



Retrospect and Prospect

- We've covered a lot of ground to this point
 - Requirements analysis
 - (aka requirements, aka analysis)
 - Design
 - (Hence, OOA/D)
 - Unified process - UP
 - (Inception, elaboration, iterations, etc.)
 - UML
 - (use cases, sequence & collaboration diagrams, class diagrams, etc.)
 - GRASP Design patterns (Chapters 16-17 & 21)



Retrospect and Prospect in Context

- Note the textbook title:
 - "Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design and the Unified Process"
- We've done a lot on both clauses
- We're going to do more
 - Especially, more on...
 - ...the "more important" 1st clause



Retrospect and Prospect in Context

- Textbook title: "Applying UML and **Patterns**: an Introduction to Object-Oriented Analysis and Design and the Unified Process"
- We've done 3 chapters on **patterns**
 - Start another (chapter 23) today
- Then more on use cases, domain models, UML (chapters 25-29)
- **Patterns** occur again in Chapter 30
- More **patterns** again (33-34)
- **Patterns** chapters are longer than average, too
 - (We won't have time to cover all but will cover the most important material)



GoF Patterns

- We've seen GRASP patterns
- There are also "GoF" patterns
 - Basically, these are just more patterns
 - ...no big conceptual difference between the categories "GoF" and "GRASP"
- GoF = **Gang of Four**
 - From the book *Design Patterns*, by 4 authors
 - (+ allusion to Chinese political history)



The Adapter Pattern

- This pattern says, *It is good to "provide a stable interface to similar components with different interfaces"*
- Example: an object that calls the installed 3rd party sales tax package
- See slide from last time for review...



Protected Variations Pattern

- polymorphism pattern, indirection pattern, pure fabrication pattern, and **protected variations pattern**
- **Example: Bundle the different tax calculator methods into a special class that acts as an interface to them, calling the right one**
- **This protects Sale from changes to the number of such methods and changes to their details**
- **This new class can be used in Sale, but changes to tax calculator vendors no longer require changes to the Sale class**
 - (Which would be counter-intuitive)

Protected Variations Pattern

- polymorphism pattern, indirection pattern, pure fabrication pattern, and **protected variations pattern**
- Recall section 22.4:
 - protected variations patterns says to
 - create a stable interface that
 - separates variations from other parts of the system
- Example:
 - multiple 3rd-party tax calculators,
 - each with its own idiosyncratic interface

The Adapter Pattern

- Recall (from slide 5) that this pattern says,
 - It is good to "provide a stable interface to similar components with different interfaces"*
- It hides the different interfaces of the different vendors
- It provides one interface to the POS system
- It does the work of figuring out how to call the 3rd party package
- Other 3rd party packages used by POS could include:
 - Credit card authorization systems, database systems, accounting systems, inventory systems,...
 - It's good to use 3rd party packages in the POS, when available
 - Why?

Adapter Pattern Vs. Protected Variations Pattern

- Protected Variations pattern (GRASP pattern)**
 - Problem:
 - designing objects, subsystems and systems so that...
 - ...variations or instabilities in them do not...
 - ...have an undesirable impact on other elements
 - Service Lookup is an example of PV because
 - Lookup service clients are protected from variations in...
 - ...the location of services
 - Another example of PV:
 - Externalizing properties in a property file
 - Adapter pattern (GoF pattern)**
 - Problem:
 - resolving incompatible interfaces, by...
 - ...providing a stable interface to similar components with different interfaces
 - Which is a subset of which?**
- (paraphrased from book)

Adapter Pattern – Naming Conventions

- What to call the 3rd party sales tax software caller class?
- How about: **calcSalesTaxAdapter**
- ...It is often good to put the *pattern name* in a *module name* that embodies the pattern
 - (recall "module")

The Factory Pattern

- Problem: how to create complicated objects
- Solution: define a factory
- Conceptually, a constructor is a kind of factory
- In reality,
 - a constructor it is called a "constructor"
 - other ways of creating objects are called "factories"
- In Java, factory methods can return more kinds of things than constructors
 - they can refuse to create
 - they can create subclass objects

Factory Pattern II

- In Java, factory methods can return more kinds of things than constructors
 - they can refuse to create an object
 - they can create + return subclass objects
- The Calendar abstract class has getInstance(...) methods that return calendars of various kinds
- POS system: factory getSale(register, cashier)
 - Could return a sale object as long as both args non-null
 - If *both* args null, refuse
 - getSale (Sale sale) could
 - return a new object if sale is null,
 - else re-initialize sale
- getShape() could create+return different shapes randomly for a shape area quiz/exercise system for high school students

Singleton Pattern

- Problem:
 - Want to allow only one instance of a class
 - (you can call that instance a *singleton*)
- Constructors can't enforce that
- A static method can
- Recall class diagrams have arcs with numbers (e.g. Sale ----- SalesLineItem)
 - ...so there can be a need for a singleton
- Need for any limitation to # of objects is similar
 - (and for that, static member variables won't do)

Strategy Pattern

- Example:
 - Pricing can change frequently
 - A sale can last for 1 afternoon and involve 10% off everything
 - Sales tax on school items can be 0 for certain days
 - Etc., etc. (I'm not a big shopper but use your imagination)

Strategy Pattern II

- Problem:
 - Design for varying, related computations
 - That is, make it easy to change the computation
- Solution:
 - Put each variation in its own class/module
 - Make a single interface to all of them

Strategy Pattern III

- Solution:
 - Put each variation in its own class
 - Make a single interface to all of them
- Each variation is a "strategy"
- The interface is a kind of *adapter*

GoF Review

- We've looked at GoF patterns
 - Adaptor, Factory, Singleton, Strategy
- Which is like a constructor but can do more things?
- Which is useful when only 1 object of a class should be creatable?
- Which is useful when only 2 objects of a class should be creatable?
- Which suggests one interface for several similar services whose definitions
 - are outside our control?
 - are part of our design?

GoF Review II

- We've looked at GoF patterns
 - Adaptor, Factory, Singleton, Strategy
- How do factories differ from constructors?
- How do adaptors differ from strategies?
- How does the singleton pattern idea extend to be more powerful than just using the static concept?