

5. Specification and Checking

Overview:

- Motivation and problems
- Specifying classes
 - Invariants
 - Pre- and post-conditions
 - Frame properties
 - Model-based specification
- Behavioral subtyping
 - Relation between specification & implementation
 - Concrete pre-post specifications
 - Abstract pre-post specifications
 - Treatment of invariants
 - Treatment of frame properties
 - Using the techniques together

Learning objectives:

- Specification and checking of properties
- Application of specification and checking techniques
- Understanding behavior
- Basis for behavioral subtyping and substitutability

5.1 Motivation and Problems

Motivation of specification:

- Documenting implementation aspects (e.g. data structure invariants):
 - How does an object behave internally?
- As *Contract* between user and provider of code:
 - How does an object interact with the environment?
- Formulating the relevant behavior, in particular:
 - When can an object be substituted by another object?

Specification with assertions:

a. Simple property:

```
...  
C cobj = new C (...);  
assert cobj.x != null ;
```

b. Loop invariant (Schleifeninvariante):

```
public static int isqrt( int y ) {
    int count = 0, sum = 1;
    while (sum <= y) {
        count++;
        sum += 2 * count + 1;
        assert count*count <= y
            && sum==(count+1)*(count+1);
    }
    return count;
}
```

c. More complex property for an AWT-fragment:

```
...
Container c;
Button b;
...
c.remove(b);
assert !EX Container cex: EX int i:
    cex.getComponents()[i] == b;
```

Pre-post specifications for methods:

```
public class IntMathOps {  
  
    /*@ public normal_behavior  
       @ requires      y >= 0  
       @ modifiable  \nothing  
       @ ensures      \result*\result <= y  
       @      &&  y < (Math.abs(\result)+1)  
       @              *(Math.abs(\result)+1);  
       @*/  
    public static int isqrt(int y) { ... }  
}
```

Class invariants:

```
public class List {  
    int length;  
    ListElems le;  
    //@ invariant length == le.leng();  
    ...  
}
```

Further kinds of functional properties:

- Event specifications:
 - occurrence of exceptions
 - modifications of variables
 - invocation of methods
- Termination
- History constraints:
 - relations between state pairs (S1,S2), where S1 happens before S2
- Temporal properties:
 - Is something true until an event happens?
 - Will something eventually happen?

Typical non-functional properties of programs:

- Resource consumption:
 - execution time
 - memory
 - I/O, network resources
- Dependency on the environment (e.g. file system)
- Robustness

Specification problems/challenges:

Examples: (Specification problems)

1. Information hiding: How can behavior based on private attributes be specified?

```
public class WhatCouldBeMySpec {
    private int a = 0;
    public void set( int p ) { a = p; }
    public int get() { return a; }
}
```

2. How are effects to the environment specified?

```
public class EnvironmentalEffects {
    public void localonly( Object mo ) {
        do_something_good();
        if( mo instanceof Atmosphere )
            ((Atmosphere) mo).pollute();
    }
}
```

3. Behavioral conformance/substitutability:

```
public class Superclass
```

```
{ public int a = 0;
```

```
    public void dincrA() { a++; }
```

```
}
```

```
public class Subclass extends Superclass
```

```
{ public void dincrA() { a--; }
```

```
}
```



Approaches to formulate specifications:

- Specification using informal language
- Specification using the means of the programming language (e.g. assert statements)
- Specification using an annotation language that is executable (we illustrate the use of JML)
- Specification with more abstract declarative language constructs

Questions:

- Which terms/functions/etc. can be used to formulate properties?
- Information hiding: Private and protected parts should not appear in public specification.
- What is the meaning of specification constructs that go beyond the programming languages?

5.2 Specifying Classes

Overview:

We consider specifications written in the Java Modeling Language (JML):

- Lightweight specifications
- Direct specifications
- Specifications with abstraction

5.2.1 Lightweight specifications

Explanation: (lightweight specification)

Lightweight (*leichte*) specifications describe certain properties of objects and methods without aiming to provide all details or a complete description of the method/class behavior.

Example: (lightweight specification)

1. Lightweight specifications help in documenting properties that cannot be expressed by the type system.

```
public class Bag {  
  
    //@ requires input != null;  
    public Bag(int[] input) {  
        n = input.length;  
        a = new int[n];  
        System.arraycopy(input, 0, a, 0, n);  
    }  
    ...  
}
```

2. Lightweight specifications can be used for annotating private program elements. This can increase readability and checkability of programs:

```
public final
class String implements ... {
    /** Used for character storage */
    //@ private invariant value != null ;
    private char value[];

    /** First index of the storage used */
    //@ private invariant offset >= 0
    private int offset;

    /** Number of characters in String */
    //@ private invariant count >= 0 ;
    private int count;

    /*@ private invariant
       @   offset + count <= value.length ;
       @*/
    ...
}
```



5.2.2 Direct specifications of objects

Assumptions:

- Specification only depends on the state of one object.
- Relevant attributes are visible to users.
- Methods only modify the target object of the invocation.
- No subtyping, no inheritance.

Specify:

- the possible states of an object X by invariants formulated over the attribute values of X;
- the method behavior by describing
 - the result value
 - the modification of the attribute values
- environmental properties by guaranteeing that attribute values of other objects are not modified.

Specification techniques are explained along with the following running example:

```
public class Point {
    /** Koordinaten */
    /*@ spec_public @*/ private float x, y;

    private float dist;
    ...
}
```

Invariants

Invariants specify range constraints of attributes and relations between attribute values:

Example: (Invariants)

```
public class Point {
    ...
    /*@ public invariant x >= 0.0 && y >= 0.0;

    /*@ private invariant
        @   dist == Math.sqrt(x*x+y*y) ;
        @*/
    ...
}
```



Meaning:

Invariants have to hold:

- in the initial state of the program
- in pre- and poststates of methods for all allocated objects (notice: the set of allocated objects may be different in pre- and poststate)
- in prestates of constructors for all allocated objects except for the newly created one;
in poststates for all allocated objects

Behavior of constructors and methods:

Constructors have to establish the invariants.
Methods must maintain the invariants.

The precondition for their application as well as the caused changes and the result are specified by *requires* and *ensures clauses*.

A complete specification should address exceptional behavior as well.

Example: (Constructor/method specs.)

```
public class Point {
    ...

    /** Creation of a Point */
    /**@ public normal_behavior
        @ requires x >= 0.0 && y >= 0.0 ;
        @ ensures this.x == x && this.y == y;
        @ also
        @ public exceptional_behavior
        @ requires x < 0.0 || y < 0.0 ;
        @ signals (IllegalArgumentException)
    @*/
    public Point ( float x, float y ) { ... }

    /**@ public normal_behavior
        @ ensures \result == this.x ;
    @*/
    public float getX() { ... }

    /** distance from origin */
    /**@ public normal_behavior
        @ ensures \result == Math.sqrt(x*x+y*y);
    @*/
    public float distance() { ... }

    ...
}
```

```

...

/** move this Point */
/*@ public normal_behavior
   @ requires x+dx >= 0.0 && y+dy >= 0.0 ;
   @ ensures   x == \old(x)+dx
   @           && y == \old(y)+dy ;
   @ also
   @ public exceptional_behavior
   @ requires x+dx < 0.0 || y+dy < 0.0 ;
   @ signals (IllegalArgumentException)
   @*/
public void move( float dx, float dy ){ ...}
}

```



Remark:

- The specification above does not give a complete account of what is modified, i.e. it is incomplete (e.g. w.r.t. attribute `dist` and method `move`).
- It does not specify what is left unchanged.
- It has no implementation that satisfies the JML specification (see below).



Specification of frame properties:

Frame specifications list the variables to which a method may assign values. This way, they specify which variables remain unchanged.

Reason for this indirect description technique:

- The number of assignable variables is smaller.
- Frame specification should as well apply to variables that are not visible or known at a method declaration.

Properties that do not *depend* on assignable variables remain unchanged. Such properties of the environment of a method execution are called **frame properties**.

Specification construct:

Modifies/assignable clause (*Veränderungsklausel*):

`assignable` <list of conditional variable expressions>

(The keyword `modifies` is also used.)

Example: (Assignable clause)

```
assignable  x, this.a, p.a.b
```

```
assignable  x, this.a if(this==p)
```



Meaning:

Methods and constructors may only assign to instance and class variables that

- are listed in the assignable clause such that the given condition (if any) is satisfied in the prestate or
- are allocated during execution of constructor or method.

If no assignable clause is given, the meaning is the same as with an empty variable list.

Example:

```
public class Point {
    ...
    /*@ public normal_behavior
       @ requires    x >= 0.0 && y >= 0.0 ;
       @ assignable this.x, this.y ;
       @ ensures    this.x == x && this.y == y;
       @ also
       @ private normal_behavior
       @ requires    x >= 0.0 && y >= 0.0 ;
       @ assignable dist ;
       @ also
       @ public exceptional_behavior
       @ requires    x < 0.0 || y < 0.0 ;
       @ signals    (IllegalArgumentException)
    @*/
    public Point ( float x, float y ) { ...}
```

```

...
/*@ public normal_behavior
   @ ensures \result == this.x ;
   @*/
public float getX() { ... }
...
/*@ public normal_behavior
   @ requires x+dx >= 0.0 && y+dy >= 0.0;
   @ assignable x, y ;
   @ ensures x == \old(x)+dx
   @          && y == \old(y)+dy ;
   @ also
   @ private normal_behavior
   @ requires x+dx >= 0.0 && y+dy >= 0.0;
   @ assignable dist ;
   @ also
   @ public exceptional_behavior
   @ requires x+dx < 0.0 || y+dy < 0.0 ;
   @ signals (IllegalArgumentException)
   @*/
public void move( float dx, float dy ){ ...}
}

```

Remark:

Private assignable clauses can be used for internal checks.

5.2.3 Specification with abstraction

Specification frameworks should provide the possibility to express properties *without referring to the implementation*.

Implementation-independent properties can be expressed by referring to a model of the system.

We consider abstract/model variables. They are for example needed, if

- concrete attributes are not declared (interfaces) or
- access rules do not allow to refer to concrete attributes in specifications (e.g. private attributes)
- implementation independency should be achieved, e.g. to support modifications in implementations.

Syntax in JML:

Abstract/model variables/attributes are declared in JML like concrete attributes prefixed with the keyword `model` as modifier.

In JML, model variables are of a Java-type.

Example: (Model attributes)

```
public class Point {
    /** Koordinaten */
    //@ public model float x, y;
    //@ public invariant x>=0.0 && y>=0.0;

    private double dist, angle ;
    /*@ private invariant dist >= 0.0
       @ && 0.0<=angle && angle<=Math.PI/2;
       @*/

    //@ private depends x <- dist, angle ;
    //@ private depends y <- dist, angle ;

    /*@ private represents x
       @ <- Math.cos(angle)*dist ;
       @*/
    /*@ private represents y
       @ <- Math.sin(angle)*dist ;
       @*/
    ...
}
```



Meaning:

Model attributes *can depend* on concrete attributes or other model attributes.

The dependency has to be specified in a *depends clause (Abhängigkeitsklausel)*.

The value of a model attribute may only depend on the values of attributes for which a dependency is declared.

A *representation clause (Repräsentationsklausel)* may define how the abstract value is computed. It links abstract specification and implementation.

Remark:

- Abstract descriptions are a fundamental issue in specification and a prerequisite for scalability.
- If specification A is more abstract than B, we also say that B *refines* or *specializes* A (is a *refinement/specialization* of A).
- Different kinds of abstraction:
 - w.r.t. level of implementation detail and non-determinism (vs. *refinement*)
 - w.r.t. provided functionality (vs. *specialization*)



Model attributes and frame properties:

If a model attribute a appears in the assignable clause of method m , m may assign to all attributes on which a depends.

Example: (Assignable model attributes)

The constructor `Point` may modify *dist* and *angle*:

```
public class Point {
    ...
    /*@ public normal_behavior
       @ requires x >= 0.0 && y >= 0.0 ;
       @ assignable this.x, this.y ;
       @ ensures this.x == x && this.y == y;
       @ also
       @ public exceptional_behavior
       @ requires x < 0.0 || y < 0.0 ;
       @ signals (IllegalArgumentException)
    @*/
    public Point ( float x, float y ) { ...}

    /*@ public normal_behavior
       @ ensures \result == this.x ;
    @*/
    public float getX() { ... }
    ...
}
```

```

...
/*@ public normal_behavior
   @ requires x+dx >= 0.0 && y+dy >= 0.0;
   @ assignable x, y ;
   @ ensures x == \old(x)+dx
   @           && y == \old(y)+dy ;
   @ also
   @ public exceptional_behavior
   @ requires x+dx < 0.0 || y+dy < 0.0 ;
   @ signals (IllegalArgumentException)
   @*/
public void move( float dx, float dy ){ ...}
}

```



Remark:

- The specification that *dist* may be modified is not necessary, because *x* and *y* depend on *dist*.
- Requiring equality within the ensures clause is problematic.



The following example demonstrates an abstract specification of a type `UnboundedStack` for which no attributes are declared.

The specification is based on a side-effect free type `JMLObjectSequence` that is predefined in JML:

„Most specifications need existing knowledge!“

Example: (Specifying types that have no concrete attributes)

```
package org.jmlspecs.models;

public /*@ pure @*/ class JMLObjectSequence
    implements JMLCollection
{
    public /*@ pure @*/
    boolean equals(Object obj) { ... }

    public /*@ pure @*/
    boolean isEmpty() { ... }

    public /*@ pure @*/
    Object first()
        throws JMLSequenceException { ... }

    public /*@ pure @*/
    /*@ non_null @*/ JMLObjectSequence
    insertFront(Object item)
        throws IllegalStateException { ... }
    ...
}
```

Pure methods must not have side-effects (assignable \nothing).

Pure constructors may only modify the attributes of the created object.

Class- and interface types are *pure*, if they only contain pure methods and constructors.


```

//@ model import org.jmlspecs.models.*;

public abstract class UnboundedStack {
  /*@ public model JMLObjectSequence theStack
    @      initially theStack != null
    @      && theStack.isEmpty();
  @*/
  //@ public invariant theStack != null;

  /*@ public normal_behavior
    @   requires   !theStack.isEmpty();
    @   assignable theStack;
    @   ensures    theStack.equals(
                \old(theStack.trailer()));
  @*/
  public abstract void pop( );

  /*@ public normal_behavior
    @   assignable theStack;
    @   ensures    theStack.equals(
                \old(theStack.insertFront(x)));
  @*/
  public abstract void push(Object x);

  /*@ public normal_behavior
    @   requires !theStack.isEmpty();
    @   ensures  \result == theStack.first();
  @*/
  public abstract Object top( );
}

```



Summarizing remarks:

- Specifications can build on implementation parts. They can be used for documentation and checking code.
- Specification techniques can support implementation independent formulations:
 - Enables to exchange implementation without change of specification
 - Allows design contract prior to implementation
 - Needs other types to express properties.
- Well-understood specification techniques for object-based programs:
 - class invariant
 - requires/ensures specifications
 - modifies clauses: Specify what is not modified by describing what is allowed to be modified

Are these techniques applicable and sufficient to master all aspects of object-oriented programs?



5.3 Behavioral Subtyping

Subtyping of programming languages enforces that

- no type errors occur, and
- there is a method implementation for each method invocation.

It does not guarantee that subtype objects **behave** like supertype objects.

Example: (no specialization)

D-objects do not behave similar to C-objects:

```
class C {
    int a;
    int getA(){ return a; }
    void incr() { a++; }
}

class D extends C {
    int getA(){ return a+2; }
    void incr() {
        throw new NullPointerException();
    }
}
```



Problem:

What does it mean that

- a subtype behaves like the supertype or
- is a specialization of the supertype?

In particular, how do we handle

- abstract types/classes?
- extensions of the state space?

The operational semantics provides a bad basis for defining behavioral subtyping.

Approach:

Define the behavioral subtype relation based on **specified** properties:

- Each subtype object has to satisfy the specification of the supertype.

For pre- and post-specifications this means:

- $\text{pre}[\text{Supertype}] \Rightarrow \text{pre}[\text{Subtype}]$
- $\text{post}[\text{Subtype}] \Rightarrow \text{post}[\text{Supertype}]$

Explanation: (Behavioral subtyping)

A subtype S is called a *behavioral* subtype of T iff S -objects behave according to the specification of T (auf Deutsch: S ist ein *konformer* Subtyp von T).



Remark:

- Behavioral subtyping depends on the **specification**.
- Main goal is to understand the relation between sub- and supertypes.
- The described techniques should be applied to informal specifications/documentation as well.



Kinds of Specialization:

- Refining implementation
- Refining nondeterminism
- Adding methods and extending state

Overview:

- Relation between specifications & implementations
- Pre-post specifications without abstract variables and without abstraction (*concrete specs*)
- Pre-post specifications with abstract variables and with abstraction (*abstract specs*)
- Treatment of invariants
- Treatment of frame properties
- Illustrating example

Relation between specification & implementation

Instead of relating an implementation of class D to the specifications of all its supertypes, we check that:

- the specification of D, $\text{spec}(D)$, is a behavioral subtype of its direct supertype,
- the implementation of D, $\text{impl}(D)$, satisfies $\text{spec}(D)$.

Illustration:

```
class C {
  pre  PMC(this,p)
  post QMC(this,p,result)
  R m( T p ) { .../* impl(m,C) */ }
}
```

```
class D extends C {
  pre  PMD(this,p)
  post QMD(this,p,result)
  R m( T p ) { .../* impl(m,D) */ }
}
```

Check:

- $\{ \text{PMD}(\text{this},p) \} \text{impl}(m,D) \{ \text{QMD}(\text{this},p,\text{result}) \}$
- $\text{PMC}(\text{this},p) \Rightarrow \text{PMD}(\text{this},p)$
- $\text{QMD}(\text{this},p,\text{result}) \Rightarrow \text{QMC}(\text{this},p,\text{result})$



Remark:

Specification languages usually support *specification inheritance* in a form that establish (b) and (c) or refined versions of it automatically. ■

Example: (Spec. inheritance in JML)

```
class C {
    /*@ public normal_behavior
       @ requires PREMC
       @ ensures POSTMC ; @*/
    R m( T p ){ ... }
}

class D extends C {
    /*@ also
       @ public normal_behavior
       @ requires PREMD
       @ ensures POSTMD ; @*/
    R m( T p ){ ... }
}
```

The specification of D is an abbreviation for:

```
class D extends C { // not JML
    /*@ public normal_behavior
       @ requires PREMC || PREMD
       @ ensures ( \old(PREMC) => POSTMC )
       @          && ( \old(PREMD) => POSTMD );
       @*/
    R m( T p ){ ... }
}
```

■

Concrete pre-post specifications

Pre- and postconditions of methods in supertypes are valid pre-/postconditions for subtype methods:

-> Inheritance of specification

Subtypes without additional attributes:

Inherited method:

- Specification is inherited without change.

Overriding method:


- Precondition inherited or new weaker precondition
- Postcondition inherited and possibly strengthened by additional properties

Example:

```
class C {
    /*@ public normal_behavior
       @ requires P ;
       @ ensures Q ;
    @*/
    C m() { ... }
}
```



```
class D extends C {
  /*@ also
    @   public normal_behavior
    @   requires P ;
    @   ensures \result instanceof D ;
  @*/
  C m() { ... }
}
```



Additional method:

- no constraints (but see about invariants).

Remark:

Similar restrictions have to be applied to the specification of exceptional behavior.

Subtypes with extended state:

If the state space is *extended* in the subtype, it can be necessary to strengthen the precondition of overriding methods w.r.t. properties concerning the additional attributes. However, this would violate:

$$\text{pre}[\text{Supertype}] \Rightarrow \text{pre}[\text{Subtype}]$$

Solution:

Use abstraction with model variables.

Abstract pre-post-specifications

The basic idea is to write specifications in terms of abstract states and to provide abstraction functions:

- from concrete objects to their abstract states
- from abstract states of subtype objects to abstract states of supertype objects.

The abstraction functions in particular provide the flexibility to handle extended state.

Here, we only consider an example:

Example:

```
class C {
    //@ public model boolean valid;
    //@ public model AS state;

    /*@ public normal_behavior
        @ requires valid && r(state) ;
        @ ensures q(state) ;
    @*/
    void m(){ ... }
}

class D extends C {
    private BD d;

    //@ private depends valid <- d;
    //@ private represents valid <- CD.pd(d) ;

    //@ private depends state <- d;
    //@ private represents state <- CD.f(d) ;
    ...
}

class E extends C {
    private BE e;

    //@ private depends valid <- e;
    //@ private represents valid <- CE.pe(e) ;

    //@ private depends state <- e;
    //@ private represents state <- CE.g(e) ;
    ...
}
```

- p_d is a predicate expressing when attribute d has a valid state.
- f is an abstraction function mapping values of concrete type BD to the abstract type AS .

The definitions of p_d and f can be tailored to the needs of class D . The same holds for p_e, g and class E .



Remarks:

- Often one assumes an explicit abstraction function that maps values of subtype objects into the state space of the supertype.
- Abstract/model variables enable two kinds of state extensions:
 - overriding the representation function
 - additional abstract variables
- The variations in the subtypes can be captured by representation functions.

Treatment of Invariants

In principle, invariants can be expressed by pre- and post-conditions. However, as a specification construct they allow to express

- restrictions on subtype behavior
- already in supertypes:
→ Invariants of supertypes have to be satisfied by additional subtype methods.

Example:

```
class C {  
    public int a = 0;  
    //@ public invariant a >= 0;  
    ...  
    // no declaration of m  
}
```

```
class D extends C {  
    ...  
    void m(){ a = -1; } // violates invariant  
    ...  
}
```



General Approach:

Invariants of supertypes have to be satisfied by all subtype methods. This can be formulated as:

$$\text{inv}[\text{Subtype}] \Rightarrow \text{inv}[\text{Supertype}]$$

This can be achieved by invariant inheritance:

=> The subtype invariant is the conjunction of the supertype invariant and additional invariant clauses specified in the subtype.

Problems:

- What is the precise meaning of an invariant?
When should invariants hold?
- How is subtyping and dynamic binding handled for invariants?

There is no well-established answer to these problems. We discuss existing solutions.

Semantical variation:

1. Invariants have to hold in *visible* states:

JML: „Invariants have to hold in each state outside of a public method’s execution and at the beginning and end of such execution.“

2. Transformation into pre- and postconditions:

Invariants have to hold in poststates of constructors. If they hold in the prestate of a method, they have to hold in the poststate as well.

For verification, both alternatives are problematic:

Let S be a subtype of T , m be a method in T and x a variable of type T holding an S -object:

... $x.m(\dots)$...

For the verification we need $\text{inv}[S]$ as precondition. But: S may not be known at the invocation site.

Example:

```
class C {
    public int a = 0;
    //@ public invariant a >= 0;
    ...
    void m() {
        ... // maintains invariant
    }
}
```

```
class Foo {
    void mfoo( C x ) {
        ... x.m() ...
    }
}
```

```
class D extends C {
    public int b = 0;
    //@ public invariant a > 1;
    //@ public invariant b >= 0;
    ...
    void m() {
        b = 4 / a ;
        ... // maintains both invariants
    }
}
```

Problem: What has to be shown for the prestate of a method invocation `x.m()`?



Classical solution:

- Make sure that invariants can only be broken by methods of the object. Thus:
 - if they are established in the post-state of each call of a constructor and method of the class,
 - then they hold in the pre-state of each call.
- Solution has problems with recursion.

Other solutions:

- Specify the invariants based on abstractions
- Allow the programmer to define where invariants should hold and reflect this in the meaning of invariants (solution of Spec#)
- Use a more sophisticated meaning of invariants

Treatment of frame properties

The specification of frame properties has to cope with two problems:

- Information hiding: not all assignable variables can be named in the specification.
- Extended state: The supertype specification cannot capture the state of additional attributes in subtypes.

Example:

```
class C {
    public int a = 0;
    private int b = 0;
    public static int c = 123;
    ...

    /*@ public normal_behavior
       @ assignable a;
       @*/
    public void m(){ a++; b++ }
}
```

```
class Foo {
    void mfoo( C x ) {
        ... x.m() ...
    }
}
```

```
class D extends C {
    public int d = 0;
    ...
    public void m(){
        super.m();
        d = 87;
        C.c = 4 ;
        ...
    }
}
```



Possible solution:

- Use abstract attributes/variables, depends- and representation clauses.
- Information hiding: Abstract attributes can depend on non-accessible attributes.
- Extended state: Depends relation can be extended in subtypes.

(Example is given in the following subsection)

Using the techniques together

Specification techniques for OO-programs have two goals:

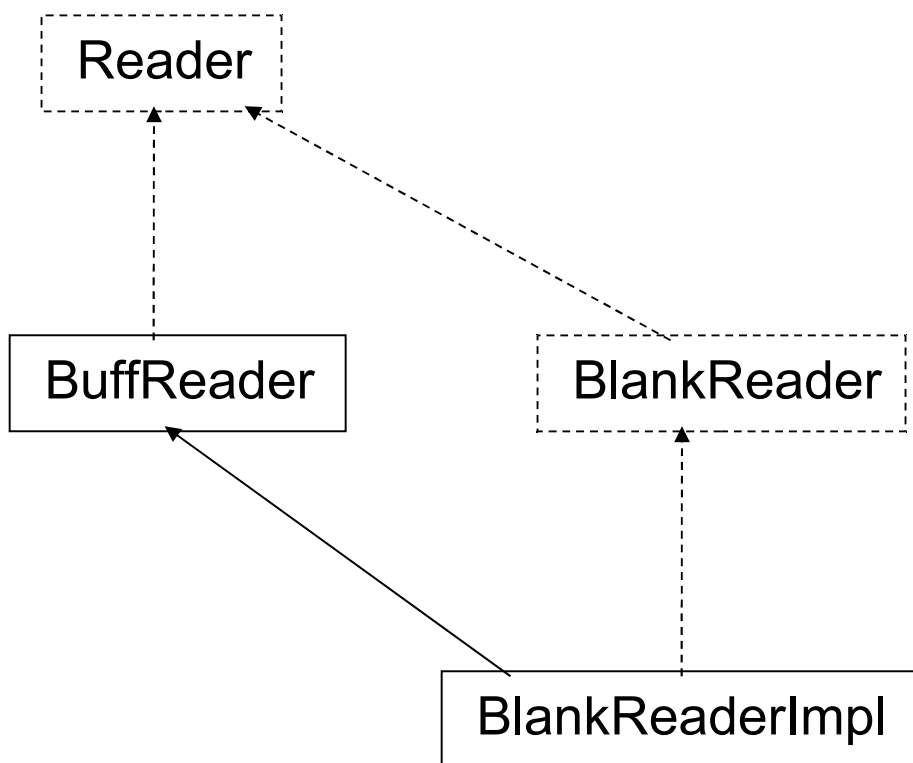
- Specification of properties by annotating programs.
- Complete specification of types as basis for behavioral subtyping.

We illustrate this by a larger example.

Example:

The following example is a Java-version of the central example given in:

K. R. M. Leino, G. Nelson: Data abstraction and information hiding, Transactions on Programming Languages and Systems 24(5): 491-553 (2002).



The example demonstrates:

- state extensions
- behavioral subtyping

```

public interface Reader {
    //@ public model instance boolean valid;
    //@ public model instance Object state;

    /*@ public normal_behavior
        @ requires valid;
        @ assignable state;
        @ ensures -1 <= \result
        @          && \result < 65535 ;
    @*/
    public int getChar();

    /*@ public normal_behavior
        @ requires valid;
        @ assignable valid, state;
    @*/
    public void close();
}

```

```

public abstract
class BufferedReader implements Reader {

    protected /*@ spec_public @*/ int lo, cur, hi;
    protected /*@ spec_public @*/ char[] buff;

    /*@ public model boolean svalid;

    /*@ public represents valid <-
        @ this != null &&
        @ 0 <= lo && lo <= cur && cur <= hi &&
        @ buff != null && hi-lo <= buff.length &&
        @ svalid ;
    @*/

    public int getChar() {
        if( cur == hi ) refill();
        if( cur == hi ) return -1;
        cur++;
        return buff[cur-lo-1];
    }

    /*@ public normal_behavior
        @ requires valid;
        @ assignable state;
        @ ensures cur == \old(cur) ;
    @*/
    public abstract void refill();

    /*@ depends valid <- lo,cur,hi, buff, svalid;
    /*@ depends state <- lo,cur,hi, buff, buff[*];
    /*@ depends svalid <-lo, hi, buff;
}

```

```

public interface BlankReader extends Reader {

    /*@ public normal_behavior
       @ requires      0 <= n;
       @ assignable   valid, state;
       @ ensures      valid  && \result == this ;
    @*/
    public BlankReader init( int n );
}

```

A non-trivial application of BlankReader and BlankReaderImpl (s. next slide):

```

public class ReaderTest {

    public static void main( String[] args ) {
        BlankReader br = new BlankReaderImpl();
        br.init(1000000);

        int count = 0;
        int chr;
        do {
            chr = br.getChar();
            count++;
        } while( chr != -1 );
        br.close();
        System.out.println(count);
    }
}

```

```

public class BlankReaderImpl
            extends BufferedReader
            implements BlankReader
{
    private int num;
    //@ private represents svalid <- hi <= num ;

    public BlankReader init( int n ) {
        num = n;
        buff = new byte[ Math.min(n, 8192) ];
        lo = 0;
        cur = 0;
        hi = buff.length;
        for( int i = 0; i < hi; i++ ) {
            buff[i] = 32;
        }
        return this;
    }

    public void refill() {
        lo = cur;
        hi = Math.min( lo+buff.length, num );
    }

    public void close() {}

    //@ private depends state <- num ;
    //@ private depends svalid <- num ;

    // representation of state not
    // considered
}

```