אוניברסיטת חיפה החוג למדעי המחשב

Department of Computer Science University of Haifa

Programming Languages Seminar

Lecture in the subject of

Abstraction and Specification

Yair Moshe

Partly based on the book:
Program Development in Java – Abstraction, Specification and Object-Oriented Design / Barbara Liskov & John Guttag
And also partly based on the course:
Object Oriented Programming and Design, Technion – IIT, Gabi Zodik & Yair Moshe

Apr. 6th, 2005

Outline

- Why software engineering is hard?
- Reducing Complexity
- Abstraction by Specification
- Rep. Invariant & Rep. Exposure
- Abstraction Function
- Subtyping and Specification
- Conclusion

The Problem – Development Failures

- **IBM survey, 1994**
  - 55% of systems cost more than expected
  - 68% overran schedules
  - 88% had to be substantially redesigned
- **Bureau of Labor Statistics, 1997**
  - For every 3 new systems put into operation, 1 cancelled
  - Probability of cancellation is about 50% for biggest systems
  - 75% of the systems are regarded as "operating failures"
- **Why?**

Because building good software is hard!

Why Software Engineering is Hard?

- **Large software systems are enormously complex**
  - Millions of "moving parts"
- **Software never dies**
  - Lots of legacy software, causes an integration nightmare
- **People expect software to be malleable**
  - After all, it's "only software"
- **We are always trying to do new things with software**
  - Relevant experience often missing
- **Software engineering is like all engineering but is different**

Software Engineering is Different

- **Little separation between design and fabrication**
  - Radical design changes during implementation
- **Ill-defined goals**
  - Enormous pressure for features
- **Tight and rapidly changing schedules**
  - Hard to anticipate needs
- **Variety of applications**
  - Banking to games
- **Huge design space**
  - Physics rarely intrudes
- **Need for flexibility**

Size is a Major Issue

- **Some industry studies suggest effort = length$^{1.5}$**
- **Adding people adds problems**
  - Nobody understands the whole system
  - Opportunities for misunderstandings
  - Management overhead

Let's Summarize the Problem

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

Rich Cook

Solution

- **No magic bullet** 
- **Development processes (e.g., RUP, XP)**
  - Tools, best practices, etc.
- **Documentation can ameliorate difficulties**
  - Undocumented software of no commercial value
- **Must reduce complexity of software**
  - Goal: Make difficulty linear with respect to size

Outline

- **Why software engineering is hard?**
- **Reducing Complexity**
- **Abstraction by Specification**
- **Rep. Invariant & Rep. Exposure**
- **Abstraction Function**
- **Subtyping and Specification**
- **Conclusion**

Reducing and Ordering Complexity

- **Divide and rule is the key**
- **We do this by**
  - Decomposition / Decoupling
  - Abstraction
- **Decomposition creates structure**
  - What kind of structure is best?
- **Abstraction suppresses details**
  - Trick is to suppress the right details

Abstraction and Specification

- **Decomposition is used to break software into components that can be combined to solve the original problem**
- **Abstractions assist in making a good choice of components**
- **Kinds of abstraction**
  – Abstraction by parameterization
  – Abstraction by specification
  – Data Abstraction (ADT – Abstract Data Types)

Abstraction by Parameterization

$$3x+2y$$

- **Hide the identity of the data by replacing it with parameters**
- **We use abstraction by parameterization in the definition of procedures/methods**
- **Generalizes the procedure so it can be used in more situations**
  – This is a good reason not to hardwire things (e.g., file names, numbers)
  – Also a good reason for not using global variables

Outline

- Why software engineering is hard?
- Reducing Complexity
- Abstraction by Specification
- Rep. Invariant & Rep. Exposure
- Abstraction Function
- Subtyping and Specification
- Conclusion

Abstraction by Specification

- For the client, hides the implementations details
  - Point of view: What the module does
- For the implementer, hides the way of use
  - Point of view : How the module is implemented
- A contract between client and implementer

Specification

- **Describing required behavior**
  - Not means of achieving it
- **Specification denotes a (usually infinite) set of programs**
  - E.g., Set of all procedures that sort lists
- **It is essential that abstractions be given precise definitions**
  - "One person's feature is another person's bug"

Procedure Specification

qualifiers return_type procedure_name(…)

requires: States any constraints on use; the constraints under which the abstraction is defined (precondition)

modifies: Lists of all modified states by the procedure

effects: Describes the behavior for all inputs not ruled out by the requires clause (postcondition). Says nothing about the procedure's behavior when the requires clause is not satisfied

Procedure Specification Examples

```
/**
 * @requires: i>0
 * @modifies: nothing
 * @effects: returns true if i is a prime number and false otherwise
**/
public static boolean isPrime(int i) {
    …
}
```

```
/**
 * @requires: value ∈ arr
 * @modifies: nothing
 * @effects: returns an i such that arr[i]==value
**/
public int find(int[] arr, int value) {
    …
}
```

- **What happens if precondition doesn't hold?**

Total vs. Partial Procedures

- **When a precondition doesn't hold, the behavior is completely unconstrained**
  – throw an exception, crash, loop forever, play a nice tune…
- **A procedure is total if its behavior is specified for all legal inputs; otherwise it is partial**
- **Should we write partial procedures?**
  + Easier to implement
  - Not as safe as total ones
  - Not well-defined everywhere
  - A weaker specification (we'll talk about this in a minute)
  - Abstraction is less general

Ideally Not

Strength of Specification

- **A stronger specification**
  - Asking less of the client
    - Has weaker preconditions
    - Making requires easier to satisfy
  - Promises more
    - Has stronger postcondition
    - Making effects harder to satisfy and/or fewer objects in modifies clause
- **If specification S_1 is weaker than specification S_2, then for every implementation I:**

  I satisfies $S_2 \rightarrow$ I satisfies S_1

Writing a Good Specification

- **A specification should have the following properties**
  - Restrictiveness
    - Strong enough
    - Make guarantees to something useful
  - Generality
    - Weak enough
    - Don’t deny legitimate implementations
  - Clarity
    - Informative
    - Coherent

Another Specification Example

- **From java.util.Vector**

```
/**
 * Replaces the element at the specified position in this Vector with the
 * specified element.
 * @modifies: this[index]
 * @effects: this_post[index] = element
 * @return: this_pre[index]
 * @throws IndexOutOfBoundsException if (index<0 || index≥size())
**/
public Object set(int index, Object element) {
    …
}
```


Outline

- **Why software engineering is hard?**
- **Reducing Complexity**
- **Abstraction by Specification**
- **Rep. Invariant & Rep. Exposure**
- **Abstraction Function**
- **Subtyping and Specification**
- **Conclusion**

Abstract Date Type

- **A data abstraction is defined by a specification**
  - A collection of procedural abstractions
- **Together, these procedural abstractions provide**
  - A set of values
  - All ways of using that set of values
- **To implement an ADT one should**
  - Select representation of instance variables - The Rep.
  - Implement operations in terms of that rep

Abstract Date Type

- **We choose the rep so that**
  - It is possible (and preferably easy) to implement operations
  - Most frequently used operations are efficient
- **Abstraction allows to change the rep in a later time**

Rep. Invariant & Abstraction Func.

- **The rep is a data structure + a set of conventions**
- **Conventions are defined by the Rep. Invariant**
  - Defines set of reachable values of the data structure
  - A set of values that is the subset of rep values that are well formed
- **Abstraction function**
  - Defines how the data structure is to be interpreted
  - i.e., how to get from the values of the rep fields to the values of the spec fields

ADT Example

```
import java.util.ArrayList;

// CharSets are finite sets of chars
public class CharSet {

  private ArrayList elements;

  // @effects: Creates an empty CharSet
  public CharSet( ) {
    elements = new ArrayList();
  }

  // @modifies: this
  // @effects: this_post = this_pre U {c}
  public void insert(char c) {
    Character character =
      new Character(c);
    elements.add(character);
  }

  // @modifies: this
  // @effects: this_post = this_pre - {c}
  public void delete(char c) {
    elements.remove(elements.indexOf(
      new Character(c)));
  }

  // @return: (c∈this)
  public boolean member(char c) {
    return elements.contains(
      new Character(c));
  }

  // @return: cardinality of this
  public int size( ) {
    return elements.size();
  }
}
```

ADT Example – Where is the Error?

- **This is an important question**
  - Tells us what needs to be fixed
- **Perhaps delete() is wrong**
  - It should remove all occurrences
- **Perhaps insert() is wrong**
  - It should not insert a char that is already in the CharSet
- **We have no way of knowing**
  - Or do we?

```
CharSet set = new CharSet();
set.insert('a');
set.insert('a');
set.delete('a');
if (set.member('a'))
    // print wrong
else
    // print right
```

This is what rep. invariant is all about

ADT Example – Rep. Invariant

- **Let's write a rep. invariant to CharSet**

  // elements has only instances of Character and no duplicates
- **And if you insist on formality**

  // ∀element e of elements, e instanceof(Character) &

  // ∀indices i,j of elements

  // elements.elementsAt(i).equals(elements.elementAt(j)) → i=j
- **Now, who's faults is it?**
  - insert()
- **We can prove correctness by showing that every operation preserves the rep. invariant**
  - Proof by induction
  - This is not always valid – let's see a counter-example

Rep. Exposure

- **Consider adding the following abstraction and implementation to CharSet**

```
// @return: An ArrayList containing the elements of this
public ArrayList getElements() {
        return elements;
}
```

- **And consider the following code**

```
CharSet set = new CharSet();
set.insert('a');
ArrayList e = set.getElements;
e.add('a');
set.delete('a');
if (set.member('a')) …
```


Rep. Exposure

- **What we have just seen is rep. exposure**
- **This is almost always evil**
- **It's not against the law**
  - But it ought to be

Avoiding Rep. Exposure

- **Exploit immutability**

```
public Character getElement(int i) {
        return (Character)elements.elementAt(i);
}
```

  - This is safe since Character is immutable
- **Make a copy**

```
public ArrayList getElements() {
        return (ArrayList)elements.clone();
}
```

  - This is safe since changes in copy will not affect the original

Checking Rep. Invariant

- **Should code check that rep. invariant holds?**
  - Yes, if inexpensive
  - Yes, as debugging code (even if expensive)
- **Rep Invariant should be checked**
  - At the start of every public method
  - At every possible end of every public method
  - At every possible end of every constructor

Checking Rep. Invariant - Example

```
public void delete(char c) {
    checkRep();
    elements.remove(elements.indexOf(new Character(c)));
    checkRep();
}

private void checkRep() throws RuntimeException {
    for (int i = 0; i < elements.size(); i++) {
        if (!elements.elementAt(i) instanceof Character)
            throw new RuntimeException(
                "An element of CharSet is not an instance of Character");
        if (elements.lastIndexOf(elements.elementAt(i)) != i)
            throw new RuntimeException(
                "Duplicate elements in CharSet");
    }
}
```


Rep. Invariant Implications

- **Makes modular reasoning possible**
  - To check whether an operation is implemented correctly, we don't need to look at any other methods. Instead, we appeal to the principle of induction.
    - As long as the representation is not exposed
- **Checking the rep invariant using checkRep() helps to discover errors**
- **One should design and record the rep invariant as part of the design of the representation, before he starts coding**
  - When trying to implementing an abstract data type, writing down the rep invariant is a good place to start

Outline

- **Why software engineering is hard?**
- **Reducing Complexity**
- **Abstraction by Specification**
- **Rep. Invariant & Rep. Exposure**
- **Abstraction Function**
- **Subtyping and Specification**
- **Conclusion**

Fixed (?) ADT Example

```
import java.util.ArrayList;

// CharSets are finite sets of chars
public class CharSet {

  private ArrayList elements;

  // @effects: Creates an empty CharSet
  public CharSet( ) {
    elements = new ArrayList( );
  }

  // @modifies: this
  // @effects: this_post = this_pre U {c}
  public void insert(char c) {
    Character character =
      new Character(encrypt(c));
    if (!elements.contains(character))
      elements.add(character);
  }

  // @modifies: this
  // @effects: this_post = this_pre - {c}
  public void delete(char c) {
    elements.remove(elements.indexOf(
      new Character(c)));
  }

  // @return: (c∈this)
  public boolean member(char c) {
    return elements.contains(
      new Character(c));
  }

  // @return: cardinality of this
  public int size( ) {
    return elements.size();
  }
}
```

ADT Example – Where is the Error?

- **Program still does the wrong thing**
- **Now who's fault is it?**
- **We have no way of knowing**
  - Or do we?

```
CharSet set = new CharSet();
set.insert('a');
if (set.member('a'))
    // print wrong
else
    // print right
```

This is what abstraction function is all about

Abstraction Function

- **Abstraction Function relates the concrete representation to the abstract value it represents**
- **Let's write abstraction function for CharSet:**

  `// AF(CharSet this) = { c | c is contained in this.elements }`
- **Once again we can safely place the blame**
  - insert() is violating the abstraction function
- **And what if we change the abstraction function to:**

  `// AF(CharSet this) = { c | decrypt(c) is contained in this.elements }`

Abstraction Function

- **Abstraction function needs to be defined properly on all representations that satisfy the rep invariant**
- **Valuable for debugging**
- **While writing rep invariant is usually easy, writing abstraction function is often a challenge**
  - Problem lies in denoting the range of abstraction function
- **The abstraction function and the specification go together, since they link the code to the abstract view of the type seen by the client**
  - The rep invariant, in contrast, can be used without any reference to the specification

Rep. Invariant & Abstraction Function

- **Rep invariant**
  - Which legal concrete values represent abstract values
  - Use induction to show that is indeed an invariant
- **Abstraction function**
  - Which abstract value each concrete value represents
- **Together allow us to examine methods independently**
  - Correctness becomes local issue
- **In practice**
  - Rep invariant is almost always worth writing
  - Abstraction function is harder to write

Outline

- **Why software engineering is hard?**
- **Reducing Complexity**
- **Abstraction by Specification**
- **Rep. Invariant & Rep. Exposure**
- **Abstraction Function**
- **Subtyping and Specification**
- **Conclusion**

Subtyping

- **Sometimes every A is a B**
  - Example: In library database, every book and CD is a library holding
- **Subtyping expresses this in the program**
  - Programmer declares A is a subtype of B
  - Meaning “every object that satisfies interface A also satisfies interface B“
  - In Java, using the extends or implements keywords
- **Goal: code written using B's specification operates correctly even if given an A**

Subtyping Non-Example

- **Does “A extends B” always imply “A is a subtype of B”?**

```
class GComponent {

  // @effects: paints this on the screen
  public void draw() { … }
  …
}

class GColorComponents extends GComponent {

  // @requires: the color of this has been set
  // @effects: paints this on the screen
  public void draw() { … }
  …
}
```


Subtyping Non-Example

- **Scenario #1**
  - Application doesn't set color before calling draw()
  - Draw fails
- **Scenario #2**
  - Application sets color before drawing to screen
  - Rationale for this design is not visible in code
  - Can forget the hidden dependency when modifying
- **Problem is**
  - GComponent.draw() specification not sufficient to use GColorComponent.draw() safely
  - GColorComponent is not a GComponent

Another Subtyping Non-Example

- **Elementary school: every square is a rectangle**

```
class Rectangle extends GComponent {
  // @effects: this_post.width = w, this_post.height = h
  void setSize(int w, int h) { ... }
}

class Square extends Rectangle { ... }
```

- **Let's choose the best spec for Square.setSize()**

```
// @requires: w = h
// @effects: this_post.width = w, this_post.height = h
void setSize(int w, int h) { ... }

// @effects: this_post.width = edgeLength, this_post.height = edgeLength
void setSize(int edgeLength) { ... }

// @effects: this_post.width = w, this_post.height = h
// @throws: BadSizeException if w != h
void setSize(int w, int h) throws BadSizeException { ... }
```

All are wrong

Liskov Substitution Principle

- **"If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T"**
- **The subtype must have the same or stronger specification as the supertype, so that the subtype can be called in all states where the supertype could be correctly called**

Inheritance

- **Inheritance should be viewed as subcontracting**
- **A subcontractor must promise to carry out the original contract, and possibly more**
- **In pratice, subtype methods must be substitutable for supertype methods**
  - No additional exceptions
  - No more requires
  - No more modifies
- **This occasionally violates our intuitions**

Outline

- **Why software engineering is hard?**
- **Reducing Complexity**
- **Abstraction by Specification**
- **Rep. Invariant & Rep. Exposure**
- **Abstraction Function**
- **Subtyping and Specification**
- **Conclusion**

Conclusion

- **Decomposition and abstraction are the key to reduce software complexity**
- **Specification gives abstraction precise definitions**
- **Rep invariant and abstraction function allow us to examine methods correctness independently**
- **Liskov substitution principle - A subtype must promise to carry out the original supertype's contract, and possibly more**