages so their functionality can be extended without modifying the source code.

This can be achieved by using inheritance to override and extend inherited behaviour.

Bertrand Meyer. Object Oriented Software Construction, 1988.

See also the “Template Method” pattern in the Design Patterns book.

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

Responsibility implies non-interference.
— Timothy Budd

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

```
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);
 }
 ...
}
```

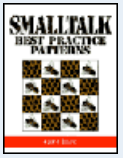
Do not take over the responsibilities of other objects. If an object is in charge of certain information, you should delegate related tasks to that object, rather than trying to handle them yourself.

For a player to move forward, it must find out what square it ultimately lands on, but *it is the responsibility of the squares on the board to interpret the logic of the game*.

For a square to let a player move forward one position, it needs to find out what is the next square on the board, but *it is the responsibility of the game to keep track of this*.

(Note that we are free to distribute responsibilities as we choose, but we should also make sure that no object is overloaded with responsibilities.)

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

If you see similar logic repeated in many methods, this is a sign that the design has not been carefully thought through.

Long methods tend to be a sign of *procedural thinking*.
Small methods are a hallmark of *OO thinking*.

See the introduction to:
Kent Beck, Smalltalk Best Practice Patterns, Prentice-Hall, 1997.

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*.

If a method is too long, take groups of statements doing related things and encapsulate them in a (protected) helper methods.

Name those methods after what the statements are doing.

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!");
 }
 ...
```

This is the main loop of the game. Notice how it expresses at a very high level of abstraction, in just 10 lines how the game proceeds, until it is over.

The details of the game's logic are expressed at the lower levels.

… 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;
}
```

Most methods have trivial implementations, which makes it very easy to implement them and ensure that they are correct.

In most cases, the method name serves as all the documentation we need.

Unfortunately the (library) queue of players does not have *intention-revealing* method names for removing elements from the front or adding them to the back, so we need some comments here.

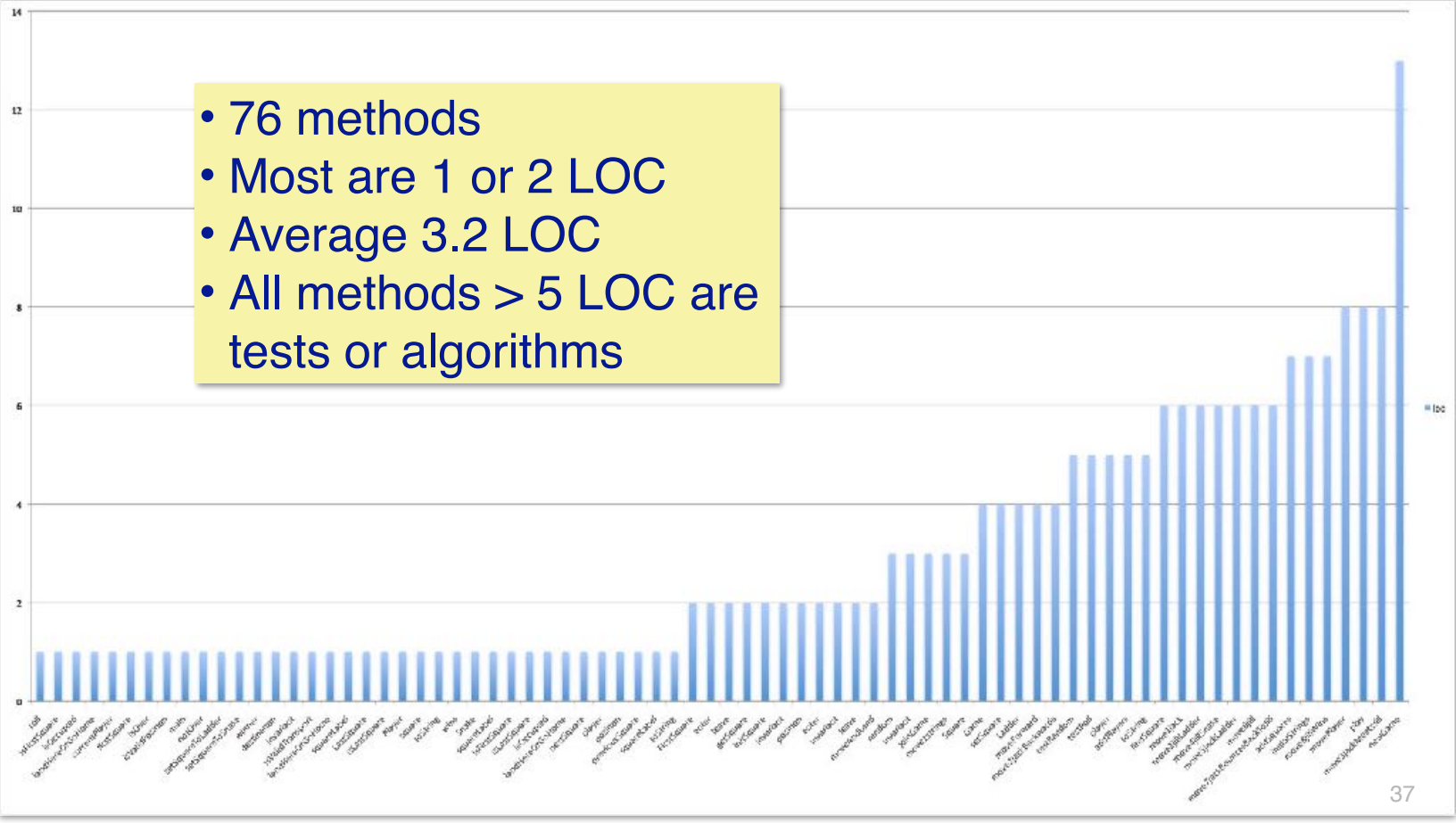
… and simple classes

```
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

Most object-oriented methods will be short and self-documenting.

Long methods are a sign of procedural thinking — the method is doing too much itself, rather than delegating tasks to other objects.

In a good OO design, the only longer methods tend to be *configuration scripts*, *algorithms*, or *tests*.

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

Design by Contract offers a concise way to express the client-supplier contract for each service of a software component.

Public operations should be annotated with *assertions* that express *preconditions* that express how the client should use the service, *post-conditions* that express what the service guarantees.

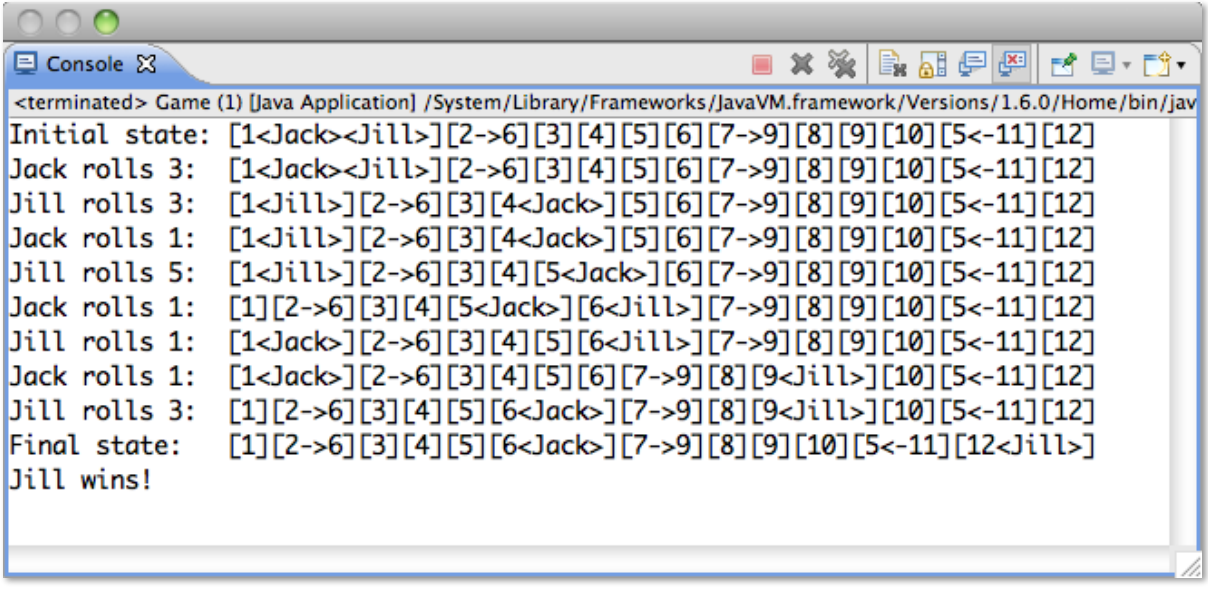
Additionally, *invariants* express what should be true before and after every service. Contract violations indicate defects in the way that the client is using the component (preconditions), or in the implementation of the component itself (post-conditions and invariants).

The next lecture will be entirely devoted to design by contract.

Bertrand Meyer. Object-Oriented Software Construction, ed., Prentice-Hall, 1997.

Demo

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

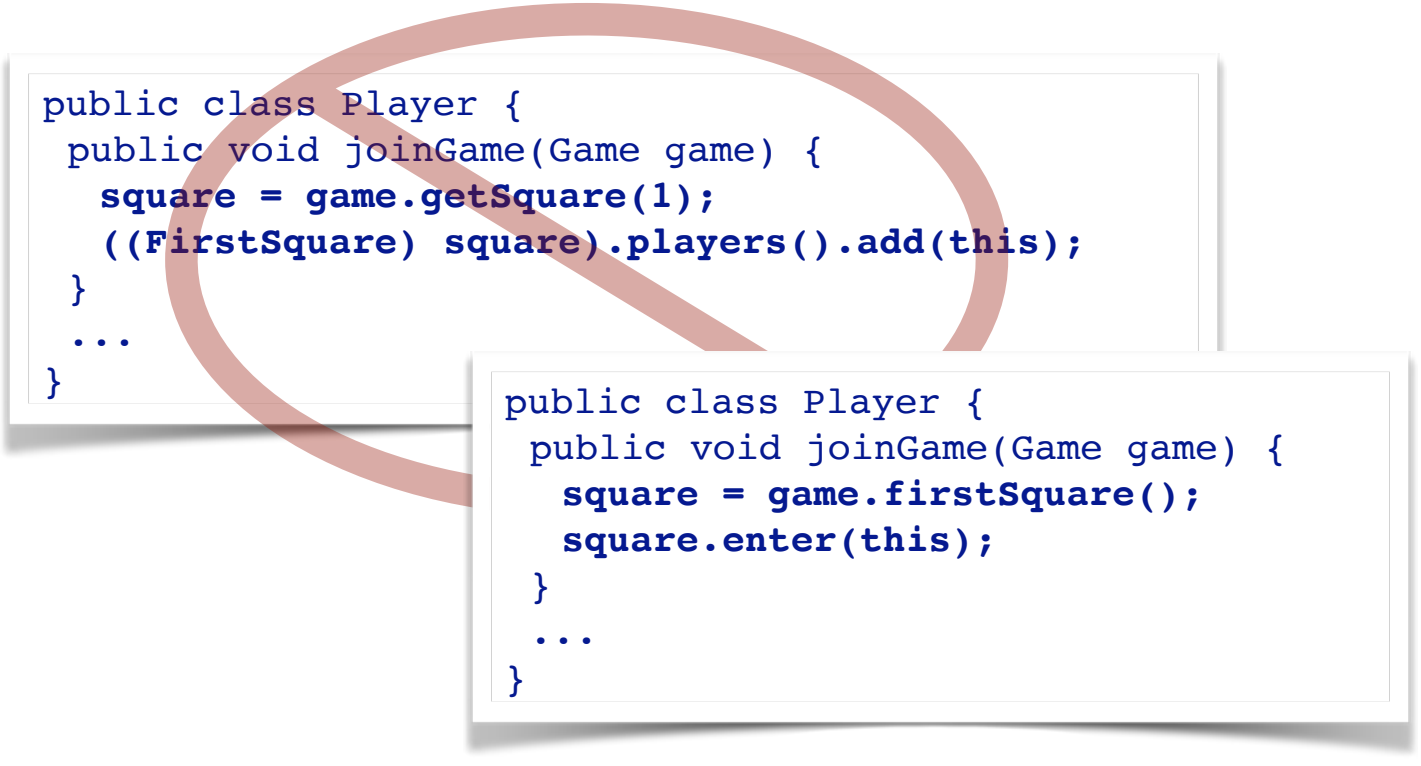


Roadmap

- 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);
 }
 ...
}
```

Names should be chosen to reveal the *intent* of the code.

In the sample code, the point is not that we want to get the square numbered 1, but that we want the first square of the game. Similarly we want to express that this player is entering the square, not how this is achieved at a low level.

Whenever you find yourself writing procedural code, extract that code into declaratively named methods that express what that code is doing.

Role Suggesting Instance Variable Name

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

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

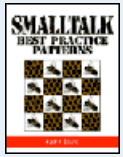
Make the name plural if the variable will hold a collection.

Several of these patterns are documented in:

Kent Beck, Smalltalk Best Practice Patterns, Prentice-Hall, 1997.

Although the book focuses on Smalltalk practices, many of the patterns apply more generally to OO design.

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

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

This is called *"Intention Revealing Selector"* in Beck's book.
(In Smalltalk, method names are called "selectors.")

We do not care *how* a square allows a player to enter it — we just care that it happens. The fact that the first square does it differently from other squares is not relevant at the client's level of abstraction.

By selecting a *declarative name* for this method, we make the code *self-documenting* and more flexible.

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

The *Law of Demeter* states that you should only send messages to: an argument passed to you; an object you create; self, super; or your class.

By obeying this law, you avoid unnecessarily coupling your classes. Instead of retrieving data from other objects, you should delegate tasks to them.

Alec Sharp in *Smalltalk by Example, McGraw-Hill, 1997*, explains this as *"Tell, don't ask"*

Karl J. Lieberherr, et al. Object-Oriented Programming: An Objective Sense of Style. In Proceedings OOPSLA '88.

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

Duplicated Code often indicates a lack of abstraction in your design. Share code with the help of inheritance or delegation.

Long Methods are signs of procedural thinking. Decompose your method into helper methods with intention-revealing names.

A very *Large Class* or a "*God Class*" has too many responsibilities. Refactor your design to distribute behaviour to other classes.

A *Long Parameter List* suggests that an object is missing. See if you can bundle arguments together into a new kind of object.

Feature Envy refers to a method that requires too much information from other objects. Redistribute the responsibility so that the work is done close to the objects.

Data Classes provide mainly accessor methods. Find the behaviour in its clients and shift the responsibilities to the data class to make it a real class.

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!

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

1) Lehmann and Belady studied the evolution of industrial software systems in the early 80s and formulated several "laws of software evolution". Even though software technology has changed, these laws are generally considered to hold true today.

Systems that are used in a real environment are always asked to do more because business changes.

Nowadays, more effort is spent in "maintenance" (i.e. after deployment) than in initial development.

Or the OS changes or other co-Programs are changing, so you have to adapt.

2) We want changes to be easy to make — small, incremental changes should be well-localized, easy to understand and verify.

Consequently we need to build software around stable abstractions.

Data (implementation) tends to change, so what do we do?

Instead we should focus on *domain concepts* and *responsibilities*. This is the key idea behind object-oriented design.

It's easier this way.

3) Each object has its own resbonsibilities, and should only do that

Data-driven approaches are bad for encapsulation because they focus too quickly on implementation (representation) of objects rather than their interface.

Instead, focus on the responsibilities of an object: what data and tasks is it responsible for?

Responsibility = what you know and maintain

This will lead you to focus on the *interface* of an object rather than its *representation*.

Identifying responsibilities will help you to discover missing objects, and it will also tell you whether an object is needed in your design or not.

4) Maybe we see that it's necessary to define an Interface when we have more or less several objects that do the same.

5)

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*

6) Modules should be both open (for extension) and closed (for modification).

7)

Do not take over the responsibilities of other objects. If an object is in charge of certain information, you should delegate related tasks to that object, rather than trying to handle them yourself.

For a player to move forward, it must find out what square it ultimately lands on, but *it is the responsibility of the squares on the board to interpret the logic of the game.*

For a square to let a player move forward one position, it needs to find out what is the next square on the board, but *it is the responsibility of the game to keep track of this.*

(Note that we are free to distribute responsibilities as we choose, but we should also make sure that no object is overloaded with responsibilities.)

8)

Names should be chosen to reveal the *intent* of the code.

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?*