Chapter 25

More Design Patterns

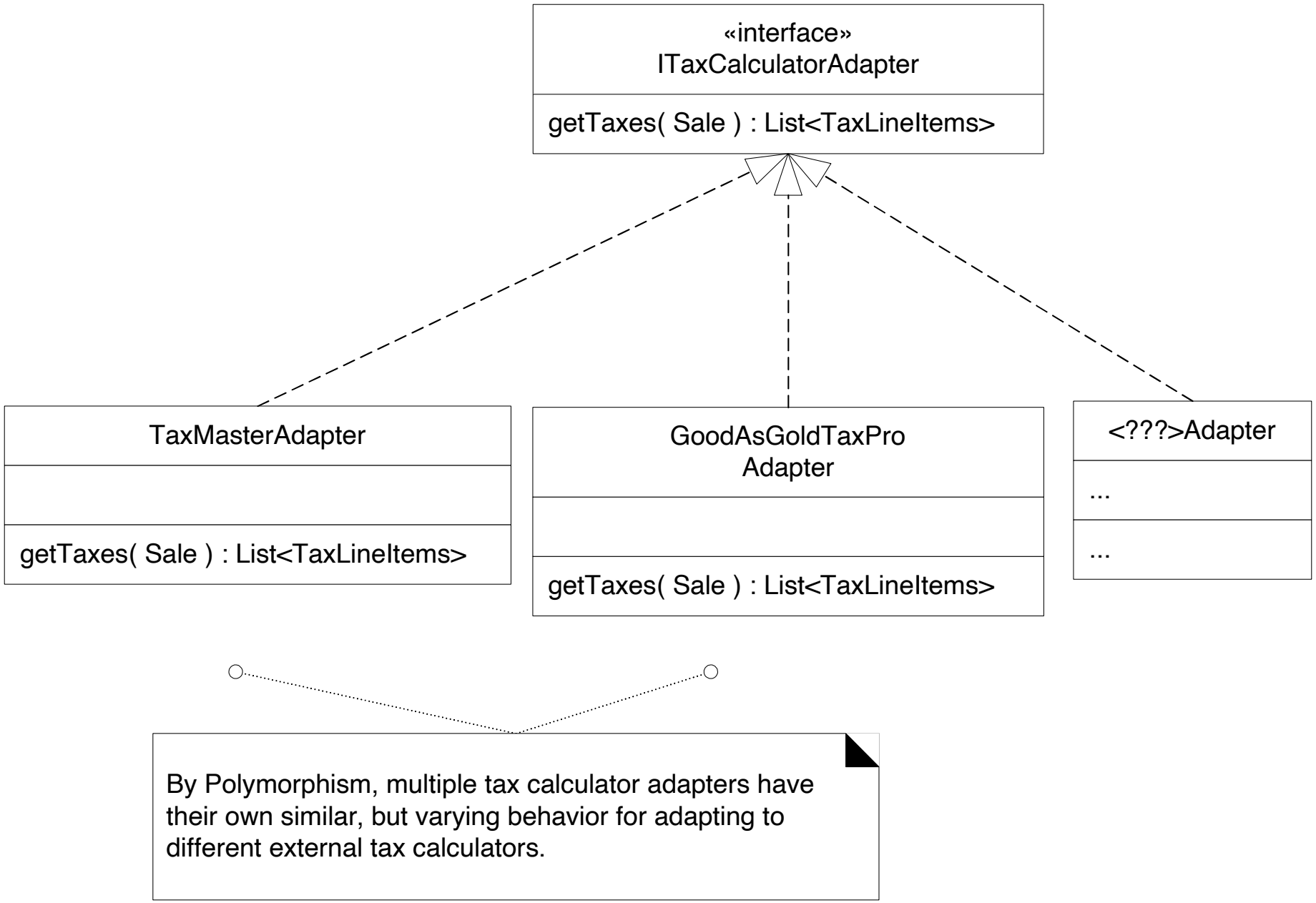
Polymorphism

- Issue: Conditional variation
  - If-then-else or switch statements
  - New variation or case:
    - Conditional statements change
    - Often in many places (methods, classes)
  - Makes it difficult to extend program, add new variations
- Examples
  - Alternatives based on type
  - Pluggable software components:
    - Replace one server with another
    - Without affecting the clients

Polymorphism

- Solution
  - Do not test for the type of an object and use conditional logic
  - Use polymorphic operations instead
- Example: Third-party tax calculator programs in the NextGen POS system
- We want to use an “adapter class”
  - We call adapter’s methods, adapter converts it to tax calculator’s methods
- We’d like our code not to depend on which tax calculator we use
- We’d like to avoid conditional statements in the adapter class
- Solution: Use one adapter class per tax calculator
  - Use polymorphism

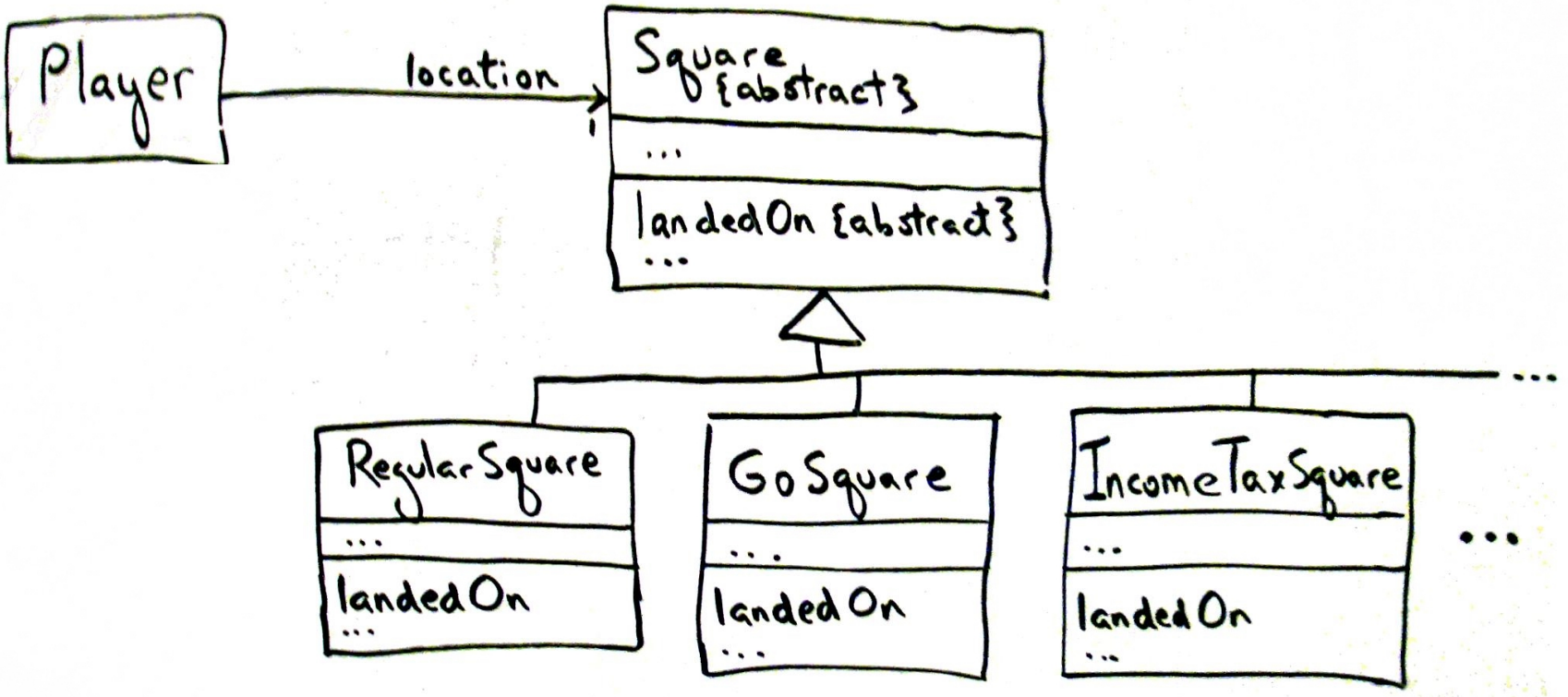
Adapter classes implement iTaxCalculatorAdapter interface



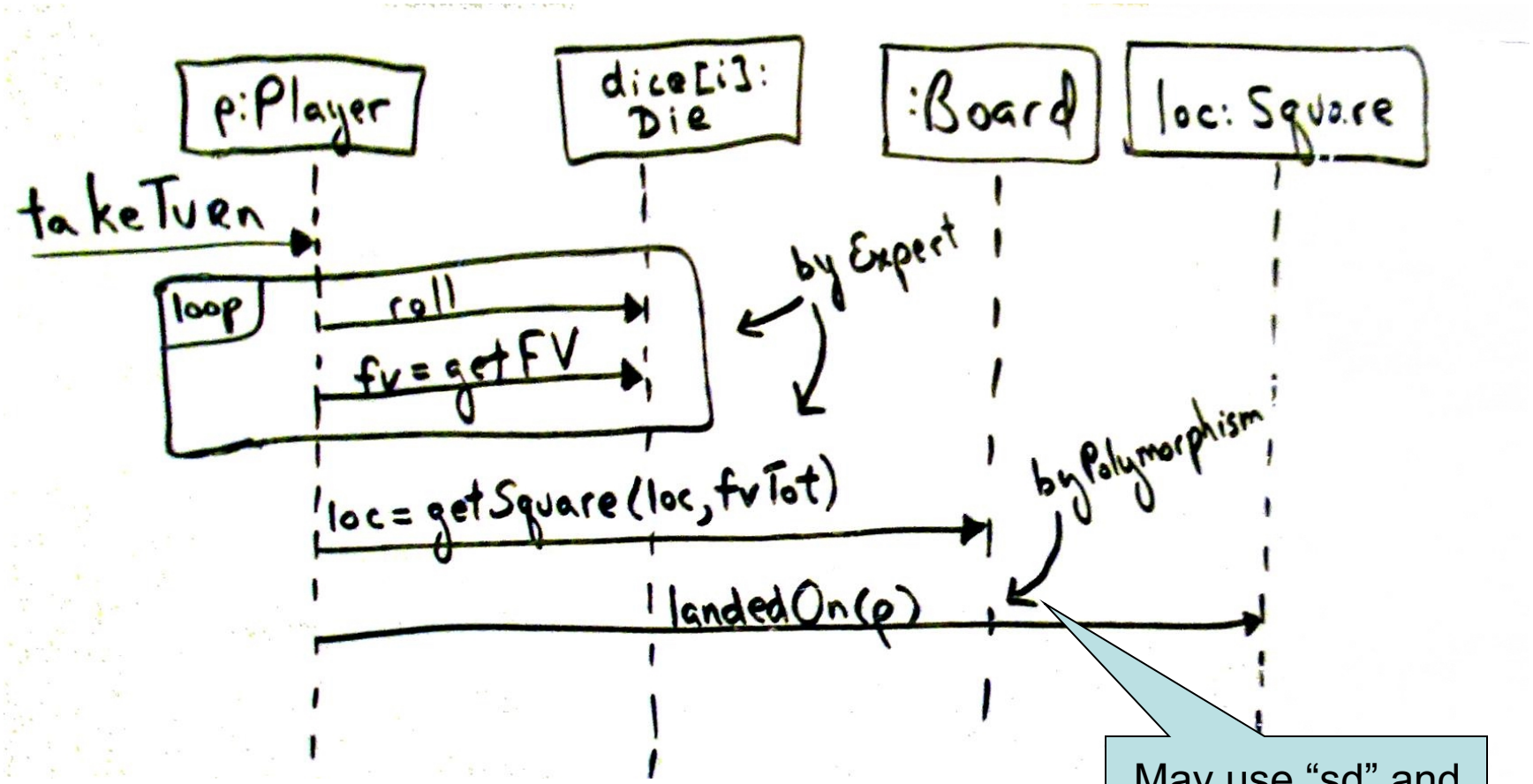
Polymorphism in Monopoly

- Issue: Different actions need to be performed when player lands on different types of square
- Examples:
  - Go square: Get $200
  - Tax square: Pay income tax (10% or $200)
  - Regular square: Do nothing (for now)
- Solution: Use abstract superclass, Square
  - RegularSquare, TaxSquare, IncomeTaxSquare subclasses

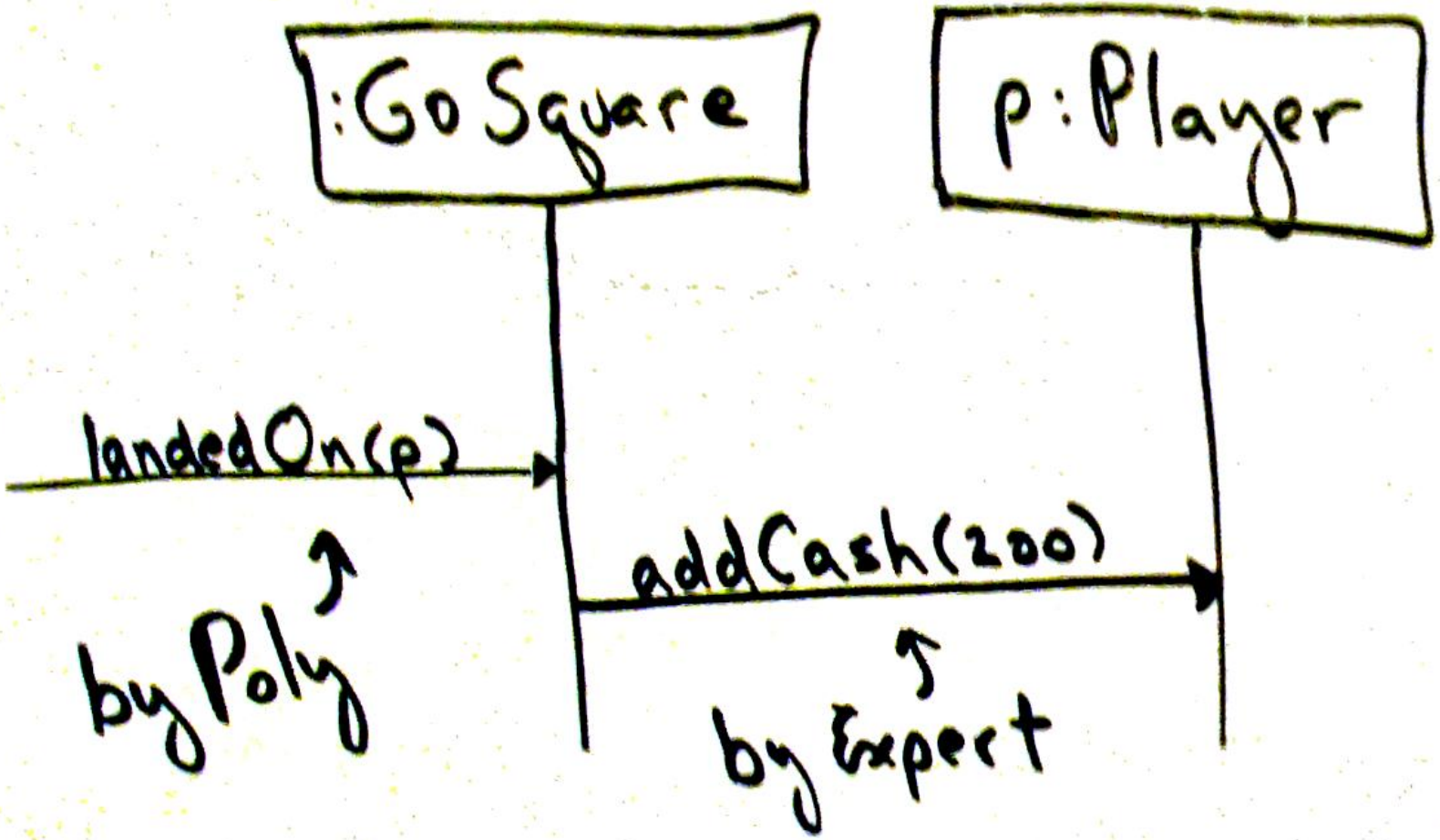
Polymorphism in Monopoly



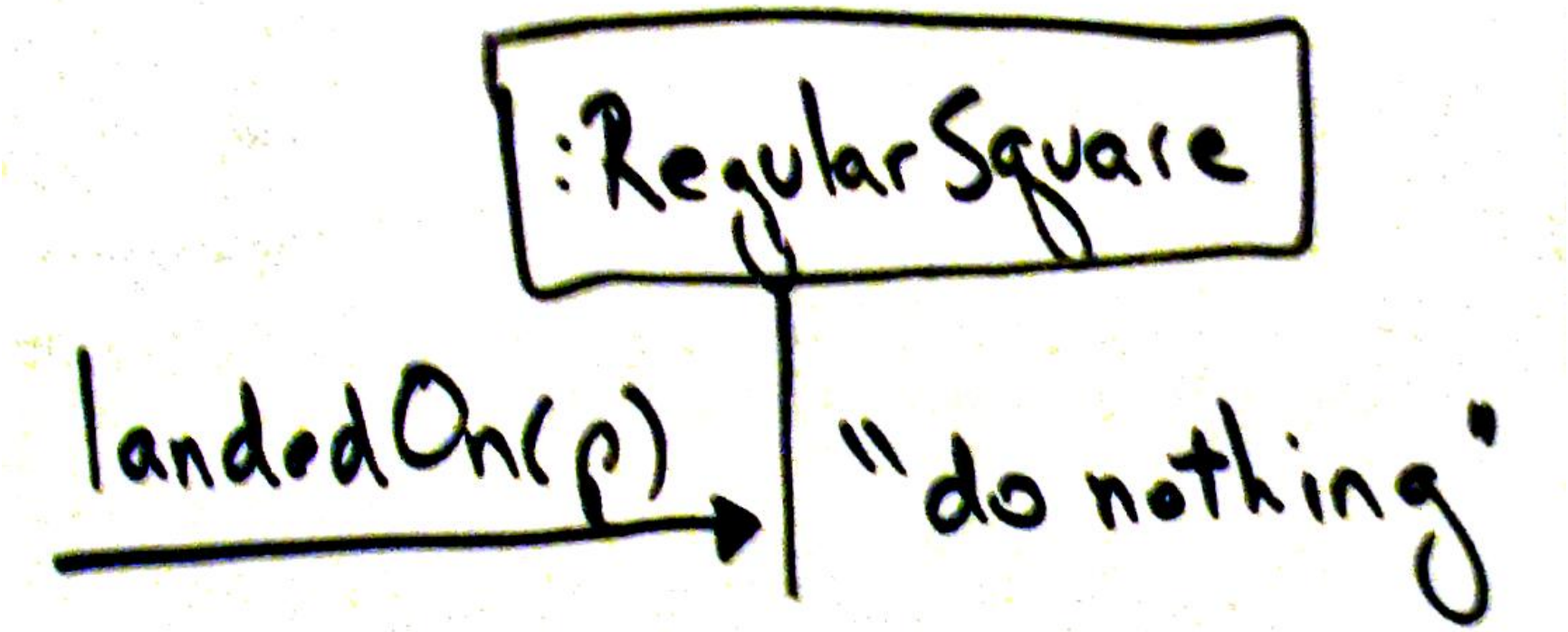
Issue: How to adapt interaction diagrams to this change?



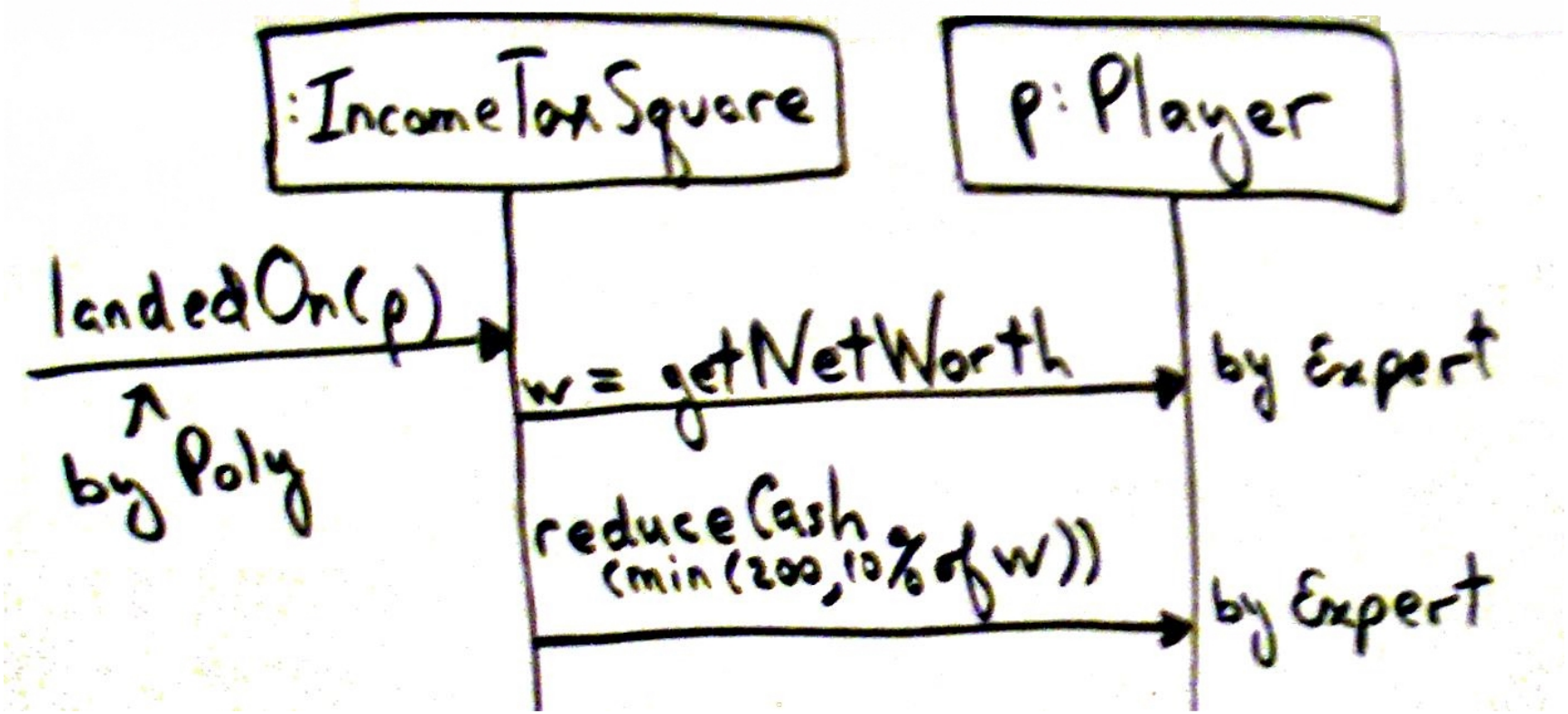
One Polymorphic Case



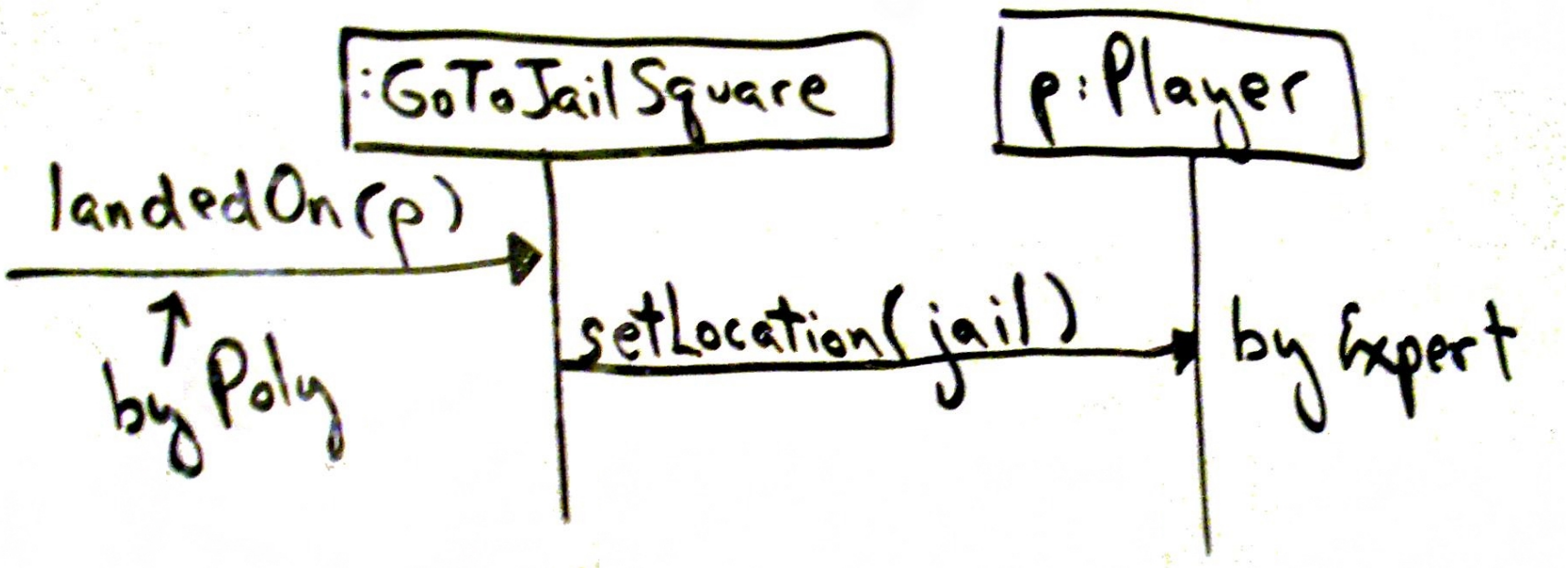
Another polymorphic case



Another polymorphic case



Another polymorphic case



- Notice design change:
  - Now Player knows location, instead of Piece
  - Decreases coupling
  - In fact, we realize “piece” class not really needed

When to use interfaces vs. abstract superclasses?

- Use interfaces whenever you can
  - Becauses of single inheritance, abstract classes are restrictive
  - Unless there are a lot of default method implementations inherited – then use abstract classes

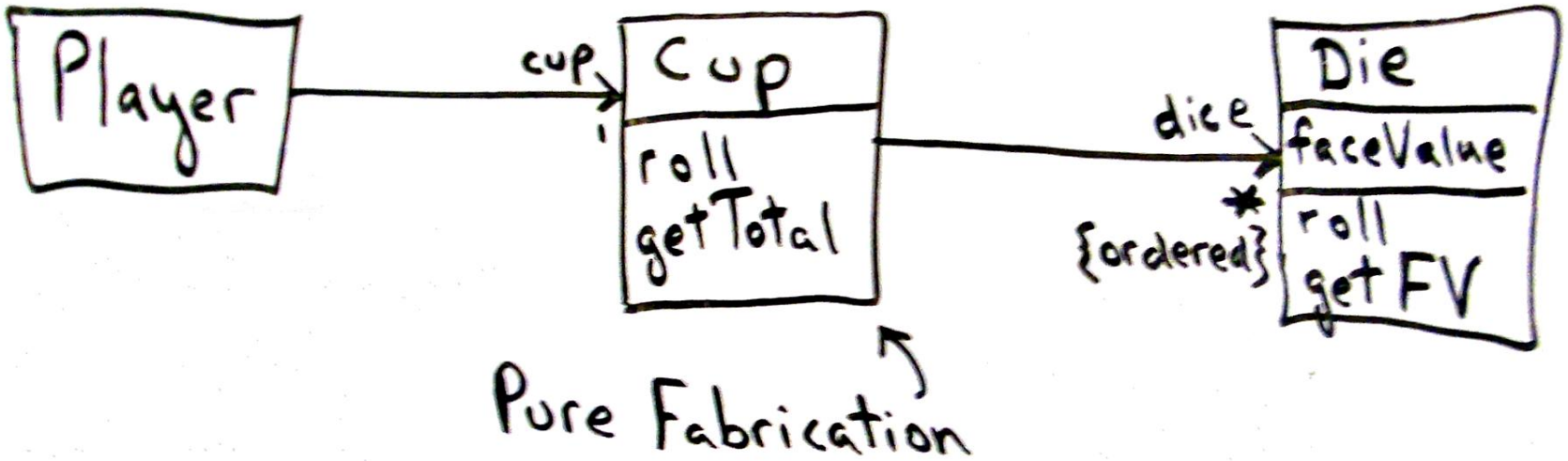
Design Pattern: Pure Fabrication

- Idea: Make up ("fabricate") new class with no direct equivalent concept in the domain
  - When other design patterns do not offer solutions with low coupling and high cohesion
- Recall example: Saving a Sale object in a database
- What goes wrong if "Sale" saves itself?
  - Task requires a lot of supporting database operations
  - Sale has to be coupled to the db interface
  - Saving to db a general task
    - A lot of other classes need the same support
    - Lots of duplication in each object

Solution: New "PersistentStorage" class

- Persistent storage not a domain concept
- We grouped a number of responsibilities into this class
- Similar issue in Monopoly:
  - Player rolls dice: Not reusable in another game
  - Summing of dice done by player
  - Can't ask for dice total without rolling again
- Improved design: Fabricate "Cup" class that does the jobs above

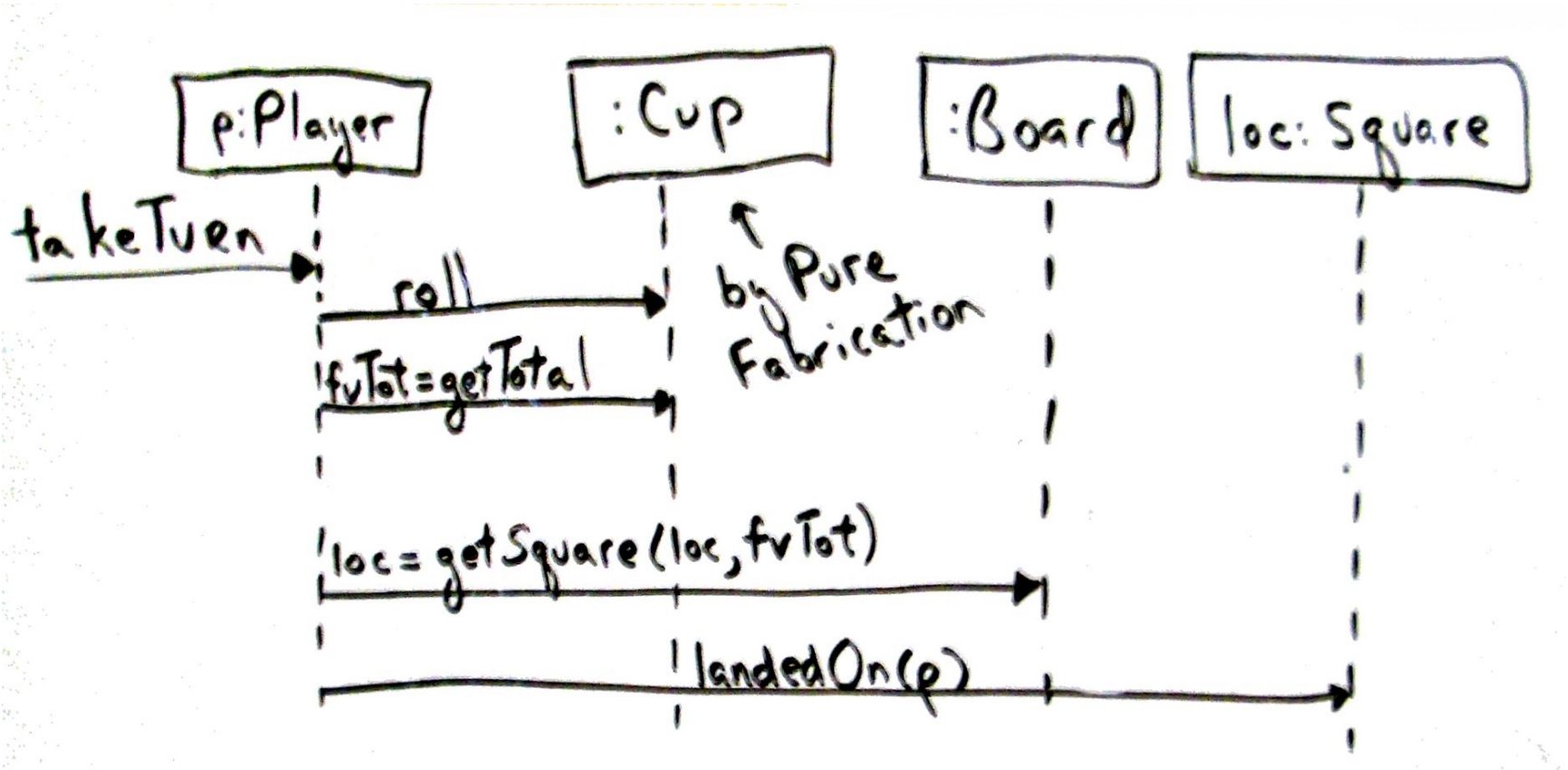
Fabrication in Monopoly: The “Cup” class



Design of Objects

- Design of objects done in one of two ways
  - Objects chosen by representational decomposition
    - Representing domain objects
    - Example:
      - TableOfContents
  - Objects chosen by behavioral decomposition
    - Group a number of related operations into a class
    - Example:
      - Algorithm object: TableOfContentsGenerator
- Don’t overdo fabrication
  - If you put all functionality in one class, you end up writing a C program

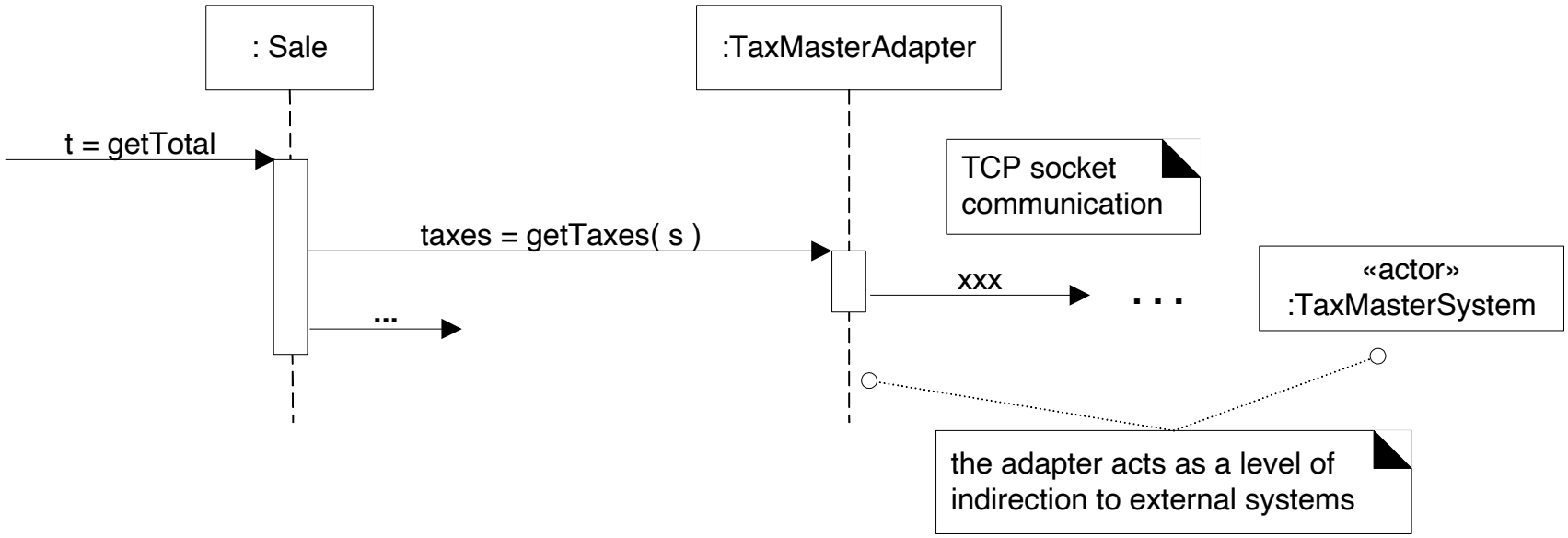
Updated Monopoly Interaction Diagram



Design Pattern: Indirection

- Issue: How to avoid coupling between two or more things
  - Why? One is likely to change a lot. Don't want the other to depend on it.
- Solution: Use indirection. Create intermediate object.
- Example: TaxCalculatorAdapter
  - Our design is independent from the third party tax program API
  - If the API changes, we only modify the adapter.
- Example: PersistentStorage
  - Just change PersistentStorage to use a different database

Indirection via the adapter class/object



Design Pattern: Protected Variations

- Issue: How to design objects, subsystems and systems so that variations or instability in these elements do not have an undesirable impact on others?
- Solution:
  - Identify points of predicted variations and instability
  - Put a stable interface around them to protect the rest of the design
- Example: External tax calculator and the class hierarchy of adapters

Examples of Mechanisms Motivated by Protected Variations

- Data-driven designs:
  - Example: Web-page style sheet read separately from the .html file
- Service look-up mechanisms: JNDI, Jini, UDDI
  - Protects designs from where services are located
- Interpreter-Driven Designs
  - Virtual machines, language interpreters, ...
- Reflective or Meta-Level Designs
  - Java Beans
  - A way to learn what attribute or bean property a "Bean" has and what method to use to access it

The Law of Demeter or "Don't talk to strangers"

- A method should send messages only to the following objects
  - "this" (or self)
  - A parameter of the method
  - An attribute of "this"
  - An element of a collection that is an attribute of "this"
  - An object created within the method
- Everyone else is a "stranger"
  - Don't talk to them!

Don’t talk to strangers

- Example of talking to strangers:

```
public void fragileMethod() {
  AccountHolder holder =
    sale.getPayment().getAccount().getAccountHolder();
}
```

- If any object relationships change along the way, method breaks
- What should we do?
  - Sale should have a method for giving us the account holder.
- Not a problem if object relationships are fixed. Example: Java libraries