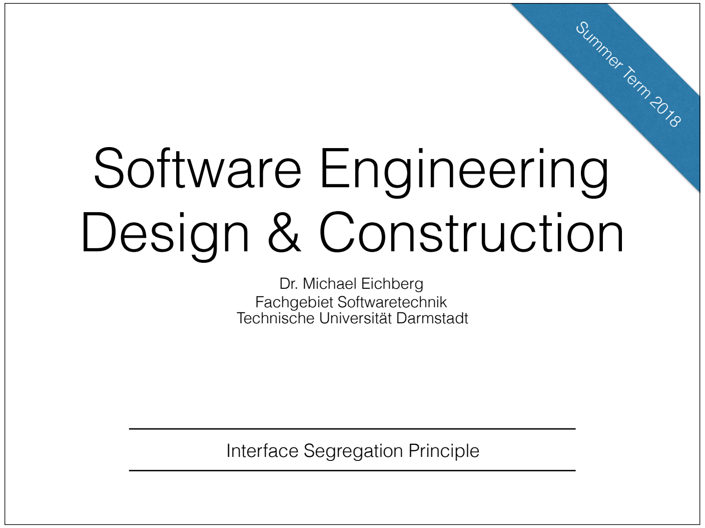
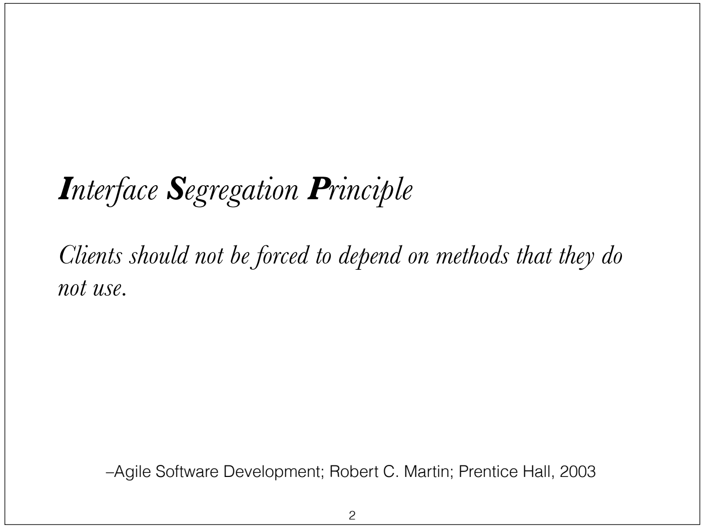


Summer Term 2018

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Interface Segregation Principle



I*nterface* ***S****egregation* ***P****rinciple*

Clients should not be forced to depend on methods that they do not use.

–Agile Software Development; Robert C. Martin; Prentice Hall, 2003

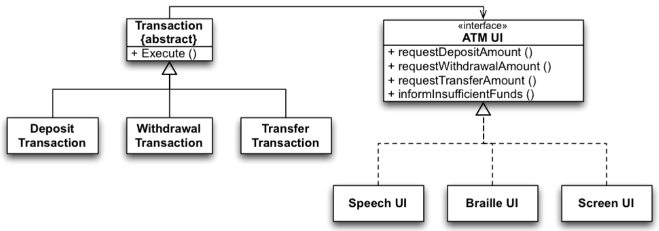
Here, clients are those classes which use a specific interface.

Introduction by Example

- Consider the development of software for an automated teller machine (ATM):
  - Support for the following types of transactions is required: **withdraw**, **deposit**, and **transfer**.
  - Support for different **languages** and support for different **kinds of UIs** is also required
  - Each transaction class needs to call methods on the GUI
    E.g., to ask for the amount to deposit, withdraw, transfer.

Introduction by Example

- Initial design of a software for an automatic teller machine (ATM):



Transaction
{abstract}
+ Execute ()

«interface»
ATM UI
+ requestDepositAmount ()
+ requestWithdrawalAmount ()
+ requestTransferAmount ()
+ informInsufficientFunds ()

Deposit Transaction

Withdrawal Transaction

Transfer Transaction

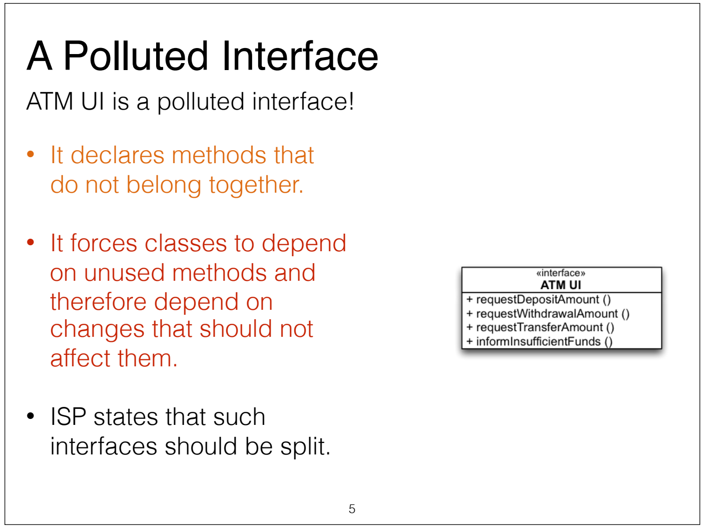
Speech UI

Braille UI

Screen UI

What do you think?

ISP tells us to avoid this. Each transaction class uses a part of the interface, but depends on all others. Any change affects all transactions.

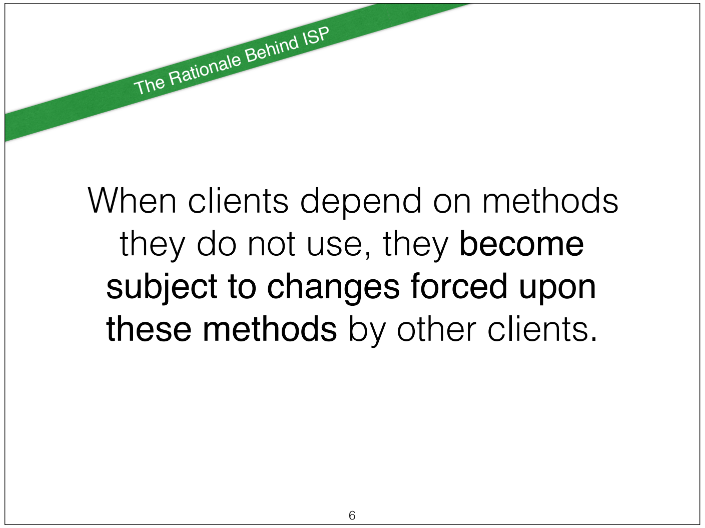


A Polluted Interface

ATM UI is a polluted interface!

- It declares methods that do not belong together.
- It forces classes to depend on unused methods and therefore depend on changes that should not affect them.
- ISP states that such interfaces should be split.

«interface»
ATM UI
+ requestDepositAmount ()
+ requestWithdrawalAmount ()
+ requestTransferAmount ()
+ informInsufficientFunds ()

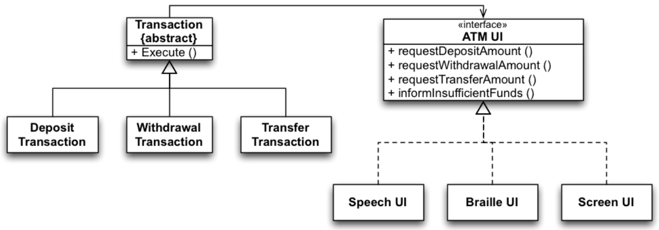


The Rationale Behind ISP

When clients depend on methods they do not use, they **become subject to changes forced upon these methods** by other clients.

This causes coupling between all clients!

How does an ISP compliant solution look like?



Transaction
{abstract}
+ Execute ()

«interface»
ATM UI
+ requestDepositAmount ()
+ requestWithdrawalAmount ()
+ requestTransferAmount ()
+ informInsufficientFunds ()

Deposit Transaction

Withdrawal Transaction

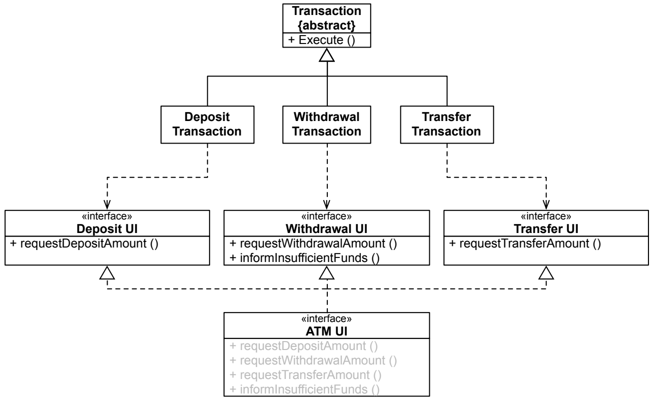
Transfer Transaction

Speech UI

Braille UI

Screen UI

An ISP Compliant Solution



Transaction
{abstract}
+ Execute ()

Deposit Transaction

Withdrawal Transaction

Transfer Transaction

«interface»
Deposit UI
+ requestDepositAmount ()

«interface»
Withdrawal UI
+ requestWithdrawalAmount ()
+ informInsufficientFunds ()

«interface»
Transfer UI
+ requestTransferAmount ()

«interface»
ATM UI
+ requestDepositAmount ()
+ requestWithdrawalAmount ()
+ requestTransferAmount ()
+ informInsufficientFunds ()

Here, the client (Deposit|Withdrawal|Transfer)Transaction only depends on a UI related interface related to its specific task.

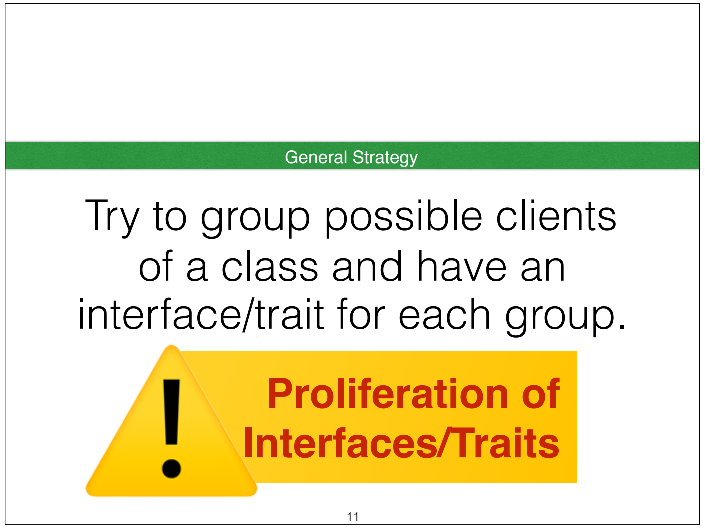
***I**nterface / **T**rait **S**egregation **P**rinciple*

Clients should not be forced to depend on methods that they do not use.

–Agile Software Development; Robert C. Martin; Prentice Hall, 2003

General Strategy

Try to group possible clients of a class and have an interface/trait for each group.



General Strategy

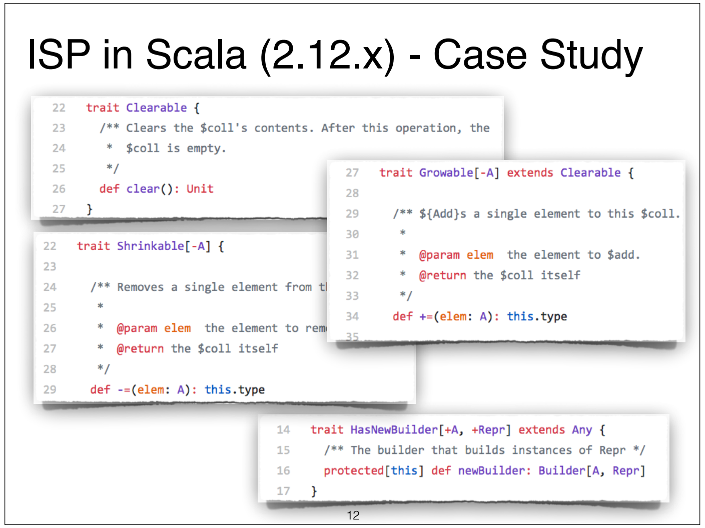
Try to group possible clients of a class and have an interface/trait for each group.

Proliferation of Interfaces/Traits

Segregating interfaces should not be overdone!

If you overdue the application of the interface segregation principle, you will end up with 2n-1 interfaces for a class with n methods.

Recall that, in general, a class implementing many interfaces may be a sign of a violation of the single-responsibility principle.



ISP in Scala (2.12.x) - Case Study

```
trait Clearable {
  /** Clears the $coll's contents. After this operation, the
   *  $coll is empty.
   */
  def clear(): Unit
}
```

```
trait Growable[-A] extends Clearable {

  /** ${Add}s a single element to this $coll.
   *
   *  @param elem  the element to $add.
   *  @return the $coll itself
   */
  def +=(elem: A): this.type
```

```
trait Shrinkable[-A] {

  /** Removes a single element from tl
   *
   *  @param elem  the element to rem
   *  @return the $coll itself
   */
  def -=(elem: A): this.type
```

```
trait HasNewBuilder[+A, +Repr] extends Any {
  /** The builder that builds instances of Repr */
  protected[this] def newBuilder: Builder[A, Repr]
}
```

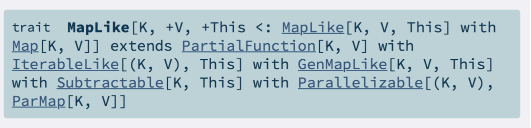
Growable and Shrinkable both define further convenience methods (e.g., ++= and -- = etc.).

Issue: The class hierarchy seems to be inconsistent, because a Shrinkable ist not Clearable, but a Growable is.

Issue: Proliferation of the interfaces.

(In 2.13. the collections API will be overhauled.)

ISP in Scala (2.12.x) - Case Study



```
trait  MapLike[K, +V, +This <: MapLike[K, V, This] with
Map[K, V]] extends PartialFunction[K, V] with
IterableLike[(K, V), This] with GenMapLike[K, V, This]
with Subtractable[K, This] with Parallelizable[(K, V),
ParMap[K, V]]
```

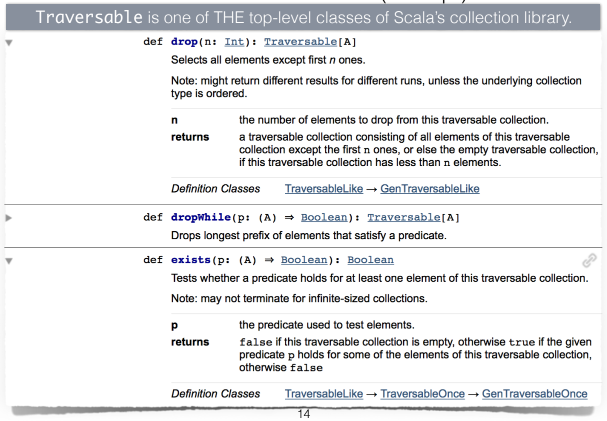
We can flexibly combine the functionality to get collection classes with a rich interface!

Issue: some of the methods may offer poor performance; e.g., filter on immutable sets is implemented by building a new collection - there will be no sharing!

(In 2.13. the collections API will be overhauled.)

Do we have an ISP violation?

`scala.collection.Traversable` (excerpt)



`Traversable` is one of THE top-level classes of Scala's collection library.

`def drop(n: Int): Traversable[A]`

Selects all elements except first *n* ones.

Note: might return different results for different runs, unless the underlying collection type is ordered.

n the number of elements to drop from this traversable collection.

returns a traversable collection consisting of all elements of this traversable collection except the first `n` ones, or else the empty traversable collection, if this traversable collection has less than `n` elements.

Definition Classes TraversableLike → GenTraversableLike

`def dropWhile(p: (A) ⇒ Boolean): Traversable[A]`

Drops longest prefix of elements that satisfy a predicate.

`def exists(p: (A) ⇒ Boolean): Boolean`

Tests whether a predicate holds for at least one element of this traversable collection.

Note: may not terminate for infinite-sized collections.

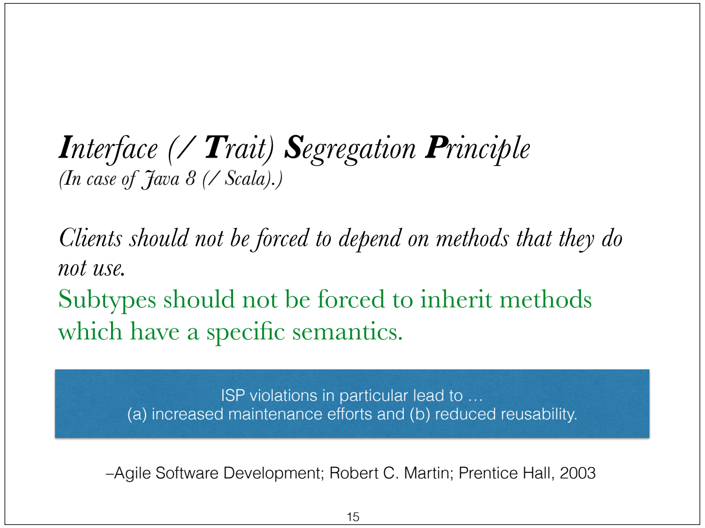
p the predicate used to test elements.

returns `false` if this traversable collection is empty, otherwise `true` if the given predicate `p` holds for some of the elements of this traversable collection, otherwise `false`

Definition Classes TraversableLike → TraversableOnce → GenTraversableOnce

If the semantics of one of the defined methods is not suitable for a custom collection that wants to inherit from Traversable (*e.g., because* `drop(n)` *should fail if n is too large*), it is no longer possible to inherit from this class (otherwise we would get a Liskov Substitution Principle violation). Splitting up the methods in two or more traits would improve reusability.

This problem became more prevalent with Java 8 because it is now possible - by means of default methods defined in interfaces - to inherit concrete methods. (The problem always existed in Scala (by means of `traits`).)



***I**nterface (/ **T**rait) **S**egregation **P**rinciple*

(In case of Java 8 (/ Scala).)

Clients should not be forced to depend on methods that they do not use.

Subtypes should not be forced to inherit methods which have a specific semantics.

ISP violations in particular lead to …
(a) increased maintenance efforts and (b) reduced reusability.

–Agile Software Development; Robert C. Martin; Prentice Hall, 2003

In this case, it is important to understand that the clients of a class are those that use the class (by invoking methods on an instance of the respective type) or which inherit from the respective class or trait.

In the previous case (i.e., in the case of the Scala library), the decision was made to avoid throwing exceptions as long as possible/to handle corner cases gracefully. This line of thinking is not suitable in all cases. If it is the violation prevents classes from inheriting from these collection classes.