SENG5802: Software Engineering II - Software Design
More patterns and Inversion of Control

Outline

Outline

Open-Closed Principle

Bertrand Meyers (1988) stated it first

Key lesson of OO design

Open to extension

Many standard design patterns are intended to provide extensibility

- Strategy
- Bridge
- Abstract Factory
- **Decorator**

Code against interfaces; avoid dependence on concrete classes.

Closed to change

Sometimes...

Time to refactor!

Test **before** adding new feature

Example (1/2)

Imagine we are writing an application for an online bookstore.

One of the requirements is to be able to display just the books published in the last year.

So you add a Filter class to your domain model with an appropriate filtering method. Or maybe you add a method to your book repository.

Example (2/2)

Your next job is to add the ability to show just the books in the user's language.

So you add another method to your filter.

This is not closed.

How could we change this design to make it open-closed?

Encapsulating Variability

Key: encapsulate just one source of variability at a time – if you can.

Avoid unnecessary coupling and potential combinatorial explosions

Consider context. There are few design rules that should never be broken.

Analysis Matrix

Find abstract concepts; these are the rows.

Enumerate usage scenarios with variations/special cases; these are the columns.

In each cell, identify the variations for the abstract concepts in those special cases.

Outline

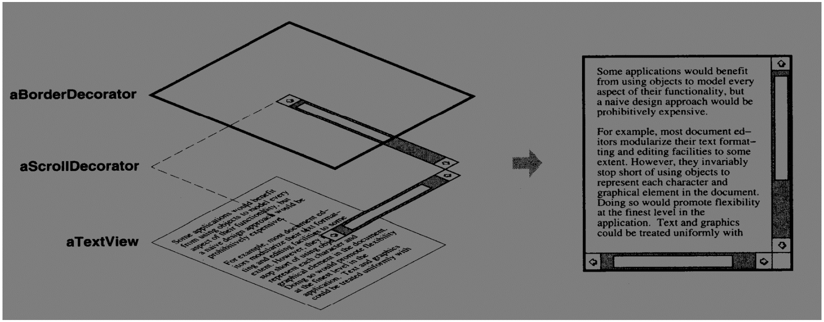
Decorator

Problem: add attributes to an object *at runtime*

Pattern conditions:

1. Each decoration “wraps” the thing it decorates
2. Client keeps track of the *last* decorator added

Motivation



Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document ed– itors modularize their text format– ting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

Outline

Composite

Problem: Represent part-whole hierarchies in a flexible way

Client can treat composite structure of arbitrary depth as a single object.

Operation:

```
for c in children do
    c.operate();
this.operate();
```

Composite Variation

If you need to build and navigate the tree...

- move `add()`, `remove()`, and `getChildren()` up
- make Component abstract

Outline

Observer pattern

Common barrier to better modularity:
The need to maintain consistent state among parts of a system, or to have one part "know about" changes in other parts.

Subject: part with changing state
Observer(s): part(s) that need to know about changes

Obvious approach: Subject directly informs Observers when it changes state.

GOF version

This is the GOF Observer. There are many like it. But, this one is theirs.

Dependency Inversion

Observers can act on changes to the Subject.

In non-OO systems, Observer is known as a “callback” interface.

Using Observer

Some useful approaches:

1. Message (“This event happened.”)
2. Self-reference passing
3. Encapsulate the change in an Event class
4. Observer registers a Strategy object, called by Subject

Order and utility are another concern

Registration is often inconvenient

Observer in the wild!

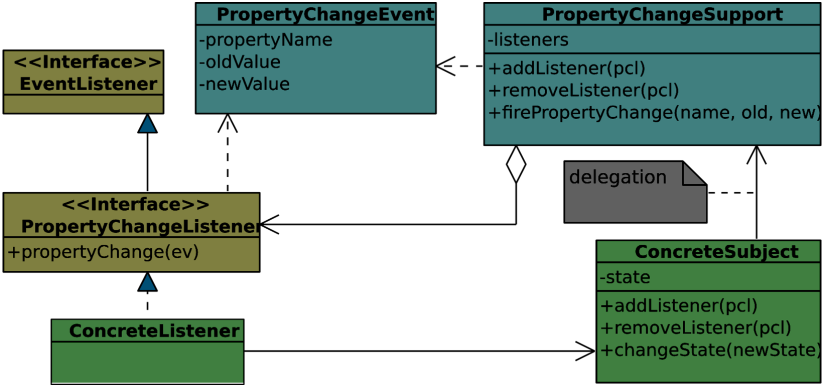


Figure: Copyright ©John Collins