Expressing Architectural Design Intent in Code

George Fairbanks
1 May 2013
Rhino Research
Software Architecture Consulting and Training
Summary

- **Design intent** is lost between design and code
  - Hard to infer design from code
  - But we can make it easier

- **Hints** in code preserve design intent
  - Easier to encode *extensional* intent
  - Hard to encode *intensional* intent (e.g. for all components ...)

- Big idea now – read patterns later
  - Read the full chapter: [http://goo.gl/NYxdy](http://goo.gl/NYxdy)
  - Talk on YouTube
It’s hard to infer design intent
Machines are solutions, not designs

• Why are they lying around?
• Won’t bad guys copy them?
• What can you learn by looking at them?
• What knowledge is needed to build your own rocket?
• What is missing from these rockets? I.e., what can’t you learn?

These are solutions, not designs
Programs also express solutions

- To a computer, these programs are equally good
- ... but not to a human

Example obfuscated code from Wikipedia

```c
void primes(int cap, int t, int composite) {
    int i, j;
    i = t / cap;
    j = t % cap;
    if (i <= 1)
        primes(cap, t+1, composite);
    else if (!j)
        primes(cap, t+1, j);
    else if (j == i && !composite)
        (printf("%d	", i), primes(cap, t+1, composite));
    else if (j > 1 && j < i)
        primes(cap, t+1, composite + !!(i % j));
    else if (t < cap * cap)
        primes(cap, t+1, composite);
}

int main() {
    primes(100, 0, 0);
}
```

```c
void p(int m, int t, int c) {
    ((t / m) <= 1) ? p(m, t+1, c) :
    !(t % m) ? p(m, t+1, t % m) :
    ((t % m)==(t / m) && !c) ?
    (printf("%d\t", (t / m)), p(m, t+1, c)) :
    ((t % m)> 1 && (t % m) < (t / m)) ?
    p(m, t+1, c + !((t / m) % (t % m))) :
    (t < m * m) ? p(m, t+1, c) : 0;
}

int main() {
    p(100, 0, 0);
}
```

Solution != design intent
In a nutshell

• A running machine is a valuable thing

• But we would also like to know:
  - How it works
  - Why it works
  - How to evolve and change it
  - The principles it embodies

• It’s hard, maybe impossible, to understand these by inspection

• Some machines are easier to understand than others

• Understandable code ‡ saved time
Model in code principle
Expressing design intent

- Kent Beck: Smalltalk Best Practice Patterns
- Intention Revealing Message (i.e., method name)

“What’s going on? Communication. ... Intention Revealing Messages are the most extreme case of writing for readers instead of the computer. As far as the computer is concerned, both versions are fine. The one that separates intention (what you want done) from implementation (how it is done) communicates better to a person.”
Beck’s example

- **Obvious solution:** invert colors in the callback

- **Expressing intent:**
  - Name method “highlight” to show intent
  - How do we highlight? We invert colors.
  - So, add another method “invertColors”

```smalltalk
doubleValueCallback(Drawable d)
    d.highlight();

highlight()
    this.invertColors();

invertColors()
    temp = this.getBackgroundColor();
    this.setBackgroundColor( this.getForegroundColor() );
    this.setForegroundColor( temp );
```

**Important idea:**
Write more code than necessary.
Extra code = *hints.*
We provide hints all the time

• Class names
   Airplane, Route

• Method names
   taxi(), takeoff(), land()

• Variable names
   fuelRemaining, flapAngle

• Object responsibilities (somewhat)

• Nesting
   Wing is part of a Plane

So, why does this work?
Model-in-code principle

• **Model-in-code principle:**
  - Expressing models in code helps comprehension and evolvability

• One kind of model: **Domain model**
  - Standard in OO programming (e.g., Booch method)
  - Objects mirror domain concepts
  - Stronger: Domain Driven Design (Eric Evans)

• **So, what about our architecture model in code?**
Architecture in code
Motivation: Why reveal architecture in code

• When reading code, want to know:
  ◆ Who talks to who
  ◆ Invariants and constraints
  ◆ Styles and patterns
  ◆ Performance requirements or guarantees
  ◆ Data structures used for communication
  ◆ Etc.

• Easy to see in architecture model

• Hard to see in code
Problem: Cannot express all architecture in code

- Source code is good at expressing solutions
- It cannot express all architecture design intent

... why not?
## Extensioonal and intensional

### Definitions

- **Extensional**: elements that are enumerated
  - E.g., “The system is composed of a modules A, B, and C”

- **Intensional**: elements that are universally quantified
  - E.g., “All filters can communicate via pipes”

<table>
<thead>
<tr>
<th>Intensional / Extensional</th>
<th>Architecture model element</th>
<th>Translation into code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Extensional</strong></td>
<td>Modules, components, connectors, ports, component assemblies</td>
<td>These correspond neatly to elements in the implementation, though at a zoomed-out higher level of abstraction (e.g., one component corresponds to multiple classes)</td>
</tr>
<tr>
<td>(defined by enumerated instances)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Intensional</strong></td>
<td>Styles, invariants, responsibility allocations, design decisions, rationale, protocols, quality attributes and models (e.g., security policies, concurrency models)</td>
<td>Implementation will conform to these, but they are not directly expressed in the code. Architecture model has general rule, code has examples.</td>
</tr>
<tr>
<td>(quantified across all instances)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Catalog of patterns
Pattern catalog

• Organizing a catalog of patterns is hard
  § Let’s organize it in slices

• **Slice 1: Module dependencies**
• Slice 2: Package structure
• Slice 3: Module ≠ Runtime
• Slice 4: Visible components
• Slice 5: Visible connectors
• Slice 6: Properties, styles, patterns
Slice 1: Module dependencies

- **Desire:** Express/enforce dependencies between modules

- **Pattern:** use tool to express dependencies
  - Option: VerifyDesign Ant task
  - Option: Framework, like .Net assemblies, OSGi bundles
  - Option: Murphy and Notkin “Reflexion”; Dean Sutherland module system; Halloran enforcing module layers (research)

- These allow you to specify dependencies
- Maybe to check them

- Some allow “do not depend on” specifications
Slice 2: Package structure

- **Desire:** Components visible in code

- Modules and components
  - No required alignment
    ... but easier if they do align

- Example of alignment:
  - modules A+B+C  \(\mapsto\) component Y
  - modules B+D+E  \(\mapsto\) component Z

- **Pattern:** align module and component boundaries
  - 1:1 module-component alignment?

- **Pattern:** module for interchange types
Slice 3: Module † Runtime

- **Centralized startup code**
  - Creation
  - Assembly
  - Initialization

- Alternative:
  - Scattered creation
  - Scattered assembly

- Aids comprehension
- Aids analysis

```java
public static void main(String[] args) {
    createPipes();
    createFilters();
    startFilters();
    ...
}
```

```java
protected static void createPipes() {
    pipeCleanupToTagging = new Pipe<EmailMessage>();
    pipeTaggingToMux = new Pipe<EmailMessage>();
    ...
}
```

```java
protected static void createFilters() {
    filterCleanup = new InputCleanupFilter();
    filterTagging = new TaggingFilter();
    ...
}
```

```java
protected static void startFilters() {
    filterCleanup.run();
    filterTagging.run();
    ...
}
```
... or use a configuration language

- **Pattern**: hoist initialization and configuration
  - Write in declarative language (e.g., XML)
  - Code reads config file, then creates, attaches, initiates
  - E.g., Struts, EJB, OSGi/Eclipse
Pattern catalog

• Organizing a catalog of patterns is hard
  ⃣ Let’s organize it in slices

• Slice 1: Module dependencies
• Slice 2: Package structure
• Slice 3: Module † Runtime
• **Slice 4: Visible components**
• Slice 5: Visible connectors
• Slice 6: Properties, styles, patterns
Slice 4: Visible components

- **Desire**: Components visible in code

- **Pattern**: Abstract class for a Filter, a kind of component
  - Differentiates components from other classes
  - Standardizes startup with run() and work() template pattern

- **Find in IDE via class hierarchy**
  - Show all subclasses of Filter or Component

```java
package infrastructure.pipeAndFilterStyle;

import infrastructure.Component;

abstract public class Filter extends Component implements Runnable {
    public void run() {
        try {
            this.work();
        } catch (Exception e) {
            System.exit(1);
        }
    }

    abstract protected void work() throws InterruptedException;
}
```
Slice 5: Visible connectors

- **Desire:** Connectors visible in code

- **Pattern:** Final class for Pipe, a kind of Connector
  - No subclasses

- **Hoists**
  - concurrency concern
  - Discuss: how?

- **Find in IDE**

```java
package infrastructure.pipeAndFilterStyle;

import infrastructure.Connector;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public final class Pipe<T> extends Connector {
    private BlockingQueue<T> myPipe = new LinkedBlockingQueue<T>();
    private boolean isClosed = false;

    public T blockingRead() throws InterruptedException {
        if (myPipe.isEmpty()) return null;
        T t = myPipe.take();
        return t;
    }

    public void blockingWrite(T t) throws InterruptedException {
        if (isClosed()) throw new IllegalStateException();
        myPipe.put(t);
    }

    public void close() throws InterruptedException {
        this.isClosed = true;
    }

    public boolean isClosed() {
        return isClosed;
    }

    public boolean isClosedAndEmpty() {
        if (isClosed() && myPipe.isEmpty()) return true;
        else return false;
    }
}
```
Slice 6: Properties, styles and patterns

- **Desire**: make properties, styles, and patterns visible

  - **Pattern**: express properties with names
    - E.g., `asynchronousWrite()`
    - E.g., `readOnlyProvidedInventoryPort`

  - **Pattern**: express properties with annotations
    - E.g., `@asynchronous write()`
    - E.g., `@readonly providedInventoryPort`
    - Enables machine checkability

  - **Pattern**: express styles/patterns with names
    - E.g., `InventoryFacade`, `ASTVisitor`
    - E.g., `TaggingFilter`, `LyricsServer`, `VOIPPeer`
Summary

• **Design intent** is lost between design à code
  - Hard to infer design from code
  - But we can make it easier

• **Hints** in code preserve design intent
  - Easier to encode *extensional* intent
  - Hard to encode *intensional* intent (e.g. for all components ...)

• Big idea now – read patterns later
  - Read the full chapter: [http://goo.gl/NYxdy](http://goo.gl/NYxdy)
  - Talk on YouTube
Shameless plugging

- **E-book $10**
  - One price, 3 formats
  - PDF, ePub, Mobi
  - No DRM
  - [http://RhinoResearch.com](http://RhinoResearch.com)
  - Coupon: SATURN2013

- **Hardback**
  - Amazon.com
  - About $35 street price

- **Free chapters**
  - This one: [http://goo.gl/NYxdy](http://goo.gl/NYxdy)
  - Other chapters on website