Verification and Validation

Barbara Russo

SwSE - Software and Systems Engineering research group

Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Motivation

- Software and systems are imperfect as they are created by human beings
- We need to ensure a certain degree of quality of the final product/system

- This is the goal of V&V

Verification - how

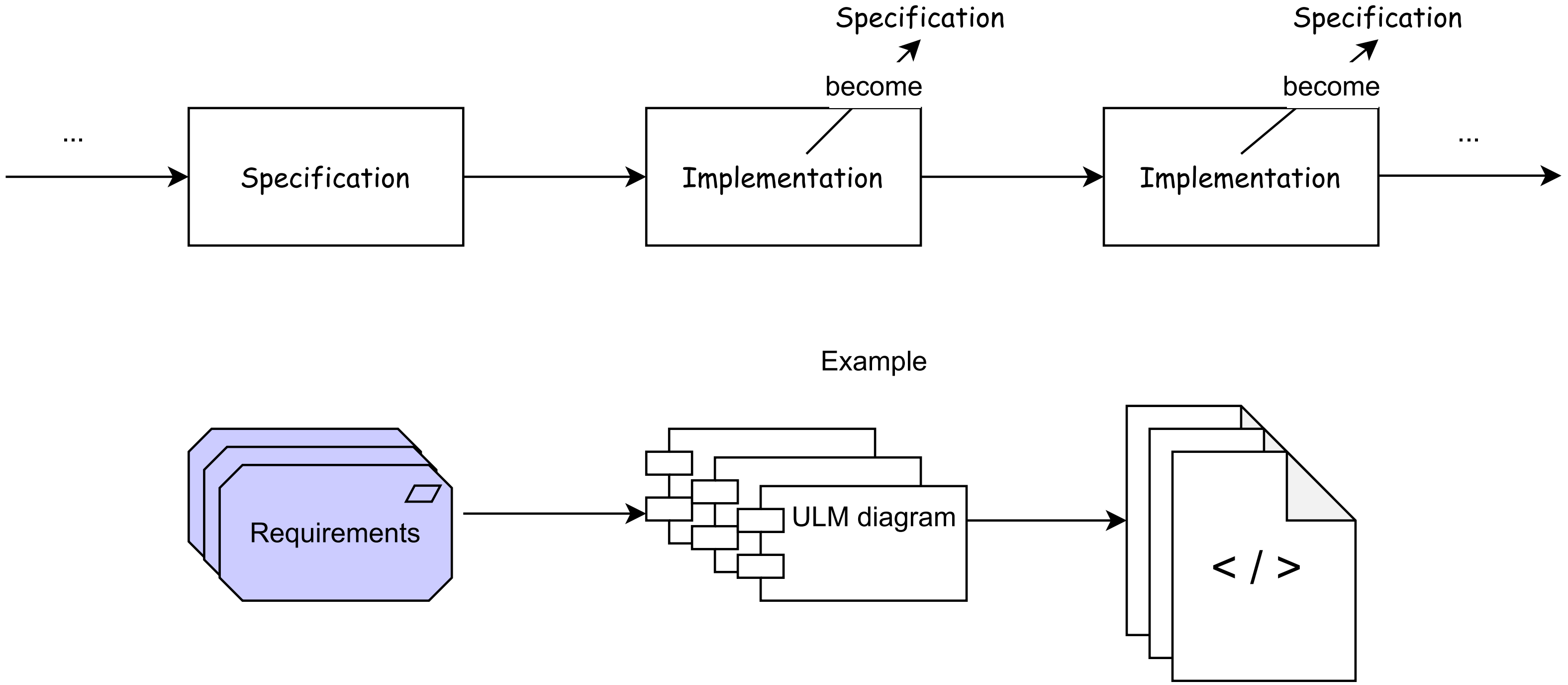
- It a process whose goal is to check the **consistency of an implementation with a specification**
- "How": check the quality of building processes
- Are we building the product right? (B. Boehm)
- Example: A music player plays (it does play) the music when I press Play

Verification

- **Check consistency between two descriptions (roles) of the system** at subsequent stages of the development process

- Examples
  - UML class diagram and its code implementation
  - Specification document and UML class diagram
  - …

Chain of Two Roles



Validation - What

- **Check the degree at which a software system fulfills user's requirements**
- "**What**": checks the product itself
- Are we building the right product ? (B. Boehm)
- <u>Example:</u> A music player plays a song (it does not show a video) when I press Play

Usefulness vs. dependability

- *Requirements are goals* of a system
- *Specifications are solutions* to achieve requirements
  - System that matches requirements ⇒ **useful system**
  - System that matches specifications ⇒ **dependable system**

Usefulness vs. dependability

- *Requirements are goals* of a system
- *Specifications are solutions* to achieve requirements
  - Validation
  - System that matches requirements ⇒ **useful system**
  - Verification
  - System that matches specifications ⇒ **dependable system**

Usefulness vs. dependability

- *Requirements are goals* of a system
- *Specifications are solutions* to achieve requirements
  - System that matches requirements ⟹ **useful system**

    Validation
  - System that matches specifications ⟹ **dependable system**

    Verification

Dependability

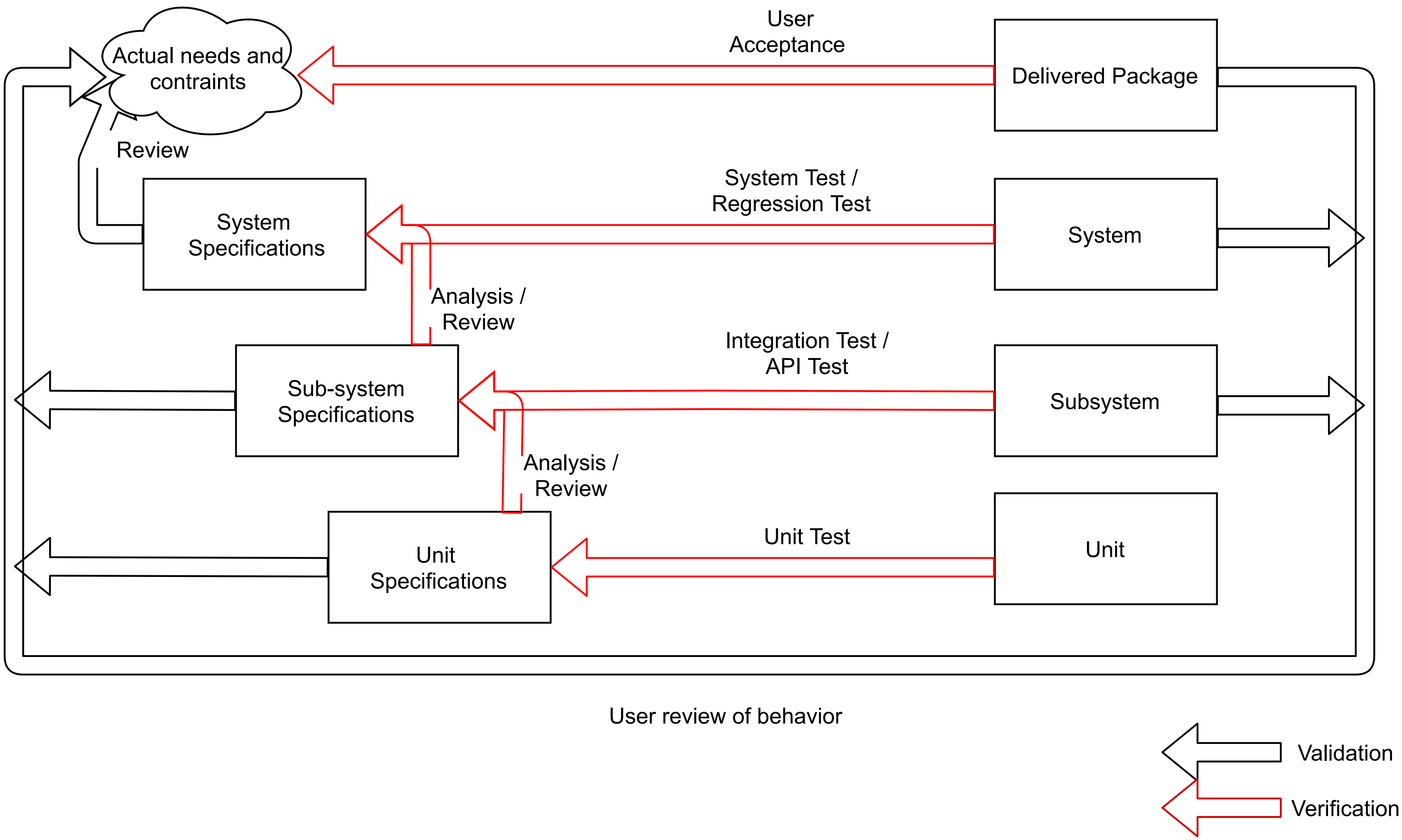
- **Degree at which a system complies with its specifications**

- Specifications are prone to defects as they have been written by human beings, but
    - Verification does not question the correctness of the specifications

Specification Self-consistency

- A verification technique assumes specification self-consistency
- **Consistency**: Specification vs specification, no conflicts
- **No ambiguity**: open to interpretations, uncertainty
- **Adherence to standards**: consistency with benchmarks

Verification & Validation activities



Verification vs. Validation

- Validation involves stakeholders' judgment
- Exercise: Discuss acceptance testing as validation technique

Examples of validation techniques

- **Acceptance testing:** customers verify and validate user stories (requirements)
- **alpha testing**: performed by users in a controlled environment. Capture operational profiles decided by the organisation
- **beta testing**: performed by users in a their own environment. Capture real operational profiles

Verification

Verification manly focuses on dependability and concerns four software/system properties

Dependability properties

- **Correctness**: consistency with specification
- **Reliability**: statistical approximation to correctness; probability that a system deviates from the expected behavior
- **Robustness:** being able to maintain operations under exceptional circumstances of not full-functionality
- **Safety:** robustness in case of hazardous behaviour (e.g., attacks)

Checking dependability

- How can we check whether our software satisfies any of the dependability properties?
- For example, correctness: given a set of specifications and a program we want to find some logical procedure (e.g., a proof) to say that the program satisfies the specifications

Undecidability of problems

Some problems cannot be solved by any computer program (Alan Turing)

The halting problem

Given a program P and an input I, it is not decidable whether P will eventually halt when it runs with that input I or it runs forever

Undecidability of program verification

- Given a program **P** and a verification technique **T** for a dependability property **Q**, we do not know whether the technique can verify the program in finite time
- ... and even when checking is feasible it might be very expensive

Inaccuracy of verification techniques

- Thus, **verification techniques are inaccurate** when checking dependability properties

- They can have **optimistic and pessimistic inaccuracy**

Optimistic Inaccuracy

- Technique that verifies a property Q can return **TRUE on programs P that do not have the property (FALSE POSITIVE)**

Example

- Testing is an *optimistic* verification technique for *correctness*
- It returns that a program is correct even if no finite number of tests can guarantee correctness

Pessimistic Inaccuracy

- Technique that verifies a property Q **can return FALSE on programs P that have the property (FALSE NEGATIVE)**
- Also called *conservative* technique

Example

- *Automatic testing* is pessimistic for *correctness* as it typically runs on rules
  - Software that does not adhere to such rules is not correct
- This can be extended to other techniques that are defined on rules (expert systems)

Accuracy: confusion matrix

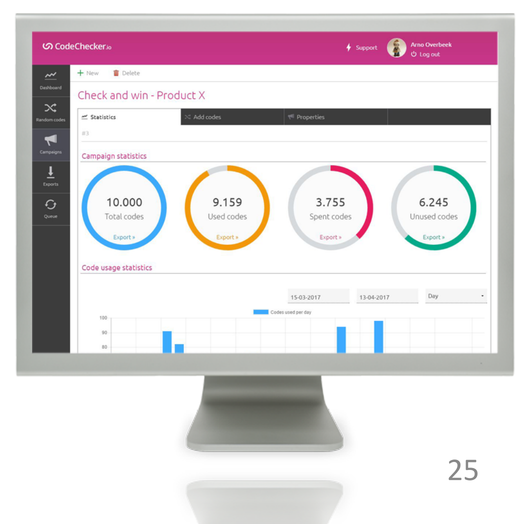
Predicted by the technique

Truth

	Pred. TRUE	*Pred. FALSE*
TRUE	*TP*	*FN*
FALSE	*FP*	*TN*

Careful!

- Being positive or negative depends on the goal of the verification activity: Carefully define what is positive!
- Example: **Unreachable code is dead code**?
- A *code checker that alerts programmers to cases of bad programming style*
- Positives: reachable code



Exercise

- Formulate negatives, false positives and false negatives
- Discuss optimistic or pessimistic accuracy of the *code checker*

PROVABLE TRUE and TRUE

- First-order logic description:

- $\models$ **P** : for program P the verification with T of property Q is **TRUE**
- $\vdash$ **P** : for program P the verification with T of property Q is **provable TRUE** or the verification technique T for Q reports TRUE on P or detects P as TRUE

Completeness for dependability

- If P has a dependability property Q ($\models$ P i.e., P has property Q), then a verification technique T reports true on P for the property Q ($\vdash$P i.e., P is verifiable with T for Q);

$$\models P \Rightarrow \vdash P$$

FN=0

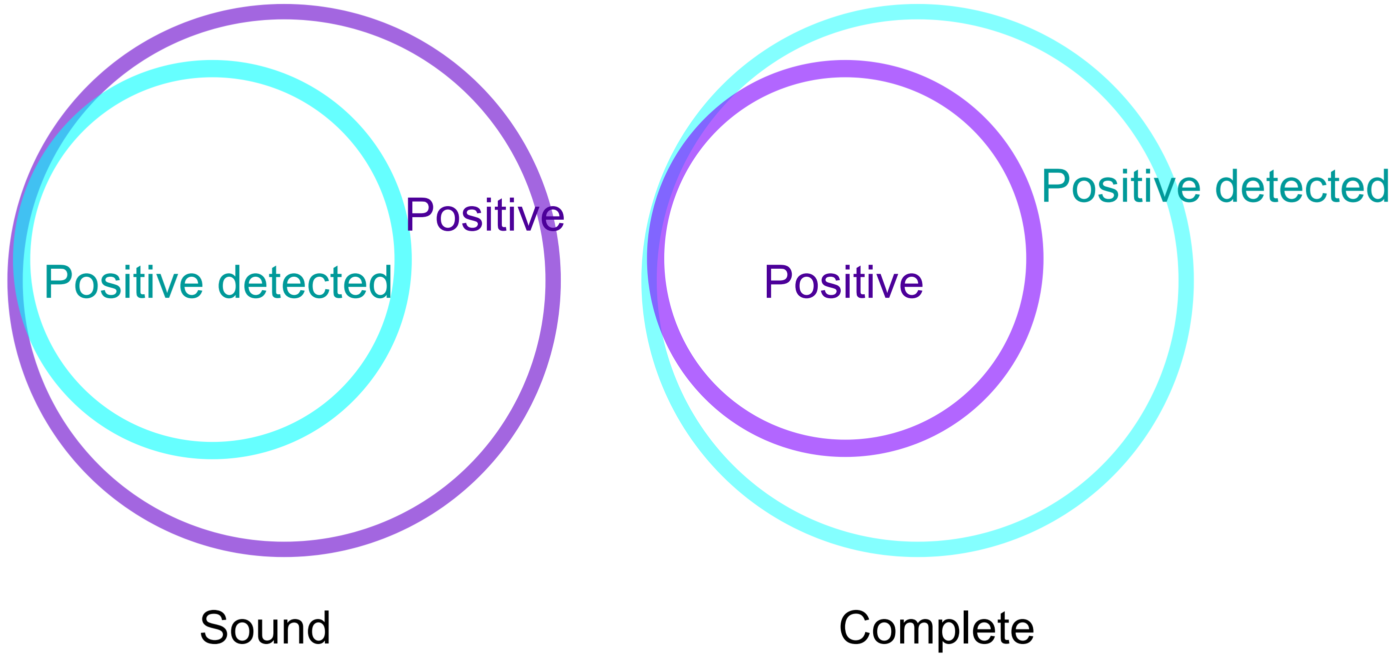
Soundness for dependability

- If a verification technique T reports true on a program P for a dependability property Q ($\vdash$P), then P has the property Q ($\models$ P)

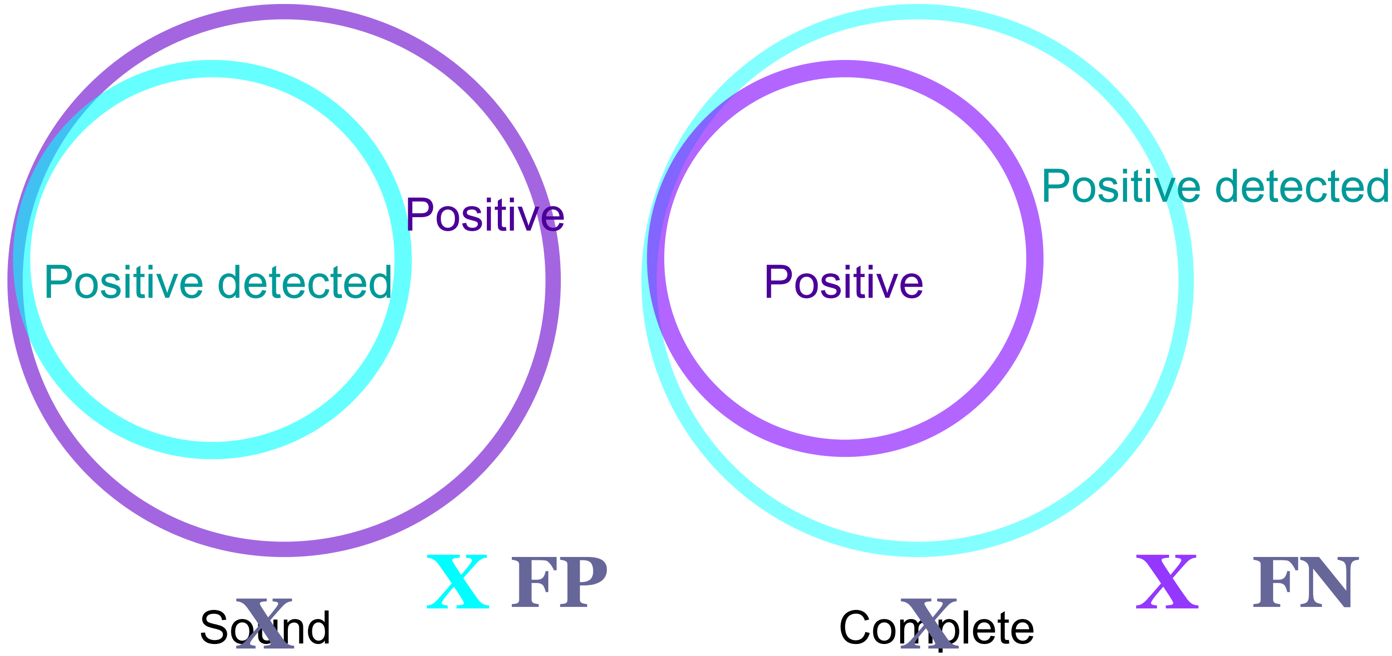
$$\vdash P \Rightarrow \models P$$

FP=0

Sound vs Complete



Sound vs Complete



Exercise

If a verification technique wrongly determines that some reachable code is unreachable, is it unsound or incomplete?

Solution

- It depends on the verification's goal mandate:
- If it is a *code checker that alerts programmers to cases of bad programming style*
- Positives: reachable code

Solution

- It is **complete**: all reachable codes are detected reachable; FN=0
- It is **sound**: all detected reachable codes are reachable; FP=0
- It is **incomplete**: it detects reachable code as unreachable (FN>0)
- It is **unsound**: it detects unreachable code as reachable (FP>0)

Example

- Rephrase (un)soundness and (in)completeness for a code checker

Solution - interpretation

- **Incomplete**: the code checker detects bad style where there is not: **waste of time and resources** to check code detected unreachable which is in fact reachable
- **Unsound**: **the code checker does not alarm** developers on bad code (unreachable): poor quality of the code

Cont. solution - exercise at slide 31

- A *dead-code-removal algorithm of an optimizing compiler*, which aims at removing unreachable code
- Positives: unreachable

Solution

- It is **unsound:** the compiler will remove code that it should not
- It is **incomplete**: unreachable code is detected reachable by depriving the compiler of an optimization
- Give a definition of soundness and completeness in this case

Note

- Optimistic = unsound
- Pessimistic = incomplete

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
- Examples:
  - Constraining the class of programs to verify
  - Separate human judgment from objective verification
  - Exploiting programming language’s feature: serialization

Example - correctness

- “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
  - <u>Substitution principle:</u> 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
  ```

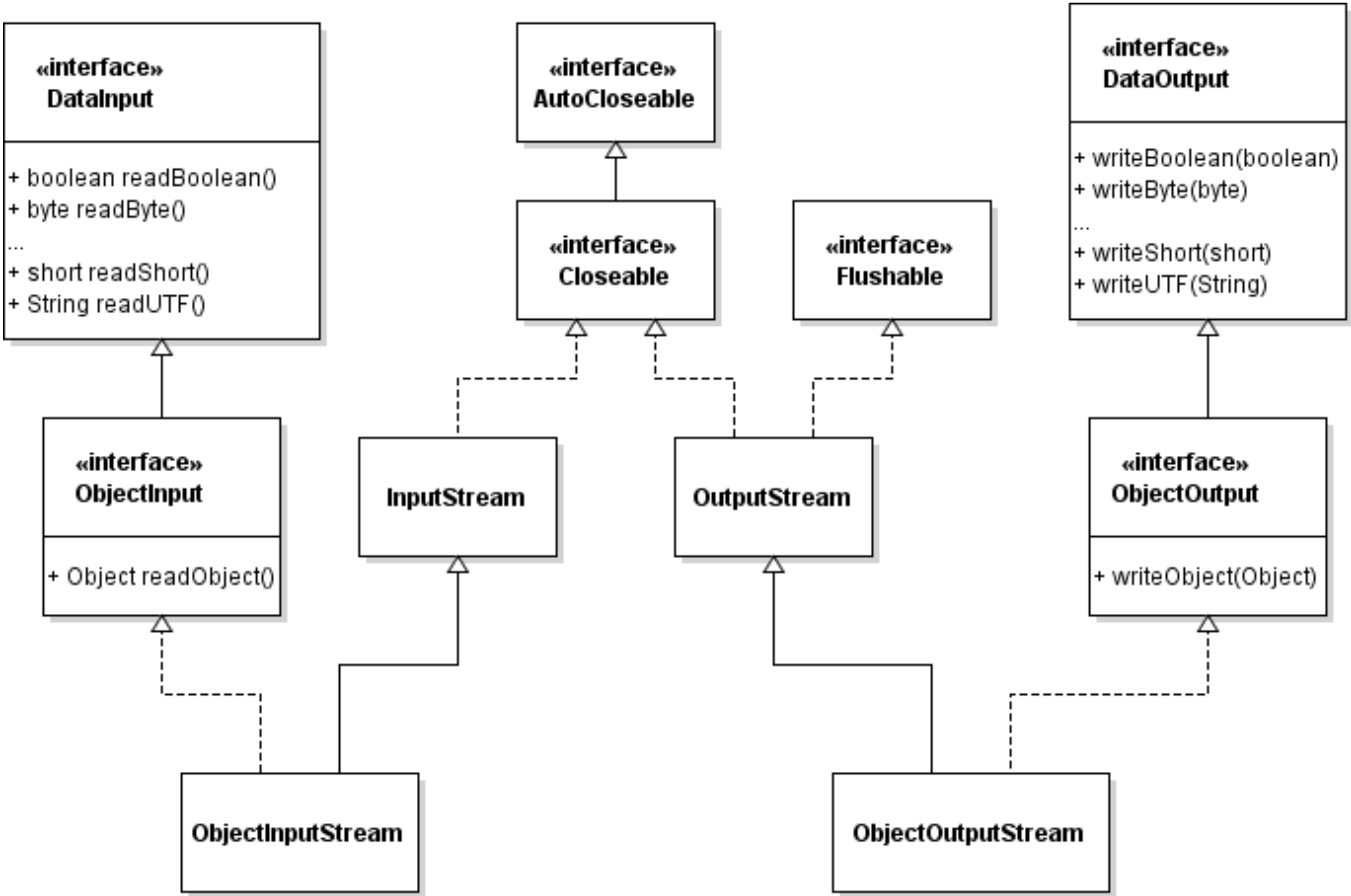
- The method serialises an Object and sends it to the output stream

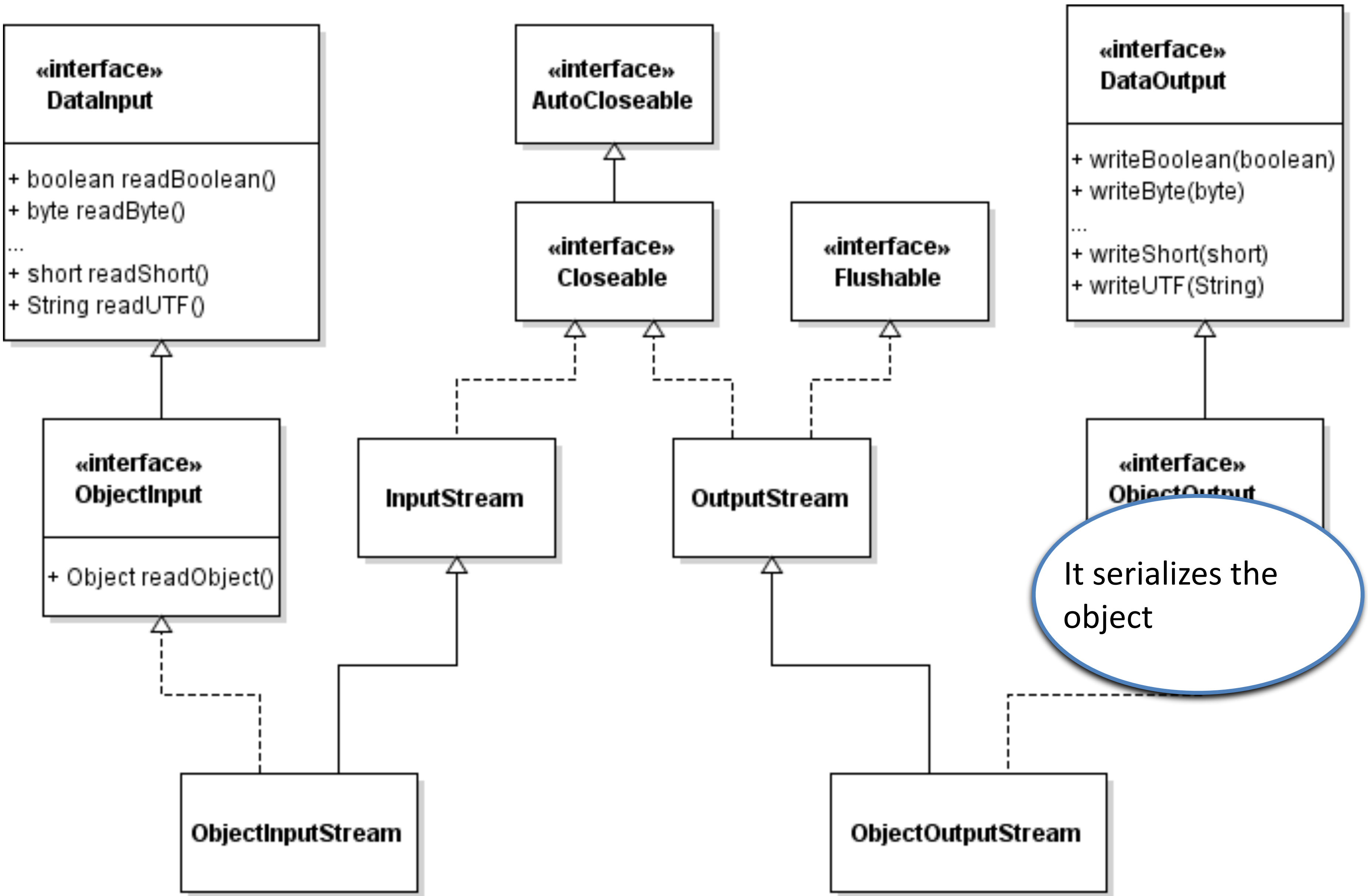
Java object serialisation

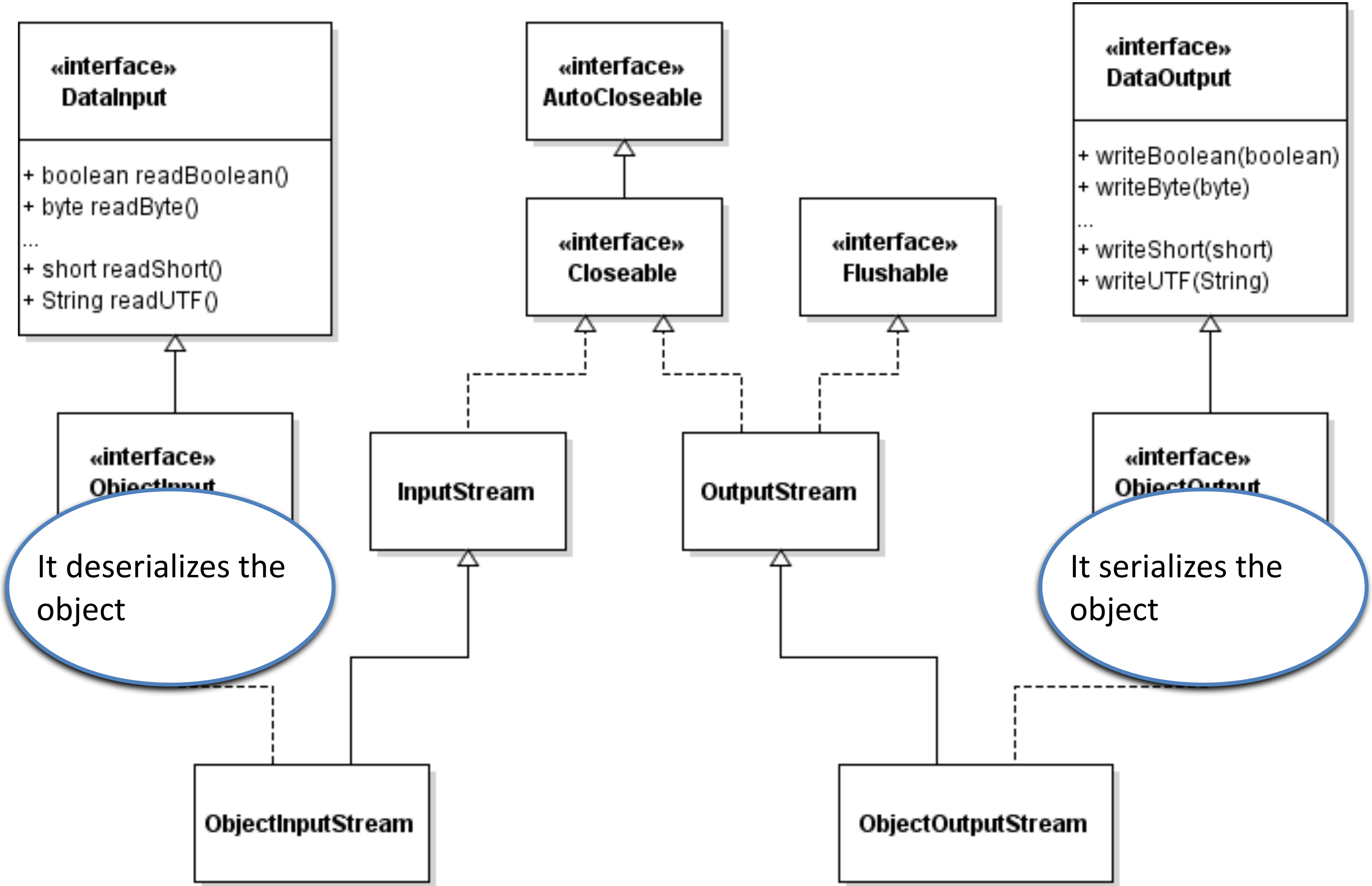
- Similarly, the ObjectInputStream class contains the method for deserialising an object:

```
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)
InputStream
OutputStream
It deserializes the object
It serializes the object
ObjectInputStream
ObjectOutputStream