Basic Principles of analysis and testing software

Barbara Russo

SwSE - Software and Systems Engineering Research Group

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

Solution: 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

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)
```

Solution: Fail safe

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

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

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**

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

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

Substituting principle

- In complex system, a direct verification can be infeasible
- Often this happens when properties are related to specific human judgements, but not only

Substituting principle

- Substituting a property Q with another one Q' that can be easier verified
- <u>Examples:</u>
  - Constraining the class of programs to verify
  - Separate human judgment from objective verification
  - Exploiting programming language's feature: serialization

Example - weaker specs

- 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 - compiler verification

```
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

Example - smoke testing

- Smoke testing: preliminary testing to reveal simple failures severe enough to, for example, reject a prospective software release.

Example - serialization

- “Race condition": interference between writing data in one process and reading or writing related data in another process (e.g., an array accessed by different threads)
- To avoid race conditions: testing the **integrity** of shared data
  - It is difficult as it is checked at run time
  - Substitution principle: adhere to a protocol of **serialisation**

Serialisation

- When group of objects or states can be transmitted as one entity and then at arrival reconstructed into the original distinct objects

Example: Java object serialisation

- An object can be represented as a **sequence of bytes** that includes the object's data as well as information about the object's type and its types of data

- After a serialised object has been written into some kind of memory, it can be read from it and deserialised: the type information and bytes that represent the object and its data can be used to recreate the object in memory
- The serialized object is not modified while is dispatched, thus the deserialized object preserves the integrity of the original object

Java object serialisation

- The ObjectOutputStream class contains the method

```
public final void writeObject(Object x)
throws IOException
```

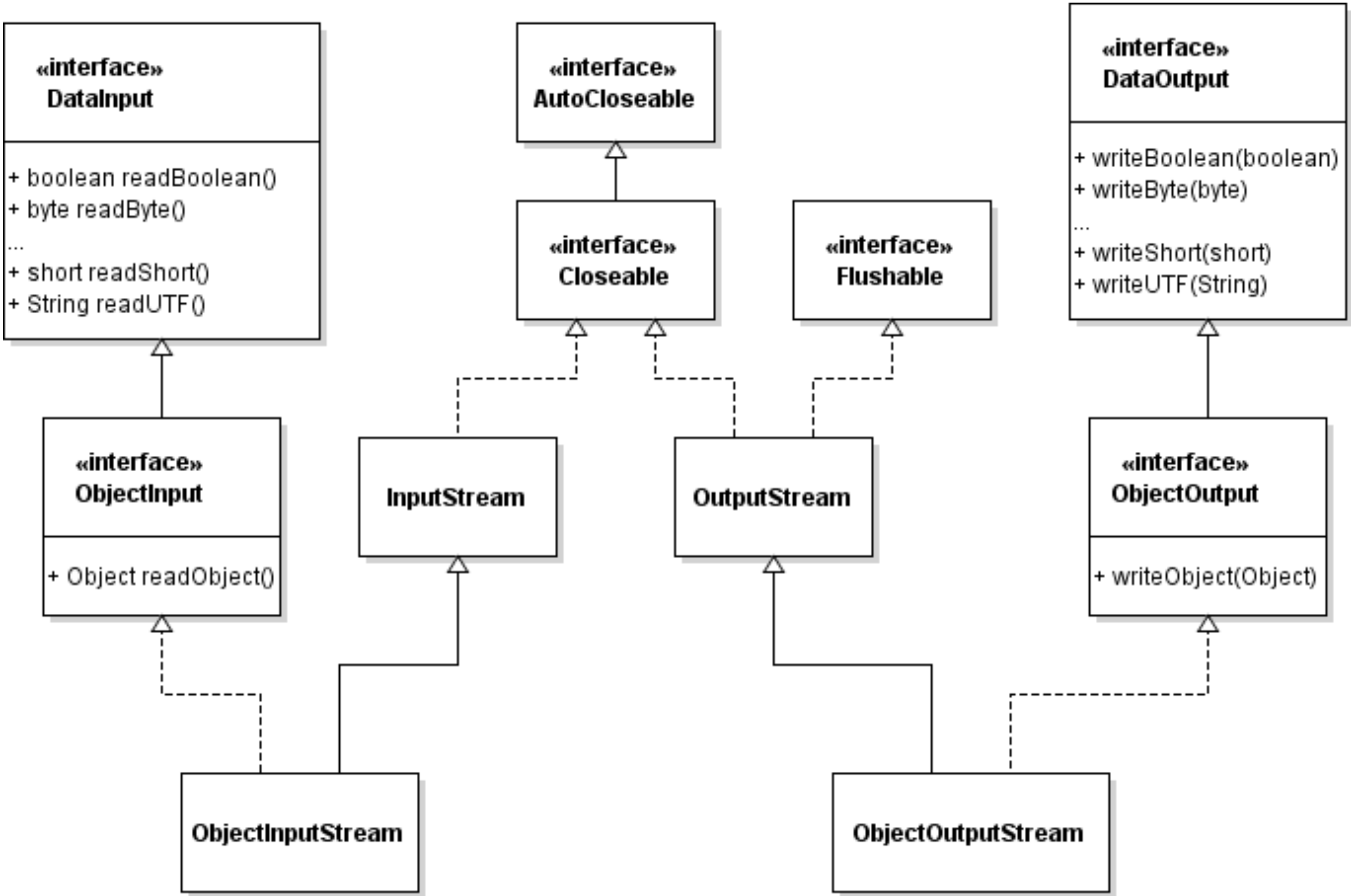
- The method serialises an Object and sends it to the output stream

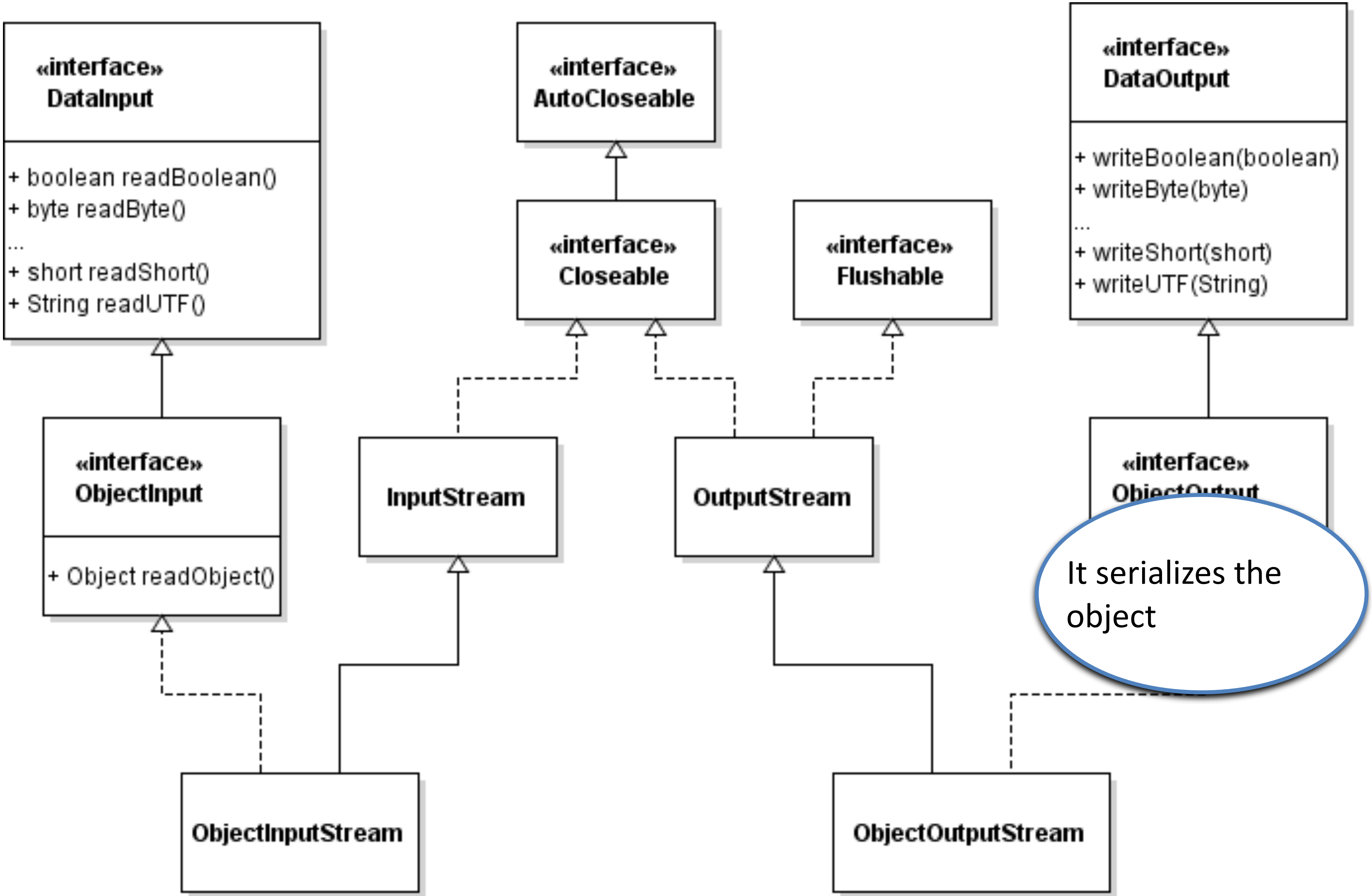
Java object serialisation

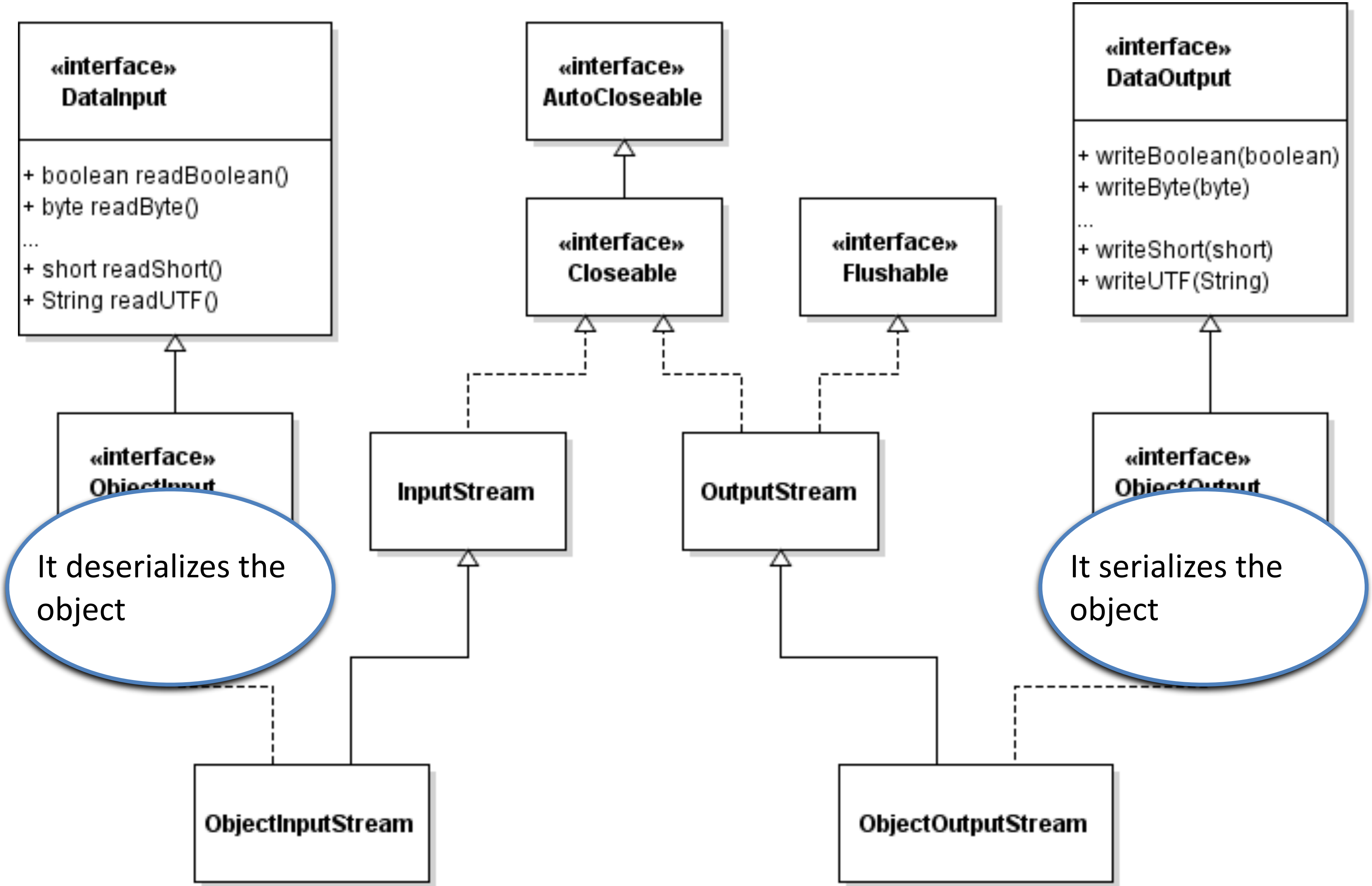
- Similarly, the ObjectInputStream class contains the method for deserialising an object:

```
public final Object readObject() throws
IOException, ClassNotFoundException
```

- This method retrieves the next Object out of the stream and deserialises it


«interface»
DataInput
+ boolean readBoolean()
+ byte readByte()
...
+ short readShort()
+ String readUTF()
«interface»
AutoCloseable
«interface»
Closeable
«interface»
Flushable
«interface»
DataOutput
+ writeBoolean(boolean)
+ writeByte(byte)
...
+ writeShort(short)
+ writeUTF(String)
«interface»
ObjectInput
+ Object readObject()
InputStream
OutputStream
«interface»
ObjectOutput
+ writeObject(Object)
ObjectInputStream
ObjectOutputStream


«interface»
DataInput
+ boolean readBoolean()
+ byte readByte()
...
+ short readShort()
+ String readUTF()
«interface»
AutoCloseable
«interface»
Closeable
«interface»
Flushable
«interface»
DataOutput
+ writeBoolean(boolean)
+ writeByte(byte)
...
+ writeShort(short)
+ writeUTF(String)
«interface»
ObjectInput
+ Object readObject()
InputStream
OutputStream
«interface»
It serializes the object
ObjectInputStream
ObjectOutputStream


«interface»
DataInput
+ boolean readBoolean()
+ byte readByte()
...
+ short readShort()
+ String readUTF()
«interface»
AutoCloseable
«interface»
Closeable
«interface»
Flushable
«interface»
DataOutput
+ writeBoolean(boolean)
+ writeByte(byte)
...
+ writeShort(short)
+ writeUTF(String)
«interface»
InputStream
OutputStream
«interface»
It deserializes the object
It serializes the object
ObjectInputStream
ObjectOutputStream

Partition

Divide and conquer: typical engineering principle

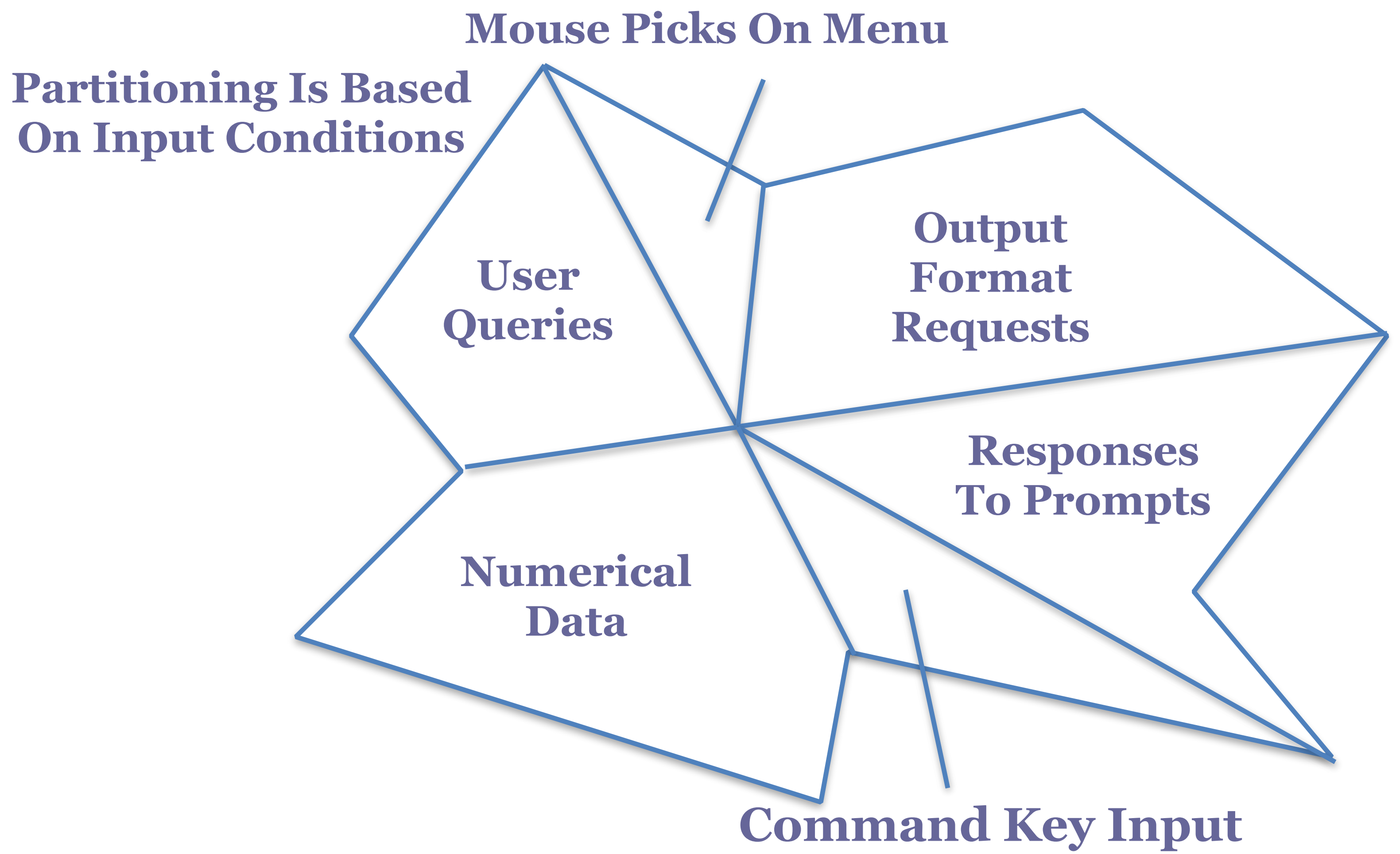
Example

- 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 - partition

- 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 such goals

Making information accessible

Models

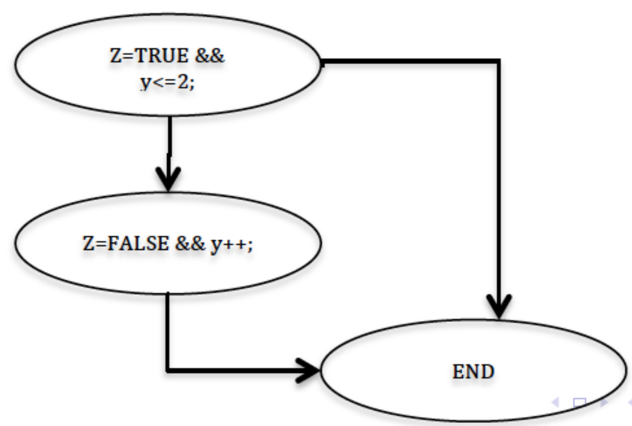
- Models are simpler than the reality as they represent reality with a limited number of factors
- They help factorise the reality and perform program analysis and testing simply and fast
- They highlight specific characteristics of the SUT

Example - Control Flow Graphs

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

```
boolean z = FALSE;
if(z && y<=2){
  z=FALSE;
  y++;
}
```

Some paths are infeasible



Feedback

Apply lessons learned from experience in process improvement and techniques

Examples - learning from experience

- Development projects provide 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