260CT
Software Engineering

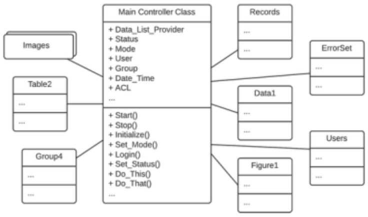
Dr. Yih-Ling Hedley
Email: aa0817@coventry.ac.uk

Design Patterns: Gang of Four

Creational	Structural	Behavioural
• AbstractFactory • Builder • FactoryMethod • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Flyweight • Proxy	• ChainOfResponsibility • Command • Interpreter • Iterator • Mediator • Memento • Observer • State • Strategy • TemplateMethod • Visitor

AntiPatterns 1

- Anti-Patterns are, contrary to Design Patterns, recurring programming practices that create problems instead of solving them, and are considered to be a bad programming practice. Example: Blob AntiPattern



AntiPatterns 1.1

- Blob AntiPattern:
  - A Single class with a **large number of attributes and/or operations**.
  - A collection of **unrelated attributes and operations** encapsulated in a single class.
  - The single controller class often nearly **encapsulates the applications entire functionality**, much like a procedural main program.
  - The class is **too complex for reuse and to modify** the system without affecting the functionality of other encapsulated objects.

AntiPatterns 2

Anti-Patterns Examples:
(http://sourcemaking.com/antipatterns)

Design By Contract (DbC) 1
(for Software)

- The process of developing software based on the notion of a contract between objects
  - One object requires a service from another object. These are thought of as the **client (customer)** object and the **supplier** object.
  - Both objects have rights and responsibilities (benefits and obligations).

Design By Contract (DbC) 2
(for Software)

- Supports production of **quality software**
- Contracts are expressed using:
  - **pre-conditions –** input values
  - **post-conditions –** output values
  - **Invariants –** allowed values in attributes

Subcontracting & Re-Use

- If the design specifies a component that must meet a particular contract, then it may need to:
  - Develop a **new component** to meet the contract.
  - **Re-use an existing component** that already meets the contract
    - a component from existing **component library**
    - a **COTS (Commercial Off-The-Shelf)** component from a commercial supplier.

Subcontracting & Re-Use $_1$

- **Reuse** is the best policy, if it is possible.
- Example: Instead of the prime contractor, class A, the existing class B could be used.
  - **Sub-contracting** the work of class A to class B.
  - Need to ensure that the subcontractor B can do the job of the contractor A.

Rules for Sub-contracting

- If class A's job is sub-contracted out to class B then class B must have:
  - **Weaker pre-conditions** than class A (**or equal**)
  - **Stronger post-conditions** than class A (**or equal**)

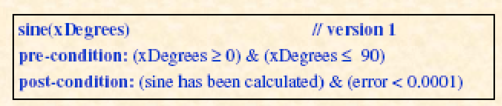
Rules: Pre-conditions

- **Weakening Pre-conditions**
  - To weaken a pre-condition is to make it **less restrictive**.
  - It will be valid in a **wider range** of situations.
  - It will make the operation more **widely usable**.

Rules: Post-conditions

- **Strengthening Post-conditions**
  - To strengthen (tighten) a post-condition is to make it **more restrictive**.
  - It define a **narrower range** of "answer" situations.
  - It means the operation will be doing a "better" job.

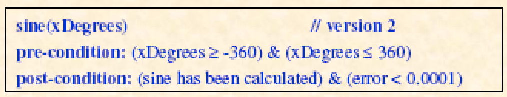
Example 1: Maths Class Version 1



```
sine(xDegrees)                          // version 1
pre-condition: (xDegrees ≥ 0) & (xDegrees ≤ 90)
post-condition: (sine has been calculated) & (error < 0.0001)
```

- This will work out the value of the sine for any angle between 0^o and 90^o
- It will return an answer accurate to at least 4 decimal places.

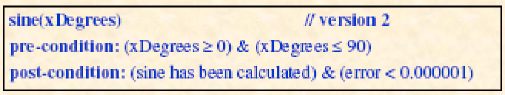
Example 1: Maths Class
Version 2: Weakening the Pre-conditions



```
sine(xDegrees)                          // version 2
pre-condition: (xDegrees ≥ -360) & (xDegrees ≤ 360)
post-condition: (sine has been calculated) & (error < 0.0001)
```

- This will work out the value of sine for a greater range of angles (between -360^o and $+360^o$) than version 1
- It **less restrictive**.
- It will be valid in a wider range of situations.
- It will make the operation **more widely usable**.

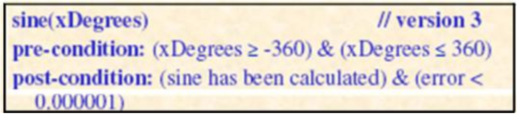
Example 1: Maths Class
Version 2: Strengthening the Post-condition



```
sine(xDegrees)                          // version 2
pre-condition: (xDegrees ≥ 0) & (xDegrees ≤ 90)
post-condition: (sine has been calculated) & (error < 0.000001)
```

- This post condition will guarantee a value of sine to better accuracy (i.e. 6 decimal places) than version 1 (originally 4 decimal places).
- It defines a **narrower range** of allowed answers.
- The operation will do a "better" job and will potentially meet the needs of more clients

Example 1: Maths Class
Version 3: Both

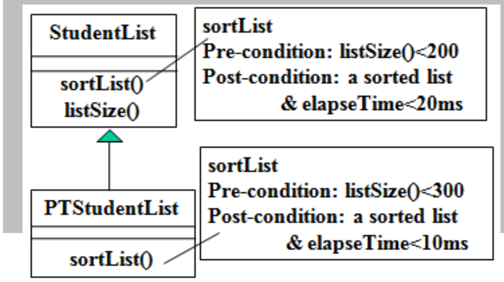


```
sine(xDegrees)                          // version 3
pre-condition: (xDegrees ≥ -360) & (xDegrees ≤ 360)
post-condition: (sine has been calculated) & (error <
    0.000001)
```

- The precondition makes the operation more widely usable.
- The post-condition allows a narrower range of outputs.
- This version will potentially meet the needs of most clients.

Design by Contract: Exercise 1

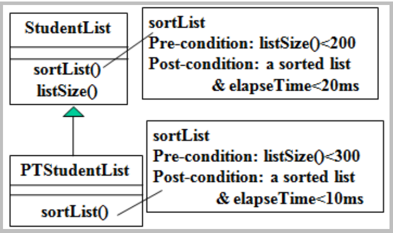
Determine whether the following allows StudentList to be sub-contracted out to its subclass, PTStudentList.



sortList
Pre-condition: listSize()<200
Post-condition: a sorted list
& elapseTime<20ms

sortList
Pre-condition: listSize()<300
Post-condition: a sorted list
& elapseTime<10ms

Design by Contract: Exercise 1 Feedback



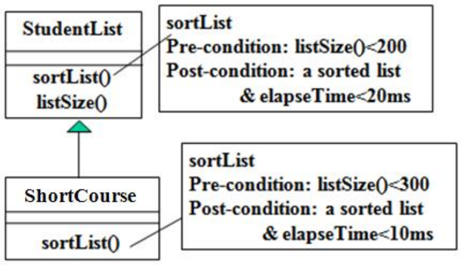
sortList
Pre-condition: listSize()<200
Post-condition: a sorted list
& elapseTime<20ms

sortList
Pre-condition: listSize()<300
Post-condition: a sorted list
& elapseTime<10ms

Feedback: Yes, PTStudentList has a weakened (widened) pre-conditions and strengthened (narrowed) post-conditions, thus StudentList canbe sub-contracted out to its subclass, PTStudentList

Design by Contract: Exercise 2

Determine whether the following allows StudentList sub-contract out to ShortCourse.



StudentList

sortList()
listSize()

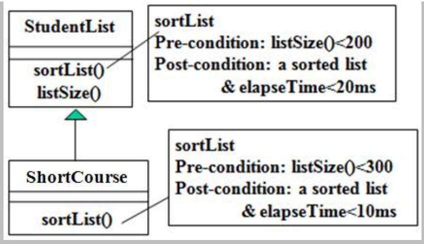
sortList
Pre-condition: listSize()<200
Post-condition: a sorted list
& elapseTime<20ms

ShortCourse

sortList()

sortList
Pre-condition: listSize()<300
Post-condition: a sorted list
& elapseTime<10ms

Design by Contract: Exercise 2 Feedback



StudentList

sortList()
listSize()

sortList
Pre-condition: listSize()<200
Post-condition: a sorted list
& elapseTime<20ms

ShortCourse

sortList()

sortList
Pre-condition: listSize()<300
Post-condition: a sorted list
& elapseTime<10ms

Feedback: No, the above design is wrongly modelled as inheritance, as ShortCourse is not a type of StudentList

Example 2: Polymorphism/Substitutability?

- StudentList cannot sub-contract out to ShortCourse. The ShortCourse class does not meet the contractual obligations of the StudentList class.
  - Although it has a weaker precondition and a stronger post-condition.
  - **Reason:** An instance of the ShortCourse class cannot be used in place of (is not substitutable for) an instance of StudentList class.
    - Cannot make use of polymorphism in this situation, even though the target language might allow it. (e.g. C++, or Java)