Code-level Techniques
(to Explicitly Capture Design Intent in a Non-formal Way)

EE382V – Software Architecture and Design Intent

Engin Uzuncaova
Daryl Shannon
Outline

– General Picture
– What is “Design by Contract”? 
– Eiffel
– Relation to Architecture and Intent
Software Architecture:
The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Design Intent:
Logical reasoning behind the mapping from the requirements domain to the architectural abstractions.

Implementation:
The process of constructing an actual artifact from a design.
Implementation vs. Interaction

- indicates what modules are present and to what modules they refer
- fails to capture architectural composition
- lines represent programming language relationships

- highlights architectural design
- reflects abstract interactions
Problems..?

• It limits the expressiveness of the architectural description only to those defined by the implementation language.

• Low-level entities used in the architecture makes it harder to reason about the architectural design.

• Algorithmic aspects of the program interfere with the architectural abstractions.
Design by Contract

What is Design by Contract™?

- A method of software construction that designs the components of a system so that they will cooperate on the basis of precisely defined contracts

How does DbC work?

- For the execution of any routine
  - DbC ensures that before execution begins, all conditions required for correct execution are met.
  - Upon completion, it ensures that the routine actually has executed as expected.
  - DbC ensures that the instance is in a valid state at all critical times.

http://www.eiffel.com
Motivation

• Software failures are expensive
  – Reliability
    • Correctness - specification
    • Robustness - ?

• Software itself is expensive
  – Reusability
An Example Contract

<table>
<thead>
<tr>
<th>Party</th>
<th>Obligations</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Provide letter or package of no more than 5 kgs, each dimension no more than 2 meters. Pay 100 francs.</td>
<td>Get package delivered to recipient in four hours or less.</td>
</tr>
<tr>
<td>Supplier</td>
<td>Deliver package to recipient in four hours or less.</td>
<td>No need to deal with deliveries too big, too heavy, or unpaid.</td>
</tr>
</tbody>
</table>

```
if new = Void then
    ...
else
    ...
end
```

```
routine_name (argument declarations) is
    -- Header comment
    require
        Precondition
    do
        Routine body, i.e. instructions
    ensure
        Postcondition
end
```

```
put_child (new: NODE) is
    -- Add new to the children of current node
    require
        new /= Void
    do
        ...
    ensure
        new.parent = Current;
        child_count = old child_count + 1
end -- put_child
```
Software Contract

- Pre-condition
- Post-condition
- Class invariant

 Assertions
(result from bug; they are not special cases)

```plaintext
put_child (new: NODE) is
  -- Add new to the children of current node
  require
    new /= Void
  do
    ... Insertion algorithm ...
  ensure
    new.parent = Current;
    child_count = old child_count + 1
  end -- put_child

invariant
  left /= Void implies (left.parent = Current);
  right /= Void implies (right.parent = Current)
```
Four Key Benefits

- Constructing correct programs
- Automatic documentation
- Debugging and testing
- Exception handling
- Reusability
Eiffel

- Eiffel development methodology
  - Pure OO, focused on quality
- Eiffel programming language
  - Eiffel compiler (to ANSI C and MSIL)
- Development environment
  - EiffelStudio, Eiffel ENVisioN
Example (1)

- Class TIME_OF_DAY
  - Instances are valid times of day
    - Accurate to the second
    - In the range 00:00:00 - 23:59:59
Example (2)

- **Class TIME_OF_DAY**
  - **Queries**
    - hour: INTEGER
    - minute: INTEGER
    - second: INTEGER
    - is_before (other: TIME_OF_DAY): BOOLEAN
  - **Commands**
    - set_hour (h: INTEGER)
    - set_minute (m: INTEGER)
    - set_second (s: INTEGER)
Example (3)

- Decision: How to represent the time of day in internal state within instances of TIME_OF_DAY

  1. Keep three integer attributes:
     1. **hour**: INTEGER
     2. **minute**: INTEGER
     3. **second**: INTEGER

  2. Keep one integer attribute:
     1. **seconds_since_midnight**: INTEGER
Example (4)

- Decision: How to represent the time of day in internal state within instances of `TIME_OF_DAY`

  1. Keep three integer attributes:
     1. `hour`: INTEGER
     2. `minute`: INTEGER
     3. `second`: INTEGER

  2. Keep one integer attribute:
     1. `seconds_since_midnight`: INTEGER
Example (5)

**Implementation in TIME_OF_DAY:**

```plaintext
set_hour (h: INTEGER) is
  -- Set the hour from `h'
  do
    hour := h
  end
```

**Client code in some other class:**

```plaintext
coffee_time: TIME_OF_DAY
  .
  .
  coffee_time.set_hour (10)
```
Example (6)

- For any routine:
  
  ```
  set_hour (h: INTEGER)
  
  • State the conditions that must be true before the routine can work correctly
    
    0 <= h and h <= 23
  
  • State the conditions that will be true after execution, if the routine has worked correctly
    
    hour = h
  ```
Example (7)

```plaintext
set_hour (h: INTEGER) is
   -- Set the hour from `h'
   require
       valid_h: 0 <= h and h <= 23
   do
       hour := h
   ensure
       hour_set: hour = h
       minute_unchanged: minute = old minute
       second_unchanged: second = old second
   end
```
Benefits

• Documentation
  • automatic documentation of the contract

• Reusability
  • well-defined contracts

• Correctness
  • pre-/post-conditions, class invariants

• Easier software development
What is the relation..?

- Design by contract
- Software architecture
- Design intent
Alloy Annotation Language

• Invariants
• Specifications
  – Preconditions (requires)
  – Postconditions (ensures)
• Method Behavior (does)
  – built from specification
  – built from code
Alloy

• First-order, declarative language
• Based on sets and relations
• Checks assertions within a set scope
Static Checking

Invariants

• The equals() method of the Java Object class
package java.lang;
Class Object {
   /** The “equals” method implements an equivalence relation:
        *) It is reflexive: for any reference value “o”,
            “o.equals(o)” should return true.
        *) It is symmetric: …
        *) It is transitive: …
        
        …
   */
   boolean equals(Object o) {
      return (this == 0);
   }
}
AAL spec of equals

package java.lang;
Class Object {
//@ invariant {
//@   // reflexive
//@    all o: Object - null | o.equals(o)
//@   // symmetric
//@    all o, o': Object - null | o.equals(o') => o'.equals(o)
//@   // transitive
//@    all o1, o2, o3: Object - null |
//@      o1.equals(o2) && o2.equals(o3) => o1.equals(o3)
//@ }
boolean equals(Object o) {
   return (this == o);
}
}
Overriding equals

Package java.awt;
class Dimension {
    int width, height;
    
    //@ does {
    //@     \result = (obj instanceof Dimension
    //@         && this.width = obj.width
    //@         && this.height = obj.height)
    //@ }
    boolean equals(Object o) {
        if (!(o instanceof Dimension))
            return false;
        Dimension d = (Dimension)o;
        return (width == d.width) &&
            (height == d.height);
    }
}

class Dimension3D extends Dimension {
    int depth;
    
    //@ does {
    //@     \result = (obj instanceof Dimension3D
    //@         super.equals(obj) &&
    //@         this.depth = obj.depth)
    //@ }
    boolean equals(Object o) {
        if (!(o instanceof Dimension3D))
            return false;
        Dimension3D d = (Dimension3D)o;
        return super.equals(o) &&
            (depth == d.depth);
    }
}
Counterexample

O1: Dimension \{width = 0, height = 1\}
O2: Dimension3D \{width = 0, height = 1, depth = 3\}
Symmetry violated: o1.equals(o2) and not o2.equals(o1)

Package java.awt;
class Dimension {
  int width, height;

  //@ does {
  //@   \result = (obj instanceof Dimension
  //@     && this.width = obj.width
  //@     && this.height = obj.height)
  //@ }
  boolean equals(Object o) {
    ...
  }
}

package java.awt;
class Dimension3D extends Dimension {
  int depth;

  //@ does {
  //@   \result = (obj instanceof Dimension3D
  //@     && super..equals(obj) &&
  //@     && this.depth = obj.depth)
  //@ }
  boolean equals(Object o) {
    ...
  }
}
Possible Fixes

- Disable subclassing
  ```java
  final class Dimension {
    ...
  }
  ```

- Check concrete class
  ```java
  boolean Dimension.equals(Object o) {
    if (!(o.getClass() == this.getClass()))
      return false;
    ...
  }
  ```
Method Behavior

• annotations can come from one of two sources
  – Specification
  – Code translation

• Therefore, we can check both the specification and the code against invariants
Code Conformance

- Static Checking
- Dynamic Checking
Static Code Conformance

all s, s’: State | 
  valid(s) && pre(s) && body(s, s’) 
  => valid(s’) && post(s, s’)
Dynamic Code Conformance

• Unit testing
  – Alloy generates test inputs using invariants and preconditions
  – Execute each input
  – Checks output against postcondition

• Runtime checking