Taming Software Complexity

October 11, 2022

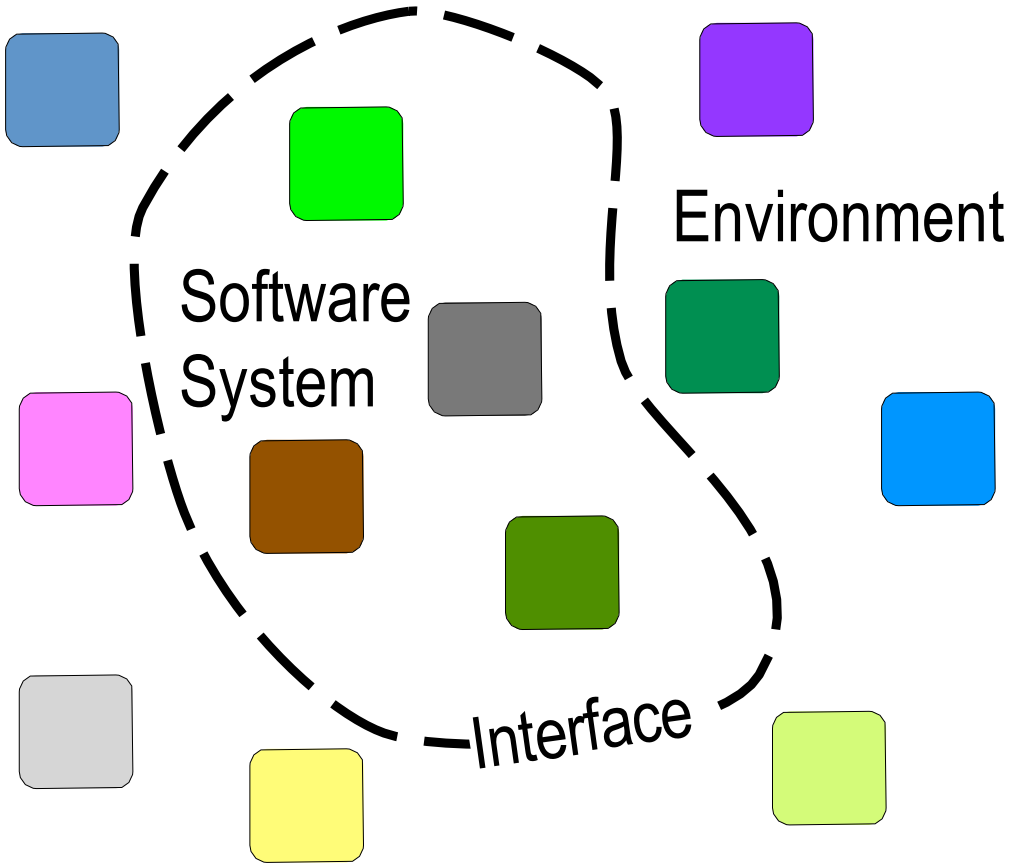
Byung-Gon Chun

(Slide credits: George Candea, EPFL)

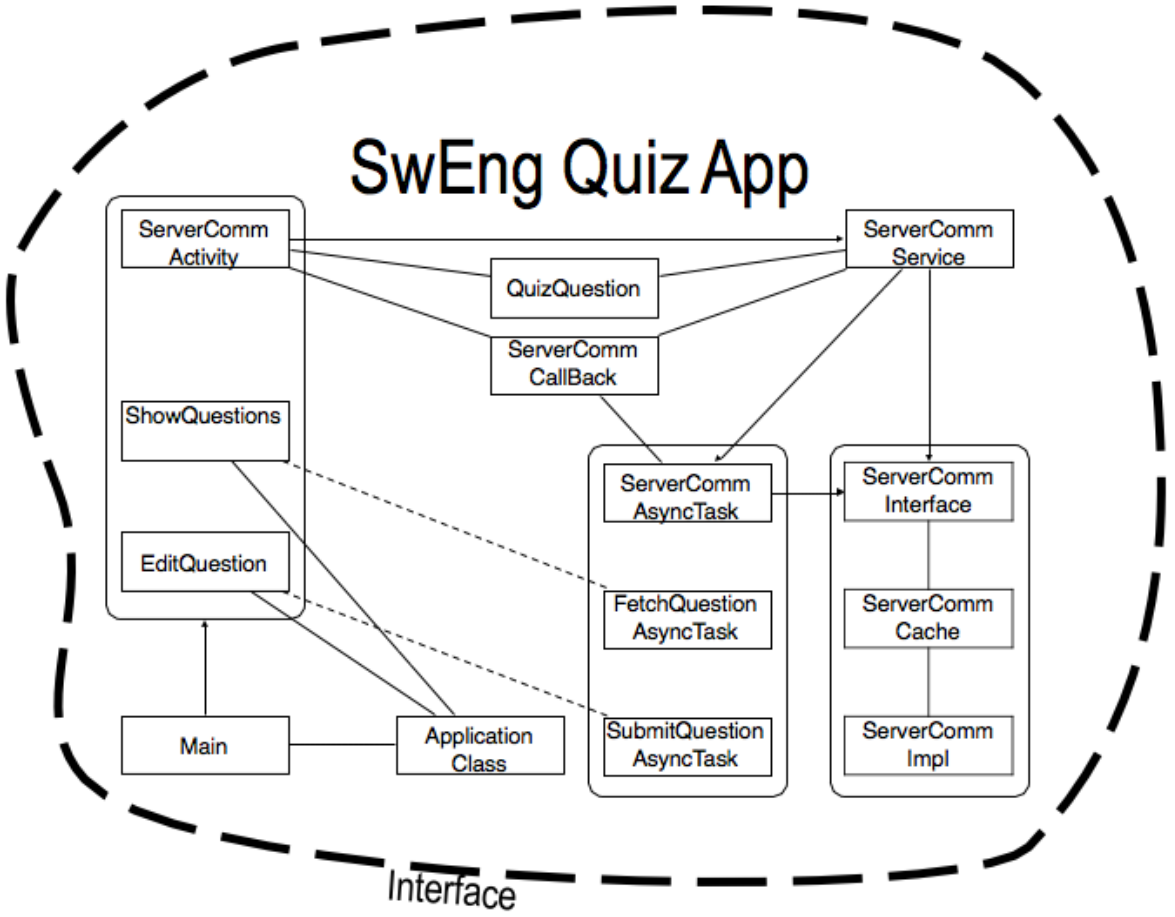
Symptoms of Complexity

What is a Software System?

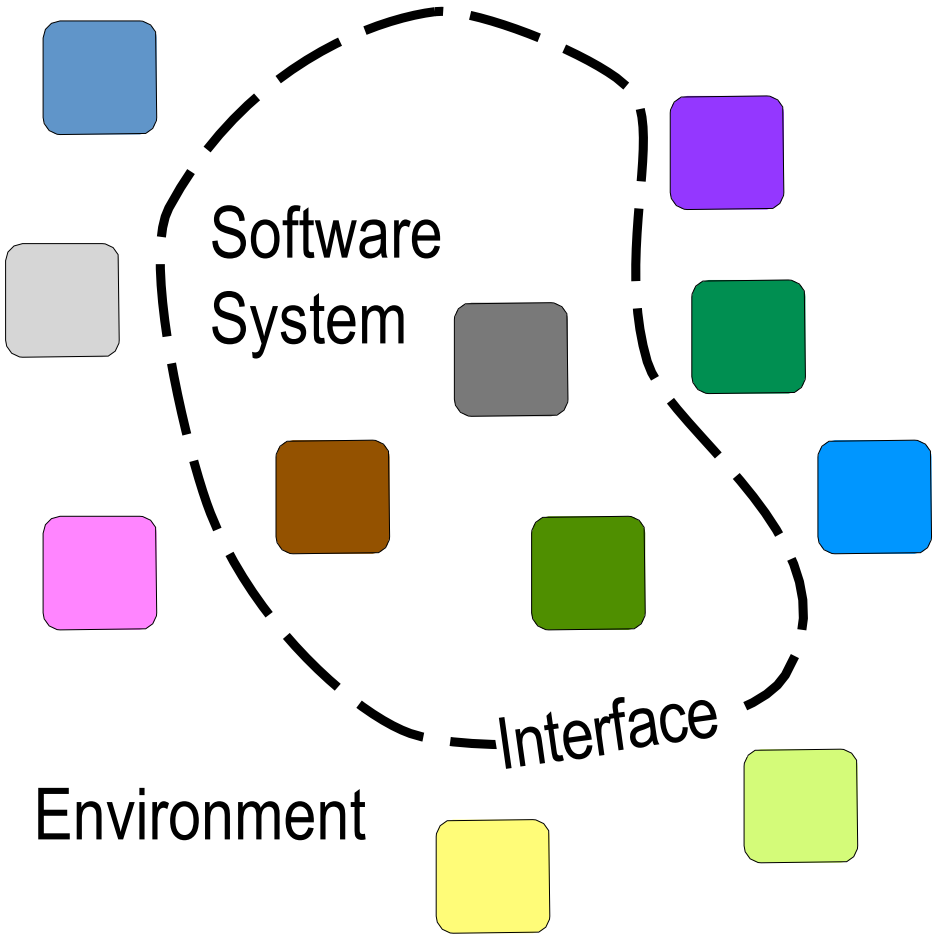
A group of interconnected components that exhibits an expected collective behavior observed at the interfaces with its environment.



Examples of Systems

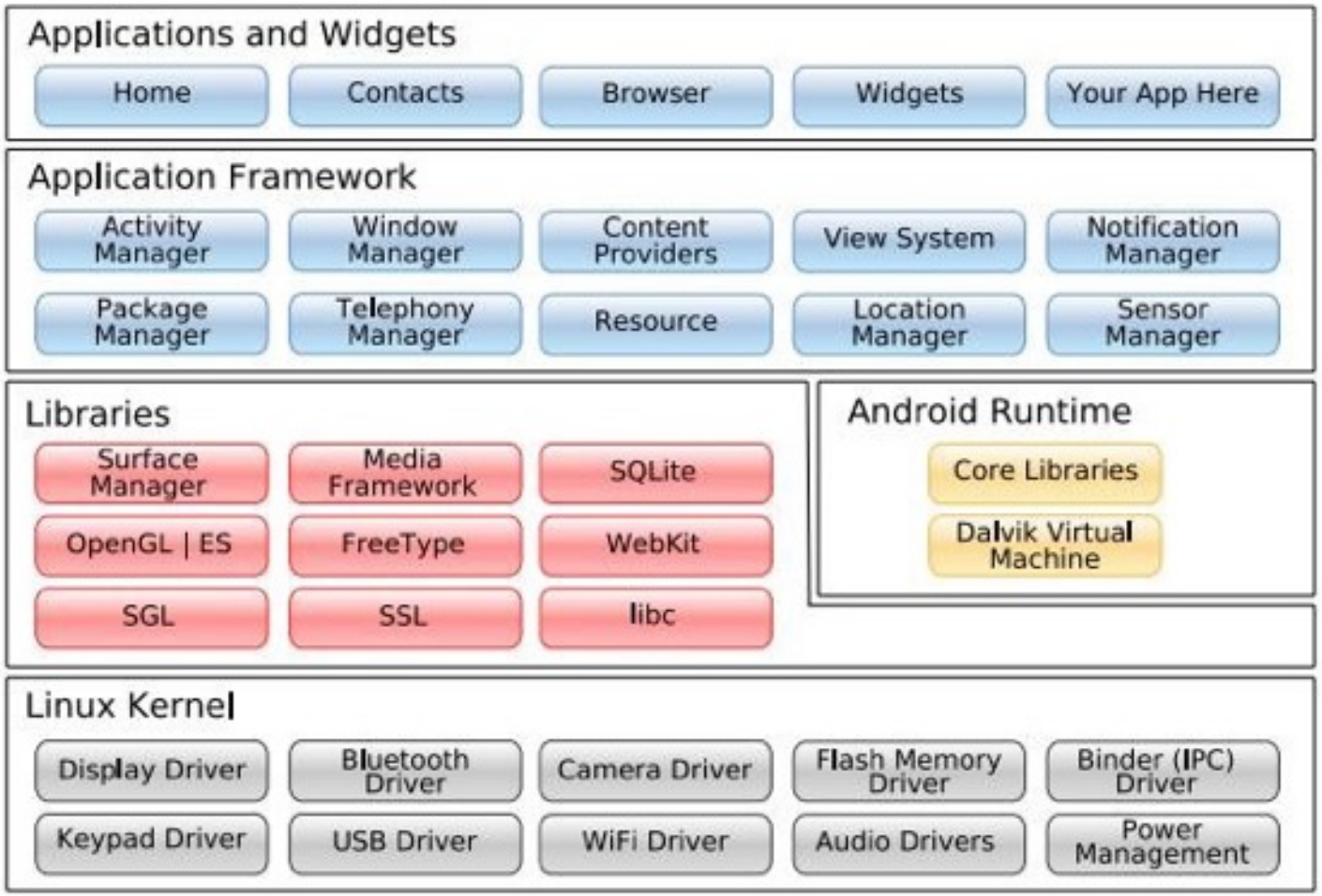


Four Symptoms of Complexity

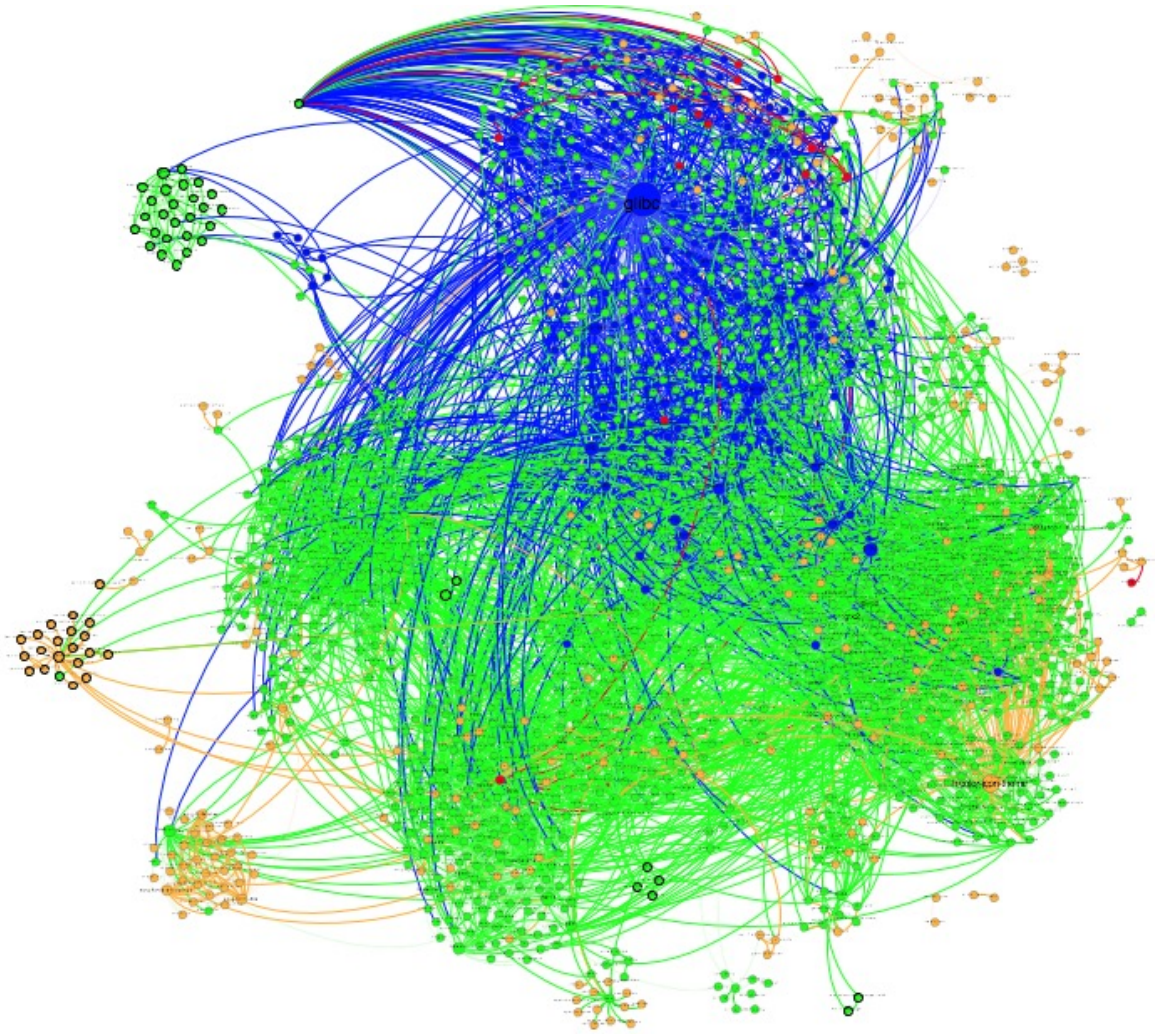


- Large number of components
- Large number of interconnections
- Many irregularities and exceptions
- High “Kolmogorov complexity”

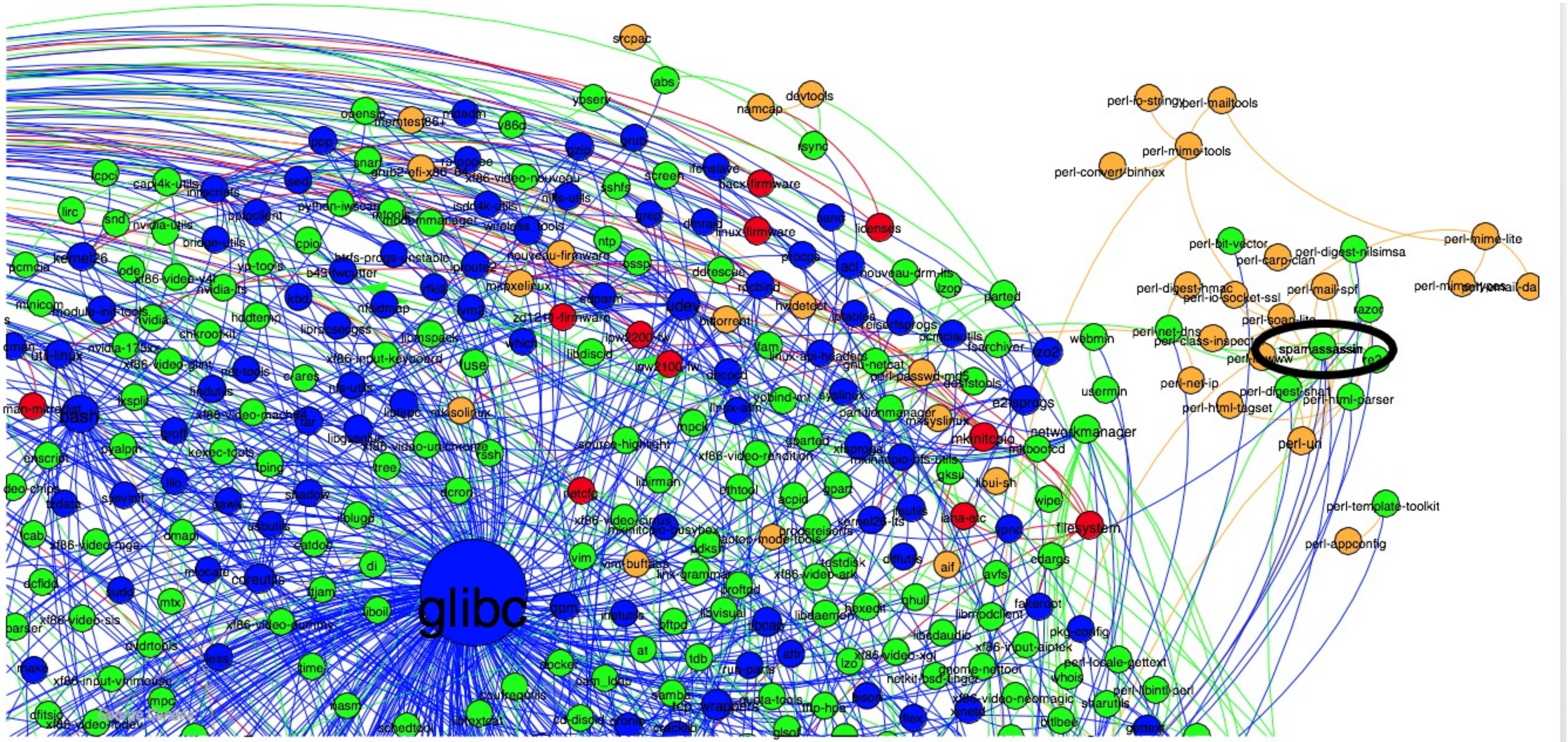
Symptom #1: Many Components



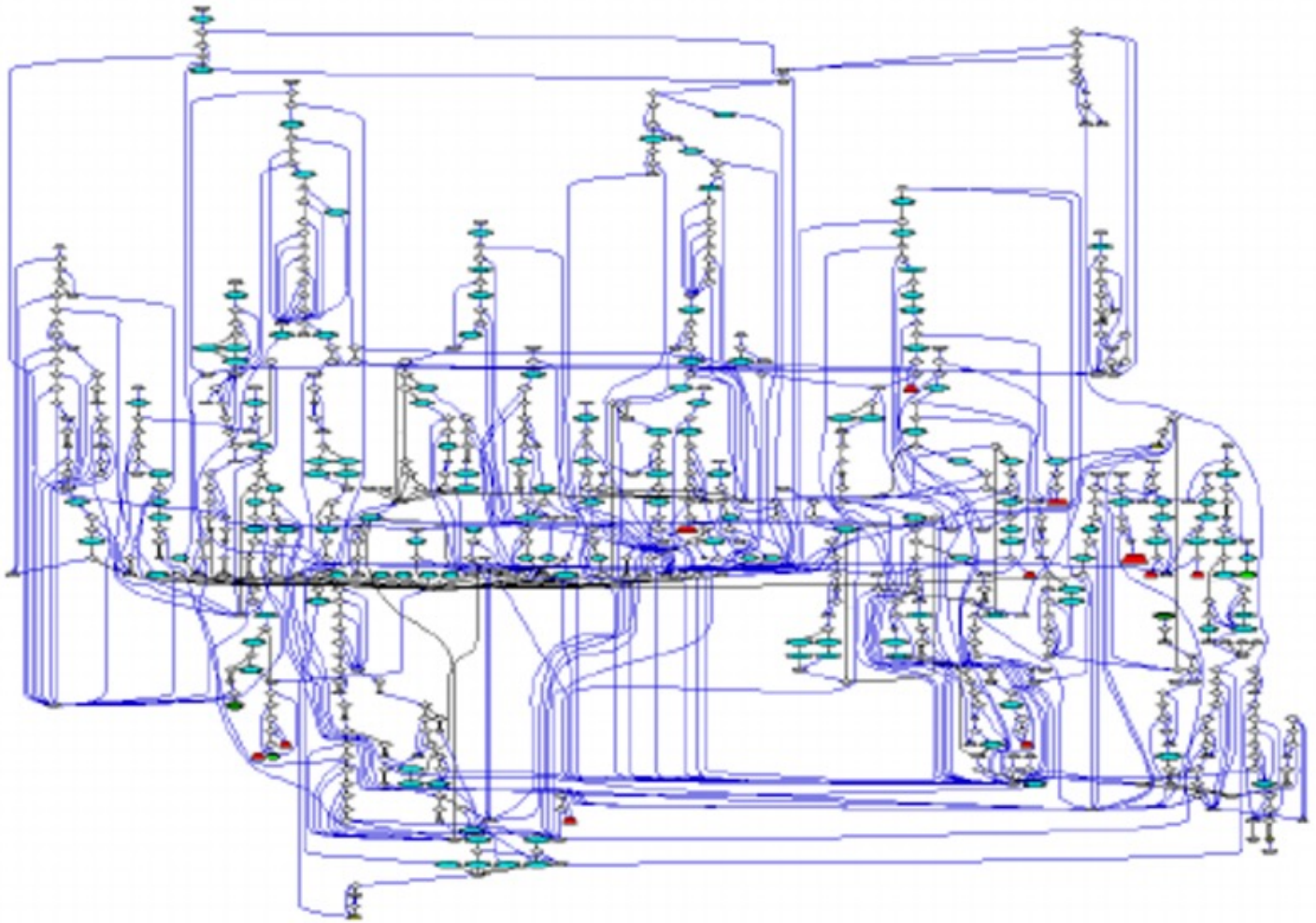
Symptom #2: Many Interconnections



Symptom #2: Many Interconnections



Symptom #3: Irregularity and Exceptions



Symptom #4: High Kolmogorov Complexity

|*AAAAAAA … AAAAB*| = 10^6+1

$\mathbf{K}$(*AAAAAAA … AAAAB*) =
|"1 million *A*s followed by 1 *B*"|

⇒ simple

|*ABDAGHDBBCAD*…| = 10^6+1

$\mathbf{K}$(*ABDAGHDBBCAD*…) = 10^6+1

⇒ complex

- Kolmogorov complexity
  - *minimal length of a description of the object*
  - *computation resources needed to specify an object*
- $\mathbf{K}$(object) >= |object| => complex
  $\mathbf{K}$(object) << |object| => simple

Complexity

- Complexity is anything related to the structure of a software system that makes it hard to understand and modify the system (John Ousterhout)

- How Does Complexity Manifest?

Symptoms of Complexity

- Make it Harder to Carry Out Development Tasks
  - Change amplification
  - Cognitive load
  - Unknown unknowns

Causes of Complexity

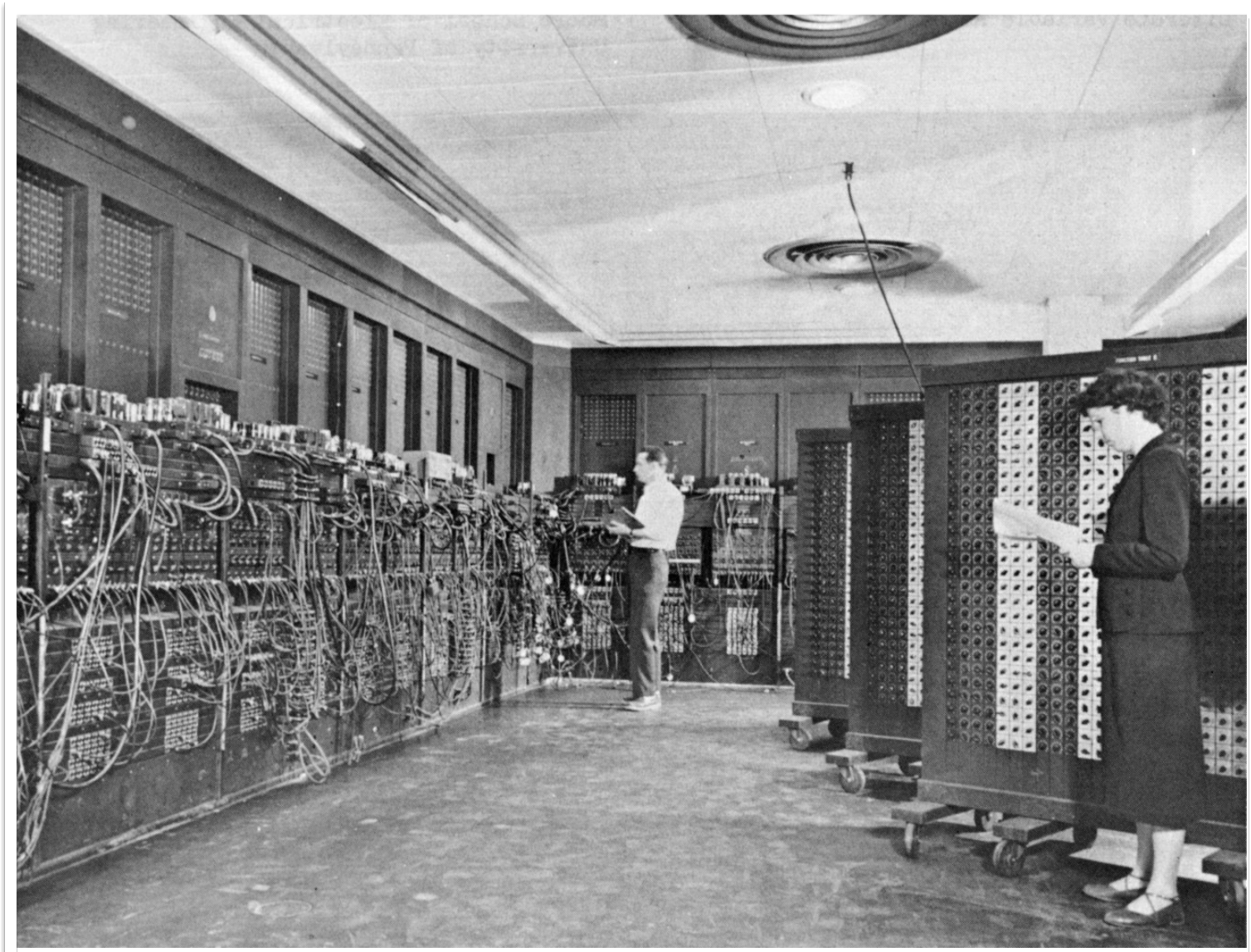
- Dependencies
- Obscurity
- Complexity is incremental

  Complexity comes about because hundreds or thousands of small dependencies and obscurities build up over time

- Complexity makes it difficult and risky to modify an existing code base

- How to tame complexity?

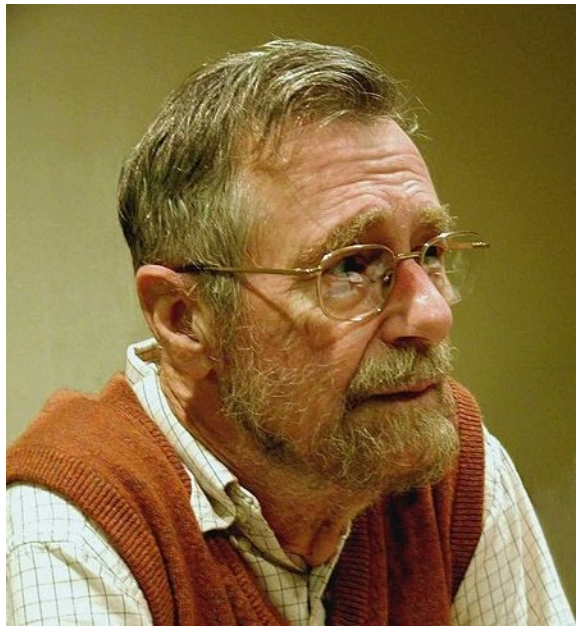
Modularity



```
MAIN0001* PROGRAM TO SOLVE THE QUADRATIC EQUATION
MAIN0002         READ 10,A,B,C $
MAIN0003         DISC = B*B-4*A*C $
MAIN0004         IF (DISC) NEGA,ZERO,POSI $
MAIN0005    NEGA R = 0.0 - 0.5 * B/A $
MAIN0006         AI = 0.5 * SQRTF(0.0-DISC)/A $
MAIN0007         PRINT 11,R,AI $
MAIN0008         GO TO FINISH $
MAIN0009    ZERO R = 0.0 - 0.5 * B/A $
MAIN0010         PRINT 21,R $
MAIN0011         GO TO FINISH $
MAIN0012    POSI SD = SQRTF(DISC) $
MAIN0013         R1 = 0.5*(SD-B)/A $
MAIN0014         R2 = 0.5*(0.0-(B+SD))/A $
MAIN0015         PRINT 31,R2,R1 $
MAIN0016 FINISH STOP $
MAIN0017      10 FORMAT( 3F12.5 ) $
MAIN0018      11 FORMAT( 19H TWO COMPLEX ROOTS:, F12.5,14H PLUS OR MINUS,
MAIN0019         F12.5, 2H I ) $
MAIN0020      21 FORMAT( 15H ONE REAL ROOT:, F12.5 ) $
MAIN0021      31 FORMAT( 16H TWO REAL ROOTS:, F12.5, 5H AND , F12.5 ) $
MAIN0022         END $
```

Spaghetti code: unstructured and difficult-to-maintain source code

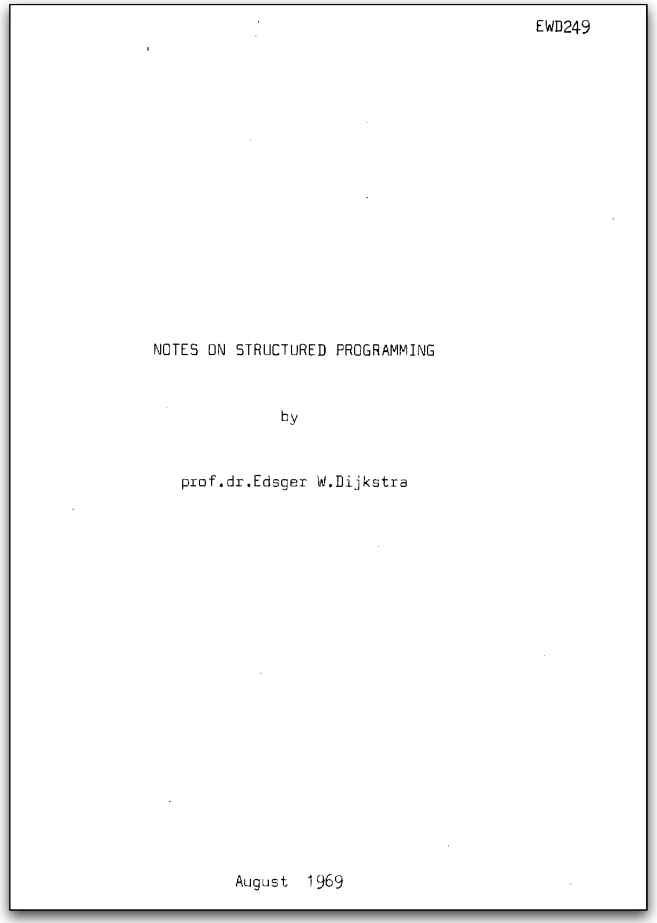
Structured Programming



Edsger Dijkstra

The competent programmer is fully aware of the strictly limited size of his own skull and therefore approaches the programming task in full humility.

Structured Programming



EWD249

NOTES ON STRUCTURED PROGRAMMING

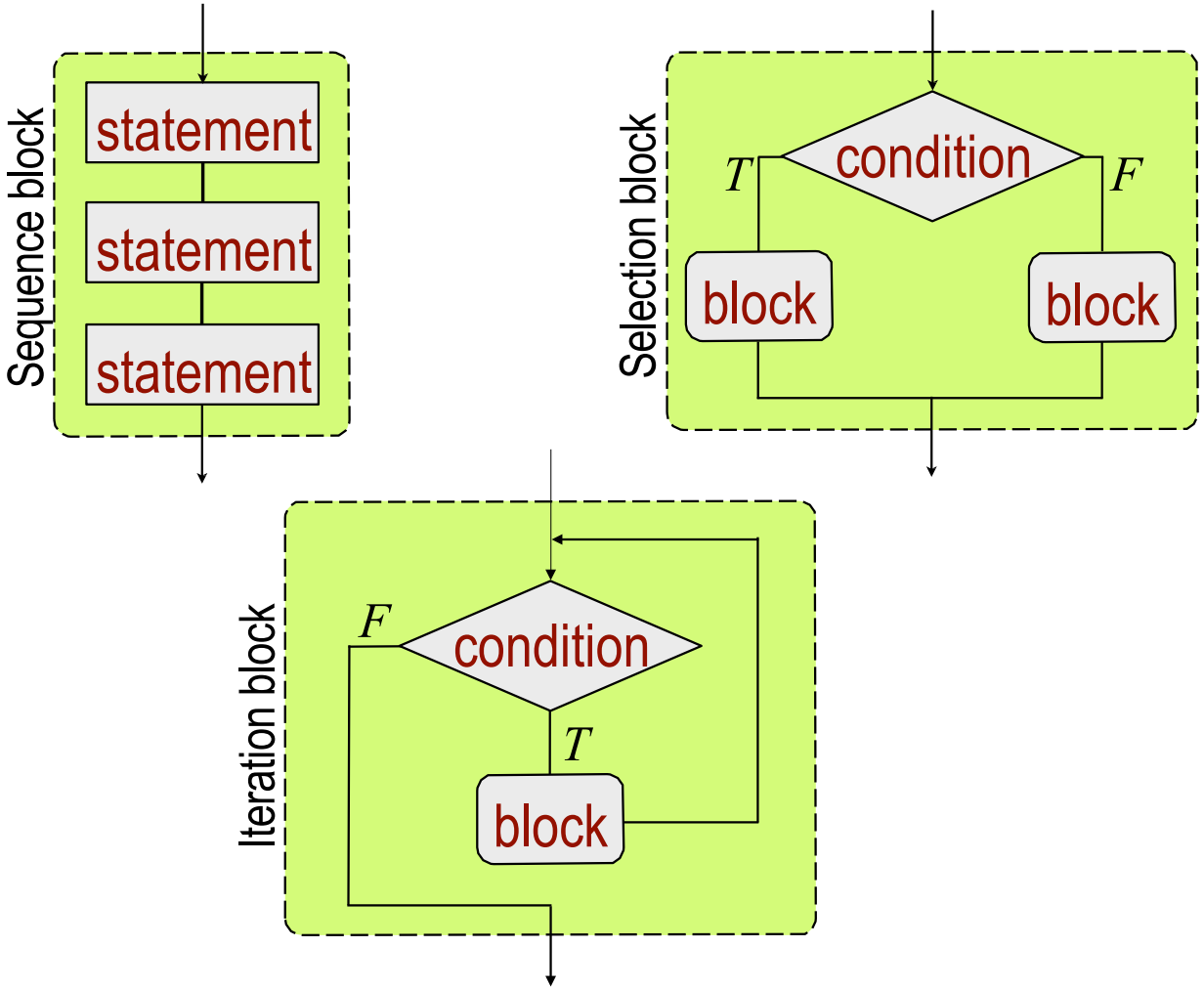
by

prof.dr.Edsger W.Dijkstra

August 1969

- Three basic constructs
  - *single-entry / single-exit control constructs*
  - *sequence, selection, iteration*
- Structured program
  - *ordered, disciplined, doesn't jump around unpredictably*
  - *can read easily and reason about ⇒ higher quality*

Structured Programming



- Three basic constructs
  - *single-entry / single-exit control constructs*
  - *sequence, selection, iteration*
- Structured program
  - *ordered, disciplined, doesn't jump around unpredictably*
  - *can read easily and reason about ⇒ higher quality*

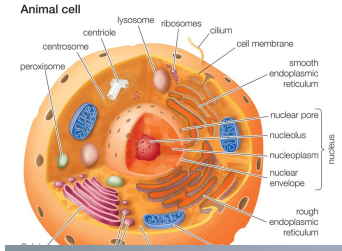
Object-Oriented Programming

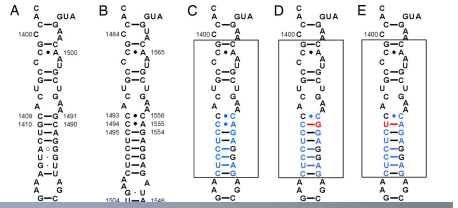
Simula67

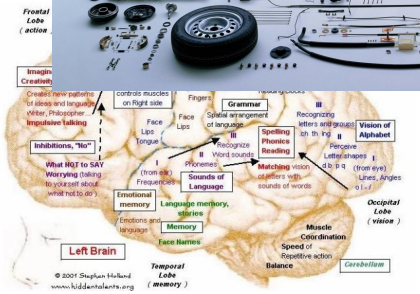
```
Begin
   Class Glyph;
      Virtual: Procedure print Is Procedure print;;
   Begin
   End;

   Glyph Class Char (c);
      Character c;
   Begin
      Procedure print;
        OutChar(c);
   End;

   Glyph Class Line (elements);
      Ref (Glyph) Array elements;
   Begin
      Procedure print;
      Begin
         Integer i;
         For i:= 1 Step 1 Until UpperBound (elements, 1) Do
            elements (i).print;
         OutImage;
      End;
   End;

   Ref (Glyph) rg;
   Ref (Glyph) Array rgs (1 : 4);

   ! Main program;
   rgs (1):- New Char ('A');
   rgs (2):- New Char ('b');
   rgs (3):- New Char ('b');
   rgs (4):- New Char ('a');
   rg:- New Line (rgs);
   rg.print;
End;
```

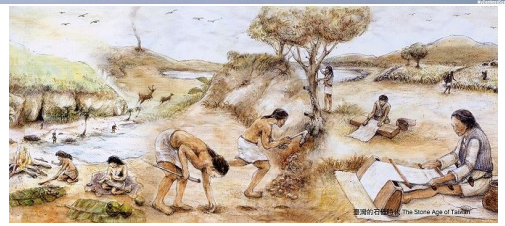
- Encapsulation in OOP
  - *self-contained modules*
  - *bind the data to the methods that process it*
- Simula67 developed in the late 60s
  - *followed by Smalltalk, C++, C#, Java, etc.*

Modularity Outside Computer Science



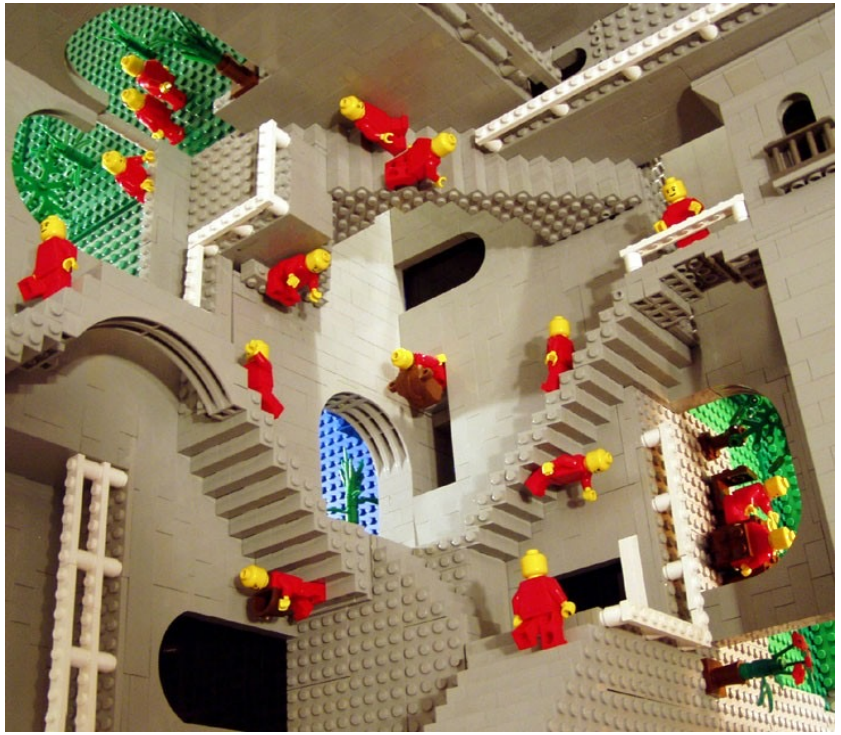






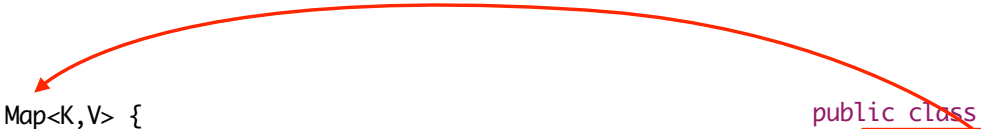
- Cell = module used to build organisms
- Gene = unit (module) of evolution
- Cognitive activity (may be) modularized
- Division of labor = modularization
- Spare parts = modules
- IKEA … modular furniture

Modularity in Software Engineering



- Modules = replaceable units
- Classes in OOP
  - *contain behaviors inside the modules*
  - *implement them in isolation*
  - *test them in isolation*
  - *configure them separately*
  - *maintain them separately*
- Ultimate goal
  - *productively reason about combination of classes*

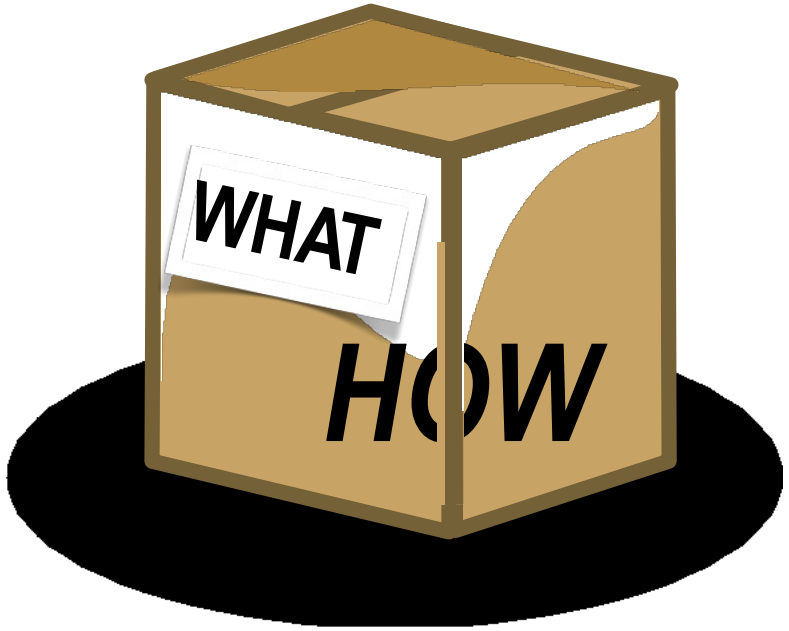
Abstraction



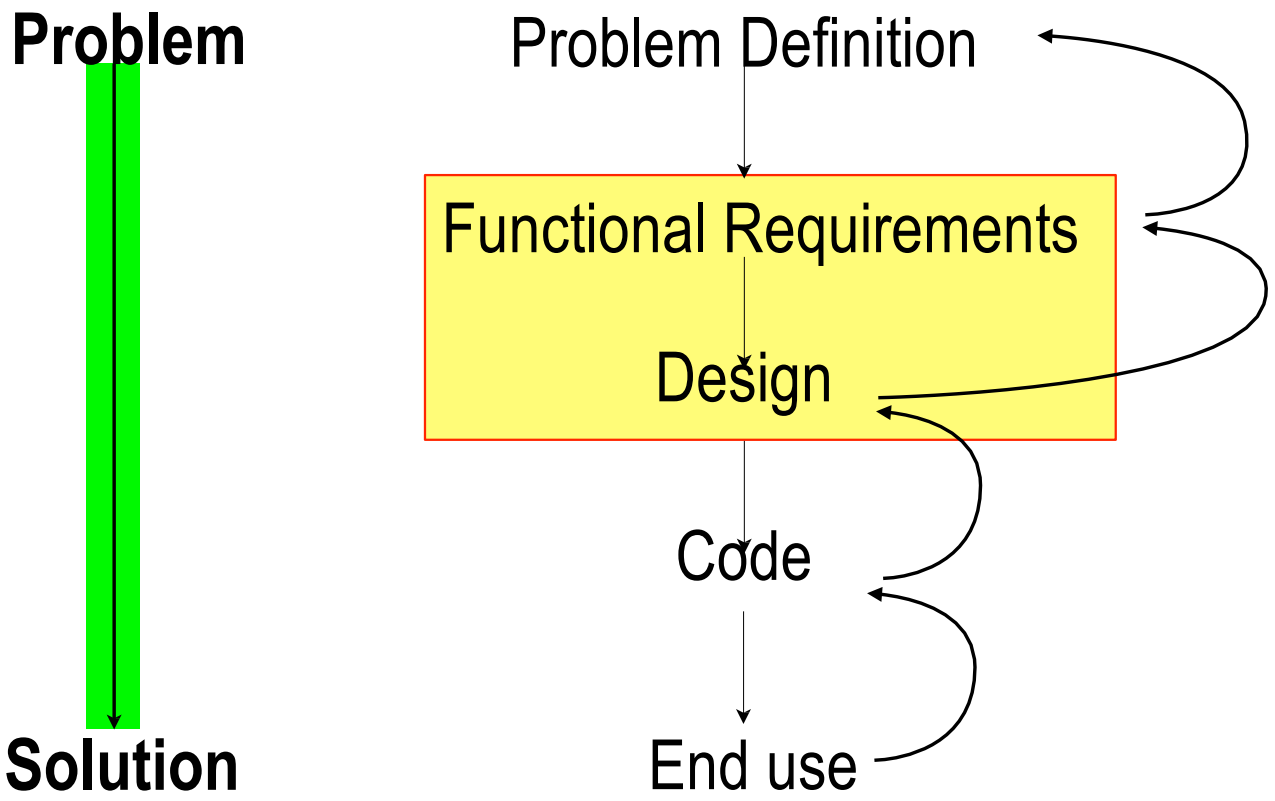
```java
public interface Map<K,V> {
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);

    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);
        boolean equals(Object o);
        int hashCode();
    }

    boolean equals(Object o);
    int hashCode();
}
```

```java
public class HashMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
{
    static final int DEFAULT_INITIAL_CAPACITY =
    16; static final int MAXIMUM_CAPACITY = 1 <<
    30; static final float DEFAULT_LOAD_FACTOR =
    0.75f; transient Entry[] table;
    transient int size;
    int threshold;
    final float loadFactor;
    transient volatile int modCount;

    public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " +
                                               loadFactor);
        int capacity = 1;
        while (capacity < initialCapacity)
            capacity <<= 1;

        this.loadFactor = loadFactor;
        threshold = (int)(capacity * loadFactor);
        table = new Entry[capacity];
        init();
    }

    static int hash(int h) {
        h ^= (h >>> 20) ^ (h >>> 12);
        return h ^ (h >>> 7) ^ (h >>> 4);
    }
    // ...
```

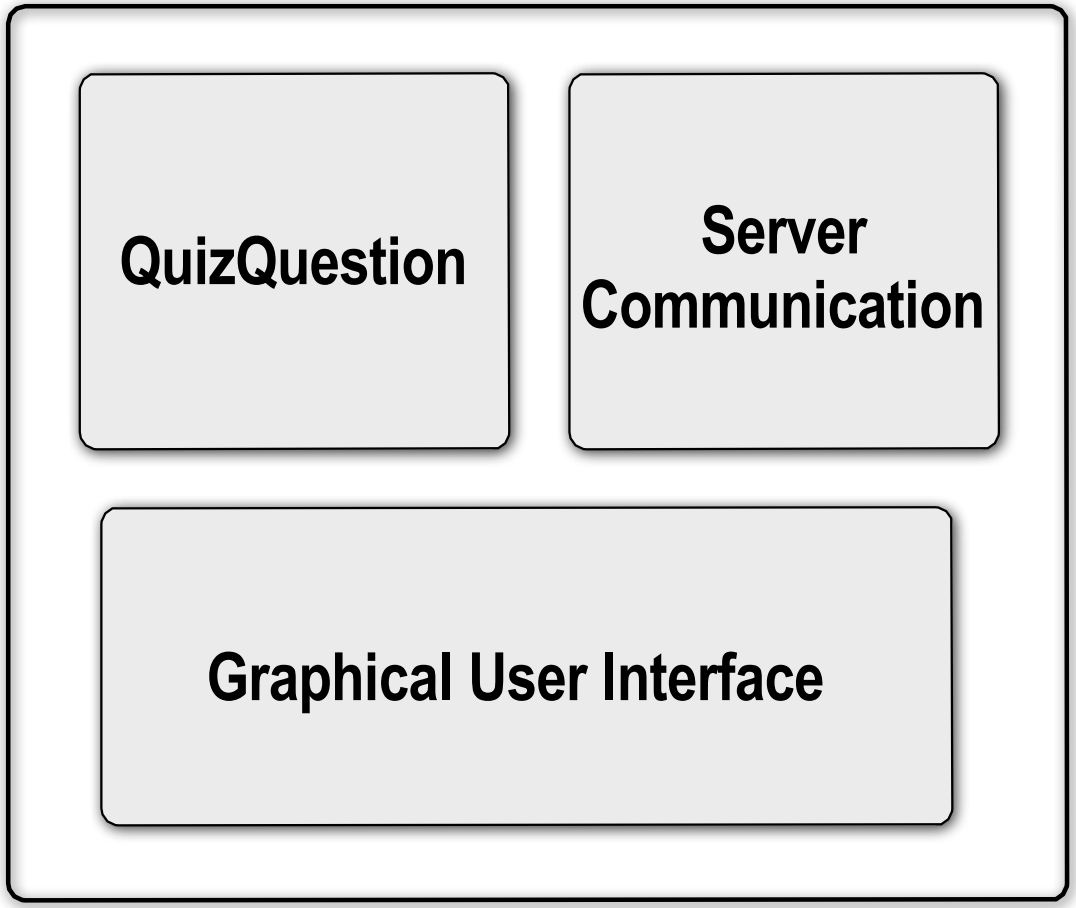
Abstraction



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

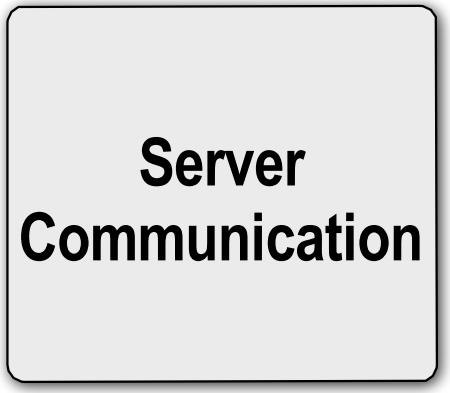

Problem
Solution
Problem Definition
Functional Requirements
Design
Code
End use

Top-Down Design



- Break down into major subsystems
  - *e.g., platform dependencies, database access, business logic, menu hierarchies*

Top-Down Design



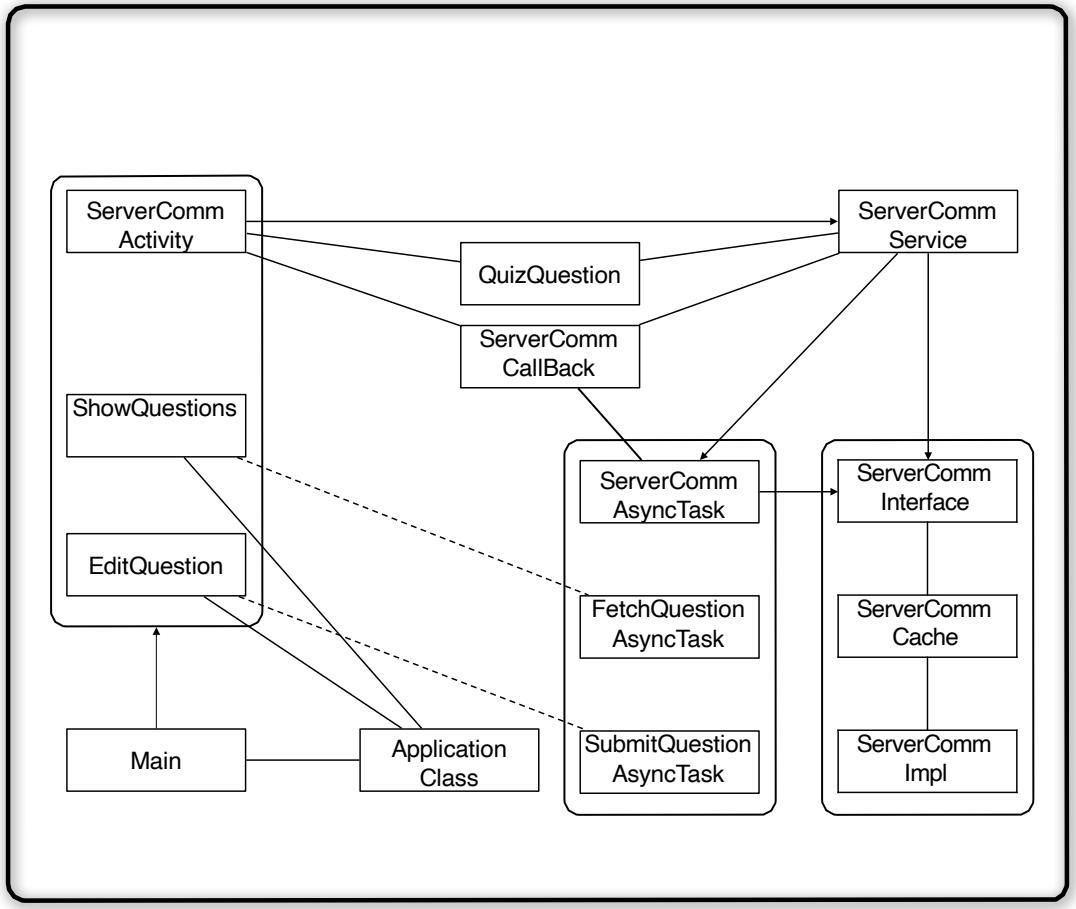
- Break down into major subsystems
  - *e.g., platform dependencies, database access, business logic, menu hierarchies*

Top-Down Design

```
public class ServerCommunication {
    public static ServerCommunicationFactory getInstance() {}

    public static void getRandomQuestion(AsyncTaskCallBack<QuizQuestion> parent,
         String session) {}
    public static void postQuestion(QuizQuestion question,
         AsyncTaskCallBack<QuizQuestion> parent, String session)
    {} public static void updateQuestion(QuizQuestion question,
         AsyncTaskCallBack<Void> parent, String session) {}
    public static void deleteQuestion(String questionId,
         AsyncTaskCallBack<Void> parent, String session) {}
    public static void authenticate(String username, String password,
         AsyncTaskCallBack<LoginStatus> parent) {}
    public static void getQuestionsOwnedBy(String username,
         AsyncTaskCallBack<List<QuizQuestion>> parent) {}
    public static void getQuestionsTaggedWith(String tag,
         AsyncTaskCallBack<List<QuizQuestion>> parent) {}
    public static void getRatings(QuizQuestion question, String session,
         AsyncTaskCallBack<AggregatedRatings> parent) {}
    public static void setRating(QuizQuestion question, String session,
         RatingType rating, AsyncTaskCallBack<RatingType> parent) {}
    public static void getMyRating(QuizQuestion question, String
    session,
         AsyncTaskCallBack<RatingType> parent) {}
    public static void getMyKarma(String session,
         AsyncTaskCallBack<Karma> parent) {}
    public static void fetchTests(String session,
         AsyncTaskCallBack<List<QuizTest>> parent) {}
    public static void fetchSingleTest(String session, String testId,
         AsyncTaskCallBack<QuizTest> parent) {}
    public static void postTestAnswers(String session, String testId,
}        int[] answers, AsyncTaskCallBack<Double> parent) {}
```

- Break down into major subsystems
  - *e.g., platform dependencies, database access, business logic, menu hierarchies*
- Define interfaces for subsystems

Top-Down Design



- Break down into major subsystems
  - *e.g., platform dependencies, database access, business logic, menu hierarchies*
- Define interfaces for subsystems
- Identify all major classes
- Define **interfaces** for these classes
  - enough detail to be directly implementable
  - not more detail than is strictly necessary
  - only specify the what, not the how

Classes <-> Real-world Objects

```
public interface StudentInterface {
    public void Student(String first, String last,
            String email, int section);
    public String toString();
    public String getLastName();
    public void setLastName(String lastName);
    public String getEmailAddr();
    public void setEmailAddr(String emailAddr);
    public String getFirstName();
    public void setFirstName(String firstName);
}
```

- Identify objects and their attributes
  - e.g., student, car, teacher, ...
- What can be done to object ?
- What can object do to other objects ?
  - containment / inheritance
- Essential vs. incidental properties
  - essential for a car: engine, wheels, doors, ...
  - incidental for a car: turbo engine, color red, ...
- Minimize complexity
  - reduce what is required to fit in one brain
  - keep incidental complexity from proliferating

Internal Class Design

```
public class HashMap<K,V>
    extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
{
    static final int DEFAULT_INITIAL_CAPACITY = 16;
    static final int MAXIMUM_CAPACITY = 1 << 30;
    static final float DEFAULT_LOAD_FACTOR = 0.75f;
    transient Entry[] table;
    transient int size;
    int threshold;
    final float loadFactor;
    transient volatile int modCount;

    public HashMap() { /* ... */ }

    public V get(Object key) { /* ... */ }
    public boolean containsKey(Object key) { /* ... */ }

    public V put(K key, V value) { /* ... */ }
    public void putAll(Map<? extends K, ? extends V> m) { /* ... */ }

    public V remove(Object key) { /* ... */ }

    public void clear() { /* ... */ }

    public Object clone() { /* ... */ }

    public Set<K> keySet() { /* ... */ }
    public Collection<V> values() { /* ... */ }
    public Set<Map.Entry<K,V>> entrySet() { /* ... */ }

    // ...
}
```

- Typically left to programmer
  - *pseudocode, look up algorithms, organize code*
- Data structures
  - *think of classes as encapsulators of state*
- Access methods
  - *how does the external world access the state ?*
- Transformation methods
  - *how does the external world take the object from one state to another ?*
- May need to rework interface
  - *internal class design reveals issues in interface*

Other Examples of Abstractions in Programming Languages

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)
```

Closure

- Closure is a function with an extended scope that encompasses nonglobal variable 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: a violation of the type system

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

Deduces that the **relevant fact about bird is "has methods Quack and Walk"**

What the object can actually do,
rather than what the object is

Duck Typing

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


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


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


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

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: ?
  – producers: ?
  – observers: ?
  – 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: ?
  - producers: ?
  - observers: ?
  - mutators: ?

Abstraction Functions and Representation Invariants

- A more formal mathematical idea of what it means **for a class to implement an ADT**, via the notions of *abstraction functions* and *rep invariants*.
- Has immediate practical application to the design and implementation of abstract types.

- In thinking about an abstract type, it helps to consider the relationship between two spaces of values.
- The space of representation values (or rep values for short) consists of the values of the actual implementation entities.
- The space of abstract values consists of the values that the type is designed to support. They are the way we want to view the elements of the abstract type, as clients of the type.

- Now of course the implementor of the abstract type must be interested in the representation values, since it is the implementor’s job to achieve the illusion of the abstract value space using the rep value space.

Interface Abstraction

A: the set of abstract values
In this case, the set of characters

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

Set

Set ← Set ∪ {c}

Set ← Set \ {c}

c ∈ Set ⇒ return true ∧ c ∉ Set ⇒ 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;
   }
}
```

- The rep space R - Strings
- The abstract space A – mathematical sets of characters

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;
    }
}
```

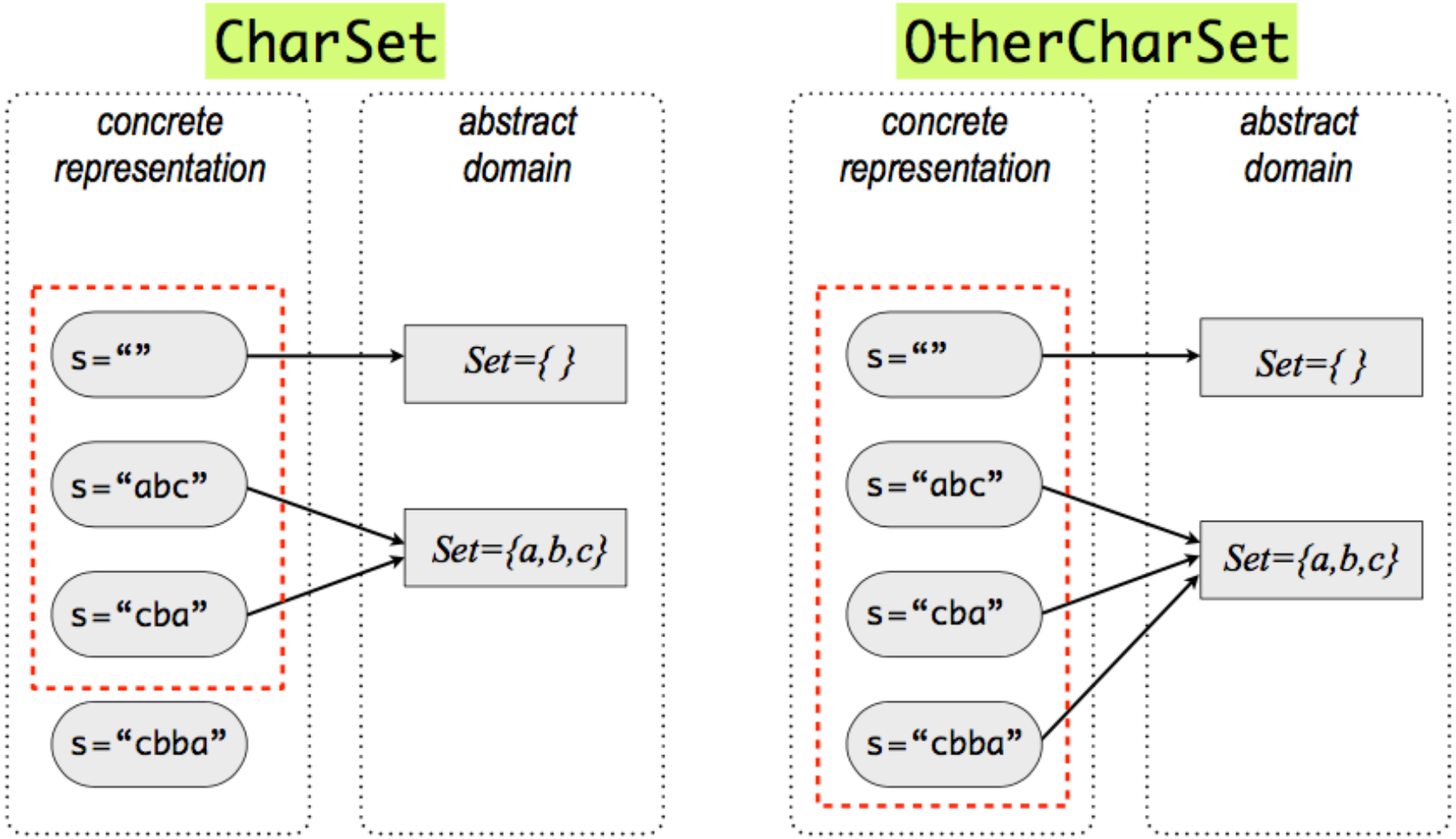
```
class OtherCharSet implements CharSetInterface {
    private StringBuffer s;

    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: R->A

Every abstract value is mapped to by some rep value

Some abstract values are mapped to by more than one rep value.

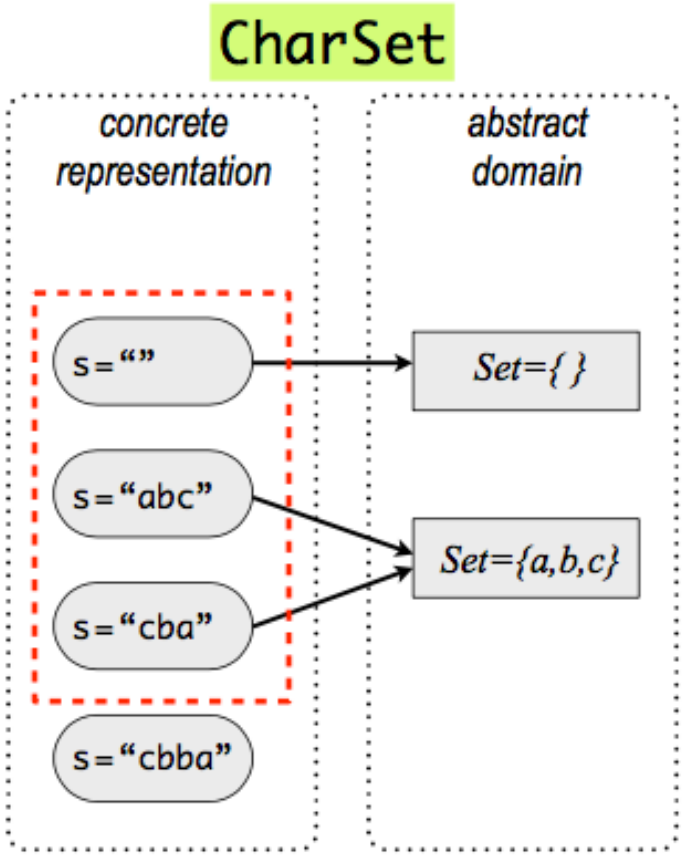
Not all rep values are mapped

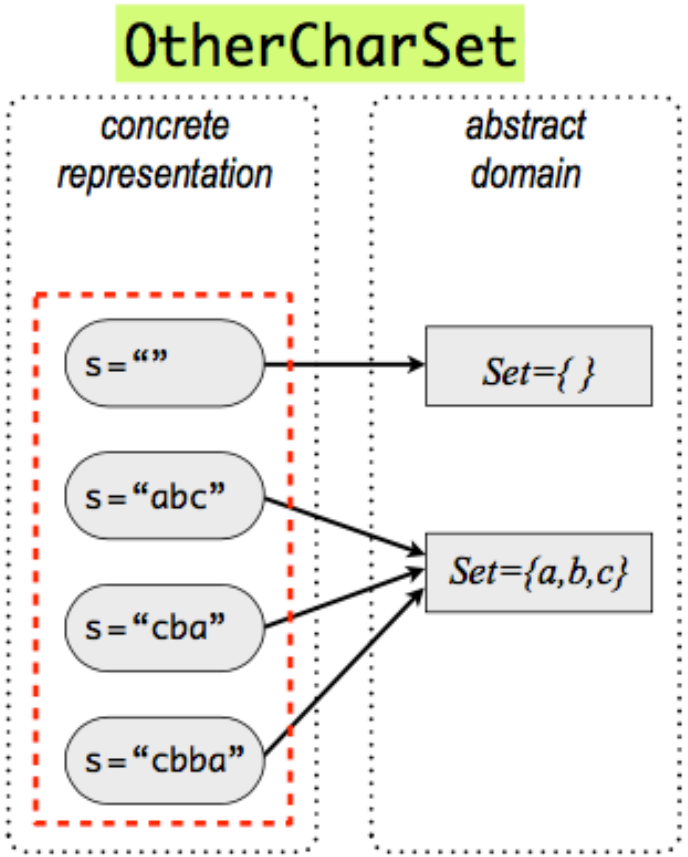


- In practice, we can only illustrate a few elements of the two spaces and their relationships; the graph as a whole is infinite.
- We describe it by giving two things:
  - An *abstraction function* that maps rep values to the abstract values they represent:
    AF : R → A
  - A *rep invariant* that maps rep values to booleans:
    RI : R → Boolean
    For a rep value r, RI(r) is true if and only if r is mapped by AF.
    Alternatively, you can think of RI as a set: it's the subset of rep values on which AF is defined.

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.





- Both the rep invariant and the abstraction function should be documented in the code, right next to the declaration of the rep itself:

```
public class CharSet_NoRepeatsRep implements Set<Character> {
    private String s;
    // Rep invariant:
    //       s contains no repeated characters
    // Abstraction Function:
    //       represents the set of characters found in s
    ...
}
```

- Same rep value space – different rep invariant:

```
public class CharSet_SortedRep implements Set<Character> {
    private String s;
    // Rep invariant:
    // s[0] < s[1] < ... < s[s.length()-1]
    // Abstraction Function:
    // represents the set of characters found in s
    …
}
```

- Even with the same type for the rep value space and the same rep invariant RI, we might still interpret the rep differently, with different abstraction functions AF
- There's no *a priori* reason to let the rep decide the interpretation.

```
public class CharSet_SortedRangeRep implements Set<Character> {
    private String s;
    // Rep invariant:
    // s.length is even
    // s[0] <= s[1] <= ... <= s[s.length()-1]
    // Abstraction Function:
    // represents the union of the ranges
    // {s[i]...s[i+1]} for each adjacent pair of characters in s
    ...
}
```

Perhaps we'll interpret consecutive pairs of characters as subranges, so that the string "acgg" represents the set {a,b,c,g}.

- The essential point is that designing an abstract type means **not only choosing the two spaces** –
  the abstract value space for the specification and
  the rep value space for the implementation –
  but also deciding what rep values to use and how to interpret them.
- It's critically important to write down these assumptions in your code, as we've done above, so that future programmers (and your future self) are aware of what the representation actually means.

Abstraction vs. Implementation

```
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) {
        // ...
    }

    // ...
}
```

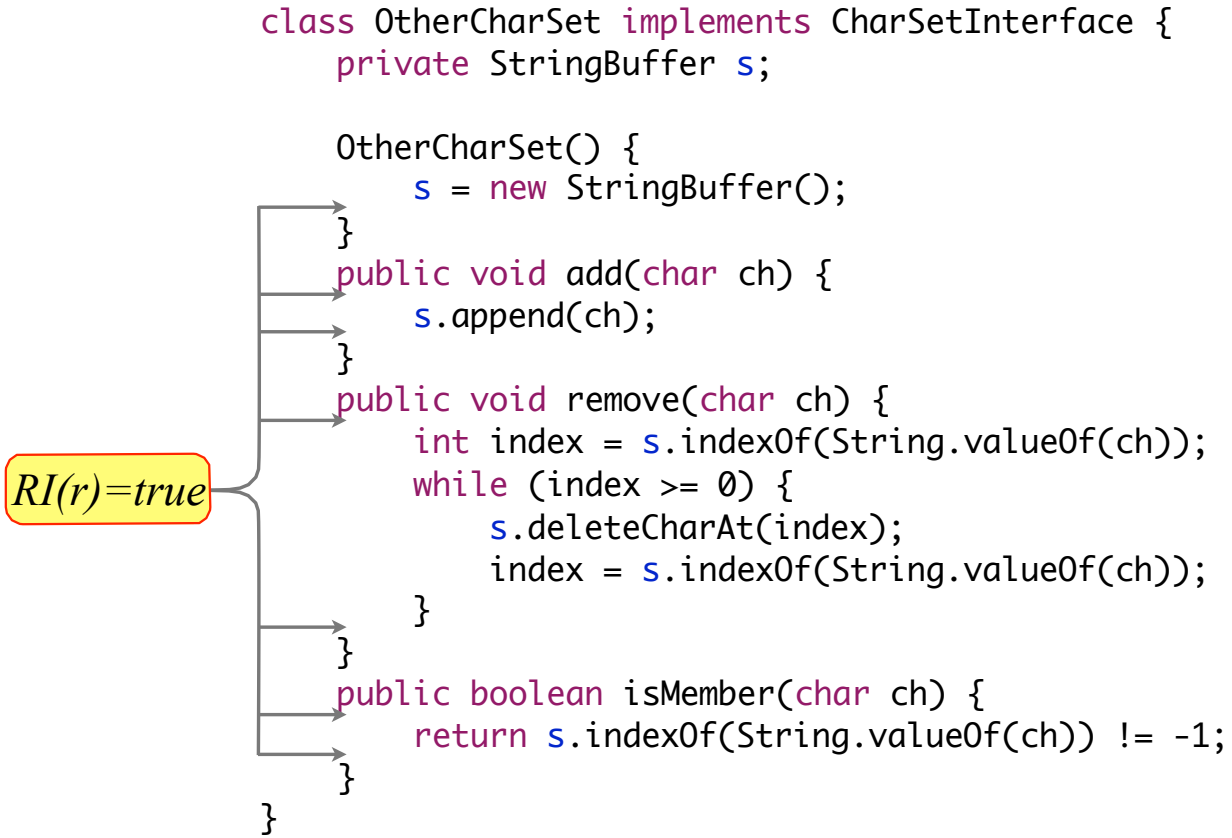
$RI(r)=true \Rightarrow AF(r)$ *turns r into desired abstraction*

- To implement abstraction
  - Choose r and RI(r)
  - Implement AF(r)
  - Ensure RI(r) is preserved

Audit Methods

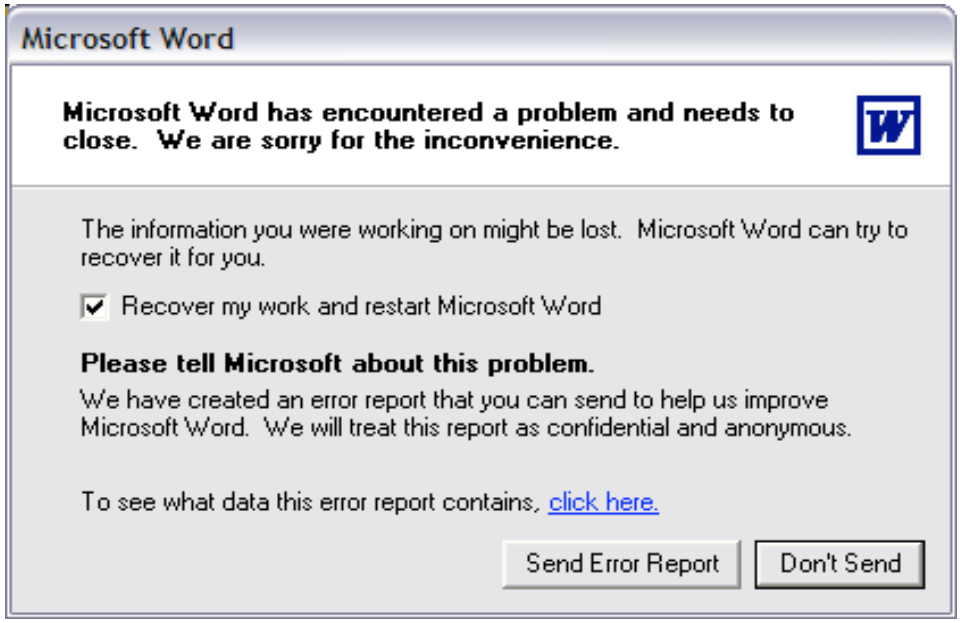
Auditing the Rep(resentation)

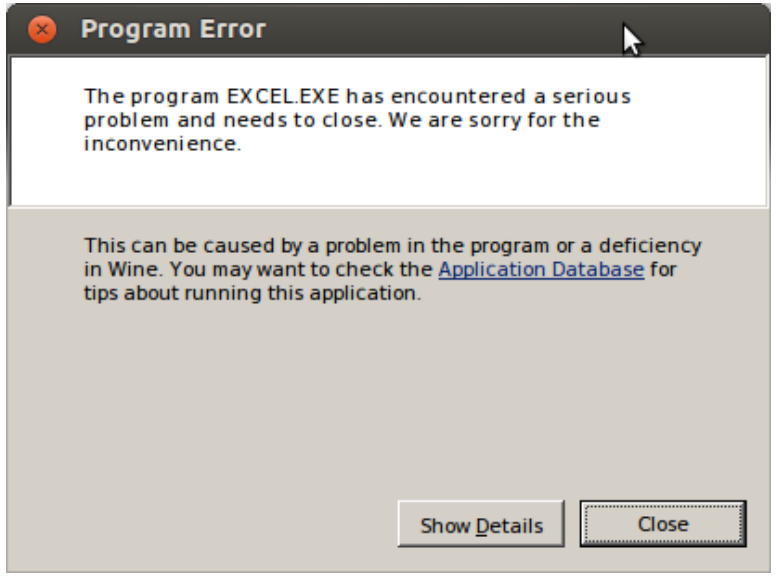
- *RI(r)=true ⇒ AF(r) turns r into desired abstraction*



- 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≠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

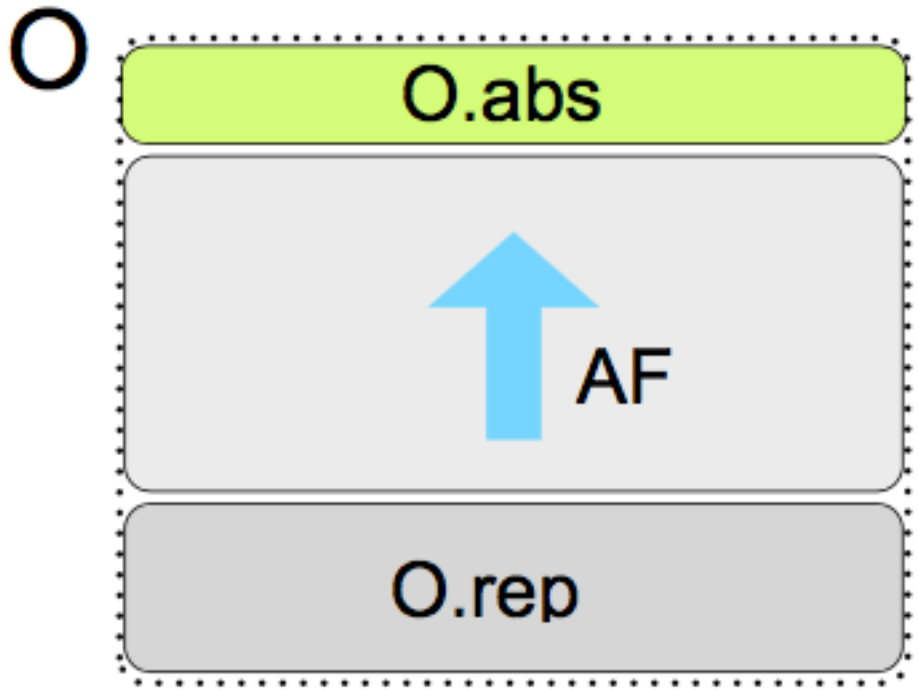
```
int auditErrors(int depth) {
   if (depth==0) {
      return 0;
   }

   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

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**



O exposes correct abstraction

⇑

O.abs = AF(O.rep)

⇑

O.auditErrors()=0 ⇒ RI(O.rep)=true

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,
      /* ...
        ...
        ... */ )
```

- 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

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

```
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);
```

ORACLE

```
void updateOracleDb(ResultSet newStuff);
```

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

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

4. Extensible Systems

```
void populateDirectory(LinkedList<User> users)
```

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

4. Extensible Systems

```
public boolean currentStatus()
```

```
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

```
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

```
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.