Advanced Java

LESSON 9: INTERFACE SEGREGATION PRINCIPLE & DEPENDENCY INVERSION PRINCIPLE

Creative Commons

Attribution 4.0 International (CC BY 4.0)

Agenda

Software Craftsmanship

Interface Segregation Principle

Dependency inversion Principle

Software Craftsmanship

Software Craftsmanship

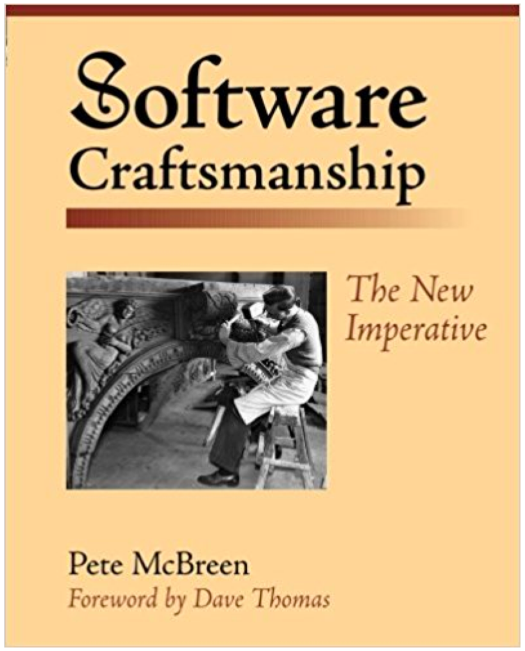
An approach to software development that emphasizes the coding skills of the software developers themselves.

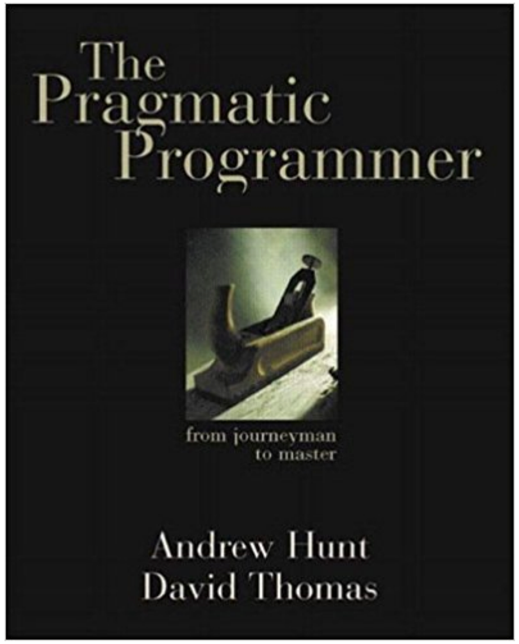
- Historically, programmers have been encouraged to see themselves as practitioners of the well-defined statistical analysis and mathematical rigor of a scientific approach with computational theory.
- This has changed to an engineering approach with connotations of precision, predictability, measurement, risk mitigation, and professionalism.
- Practice of engineering led to calls for licensing, certification and codified bodies of knowledge as mechanisms for spreading engineering knowledge and maturing the field.

Wikipedia.org

Software Craftsmanship

- **The Pragmatic Programmer** by Andy Hunt and Dave Thomas
- **Software Craftsmanship** by Pete McBreen





Software Craftsmanship

As aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

- Not only working software,
  but also well-crafted software
- Not only responding to change,
  but also steadily adding value
- Not only individuals and interactions,
  but also a community of professionals
- Not only customer collaboration,
  but also productive partnerships

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.

http://manifesto.softwarecraftsmanship.org/

Interface Segregation Principle

Interface Segregation Principle

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

Interface Segregation Principle

Do not impose the clients with the burden of implementing methods that they don’t actually need.

- Interfaces should be thin or fine-grained and not force the implementation of unused methods.
- Thin interfaces are also called **Role Interfaces.**
- Fat interfaces lead to inadvertent coupling between classes.

In-Class Activity

1. Download, "ISPExample" project from Blackboard.
2. Read the "ReadMe.txt" file and follow the instructions.

Dependency inversion principle

Dependency Inversion Principle

- HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.
- ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.

Dependency Inversion Principle

Three guiding principles for quality:

1. Code must not be rigid

   (able to plug in different modules without much change)

2. Code must not be fragile

   (an edit in one place shouldn’t cause code to break in other places)

3. Code must be flexible and portable

   (easily adapts to change, easily used in new projects)

Dependency Inversion Principle

- Worker objects (low-level modules in the DIP) must have at least some common methods to be polymorphic (remember, the DIP is all about flexibility – having interchangeable low-level modules).
- However, not all methods need to be common. You simply lose polymorphism on these methods.
- Also, the constructor can never be polymorphic because the class name is the constructor name and two or more classes can never have the same name.

Dependency Inversion Principle

Steps to implement DIP:

1. Identify work that needs to be done in different ways. Example, output needs to be sent to the console or sent to a GUI.
2. Create a class named to represent that work. Example GuiOutput. In that class put at least one common method to perform that work. Example: doOuput. Make sure the method name is generic. If the method needs supplied data to function, pass that in via a constructor or setter method. Avoid using the common method by passing a parameter. The data type of the parameter may not be the same for different worker objects.
3. Create an interface based on your worker class. Then create any additional worker classes you need based on that interface.

Dependency Inversion Principle

Steps to implement DIP (Continue):

4. Create a high-level class – always a concrete class. Then declare properties using abstraction to represent your low-level classes. These are components used by the high-level class to perform work (delegation). How do these objects get into your high-level class so they can be used? Pass them in via a constructor or setter method.
5. Now create a method in the high-level class to perform the work. You can used the same method name as used in the low-level class or create a new one. Example: doOutput. Now program that method in the high-level class to delete the work to the low-level object.
6. Now to use this DIP solution in a program you must 1) instantiate and low-level object using the Liskov Substitution Principle, 2) instantiate a high-level class and pass in the low-level object, and finally, 3) use the high-level object (e.g., where output is needed) to call the appropriate command method (e.g., doOutput).

In-Class Activity

1. Download, "LabDIP.zip" project from Blackboard.
2. Work on Lab2 first.
3. Read the "ReadMe.txt" file and follow the instructions.