Taming Software Complexity (2)

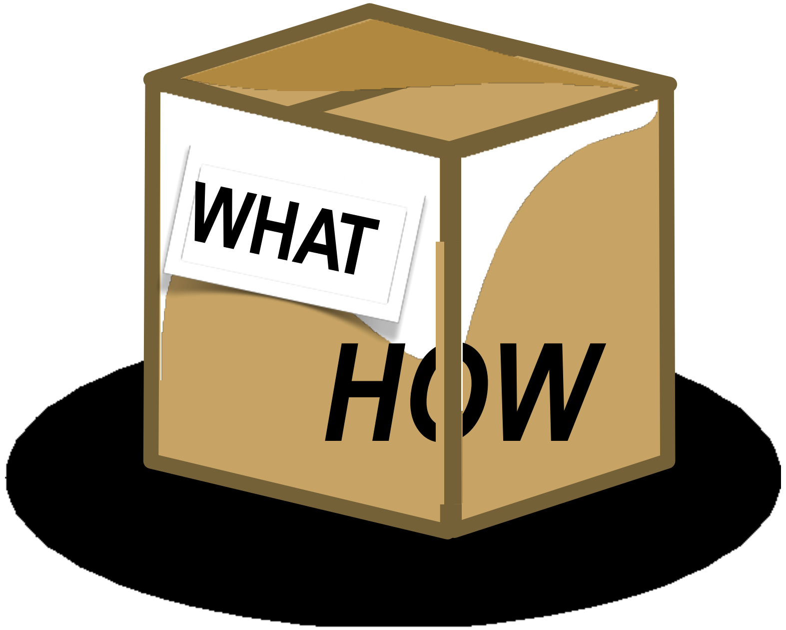
September 28, 2017

Byung-Gon Chun

(Slide credits: George Candea, EPFL)

Other Examples of Abstractions in Programming Languages

Abstraction



- Abstraction
  - *specifies "what" a component/subsystem does together*
  - *with modularity, it separates "what" from "how"*

Other Examples of Abstractions in Programming Languages

- Function, procedure, routine, method
- Thread
- Lambda function
- Abstract data type

Function, Procedure, Routine, Method, Thread

- Function, procedure, routine, method:
  a portion of code within a larger program that performs a specific task and is relatively independent of the remaining code
- Thread:
  A single execution sequence that represents a separately schedulable task

Lambda Function

- Lambda functions (a.k.a. anonymous functions)
  python: lambda argument_list: expression
  E.g., lambda x, y : x + y
  - map(func, seq), filter(func, seq), reduce(func, seq)
    numbers = [1, 2, 3, 4]

    mappedNumbers = map(lambda x: x + 1, numbers)

    filteredNumbers = filter(lambda x: x % 2, mappedNumbers)

    finalNumber = reduce(lambda x, y: x + y, filteredNumbers)

Abstract Data Types

- A mathematical model for data types
- Defined in terms of possible values, possible operations on data of this type, and the behavior of these operations
- Types can be classified into mutable and immutable types
- Operations of an abstract data type
  - Creator: t* → T
  - Producer: T+, t* → T
  - Observer: T+, t* → t
  - Mutator: T+, t* → void|t|T
    T is the abstract type itself; each t is some other type.
    + marker: the type may occur one or more times in that part of the signature,
    * marker: it occurs zero or more times

Abstract Data Type Example

- **int** is Java’s primitive integer type.
  int is immutable, so it has no mutators

  – creators: the numeric literals 0, 1, 2, …
  – producers: arithmetic operators +, -, ×, ÷
  – observers: comparison operators ==, !=, <, >
  – mutators: none (it’s immutable)

Abstract Data Type Example

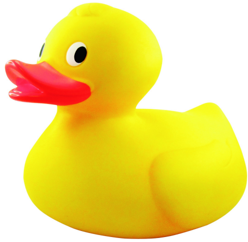
- **List** is Java’s list interface.
  List is mutable.
  List is also an interface, which means that other classes provide the actual implementation of the data type. These classes include ArrayList and LinkedList.
  - creators: ArrayList and LinkedList constructors, Collections.singletonList
  - producers: Collections.unmodifiableList
  - observers: size, get
  - mutators: add, remove, addAll, Collections.sort

Closure

- Closure is a function with an extended scope that encompasses nonglobal variables references in the body of the function but not defined there

```
def make_averager():
    series = []

    def averager(new_value)
        series.append(new_value)
        total = sum(series)
        return total / len(series)

    return averager
```

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
```

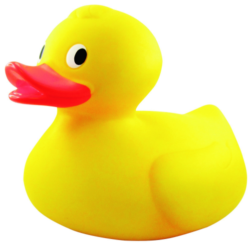


Duck Typing

```
// Java/C#

class Duck {
      public void Quack() { ... }
      public void Walk() { ... }
}
class OtherDuck {
      public void Quack() { ... }
      public void Walk() { ... }
}
...
void M(Duck bird) {
      bird.Quack();
      bird.Walk();
}
...

M(new Duck());
M(new OtherDuck());
```

Compile error

```
//Duck-typed lang

void N(Ducktyped bird) {
      bird.Quack();
      bird.Walk();
}
...
N(new Duck())
N(new OtherDuck())
```



Duck Typing

```
def calc(a,b):
   return a+b

calc(1,2)
    --> will return 1+2 = 3

calc('hello','world')
    --> will return hello world

calc([1],[2])
   --> will return [1,2]
```

Designing Good Interfaces

The 7 Guidelines for Good Interfaces

- Clean and lean
- Loosely coupled
- Portable
- Extensible
- Stratified
- Reusable
- Maintainable

1. Clean and Lean Interfaces

```
public class CleverHashMap<K,V> extends
   AbstractMap<K,V>
   implements Map<K,V>, Cloneable, Serializable
{
  // ...
   public V get(Object key) {
       /* ... */ }
   public V getFast(Object key, int hash) {
       /* ... */
   }
   public V getAndPut(Object key, K key, V value) {
       /* ... */
   }
   public V getLater(Object key, List<K> destination) {
       /* ... */
   }
   // ...
 }
```

- Compartmentalized
  - Focus on one conceptual piece in isolation
- Avoid being "too clever"
  - Avoid surprises
- Aim for minimality
  - Perfection means nothing can be removed
  - Reduce extra code that needs to be developed, reviewed, tested, debugged, …

2. Loosely Coupled Interfaces

```
public boolean promoteStudent(int studentId,
      Name name,
      StreetAddress address,
      EmailAddress email,
      /* ...
        ...
        ... */ )
```

```
public boolean promoteStudent(Student student)
```

- Minimize “bonds” to other classes
  - Reduces integration work, testing, maintenance
- Metrics for coupling?
  - Number of arguments to a public method
  - Number of public methods in interface
- Avoid implicit connections
  - Via assumptions about internal operation
  - E.g., caller initializes only those parts of an object it knows are used by the callee

Information Hiding

```java
public class Student {
   private int id;
   private static int currentMaxId=0;
   Student() {
      ++currentMaxId;
      id = currentMaxId;
   }
   public int getId() {
      return id;
   }
   // ...
}
```

```java
public class Student {
   private StudentId id;
   Student() {
      id = new StudentId();
   }
   public StudentId getId() {
      return id;
   }
   public static class StudentId {
      private static int currentMaxId=0;
      private final int id;
      StudentId() {
         ++currentMaxId;
         id = currentMaxId;
      }
      // ...
   }
}
```

- Keep secrets
  - Never reveal more than is necessary
- Hide what is likely to change
  - E.g., format of a file, data type implementation
  - Reduces strength of coupling
  - Protects you from legacy code

3. Portable Interfaces

```
String lookupInWindowsRegistry(String key);
void storeInKeychain(Certificate cert);
void updateOracleDb(ResultSet newStuff);
```

```
public class javax.swing.Box implements Serializable
```

- Avoid exposing environment
  - Operating system specificities
  - Non-portable classes
  - Classes from non-standard third party packages

4. Extensible Systems

```
public enum StatusType { OK, FAILED }
public enum StatusType { OK, FAILED, RECOVERING }

public StatusType currentStatus()
```

- Good interfaces → extensible system
  - *can extend the system without violating structure*
  - *can change a piece without affecting others*
- Segregate volatile from stable
  - *anticipate change*

5. Stratified Interface

- Use layers of abstraction
  - like the layers of an onion
  - can layer clean abstractions over ugly code
  - can wrap non-portable classes
- Java nested classes
  - Offer a convenient tool for stratification

```java
public class HashMap<K,V>
{
  // ...
  transient int size;
  // ...
  public Collection<V> values() {
    return new Values();
  }

  private final class Values extends AbstractCollection<V> {
    public Iterator<V> iterator() {
      return newValueIterator();
    }
    public int size() {
      return size;
    }
    public boolean contains(Object o) {
      return containsValue(o);
    }
    public void clear() {
      HashMap.this.clear();
    }
  }

  Iterator<V> newValueIterator()   {
    return new ValueIterator();
  }

  private final class ValueIterator
    extends HashIterator<V> {
    public V next() {
      return nextEntry().value;
    }
  }
}
```

6. Reusable Interface

```
public interface Map<K,V> {
   boolean isEmpty();
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   V get(Object key);
   V put(K key, V value);
   V remove(Object key);
   void putAll(Map<? extends K, ? extends V> m);
   void clear();
   Set<K> keySet();
   Collection<V> values();
   Set<Map.Entry<K, V>> entrySet();
   interface Entry<K,V> { K
      getKey();
      V getValue();
      V setValue(V value);
   }
 }
```

- Capture fundamental attributes corresponding to level of abstraction
  – Enables subsequent reuse
  – Generality results from focus on essence

```java
public interface Map<K,V> {
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);
    V put(K key, V value);
    V remove(Object key);
    void putAll(Map<? extends K, ? extends V> m);
    void clear();
    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet();
```

javax.script.SimpleBindings

java.security.AuthProvider

java.security.Provider
javax.swing.UIDefaults

java.awt.RenderingHints

java.util.AbstractMap<K,V>
java.util.EnumMap<K,V>
java.util.HashMap<K,V>
java.util.Hashtable<K,V>
java.util.IdentityHashMap<K,V>
java.util.LinkedHashMap<K,V>
java.util.Properties
java.util.TreeMap<K,V>
java.util.WeakHashMap<K,V>

java.util.jar.Attributes
java.util.concurrent.ConcurrentHashMap<K,V>
java.util.concurrent.ConcurrentSkipListMap<K,V>

javax.print.attribute.standard.PrinterStateReasons

javax.management.openmbean.TabularDataSupport

7. Maintainable Interface

```
public interface Map<K,V> {
   boolean isEmpty();
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   V get(Object key);
   V put(K key, V value);
   V remove(Object key);
   // ...
}
```

```
public interface BadMap<K,V> {
   boolean isEmpty();
   boolean hasWithin(Object key);
   boolean canFind(Object value);
   V retrieve(Object key);
   V put(K key, V value);
   V remove(Object key);
   // ...
}
```

- Self-explanatory design
  - Write-once/read-many
  - Anticipate questions
  - Avoid surprises
- Use hierarchy
- Assign a responsibility to each class
- Design for testability
  - Include test interfaces
- *Be paranoid*
  - *Always think how someone could misuse your interface or class*

Frequent End Result

Sir Charles Antony Richard Hoare

There are two ways of constructing a software design:

One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.

The first method is far more difficult.

Abstraction Functions and Representation Invariant

- *Abstract Value:* What an instance of a class is suppose to represent.
- *Concrete Representation:* How the abstract state of a class is represented within a Java object.
- *Representation Invariant:* A condition that must be true over all valid concrete representations of a class. The representation invariant also defines the domain of the abstraction function.
- *Abstraction Function:* A function from an object's concrete representation to the abstract value it represents.

Interface Abstraction

$abstraction = AF(representation)$
$AF: R => A$

```
interface CharSetInterface {
    void add(char c);
    void remove(char c);
    boolean isMember(char c);
}
```

Set

$Set \leftarrow Set \cup \{c\}$

$Set \leftarrow Set \setminus \{c\}$

$c \in Set \Rightarrow \text{return true} \wedge c \notin Set \Rightarrow \text{return false}$

Class Implementation

```
class CharSet implements CharSetInterface {
    private StringBuffer s;

    CharSet() {
        s = new StringBuffer();
    }
    public void add(char ch) {
        if (!isMember(ch)) {
            s.append(ch);
        }
    }
    public void remove(char ch) {
        int index = s.indexOf(String.valueOf(ch));
        if (index >= 0) {
            s.deleteCharAt(index);
        }
    }
    public boolean isMember(char ch) {
        return s.indexOf(String.valueOf(ch)) != -1;
    }
}
```

Internal Representation

```
class CharSet implements CharSetInterface {
    private StringBuffer s;

    CharSet() {
        s = new StringBuffer();
    }
    public void add(char ch) {
        if (!isMember(ch)) {
            s.append(ch);
        }
    }
    public void remove(char ch) {
        int index = s.indexOf(String.valueOf(ch));
        if (index >= 0) {
            s.deleteCharAt(index);
        }
    }
    public boolean isMember(char ch) {
        return s.indexOf(String.valueOf(ch)) != -1;
    }
}
```

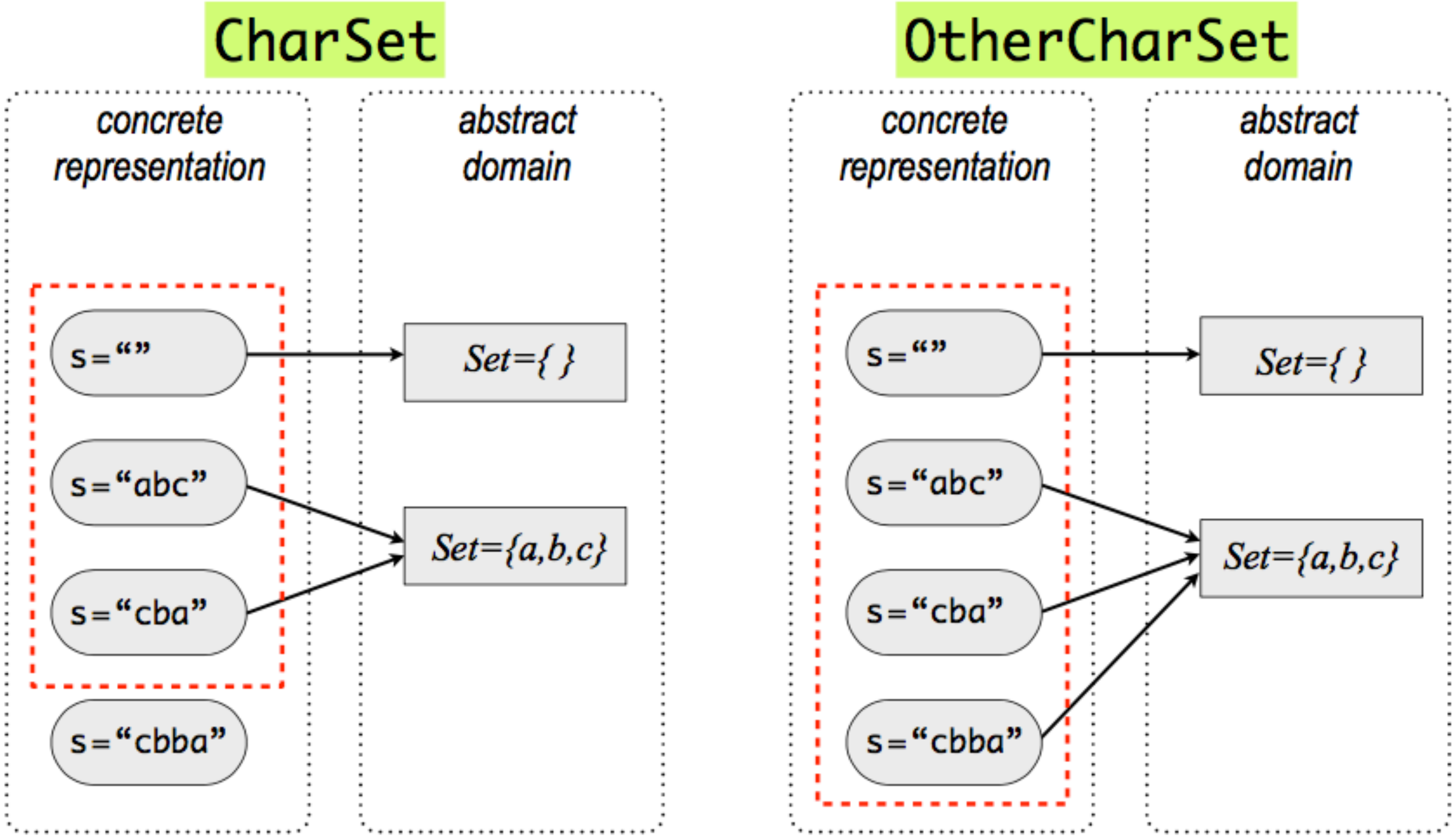
- Valid values
  - *"a", "cba", etc.*
- Is "*cbba*" a valid CharSet

Alternate Implementation

```
class CharSet implements CharSetInterface {
    private StringBuffer s;

    CharSet() {
        s = new StringBuffer();
    }
    public void add(char ch) {
        if (!isMember(ch)) {
            s.append(ch);
        }
    }
    public void remove(char ch) {
        int index = s.indexOf(String.valueOf(ch));
        if (index >= 0) {
            s.deleteCharAt(index);
        }
    }
    public boolean isMember(char ch) {
        return s.indexOf(String.valueOf(ch)) != -1;
    }
}
```

```
class OtherCharSet implements CharSetInterface {
    private StringBuffer s;

    OtherCharSet() {
        s = new StringBuffer();
    }
    public void add(char ch) {
        s.append(ch);
    }
    public void remove(char ch) {
        int index = s.indexOf(String.valueOf(ch));
        while (index >= 0) {
            s.deleteCharAt(index);
            index = s.indexOf(String.valueOf(ch));
        }
    }
    public boolean isMember(char ch) {
        return s.indexOf(String.valueOf(ch)) != -1;
    }
}
```

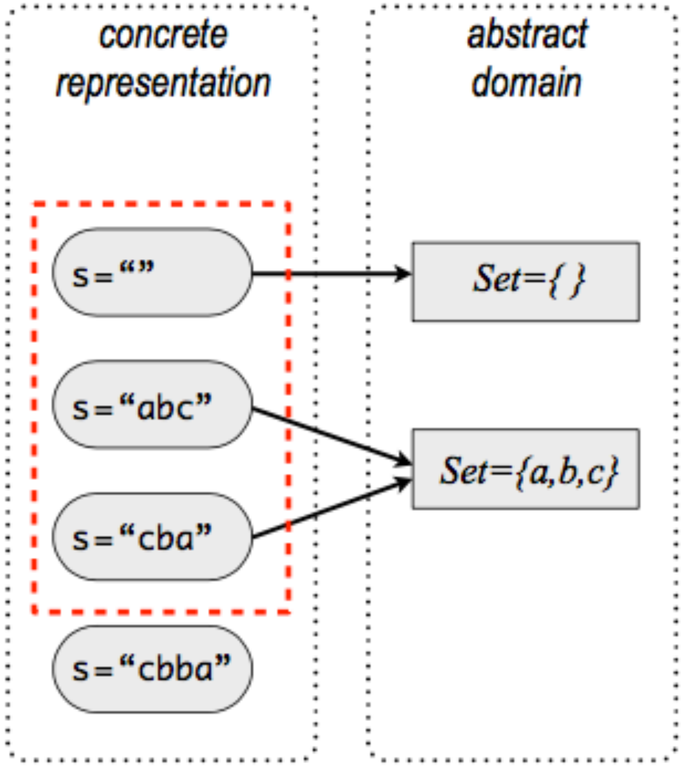
Abstraction Function



Rep(resentation) Invariant

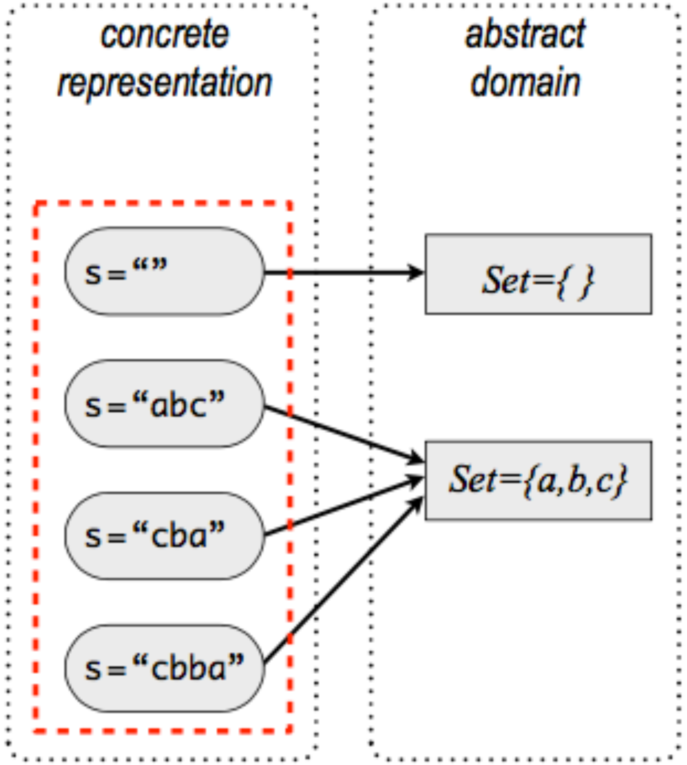
A condition that must be true over all valid concrete representations of a class. The representation invariant also defines the domain of the abstraction function.

`CharSet`



$$RI(r) : r.s \neq null \ \wedge$$
$$r.s \text{ contains no duplicates}$$

`OtherCharSet`



$$RI(r) : r.s \neq null$$

Abstraction vs. Implementation

```java
public class HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
{
    // ...
    transient Entry[] table;
    transient int size;
    int threshold;
    final float loadFactor;
    transient volatile int modCount;

    // ...
    public HashMap(int initialCapacity) {
        this(initialCapacity, DEFAULT_LOAD_FACTOR);
    }

    public V put(K key, V value) {
        // ...
    }

    // ...

    private boolean containsNullValue() {
        // ...
    }

    void addEntry(int hash, K key, V value, int bucketIndex) {
        // ...
    }

    // ...
}
```

$RI(r) = true \Rightarrow AF(r)$ *turns r into desired abstraction*

- To implement abstraction
  - Choose r and RI(r)
  - Implement AF(r)
  - Ensure RI(r) is preserved
- State machine view
  - Abstraction = state machine
  - Implementation = state machine
  - Implementation simulates abstraction

Audit Methods

Auditing the Rep(resentation)

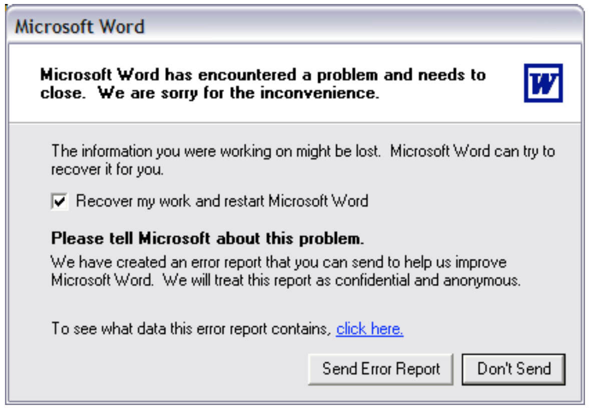
- *RI(r)=true ⇒ AF(r) turns r into desired abstraction*

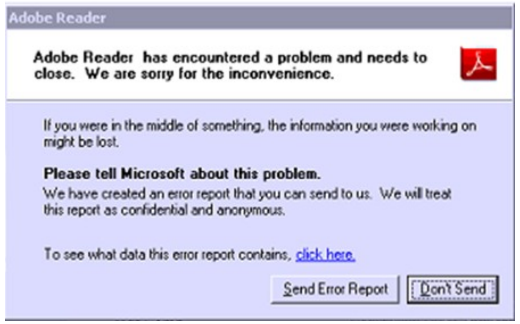
```
class OtherCharSet implements CharSetInterface {
    private StringBuffer s;

    OtherCharSet() {
        s = new StringBuffer();
    }
    public void add(char ch) {
        s.append(ch);
    }
    public void remove(char ch) {
        int index = s.indexOf(String.valueOf(ch));
        while (index >= 0) {
            s.deleteCharAt(index);
            index = s.indexOf(String.valueOf(ch));
        }
    }
    public boolean isMember(char ch) {
        return s.indexOf(String.valueOf(ch)) != -1;
    }
}
```

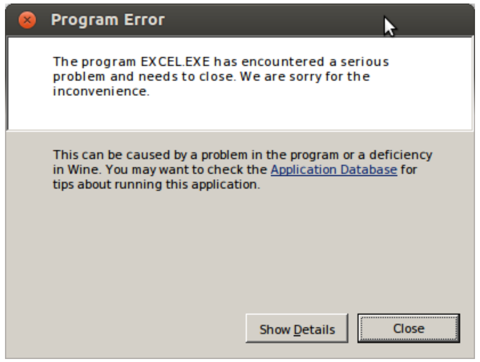
RI(r)=true

- Does rep satisfy the rep invariant?
  - i.e., can the abstraction function be applied to it?
  - should have audit method for every non-trivial object and subsystem
- Connects to pre- and post-conditions
  - rep invariant must hold both upon entry and exit

Audits in the Real World







Audits Using Assertions

```
assert invariant : details
```

$RI(r) : r.s \neq null$

```
class OtherCharSet implements CharSetInterface {
   private StringBuffer s;
   OtherCharSet() {
      s = new StringBuffer();
      assert s!=null : "Crazy! A just-allocated string is nu
   }
   public void add(char ch) {
      assert s!=null : "add: must have set s to null somewhe
      s.append(ch);
      assert s!=null : "Crazy! Just dereferenced s yet now i
   }
   public void remove(char ch) {
      assert s!=null : "remove: must have set s to null some
      int index = s.indexOf(String.valueOf(ch));
      while (index >= 0) {
         s.deleteCharAt(index);
               index = s.indexOf(String.valueOf(ch));
      }
      assert s!=null : "remove: s is null on exit";
   }
```

Audit Method for OtherCharSet

```
class OtherCharSet implements CharSetInterface {
  private StringBuffer s;

  OtherCharSet() {
    s = new StringBuffer();
  }
  public void add(char ch) {
    s.append(ch);
  }
  public void remove(char ch) {
    int index = s.indexOf(String.valueOf(ch));
    while (index >= 0) {
      s.deleteCharAt(index);
      index = s.indexOf(String.valueOf(ch));
    }
  }
  public boolean isMember(char ch) {
    return s.indexOf(String.valueOf(ch)) != -1;
  }
```

```
public int auditErrors() {
  if (s==null) {
    auditLog.error("OtherCharSet: rep
        invariant does not hold");
    return 1;
  }
  return 0;
}
```

The Audit Method

```
int auditErrors(int depth) {
  if (depth==0) {
    return 0;
  }

  int auditErrors=0;
  foreach non-elemental field subRep of the rep {
    auditErrors += subRep.auditErrors(depth-1);
  }

  foreach component of the RI(rep) {
    if component does not hold {
      auditLog.error(information on this violation);
      ++auditErrors;
    }
  }

  return auditErrors;
}
```

Recursive Auditing

```java
public class ServerCache implements ServerCommunicationFactory {

    private ConcurrentHashMap<Integer, AggregatedRatings> mAllRatings =
        new ConcurrentHashMap<Integer, AggregatedRatings>();
    private HashMap<String, RatingType> mMyRatings = new HashMap<String, RatingType>();
    private ConcurrentLinkedQueue<QuizQuestion> newQuestions = new
    ConcurrentLinkedQueue<QuizQuestion>(); private ServerCommunicationFactory mRealServer;
    private Disk mDisk;

    public int auditErrors(int depth) {
...
```

```java
public final class Disk implements IDisk {

    private static final int DEFAULT_CACHE_SIZE =
    20; private ICache<Integer, JSONObject> mCache;
    private static final Disk THE_DISK =
            new Disk(new LruCache<Integer, JSONObject>(DEFAULT_CACHE_SIZE));

    public int auditErrors(int depth) {
...
```

```java
public class LruCache<K, V> extends LinkedHashMap<K, V> {

    private final int mMaxSize;
    private final static float LOAD_FACTOR = 0.75f;
    private final static boolean ACCESS_ORDER =
    true;
    public int auditErrors(int depth) {
...
```

Uses of Auditing

```
int auditErrors(int depth) { if
    (depth==0) {
        return 0;
    }

    int auditErrors=0;
    foreach non-elemental field subRep of the rep {
        auditErrors += subRep.auditErrors(depth-1);
    }

    foreach component of the RI(rep) {
        if component does not hold {
            auditLog.error(information on this violation);
            ++auditErrors;
        }
     }

    return auditErrors;
}
```

- Testing
  - Check consistency before and after unit tests
- Development
  - Write audit method along with code
  - Maintain it, as the code changes
  - Forces you to think, simplify, streamline your structures
- Debugging
  - Invoke auditErrors() periodically during execution
  - Helps detect errors early
- **Beware of concurrency**