More patterns & hints

Tijs van der Storm
(storm@cwi.nl / @tvdstorm)

CWI Centrum Wiskunde & Informatica

University of Amsterdam

Recap

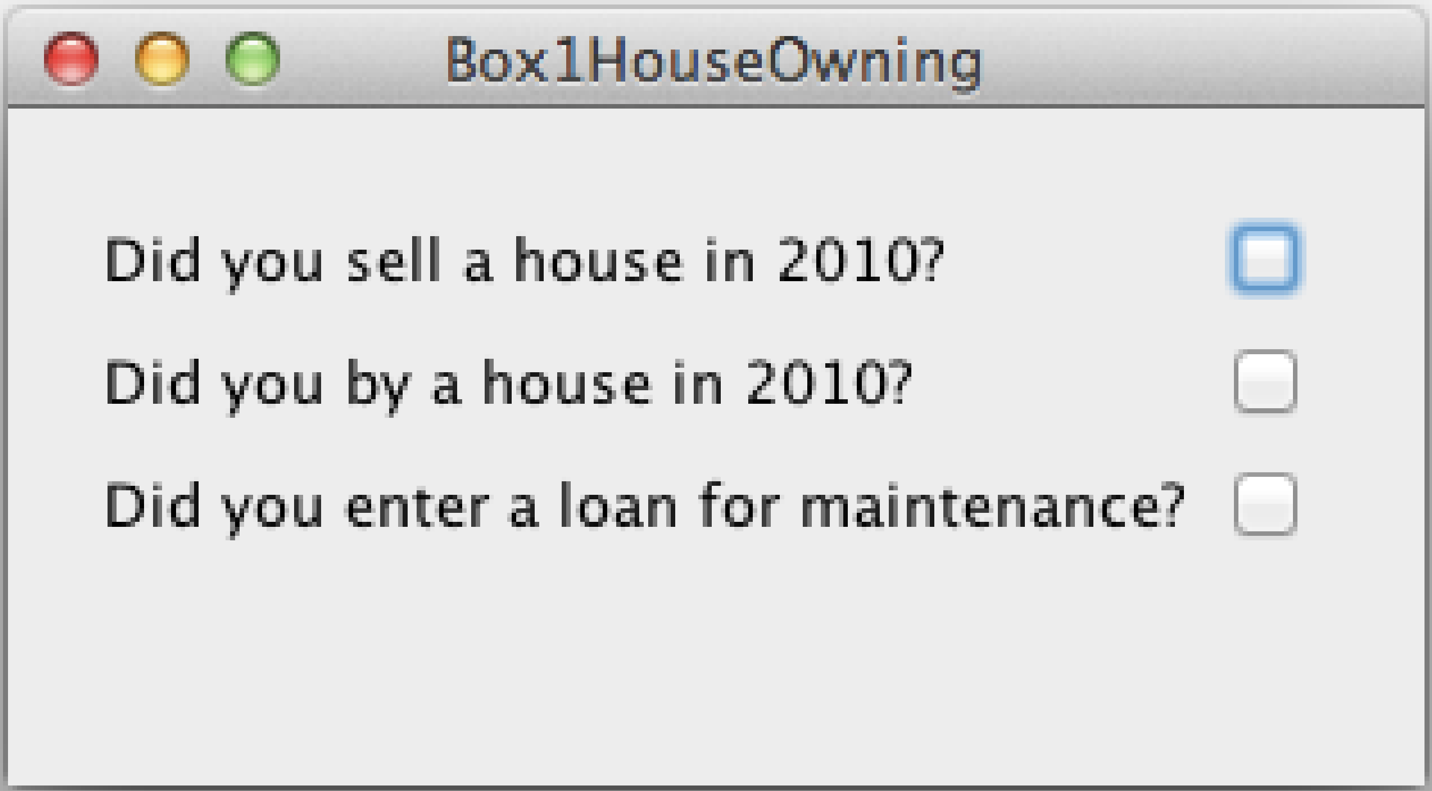
- Parsing: turns text into tree
- Grammars *describe* syntax
- Generate parser from grammar
- Generated code *creates* AST nodes
- Abstract Syntax Tree: tree without syntactic noise (layout, comments, keywords, ...)

More recap

- Use Composite for ASTs
- Use Visitor for traversal of ASTs
  - (or Interpreter)
- Separate typeOf from type checking
- Separate statement checking from expression checking

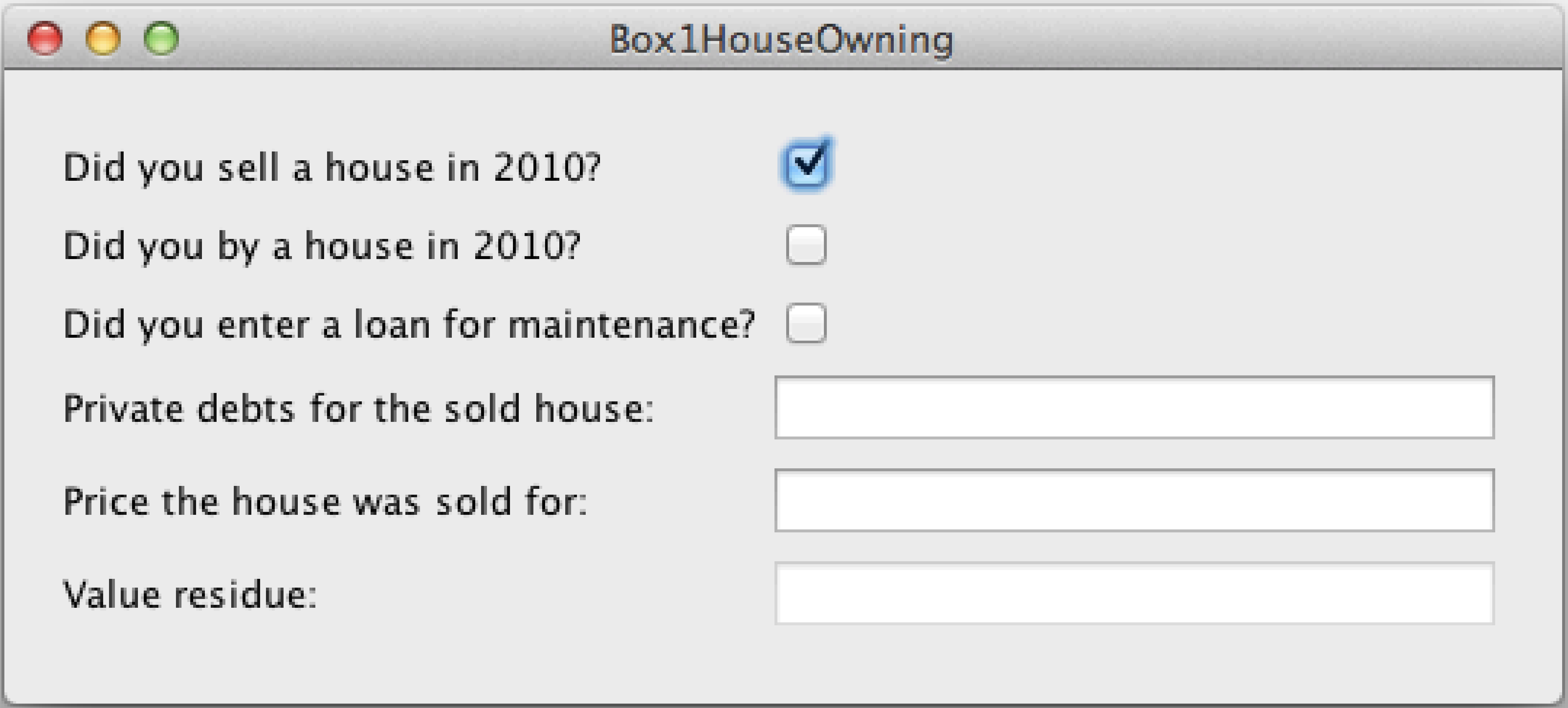
Revisiting an example

```
// Stuff dealing with house ownership
form Box1HouseOwning {
  "Did you sell a house in 2010?" hasSoldHouse: boolean
  "Did you by a house in 2010?" hasBoughtHouse: boolean
  "Did you enter a loan for maintenance?" hasMaintLoan: boolean
  if (hasSoldHouse) {
    "Private debts for the sold house:" privateDebt: integer
    "Price the house was sold for:" sellingPrice: integer
    "Value residue:" valueResidue
         = integer (sellingPrice - privateDebt)
  }
}
```


Box1HouseOwning

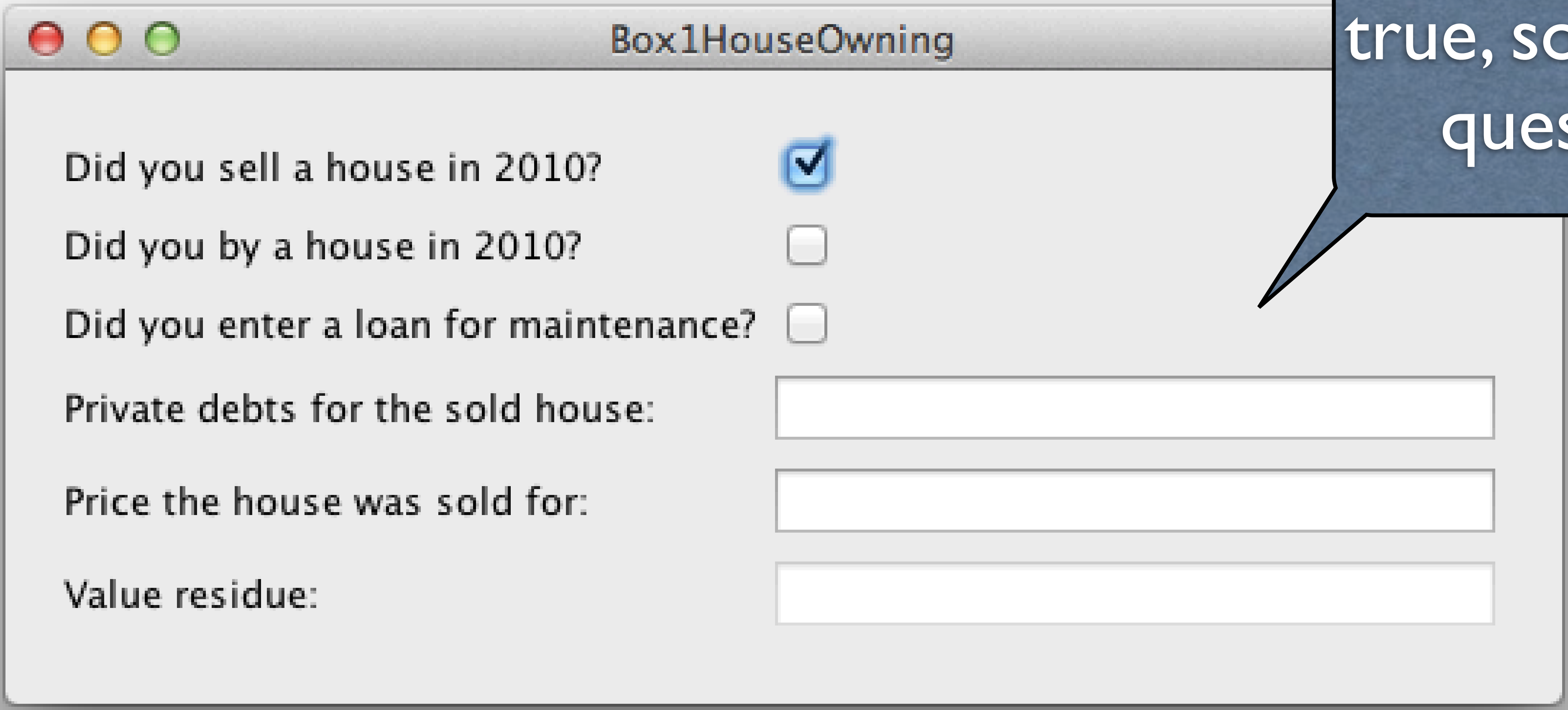
- Did you sell a house in 2010? ☐
- Did you by a house in 2010? ☐
- Did you enter a loan for maintenance? ☐

Initial state: first 3 question have *true* as condition



Box1HouseOwning

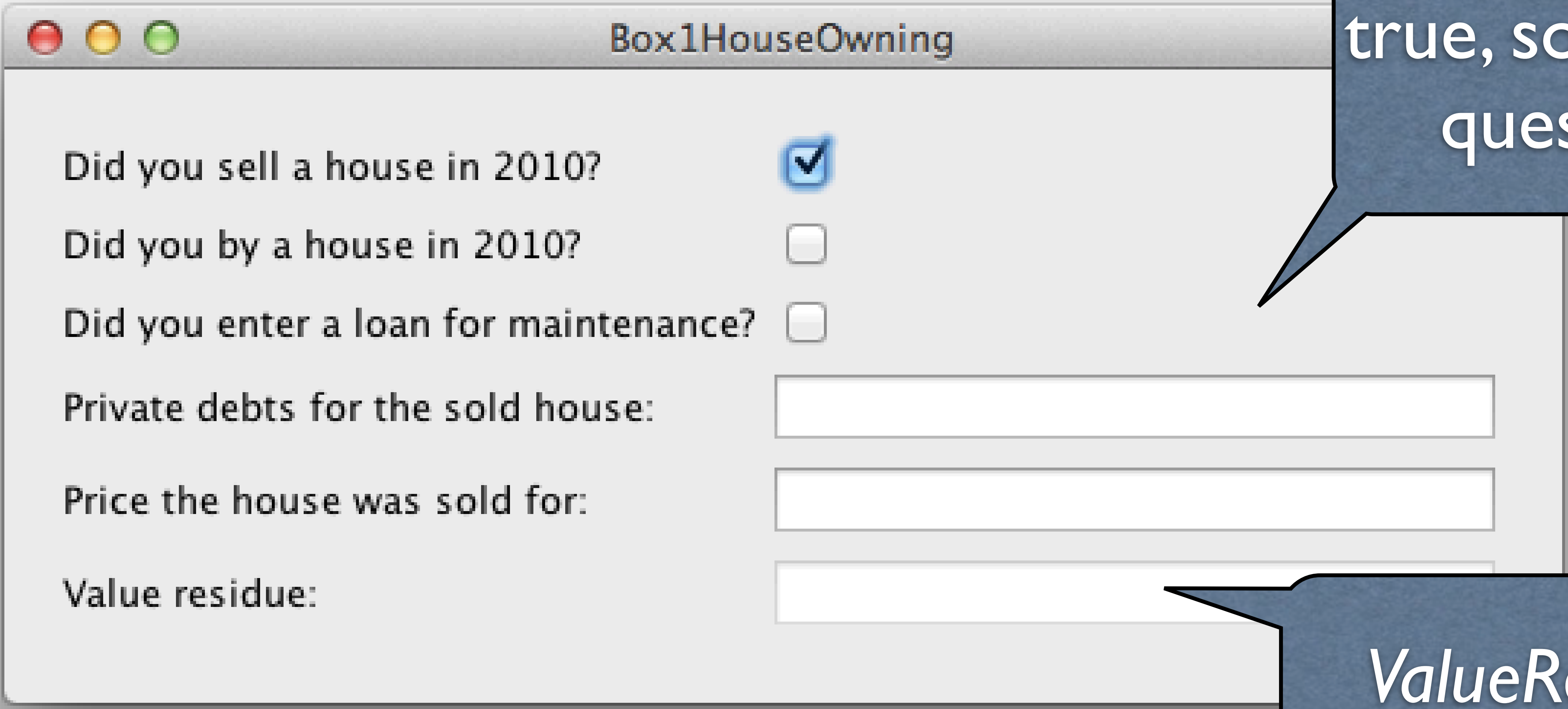
Did you sell a house in 2010?	✓
Did you by a house in 2010?	☐
Did you enter a loan for maintenance?	☐
Private debts for the sold house:	
Price the house was sold for:	
Value residue:	



Box1HouseOwning

Did you sell a house in 2010?	✓
Did you by a house in 2010?	☐
Did you enter a loan for maintenance?	☐
Private debts for the sold house:	
Price the house was sold for:	
Value residue:	

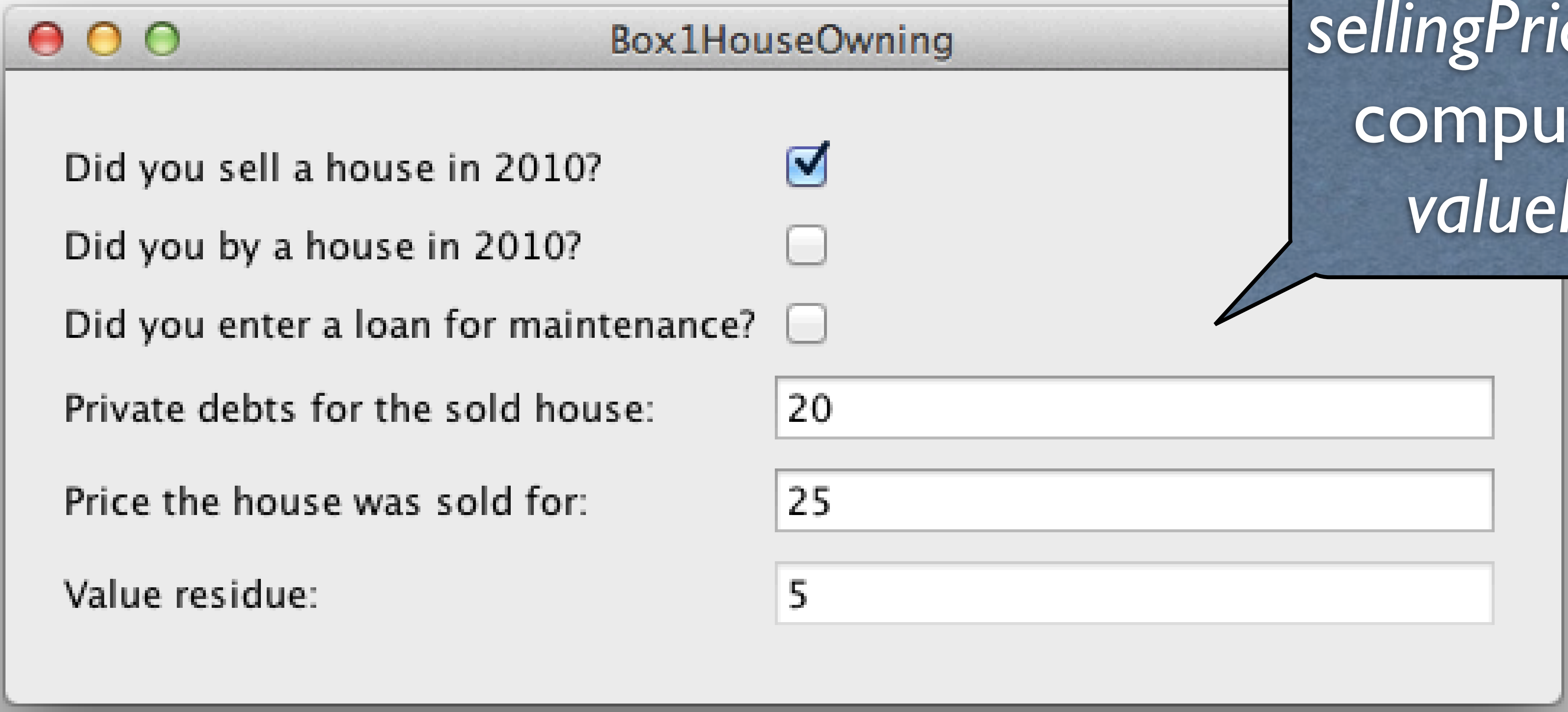
Now *hasSoldHouse* is true, so 3 more questions


Box1HouseOwning

Did you sell a house in 2010?	✓
Did you by a house in 2010?	☐
Did you enter a loan for maintenance?	☐
Private debts for the sold house:	
Price the house was sold for:	
Value residue:	

Now *hasSoldHouse* is true, so 3 more questions

ValueResidue is computed (widget is read-only); value is still undefined


Box1HouseOwning

Did you sell a house in 2010?	✓
Did you by a house in 2010?	☐
Did you enter a loan for maintenance?	☐
Private debts for the sold house:	20
Price the house was sold for:	25
Value residue:	5

Filling in *privateDebt* and *sellingPrice* triggers computation of *valueResidue*

So what do we need

- Draw widgets if conditions are true
- Listen to change events to trigger computed questions and conditions
- Make certain parts of GUI (in)visible depending on conditions.

Interpretation

- typeOf: Exp → Type
- typeCheck: Exp → List<Error>
- typeCheck: Stat → List<Error>
- “abstract interpreters”

Interpretation

- eval: Exp → Value
- render: Stat → GUI + observers

Modularizing the real stuff

- Expressions have no visible representation in the GUI
- Rendering of questions is *dependent* on expression evaluation, but not the other way around
- → Separate rendering from expression evaluation

```java
public class And extends Binary {

   public And(Expr lhs, Expr rhs) {
      super(lhs, rhs);
   }

   @Override
   public <T> T accept(Visitor<T> visitor) {
      return visitor.visit(this);
   }

   @Override
   public Type typeOf(Map<Ident, Type> typeEnv) {
      return new Bool();
   }

}
```

```java
public class And extends Binary {

   public And(Expr lhs, Expr rhs) {
      super(lhs, rhs);
   }

   @Override
   public <T> T accept(Visitor<T> visitor) {
      return visitor.visit(this);
   }

   @Override
   public Type typeOf(Map<Ident, Type> typeEnv) {
      return new Bool();
   }

}
```

Composite pattern

```java
public class And extends Binary {

   public And(Expr lhs, Expr rhs) {
      super(lhs, rhs);
   }

   @Override
   public <T> T accept(Visitor<T> visitor) {
      return visitor.visit(this);
   }

   @Override
   public Type typeOf(Map<Ident, Type> typeEnv) {
      return new Bool();
   }

}
```

Composite pattern

Visitor pattern

```
public class And extends Binary {

   public And(Expr lhs, Expr rhs) {
      super(lhs, rhs);
   }

   @Override
   public <T> T accept(Visitor<T> visitor) {
      return visitor.visit(this);
   }

   @Override
   public Type typeOf(Map<Ident, Type> typeEnv) {
      return new Bool();
   }

}
```

Composite pattern

Visitor pattern

Interpreter pattern

- Exp.typeOf computes *Types*
- Exp.eval will compute *Values*
  - (or: Exp.accept and Visitor.visit)

```java
public abstract class Value { }
```

```java
public class Int extends Value {
   private final Integer value;

   public Int(Integer value) {
      this.value = value;
   }
}
```

```java
public class Str extends Value {
   private final String value;

   public Str(String value) {
      this.value = value;
   }
}
```

```java
public class Bool extends Value {
   private final boolean value;

   public Bool(boolean value) {
      this.value = value;
   }
}
```

Why not Object?

- Type unsafe: *eval* could accidentally return anything
- Semantics of built-in Integer, Boolean etc. not necessarily the same
    - “accidental reuse”
- Cannot extend built-in Integer, Boolean etc. with behavior

```java
public class Eval implements Visitor<Value> {

   private final Map<Ident, Value> env;

   public Eval(Map<Ident, Value> env) {
      this.env = Collections.unmodifiableMap(env);
   }


   @Override
   public Value visit(Add exp) {
         ...
   }


   @Override
   public Value visit(Div exp) {
         ...
   }
}
```

```java
public class Eval implements Visitor<Value> {

   private final Map<Ident, Value> env;

   public Eval(Map<Ident, Value> env) {
      this.env = Collections.unmodifiableMap(env);
   }


   @Override
   public Value visit(Add exp) {
         ...
   }


   @Override
   public Value visit(Div exp) {
         ...
   }
}
```

Eval returns *Values*

```
public class Eval implements Visitor<Value> {

   private final Map<Ident, Value> env;

   public Eval(Map<Ident, Value> env) {
      this.env = Collections.unmodifiableMap(env);
   }


   @Override
   public Value visit(Add exp) {
         ...
   }


   @Override
   public Value visit(Div exp) {
         ...
   }
}
```

Eval returns *Values*

The environment

```
public class Eval implements Visitor<Value> {

  private final Map<Ident, Value> env;

  public Eval(Map<Ident, Value> env) {
    this.env = Collections.unmodifiableMap(env);
  }


  @Override
  public Value visit(Add exp) {
        ...
  }


  @Override
  public Value visit(Div exp) {
        ...
  }
}
```

Eval returns *Values*

The environment

Interpret every kind of expression

```java
@Override
public Value visit(Add exp) {
   Value l = exp.getLhs().accept(this);
   Value r = exp.getRhs().accept(this);
   return l.add(r);
}

@Override
public Value visit(Mul exp) {
   Value l = exp.getLhs().accept(this);
   Value r = exp.getRhs().accept(this);
   return l.mul(r);
}
```

Eval *lhs* and *rhs* and then *add*

```
@Override
public Value visit(Add exp) {
   Value l = exp.getLhs().accept(this);
   Value r = exp.getRhs().accept(this);
   return l.add(r);
}

@Override
public Value visit(Mul exp) {
   Value l = exp.getLhs().accept(this);
   Value r = exp.getRhs().accept(this);
   return l.mul(r);
}
```

```java
@Override
public Value visit(Add exp) {
   Value l = exp.getLhs().accept(this);
   Value r = exp.getRhs().accept(this);
   return l.add(r);
}

@Override
public Value visit(Mul exp) {
   Value l = exp.getLhs().accept(this);
   Value r = exp.getRhs().accept(this);
   return l.mul(r);
}
```

Eval *lhs* and *rhs* and then *add*

Eval *lhs* and *rhs* and then *multiply*

Variable lookup

```
@Override
public Value visit(Ident var) {
   if (env.containsKey(var)) {
      return env.get(var);
   }
   return Undefined.UNDEF;
}
```

Special value (like ErrorType) if a variable is not bound yet

Real computation on *Values*

- Abstract *Value* type "supports" operations of all types
- Subclasses override where needed
  - E.g., Int extends Value, overrides *add*, *mul*, etc.
- Type checking ensures only the right methods will be called.

```
    public Value add(Value arg) {
       throw new UnsupportedOperationException();
    }

    public Value pos() {
       throw new UnsupportedOperationException();
    }

    public Value div(Value arg) {
       throw new UnsupportedOperationException();
    }

    public Value mul(Value arg) {
       throw new UnsupportedOperationException();
    }

    public Value sub(Value arg) {
       throw new UnsupportedOperationException();
    }

    public Value and(Value arg) {
       throw new UnsupportedOperationException();
    }
```

Public interface

Default methods in *Value*

```
protected Value addInt(Int arg) {
   throw new UnsupportedOperationException();
}

protected Value divInt(Int arg) {
   throw new UnsupportedOperationException();
}

protected Value mulInt(Int arg) {
   throw new UnsupportedOperationException();
}

protected Value subInt(Int arg) {
   throw new UnsupportedOperationException();
}
```

Implementation with double dispatch

```
@Override
public Value add(Value arg) {
   return arg.addInt(this);
}

@Override
public Value sub(Value arg) {
   return arg.subInt(this);
}

@Override
public Value div(Value arg) {
   return arg.divInt(this);
}

@Override
public Value mul(Value arg) {
   return arg.mulInt(this);
}
```

Implementation in *Int* (subclass of *Value*)

```java
/*
 * NB: below the arguments are reversed
 * because of double dispatch.
 */
@Override
protected Value addInt(Int arg) {
   return new Int(arg.getValue() + getValue());
}

@Override
protected Value subInt(Int arg) {
   return new Int(arg.getValue() - getValue());
}

@Override
protected Value mulInt(Int arg) {
   return new Int(arg.getValue() * getValue());
}

@Override
protected Value divInt(Int arg) {
   return new Int(arg.getValue() / getValue());
}
```

Finally! Real computation

Rendering

- Just an interpretation...
- No values, but "drawing on a canvas"
- And installing observers.

Remember?

```
public interface Visitor {
   void visit(Computed stat);
   void visit(Answerable stat);
   void visit(IfThen stat);
   void visit(IfThenElse stat);
   void visit(Block stat);
}
```

```
public class Renderer implements Visitor {
   private final JPanel panel;
   private final State state;

   public static JPanel render(Stat stat, State state) {
      Renderer r = new Renderer(state);
      stat.accept(r);
      return r.getPanel();
   }

   private JPanel getPanel() {
      return panel;
   }

   private Renderer(State state) {
      this.state = state;
      this.panel = new JPanel();
   }
   ...
}
```

Static entry point

```java
@Override
public void visit(Answerable stat) {
   addLabel(stat.getLabel());
   Control ctl = typeToWidget(stat.getType(), true);
   registerHandler(stat, ctl);
   add(ctl);
}
```

```
@Override
public void visit(Answerable stat) {
   addLabel(stat.getLabel());
   Control ctl = typeToWidget(stat.getType(), true);
   registerHandler(stat, ctl);
   add(ctl);
}
```

“Draw” the label

```
@Override
public void visit(Answerable stat) {
   addLabel(stat.getLabel());
   Control ctl = typeToWidget(stat.getType(), true);
   registerHandler(stat, ctl);
   add(ctl);
}
```

“Draw” the label

Get a widget based on type

```
@Override
public void visit(Answerable stat) {
   addLabel(stat.getLabel());
   Control ctl = typeToWidget(stat.getType(), true);
   registerHandler(stat, ctl);
   add(ctl);
}
```

“Draw” the label

Get a widget based on type

Another visitor!

```
@Override
public void visit(Answerable stat) {
   addLabel(stat.getLabel());
   Control ctl = typeToWidget(stat.getType(), true);
   registerHandler(stat, ctl);
   add(ctl);
}
```

"Draw" the label

Get a widget based on type

Add an event listener

Another visitor!

```java
@Override
public void visit(Answerable stat) {
   addLabel(stat.getLabel());
   Control ctl = typeToWidget(stat.getType(), true);
   registerHandler(stat, ctl);
   add(ctl);
}
```

"Draw" the label

Get a widget based on type

Add an event listener

Draw the widget

Another visitor!

```java
@Override
public void visit(Computed stat) {
   addLabel(stat.getLabel());
   Control ctl = typeToWidget(stat.getType(), false);
   registerComputedDeps(stat, ctl);
   registerPropagator(stat);
   initValue(stat, ctl);
   add(ctl);
}
```

Mostly idem.

```
@Override
public void visit(Computed stat) {
   addLabel(stat.getLabel());
   Control ctl = typeToWidget(stat.getType(), false);
   registerComputedDeps(stat, ctl);
   registerPropagator(stat);
   initValue(stat, ctl);
   add(ctl);
}
```

```java
@Override
public void visit(Computed stat) {
   addLabel(stat.getLabel());
   Control ctl = typeToWidget(stat.getType(), false);
   registerComputedDeps(stat, ctl);
   registerPropagator(stat);
   initValue(stat, ctl);
   add(ctl);
}
```

Mostly idem.

Make it listen to other questions

```java
@Override
public void visit(Computed stat) {
   addLabel(stat.getLabel());
   Control ctl = typeToWidget(stat.getType(), false);
   registerComputedDeps(stat, ctl);
   registerPropagator(stat);
   initValue(stat, ctl);
   add(ctl);
}
```

Mostly idem.

Make it listen to other questions

Propagate recomputation

```
@Override
public void visit(Computed stat) {
   addLabel(stat.getLabel());
   Control ctl = typeToWidget(stat.getType(), false);
   registerComputedDeps(stat, ctl);
   registerPropagator(stat);
   initValue(stat, ctl);
   add(ctl);
}
```

Mostly idem.

Make it listen to other questions

Propagate recomputation

Initialize the value

```java
@Override
public void visit(final IfThenElse stat) {
   JPanel tru = render(stat.getBody(), state);
   JPanel fls = render(stat.getElseBody(), state);
   registerConditionDeps(stat.getCond(), tru, fls);
   tru.setVisible(false);
   fls.setVisible(false);
   addPanel(tru);
   addPanel(fls);
}
```

Use *render* recursively

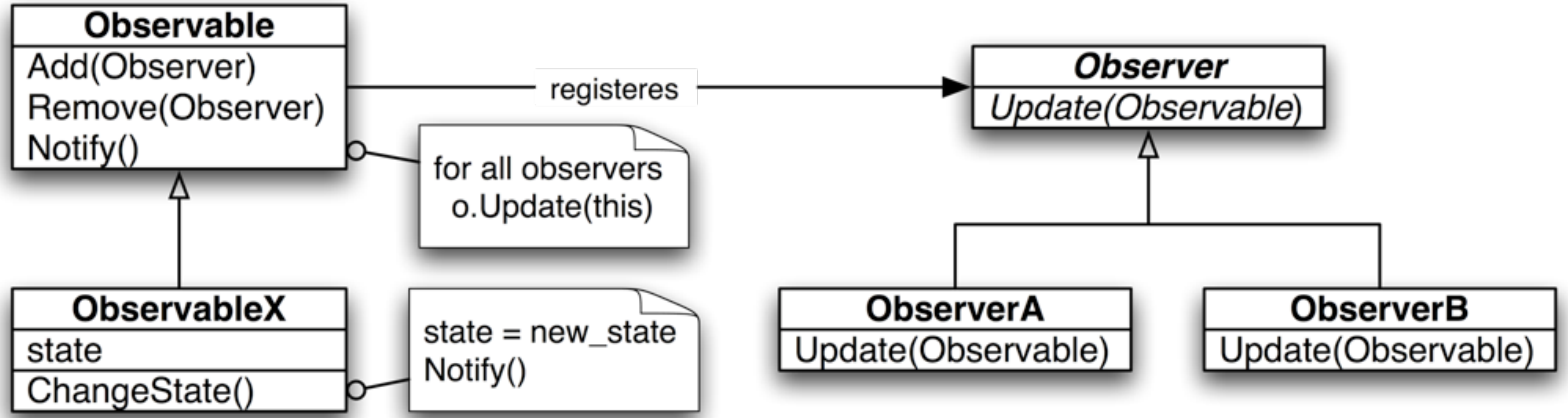
```
@Override
public void visit(final IfThenElse stat) {
   JPanel tru = render(stat.getBody(), state);
   JPanel fls = render(stat.getElseBody(), state);
   registerConditionDeps(stat.getCond(), tru, fls);
   tru.setVisible(false);
   fls.setVisible(false);
   addPanel(tru);
   addPanel(fls);
}
```

```java
@Override
public void visit(final IfThenElse stat) {
   JPanel tru = render(stat.getBody(), state);
   JPanel fls = render(stat.getElseBody(), state);
   registerConditionDeps(stat.getCond(), tru, fls);
   tru.setVisible(false);
   fls.setVisible(false);
   addPanel(tru);
   addPanel(fls);
}
```

Use *render* recursively

Make sure something happens if condition is recomputed

Observer Pattern



Dependencies

- Every question (answerable AND computed) is an *Observable*
- Expressions need reevaluation when one of its *used variables* (i.e. questions) changes.
- → Computed questions & conditions are *Observers* of the questions defining its *used* variables.

In Java...

- Class *Observable*
  - gives you: *setChanged*, *notifyObservers*, and *addObserver*
- Interface *Observer*
  - implement *update*
  - will be called if observable calls *notifyObservers*

```
public class State {
  private final Map<Ident, Value> env;
  private final Map<Ident, Observable> observables;

  public State() {
    this.env = new HashMap<Ident, Value>();
    this.observables = new HashMap<Ident, Observable>();
  }

  public void addObserver(Ident x, Observer obs) {
    observables.get(x).addObserver(obs);
  }

  public void putObservable(Ident x, Observable obs) {
    observables.put(x,  obs);
  }

  ...

}
```

Current values

```
public class State {
   private final Map<Ident, Value> env;
   private final Map<Ident, Observable> observables;

   public State() {
      this.env = new HashMap<Ident, Value>();
      this.observables = new HashMap<Ident, Observable>();
   }

   public void addObserver(Ident x, Observer obs) {
      observables.get(x).addObserver(obs);
   }

   public void putObservable(Ident x, Observable obs) {
      observables.put(x,  obs);
   }

   ...

}
```

Current values

Observables

```
public class State {
   private final Map<Ident, Value> env;
   private final Map<Ident, Observable> observables;

   public State() {
      this.env = new HashMap<Ident, Value>();
      this.observables = new HashMap<Ident, Observable>();
   }

   public void addObserver(Ident x, Observer obs) {
      observables.get(x).addObserver(obs);
   }

   public void putObservable(Ident x, Observable obs) {
      observables.put(x,  obs);
   }

   ...

}
```

```
public class State {
   private final Map<Ident, Value> env;
   private final Map<Ident, Observable> observables;

   public State() {
      this.env = new HashMap<Ident, Value>();
      this.observables = new HashMap<Ident, Observable>();
   }

   public void addObserver(Ident x, Observer obs) {
      observables.get(x).addObserver(obs);
   }

   public void putObservable(Ident x, Observable obs) {
      observables.put(x,  obs);
   }

   ...

}
```

Current values

Observables

Add observers

Two kinds of observers

- Condition observers
  - (IfThen and IfThenElse)
- Expression observers
  - (Computed questions)

```java
public class ComputedObserver implements Observer {
  private final Control control;
  private final State state;
  private final Computed stat;

  ...

  @Override
  public void update(Observable o, Object arg) {
    Value value = stat.getExpr().accept(new Eval(state.getEnv()));
    state.putValue(stat.getName(), value);
    state.notify(stat.getName());
    control.setValue(value);
  }
}
```

```java
public class ComputedObserver implements Observer {
   private final Control control;
   private final State state;
   private final Computed stat;

   ...

   @Override
   public void update(Observable o, Object arg) {
      Value value = stat.getExpr().accept(new Eval(state.getEnv()));
      state.putValue(stat.getName(), value);
      state.notify(stat.getName());
      control.setValue(value);
   }
}
```

Evaluate expression

```java
public class ComputedObserver implements Observer {
   private final Control control;
   private final State state;
   private final Computed stat;

   ...

   @Override
   public void update(Observable o, Object arg) {
      Value value = stat.getExpr().accept(new Eval(state.getEnv()));
      state.putValue(stat.getName(), value);
      state.notify(stat.getName());
      control.setValue(value);
   }
}
```

Evaluate expression

Store the value

```
public class ComputedObserver implements Observer {
   private final Control control;
   private final State state;
   private final Computed stat;

   ...

   @Override
   public void update(Observable o, Object arg) {
      Value value = stat.getExpr().accept(new Eval(state.getEnv()));
      state.putValue(stat.getName(), value);
      state.notify(stat.getName());
      control.setValue(value);
   }
}
```

Evaluate expression

Store the value

Notify dependants

```java
public class ComputedObserver implements Observer {
   private final Control control;
   private final State state;
   private final Computed stat;

   ...

   @Override
   public void update(Observable o, Object arg) {
      Value value = stat.getExpr().accept(new Eval(state.getEnv()));
      state.putValue(stat.getName(), value);
      state.notify(stat.getName());
      control.setValue(value);
   }
}
```

Evaluate expression

Store the value

Notify dependants

Update the GUI

```
public class ConditionObserver implements Observer {
   ...
   @Override
   public void update(Observable o, Object arg) {
      Value value = cond.accept(new Eval(state.getEnv()));
      boolean visible = value.isDefined() && ((Bool)value).getValue();
      tru.setVisible(visible);
      if (fls != null) {
         fls.setVisible(!visible);
      }
   }

}
```

Evaluate condition

```
public class ConditionObserver implements Observer {
   ...
   @Override
   public void update(Observable o, Object arg) {
      Value value = cond.accept(new Eval(state.getEnv()));
      boolean visible = value.isDefined() && ((Bool)value).getValue();
      tru.setVisible(visible);
      if (fls != null) {
         fls.setVisible(!visible);
      }
   }

}
```

```
public class ConditionObserver implements Observer {
   ...
   @Override
   public void update(Observable o, Object arg) {
      Value value = cond.accept(new Eval(state.getEnv()));
      boolean visible = value.isDefined() && ((Bool)value).getValue();
      tru.setVisible(visible);
      if (fls != null) {
         fls.setVisible(!visible);
      }
   }

}
```

Evaluate condition

Update visibility of widgets in branches

Patterns used

- Composite: AST, *Type*, *Value*
- Visitor: *typeCheck*, *eval*, *render*, *typeToWidget*
- Interpreter: *typeOf*
- Null Object: *Undefined*, *ErrorType*
- Double dispatch: binary operations
- Observer pattern: dependencies