Basic Principles of analysis and testing software

Software Reliability and Testing - Barbara Russo
SwSE - Software and Systems Engineering Research Group

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Basic principles of analysis and testing

- As in any engineering discipline, techniques of software analysis and testing follow few key **principles**
- Principles aim at **distinguishing** one technique from another and determining the **scope and the limits** of the technique itself.

Sensitivity

Better to fail every time than sometimes

Sensitivity

- Sensitivity requires techniques of abstraction: system behaviour cannot be related to specific circumstances

Example in Java

- Run-time **exceptions** help detect errors in a systematic way
- **ArrayIndexOutOfBoundException**:
  - it checks that the number of entries of an array does not exceed the available length of an array.
  - In languages like C, this is not checked and the array can be overwritten (wrapping) or the input can be cut with no notice to the execution thread

Example in Java

- **ConcurrentModificationException:**
  - When one or more thread is iterating over a collection, in between, one thread changes the structure of the collection (**race condition**)
  - These changes can lead to **unexpected behaviour** that might cause a failure

Fail fast

- Fail fast iterator while iterating through the collection, **instantly throws ConcurrentModificationException** if there is any structural modification of the collection
- Thus, when a concurrent modification occurs, the iterator fails **quickly and cleanly**, rather than risking arbitrary, non-deterministic behaviour at an undetermined time in the future.

Two scenarios

- Single Threaded Environment:
  - After the creation of the iterator, the structure is modified at any time **by any method other than iterator's** own remove method
- Multiple Threaded Environment:
  - If one thread is modifying the structure of the collection while **another thread** is iterating over it

Example

```
/* import what you need*/
public class FailFastExample{
   public static void main(String[] args){
        Map<String,String> premiumPhone = new HashMap<String,String>();
        premiumPhone.put("Apple", "iPhone");
        premiumPhone.put("HTC", "HTC one");
        premiumPhone.put("Samsung","S5");
        Iterator iterator = premiumPhone.keySet().iterator();
        while (iterator.hasNext()){
           System.out.println(premiumPhone.get(iterator.next()));
           premiumPhone.put("Sony", "Xperia Z");
        }
   }
```

Output

```
iPhone
Exception in thread "main"
java.util.ConcurrentModificationException
        at java.util.HashMap$HashIterator.nextEntry(Unknown Source)
        at java.util.HashMap$KeyIterator.next(Unknown Source)
        at FailFastExample.main(FailFastExample.java:xx)
```

Fail safe

- Fail Safe Iterator **makes copy of the internal data structure** (object) and iterates over the copied data structure
- Any structural modification done to the iterator affects the copied data structure
- Thus, original data structure remains structurally unchanged

Fail safe

- No ConcurrentModificationException throws by the fail safe iterator
- Two issues associated with Fail Safe Iterator are :
  - **Overhead** of maintaining the copied data structure i.e memory
  - It does not guarantee that the data being read is the data currently in the **original data structure**

Example

```
/* import what you need*/
public class FailSafeExample{
   public static void main(String[] args{
      ConcurrentHashMap<String,String> premiumPhone =  new
      ConcurrentHashMap<String,String>();
      premiumPhone.put("Apple", "iPhone");
      premiumPhone.put("HTC", "HTC one");
      premiumPhone.put("Samsung","S5");
      Iterator iterator = premiumPhone.keySet().iterator();
      while (iterator.hasNext()) {
         System.out.println(premiumPhone.get(iterator.next()));
         premiumPhone.put("Sony", "Xperia Z");       }
}
```

Output

- iPhone
- HTC one
- S5

Differences

	Fail Fast Iterator	**Fail Safe Iterator**
Throw ConcurrentModification	Yes	No
Clone object	No	Yes
Memory Overhead	No	Yes
Examples	HashMap,Vector,ArrayList, HashSet	CopyOnWriteArrayList, ConcurrentHashMap

Other examples in testing

- When it uses a systematic strategy (e.g., using checklists or guidelines), **code inspection** can help find faults on regular basis

Redundancy

Making intention explicit

Redundancy

- From information theory: redundancy means **dependency** between transmissions.
  - Solution: create **guards** against transmission errors
- In software, redundancy means **consistency** between intended and actual system behaviour
  - Solution: create **guards** for artefacts consistency, **making intention explicit**

Examples

- Dependency among parts of code by using a variable
  - A variable is defined and then used elsewhere
- **Type declaration** is a form a redundancy
  - Type declaration constraints the way a variable is used in other part of the code
  - The compiler checks the correct use of the variable against its declared type

Restriction

Making the problem easier (substituting principle) or reducing the class under test

Substituting principle

- Verifying properties can be infeasible
  - **Substituting** a property with one that can be easier verified or
  - **constraining** the class of programs to verify

Examples

- A weaker spec may be easier to check:
  - It is impossible (in general) to show that pointers are used correctly, but the simple Java requirement that pointers are initialised (not null) before use is simple to enforce

Example

```
static void questionable(){
     int k;
     for(int i=0; i<10;i++){
          if(someCondition(i)){
               k=0;
          }
     }
}
```

Example

- Compilers cannot be sure that k will ever be initialised: it depends on the condition
- Make the problem easier: modern Java compilers do not allow this code

Partition

Divide and conquer: typical engineering principle

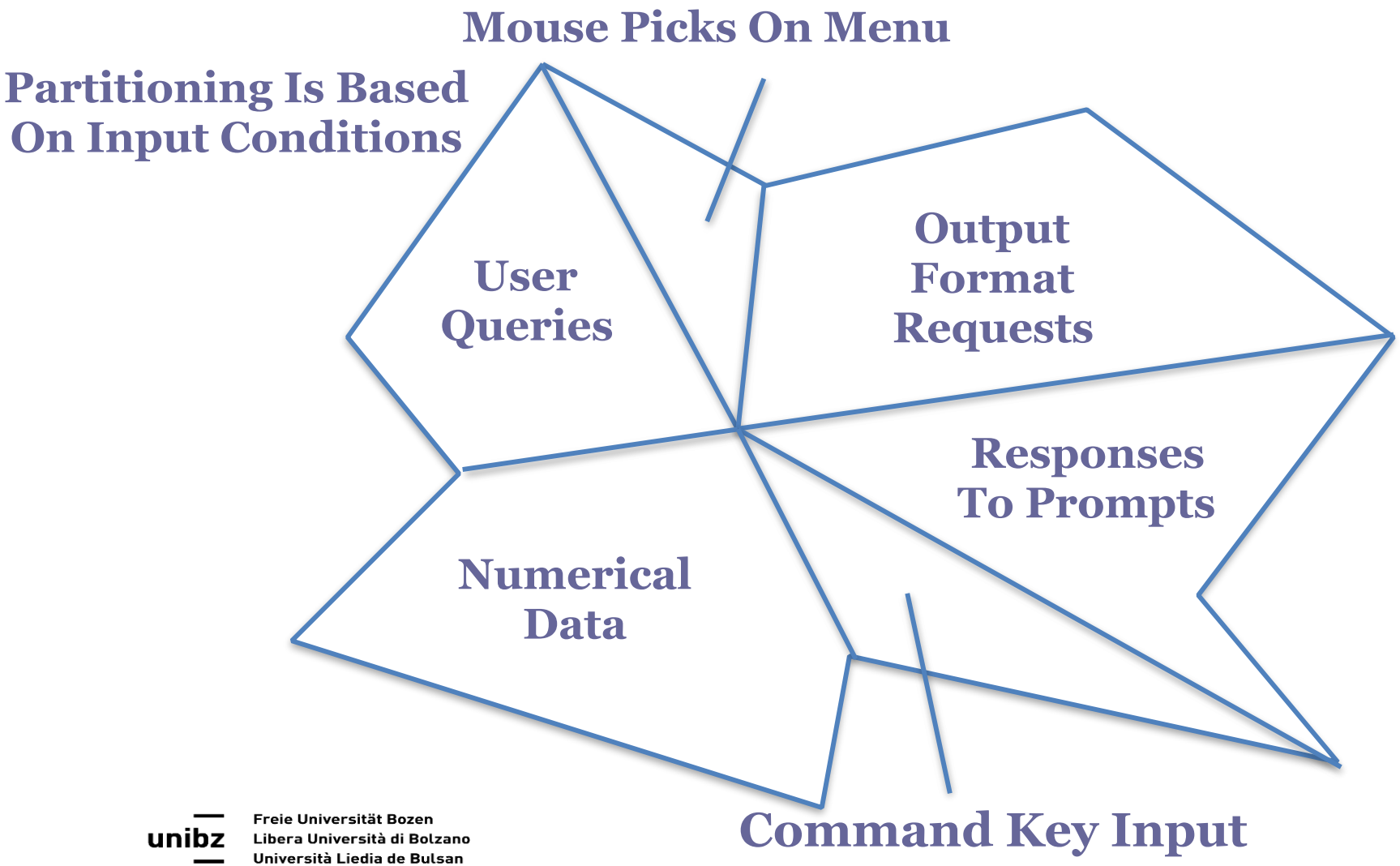
Example

- Divide testing into: unit testing, integration testing, subsystem and system testing to focus on different types of faults at different stages
  - At each stage, take advantage of the result of the previous stage

Example

- Divide input into classes of **equivalent expected output**
- Then we use test criteria to identify representatives in classes to test a program

Equivalence partitioning



Visibility

Setting goals and methods to achieve those goals

Making information accessible

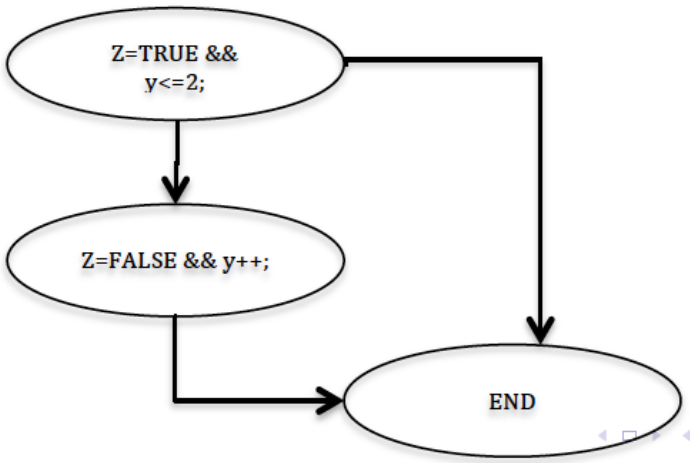
Models

- Models are simpler than the reality as they represent reality with a limited number of factors
- They help factorise the reality and perform program analysis and testing simply and fast

Control Flow Graphs

- CFG keeps information of next instruction to be executed and neglects variable values

```
1 boolean z = FALSE;
2 if(z && y<=2){
3   z=FALSE;
4   y++;
5 }
Some paths are infeasible
```



Feedback

Apply lessons learned from experience in process improvement and techniques

Examples

- Learning from experience: Each project provides information to improve the next
  - **Checklists** are built on the basis of errors revealed in the past
  - **Error taxonomies** can help in building better test selection criteria
  - **Design guidelines** can avoid common pitfalls

Examples

- Iterative testing in eXtreme programming
- Prototyping
- Data mining

Lessons learned

- There are few principles which testing techniques must adhere to
- Some coding languages provide syntax to prevent faults