Further library and clean code constructions

July 3, 2017

Plan-o

- Homework 4 / Piazza discussion
- Homework 5 on Monday
- Final projects
- Midterm
- Material for today

Homework 4

- Typos: fixed (as far as I know)
- Pull issues?
- Missing data points in city data
- Questions?

Final Projects

Final Projects, Option 1

- Contribute a patch to any open source project
  - Must be code (not documentation)
  - Can be tests, can be closing a bug
  - I will help as much as possible!

Examples

- ffmpeg-cli-wrapper
  https://github.com/bramp/ffmpeg-cli-wrapper
- Phonograph
  https://github.com/kabouzeid/Phonograph
- GnuCash (Android)
  https://github.com/codinguser/gnucash-android
- Mango
  https://github.com/jfaster/mango
- Ninja
  https://github.com/ninjaframework/ninja
- JSoup
  https://github.com/jhy/jsoup

Final Projects, Option 2

- Write an interesting Android application
    - Some persistent storage (locally or network)
    - At least multiple panels
    - I will help as much as possible!

Examples

- Todo Lists
- Media Players
- RSS Aggregators
- Something more interesting you think up :)

Preference?

Midterm

- This Friday, 30-60 minutes
- Topics
  - Reading and understanding Java code
  - Writing some java by hand (roughly)
  - Principals we've discussed (structuring code well, taxonomies, etc.)
  - Writing documentation, writing code based on documentation

Visibility

Writing Better Libraries and Code

Private / protected / public

- Public is the most permissive
- Everything else gets in my way
- Why not always use public?

Private / protected / public

- Help your users understand where they should focus
    - E.x.: <u>Socket</u>
- Maintain future flexibility
    - E.x.: <u>Thread</u>
- Signal to other developers

Private / protected / public

- Public
    - What you want users of your code to see
- Protected
    - What you want extenders to your code to see
- Private
    - Implementation details you're abstracting over

Visibility

Property Visibility

- Properties have visibility
- Properties represent internal state of objects
- We want to guard this internal state

Why Guard State?

- Future flexibility
- Change where that state is stored
- Add logic into how state is processed

Getters and Setters

- Don't allow callers to access properties
- Use methods to "guard" all properties

Temperature.java –>

DBRecord.java –>

Inheritance vs. Composition

Inheritance vs. Composition

- **Inheritance**
  Code reuse through **extending** classes

- **Composition**
  Code reuse through **using** other classes

Inheritance

- Push code into a parent class to share code
  - Shared code: Parent
  - Specific code: Child
- Example: Shape#getArea()
  - Square
  - Rectangle
- Fragile, lots of refactoring

Composition

- Have many small classes
- Create instances of functionality specific classes to do your work
- Rely on the type system / inheritance for describing data

JSONSender.java –>

Dependency Injection

Dependency

- When on class relies on another to perform a task
- A good thing!
  - Code reuse
  - Specific, single purpose classes
- Downside
  - Rigidness / inflexible

Solution

- Don't hard code the classes your code uses
- Pass dependencies as parameters
- Use interfaces to make things "safe", but not overly specified

{FileCacher, Expensive}.java –>

Refactoring

Refactoring

- Reorganizing your code
  - improve the structure
  - easier to read
  - better match your data
  - etc

When to Refactor?

- Repetition
    - DRY: Don't Repeat Yourself
- Repetitive code should be moved into:
    - Static classes
    - Other methods
    - Other classes

When to Refactor?

- Indentation
  - If you're more than 4 levels deep, break things up into smaller functions
  - Ex: Linux kernel https://www.kernel.org/doc/html/latest/process/coding-style.html

When to Refactor?

- Complexity
  - Consider code as a call graph
  - How many edges do we have?
  - Lots: very complex, refactor
- Cyclomatic complexity: $M = E - N + 2P$

Complexity.java –>