AARHUS UNIVERSITET

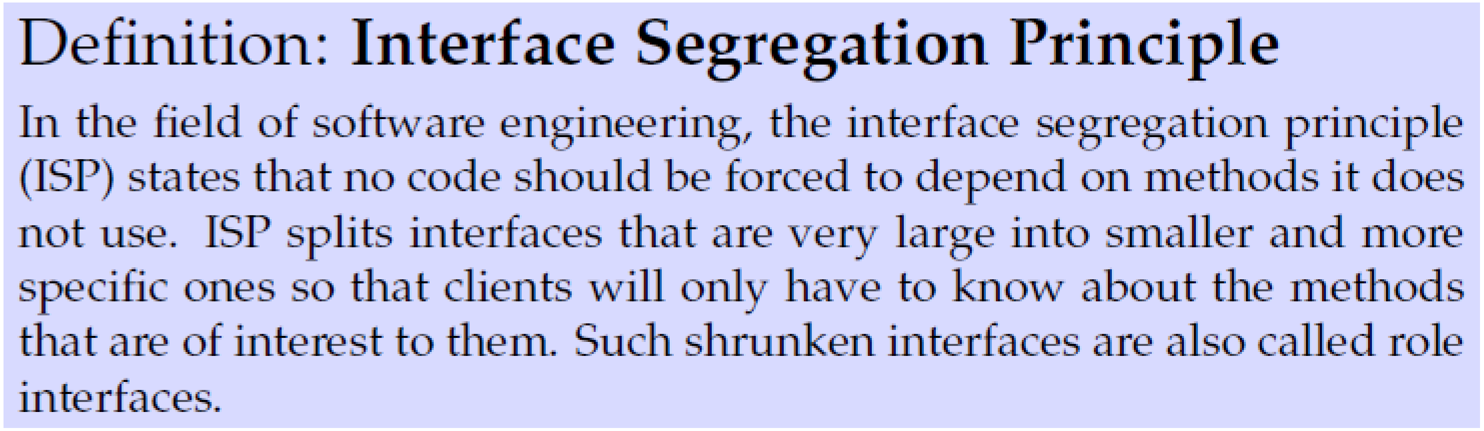
Software Engineering and Architecture

Interface Segregation Principle

Role and Private Interfaces

ISP

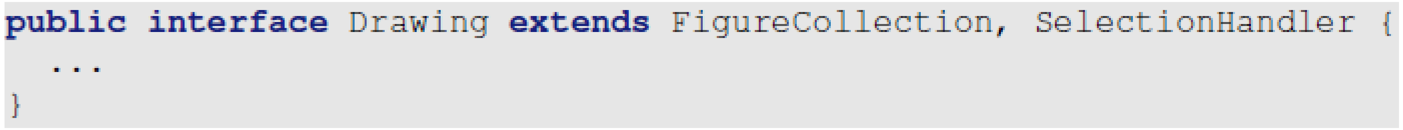
- “Students should not use my Father interface”…
- Or ‘do not depend on methods you do not use’



Definition: Interface Segregation Principle

In the field of software engineering, the interface segregation principle (ISP) states that no code should be forced to depend on methods it does not use. ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces.

- Example:

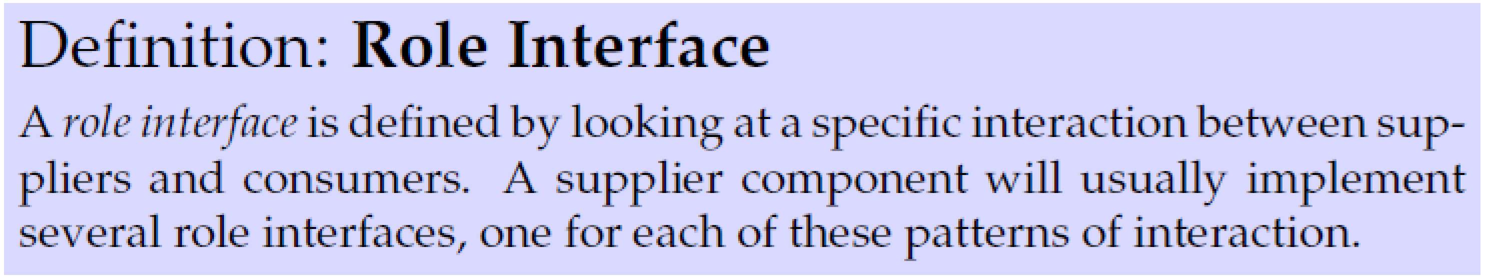


```
public interface Drawing extends FigureCollection, SelectionHandler {
  ...
}
```

AARHUS UNIVERSITET

Fine-grained Roles

- The 'more specific' role is expressed as a Role Interface



Definition: **Role Interface**

A *role interface* is defined by looking at a specific interaction between suppliers and consumers. A supplier component will usually implement several role interfaces, one for each of these patterns of interaction.

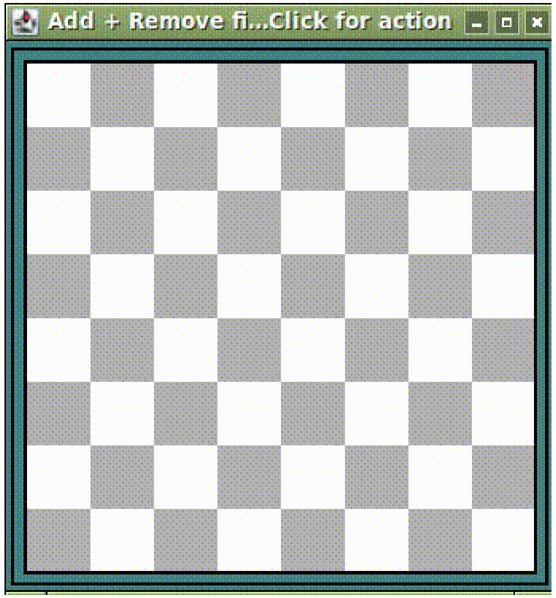
Martin Fowler

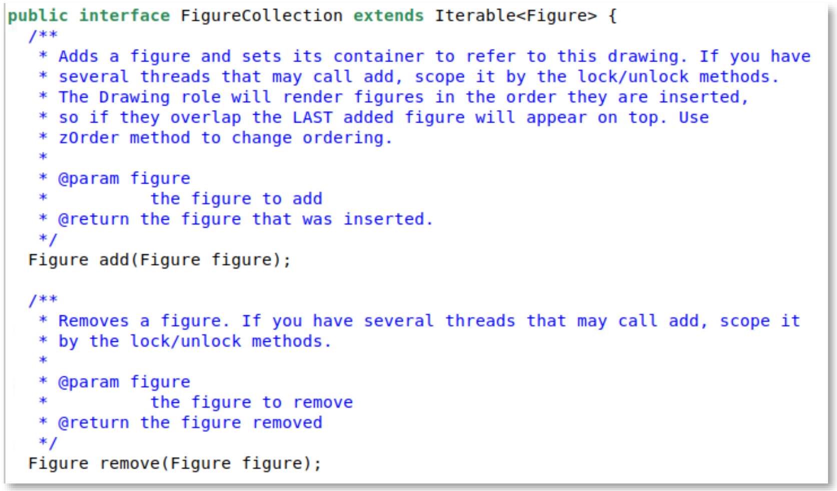
- Ala again
  - I provide a 'teacher' interface (one role interface)
  - And a 'taxpayer' interface (another role interface)
  - Etc.

AARHUS UNIVERSITET

Example

- The FigureCollection in MiniDraw only deals with *adding, removing, and iterating the collection of Figures in MiniDraw*





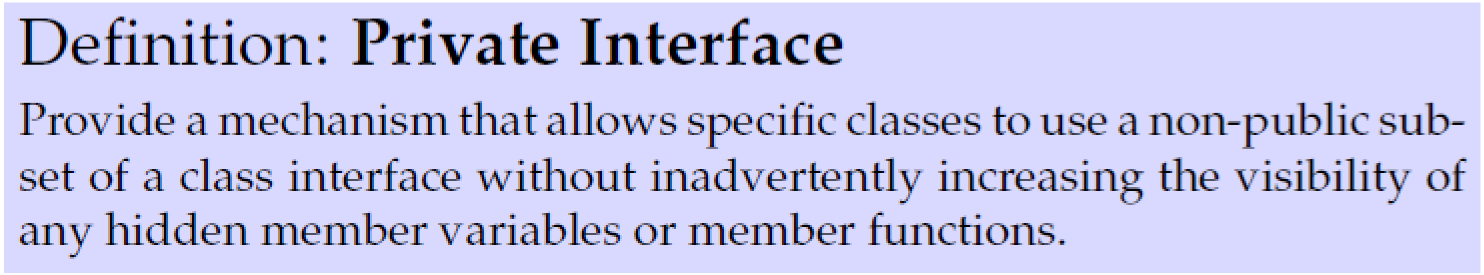
```
public interface FigureCollection extends Iterable<Figure> {
  /**
   * Adds a figure and sets its container to refer to this drawing. If you have
   * several threads that may call add, scope it by the lock/unlock methods.
   * The Drawing role will render figures in the order they are inserted,
   * so if they overlap the LAST added figure will appear on top. Use
   * zOrder method to change ordering.
   *
   * @param figure
   *          the figure to add
   * @return the figure that was inserted.
   */
  Figure add(Figure figure);

  /**
   * Removes a figure. If you have several threads that may call add, scope it
   * by the lock/unlock methods.
   *
   * @param figure
   *          the figure to remove
   * @return the figure removed
   */
  Figure remove(Figure figure);
```

A specific interaction (add+remove) between the UI and the Drawing, expressed as the Role Interface 'FigureCollection'

Private Interface

- Role interfaces are often used to enforce more specific *encapsulation* than is possible using *private/public* methods and instance variables…



Definition: Private Interface

Provide a mechanism that allows specific classes to use a non-public subset of a class interface without inadvertently increasing the visibility of any hidden member variables or member functions.

James Newkirk

- Let us make an example, highly inspired by our project…

AARHUS UNIVERSITET

Example

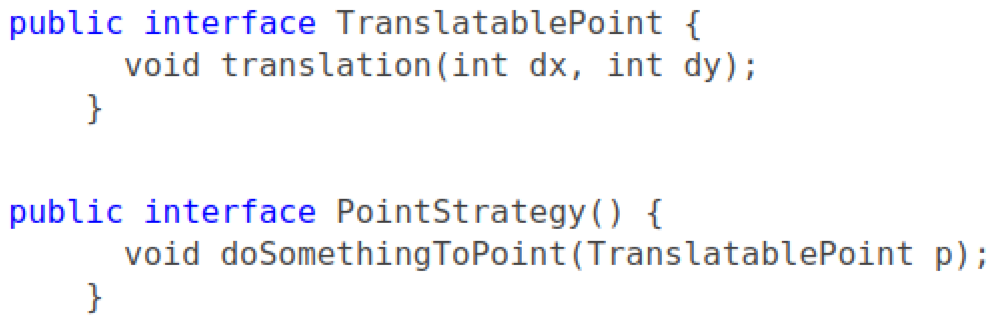
- We have a system/framework/**Façade** which presents (x,y) points to outside code, but that out side code must *never modify the (x,y) values!*
- *Read-only* **Role interface** is a solution to that.
  - Only accessor methods, no mutator methods…

```
public interface Point {
      int getX();
      int getY();
    }

public interface Facade {
      Point getPoint();
    }
```

Example

- However, internal classes inside the Façade of course needs to mutate the state of these (x,y) points.
- Let us say that one class needs to translate (dx,dy) points
- **Private Interface** is a solution to that



```
public interface TranslatablePoint {
      void translation(int dx, int dy);
    }

public interface PointStrategy() {
      void doSomethingToPoint(TranslatablePoint p);
    }
```

Example

- Now the internal, implementing, class of course implements both

```
public class StandardPoint implements Point, TranslatablePoint {
      [all three methods implemented here]
    }

public class MyFacade implements Facade {
      StandardPoint point;
      Point getPoint() { return point; }
    }
```

- That is, if you use ‘getPoint()’ from the outside you only get access to ‘getX()’ and ‘getY()’

AARHUS UNIVERSITET

Example

- Now, an internal PointStrategy can translate points like

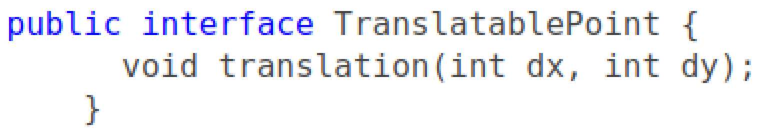
```
public class Strategy1 implements PointStrategy {
      void doSomethingToPoint(TranslatablePoint p) {
        p.translation(+3, +7);
      }
    }
```

- And can be called internally like

```
strategy.doSomethingToPoint(point);
```

Example

- However, a PointStrategy cannot access (x,y)…



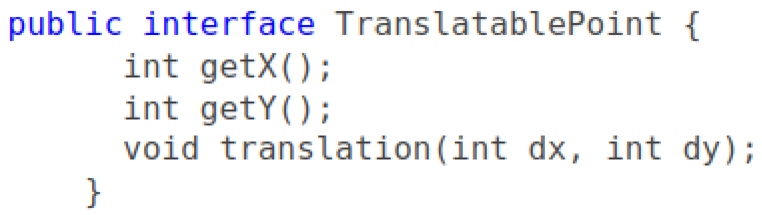
```java
public interface TranslatablePoint {
    void translation(int dx, int dy);
}
```

- However, of course it is often the case, that we need just that.

- *Exercise: How do we solve that?*

Solution 1:

- Fine-grained solution: **Missing accessor methods**
  - Just add those methods that are missing



```
public interface TranslatablePoint {
      int getX();
      int getY();
      void translation(int dx, int dy);
    }
```

- Pro
  - Can select just the right set of accessors
    - (here it is both of them, but if read-only had 20, we may just pick the two we need).
- Con
  - Same methods are now present in two interfaces

AARHUS UNIVERSITET

Ups?

- Uhum – how does that work in Java?

```
public interface Point {
    int getX();
    int getY();
}
```

```
public interface TranslatablePoint {
    int getX();
    int getY();
    void translation(int dx, int dy);
}
```

```
public class StandardPoint implements Point, TranslatablePoint {
    [all three methods implemented here]
}
```

- StandardPoint must now implement ‘getX()’ twice or???

- Exercise: What happens?

AARHUS UNIVERSITET

Solution 2:

- Coarse-grained (Lazy) approach: **Extend existing**
  - Just implement both

```
public interface TranslatablePoint extends Point {
      void translation(int dx, int dy);
    }
```

- Pro
  - Less typing
  - ***You can actually Program to an Interface in the façade impl!***
- Con
  - You get all methods

```
public class MyFacade implements Facade {
     TranslatablePoint point;
      Point getPoint() { return point; }
    }
```

Mandatory Note

- We have *read-only* role interfaces for Card and Hero in HotStone.
  - But Game’s implementation and strategies need to manipulate them…
  - *Use private interfaces for that ☺*

- Strategies needs special mutations of Game
  - *Use private interface(s) for that ☺*

- ***Now you ‘program to an interface’***, and avoid hard coupling to, say, StandardGame etc.