

So Far ...

Part 1: OOAD Intro

Part 2: Inception

Part 3: Elaboration—Iteration 1

- Iteration 1—Basics
- Domain Models
- System Sequence Diagrams
- Operation Contracts
- Requirements to Design—Iteratively
- Logical Architecture and UML Package Diagrams
- On to Object Design
- UML Interaction Diagrams (Self Study)
- UML Class Diagrams (Self Study)
- GRASP: Designing Objects with Responsibilities

- Object Design Examples with GRASP
- Designing for Visibility
- Mapping Designs to Code
- **Test-Driven Development and Refactoring**

Part: 4 Elaboration Iteration 2—More Patterns

- GRASP: More Objects with Responsibilities
- Applying GoF Design Patterns

Test-Driven Development and Refactoring

Abdulkareem Alali

Ack Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

Logic is the art of going wrong with confidence.
—Joseph Wood Krutch

Intro

Extreme Programming (XP) **promotes**

- Writing the **tests** first
- Continuous code **refactoring**

Why?

- **Improve its quality**
- **Less duplication**
- **Increased clarity**

Modern tools **support** practices, OO developers **swear** by their value

Unit Testing First

Testing individual **components**, individual **classes**

In OO unit testing TDD-style (Test Driven Dev.), test code is written before the class to be tested

1. Imagining a production code,
2. Write a little test code,
3. Then write a little production code,
4. Make it pass the test,
5. ... then 1 & write some more test code, etc.

Unit Testing First, Why?

1. Unit tests get written—

Human nature, if left as an afterthought, writing unit test is avoided

2. Programmer Satisfaction—

- Test-last, or Just-this-one-time-I'll-skip-writing-the-test development Traditional style,
 - developer writes production code, debugs, then add unit tests,
 - it doesn't feel satisfying, you may even **hate** it!
- Human psychology. Test is written first, Pass Test, Can you?, I challenge you or myself?
 - Code is cut to pass the tests, feel of accomplishment—meeting a goal!

Unit Testing First, Why?

3. Clarification of detailed interface and behavior—

Writing tests, you **imagine** code exists, **details of public view of methods**

- Name, return value, parameters, and behavior

That **improves/clarifies** the detailed design;

designing your code before writing it

4. Provable, repeatable, automated verification—

Having **hundreds or thousands** of unit tests provides **verification of correctness**, runs automatically, it's easy

Unit Testing First, Why?

5. Confidence in change—

Unit test suite provides immediate **feedback** if the change caused an error

You write your own tests for your own code—

Who is better than the authors to write unit tests of their own code?

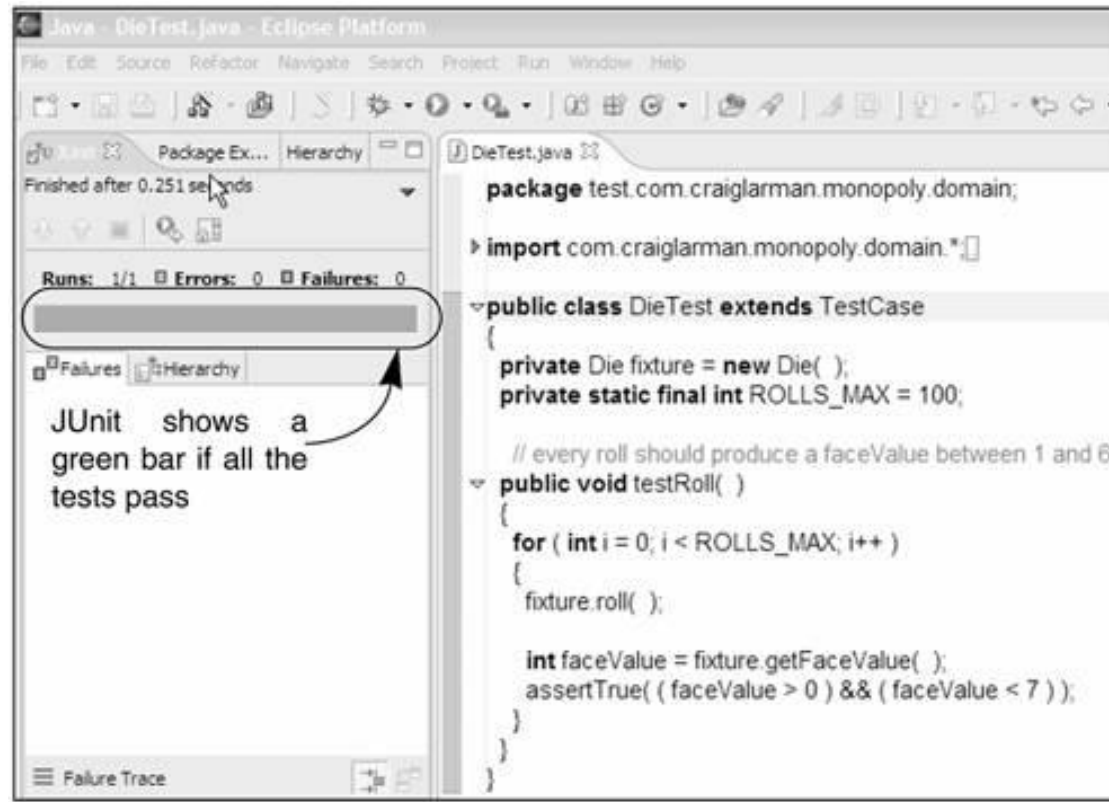
Frameworks

Most popular unit testing framework is the [xUnit](#) family (for many languages)

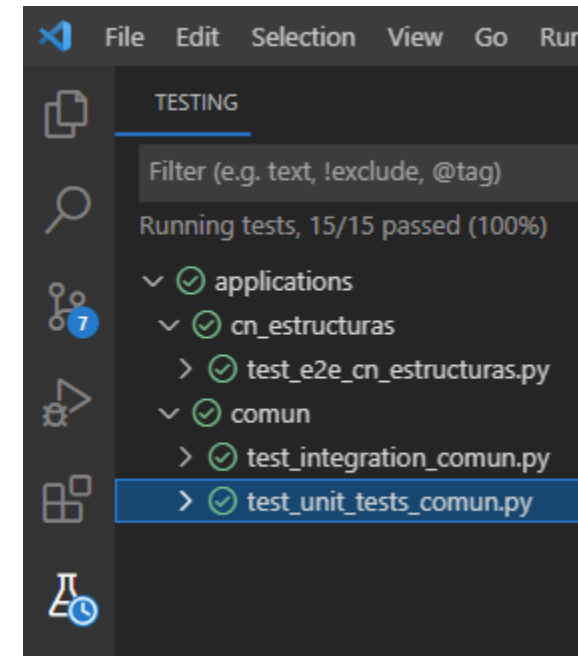
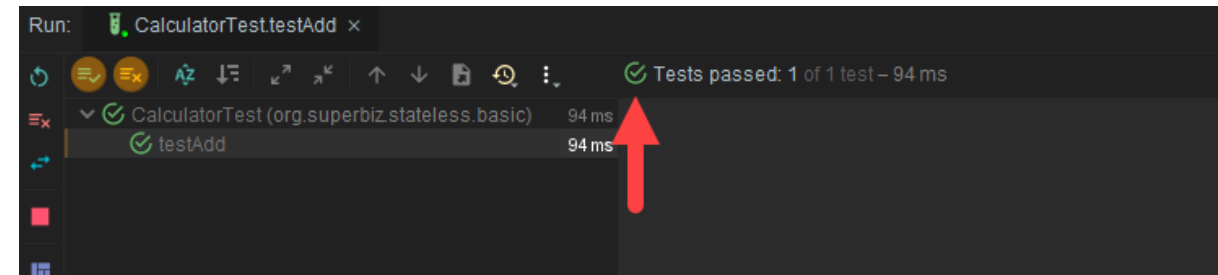
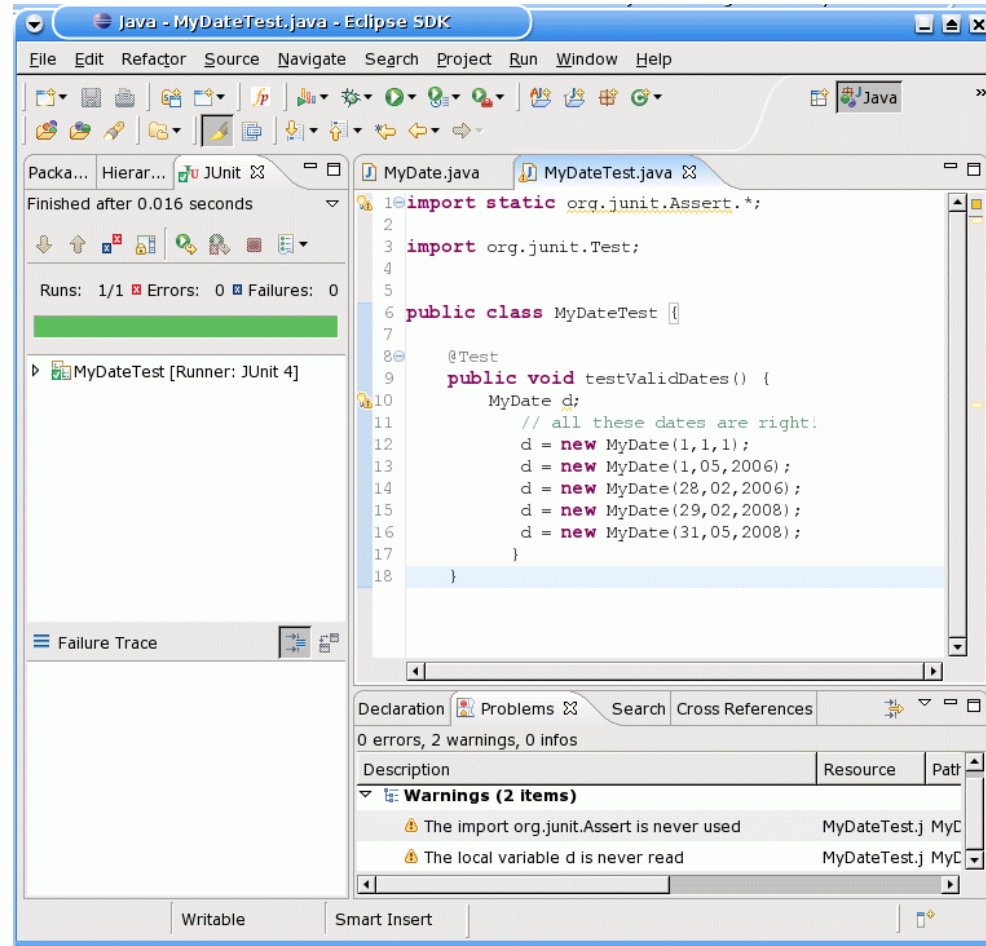
[JUnit](#) for Java, [NUnit](#) for .NET, etc. xUnits are integrated into IDEs (e.g. Eclipse, MS Visual Studio)

Keep the bar green to keep the code clean

TDD and JUnit in Eclipse



TDD and JUnit in a popular IDEs, Eclipse, IntelliJ, vscode



TDD —POS

Before programming **Sale** class, write unit testing method in a **SaleTest** class that does the following:

1. Create a **Sale**—the thing to be tested (also known as the [fixture](#))
2. Add some line items for the public ***makeLineItem*** method to test
3. Ask for the total, and verify that it is the expected value, using the ***assertTrue*** method

TDD How —POS

Do not write all the unit tests for **Sale** first; rather,

- Write only **one test** method,
- **Implement** the solution in class **Sale** to make it pass,
- and then **repeat**

To use xUnit, create test class that **extends** xUnit **TestCase** class

Write unit testing methods (perhaps several) **for each public method** of the **Sale** class

TDD How —POS

Exceptions include **trivial** (and usually auto-generated) get and set methods

To test method *MakeLineItem*, it is an idiom to name the testing method *testMakeLineItem*

1. Write *testMakeLineItem* test method,
2. then *Sale.makeLineItem* method to pass test

SaleTest

```
public class SaleTest extends TestCase
{
    // ...

    // test the Sale.makeLineItem method
    public void testMakeLineItem()
    {
        // STEP 1: CREATE THE FIXTURE

        // -this is the object to test
        // -it is an idiom to name it 'fixture'
        // -it is often defined as an instance field rather than
        // a local variable
        Sale fixture = new Sale();

        // set up supporting objects for the test
        Money total = new Money( 7.5 );
        Money price = new Money( 2.5 );
        ItemID id = new ItemID( 1 );
        ProductDescription desc =
            new ProductDescription( id, price, "product 1" );
    }
}
```

SaleTest

```
// STEP 2: EXECUTE THE METHOD TO TEST
```

```
// NOTE: We write this code **imagining** there  
// is a makeLineItem method. This act of imagination  
// as we write the test tends to improve or clarify  
// our understanding of the detailed interface to  
// to the object. Thus TDD has the side-benefit of  
// clarifying the detailed object design.
```

```
// test makeLineItem  
sale.makeLineItem( desc, 1 );  
sale.makeLineItem( desc, 2 );
```

```
// STEP 3: EVALUATE THE RESULTS
```

```
// there could be many assertTrue statements  
// for a complex evaluation
```

```
// verify the total is 7.5  
assertTrue( sale.getTotal().equals( total ) );
```

```
}
```

```
}
```


Refactoring

Is a structured, disciplined method to rewrite or **restructure** existing code without changing its external behavior

Via applying **small transformation** steps combined with **re-executing tests** each step,

With the goal to improve the design, structure, and implementation of the software while preserving its functionality

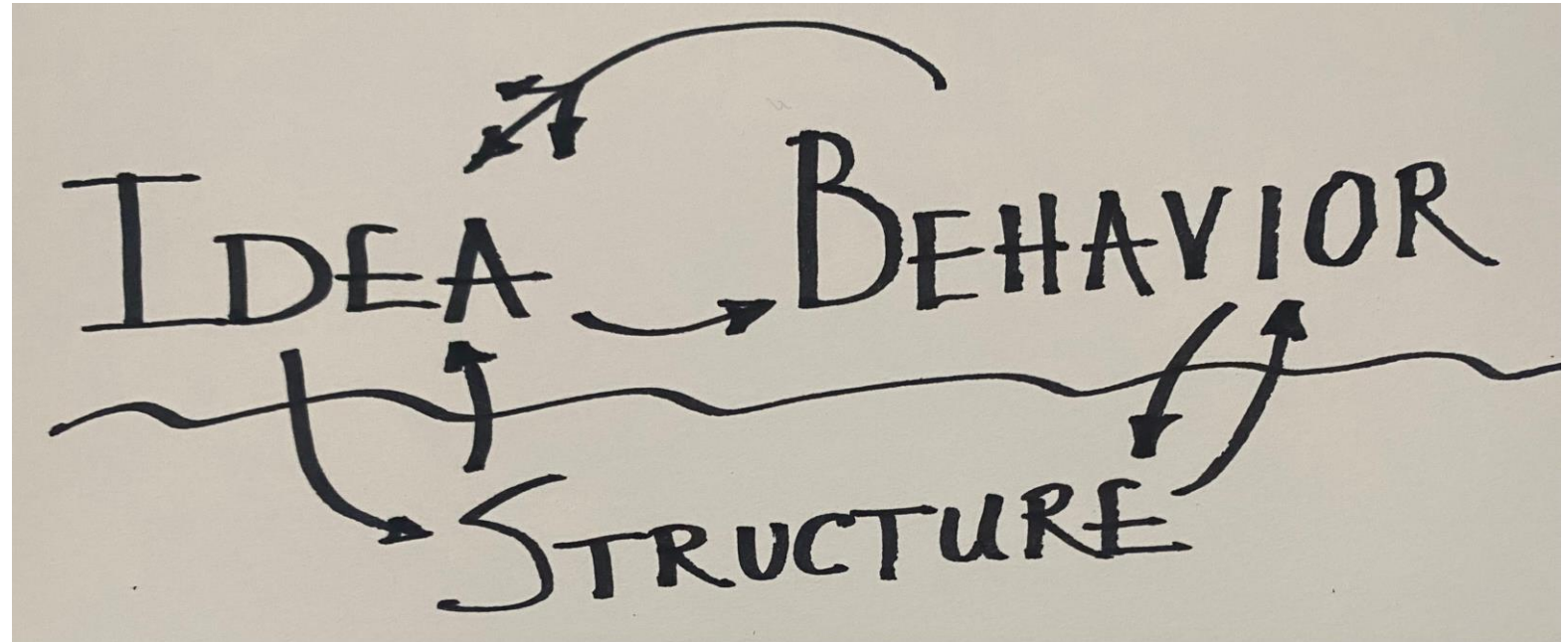
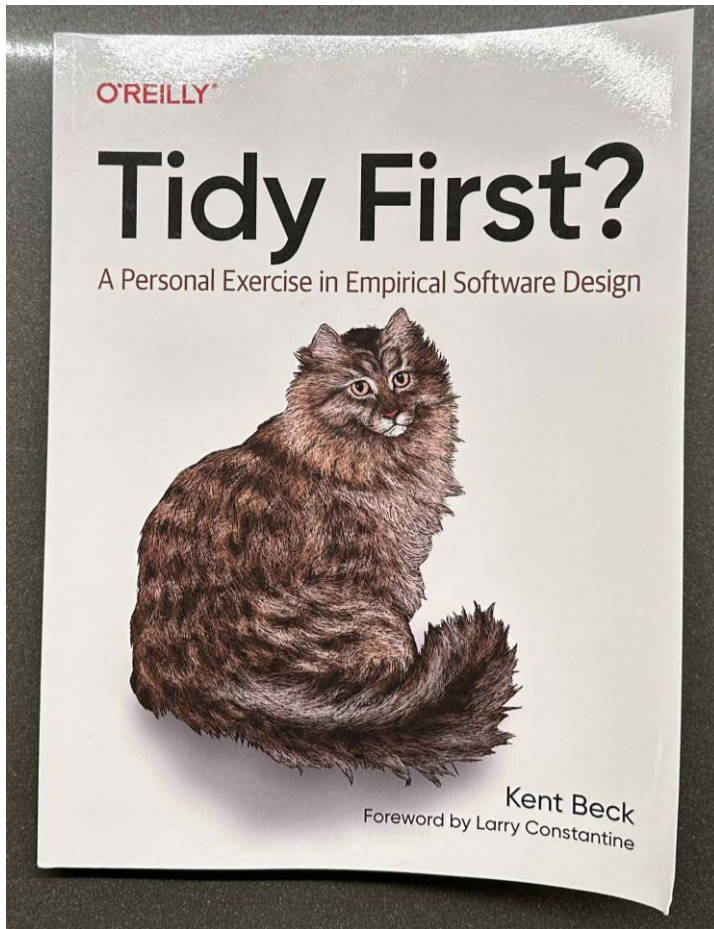
An **XP practice**, part of iterative methods, including UP

Refactoring, Why?

- More efficient by addressing dependencies and complexities
- More maintainable or reusable by increasing efficiency and readability
- Cleaner so it is easier to read and understand
- Easier for software developers to find and fix bugs or vulnerabilities

Summary, easier and resistant to change!!

Refactoring, README



youtu.be/XmsyvStDuqI

Refactoring and TDD

Refactoring is applying small **behavior preserving transformations** (each called a 'refactoring'), one at a time

After each transformation, the unit tests are **re-executed** to prove that the refactoring **did not cause a failure**

**Relationship between refactoring and TDD—
Unit tests support refactoring**

Refactoring, Why?

Each refactoring is small

A series of transformations—each followed by executing the unit tests **again and again**

Produces a major restructuring of code and design (for the better), while ensuring behavior remains the same

Refactoring, Why?

- More efficient by addressing dependencies and complexities
- More maintainable or reusable by increasing efficiency and readability
- Cleaner so it is easier to read and understand
- Easier for software developers to find and fix bugs or vulnerabilities

Summary, easier and resistant to change!!

Code Smells

Code that's been **well-refactored** is short, tight, clear, and without duplication—A work of a master programmer. Code that doesn't have these qualities smells bad or has code smells, **poor design**

Signs of Code Smell

- Duplicated code
- Big method
- Class with many instance variables
- Class with lots of code, non cohesive
- Strikingly similar subclasses
- Little or no use of interfaces in the design
- High coupling between many objects

Refactoring Activities

- Remove duplicate code
- Improve clarity
- Make long methods shorter
- Remove the use of hard-coded literal constants

Refactorings Have Names, 100!

Refactoring	Description
<i>Extract Method</i>	Transform a long method into a shorter one by factoring out a portion into a private helper method
<i>Extract Constant</i>	Replace a literal constant with a constant variable
<i>Introduce Explaining Variable</i>	Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose
<i>Replace Constructor Call with Factory Method</i>	Replace using the new operator and constructor call with invoking a helper method that creates the object (hiding details)

Extract Method Refactoring

Example

Player.*takeTurn* has an initial section of code that **rolls** the **dice** and **calculates** the **total** in a **loop**

Make the ***takeTurn*** method shorter, clearer, and better supporting **High Cohesion** by extracting that code into a private helper method called ***rollDice***

Extract Method Refactoring —Before Refactoring

```
public class Player
{
    private Piece  piece;
    private Board  board;
    private Die[]  dice;
    // ...

    public void takeTurn()
    {
        // roll dice
        int rollTotal = 0;
        for (int i = 0; i < dice.length; i++)
        {
            dice[i].roll();
            rollTotal += dice[i].getFaceValue();
        }

        Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
        piece.setLocation(newLoc);
    }

    } // end of class
```

Extract Method Refactoring

—After Refactoring

```
public class Player
{
    private Piece  piece;
    private Board  board;
    private Die[]  dice;
    // ...

    public void takeTurn()
    {
        // the refactored helper method
        int rollTotal = rollDice();

        Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
        piece.setLocation(newLoc);
    }

    private int rollDice()
    {
        int rollTotal = 0;
        for (int i = 0; i < dice.length; i++)
        {
            dice[i].roll();
            rollTotal += dice[i].getFaceValue();
        }
        return rollTotal;
    }

    // end of class
}
```

162624 17-NOV-2016 131.123.1.

Introduce Explaining Variable —Before Refactoring

```
// good method name, but the logic of the body is not clear
boolean isLeapYear( int year )
{
    return( ( ( year % 400 ) == 0 ) ||
            ( ( ( year % 4 ) == 0 ) && ( ( year % 100 ) != 0 ) ) );
}
```

Introduce Explaining Variable —After Refactoring

Clarifies, simplifies, and reduces the need for comments

```
// that's better!
boolean isLeapYear( int year )
{
    boolean isFourthYear = ( ( year % 4 ) == 0 );
    boolean isHundrethYear = ( ( year % 100 ) == 0 );
    boolean is4HundrethYear = ( ( year % 400 ) == 0 );
    return (
        is4HundrethYear
        || ( isFourthYear && ! isHundrethYear ) );
}
```

IDE Support for Refactoring —Before Refactoring



IDE Support for Refactoring —After Refactoring

```
public void takeTurn()
{
    int rollTotal = rollDice();

    Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
    piece.setLocation(newLoc);
}

private int rollDice()
{
    // roll dice
    int rollTotal = 0;
    for (int i = 0; i < dice.length; i++)
    {
        dice[i].roll();
        rollTotal += dice[i].getFaceValue();
    }
    return rollTotal;
}
```


AI CAN

You can prompt engineer your code into a better state by following patterns learned by an LLM

So Far ...

Part 1: OOAD Intro

Part 2: Inception

Part 3: Elaboration—Iteration 1

- Iteration 1—Basics
- Domain Models
- System Sequence Diagrams
- Operation Contracts
- Requirements to Design—Iteratively
- Logical Architecture and UML Package Diagrams
- On to Object Design
- UML Interaction Diagrams (Self Study)
- UML Class Diagrams (Self Study)
- GRASP: Designing Objects with Responsibilities

- Object Design Examples with GRASP
- Designing for Visibility
- Mapping Designs to Code
- Test-Driven Development and Refactoring

Part: 4 Elaboration Iteration 2—More Patterns

- **GRASP: More Objects with Responsibilities**
- Applying GoF Design Patterns
- S.O.L.I.D

Part: 4 Elaboration Iteration 2—More Patterns

GRASP: More Objects with Responsibilities

Abdulkareem Alali

Ack Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

Luck is the residue of design.

—Branch Rickey

There Are Nine GRASP Patterns

1. **Creator**
2. **Information Expert**
3. **Low Coupling**
4. **Controller**
5. **High Cohesion**
6. **Indirection**
7. **Pure Fabrication**
8. **Polymorphism**
9. **Protected Variations**

Polymorphism

Polymorphism

- Alternatives based on type (Types with similar functionalities and different behavior)
 - If **If-then-else** or **case** statement **conditional** logic used,
 - A new variation arises,
 - Requires modification of logic often in many places,
 - Hard to extend a program
- **Pluggable** software components
 - e.g. Viewing components in client-server relationships, how can you replace **one server component** with another, without affecting the client?

Polymorphism

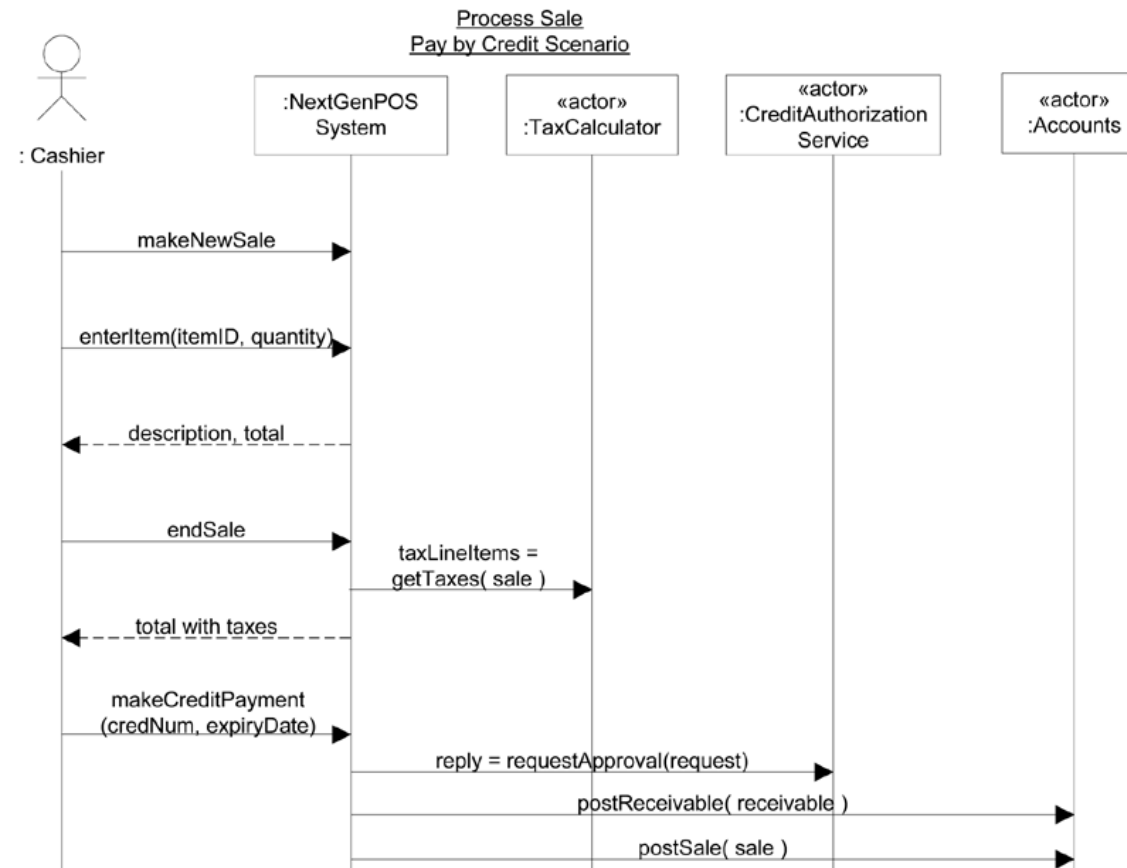
When **related alternatives** or **behaviors vary by type** (class), with similar **functionality**

Assign responsibility for the behavior
—using **polymorphic operations**—
to the **types** for which the **behavior varies**

POS

—SSD for Process Sale ... Updated

—External Systems



POS

—Support 3d-Party Tax Calculators?

Multiple external third-party tax calculators

e.g. Tax-Master, Good-As-Gold, TaxPro, etc.

System needs to integrate with different APIs

Behavior of calculator **varies** by **type** of calculator

POS

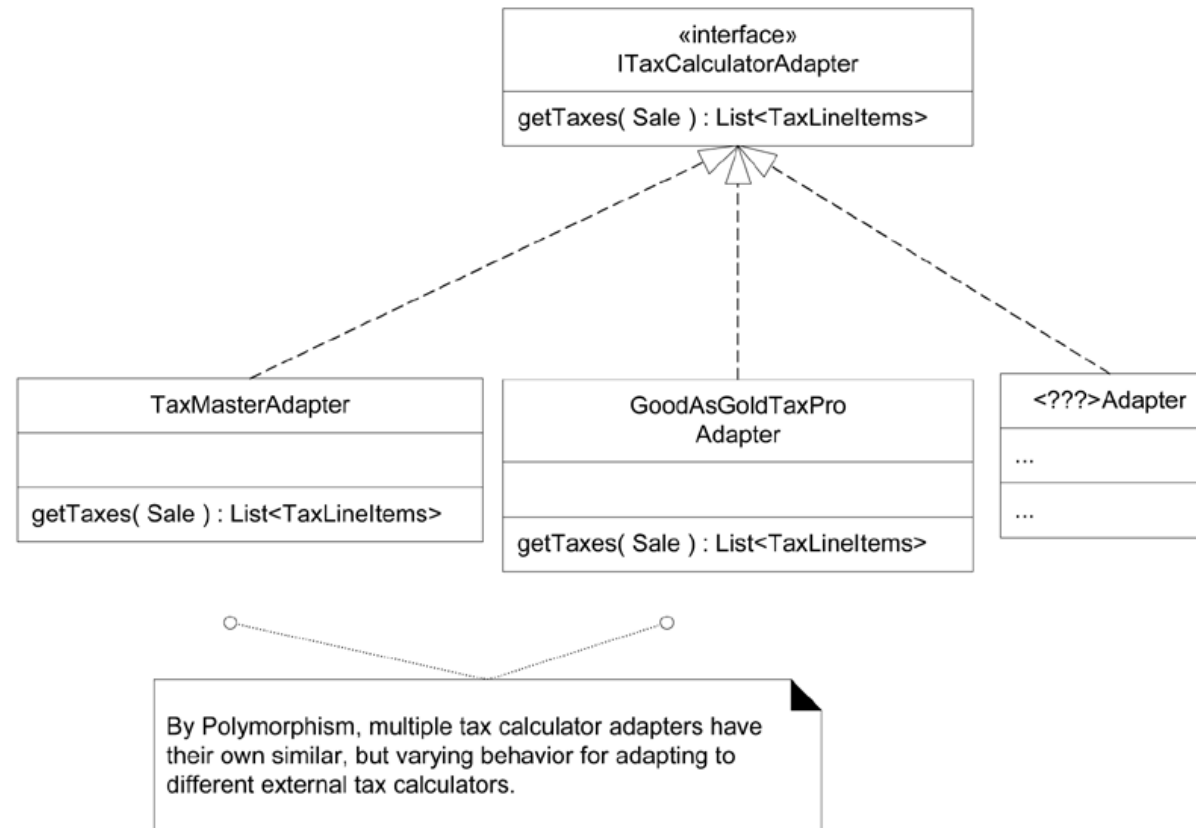
—Support 3d-Party Tax Calculators?

*What **objects** should be responsible for handling these *varying external tax calculator interfaces*?*

- By **Polymorphism**, assign responsibility to different calculator (or calculator **Adapter** from **GoF**)
- Implement a *polymorphic* **getTaxes** operation

POS

—3d-Party Tax Calculators



POS

—3d-Party Tax Calculators

Calculator adapter objects are not external calculators but local software objects

that represent external calculators (**Adapters**)

Sending a message to the local object,

Calls external calculator in its native API

POS

—3d-Party Tax Calculators

Each ***getTaxes*** method takes the **Sale** object as a parameter, so that the calculator can [analyze the sale](#)

Implementation of each ***getTaxes*** method will be different:

- e.g. **TaxMasterAdapter** will adapt request to the API of Tax-Master, etc

Monopoly

How to Design for Different Square Actions?

Player lands on **Go square**, receive \$200,

Land on the **Income Tax square**, gets a 10% deduction of cash, etc.

A different rule for different types of squares

Do not test for the type of an object and use conditional logic to perform varying alternatives based on type

Monopoly

—Design for Different Square Actions?

// Bad Design

SWITCH ON square.type

 CASE GoSquare:

 player receives \$200

 CASE IncomeTaxSquare:

 player pays tax

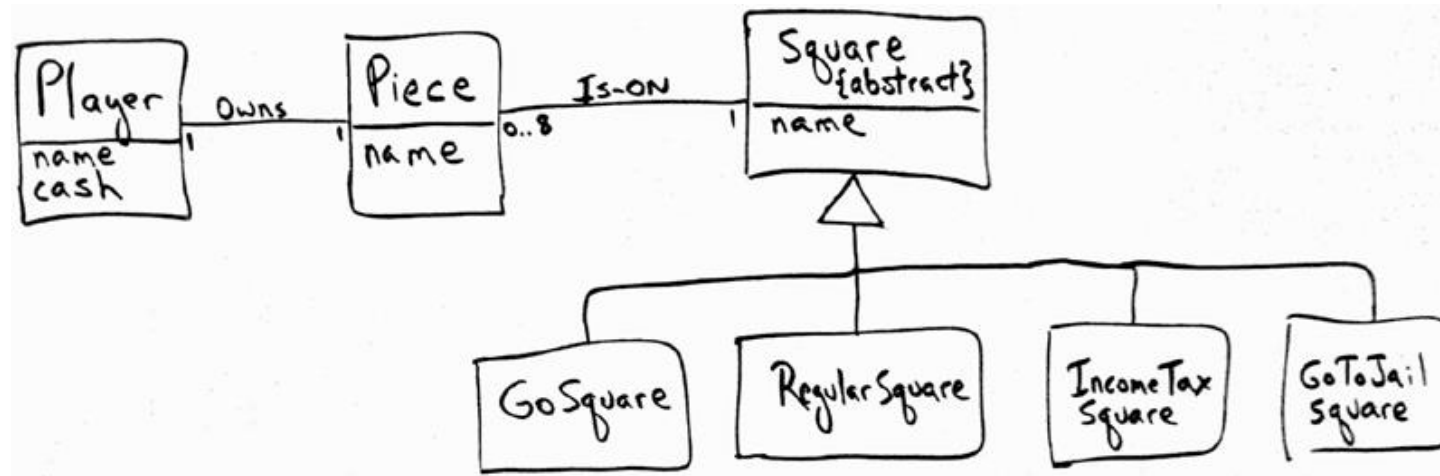
 CASE GoToJailSquare:

 player locked in jail

...

Monopoly

—Design for Different Square Actions?



Monopoly

—Design for Different Square Actions

Create a **polymorphic operation** for each type for which the behavior varies

What is the operation that varies?

It's what happens when a player lands on a square

Polymorphic operation is ***landedOn***

Non **default behavior** in the superclass,

{abstract} declaration of the polymorphic operation in superclass

Monopoly

—Design for Different Square Actions

*What object should send the **landedOn** message to the square that a player lands on?*

Player knows its location square (Low Coupling + Expert)

Player is a good to send message, **Player** visible to **Square**

Naturally, this message should be sent at the **end** of the **takeTurn** method

Class Player

```
public class Player
{
    private String name;
    private Piece piece;
    private Board board;
    private Die[] dice;

    public Player(String name, Die[] dice, Board board)
    {
        this.name = name;
        this.dice = dice;
        this.board = board;
        piece = new Piece(board.getStartSquare());
    }

    public void takeTurn()
    {
        // roll dice
        int rollTotal = 0;
        for (int i = 0; i < dice.length; i++)
        {
            dice[i].roll();
            rollTotal += dice[i].getFaceValue();
        }

        Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
        piece.setLocation(newLoc);
    }

    public Square getLocation()
    {
        return piece.getLocation();
    }

    public String getName()
    {
        return name;
    }
}
```

Class MonopolyGame

```
public class MonopolyGame
{
    private static final int ROUNDS_TOTAL = 20;
    private static final int PLAYERS_TOTAL = 2;
    private List players = new ArrayList( PLAYERS_TOTAL );
    private Board board = new Board( );
    private Die[] dice = { new Die(), new Die() };

    public MonopolyGame( )
    {
        Player p;
        p = new Player( "Horse", dice, board );
        players.add( p );
        p = new Player( "Car", dice, board );
        players.add( p );
    }

    public void playGame( )
    {
        for ( int i = 0; i < ROUNDS_TOTAL; i++ )
        {
            playRound();
        }
    }

    public List getPlayers( )
    {
        return players;
    }

    private void playRound( )
    {
        for ( Iterator iter = players.iterator( ); iter.hasNext( ); )
        {
            Player player = (Player) iter.next();
            player.takeTurn();
        }
    }
}
```

Monopoly

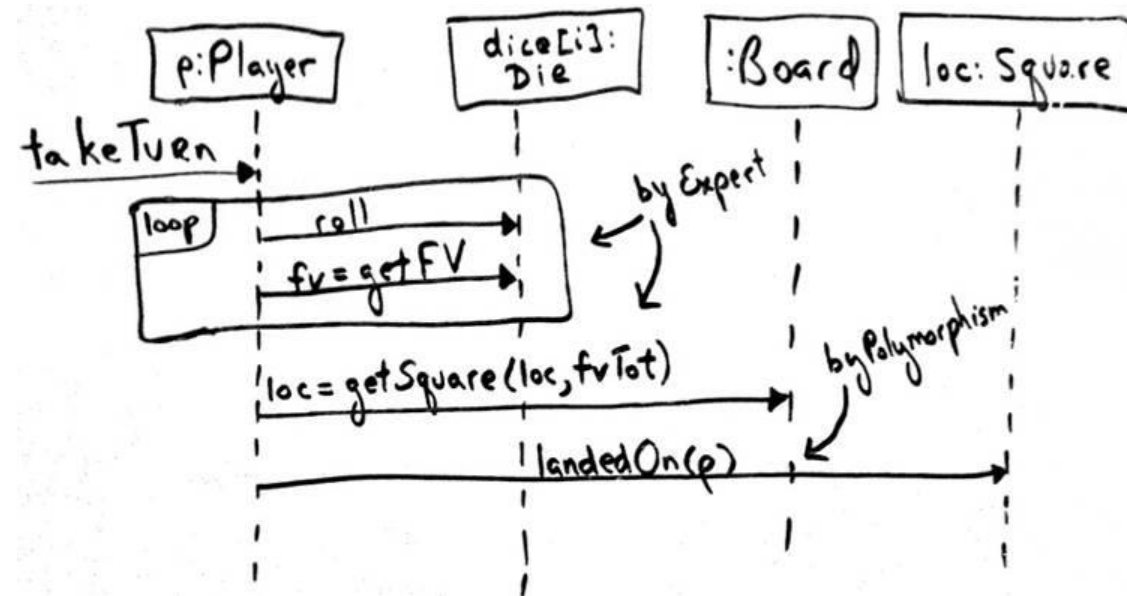
- Design for Different Square Actions
- Applying Polymorphism

Player object is labeled 'p' so that in the *landedOn* message we can refer to that object in the [parameter](#) list

Square object is labeled loc ('location')

and same label as return *getSquare* message; It is the same object

How to Design for Different Square Actions? Applying Polymorphism



Polymorphic In Terms of GRASP

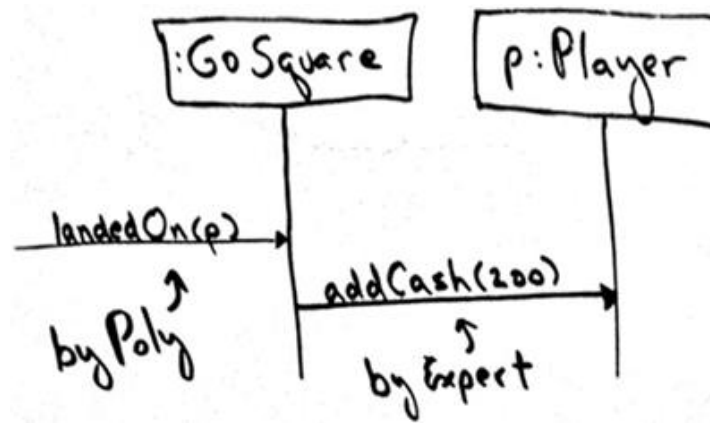
GoSquare—By **LRG**, **Player** knows its cash

By Expert, **Player** should be sent an *addCash* message

Square needs *visibility* to **Player** to send message

Player is passed as a parameter 'p' in the *landedOn*

GoSquare Case



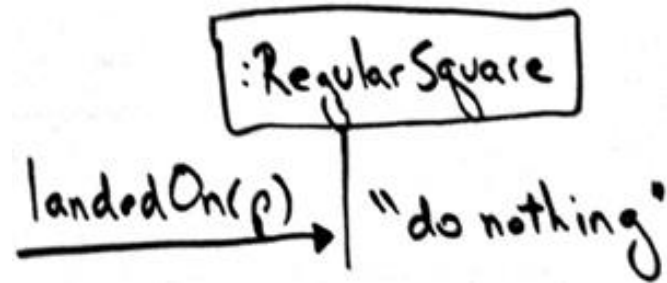
RegularSquare

RegularSquare—Nothing happens. Body of this method will be **empty**—sometimes called a NO-OP (no operation) method

Why send p (instance of **Player**) to **Square**, if NO-OP?

Note that to make the magic of **polymorphism** work, we need to use this approach to **avoid special case logic**

RegularSquare Case



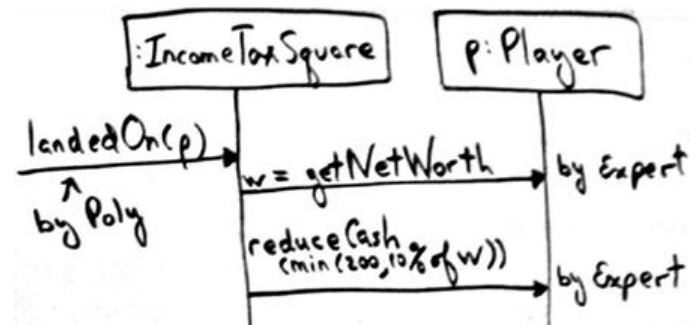
IncomeTaxSquare

IncomeTaxSquare—Calculate 10% of the player's net worth

By **LRG** and by Expert, who should know this?
Player

Thus the **Square** asks for the **Player's** worth, and then **deducts** the appropriate amount

IncomeTaxSquare Case



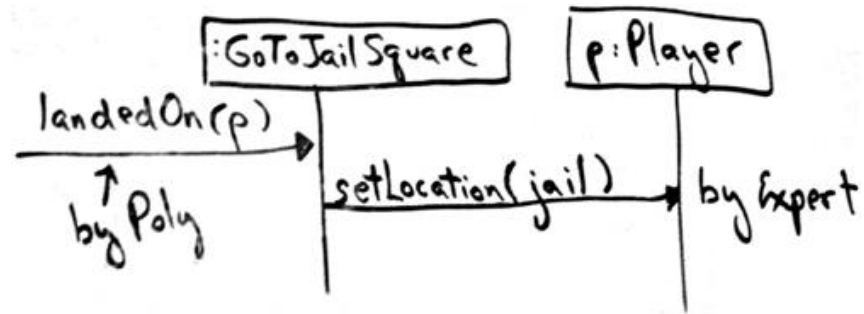
GoToJailSquare

GoToJailSquare—**Player's** location must be changed

By Expert, it should receive a ***setLocation*** message

GoToJailSquare will be initialized with an attribute referencing the **JailSquare**, so that it can pass this square as a parameter to the **Player**

GoToJailSquare Case



Improving Coupling

—OO Design refinement

Piece remembers square location but **Player** does not

Player must extract the location from the **Piece**

Send the *getSquare* message to the **Board**

Then re-assign the new location to the **Piece**

—Weak design point!

Player sends *landedOn* message to its **Square**, it becomes even weaker. Problems in coupling.

Improving Coupling

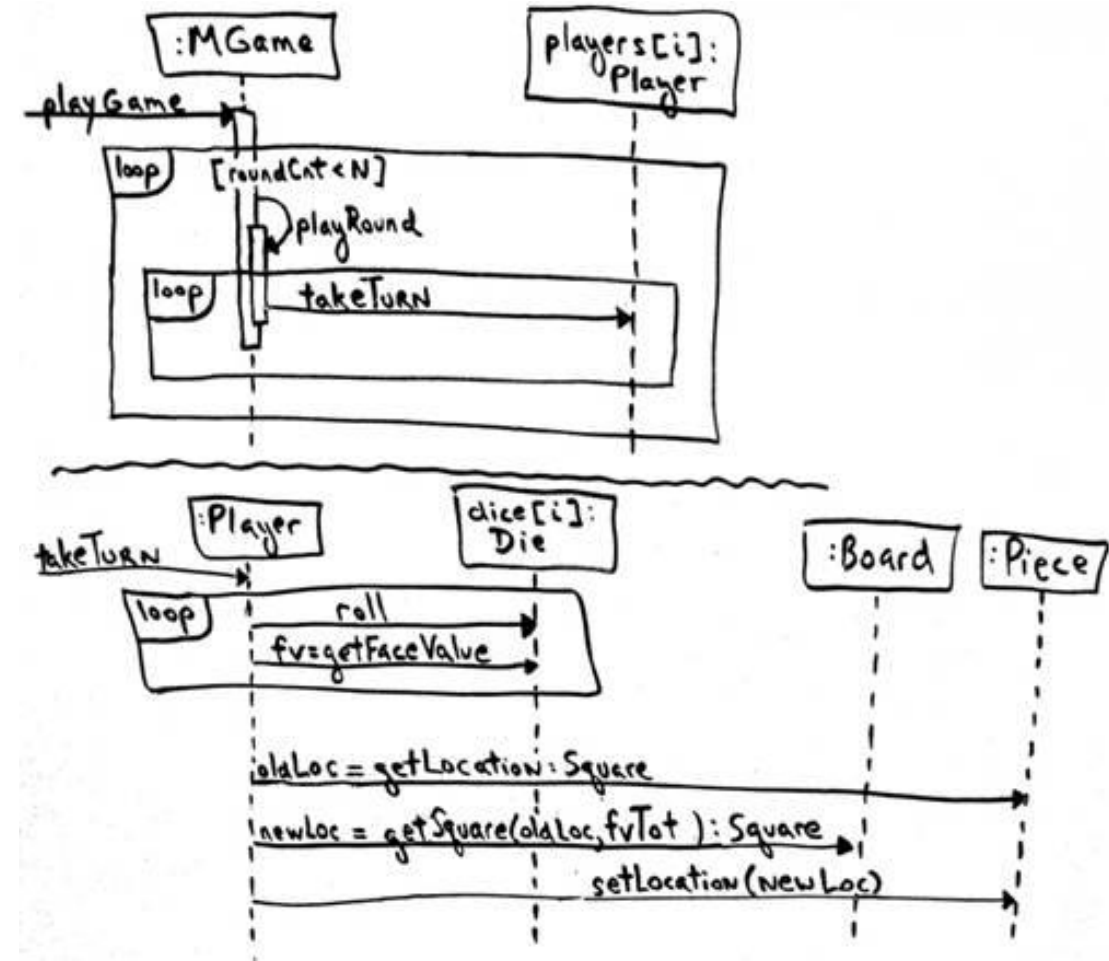
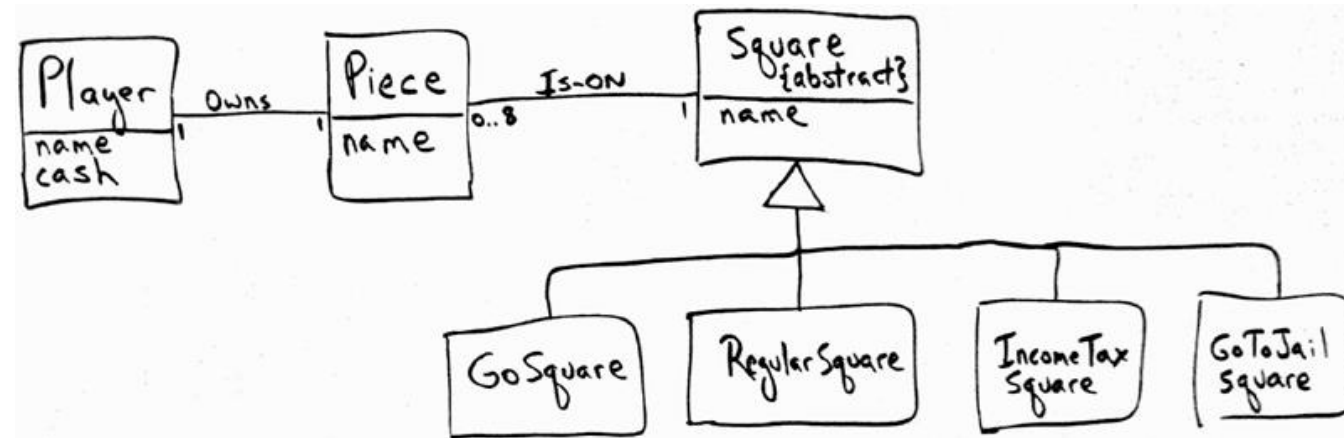
Player needs permanently to know its own **Square** location rather than **Piece**, since the **Player** keeps collaborating with its **Square**

A **refactoring** opportunity to improve coupling

- Object **A** keeps **needing** the data in object **B** it implies:
 - Either Object **A** should **hold** that **data**
 - Object **B** should have the **responsibility** (by Expert) rather than object **A**

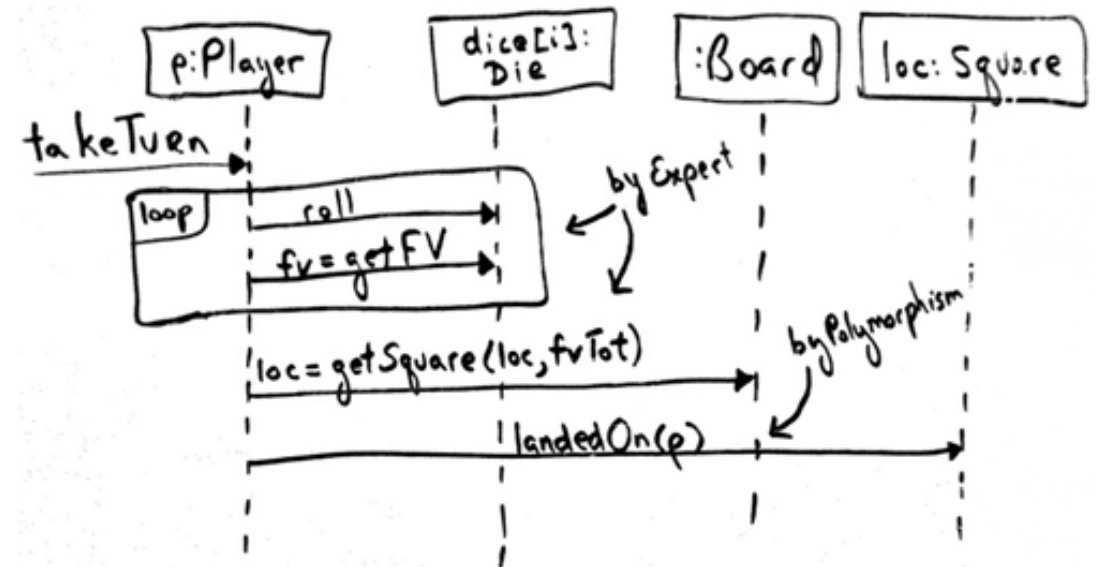
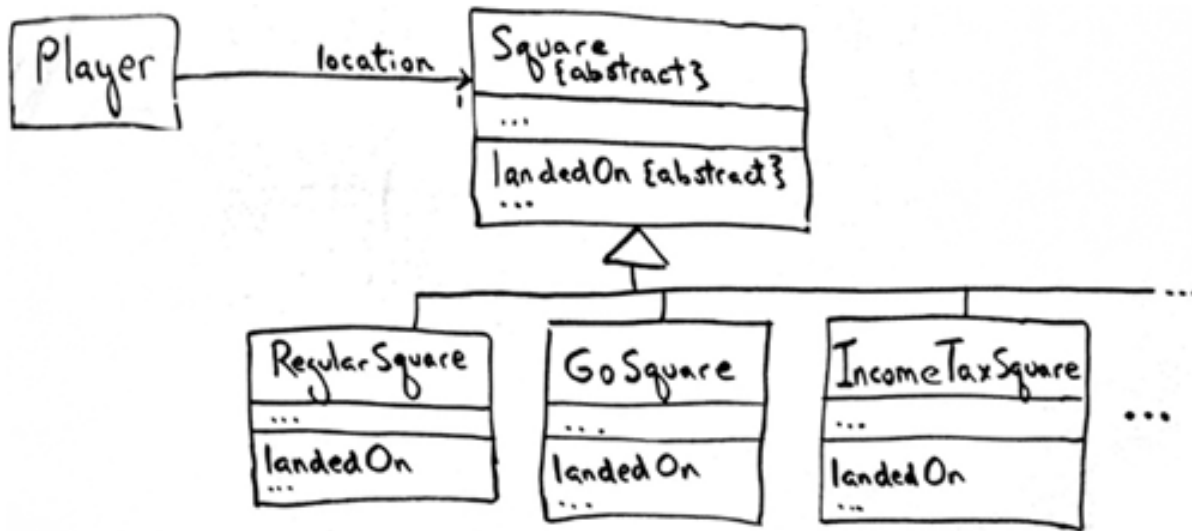
Player object can fulfill the role of the **Piece**

Improving Coupling – old *takeTurn*



Improving Coupling

– new *takeTurn*



When to Design with Interfaces

Support **polymorphism** without being **committed** to a particular class hierarchy

Abstract Superclass **AC** used without an interface, any new polymorphic solution must be a subclass of **AC**, limiting in single-inheritance languages such as Java and C#

For a class hierarchy with an abstract superclass **C1**, consider making an interface **I1** that corresponds to the public method signatures of **C1**, and then declare **C1** to implement the **I1** interface

A Flexible Evolution Point For Unknown Future Cases

Interfaces, Benefits?

- Extensions required for **new variations** are easy to add
- **Pluggable**, New implementations can be introduced **without affecting clients**

Pure Fabrication

Pure Fabrication

– Problem?

OOD **characterized** by software classes **representations** of concepts in the real-world problem domain to lower the representational gap (LRG)

Many situations, assigning responsibilities to domain layer software classes **leads to problems** in terms of

- **poor cohesion or**
- **coupling, or**
- **low reuse potential**

Pure Fabrication

—Solution?

Artificial or convenience class that does not represent a problem domain concept

—Something **made up**, to support **high cohesion**, **low coupling**, and **reuse**

Design of the fabrication is very clean, or **pure**

—**Pure Fabrication**

POS

—Saving a Sale Object in a DB

—Problem?

Save **Sale** instances in a relational database, By **Information Expert**, **Sale** class saves itself, **Sale** has data to save (e.g. **Active Record** .)

Implications:

- Large number of supporting DB operations (CRUD), **Sale** becomes **incohesive**
- **Sale** coupled to relational DB interface, not close but far (a database interface)
- Saving objects in a relational database is a very general task, **poor reuse** or
- Lots of duplication in other classes that do the same thing

POS

- Saving a Sale Object in a DB**
- Solution?**

Leads **low cohesion**, **high coupling**, and **low reuse** potential
—exactly the kind of **desperate situation** that calls for **making something up**

PersistentStorage is a Pure Fabrication

e.g. **Table Data Gateway** 

Save Sale via PersistentStorage

PersistentStorage is not a **domain concept**, but something made up or **fabricated** for the **convenience** of the software developer

Solves:

- **Sale** remains **well-designed**, with **high cohesion** and **low coupling**, **stable**
- **PersistentStorage** is **cohesive**, sole purpose of **storing or inserting objects in a persistent storage medium**
- **PersistentStorage** is **generic** and **reusable** object.

Save Sale via PersistentStorage

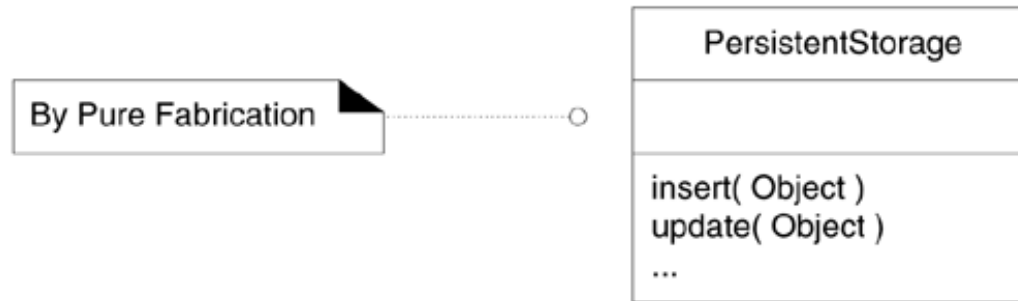
Eliminated bad design based on **Expert**, with **poor cohesion** and **coupling**,
with a **good** design in which there is greater potential for **reuse**

GRASP patterns, responsibilities placed but shifted from **Sale** (by **Expert**) to a **Pure Fabrication**

Save Sale via PersistentStorage

ORM like solution,

Object-Relational Mapping e.g. Hibernate Tools [.](#)



Monopoly

- Class **Player**
- Handling Dice After Refactoring

```
public void takeTurn()
{
    // the refactored helper method
    int rollTotal = rollDice();
}
```

```
public void takeTurn()
{
    // the refactored helper method
    int rollTotal = rollDice();

    Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
    piece.setLocation(newLoc);
}

private int rollDice()
{
    int rollTotal = 0;
    for (int i = 0; i < dice.length; i++)
    {
        dice[i].roll();
        rollTotal += dice[i].getFaceValue();
    }
    return rollTotal;
}
```

162624 17-NOV-2016 131.123.13

Monopoly

- Handling Dice
- Problem?

Player *rolls* all the dice and sums the total

Dice are very general objects, usable in many games

Summing service is not generalized for use in other games

It is not possible to simply ask for the current dice total without rolling the dice again

Monopoly

—Handling Dice

—Solution?

Pure Fabrication—make something up to conveniently provide related services

No **Cup** for dice in Monopoly, many games do use a dice cup

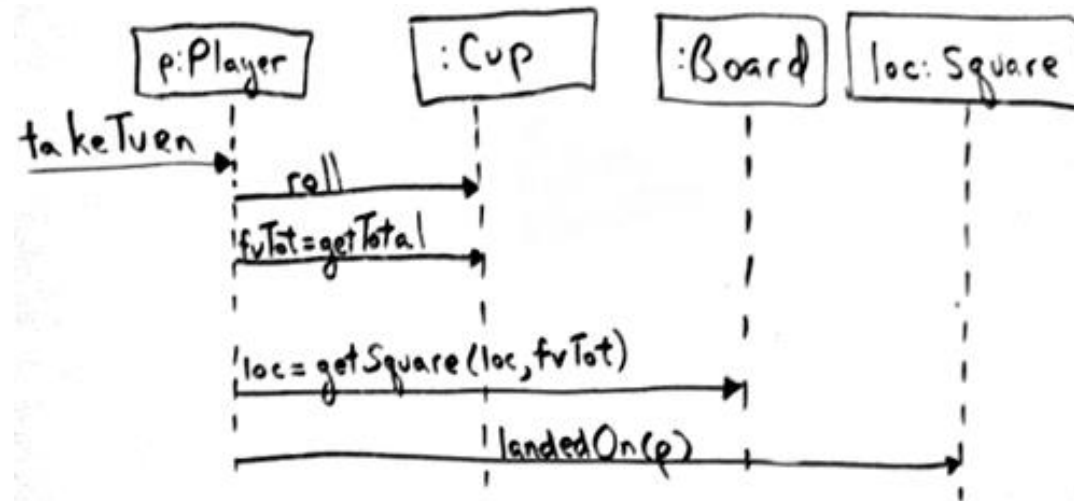
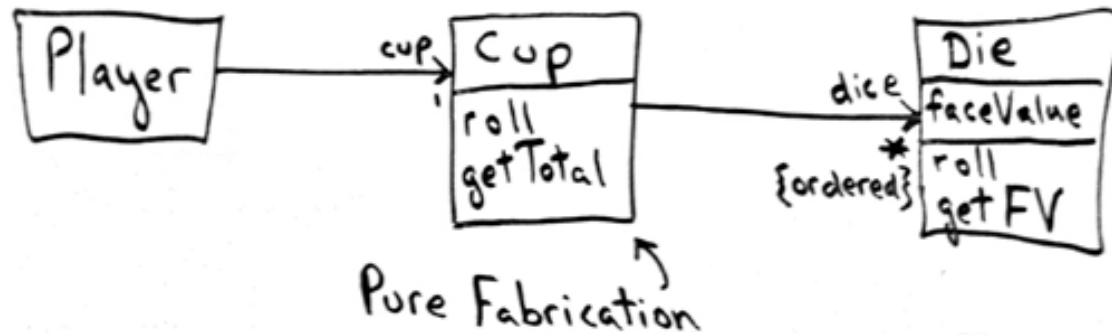
Cup hold all the dice, ***roll*** them onto a table, and know their total

Cup holds a collection of many **Die** objects.

When one sends a ***roll*** message to a **Cup**, it sends a ***roll*** message to all its dice

Monopoly

- Handling Dice
- Solution



Design of Objects

– Two Groups

1. **Representational Decomposition**
2. **Behavioral Decomposition**

Design of Objects

– Two Groups

1. **Sale** is by **representational decomposition**; software class represents a domain concept (**LRG**)

2. **Grouping behaviors or by algorithm**, without much concern for (**LRG**) class with a **real-world domain concept name**, **behavioral decomposition**

e.g. **PersistentStorage**, **ProcessSaleHandler**

Design of Objects

—Two Groups

In a Word document software, “algorithm” that is a **TableOfContentsGenerator**

- Generates table of contents (helper/convenience class) with no concern using domain vocabulary of books and documents

To contrast, a software class named **TableOfContents** is inspired by [representational decomposition](#)

Design of Objects

—Two Groups

1. Some software classes are inspired by representations of the domain
2. Some are simply made up designed to group together some common behavior

Pure Fabrication is usually based on related functionality,
function-centric or behavioral object

Examples of Pure Fabrications (behavioral design patterns):
Adapter, Strategy, Command, etc.

Overusing Pure Fabrication

Functions just become objects

**Needs To Be Balanced With The Ability
To Design With Representational
Decomposition**

Overusing Pure Fabrication

Too many **behavior objects** with responsibilities not **co-located** with the information,
fulfillment which can adversely **affect coupling**

Symptom: Most of the **data** inside the objects is being passed to other objects to **process**

Indirection

Indirection

Problem:

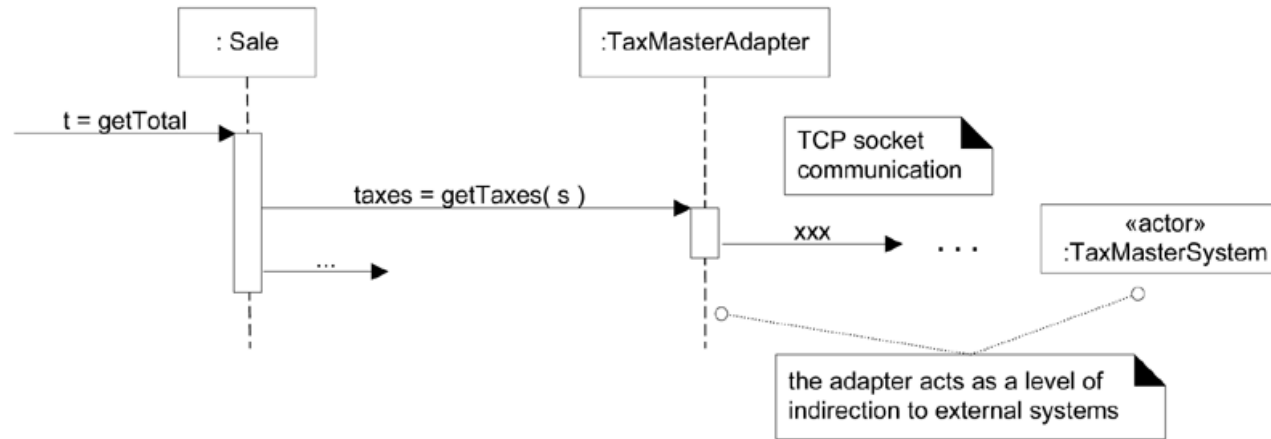
Where to assign a responsibility, to avoid **direct coupling** between two (or more) things?

Solution:

Assign the responsibility to an **intermediate object** to mediate between other components or services so that they are **not directly coupled**

Indirection

—e.g. TaxCalculatorAdapter



Indirection —e.g. TaxCalculatorAdapter

These objects act as **intermediaries** to the external tax calculators

Via polymorphism, they provide a **consistent** interface to the inner objects and hide the **variations** in the external APIs

By adding a level of indirection and adding polymorphism, the adapter

Objects protect the inner design against **variations** in the external interfaces

Indirection

—e.g. PersistentStorage

Pure Fabrication example of decoupling the **Sale** from the relational database services

Through a **PersistentStorage** class which is also an example of assigning responsibilities to support Indirection

PersistentStorage acts as a intermediary between the **Sale** and the DB

Protected Variations

Protected Variations

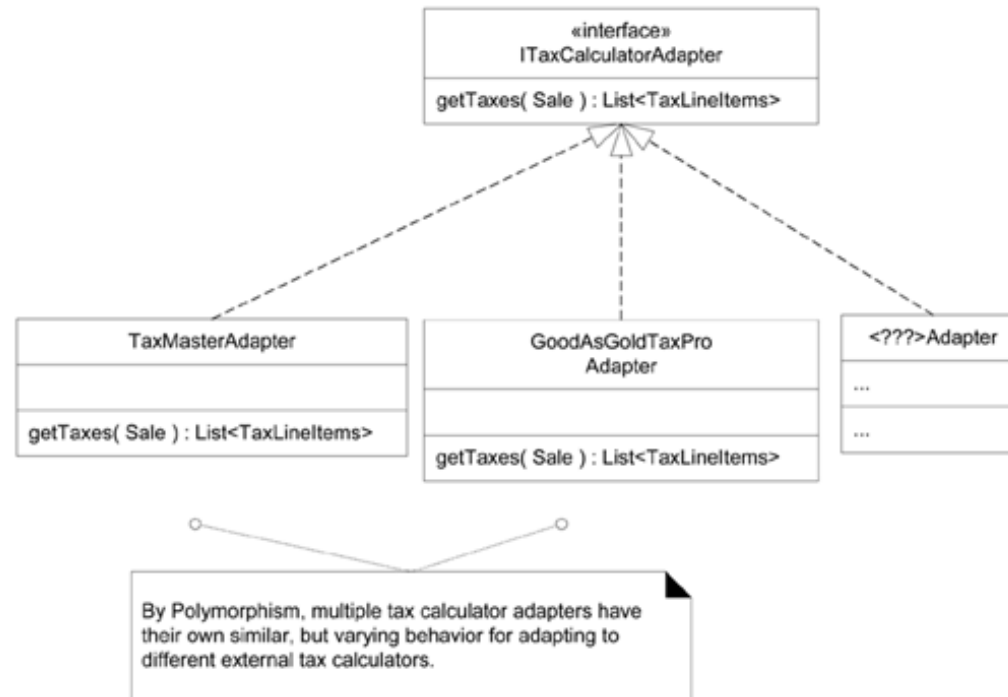
Problem

How to design objects, subsystems, and systems so that the **variations or instability** in these elements does not have an **undesirable impact** on other elements?

Solution

1. Identify points of **predicted variation or instability**;
2. Assign responsibilities to create a **stable interface** around them

External Tax Calculator



External Tax Calculator

Point of instability or variation is **the different interfaces** or **APIs** of external tax calculators

The POS system needs to be able to **integrate** with many **existing** tax calculator systems

And also with **future third-party** calculators not yet in existence

External Tax Calculator

By adding a level of **indirection**,

An **interface**, and using **polymorphism** with **various**
ITaxCalculatorAdapter **implementations**,

protection within the system from variations in external APIs is achieved

Internal objects collaborate with a **stable interface**;

Various adapter implementations **hide the variations** to the
external systems

Mechanisms Motivated by Protected Variations

PV is a root principle motivating most of the mechanisms and patterns in programming and design to:

Provide flexibility and Protection from variations

Variations in data,
behavior,
hardware,
software components,
operating systems, and more ...

Protected Variations, Motivation

Maturation of a developer or architect can be seen in their growing knowledge of ever-wider mechanisms to achieve PV

- To pick the appropriate PV battles worth fighting
- Their ability to choose a suitable PV solution

We learned about

data encapsulation,

interfaces,

and polymorphism—

all core mechanisms to achieve **PV** plus others ...