THE UNIVERSITY OF MELBOURNE

SWEN30006
Software Modelling and Design

# GRASP: More Objects with Responsibilities

Larman Chapter 25

*Luck is the residue of design.*

*—Branch Rickey*

# Objectives

*On completion of this topic you should be able to:*

- Apply the remaining GRASP patterns.

# GRASP Patterns

*Previously we covered five GRASP patterns:*

- Information Expert, Creator, High Cohesion, Low Coupling, and Controller

*We now cover the final four:*

- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

# Polymorphism

## Problem

*How to handle alternatives based on type?*

- Fundamental theme in programs
- Adding new alternatives if using if-then-else or case-statement can require mods in many places

*How to create pluggable software components?*

- E.g. Client-Server relationship:
  - How to replace Server component without affecting Client?

# Polymorphism

**Solution**

When related alternatives vary by type (class):

- use operations with the same interface
- to assign responsibilities for the behaviour
- to the types for which the behaviour varies.

*Corollary:*

- Don’t use conditionals to select the alternative.

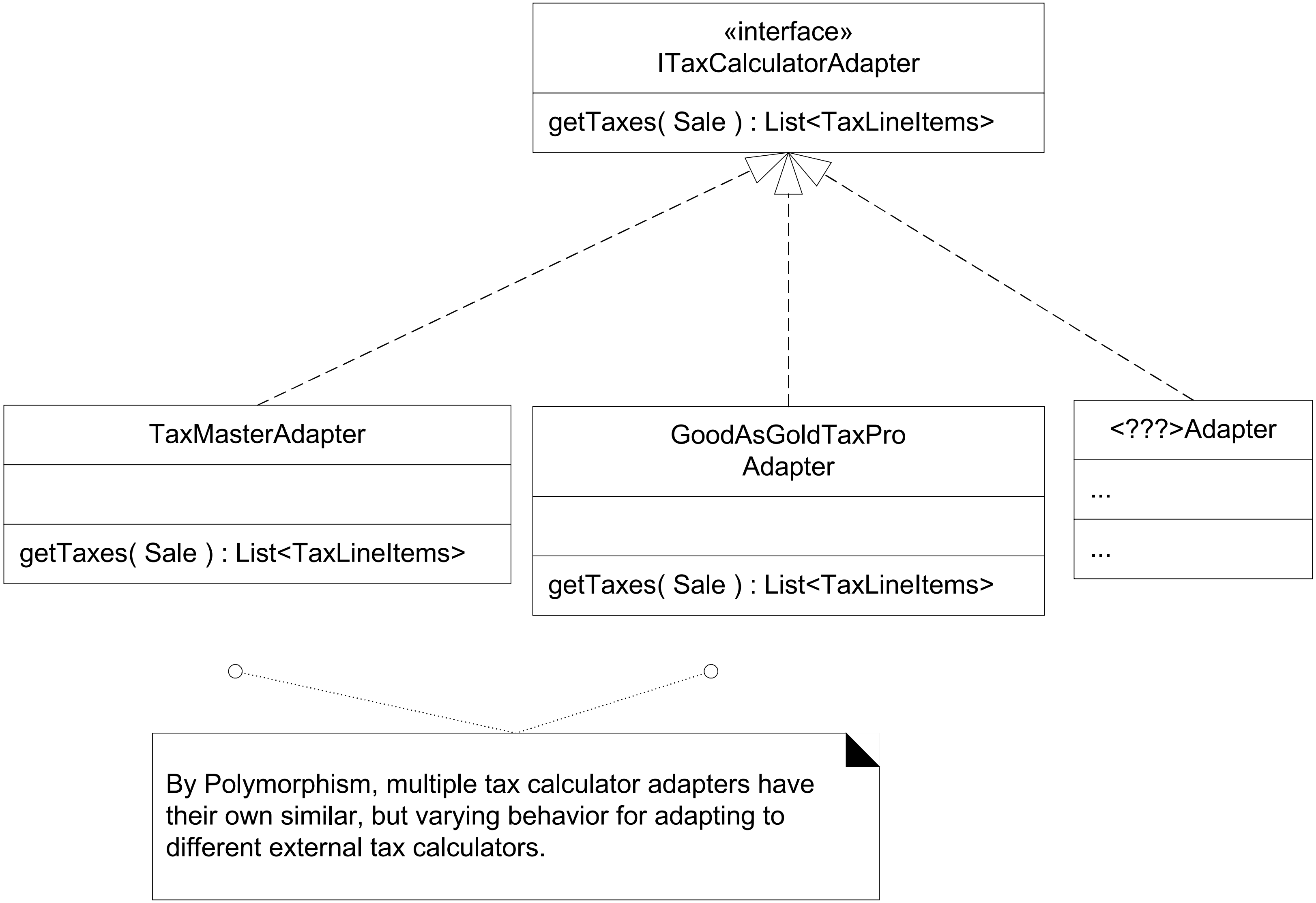
*Polymorphic:*

- giving a single interface to entities of different types.

# Example: NextGen Pos Tax Calcs.

- Multiple external 3rd-party tax calculators which must be supported.
- Each calculator has a different interface, with similar but varying behaviour, e.g
  - Raw TCP socket protocol
  - SOAP interface
  - Java RMI interface
- Objects responsible for handling varying interfaces?
  - Different types of adaptors with different behaviours

# Polymorphism: External Tax Calculators



«interface»
ITaxCalculatorAdapter

getTaxes( Sale ) : List<TaxLineItems>

TaxMasterAdapter

getTaxes( Sale ) : List<TaxLineItems>

GoodAsGoldTaxPro
Adapter

getTaxes( Sale ) : List<TaxLineItems>

<???>Adapter

...

...

By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

# Example: Monopoly Squares

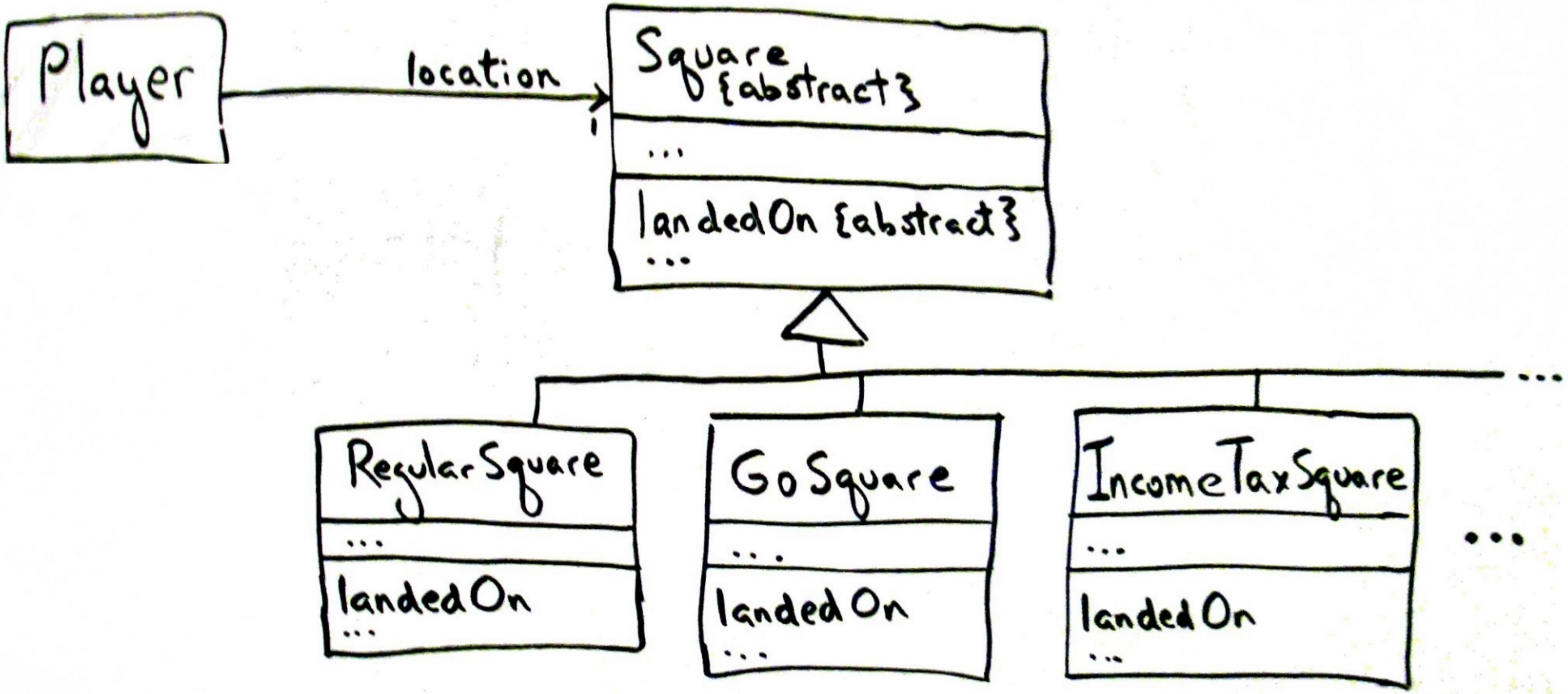
- Different square types on board with different behaviour

*Don’t want:*

```
// bad design for “landedOn”
SWITCH ON square.type
CASE GoSquare: player receives $200
CASE IncomeTaxSquare: player pays tax
…
```

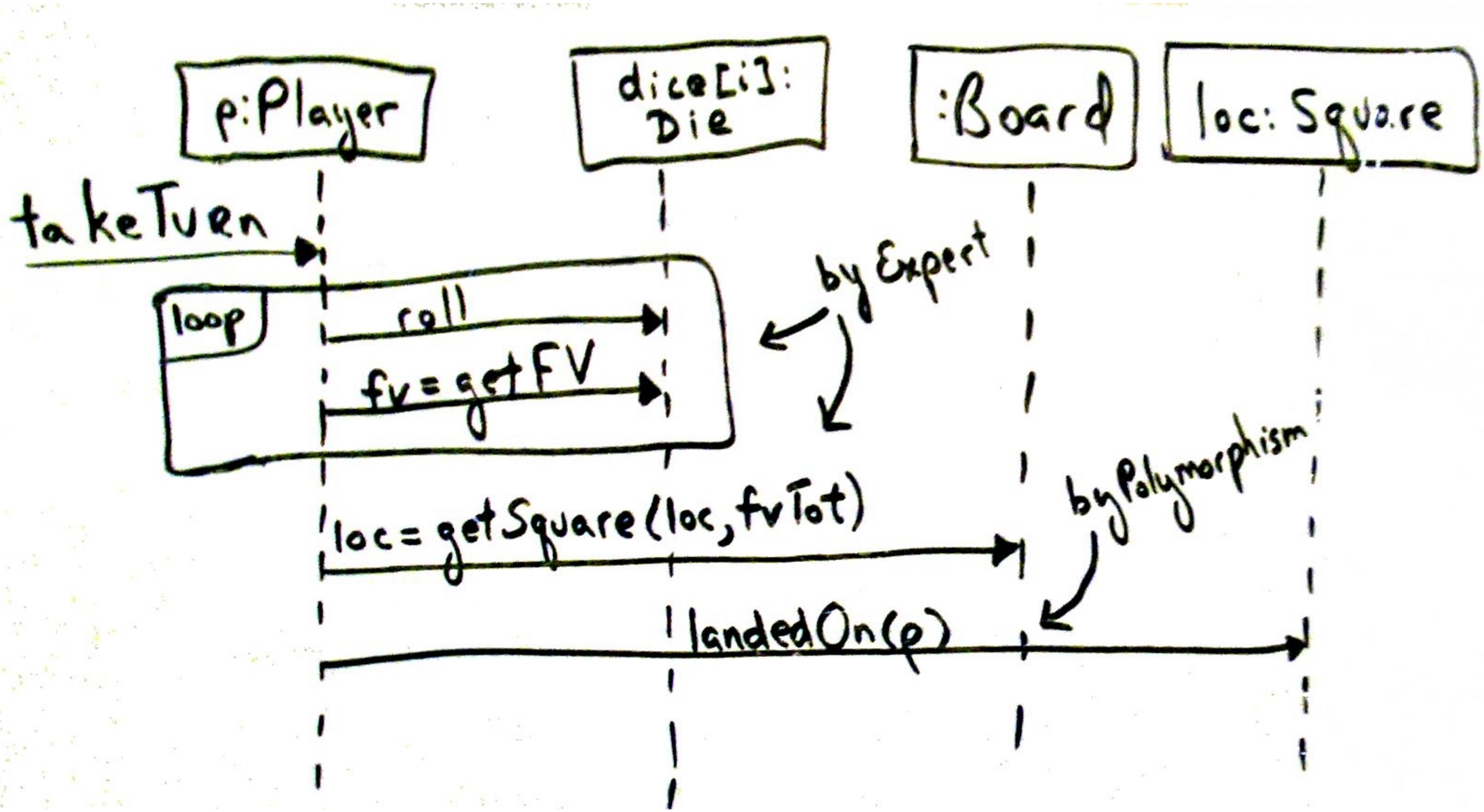
- Want polymorphic op for each type/behaviour

# Polymorphism: Monopoly Squares

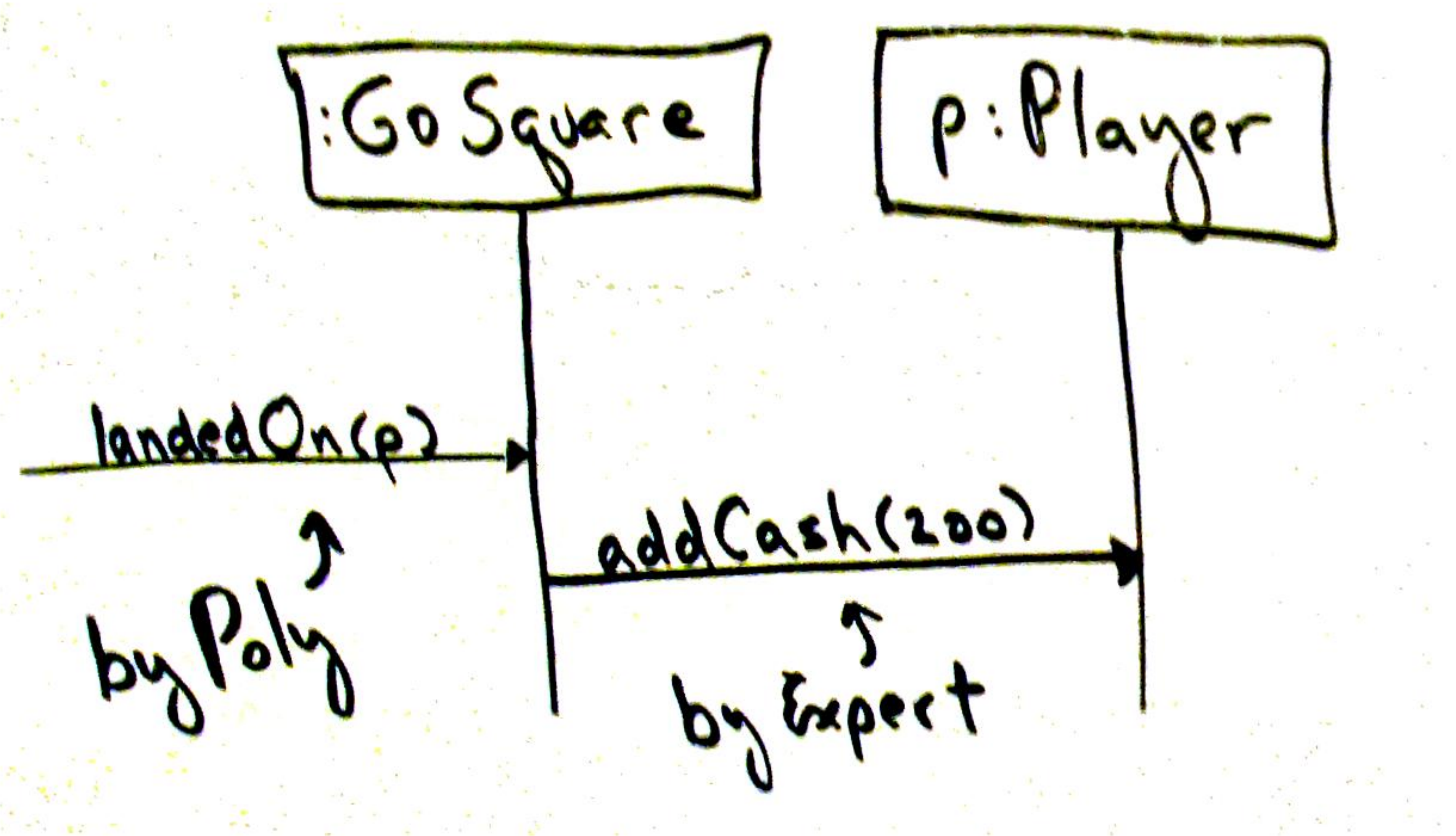


**Guideline:** If no default behaviour, declare superclass poly op *{abstract}*

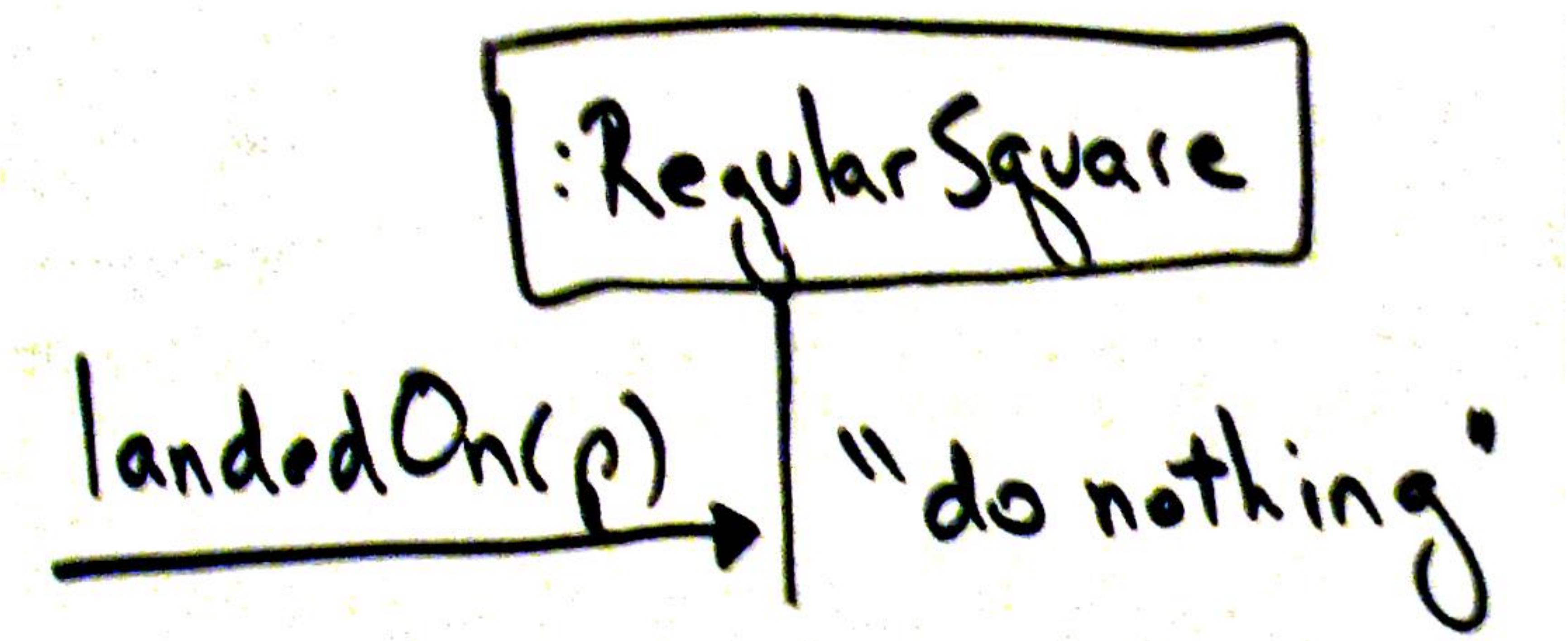
# Polymorphism: Dynamic Behaviour



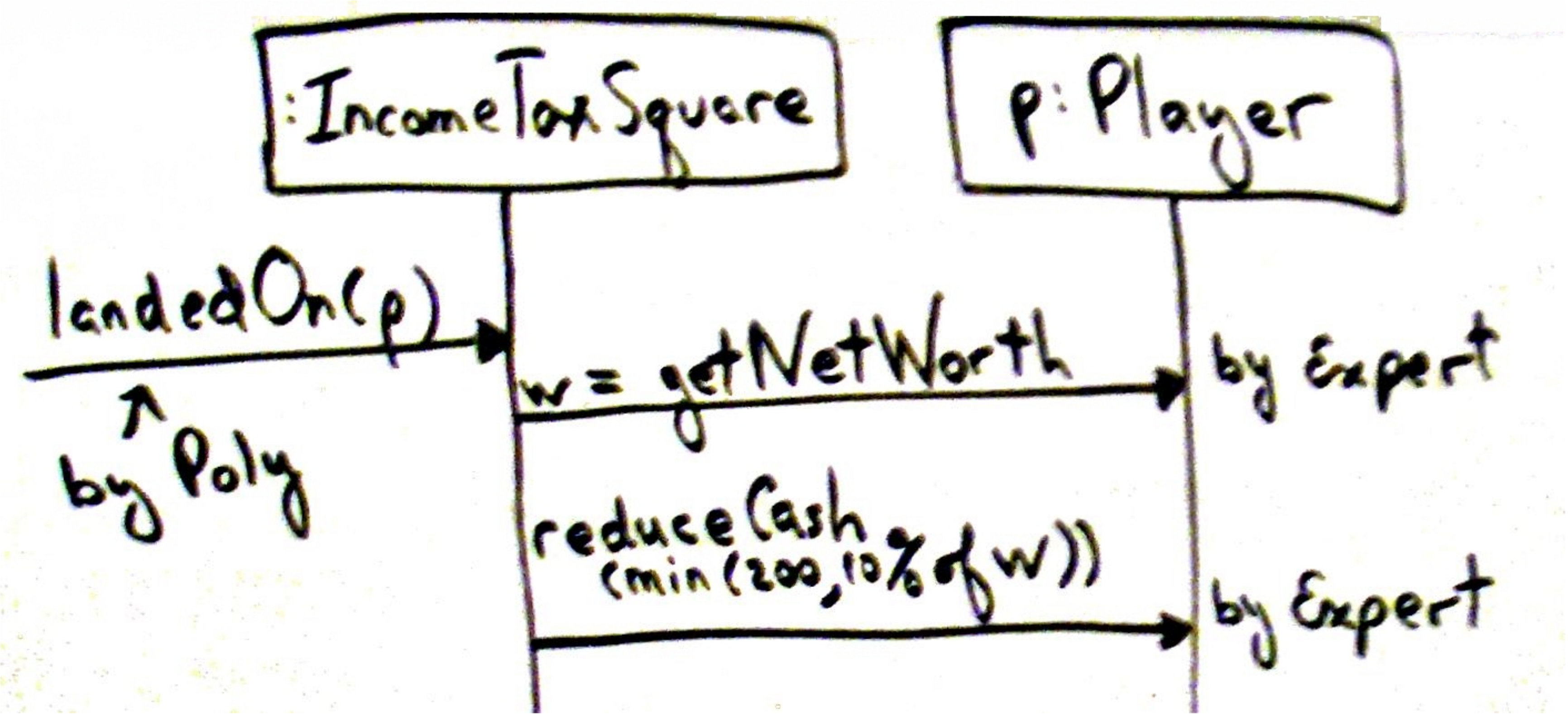
# The *GoSquare* Case



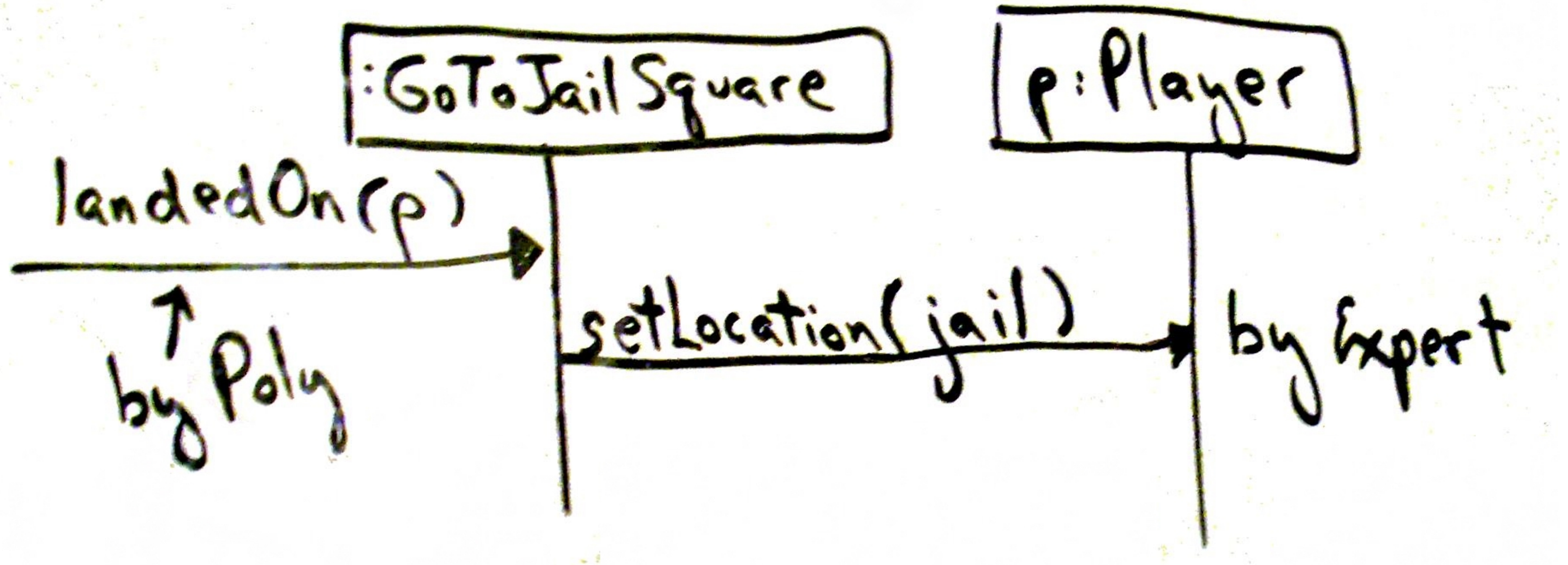
# The *RegularSquare* Case



# The *IncomeTaxSquare* Case



# The *GoToJailSquare* Case



# Polymorphism

## Contraindications

- See contraindications for Protected Variations

# Pure Fabrication

## Problem

*Which object should have responsibility when solutions offered by (e.g.) Expert violate High Cohesion and Low Coupling?*

- Use domain objects in design to lower representational gap
- Sometimes using only domain objects in design result in poor coupling, cohesion, or reuse

# Pure Fabrication

## Solution

Assign a highly cohesive set of responsibilities to a class not in the problem domain:

- Made up to support high cohesion, low coupling, and reuse.
- Fabrication of the imagination, for the purpose of ensuring that the design is very *pure*
- Name is also an English idiom for an intentional lie, something you do when desperate!

*Pattern justifies increasing the representational gap.*

# NextGen POS: DB of Sales

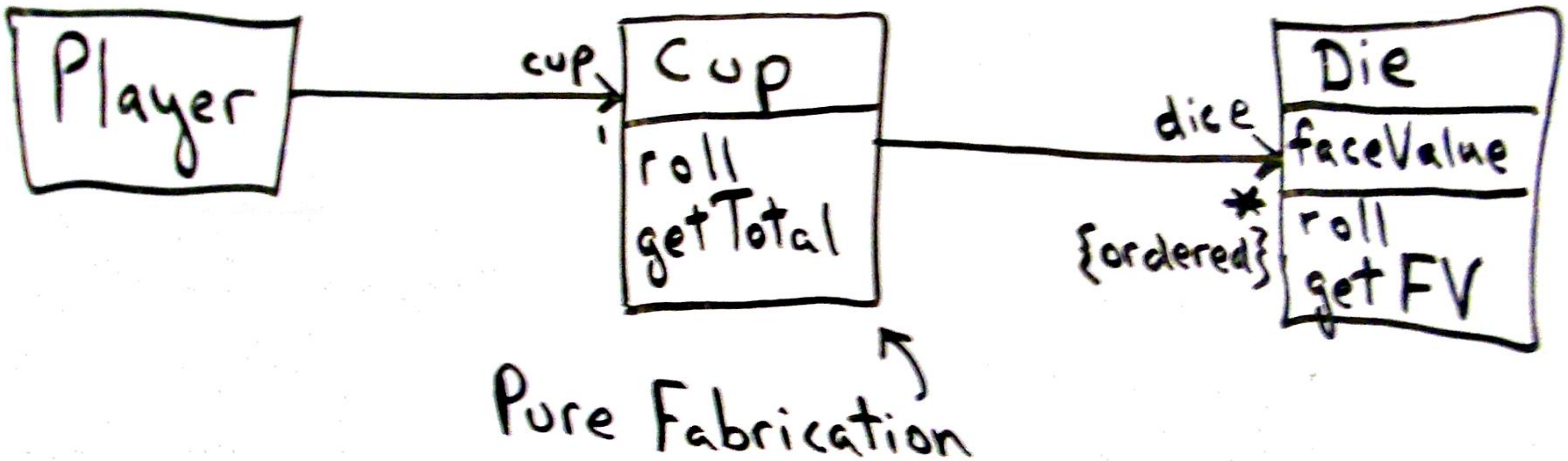
Want to save *Sale* objects in database:

- *Sale* is Information Expert, *but*
- Lots of DB-oriented operations required (not related to concept of *Sale*-ness) [low cohesion]
- *Sale* needs access to DB interface [increased coupling] which is tech. specific, not domain-oriented [lower rep. gap]
- Save in DB a general task [poor reuse, duplication]
- *Soln:* Fabricate a PersistentStore
    - insert(Object), update(Object), …

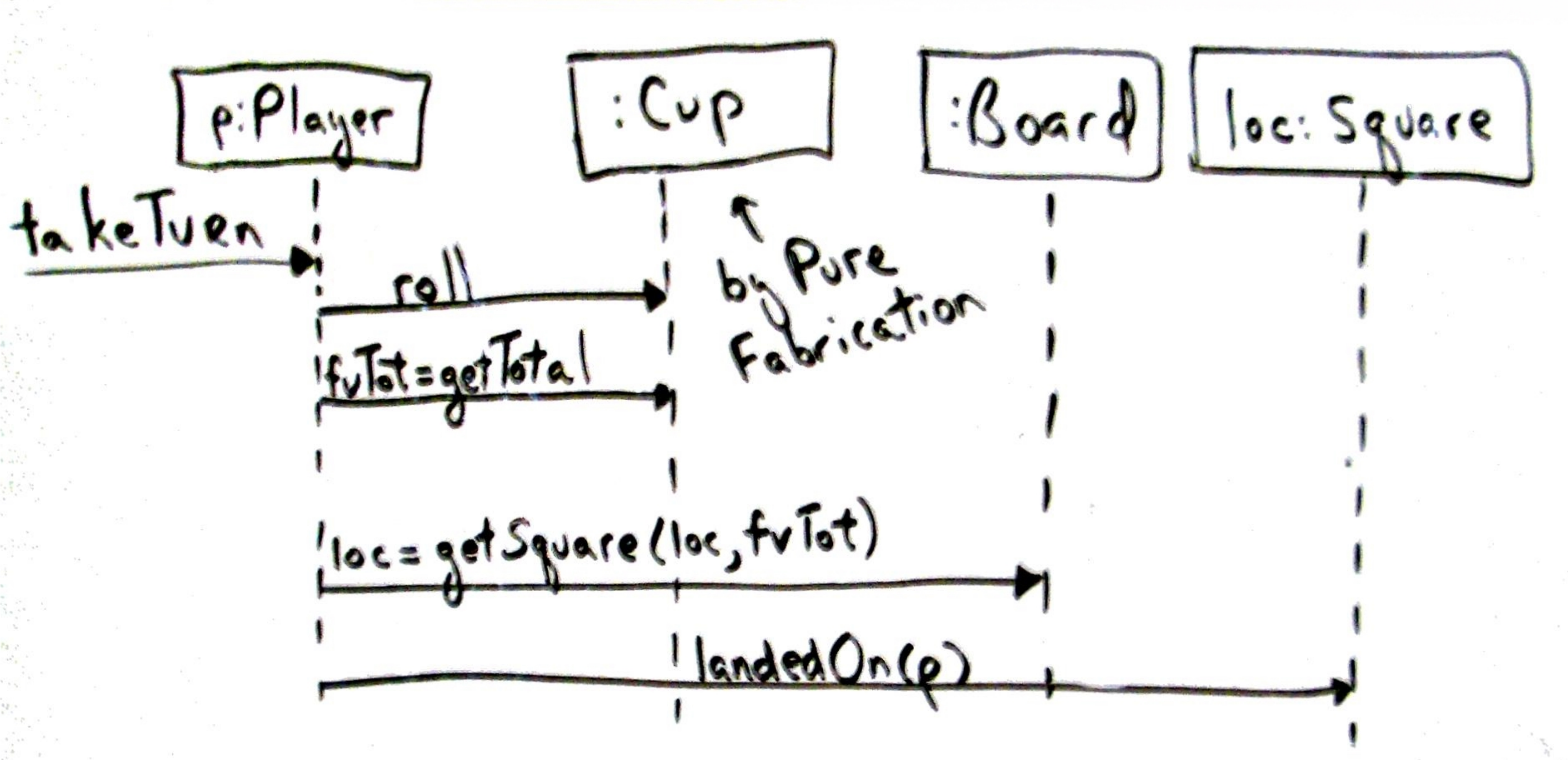
# Monopoly: DCD for a *Cup*

Have reusable *dice* class and *Player:takeTurn* method which has player roll and total dice:

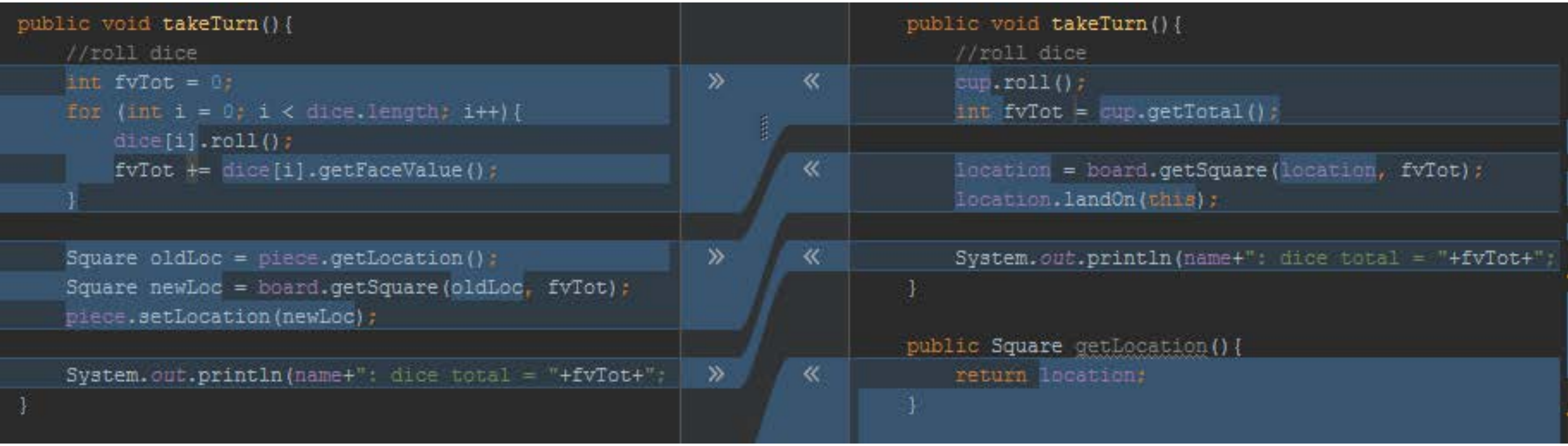
- Need to refer back to dice without re-roll



# Using the Cup in a Monopoly Game



# takeTurn(): Iteration 1 vs 2



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

    Square oldLoc = piece.getLocation();
    Square newLoc = board.getSquare(oldLoc, fvTot);
    piece.setLocation(newLoc);

    System.out.println(name+": dice total = "+fvTot+";
}
```

```java
public void takeTurn(){
    //roll dice
    cup.roll();
    int fvTot = cup.getTotal();

    location = board.getSquare(location, fvTot);
    location.landOn(this);

    System.out.println(name+": dice total = "+fvTot+";
}

public Square getLocation(){
    return location;
}
```

# Pure Fabrication

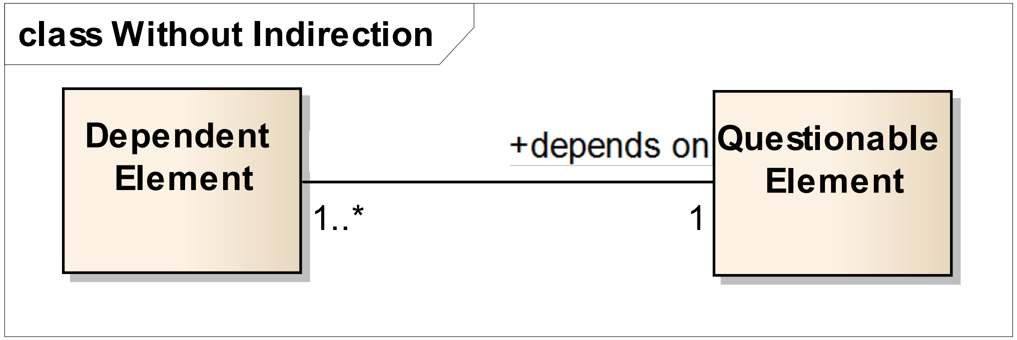
## Contraindications

- Sometimes overused as excuse to add new objects
- Can be driven by behavioural decomposition into functions, resulting in functions just being grouped into objects.
- Needs to be balanced with representational decomposition: assigning responsibilities to classes that have the required information
- Poorly applied, can adversely affect coupling

# Indirection

## Problem

- *Where to assign a responsibility to avoid direct coupling between two (or more) s/w elements?*
- *How to de-couple objects so that low coupling is supported and reuse potential remains higher?*

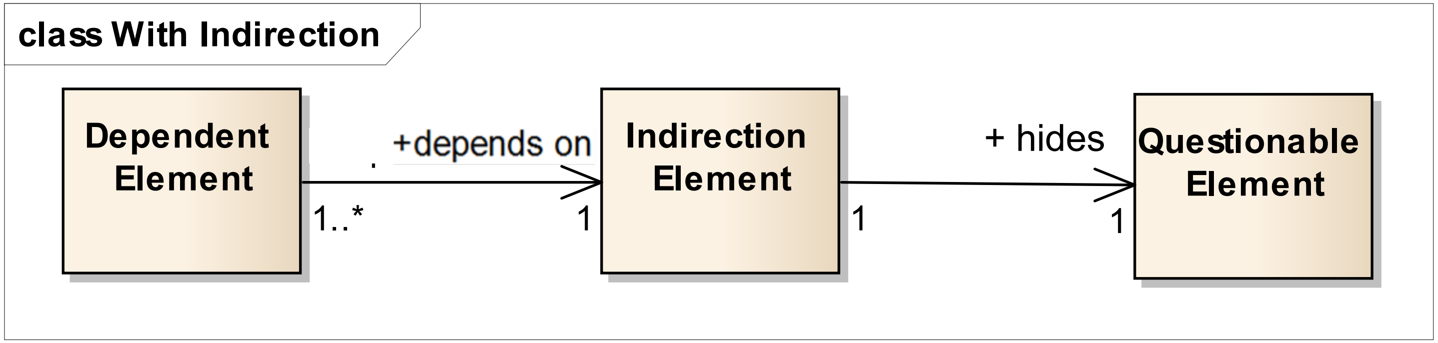


# Indirection

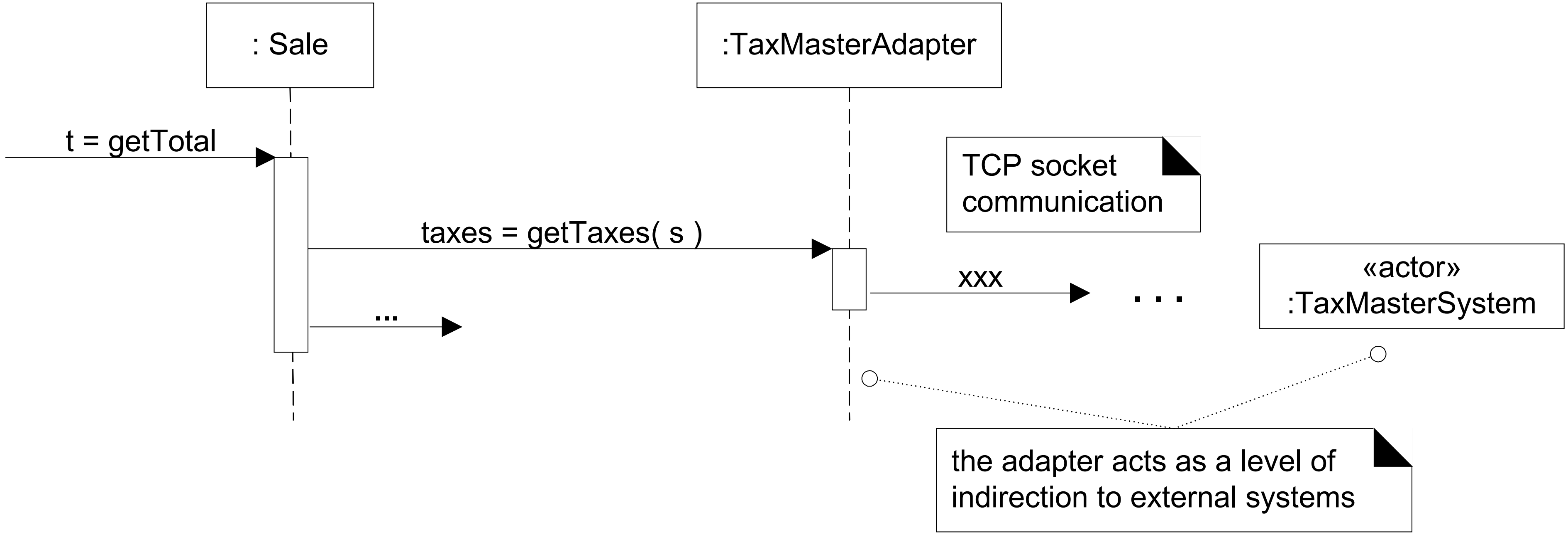
## Solution

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

- Intermediary creates an *indirection* between the other components.



# NextGen POS: Indirection via the Adaptor



# Indirection

*Old adage:*

> “Most problems in computer science can be solved by another level of indirection”

**Contraindications**

- Higher complexity in design needs to be justified by the lower coupling
- *Counter adage:*

> “Most problems in performance can be solved by removing another layer of indirection!”

# 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?*

Points of change include:

- *variation point:* variations in existing system or requirements (e.g. multiple tax calculators)
- *evolution point:* speculative variations that may arise in the future

# Protected Variations

## Solution

- Identify points of known or predicted variation or instability
- Assign responsibilities to create a stable interface* around them

**Interface:* a means of access (not only a programming language interface or Java interface)

# Mechanisms motivated by PV

- ❑ Core Protected Variations Mechanisms
- ❑ Data-Driven Design
- ❑ Service Lookup
- ❑ Interpreter-Driven Design
- ❑ Reflective or Meta-Level Designs
- ❑ Uniform Access
- ❑ Standard Languages
- ❑ The Liskov Substitution Principle (LSP)
- ❑ Structure-Hiding Designs

***See: Larman p428-432***

# Open-Closed Principle (OCP)

> Modules should be both open (for extension; adaptable) and closed (to modification that affects clients). – *Bertrand Meyer*

- OCP “module” includes methods, classes, subsystems, etc.
- E.g. A class can be closed w.r.t. instance field definitions (restricted to access methods only), but open to modification of the definitions.
- OCP and PV are essentially two expressions of the same principle, with different emphasis

# Protected Variations

## Contraindications

- ❑ Cost of speculative “future-proofing” at evolution points can outweigh the benefits
  - It may be cheaper/easier to rework a simple “brittle” design

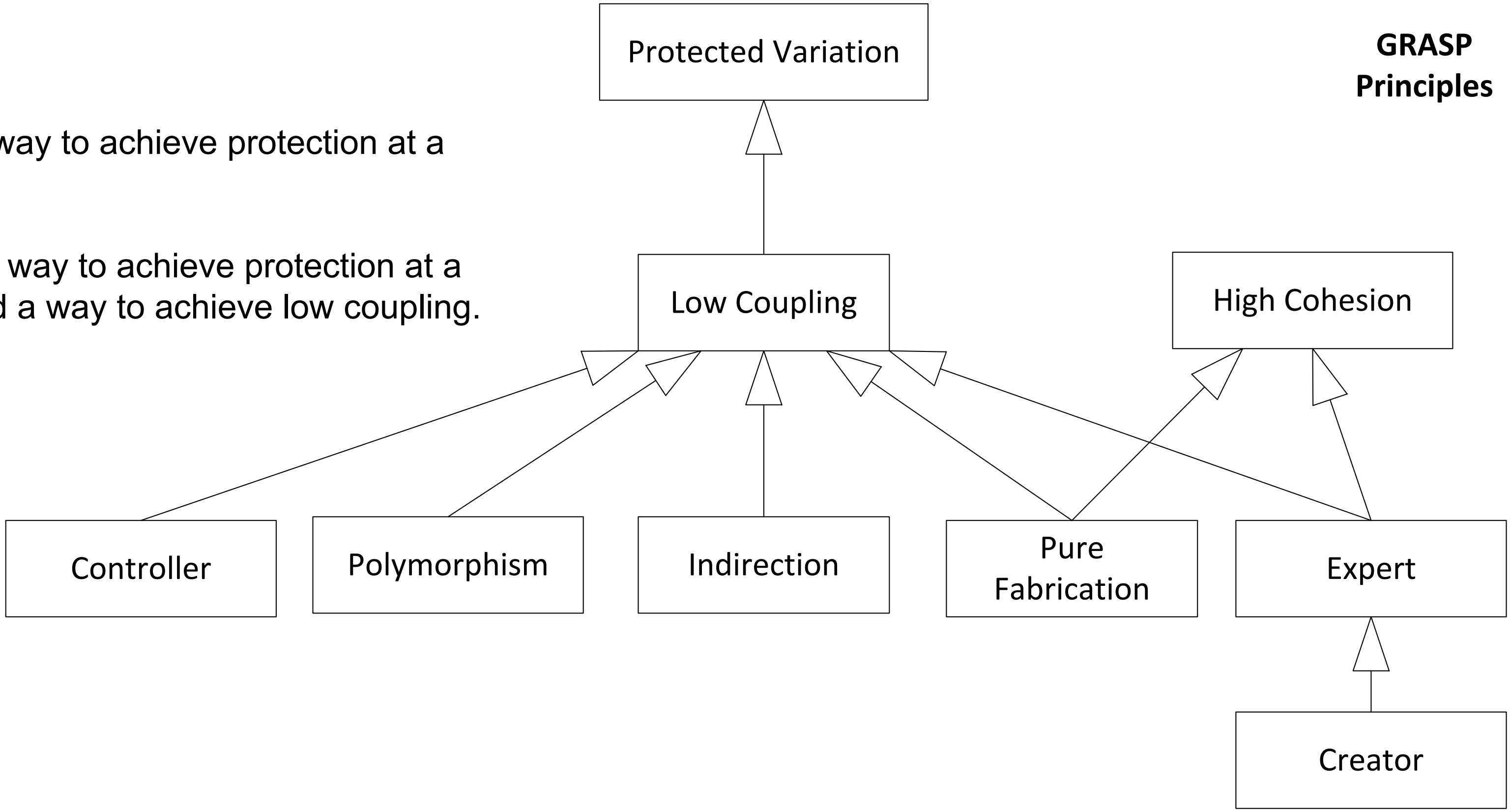
- *Novice designers* tend toward brittle designs
- *Intermediate designers* tend towards overly general and flexible designs (in ways not used)
- *Expert designers* get the balance right

# Relationship between GRASP Principles

*For example:*

Low coupling is a way to achieve protection at a variation point.

Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.


Protected Variation
GRASP Principles
Low Coupling
High Cohesion
Controller
Polymorphism
Indirection
Pure Fabrication
Expert
Creator