Patterns Continued: Creational Patterns

July 26, 2017

House Keeping

Final Exam

- August 4th, 12-2pm @ Burnham Hall 208
- Similar structure to midterm
- Covers all class material (e.g. midterm topics are fair game)

Final Projects

- **3rd progress report**
  Friday, July 28 @ noon

- **Final submission**
  Monday, July 31 @ noon

Final Project Submissions

- Host your project on GitHub
- Add TA Minh (jemiar) and I (psnyde2) to have read-write access to your repo

- Add a new file in your CS342 repo called `project/final.md`

project/final.md

- All group members
- If fixing an issue
    - Link to the issue you're fixing
    - Description of how to use / trigger your fix
- If writing an application
    - 1-3 paragraph description of your application
    - Complete description of how to run your application
- Link to GitHub project where we can get code

Final Presentation Schedule

Monday, July 31	Wednesday, August 2
Patrick and Adam	Hubert
Gregory	Casey
Frank and Jake	Julio and Ahel
Victor and Joseph	Alejandro
Ahmed	Li

Bonus Office Hours

- Wednesday, July 26: 2-4pm
- Thursday, July 27: 1-5pm
- Friday, July 28: 2-5pm

Code Patterns

Patterns Review

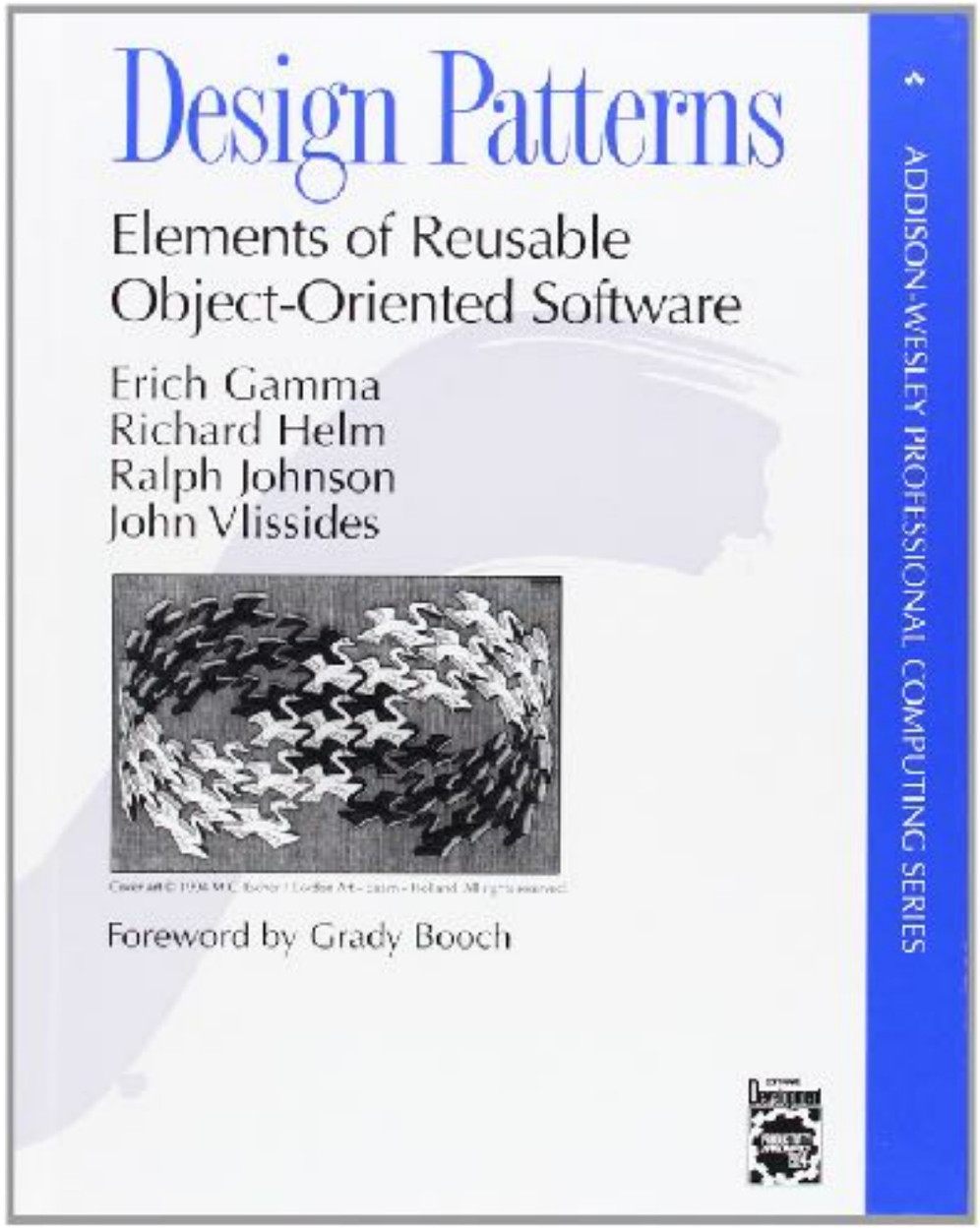
- Application / Architectural Patterns
  - Idioms for structuring an application
  - Example: Model-View-Controller
- Project Patterns
  - Idioms for laying a project out on disk, running tests, dependency management, etc
  - Example: Maven archetypes

Code Patterns

- "Creational patterns"
- Patterns for creating and controlling objects
- Ease, or restrict, the creation of new objects in your system

Design Patterns

- Design Patterns: Elements of Reusable Object-Oriented Software
- "Gang of Four"
- 1994-95
- Effort to establish good practices in object oriented languages



Design Patterns

- Dependency injection pattern
- Object pool / Pool pattern
- Singleton pattern
- Builder pattern
- Lazy initialization pattern
- Factory pattern

Design Patterns

- **Dependency injection pattern**
- **Object pool / Pool pattern**
- **Singleton pattern**
- **Builder pattern**
- **Lazy initialization pattern**
- **Factory pattern**

Singleton Pattern

Problem Scenario

- We're tasked to build some code to represent the configuration of the application
  - Ex. Path to write logs to, port to listen on, etc.
- Only have one configuration possible at a time

singleton/AppConfig.java –>

Problem Scenario

- Multiple configuration instances
- Which is correct? What if they come out of sync?
- Better to have only one instance in the system

Singleton Solution

- Make all constructors private
- Create a static method that can create an instance
- Have a static method that'll create and return that single instance

singleton/AppConfig.java –>

Singleton Conclusion

- Ensures that there can only be one instance
- Applicable for registries, settings, shared connections
- Basically global state, easy to mis-use / abuse

Builder Pattern

Problem Scenario

- Some objects require lots of parameters to instantiate
- The latter ones tend to be optional
- Providing a constructor for every possible set of options is redundant and messy

Example: URI

scheme:[//[user[:password]@]host[:port]][/path][?query][#fragment]

https://www.cs.uic.edu/~psnyder/cs342-summer2017/

mailto:psnyde2@uic.edu

ftp://psnyde2:margeforme@ftp.uic.edu/~psnyde2/example.file

builder/CS342URL.java –>

Problem Definition

- Shouldn't allow invalid / inconsistent state
- Providing all possible constructors is infeasible
- How to have optional parameters, with sane constructors

Builder Solution

- Split creation task into two classes
- Target class
  - Cannot be directly instantiated
  - Takes a large number of optional parameters
- Builder class
  - Is able to instantiate target class
  - Builds the object in steps

builder/CS342URL.java –>

Builder Overview

- Useful when objects have lots of state that must be kept in sync
- Identifiable by <Target> and <Target>Builder
- Bandaid for Java lacking optional, named parameters in Java
- Examples:
  - URLBuilder - URLBuilder
  - OKHttp - Request.Builder()

Lazy Initialization pattern

Problem Scenario

- Some properties and data in classes takes a long time to generate
- A program will only use a subset of functionality in a program
- How to only pay for the functionality we need?

Problem Example

- We have classes that represent data in the database

- We'd like to only connect to the database when it's needed

- Creating the database connection is slow and expensive

lazy/DatabaseData.java –>

Lazy Initialization Solution

- Identify properties / data that are expensive to create
- Delay creating them
- Reuse instances you've already payed the price for

singleton/AppConfig.java –>

Lazy Initialization Solution

- Useful for delaying expensive, maybe needed costs

- Can be abused / anti-pattern

- If over used, can make it difficult to reason about what operations are fast and slow