AARHUS UNIVERSITET

Software Engineering and Architecture

Interface Segregation Principle
Role and Private Interfaces

AARHUS UNIVERSITET

Roles…

- … will be treated in much more detail soon, but…

> Definition: **Role (Software)**
> A set of responsibilities and associated protocols.

- We use *interface* in our OO language to express a *role*
  - *interface Game*
    - Express the responsibilities of our HotStone game
      - Allow clients to inspect game state, play a card, attack a hero, etc.
  - *Interface Card*
    - Express the responsibilities of a HotStone card
      - Store mana cost, health, etc. Allow clients to get those values.

AARHUS UNIVERSITET

Role-Object

- Objects implement/”play” these roles
  - Class StandardGame implements Game
  - Game game = new StandardGame(….);
  - Now we know that ‘game’ object can *act* as defined by the role ‘Game’
- *A role can be played by many different objects*
  - ‘Game’ is implemented by about 90 SWEA groups!
    - All different (except if a group cheats ☺)!
- *An object can play many different roles*
  - *Huh?*

AARHUS UNIVERSITET

Role-Object

- *An object can play many different roles*
  - My roles:
    - Teacher, researcher, husband, father, taxpayer, colleague, friend, …
- But not at the same time…
  - I am a father and a teacher. But I *alternate* between the roles…
    - Student: “Please, Henrik, can you fix my flat bike tire?”
      - **No I will not! That responsibility belongs to the Father role**
    - Student: “Could you explain the ‘Role’ concept in programming?”
      - Yes, I will do that. That responsibility belongs to the Teacher role.
    - Child: “Could you explain the ‘Role’ concept in programming?”
      - Uhum, probably not relevant, unless that child is a student of mine…

AARHUS UNIVERSITET

Bottom-line

- An object may also serve different roles, but not all that interact with it are *allowed* to use the full set of roles
  - Only my children may interact with my father interface
  - Only Danish Tax may interact with my taxpayer interface

- Our analogy
  - The Game object currently serves two different roles
    - Client/UI users that *only are allowed to call the official 'Game' interface* *(ala: 'Teacher' interface)*
    - Strategies that *must be allowed to alter internal game state*, like subtracting -2 health to a hero object *(ala: 'Father' interface)*

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 {
  ...
}
```

Fine-grained Roles

- The ‘more specific’ role is expressed as a Role Interface

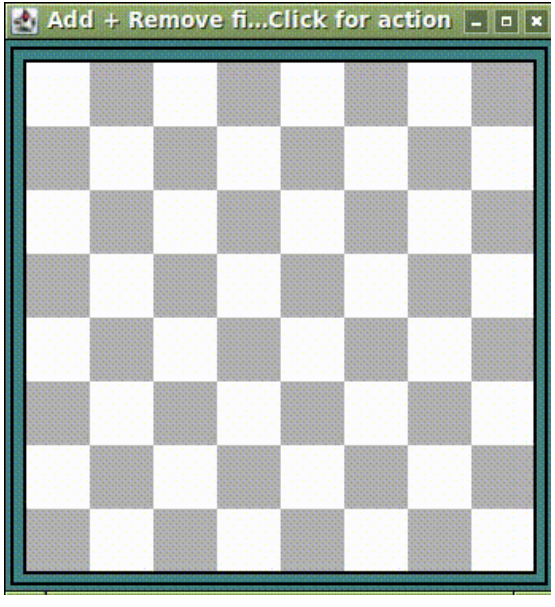
> **Definition: Role Interface**
>
> A *role interface* is defined by looking at a specific interaction between suppliers and consumers. A supplier component will usually implement several role interfaces, one for each of these patterns of interaction.

Martin Fowler

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

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…

Example

- We have a system/framework/**Façade** which presents (x,y) points to outside code, but that outside 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()’

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)…

```
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

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 approach: **Extend existing interface**
  - 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; }
}
```

AARHUS UNIVERSITET

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.

AARHUS UNIVERSITET

Mandatory Note

- And iff you use the ‘extending existing interface’
  - interface MutableGame implements Game { (mutators here) }
- … you can now *remove* all casts and all *hard couplings* to your StandardGame/StandardCard etc
  - Map<Player, List<**MutableCard**>> handMap = …
  - Interface HeroPowerStrategy {
  - public void usePower(**MutableGame** game);
  - }
- … and still have the full flexibility of providing any concrete class that implements MutableX in your