Summer Term 2018

# Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

A Critical View on Inheritance

1

A smart home has many features that are controlled automatically:
Heating, Lighting, Shutters, …

We want to develop a software that helps us to control our smart home.

---

## A Critical View On Inheritance

Inheritance is the main built-in variability mechanism of OO languages.



2

Common functionality can be implemented by a base class and each variation can be implemented by a separate subclass.

- In the following, we analyze the **strengths** and **deficiencies** of inheritance with respect to supporting variability.
- Many design patterns that we will discuss in the following sections propose solutions to compensate for deficiencies of inheritance.

This section serves as a bridge between the block on design principles and the blocks about design patterns and advanced languages.

## Desired Properties
(of Programming Languages)

- Built-in support for OCP
- Good Modularity
- Support for structural variations
- Variations can be represented in type declarations

3

3

A good support for OCP, **reduces the need to anticipate variations**. Inheritance allows replacing the implementation of arbitrary methods of a base class (unless it is explicitly forbidden, e.g., in Java methods can be declared as `final`).

*Of course, support for variability in a class is conditioned by the granularity of its methods and the abstractions built-in.*

When we achieve good modularity, **the base class can remain free of any variation-specific functionality**; each variation is implemented in a separate subclass.

In general, **inheritance allows to design the most suitable interface for each variation.**
Different variations of a type may need to extend the base interface with variation-specific fields and methods. (In addition to varying the implementation of the inherited base interface.)

The property that variations can be represented in type declarations is necessary for type-safe access of variation-specific interfaces.

---

## Variation of selection functionality of table widgets.

Desired Properties By Example

```
class TableBase extends Widget {
  TableModel model;
  String getCellText(int row, int col){ return model.getCellText(row, col); }
  void paintCell(int r, int c){ getCellText(row, col) … }
}
abstract class TableSel extends TableBase {
  abstract boolean isSelected(int row, int col);
  void paintCell(int row, int col) { if (isSelected(row, col)) … }
}
class TableSingleCellSel extends TableSel {
  int currRow; int currCol;
  void selectCell(int r, int c){ currRow = r; currCol = c; }
  boolean isSelected(int r, int c){ return r == currRow && c == currCol; }
}
class TableSingleRowSel extends TableSel {
  int currRow;
  void selectRow(int row) { currRow = row; }
  boolean isSelected(int r, int c) { return r == currRow; }
}
class TableRowRangeSel extends TableSel { … }
class TableCellRangeSel extends TableSel { … }
```

4

4

The modularization of these variations by inheritance is illustrated by the given (pseudo-)code:

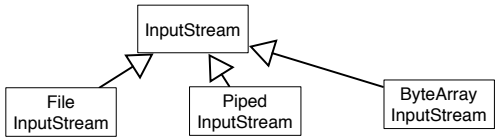
- TableBase implements basic functionality of tables as a variation of common functionality for all widgets, e.g., display of tabular data models.
- The abstract class TableSel extends TableBase with functionality that is common for all types of table selection, e.g., rendering of selected cells.
- TableSingleCellSel, TableSingleRowSel, TableRowRangeSel, and TableCellRangeSel implement specific types of table selections.

# Assessment

- **Built-in support for OCP**: The implementation of paintCell in TableSel can be overridden and we could create further selection models.
- **Good modularity**: Each table selection model is encapsulated in a separate class.
- **Support for structural variations**:
  - Different operations and variables are declared and implemented by TableSingleCellSel and TableSingleRowSel: currRow, currCel, selectCell and currRow, selectRow, respectively.
  - Can design the most suitable interface for each type of table selection.
  - Do not need to design a base interface that fits all future variations.

## Non-Reusable, Hard-to-Compose Extensions

An Extract from Java's Stream Hierarchy




5

5

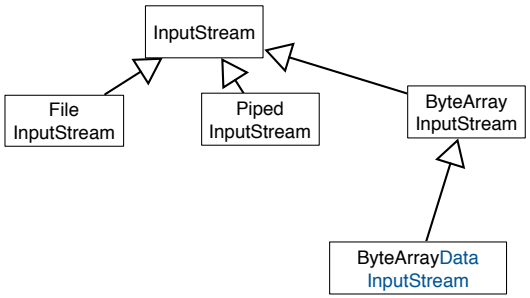
Consider an extract from java.io package that consists of classes for reading from a source. Streams abstract from concrete data sources and sinks:

- `InputStream` is root of stream classes reading from a data source.
- `FileInputStream` implements streams that read from a file.
- `PipedInputStream` implements streams that read from a PipedOutputStream. Typically, a thread reads from a PipedInputStream data written to the corresponding PipedOutputStream by another thread.
- `ByteArrayInputStream` implements streams that read from memory.

---

## Non-Reusable, Hard-to-Compose Extensions

Handling Streams




6

6

Need a variation of `ByteArrayInputStream` capable of reading whole sentences and not just single bytes.

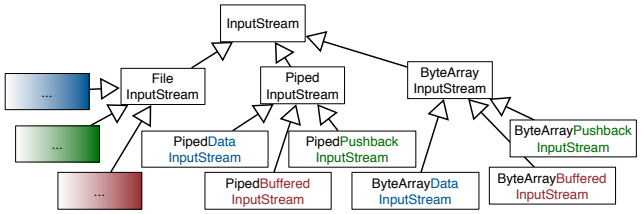
We could implement it as a subclass of `ByteArrayInputStream`. The blue part in the name of the class denotes the delta (`DataInputStream`) needed to implement this variation.

Further Variations that are conceivable:

- Reading whole sentences with other kinds of streams:
  - `FileInputStream` objects that are able to read whole sentences.
  - `PipedInputStream` should read whole sentences too.
  - …
- Writing the given data back ("red" in the following slide)
- Buffering content ("green" in the following slide),
- Counting the numbers of lines processed,
- …

## Non-Reusable, Hard-to-Compose Extensions

Handling Streams




Each kind of variation would have to be re-implemented for all kinds of streams, for all meaningful combinations of variations





# Assessment

The design is complex and suffers from a huge amount of code duplication.

## Non-Reusable, Hard-to-Compose Extensions

Extensions defined in subclasses of a base class cannot be reused with other base classes.

E.g., the *Pushback* related functionality in `FilePushbackInputStream` cannot be reused.





# Result

- Code duplication:
  A particular type of variation needs to be re-implemented for all siblings of a base type which results in code duplication.
  Large number of independent extensions are possible:
  - For every new functionality we want.
  - For every combination of every functionality we want.

- Maintenance nightmare: exponential growth of number of classes.

# Weak Support for Dynamic Variability

Variations supported by an object are fixed at object creation time and cannot be (re-)configured dynamically.

A buffered stream is a buffered stream is a buffered stream… It is not easily possible to turn buffering on/off, if buffering is implemented by means of subclassing.



9

# Dynamic Variability Illustrated

The configuration of an object's implementation may depend on values from the runtime context.

- Potential Solution:
  Mapping from runtime values to classes to be instantiated can be implemented by conditional statements.
  ```
  …if(x) new Y() else new Z() …
  ```
- Issue:
  Such a mapping is error-prone and not extensible. When new variants of the class are introduced, the mapping from configuration variables to classes to instantiate must be changed.

Rigid Design



10

Example

Table widget options may come from some dynamic configuration panel; depending on the configuration options, different compositions of table widget features need to be instantiated.

## Dynamic Variability Illustrated

The configuration of an object's implementation may depend on values from the runtime context.

- Potential Solution:
  Using dependency injection.
- Issue:
  Comprehensibility suffers:
  - Objects are (implicitly) created by the Framework
  - Dependencies are not directly visible/are rather implicit



11

Example

Dependency injection is commonly used by Enterprise Application Frameworks such as Spring, EJB,… or by frameworks such as Google Guice.

## Dynamic Variability Illustrated

The behavior of an object may vary depending on its state or context of use.

- Potential Solution:
  Mapping from runtime values to object behavior can be implemented by conditional statements in the implementation of object's methods.
- Issue:
  Such a mapping is error-prone and not extensible. When new variants of the behavior are introduced, the mapping from dynamic variables to implementations must be changed.



12

Example

An account object's behavior may vary depending on the amount of money available. The behavior of a service then may need to vary depending on the client's capabilities.

# The Fragile Base Class Problem

Cf. Item 17 of Joshua Bloch's, **Effective Java**.





## An Instrumented HashSet

The Fragile Base Class Problem Illustrated

```
import java.util.*;
public class InstrumentedHashSet<E> extends HashSet<E> {
  private int addCount = 0;
  public InstrumentedHashSet() {    }
  public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
  }

  @Override public boolean add(E e) { addCount++; return super.add(e); }
  @Override public boolean addAll(Collection<? extends E> c) {
    addCount += c.size();
    return super.addAll(c);
  }
  public int getAddCount() { return addCount; }

  public static void main(String[] args) {
    InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
    s.addAll(Arrays.asList("aaa", "bbb", "ccc"));
    System.out.println(s.getAddCount());
  }
}
```

Output?





Suppose we want to implement HashSets that know the number of added elements; we implement a class `InstrumentedHashSet` that inherits from `HashSet` and overrides methods that change the state of a `HashSet` …

The answer to the question is **6** because the implementation of `addAll` in `HashSet` internally calls this.add(...). Hence, added elements are counted twice.

## An Instrumented HashSet

The Fragile Base Class Problem Illustrated

```
import java.util.*;
public class InstrumentedHashSet<E> extends HashSet<E> {
  private int addCount = 0;
  public InstrumentedHashSet() {    }
  public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
  }

  @Override public boolean add(E e) { addCount++; return super.add(e); }
  // @Override public boolean addAll(Collection<? extends E> c) {
  //  addCount += c.size();
  //  return super.addAll(c);
  // }
  public int getAddCount() { return addCount; }

  public static void main(String[] args) {
    InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
    s.addAll(Arrays.asList("aaa", "bbb", "ccc"));
    System.out.println(s.getAddCount());
  }
}
```

Does this really(!) solve the problem?

15

15

**Ask yourself: Is the counting problem solved, by not overriding `addAll`?** For the moment, yes. But, not principally.

What if in the future the designers of `HashSet` decide to re-implement `addAll` to insert the elements of the parameter collection as a block rather than by calling add on each element of the collection? Might be necessary for efficiency reasons.

---

## The Fragile Base Class Problem in a Nutshell

Changes in base classes may lead to unforeseen problems in subclasses.

Inheritance Breaks Encapsulation

16

16

**You can modify a base class in a seemingly safe way**. But this modification, when inherited by the derived classes, might cause them to malfunction.

You can't tell whether a base class change is safe simply by examining the base class' methods in isolation. You must look at (and test) all derived classes as well.
You must check all code that uses the base class and its derived classes; this code might also be broken by the changed behavior.

A simple change to a key base class can render an entire program inoperable.

I

## Fragility by **dependencies on the self-call structure**

The Fragile Base Class Problem in a Nutshell

- The fragility considered so far is caused by dependencies on the self-call structure of the base class.
- Subclasses make assumptions about the calling relationship between non-`private` methods of the base class.
- These assumptions are implicitly encoded in the overriding decisions of the subclass.
- If these assumptions are wrong or violated by future changes of the structure of superclass' self-calls, the subclass's behavior is broken.

17

17

**Is it possible to solve the fragile-base class problem by avoiding assumptions about the self-call structure of the base class in the implementations of the subclasses?**

---

II

I/III

## Fragility by addition of new methods

The Fragile Base Class Problem in a Nutshell

- Fragility by **extending a base class with new methods** that were not there when the class was subclassed.
- Example
  - Consider a base collection class.
  - To ensure some (e.g., security) property, we want to enforce that all elements added to the collection satisfy a certain predicate.
  - We override every method that is relevant for ensuring the security property to consistently check the predicate.
  - Yet, the **security may be defeated unintentionally** if a new method is added to the base class which is relevant for the (e.g., security) property.

18

18

Several holes of this nature had to be fixed when `java.util.Hashtable` and `java.util.Vector` were retrofitted to participate in the Java Collection Frameworks.

II

II/III

# Fragility by addition of new methods

The Fragile Base Class Problem in a Nutshell

- Fragility by **extending a base class with a method** that was also added to a subclass.
  I.e., we accidentally capture a new method; the new release of the base class accidentally includes a method with the same name and parameter types.
- If the return types differ, your code will not compile anymore because of conflicting method signatures.
- If the signatures are compatible, your methods may get involved in things you never thought about.



19

II

II/III

# Fragility by addition of new methods

The Fragile Base Class Problem in a Nutshell

- Fragility by **adding an overloaded method to the base class**; the new release of the base class accidentally includes a method which makes it impossible for the compiler (in 3rd party code) to determine the call target of your call.

```
class X { void m(String){…} ; void m(Object o){…}/*added*/ }

<X>.m(null)  // the call target is not unique (anymore)
```



20

Though this issues is generally trivial to fix - we just have to type “null”; eg., `foo((X) null)` - to ensure that the right method is called, the client still needs to updated. Note, that this problem does not exist, if we follow the general best practice of not using `null` and using `Option[T]/Optional<T>` instead.

# Taming Inheritance

Implementation inheritance (`extends`) is a powerful way to achieve code reuse.

But, if used inappropriately, it leads to fragile software.



21

In the following, we discuss rules of thumb for making "good use" of inheritance.

## Dos and Don'ts

Taming Inheritance

Closed Packages Assumption

- It is *always* safe to use inheritance within a package. The subclass and the superclass implementation are under the control of the same programmers.
- It is also OK to **extend classes specifically designed and documented for extension**.
- Avoid inheriting from concrete classes not designed and documented for inheritance across package boundaries.



22

23

*Design and document for inheritance or else prohibit it.*

-Joshua Bloch, Effective Java



24

# Classes Must Document Self-Use

Taming Inheritance

- Each public/protected method/constructor must **indicate self-use**:
  - Which overridable methods it invokes.
  - In what sequence.
  - How the results of each invocation affect subsequent processing.
- A class must document any circumstances under which it might invoke an overridable method.

(Invocations might come from background threads or initializers; indirect invocations can also come from static initializers.)

Packages are considered closed!

## *Common Conventions* for Documenting Self-Use

Taming Inheritance

- The description of self-invocations to overridable methods is given at the end of a method's documentation comment.
- The description starts with "`This implementation …`". Indicates that the description tells something about the internal working of the method.



25

Overridable method = non-final and either public or protected

## Example of Documentation On Self-Invocation

- Taken from: `java.util.AbstractCollection`

```
public boolean remove(Object o)

Removes a single instance of the specified element from this
collection.
…
This implementation removes the element from the collection
using the iterator's remove method.
Note that this implementation throws an
UnsupportedOperationException if the iterator returned by
this collection's iterator() method does not implement the
remove(…) method.
```



26

The documentation makes explicit that overriding iterator() will affect the behavior of remove and what the effect would be.

## Documenting Self-Use In API Documentation

Do implementation details have a rightful place in a good API documentation?

White-Box Use

Black-Box Use

27

27

The answer is simple: It depends!

- Keep in mind: There are two kinds of clients of an extensible class:
  - Ordinary clients create instances of the class and call methods in its interface (black-box use).
  - Clients that extend the class via inheritance (white-box use).
- Ordinary clients should not know such details.
  … At least as long as a mechanism for LSP is in place.
- Subclassing clients need them. That's their "interface".

Current documentation techniques and tools lack proper means of separating the two kinds of API documentations.

---

## Example of Documentation On Self-Invocation

- Taken from: `java.util.AbstractList`

```
protected void removeRange(int fromIndex, int toIndex)

Removes from a list …

This method is called by the clear operation on this list and
its sub lists. Overriding this method to take advantage of the
internals of the list implementation can substantially improve
the performance of the clear operation on this list and its sub
lists…

This implementation gets a list iterator positioned before
fromIndex and repeatedly calls ListIterator.next and
ListIterator.remove. Note: If ListIterator.remove requires
linear time, this implementation requires quadratic time.
```

28

28

A class must document the supported hooks to its internals. These internals are irrelevant for ordinary users of the class. But, they are crucial for enabling subclasses to specialize the functionality in an effective way.

## Carefully **Design** and **Test** Hooks To Internals

- Provide as few protected methods and fields as possible.
- Each of them represents a commitment to an implementation detail.
- Designing a class for inheritance places limitations on the class.
- Do not provide too few hooks.
- A missing protected method can render a class practically unusable for inheritance.

29

29

How to decide about the protected members to expose?

W.r.t. designing the internal hooks and making decisions about the kind and number of internal hooks, *no silver bullet exists*. You have to think hard, take your best guess, and test.

Test your class for extensibility before releasing them. By writing test subclasses (At least one subclass should be written by someone other than the superclass author).

---

Standing Rule

## Constructors Must Not Invoke Overridable Methods

30

30

## Constructors Must Not Invoke Overridable Methods

```
class JavaSuper {
  public JavaSuper() { printState(); }

  public void printState() { System.out.println("no state"); }
}

class JavaSub extends JavaSuper {
    private int x = 42; // the result of a tough computation

    public void printState() { System.out.println("x = " + x); }
}

class JavaDemo {
    public static void main(String[] args) {
        JavaSuper s = new JavaSub();
        s.printState();
    }
}
```

Result:
x = 0
x = 42

31

31

Ask yourself: What is printed on the screen?

### Problem

An overridable method called by a constructor may get invoked on a non-initialized receiver.
As a result a failure may occur.

### Reason

- The superclass constructor runs before the subclass constructor.
- The overridden method will get invoked before the subclass constructor has been invoked.
- The overridden method will not behave as expected if it depends on any initialization done by the subclass constructor.

---

## Constructors Must Not Invoke Overridable Methods

```
class ScalaSuper {

    printState(); // executed at the end of the initialization

    def printState() : Unit = { println("no state") }
}

class ScalaSub extends ScalaSuper {
    var y: Int = 42 // What was the question?
    override def printState() : Unit = { println("y = "+y) }
}

object ScalaDemo extends App {
    val s = new ScalaSub
    s.printState() // after initialization
}
```

Result:
y = 0
y = 42



32

Ask yourself: What is printed on the screen?

For further details: [Scala Language Specification](http://www.scala-lang.org/docu/files/ScalaReference.pdf).

## Constructors Must Not Invoke Overridable Methods

```
class Super {

    // executed at the end of the initialization
    printState();

    def printState() : Unit = { println("no state") }
}

class Sub(var y: Int = 42) extends Super {
    override def printState() : Unit = { println("y = "+y) }
}

object Demo extends App {
    val s = new Sub
    s.printState() // after initialization
}
```

Result:
y = 42
y = 42

Idiomatic Scala

33

33

Ask yourself: What is printed on the screen?
Here, a standard class parameter is used to define the field value.

Recommended reading: *How Scala Experience Improved Our Java Development; http://spot.colorado.edu/~reids/papers/how-scala-experience-improved-our-java-development-reid-2011.pdf*

---

## *Initializers* Must Not Invoke *Overridable Methods*

```
trait Super {
    val s: String
    def printState() : Unit = { println(s) }

    printState();
}

class Sub1 extends Super { val s: String = 110.toString }
class Sub2 extends { val s: String = 110.toString } with Super


new Sub1()
new Sub2()
```

Result:
null
110

34

34

Ask yourself: What is printed on the screen?
In case of traits it is possible to use a so-called ***early field definition*** *clause (*`extends { … }`*)* to define the field value before the super type constructor is called.

For further details: *Scala Language Specification (5.1.6 Early Definitions); http://www.scala-lang.org/docu/files/ScalaReference.pdf*