DA343A
Föreläsning 13:
“More OO” (Code Re-Use)

Ben Blamey

Why Code Re-use?

- Saves Time (development, documenting, testing)
- Reduces Bugs
- Can reduce software size

1. Make our code more reusable
2. Oppurtunities to re-use existing code

Code Reuse 1 – Class Inheritance

- We can re-use existing code through use of the extends keyword
- Design classes for inheritance/extensibility:
  - Avoid use of final
    - Note that final was used extensively in the JARs for the assignments to guide you towards the intended solution.
  - Keep methods small, avoid side-affects.
  - Annotate key methods with ‘protected’ visibility
    - This way, they are visible to sub-classes even in different packages.
  - Add “setters” with protected visibility, to allow use by derived classes.
    - Instead of attributes.
    - This way, the base class can implement e.g. callbacks, like PropertyChangedListener.
  - Design for future changes – it may be required to add e.g. callbacks in the future! Remember you may not have access to the re-using code.

Code Reuse 2 – Generics

- When implementing your own class, consider using type parameters – e.g. Buffer<T> rather than e.g. StringBuffer.
- This way, your class can be re-used with a variety of different types.

Code Reuse 3 - Composition

- Composition can be more flexible than inheritance, and easier to maintain.
- Consider composing existing classes to create the class you need.
- Consider refactoring your existing code into smaller classes, so you can re-use them elsewhere.
- We already follow this pattern without thinking about it, e.g.
  - e.g. List<T> in Buffer
- Think about oppurtunities to “factor out” a re-usable utility-class, and use it through an attribute.

Code Reuse 4 – Wrapper Pattern

- Typically implemented as a wrapper class, which presents a simplified API.
- Methods in the wrapper class delegate to the re-used class, which may be more complicated than necessary.
- Common examples:
  - Thread-safe collections.
  - Collections which provided additional notifications/callbacks.
  - Read-only collections.
  - Collections which provide validation.

Code Reuse 5 – Depend on “Abstractions” not “concretes”

- ”Abstraction” means abstract classes, or ideally Interfaces.
- Instead of implementing classes which depend on other classes, use an interface which provided the methods needed.
- Often these objects are passed into the constructor, or through setters.
- Simple cases: don’t depend on LinkedList<T>, when what you need is a List<T>, or even only Iterable<T>.
- Consider defining your own interface with the methods you need.
- Depend on this interface instead.
- This makes it possible to re-use your classes in a broader range of cases.
- Depending on a “concrete” implementation means you are stuck with that one!
- This is called the **“Dependency Inversion Principle”**

4 DIP

Överväg ett system för att hantera elevernas information:

Informationen om varje elev finns i en klass "Student". Data lagras i en databas, vars detaljer är inkapslade i en annan klass, som kallas "SQLDatabase". Klassen Student interagerar med klassen SQLDatabase för att lagra sina data i databasen.

Det finns ett krav på att lägga till stöd för ytterligare en databasteknik som kallas "Mongo", som kräver en egen klass med samma offentliga metoder som SQLDatabase-klassen.

För att modifiera koden enligt DIP (Dependency Inversion Principle), vilka nya klasser/gränssnitt (classes/interfaces) skulle vi behöva? Föreslå vettiga engelska namn för dem:

Skriv in ditt svar här

Ny Interface/Gränsnitt: (t.ex) Database
Ny klass: (t.ex) MongoDatabase.

Beskriv tydligt alla relationer mellan klasserna i den nya reviderade koden:

Skriv in ditt svar här

SQLDatabase implements Database interface.
MongoDatabase implements Database interface.
Student depends on/uses Database interface (neither of the classes directly).