Agile Refactoring for Making Systems More Adaptable

Joseph W. Yoder

The Refactory, Inc.

October 11, 2009

joe@refactory.com

http://www.refactory.com

The Refactory Principals

John Brant
Brian Foote
Don Roberts
Joe Yoder
Ralph Johnson

Refactory Affiliates

Joseph Bergin
Fred Grossman
Bill Opdyke

Short Instructor Bio

Joseph Yoder began working with software in the mid 1980's, and has developed robust systems for many companies and organizations with global impact. He is a founder and principle of The Refactory, Inc., a company focused on software architecture, design, implementation, consulting and mentoring of all facets of software development. Joe is an international speaker and pattern author on software architecture, design, and implementation. Joseph specializes in Object-Oriented Analysis and Design, C#, Java, Smalltalk, Patterns, Agile Methods, Adaptable Systems, Refactoring, Reuse, and Frameworks. He has mentored developers on many types of software applications.

Joseph Yoder is a longstanding member and vice-chair of the board of The Hillside Group, a group dedicated to improving the quality of software development. Joseph has chaired the Pattern Languages of Programming Conferences (PLoP), as well as given many tutorials and talks at conferences such as JAOO, QCon, OOPSLA and ECOOP. He is co-author of the Big Ball of Mud pattern, which illuminated many fallacies in the approach to software architecture.

Joseph Yoder currently resides in Urbana, Illinois. He teaches Agile Methods such as XP, Design Patterns, Object Design, Refactoring, and Testing in industrial settings and mentors developers on these concepts. Joseph currently oversees a team of developers who have constructed an order fulfillment system based on enterprise architecture using the .NET environment. His other recent work includes working in both the Java and .NET environments, and deploying Domain-Specific Languages for clients. Joe thinks software is still too hard to change. He wants to do something about this and believes that putting the ability to change software in the hands of the people with the knowledge to change it is a promising avenue to solve this problem.

Agile Principles

- Scrum, TDD, Refactoring, Regular Feedback, Testing, More Eyes, …
- Good People! Face-To-Face conversation.
- Continuous attention to technical excellence!
- Motivated individuals with the environment and support they need. Retrospectives!
- Allow Requirements to Change! Encourage Software Evolution as needed!

Agile Encourages Changes

- Very Little Upfront Design!
- Piecemeal Growth! Small Iterations!
- Late changes to the requirements of the system!
- Continuously Evolving the Architecture!

Adapting to Changes requires the code to change and Refactoring supports changes to the code in a "safe" way.

Agile & Refactoring

- A key part of Agile is to allow things to change and adapt as dictated by business needs!
- To support these changes, Refactoring is encouraged by most Agile practitioners.
- Agile has helped Refactoring be accepted into the mainstream development process (even sometimes encouraged).

Ground Breaking Work

Before there was Agile…

Started at the University Of Illinois

Bill Opdyke & Don Roberts PHD Thesis

Refactoring Browser by John Brant and Don Roberts (first commercial tool)

Refactoring Definition

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure…Martin Fowler, **Refactoring** *Improving the Design of Existing Code; Addison Wesley , 1999*

What is Refactoring

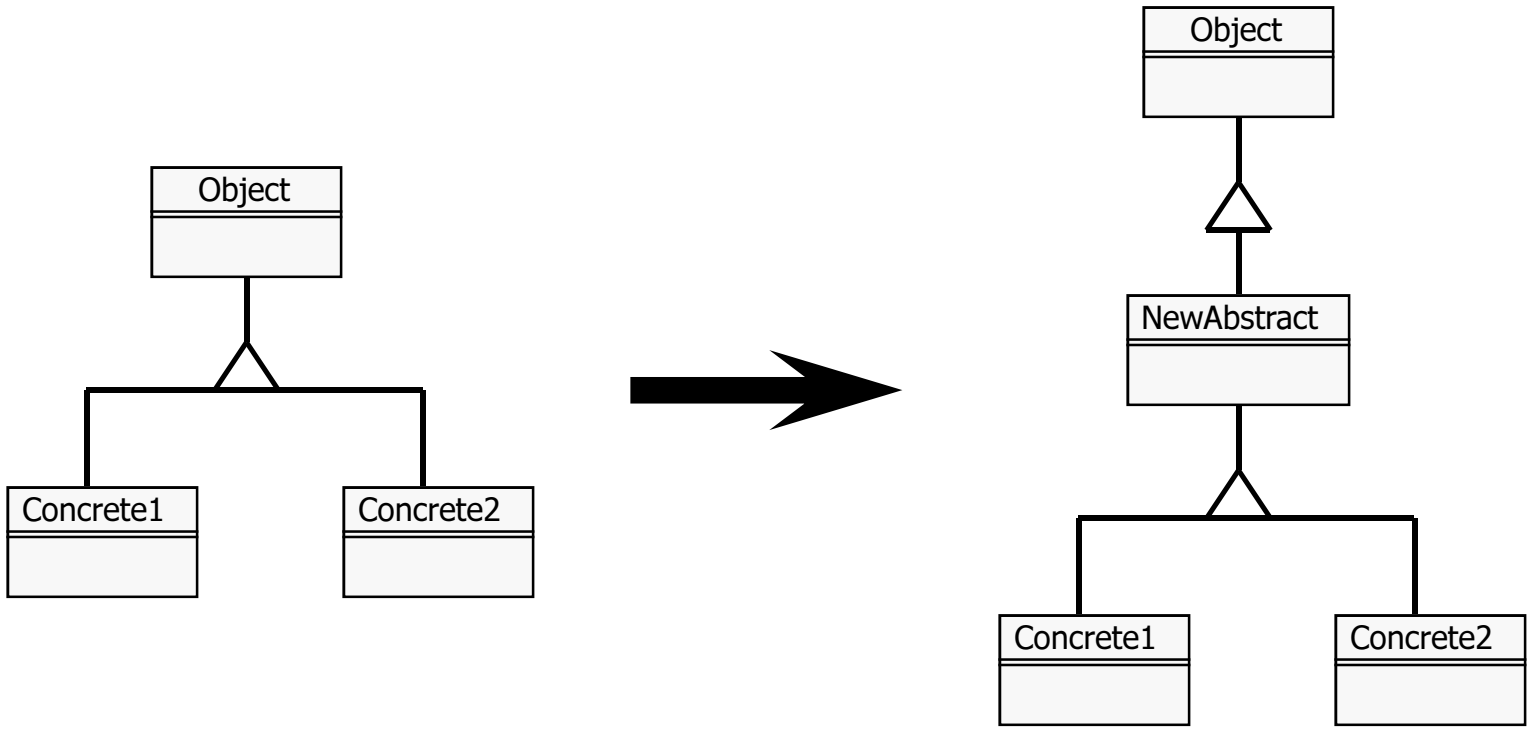
Refactoring is a controlled technique for improving the design of an existing code base

Small behavior-preserving transformations,

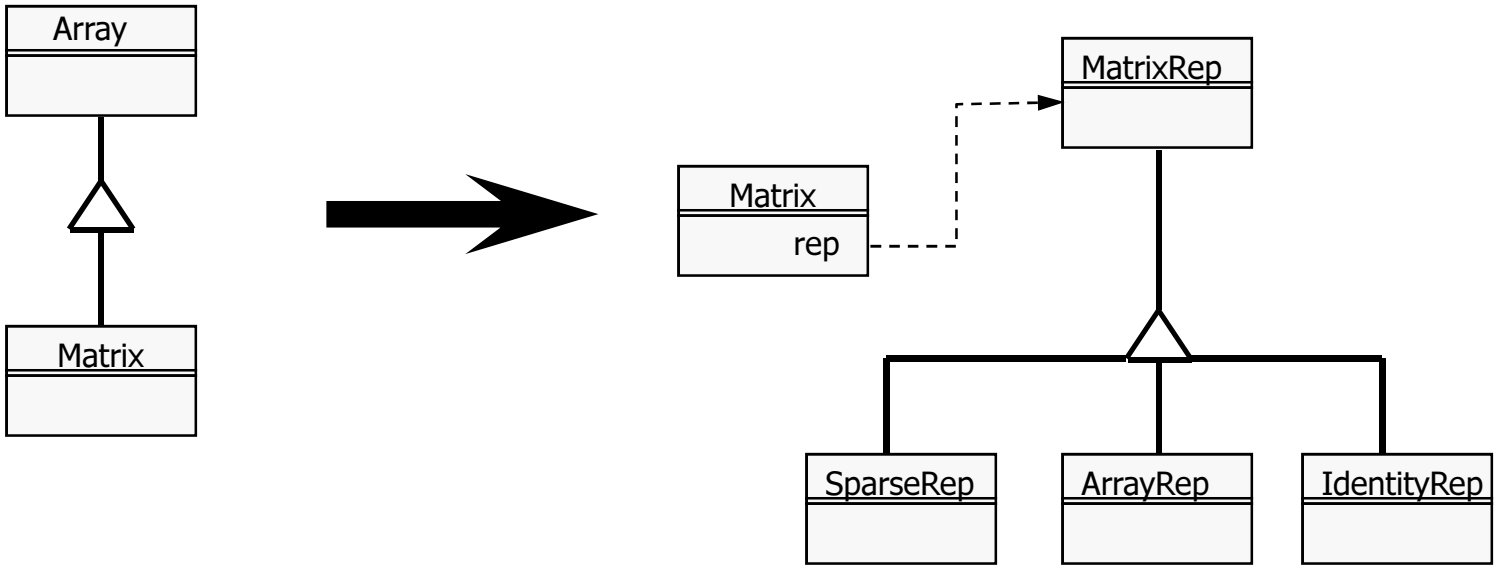
- Each of might be "too small to be worth doing"
- Cumulative effect of transformations is significant
- Small steps help reduce the risk of introducing errors
- Avoid having the system broken while you are carrying out the restructuring
- Gradually refactor a system over a period of time

A Simple Refactoring

Create Empty Class



A Complex Refactoring



Adapted from Don Roberts, The Refactory, Inc.

Refactoring Code Smells

A *code smell* is a hint that something has gone wrong somewhere in your code. Use the smell to track down the problem. KentBeck (with inspiration from the nose of Massimo Arnoldi) seems to have coined the phrase in the "OnceAndOnlyOnce" page, where he also said that code "wants to be simple". *Bad Smells in Code* was an essay by KentBeck and MartinFowler, published as Chapter 3 of Refactoring Improving The Design Of ExistingCode.

----Wards Wiki

Code Smells Review

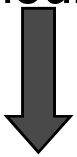
- Duplicate Code, Long Method, Large Class,
- Long Parameter List, Divergent Change,
- Shotgun Surgery, Feature Envy, Data Clumps
- Primitive Obsession, Switch Statements
- Parallel Inheritance, Lazy Class
- Speculative Generality, Temporary Field
- Message Chains, Middle Man
- Inappropriate Intimacy, Incomplete Lib Classes
- Data Class, Refused Bequest, Comments

Comments

- We are not against comments but…
- If you see large methods that have places where the code is commented, use *Extract Method* to pull that out to a comment

Comments Example

```
void printOwing (double amount) {
    printBanner();
    //print details
    System.out.println (name: " + _name);
    System.out.println (amount: " + amount);
    …}
```



```
void printOwing (double amount) {
    printBanner();
    printDetails();
     …}
void printDetails (double amount) {
    System.out.println (name: " + _name);
    System.out.println (amount: " + amount);}
```

Agile & Testing

- Many Agile practices highly encourage Testing as one of their core practices.
- Processes like XP support developers writing many unit tests (XUnit).
- Test Driven Development (TDD) is usually considered a key principle of Agile.
- Testing was a key principle of Refactoring before there was Agile.

Prerequisites of Refactoring

- Since you are changing the code base, it is IMPORTANT to validate with tests.
- There are also a time to refactor and a time to wait.
- Need both Unit Testing and Integrated Tests for making sure nothing breaks

Deciding Whether A Refactoring is Safe

- "A refactoring shouldn't break a program."
  - What does this mean?
- A *safe* refactoring is *behavior preserving*.
- It is important not to violate:
  - naming/ scoping rules.
  - type rules.
- "The program should perform the same after a refactoring as before."
- Satisfying timing constraints.

Using Standard Tools

- To be safe, must have tests
- Should pass the tests before and after refactoring
  - Commercial Testing Tools
  - Kent Beck's Testing Framework (SUnit, JUnit, Nunit, …)
- Take small steps, testing between each
- Java and C# tools are getting better

Catalogue of Refactorings

- Composing Method
- Moving Features
- Organize Data
- Simplifying Conditionals
- Simpler Method Calls
- Generalization

From Fowlers Book

Composing Method Catalogue

- Extract Method, Inline Method,
- Inline Temp, Replace Temp with Query,
- Introduce Explaining Variable,
- Split Temporary Variable,
- Remove Assignments to Parameters,
- Replace Method with Method Object,
- Substitute Algorithm

Extract Method

```
void printOwing(double amount) {
   printBanner();

   //print details
   System.out.println("name:" + _name);
   System.out.println("amount:" + _amount);

}
```

Extract Method (2)

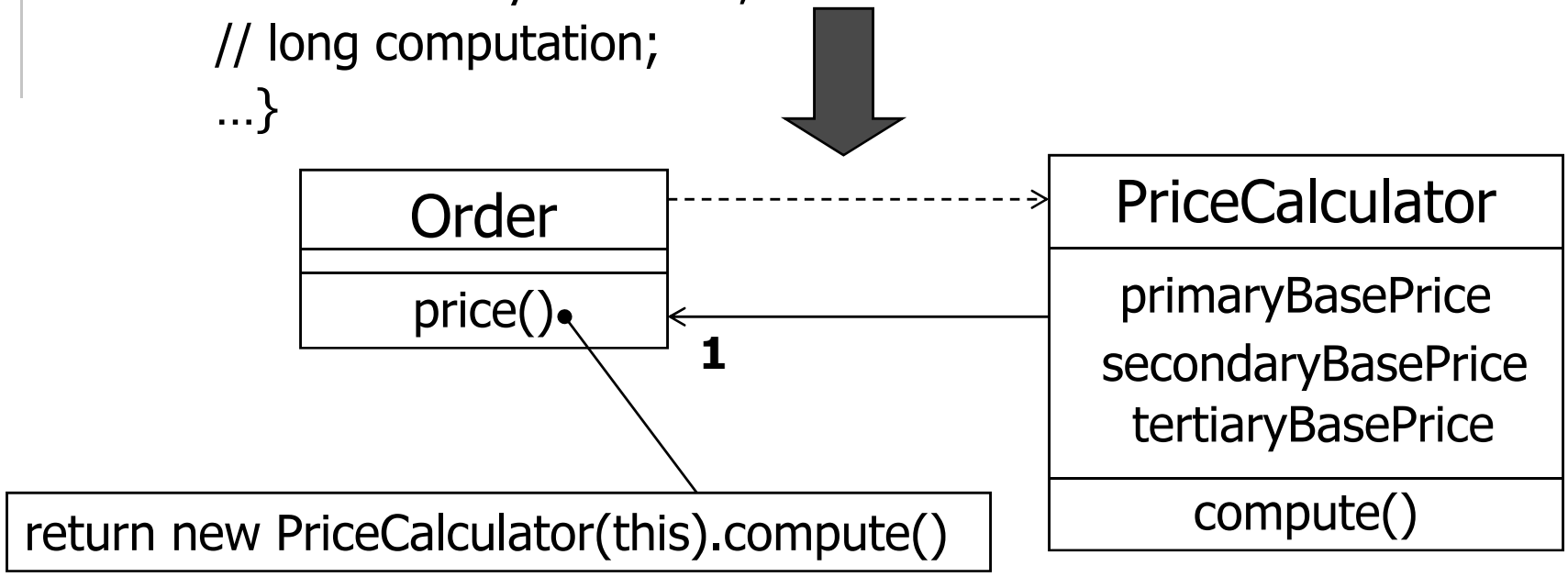
```
void printOwing (double amount) {
   printBanner();
   printDetails(amount);
}

void printDetails (double amount) {
   System.out.println("name:" + _name);
   System.out.println("amount:" + _amount);
}
```

Extract Method Mechanics

- Create a new method and name it after the intention of the code to extract.
- Copy the extracted code from the source to the new method.
- Scan the extracted code for references to any variables that are local to the source method.
- See whether any temps are used only within extracted code.
- Pass into target method as parameters local-scope variables that are read from the extracted code.
- Replace the extracted code in the source method.

Remove Method with Method Object

```
Class Order ...
    double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // long computation;
        ...}
```



Remove Method to Method Object Mechanics

- Create a new class named after method.
- Give the new class a final field for the object that hosted the original method.
- Give the new class a constructor for the original object and each parameter.
- Give the new class a compute method.
- Copy the body of original method to compute.
- Compile.
- Replace the old method with the one that creates the new object and calls compute.

Simplifying Conditionals Catalogue

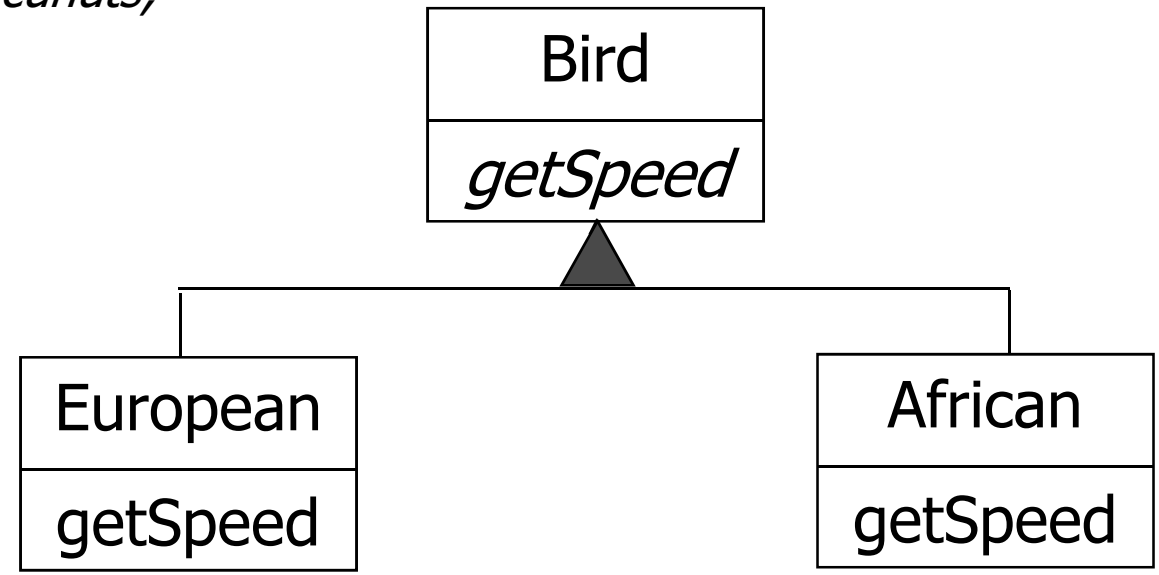
- Decompose Conditional, Consolidate Conditional Expression, Consolidate Duplicate Conditional Fragments, Remove Control Flag, Replace Nested Conditionals with (Guard Clauses | Polymorphism), Introduce Null Object, Introduce Assertion

Replace Conditional with Polymorphism

- You have a conditional that chooses different behavior depending on the type of an object
- *Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract!*

Replace Conditional with Polymorphism

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() – getLoadFactor() * _number
    ofCocunuts;
    …}
```



Replace Polymorphism Mechanics

- If the conditional is part of a larger statement, take apart the conditional and use Extract Method.
- If necessary, use Move Method to place the conditional at the top of the inheritance hierarchy.
- Create classes and copy the body of the leg of the conditional into the subclass.
- Compile and Test…and continue on.

Moving Code "Refactoring"

To move a function to a different class, add an argument to refer to the original class of which it was a member and change all references to member variables to use the new argument.

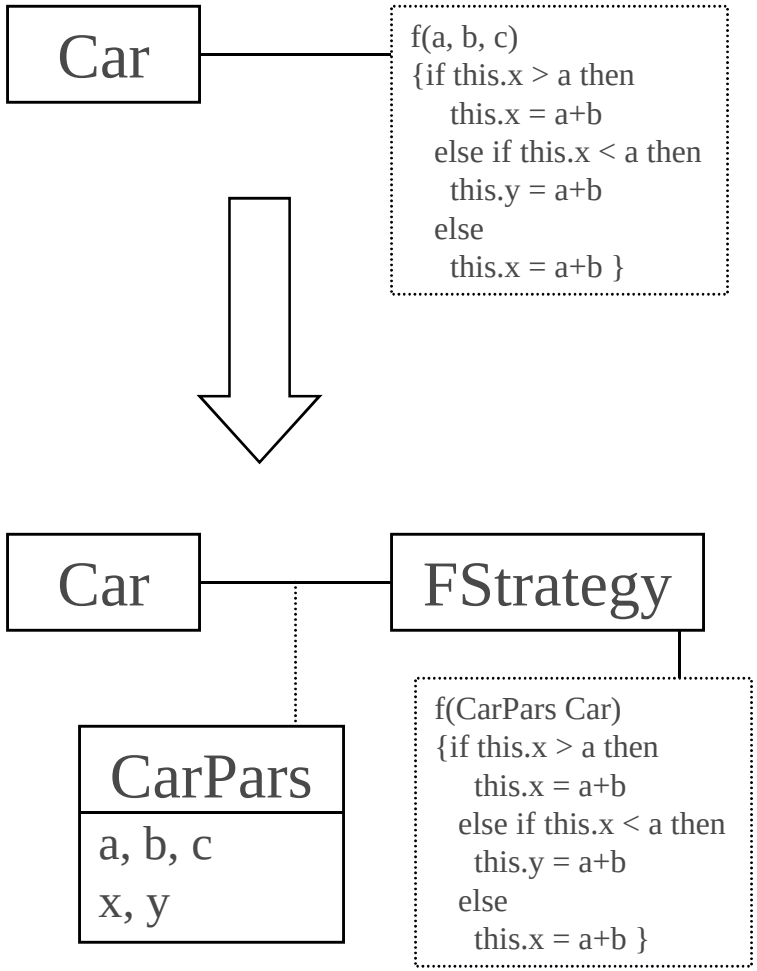
If you are moving it to the class of one of the arguments, you can make the argument be the receiver.

Moving function f from class X to class B

```
class X {
    int f(A anA, B aB){
          return (anA.size + size) / aB.size;
    } ...
class B {
    int f(A anA, X anX){
          return (anA.size + anX.size) / size;
    } ...
```

Moving Code "Refactoring"

- You can also pass in a parameter object which gives the algorithm all of the values that it will need.
- Inner Classes can help by providing access to values that the algorithm may need.



Refactoring and Design Patterns (1)

What varies	Design Pattern
Algorithms	Strategy, Visitor
Actions	Command
Implementations	Bridge
Response to change	Observer

Refactoring and Design Patterns (2)

What varies	Design Pattern
Interactions between objects	Mediator
Object being created	Factory Method, Abstract Factory, Prototype
Structure being created	Builder
Traversal Algorithm	Iterator
Object interfaces	Adapter
Object behavior	Decorator, State

Refactoring Summary

- We have seen the mechanics for some of the Refactorings.
- When you find smelly code, you often apply Refactorings to clean your code.
- Refactorings do often apply Design Patterns.

Allowing Systems to Adapt

- Requirements change within application's domain.
- Domain Experts know their domain best.
- Business Rules are changing rapidly.
- Applications have to quickly adapt to new business requirements.
- Agile Development Welcomes Changes to the Requirements, even late in the development!
- One way to Adapt is to just Refactor to new business requirements…though sometimes…

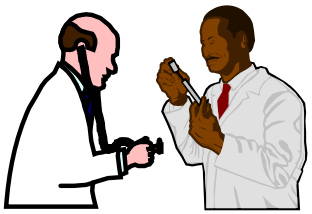
Forces – Shearing Layers

- Who (Business Person, Analyst, Developer)
- What (Business Rule, Persistence Layer,…)
- When (How often, How fast)

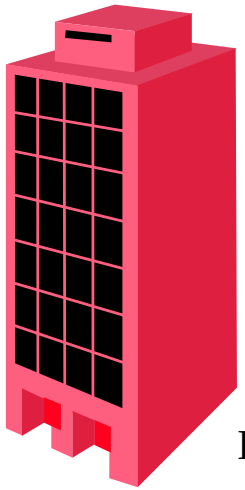
There is a different rate of change on the system.

Foote & Yoder - Ball of Mud PLoPD4

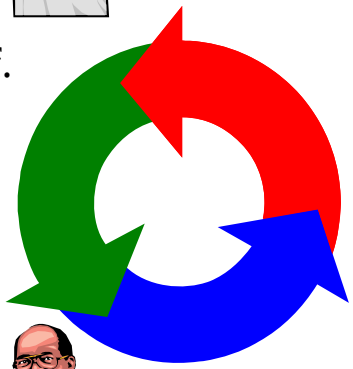
Newborn Screening Refactoring Example



Doctor, HealthProf.



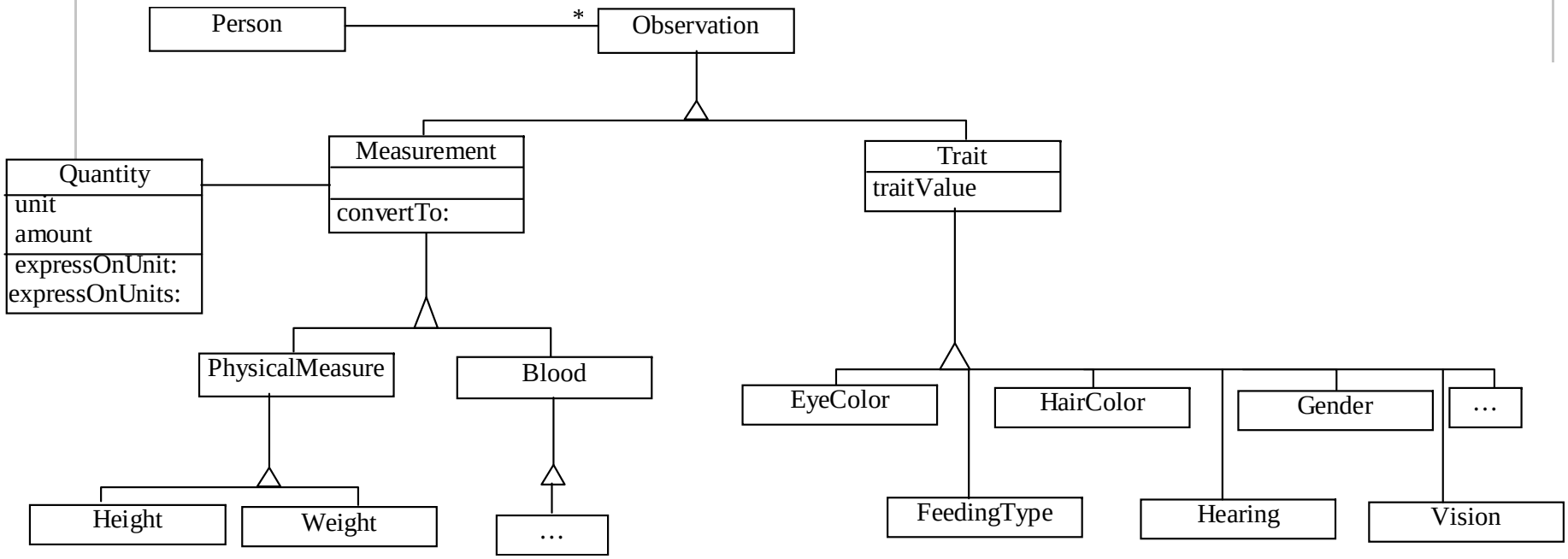
Hospital, Lab



Mother, Infant

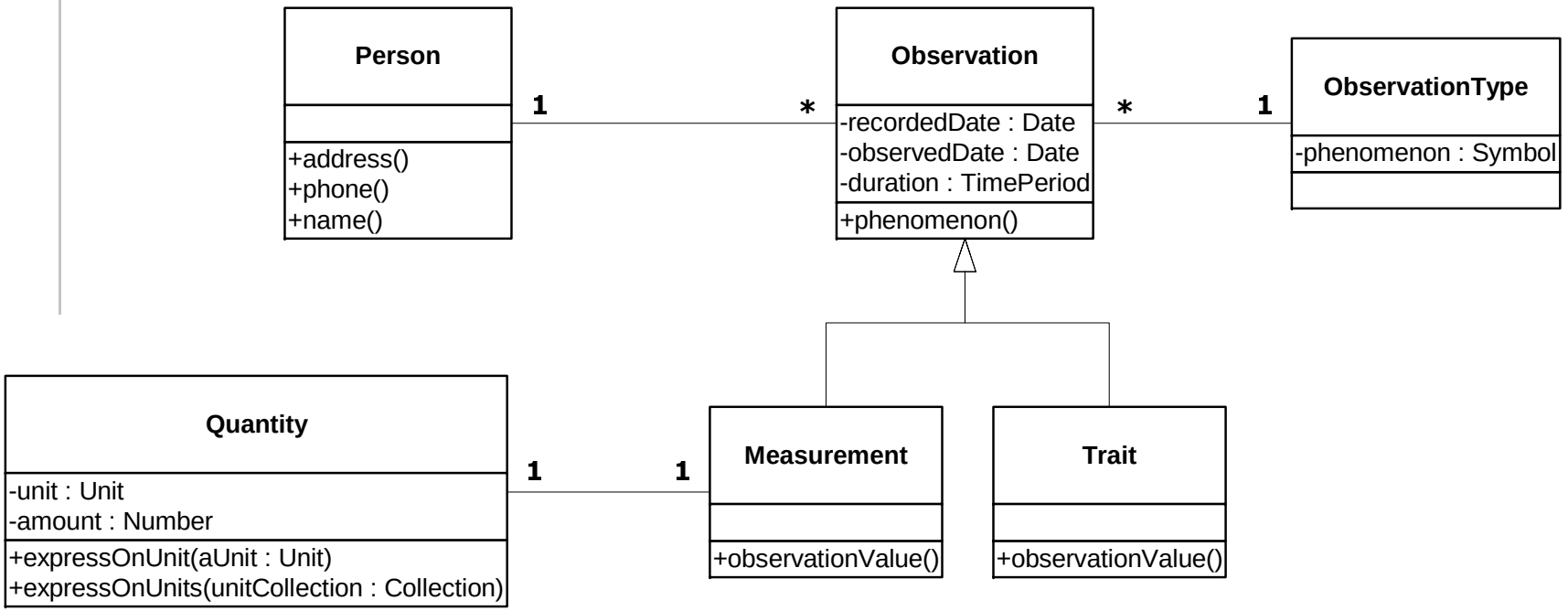
Blood Specimen

Medical Observation – First Model

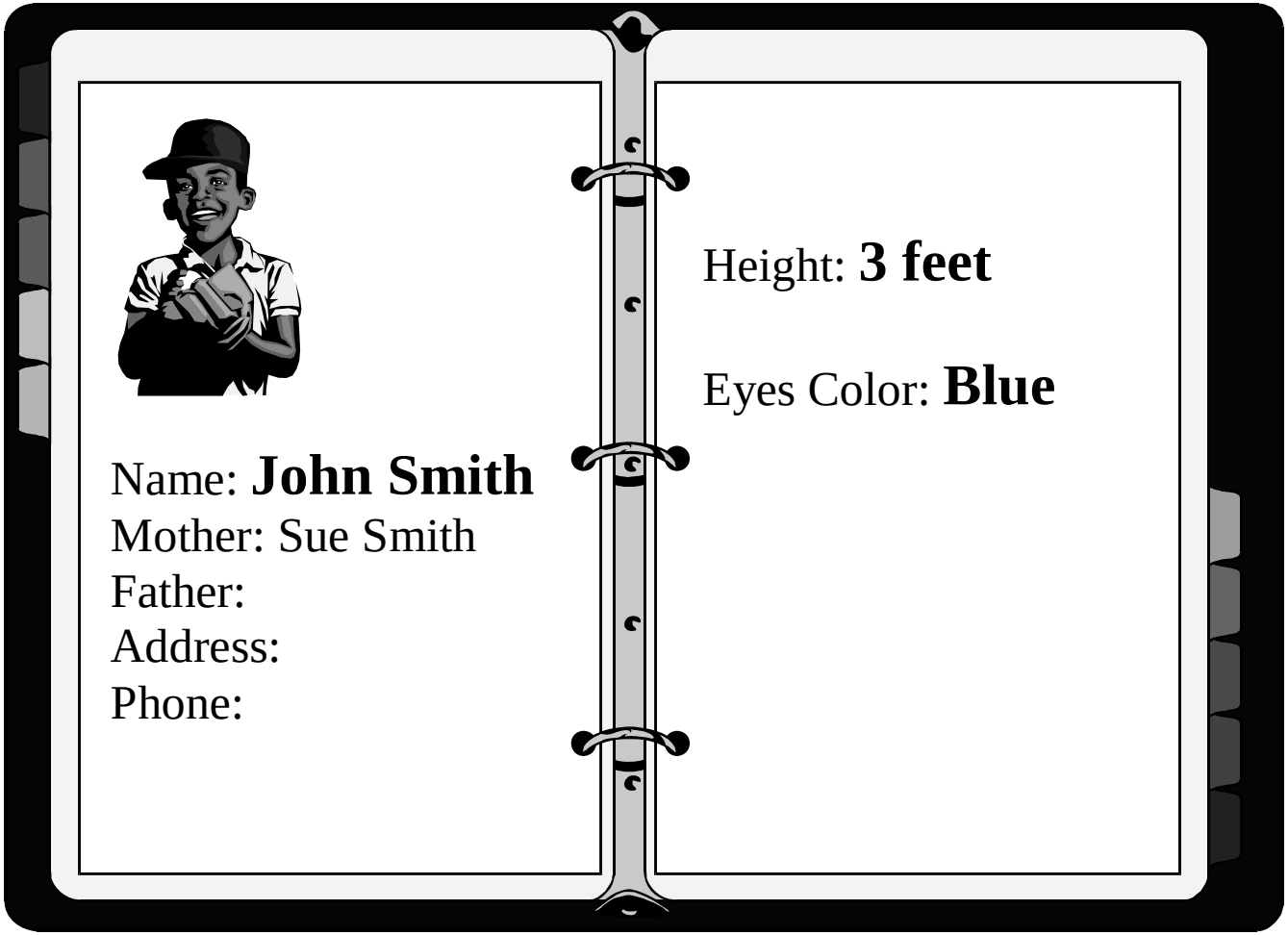


What happens when a new observation is required?

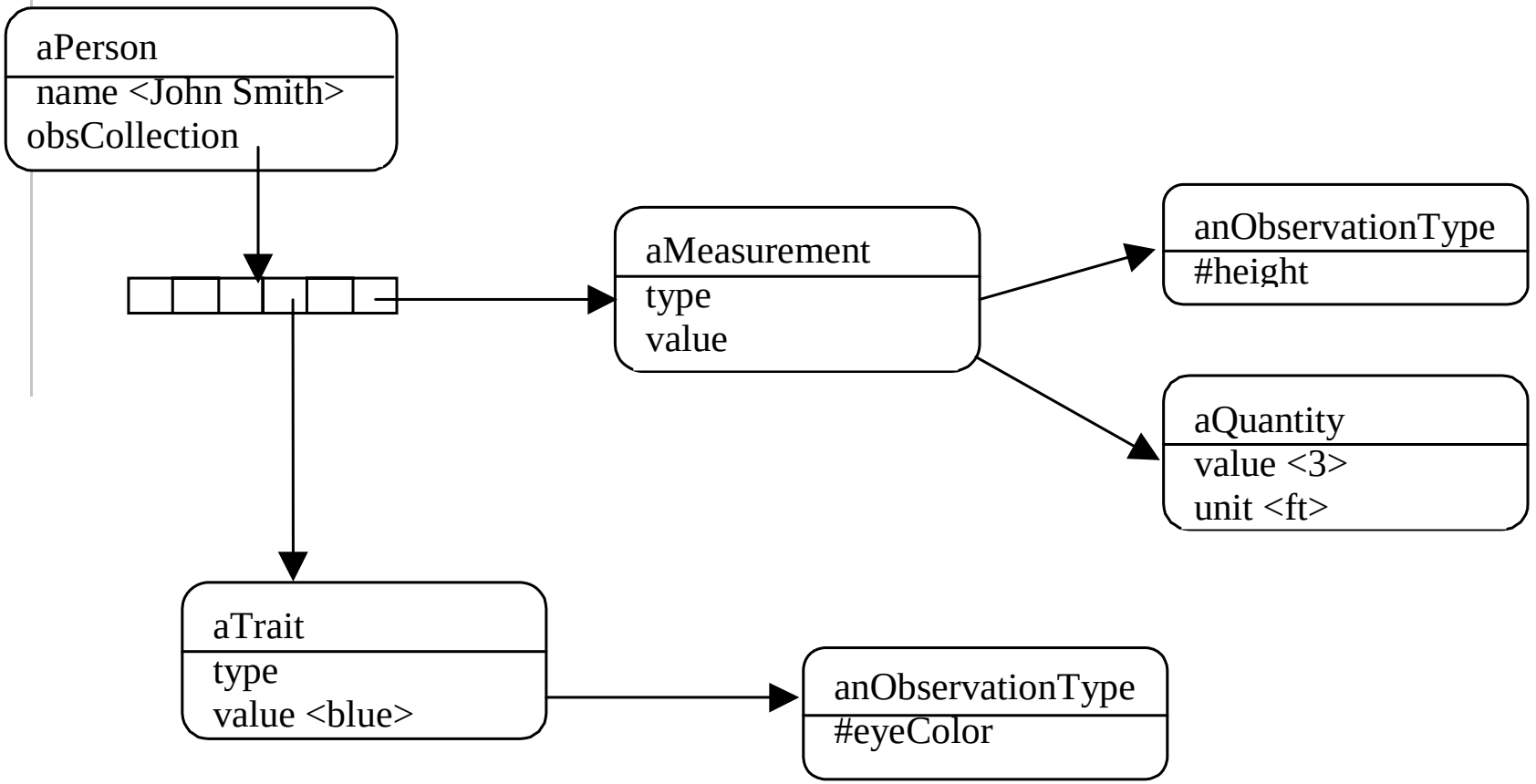
Observation Design (1st Refactoring)



Observation Design
Example



Observation Design
(instance diagram)

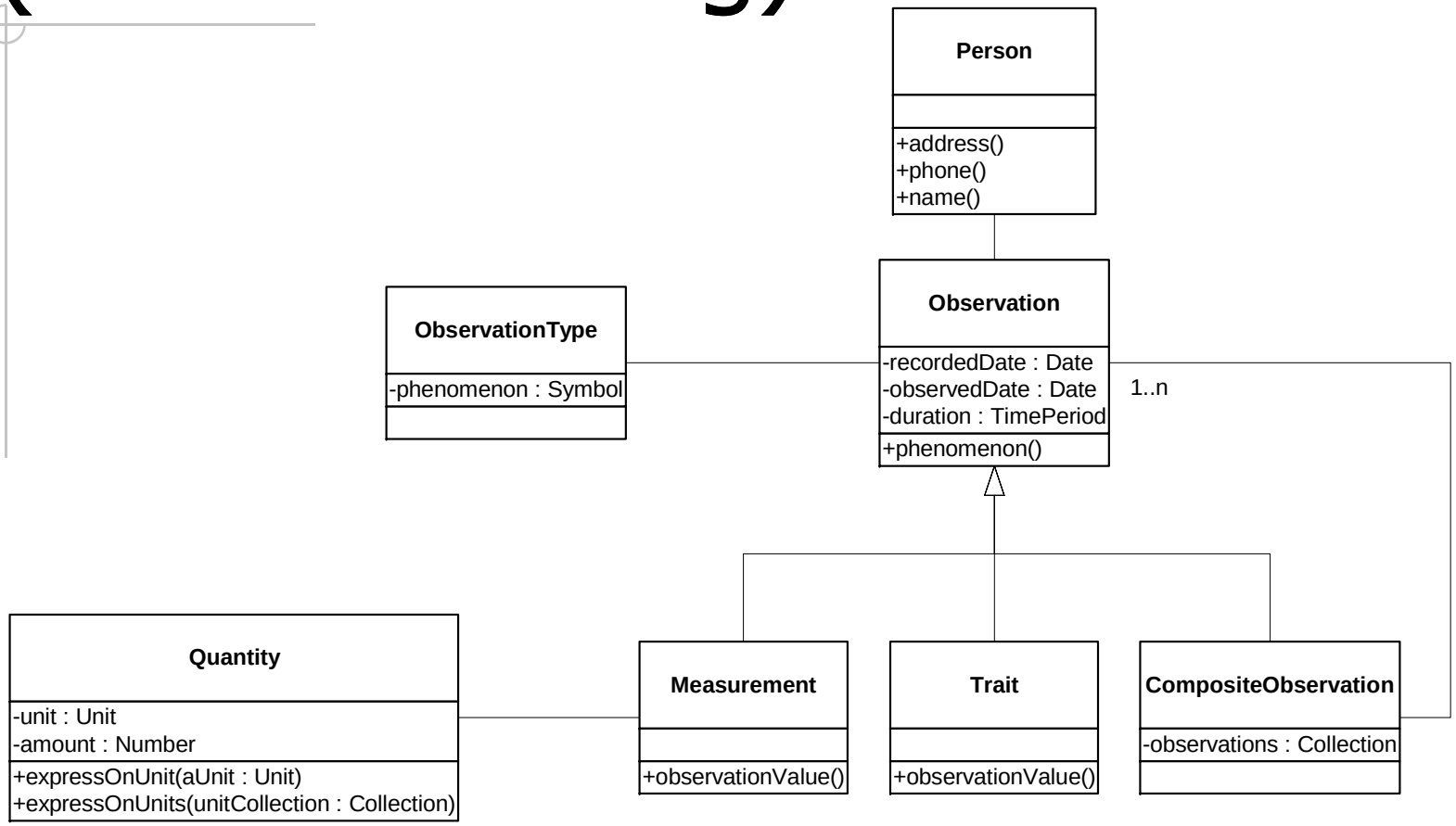


Composing Observations

Observations can be more complex

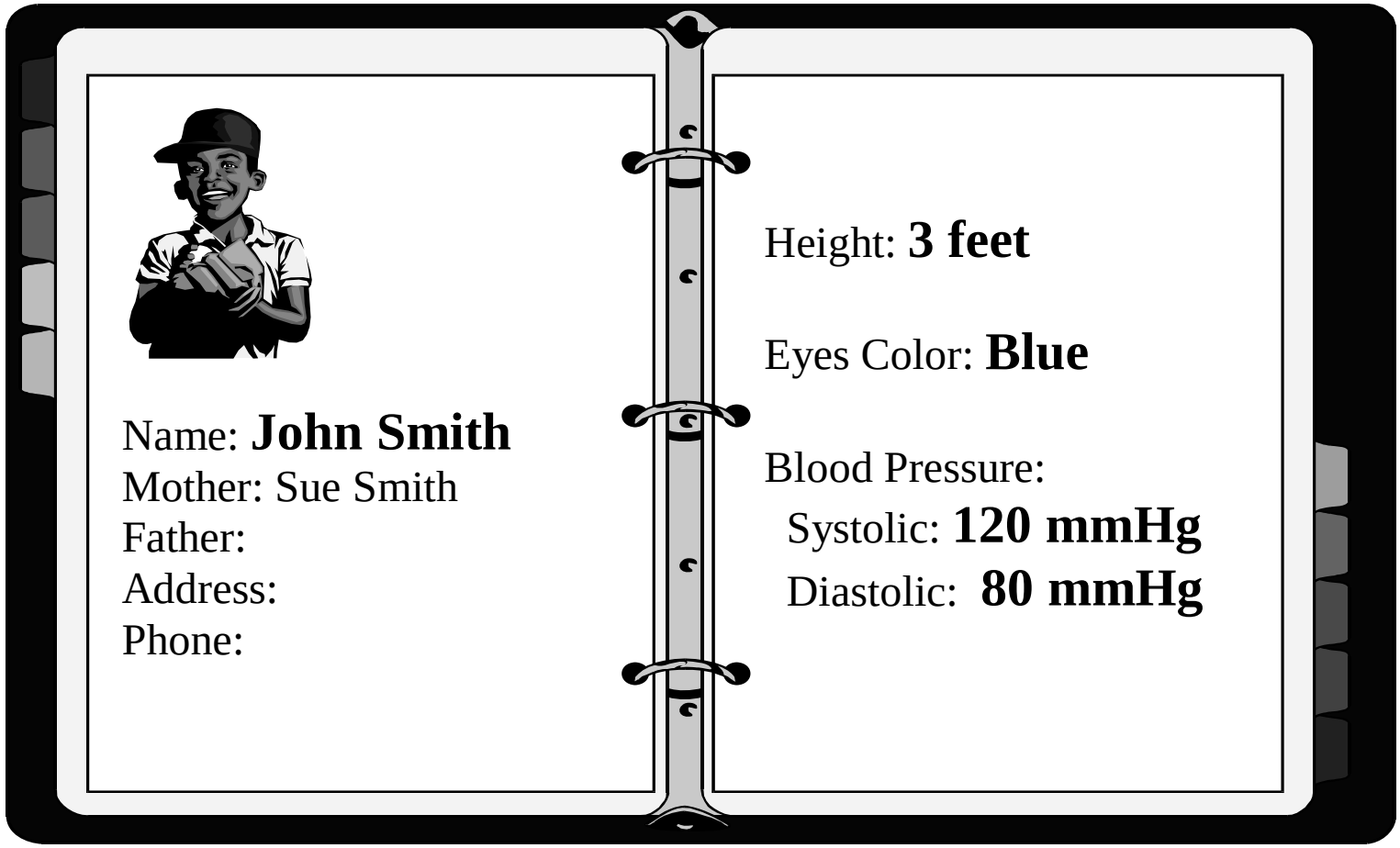
- Cholesterol
  - Components: HDL, LDL
- Blood Pressure
  - Components: Systolic, Diastolic
- Vision
  - Components: Left Eye, Right Eye

Composite Observation Design (2nd Refactoring)

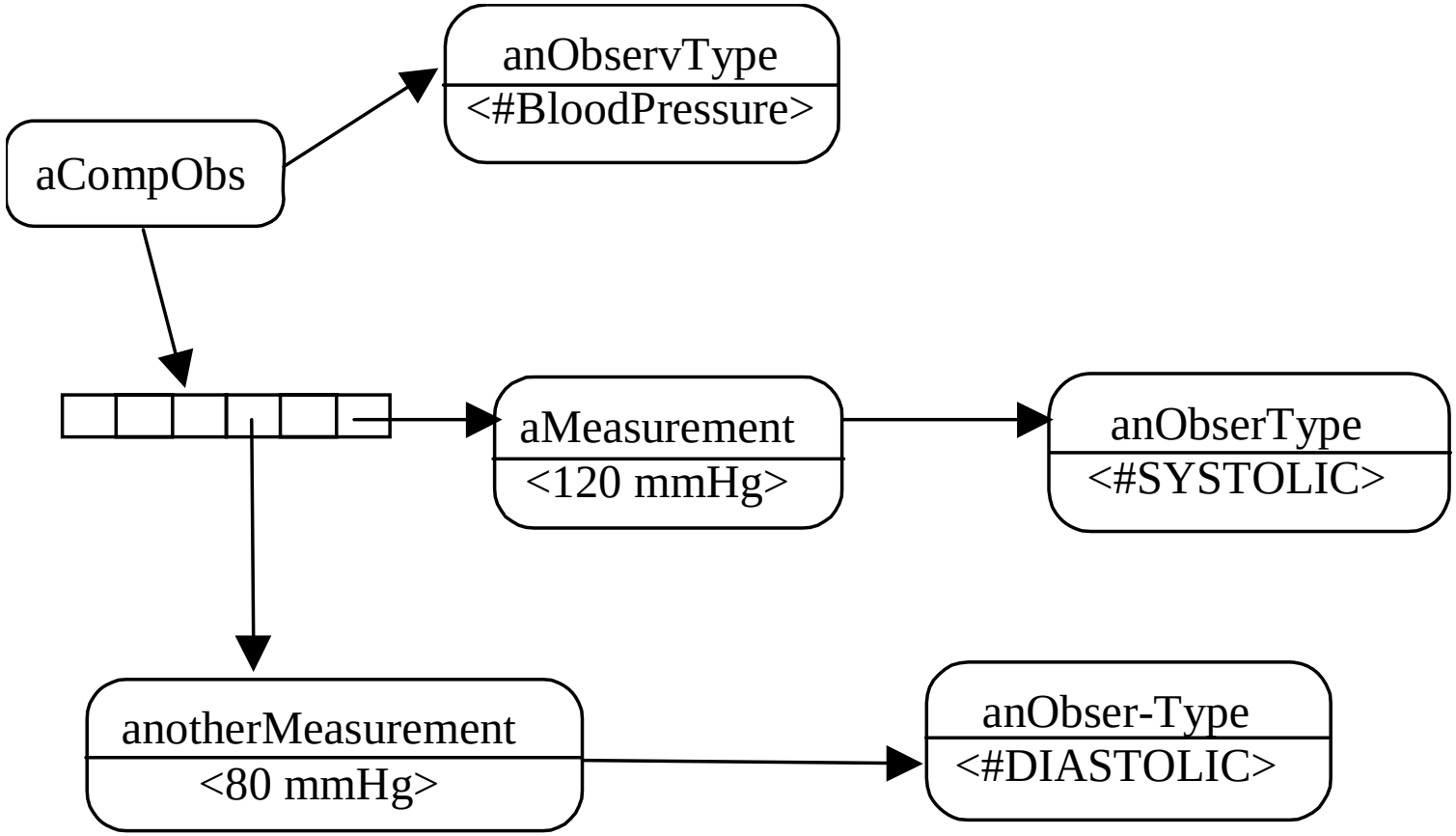


Composite Pattern (GOF)

Observation Design
Example

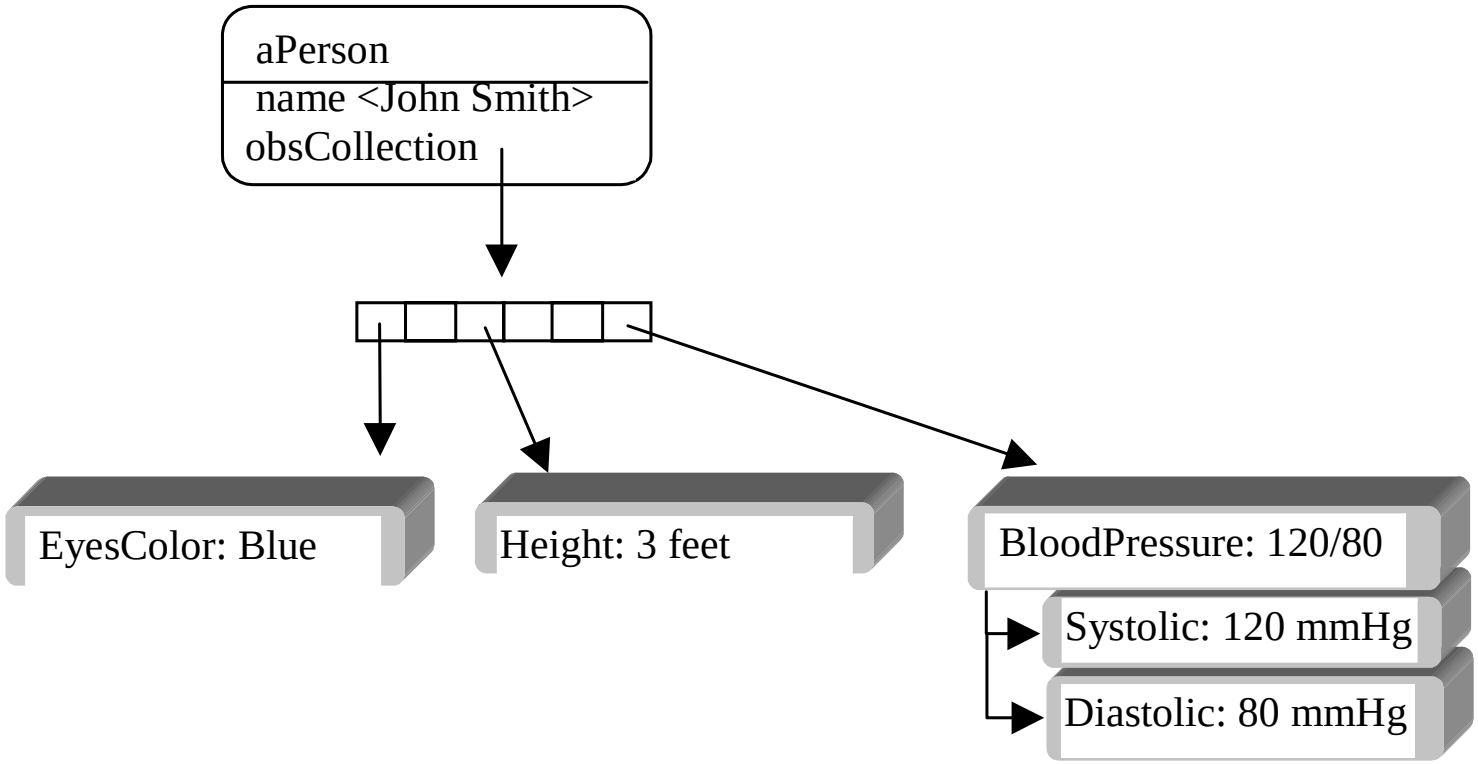


Composite Observation Design
(instance diagram)



Composite and Primitive Observation Design

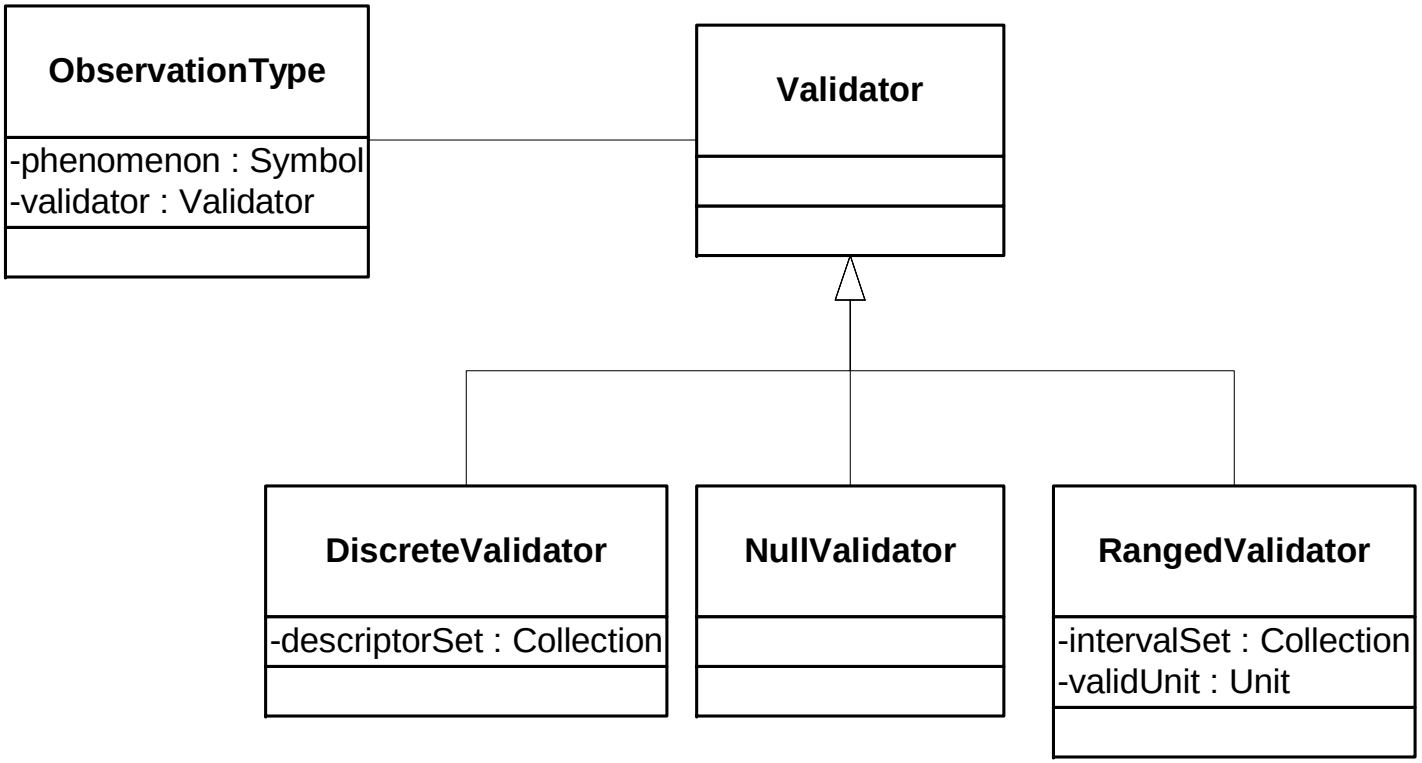
What we know about John?



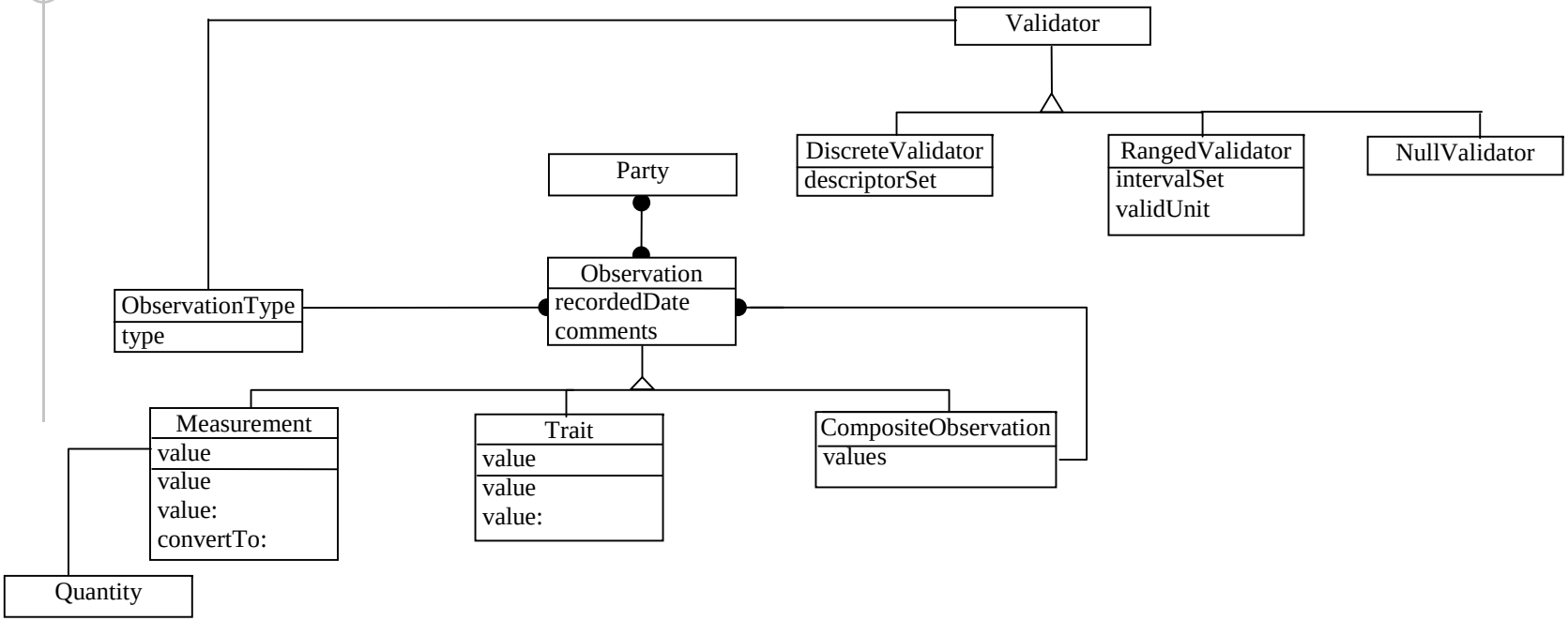
Validating Observations

- Each Observation has its own set of legal values.
  - Baby's Weight: [0 30] pounds
  - HepatitisB: {positive, negative}
  - Left/Right Vision: {normal, abnormal}
- GUI can enforce legal values
  - but we want business rules in domain objects

Validating Observations Design (3rd Refactoring)



Overall Observation Design

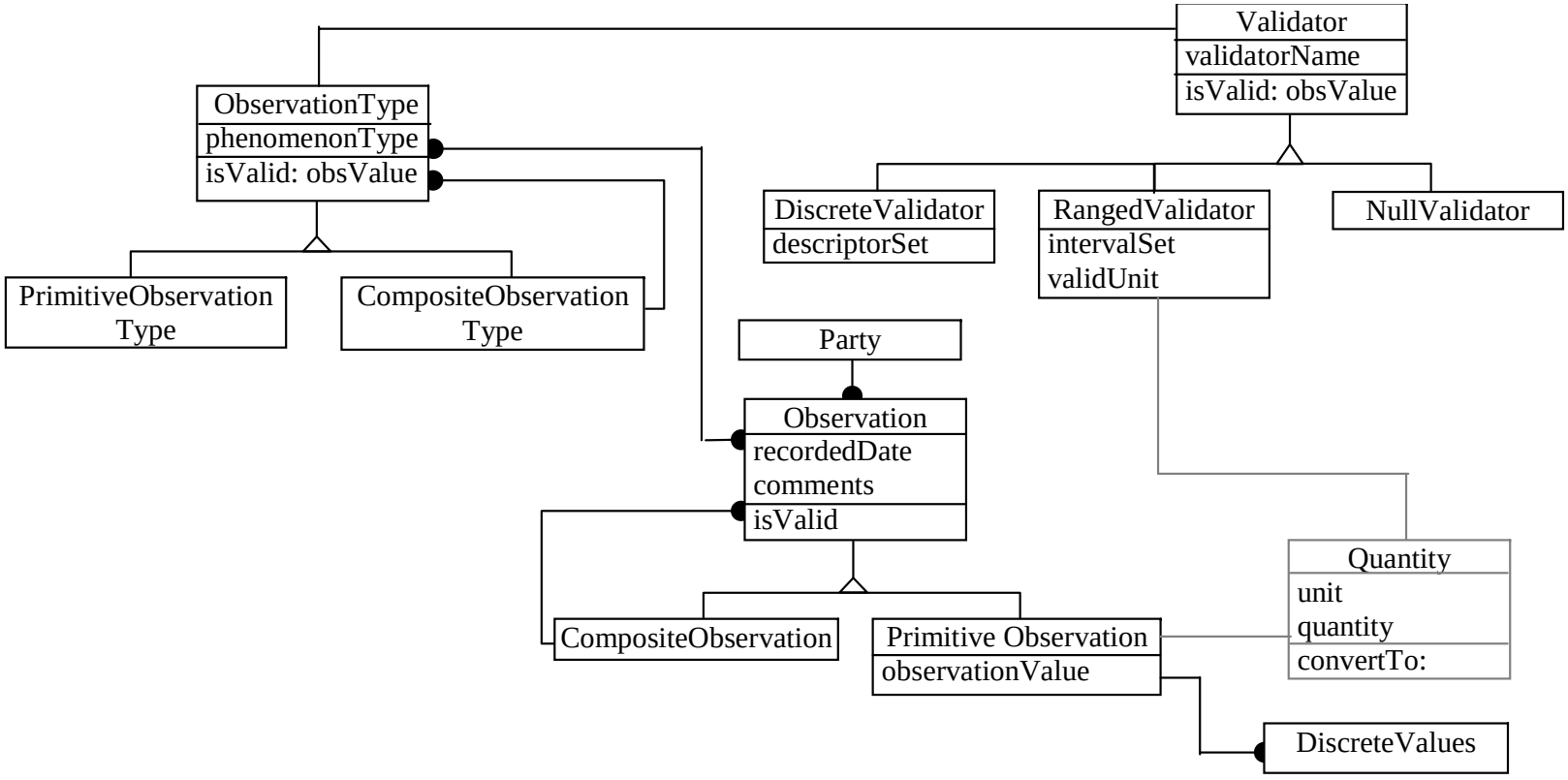


Is everything an Observation?

How does the model specify the structure of the Composite?

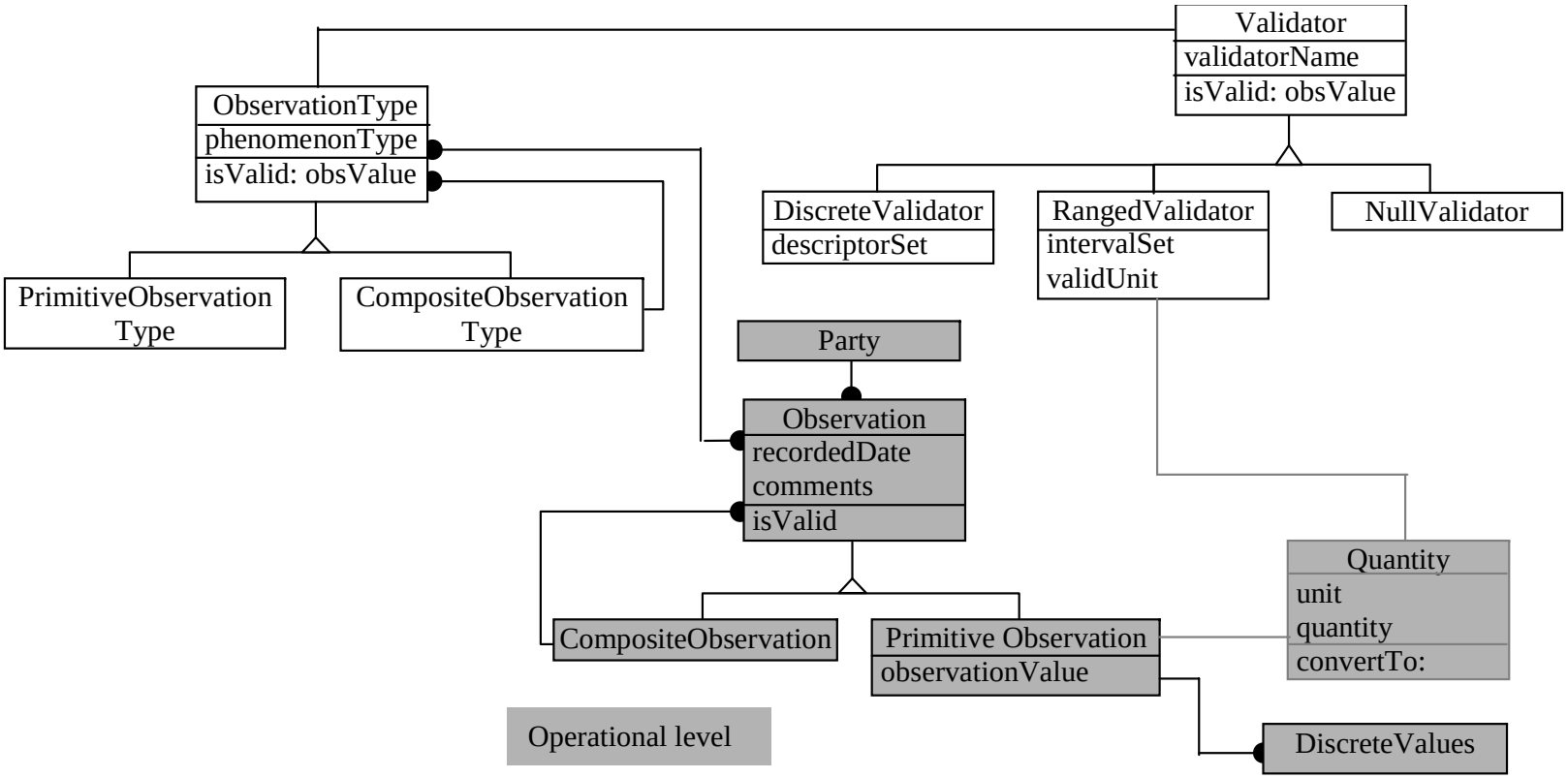
What is the relationship between Trait and DiscreteValidator?

Observation Design



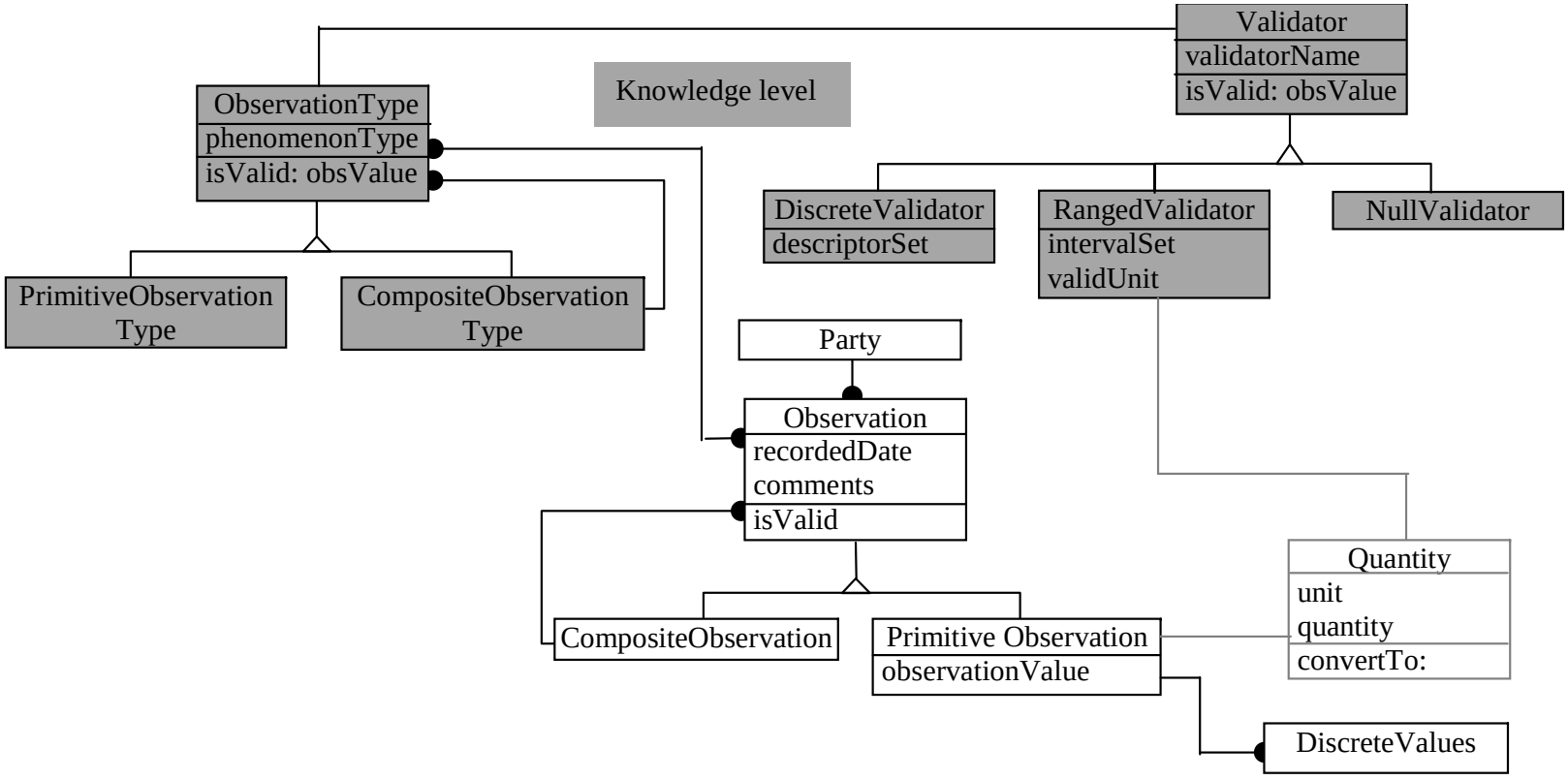
Current Design

Observation Design



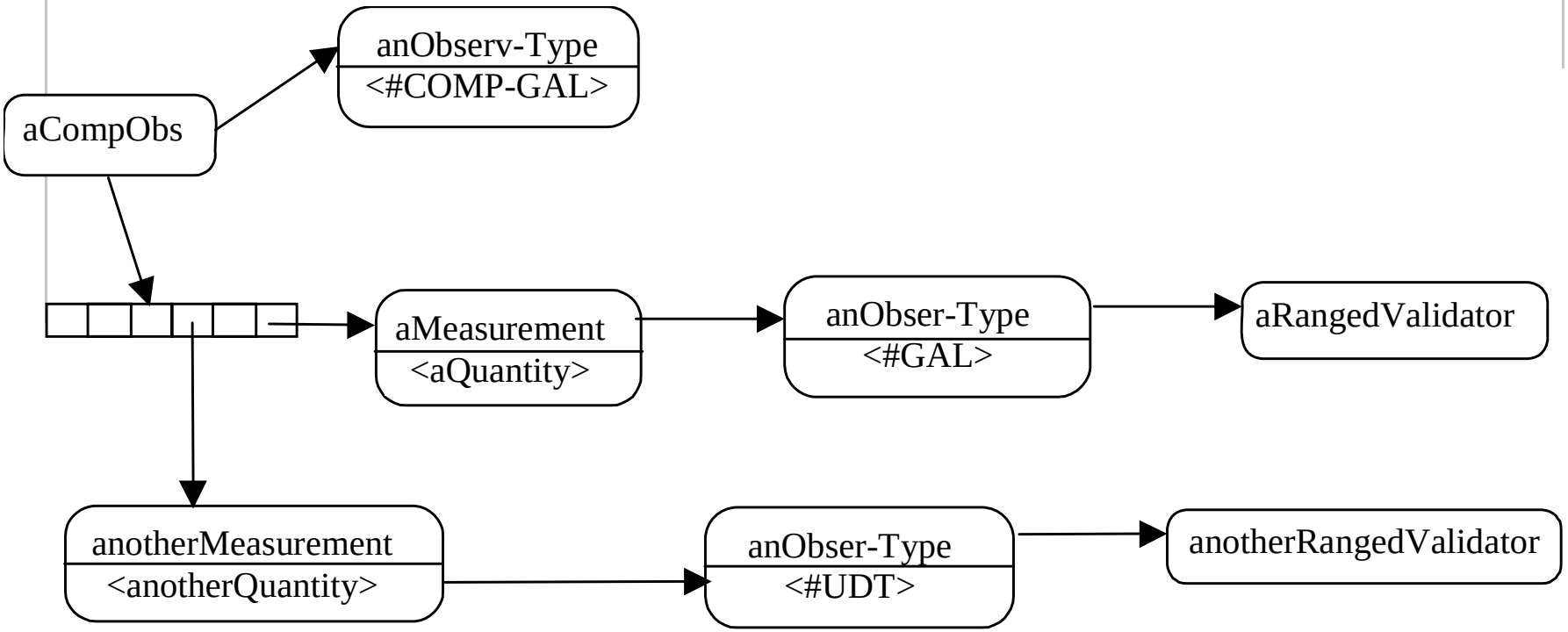
Current Design

Observation Design

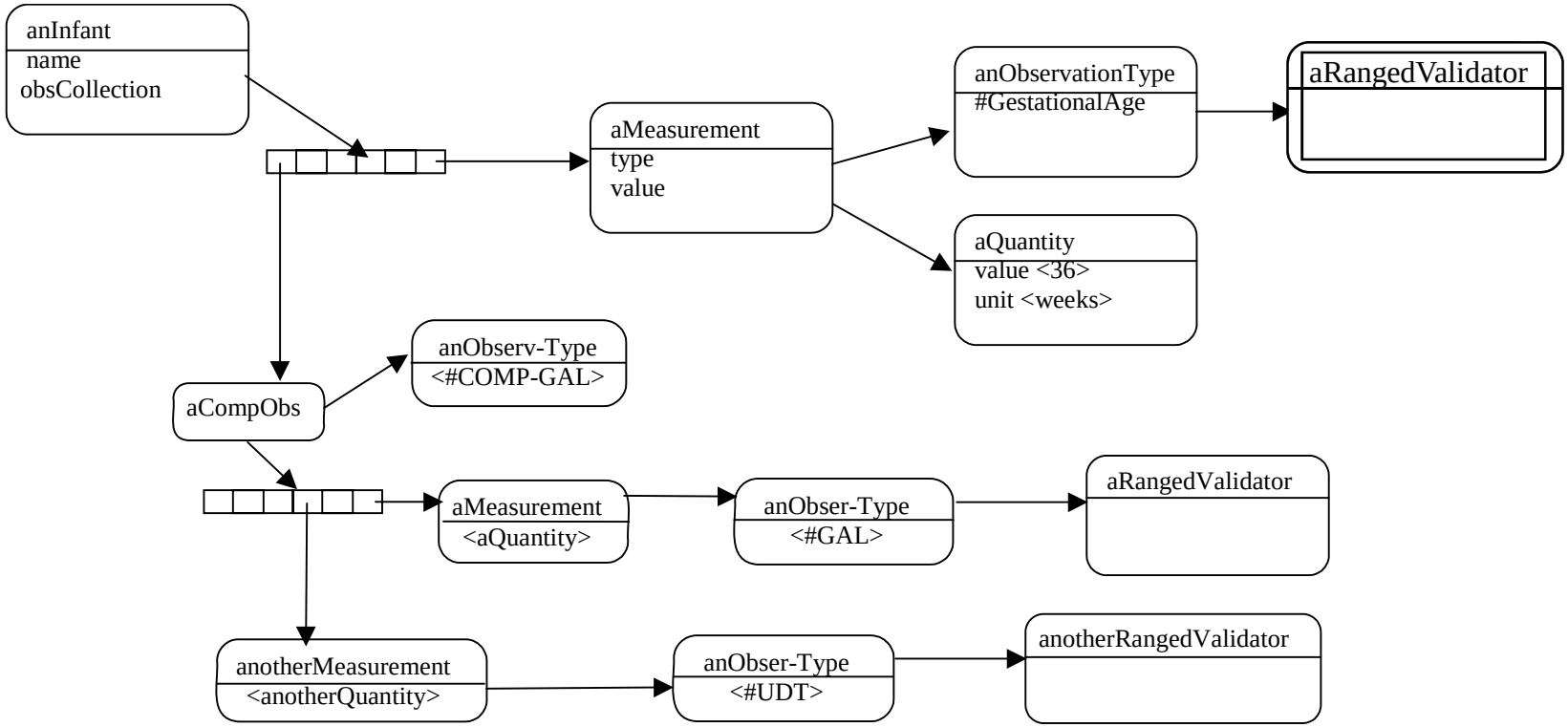


Current Design

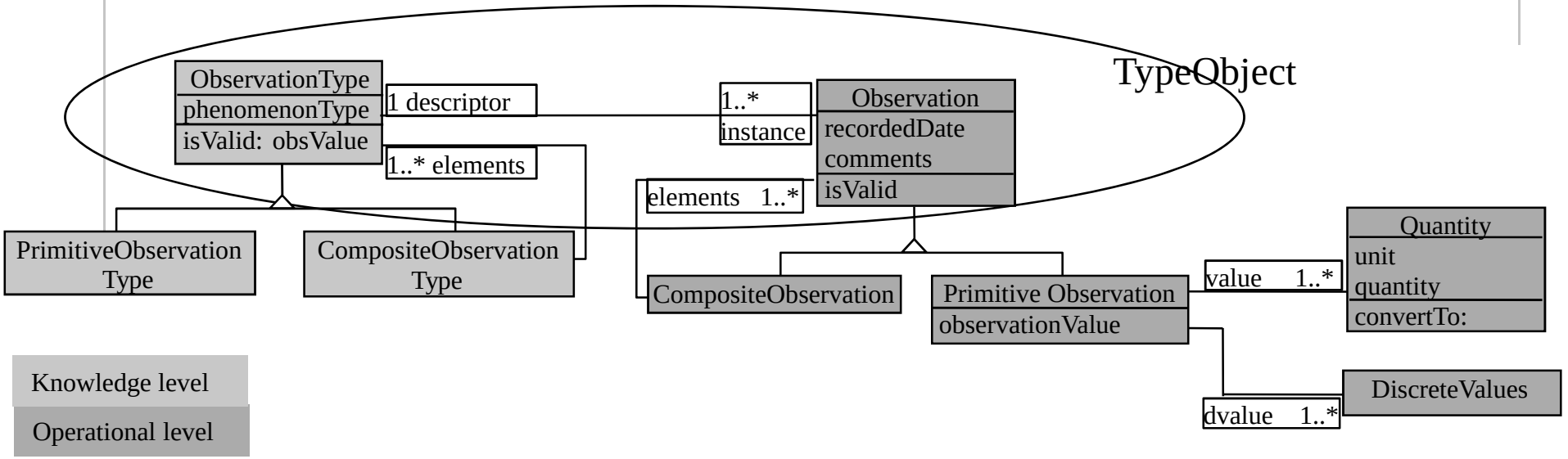
Observation Design
(instance diagram)



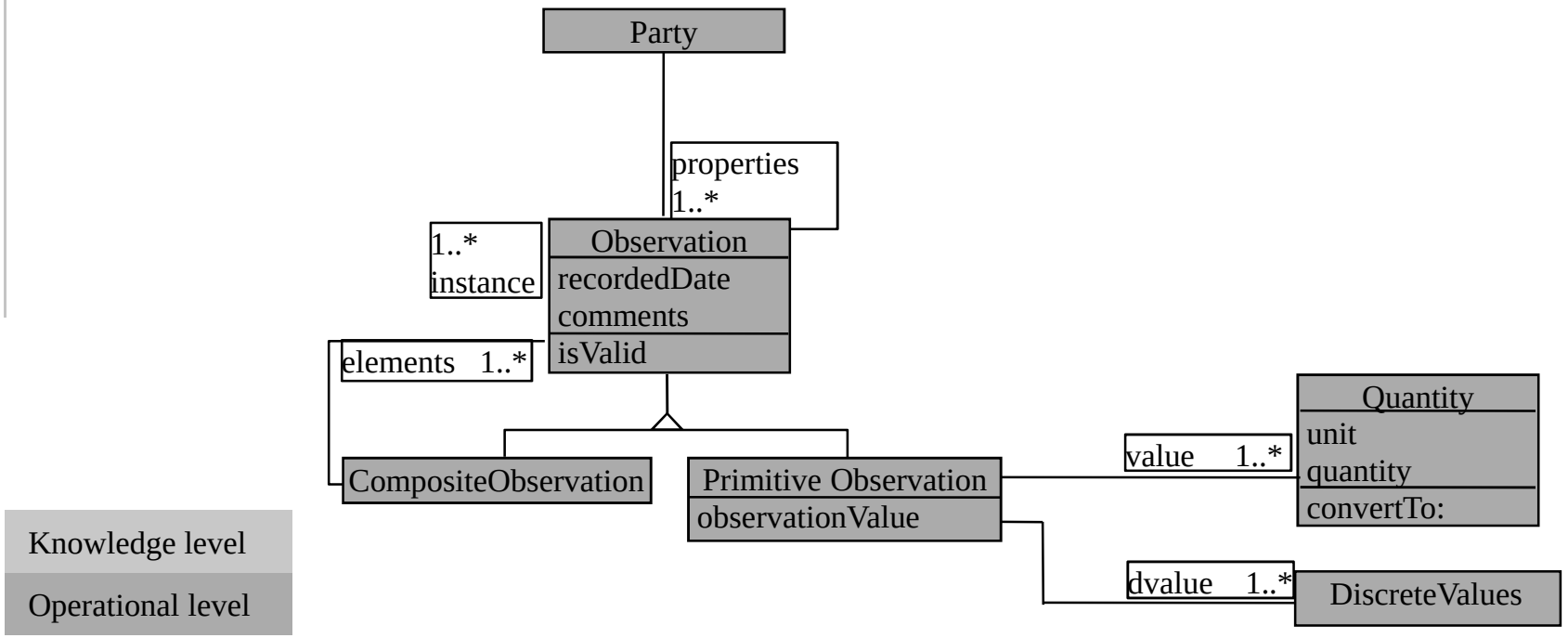
Observation Design
(instance diagram)



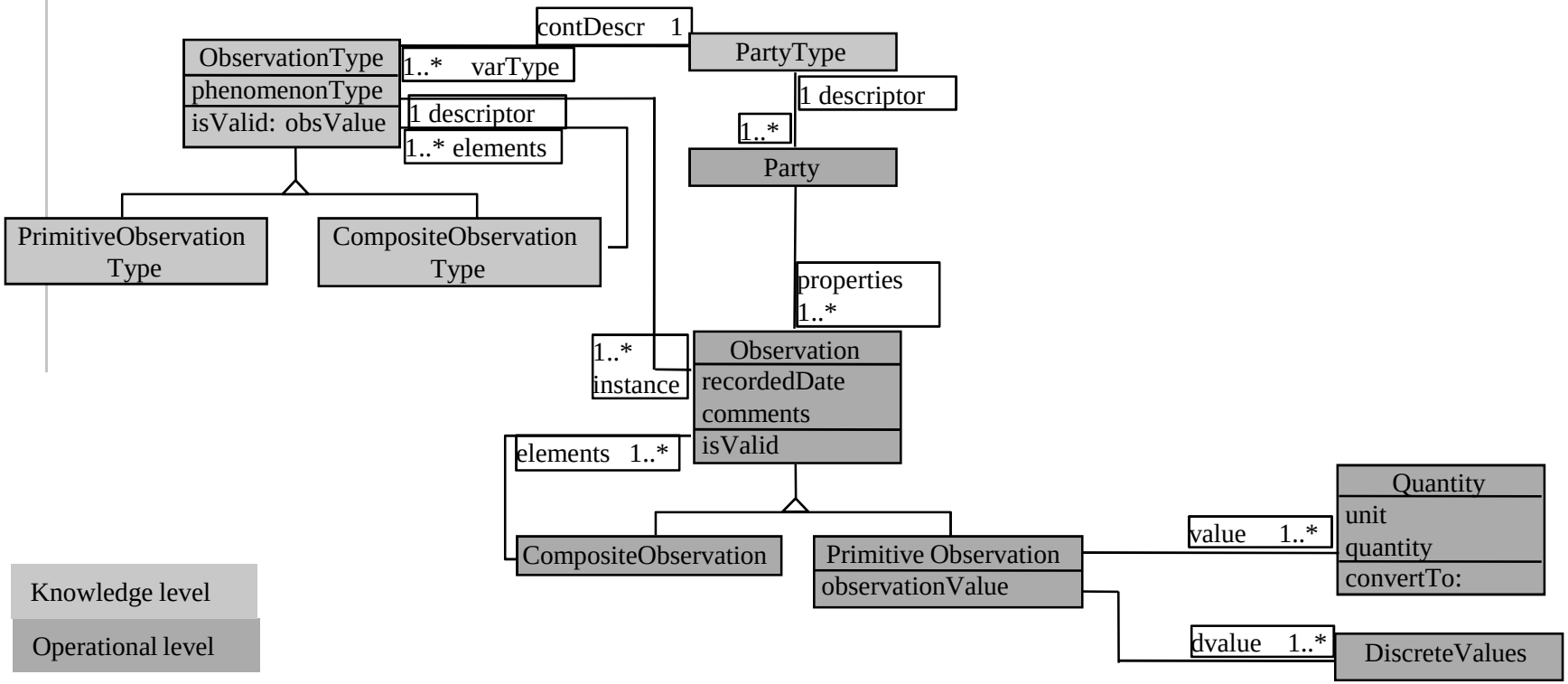
Observations: TypeObject



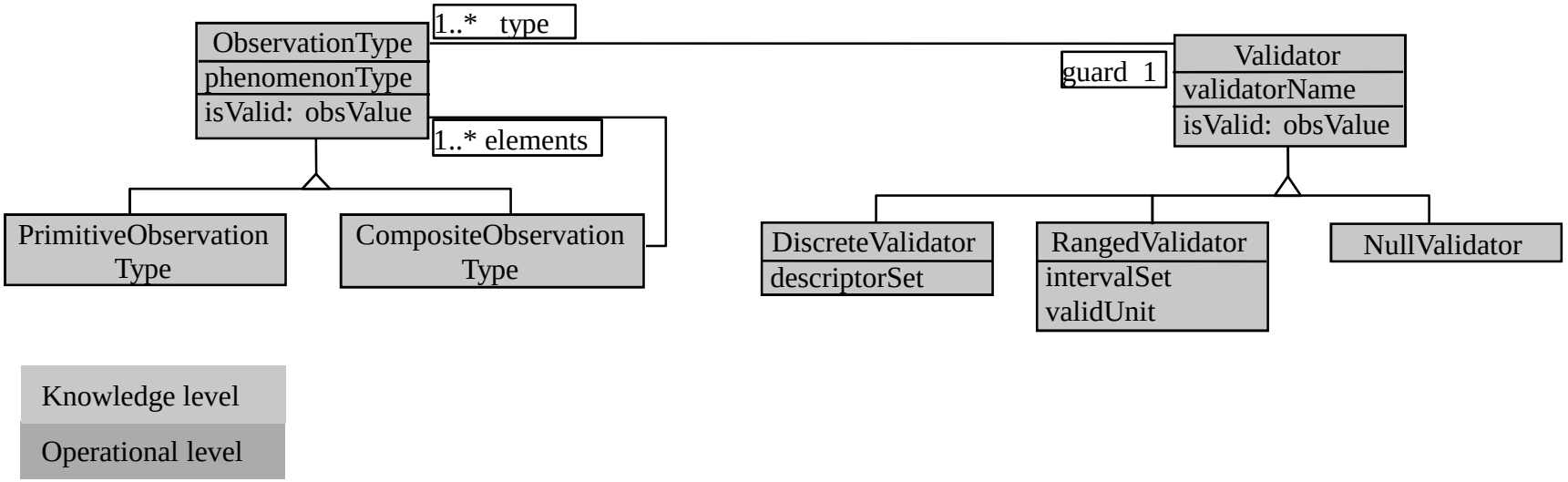
Observations: Properties



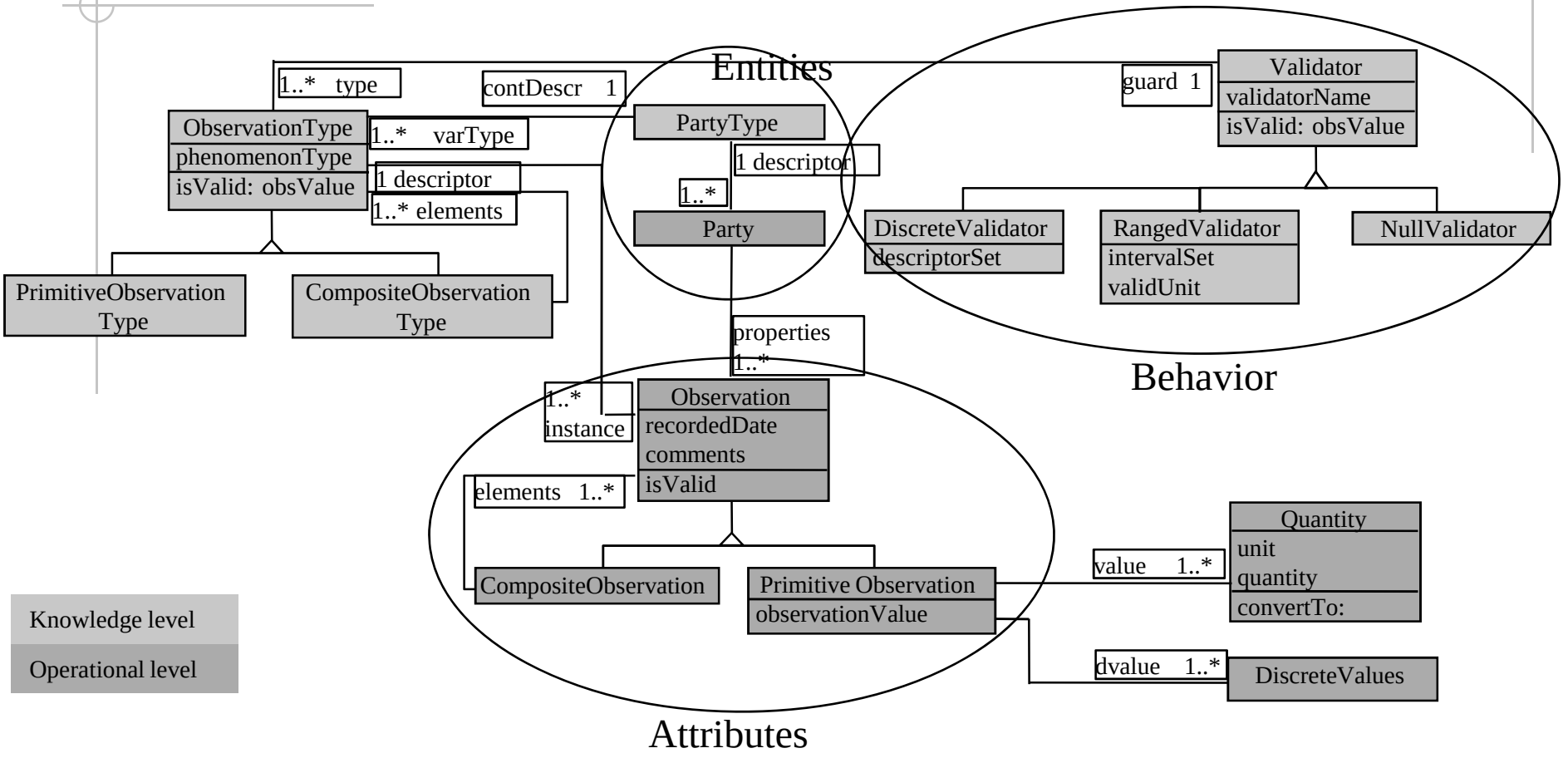
Observations: TypeSquare



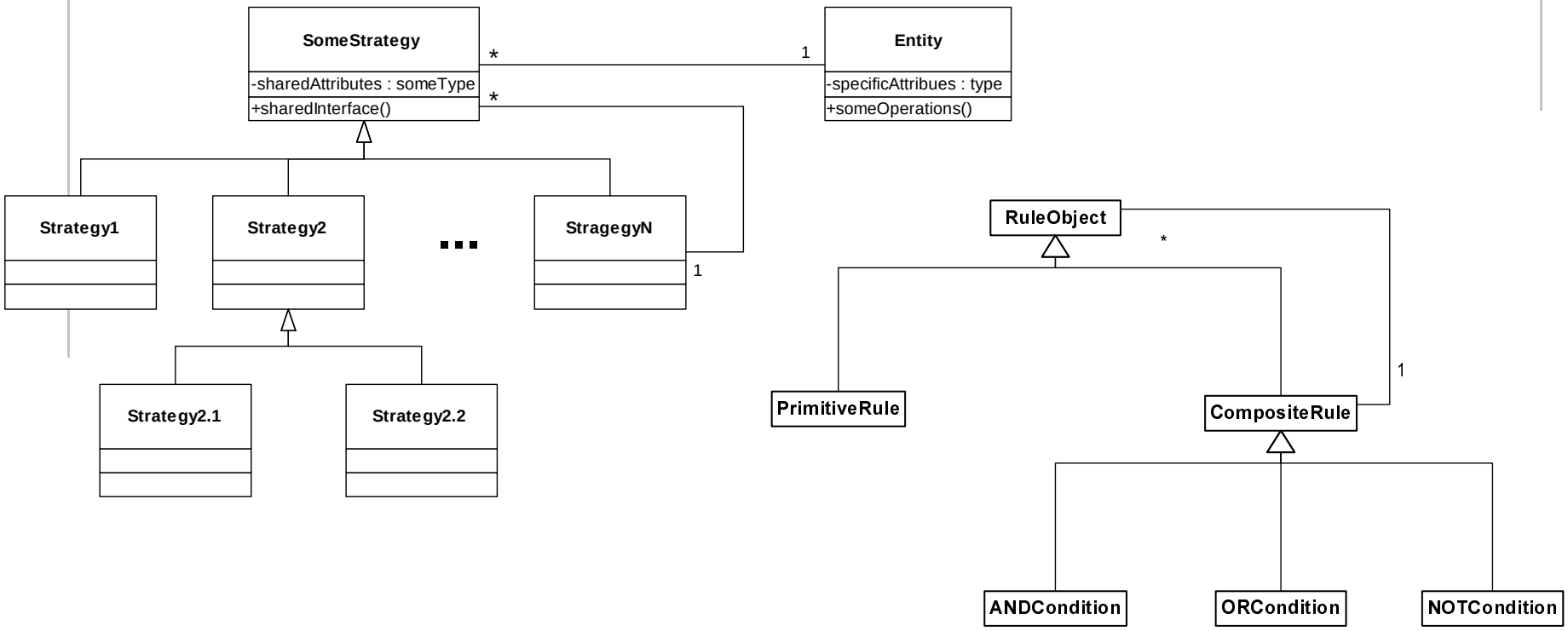
Observations: Strategy



Medical Observations Design



Strategies/Interpreters/RuleObjects (Behavior/Methods)



Design Patterns - GOF95

Composite Strategies ➔ Interpreter

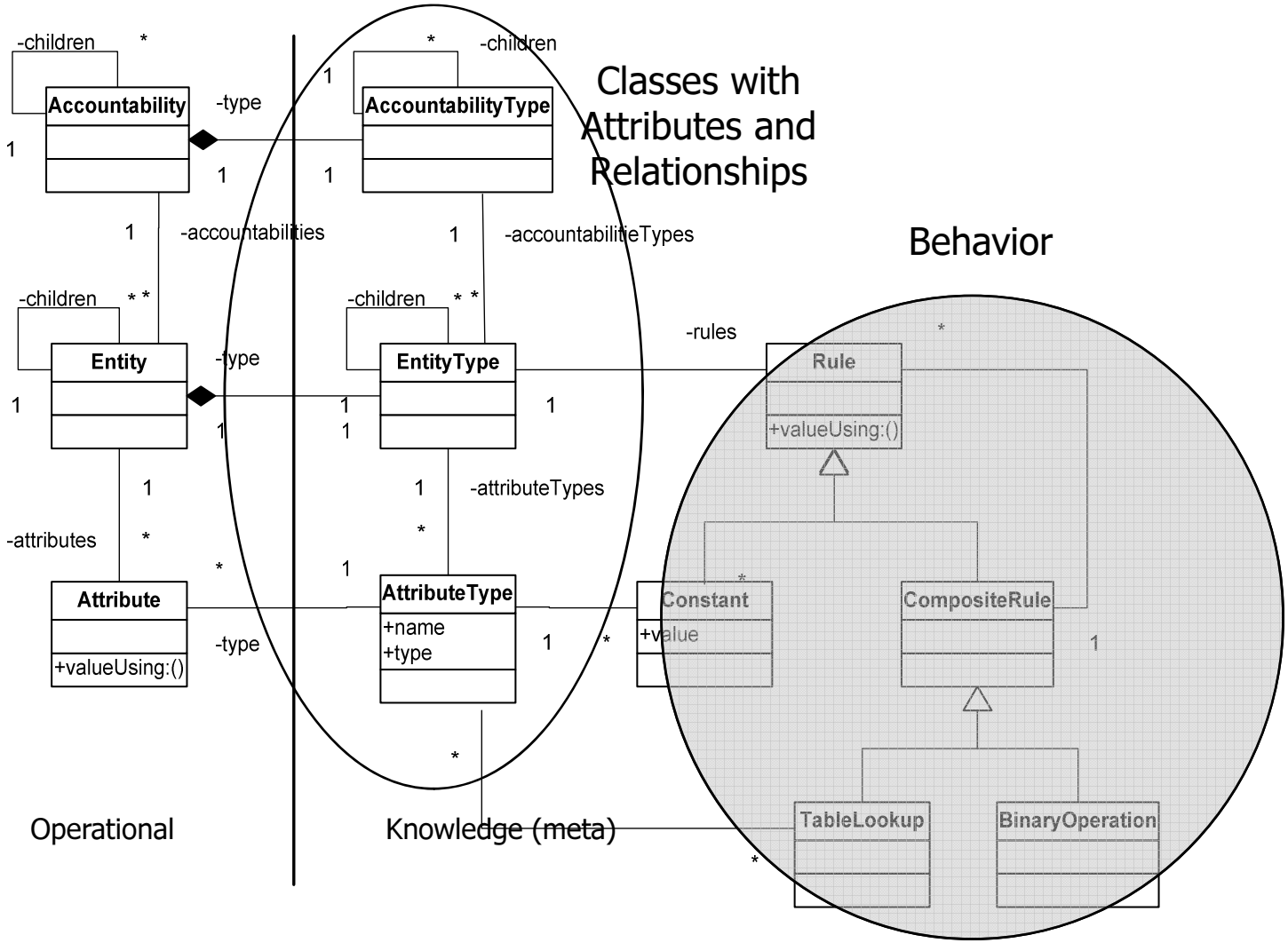
Composite Strategies

Problem: Strategy leads to a big class hierarchy, one class for each kind of policy.

Solution: Make Composite Strategies using Primitive Operations.

=> Interpreter pattern

Adaptive Object Model "Common Architecture"



Process for Developing Adaptable Systems

- Developed Iteratively and Incrementally.
  - Be Agile, Scrum, XP, Retrospective
- Get Customer Feedback early and often.
  - User Scenarios – User Stories…
- *Add flexibility only when and where needed.*
- Provide Test Cases and Suites for both the Object-Model and the Meta-Model.
- Develop Support Tools and Editors for manipulating the metadata.

Very similar to Evolving Frameworks by Roberts and Johnson PLoPD3

Refactoring Addresses Some Key Leverage Points

- Refactoring is a technique that works with Brooks' "promising attacks" (from "No Silver Bullet"):
  - buy rather than build: restructuring interfaces to support commercial SW
  - grow don't build software: software growth involves restructuring (isn't this core to Agile???)
  - requirements refinements and rapid prototyping: refactoring supports such design exploration, and adapting to changing customer needs
  - support great designers: a tool in a designer's tool chest.

Papers and Web Sites

- Bill Opdyke's Thesis
- Don Robert's Thesis
  - http://st-www.cs.illinois.edu/users/droberts/thesis.pdf
- Refactoring Browser
  - http://st-www.cs.illinois.edu/users/brant/Refactory
- Evolving Frameworks
  - http://st-www.cs.uiuc.edu/users/droberts/evolve.html
- Extreme Programming
  - http://www.c2.com/cgi-bin/wiki?ExtremeProgramming

More Web Sites

- Wiki wiki web
  - http://c2.com/cgi/wiki?WikiPagesAboutRefactoring
- The Refactory, Inc.
  - http://www.refactory.com
- Martin Fowler's Refactoring Pages
  - http://www.refactoring.com/
- Adaptive Object Models
  - http://www.adaptiveobjectmodel.com/

That's All