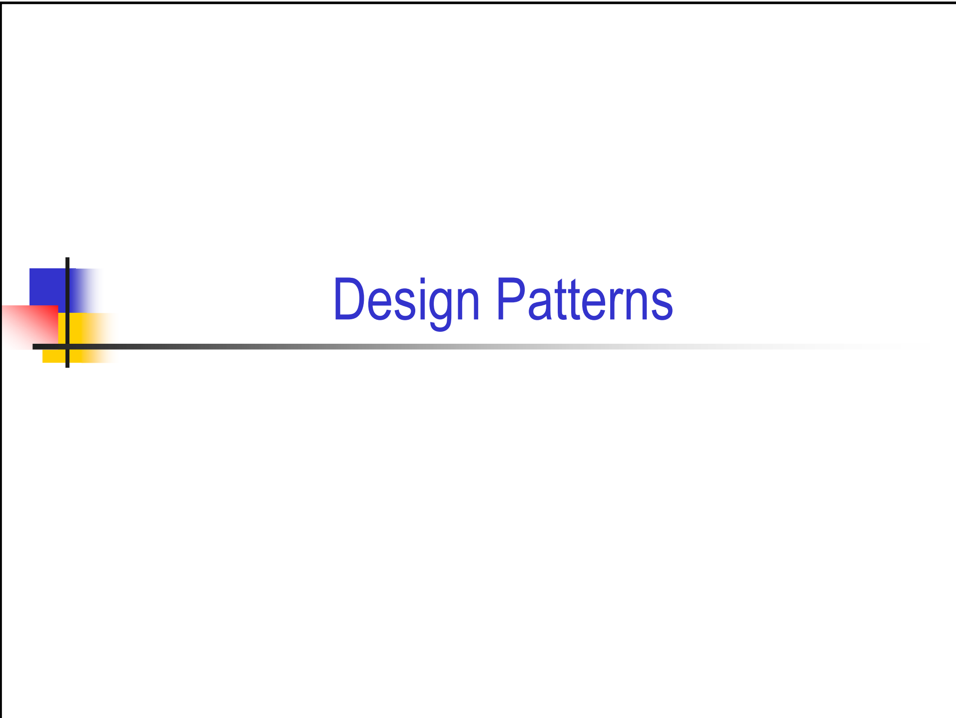


Introduction to Patterns

- The recurring aspects of designs are called *design patterns.*
  - A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context
  - Many of them have been systematically documented for all software developers to use
  - A good pattern should
    - Be as general as possible
    - Contain a solution that has been proven to effectively solve the problem in the indicated context.

  Studying patterns is an effective way to learn from the experience of others

Motivation for Design Patterns

- Most software systems contain certain common aspects that are frequently reinvented for each system
- Solutions to these common problems may vary in quality from system to system
- Design patterns seeks to communicate these classic solutions in an easy to understand manner

What are Design Patterns?

- Design Patterns communicate solutions to common programming problems
- The seminal book on design patterns, *Design Patterns, Elements of Reusable Object-Oriented Software* by Gamma et al, identifies three categories of design patterns
  - Creational
  - Structural
  - Behavioral

Pattern description

Context:
- The general situation in which the pattern applies

Problem:
- A short sentence or two raising the main difficulty.

Forces:
- The issues or concerns to consider when solving the problem

Solution:
- The recommended way to solve the problem in the given context.
  - 'to balance the forces'

Antipatterns: (Optional)
- Solutions that are inferior or do not work in this context.

Related patterns: (Optional)
- Patterns that are similar to this pattern.

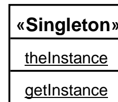
References:
- Who developed or inspired the pattern.

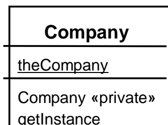
The Singleton Pattern

- ***Context***:
  - It is very common to find classes for which only one instance should exist (*singleton)*
- ***Problem***:
  - How do you ensure that it is never possible to create more than one instance of a singleton class?
- ***Forces***:
  - The use of a public constructor cannot guarantee that no more than one instance will be created.
  - The singleton instance must also be accessible to all classes that require it

Singleton

- ***Solution:***





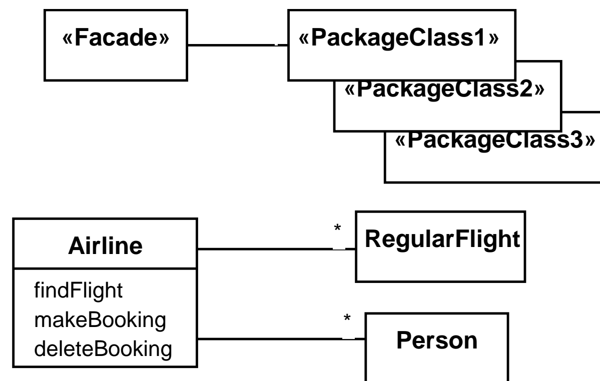

```
if (theCompany==null)
  theCompany= new Company();

return theCompany;
```


The Controller Façade Pattern

- ***Context***:
  - Often, an application contains several complex packages.
  - A programmer working with such packages has to manipulate many different classes
- ***Problem***:
  - How do you simplify the view that programmers have of a complex package?
- ***Forces***:
  - It is hard for a programmer to understand and use an entire subsystem
  - If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.

Façade

- ***Solution:***

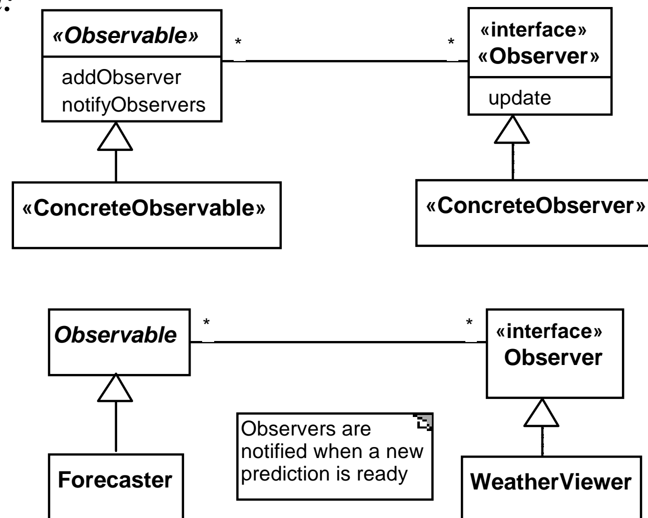


The Observer Pattern

- ***Context***:
  - When an association is created between two classes, the code for the classes becomes inseparable.
  - If you want to reuse one class, then you also have to reuse the other.
- ***Problem***:
  - How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?
- ***Forces***:
  - You want to maximize the flexibility of the system to the greatest extent possible

Observer

- *Solution:*



Observer

- Antipatterns:
  - Connect an observer directly to an observable so that they both have references to each other.
    - Observers "poll" observables for changes
    - Observerables "call" update methods directly
  - Make the observers *subclasses* of the observable.

Pattern Difficulties and Risks

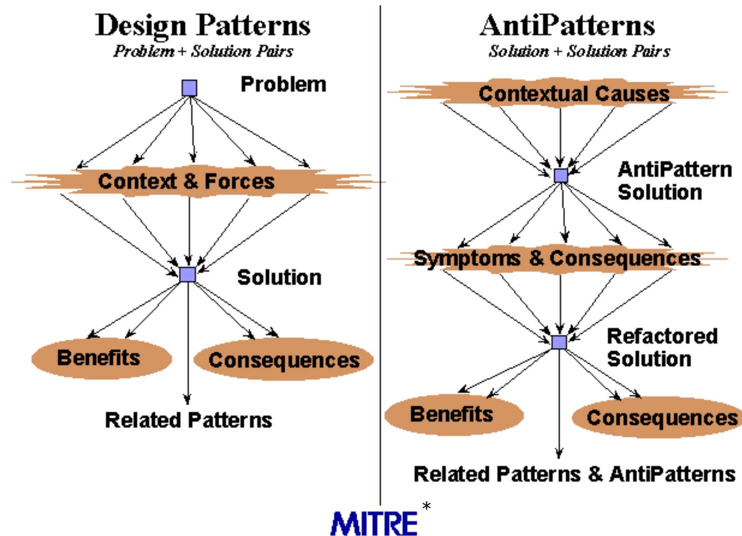
- **Patterns are not a panacea:**
  - Whenever you see an indication that a pattern should be applied, you might be tempted to blindly apply the pattern. However this can lead to unwise design decisions .
- *Resolution:*
  - *Always understand in depth the forces that need to be balanced, and when other patterns better balance the forces.*
  - *Make sure you justify each design decision carefully.*

Pattern Difficulties and Risks

- **Developing patterns is hard**
  - Writing a good pattern takes considerable work.
  - A poor pattern can be hard to apply correctly
- *Resolution:*
  - *Do not write patterns for others to use until you have considerable experience both in software design and in the use of patterns.*
  - *Take an in-depth course on patterns.*
  - *Iteratively refine your patterns, and have them peer reviewed at each iteration.*

Evaluating Designs

- The application of "well-known" design patterns that promote loosely coupled, highly cohesive designs.
- Conversely, identify the existence of recurring *negative* solutions – AntiPatterns
- AntiPattern : use of a pattern in an inappropriate context.
- Refactoring : changing, migrating an existing solution (antipattern) to another by improving the structure of the solution.

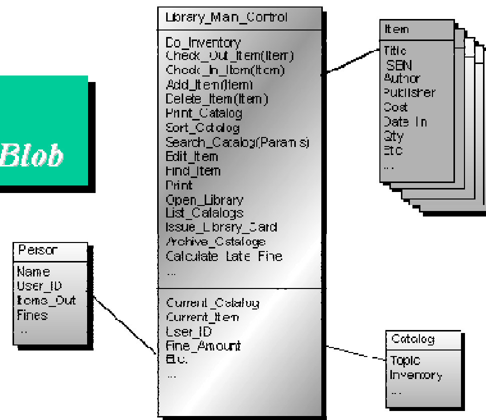


*Slides from MITRE organization : www.mitre.org

Development AntiPattern:

The Blob

Example: The Library Blob



MITRE

Development AntiPattern:

The Blob

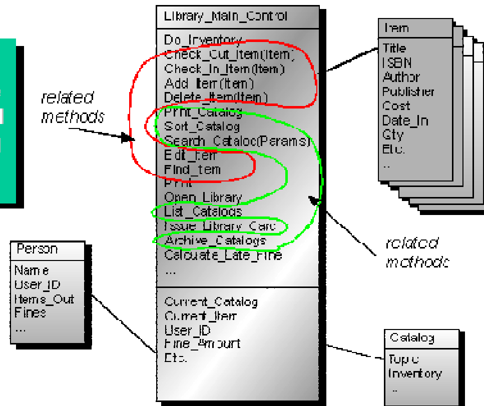
- **Symptoms**
  - Single class with many attributes & operations
  - Controller class with simple, data-object classes.
  - Lack of OO design.
  - A migrated legacy design
- **Consequences**
  - Lost OO advantage
  - Too complex to reuse or test.
  - Expensive to load

MITRE

Development AntiPattern:

The Blob - Refactoring

Step 1:
Identify or categorize related attributes and operations according to contracts.

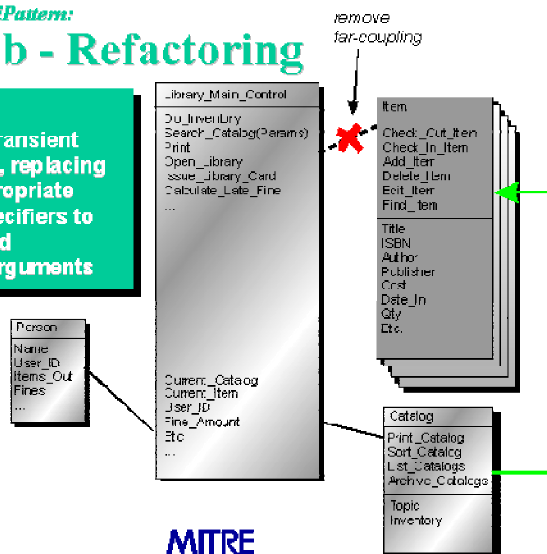


MITRE

Development AntiPattern:

The Blob - Refactoring

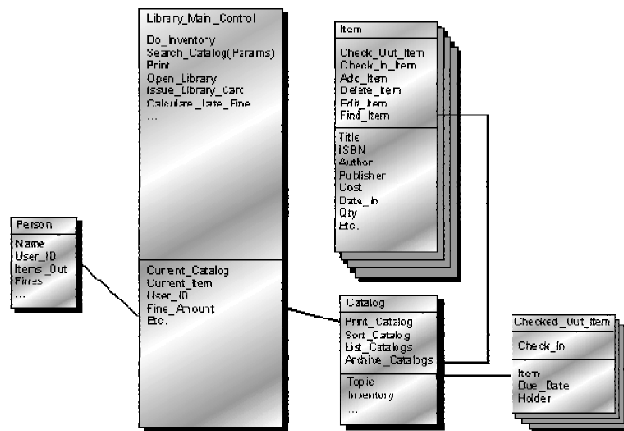
Final Step:
Remove all transient associations, replacing them as appropriate with type specifiers to attributes and operations arguments



MITRE

Development AntiPattern:

The Blob Refactored



MITRE