Johns Hopkins Engineering

Software Patterns

Module: Introduction

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Software Patterns - 605.707.31

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

- Christopher Alexander

Goals

- Initial Catalog
- Recognition
- Application

Instructional Format

Teaching Method

- Socratic

  Discussion Leading

  Participation

- Minimal lecturing

- Final Project

Grading

Discussion

- Participation (20%)
- Weekly Assignments (30%)
- Discussion Leadership (20%)

Final Project (30%)

- Pattern Study
- Re-architect Assignment

Syllabus

Week 1-13
- Patterns from “Design Patterns”
- Other kinds of Patterns

Week 15
- Project presentation

Johns Hopkins Engineering

Software Patterns

Module: History of Patterns

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Why Patterns

- Someone else has already solved the problem
- Others have confirmed that the solution works
- Patterns provide a higher level of abstraction than objects
- Patterns evince the principles of object orientation

History of Patterns

Christopher Alexander - architect

- Theory of Patterns
- "A Pattern Language"
- Involve the user in the design of a building

Ward Cunningham and Kent Beck (XP)

- Five-pattern language for Smalltalk programs
- Observed improved user interfaces in students' programs

History of Patterns

Erich Gamma

- PhD Thesis on ET++
- Met Richard Helm and began cataloging design patterns

James Coplien

- Catalogued C++ patterns (idioms) for "Advanced C++ Programming"

Hillside Group

- Brought all of these people together in 1993 to discuss objects and patterns
- Planned first PLoP conference in 1994

History of Patterns

Gang of Four

- Also came out in 1994
- Sold phenomenally at OOPSLA

PLoP

- Published PLoPD (up to vol. 4 now)
- Spawned ChiliPLoP, EuroPLoP, KoalaPLoP, etc.

What is a Pattern?

A means of capturing knowledge

- Someone has seen the problem before

A reusable artifact

- More easily reused than code

What is a Pattern?

A design pattern systematically names, motivates, and explains a general design that addresses a **recurring design problem** in object-oriented systems. It describes the **problem**, the **solution**, **when to apply** the solution, and its **consequences**. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.

- GoF

Tools Used by Patterns

Composition instead of Inheritance

Interaction through Interfaces instead of Implementations

Additional Layers of Indirection

Reallocation of Roles

What Are The Parts of a Pattern?

Name

- Handle or identifier that describes the pattern in one or two words

Problem Statement

- The problem and the context in which it occurs

Solution

- Parts of the design and their relationships

Consequences

- Good and bad results of using the pattern

Johns Hopkins Engineering

Software Patterns

Module: Null Object – Introductory Discussion

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Null Object

Null Object Pattern (PLoP3, Bobby Woolf)

- Intent: A Null Object provides a surrogate for another object that shares the same interface but does nothing.

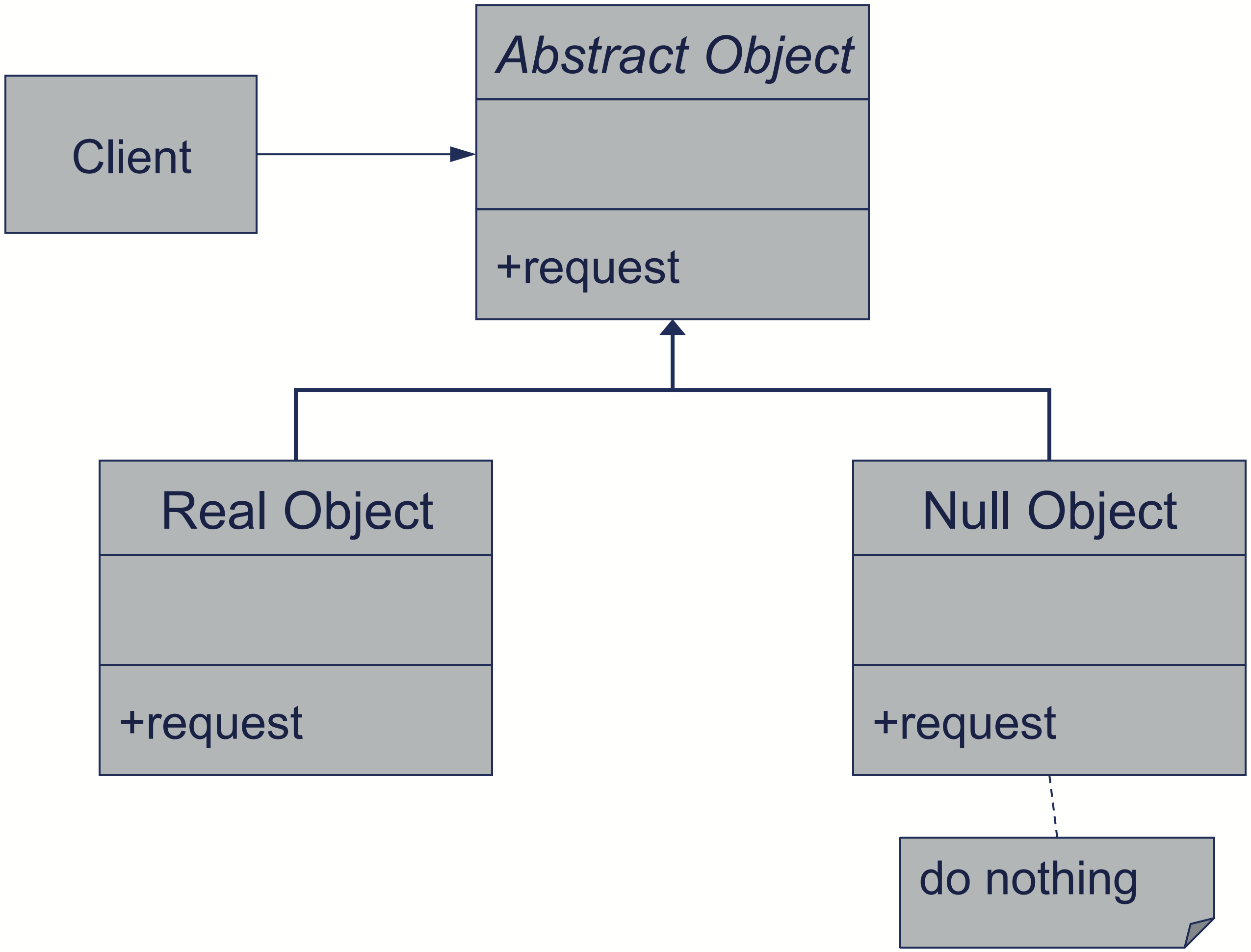
- AKA: Active Nothing

Null Object

Participants:

- Client: requires a collaborator with a specific interface
- Abstract Object: defines a type and declares the interface, implements default behavior common to all classes
- Real Object: defines a concrete subclass of Abstract Object whose instances provide useful behavior that Client expects
- Null Object: provides an interface identical to Abstract Objects, implements its interface to do nothing. When there is more than one way to do nothing more than one Null Object may be required.

Null Object


Client
Abstract Object
+request
Real Object
+request
Null Object
+request
do nothing

Null Object Consequences

Uses polymorphic classes.	*Forces encapsulation.*
Simplifies client code.	*May cause class explosion.*
Encapsulates do nothing behavior.	*Forces uniformity.*
Makes do-nothing behavior reusable.	*Is non-mutable.*

Null Object Questions

- Does Null Object work as a list terminator?
  - e.g., instead of:
    - *for (Obj o = list.start(); o != NULL; o=o.next()) {}*

- What behaviors would not benefit from representation as a Null Object?

- At what level is it appropriate to implement the null object behavior? I.e., does every interface in a type hierarchy need one?

Johns Hopkins Engineering

Software Patterns

Module: Forms of Patterns

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

What is a Pattern Not?

- A panacea
  - “When someone says, ‘I want a programming language in which I need only say what I wish done,’ give him a lollipop.” - Alan Perlis (Yale University)
- A recipe
- A template
  - “The solution is customized and implemented to solve the problem in a particular context.”

What Isn’t a Pattern?

- Language Feature
  - Polymorphism
  - Encapsulation
- Style
  - Code indentation isn’t a pattern. (But there could be a pattern that describes how to indent code.)
- Principle

How are patterns used?

- Analysis
  - How to understand the requirements of a system.
  - "Analysis Patterns," by Martin Fowler
- Design
  - How to convert requirements into programs
  - How to implement the programs
- Refactoring
  - Fixing "bad" programs

Types of Patterns

Architecture

- Layers

Design

- Singleton

Idiom

- Envelope

Analysis

- Units and Measurements

Organizational/ Process

- Work Queue, Size the Organization

Anti-

- Big Ball of Mud, Cargo Cult

Forms of Patterns

- Alexandrian essay style
- Configuration of objects and protocols as in GoF
- Programming idioms
- Story
- Generative
  - Pattern Languages

What makes a pattern good?

- QWAN
  - What are the measures of goodness for software?
  - Is there more than "It meets the requirements?"

- Principles followed by GoF
  - Design to interfaces.
  - Favor composition over inheritance.
  - Find what varies and encapsulate it.