# Four More GRASP Principles

Curt Clifton

Rose-Hulman Institute of Technology

Q1

# Four More
# GRASP Principles

- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

# Polymorphism

- **Problem**: How do we handle alternatives based on type? How do we create pluggable software components?
  - Chained *if*s and lots of *switch* statements are a bad code smell → new types require finding conditions and editing
  - Pluggable components require swapping one module for another without changing surrounding design

Q2

# Polymorphism

- **Problem**: How do we handle alternatives based on type? How do we create pluggable software components?
- **Solution**: When related alternatives vary by type, assign responsibility to the types for which the behaviors vary.
  - I.e., Use subtypes and polymorphic methods
  - Corollary: Avoid *instanceof* tests

Q3

# Example

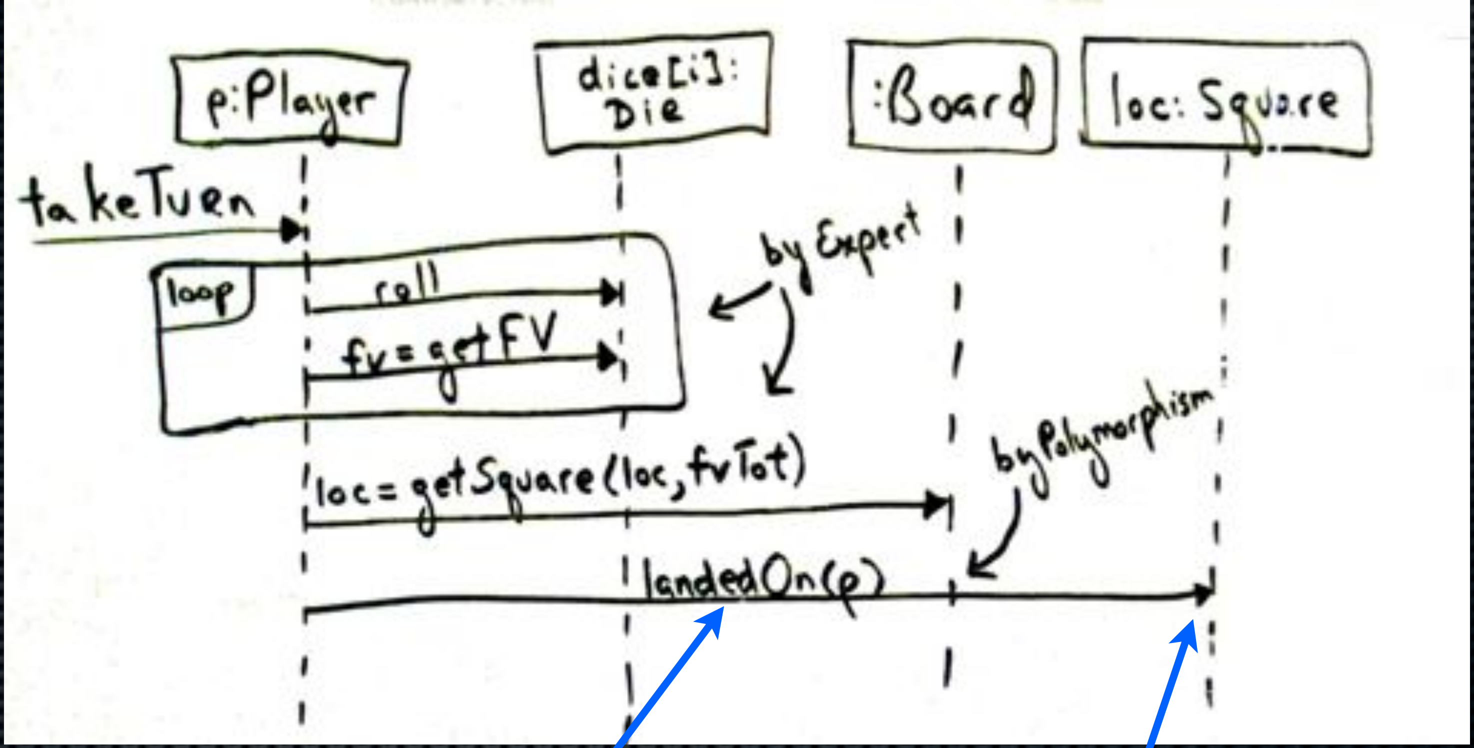
- **Bad**:

```
switch (square.getType()) {
case GO:
        …
case INCOME_TAX:
        …
case GO_TO_JAIL:
        …
default:
        …
}
```

What happens when we need to add other sorts of squares in future iterations?

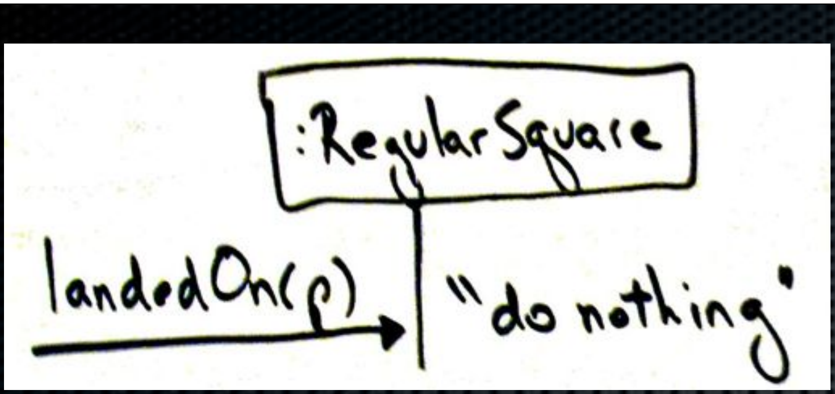
**Solution**: Replace switch with polymorphic method call

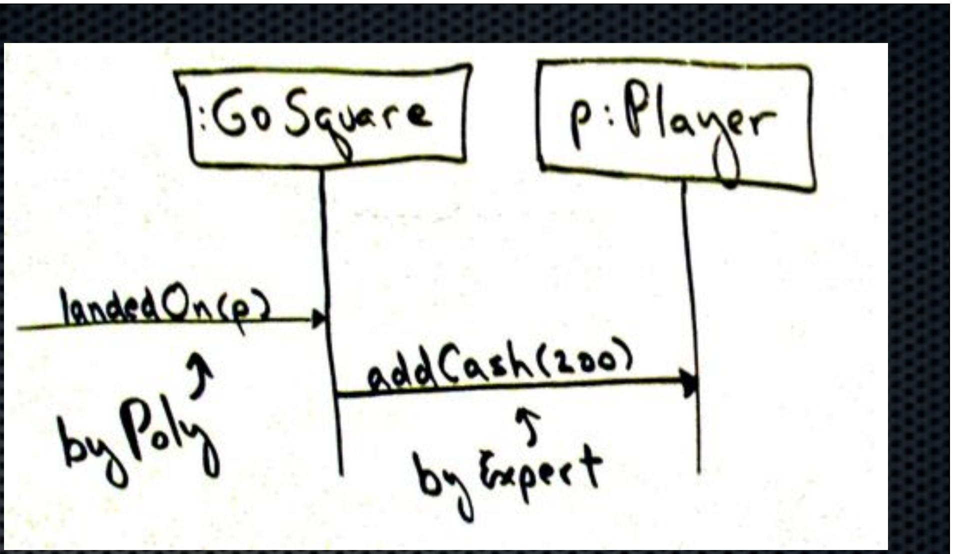
# Example (continued)

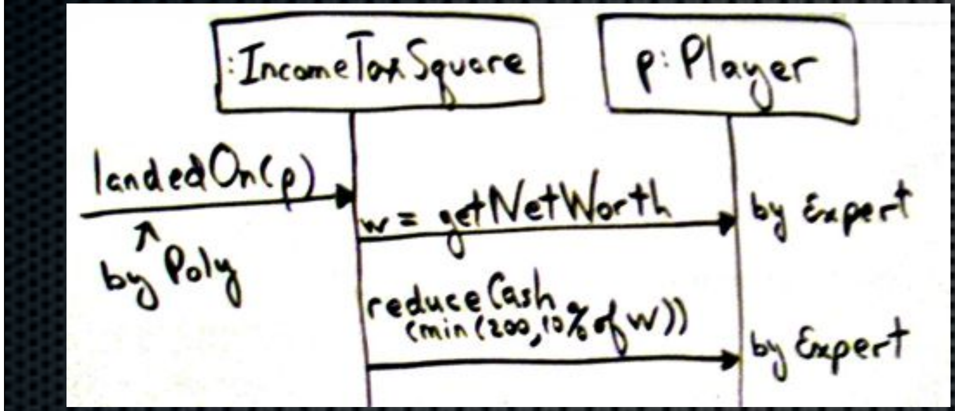


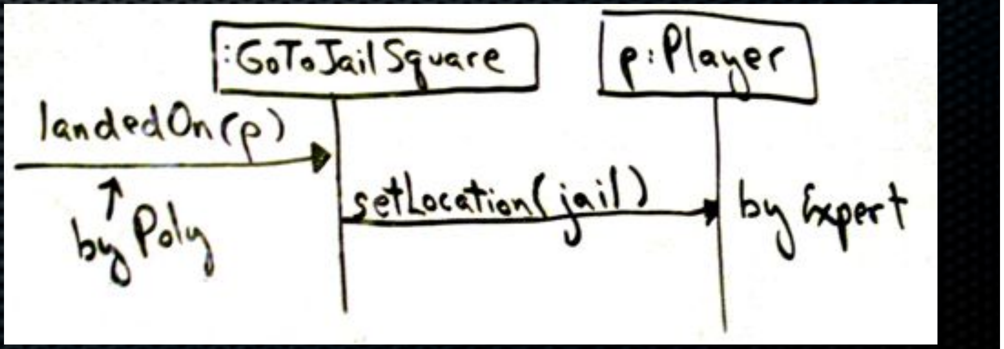
Make abstract unless clear default behavior

Details of polymorphic method drawn separately


:RegularSquare
landedOn(p)
"do nothing"


:GoSquare
p:Player
landedOn(p)
by Poly
addCash(200)
by Expert


:IncomeTaxSquare
p:Player
landedOn(p)
by Poly
w = getNetWorth
by Expert
reduceCash (min(200,10% of w))
by Expert


:GoToJailSquare
p:Player
landedOn(p)
by Poly
setLocation(jail)
by Expert

# Polymorphism Notes

- A design using Polymorphism can be easily extended for new variations
- When should supertype be an interface?
  - Don’t want to commit to a class hierarchy
  - Need to reduce coupling
- Contraindication: speculative future-proofing

Don’t be too clever!

Q4,5

# Team Polymorphism

Q6

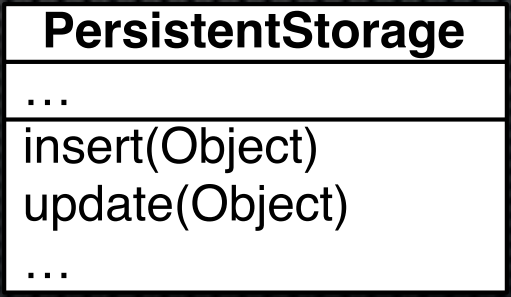
# Pure Fabrication

- **Problem**: What object should have responsibility when solutions for low representation gap (like Info. Expert) lead us astray (i.e., into high coupling and low cohesion)
- **Solution**: Assign a cohesive set of responsibilities to an artificial (not in the domain model) class

Q7,8

# Example

- How might we design for saving a *Sale* object in a database?
  - What does Info. Expert say?
  - Instead, a Pure Fabrication solution:



**PersistentStorage**

…

insert(Object)
update(Object)
…

# Common Design Strategies

- Representational decomposition
- Behavioral decomposition

Pure Fabrications are often behavioral decompositions

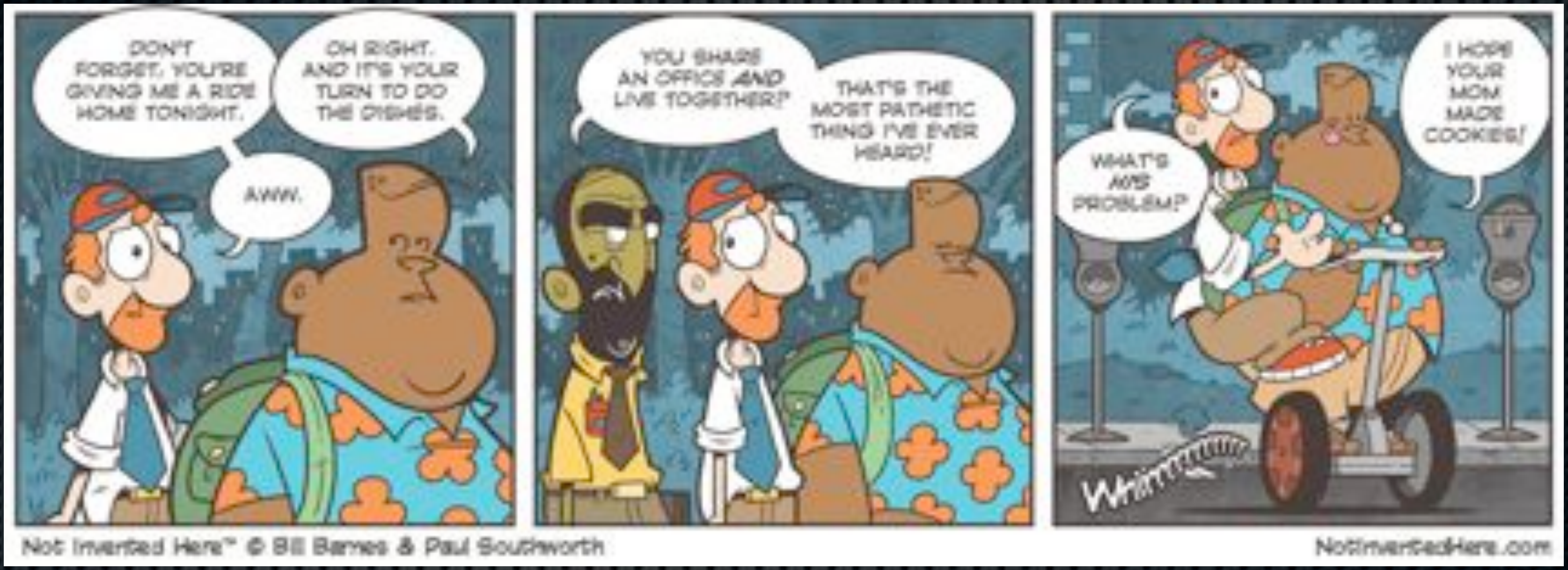
# Notes on Pure Fabrication

- Benefits:
  - Higher cohesion
  - Greater potential for reuse
- Contraindications:
  - Can be abused to create too many behavior objects
  - Watch for data being passed to other objects for calculations

Keep operations with data unless you have a good reason not to

Q9

# Cartoon of the Day


DON'T FORGET, YOU'RE GIVING ME A RIDE HOME TONIGHT.
OH RIGHT. AND IT'S YOUR TURN TO DO THE DISHES.
AWW.
YOU SHARE AN OFFICE AND LIVE TOGETHER?
THAT'S THE MOST PATHETIC THING I'VE EVER HEARD!
WHAT'S HIS PROBLEM?
I HOPE YOUR MOM MADE COOKIES!
Whirrrrrr
Not Invented Here™ © Bill Barnes & Paul Southworth
NotInventedHere.com

Used with permission. http://notinventedhe.re/on/2009-10-13

# Indirection

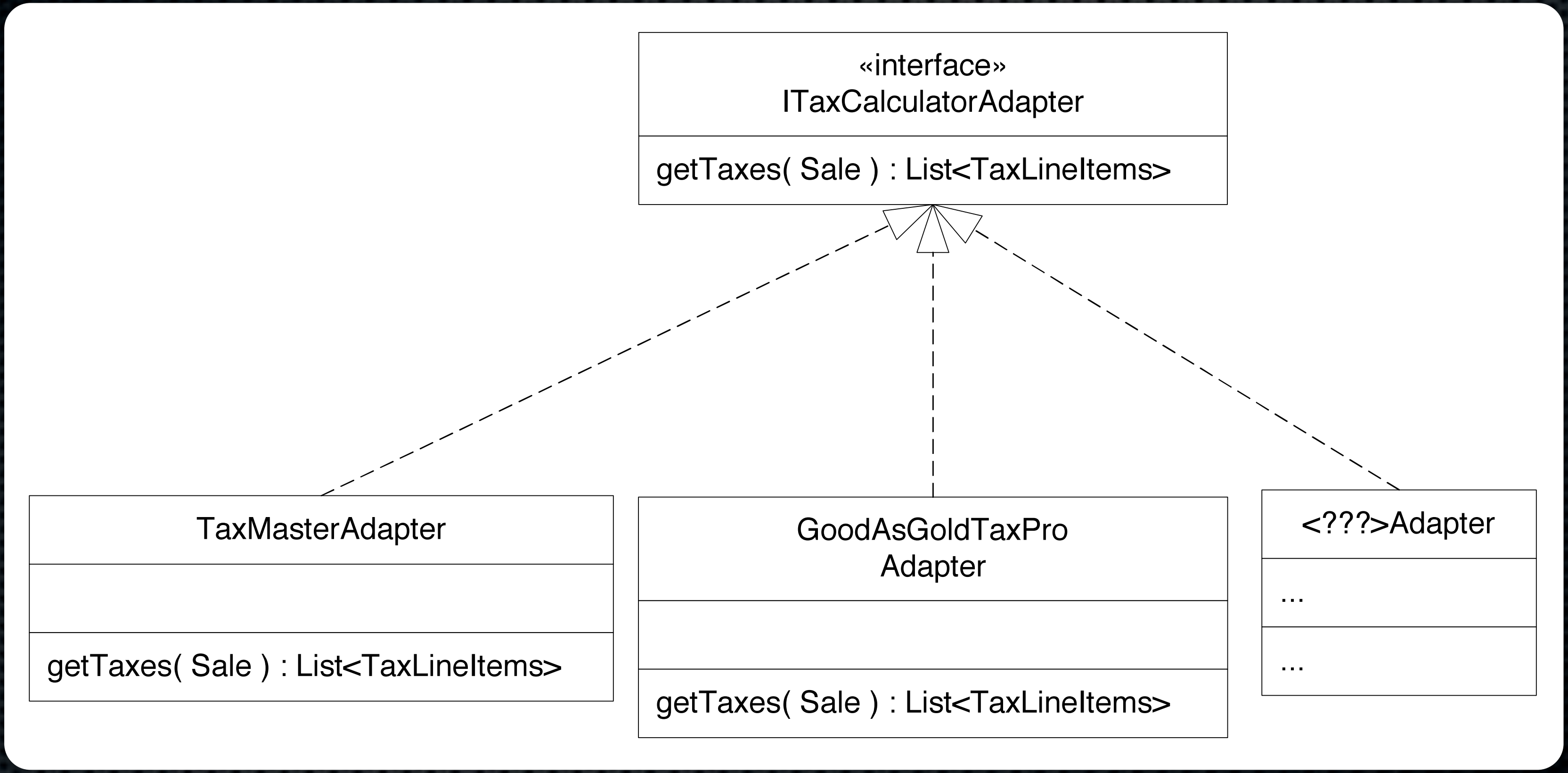
There is no problem in computer science that cannot be solved by an extra level of indirection.
— David Wheeler

# Indirection

- **Problem**: Where do we assign responsibility if we want to avoid direct coupling between two or more objects?
- **Solution**: Assign responsibility to an intermediate object to mediate between the other components

Q10,11

# Indirection and Polymorphism Example


«interface»
ITaxCalculatorAdapter
getTaxes( Sale ) : List<TaxLineItems>
TaxMasterAdapter
getTaxes( Sale ) : List<TaxLineItems>
GoodAsGoldTaxPro
Adapter
getTaxes( Sale ) : List<TaxLineItems>
<???>Adapter
...
...

# Protected Variation

Key Concept

- **Problem**: How do we design objects and systems so that instability in them does not have undesirable effects on other elements?
- **Solution**: Identify points of predicted instability (variation) and assign responsibilities to create a stable interface around them
- Example: *ITaxCalculatorAdaptor*

Instability here doesn’t mean “crashy”. It means prone to change or evolve.

Q12,13

# Protected Variation is Pervasive in Computing

- Virtual machines and operating systems
- Data-driven designs (e.g., configuration files)
- Service lookup (URLs, DNS)
- Uniform access to methods/fields (Ada, Eiffel, C#, Objective-C, Ruby, …)
- Standard languages (SQL)
- Liskov Substitution Principle

# Law of Demeter, or "Don't Talk to Strangers"

Special case of PV

- Within a method, messages should only be sent to:
  - *this*
  - a parameter
  - field of *this*
  - element in collection of field of *this*
  - new objects

Better: Don't talk to strangers **who seem unstable**

This guideline warns against code like:
*sale.getPayment().getAccount().getAccountHolder()*

# Notes on
Protected Variations

- Benefits (if we guessed variation points correctly):
  - Extensions easy to addCan plug in new implementations
  - Lower coupling
  - Lower cost of change
- Risk: watch out for speculative future-proofing

Q14

# Protected Variations by Other Names

- *Information hiding* [Parnas72]
  - “We propose instead that one begins with a list of difficult design decisions which are likely to change. Each module is then designed to hide such a decision from the others.”
- *Open-Closed Principle* [Meyer88]
  - “Modules should be both open (for extension …) and closed (… to modification[s] that affect clients)”