Antipatterns

Negative solutions that leads *more* problems than they are indented to solve

Design patterns

- Time tested *solution* to a recurring problem in a specific context.
- Benefits exceeds negative consequences

Antipatterns

- Commonly occurring *solution* to a problem that generates bad results
  - Repeated software failures
  - common pitfalls to avoid
- Negative consequences exceeds benefits

Jim Coplien: "an anti-pattern is something that looks like a good idea, but which backfires badly when applied."

Antipatterns

- negative consequences.
  - code harder to read, maintain, test, and extend, ultimately reducing the overall quality

Importance:

- Identifying bad practices can be as valuable as identifying good practices.
  - *"It's not fun documenting the things that most people agree won't work, but it's necessary because many people may not recognize the AntiPattern."*

http://c2.com/cgi/wiki?AntiPattern

Anti-pattern: Result of

- poor design choices, lack of experience, misunderstanding of best practices
- applying a perfectly good pattern in the wrong context.

> "The resulting code tends to be full of factories that only ever return a single thing, that thing being a proxy to a delegate that wraps a class that has a method that has the three lines of code you actually want to execute."

- **Our Aim:** diagnose and fix the common failures we will inherit in real-world code.

Categories

1. Software Development antipatterns
   - Bad Coding/design solutions
2. Architectural antipatterns
   - System-level antipatterns.
   - Common mistakes in creation, implementation, and management of architecture.
3. Project Management antipatterns

- Antipattern web site www.antipatterns.com

1) Poltergeist

- short-lived, stateless objects created to perform simple tasks and then *discarded.*
  - These objects often serve as intermediaries that add unnecessary complexity to the system.

- A.k.a. Proliferation of classes
- Poltergeists add no significant value
  - only perform a single action
  - or pass data between other objects
  - without maintaining any state or long-term responsibility

Activity

AuditLogger.log("Task process started"); before the task process begins.

```
class TaskInitializer {
    public void initializeTask(String name) {
        Task task = new Task(name);
        TaskExecutor executor = new TaskExecutor();
        executor.executeTask(task);
    }
}
class TaskExecutor {
    public void executeTask(Task task) {
        task.execute();
    }
}
class TaskManager {
    public void manageTask(String name) {
        TaskInitializer initializer = new TaskInitializer();
        initializer.initializeTask(name);
    }
}
```

Poltergeist:

- only to perform a single action or pass data between other classes,
- without maintaining any state or long-term responsibility

```
public class Main {
    public static void main(String[] args) {
        TaskManager manager = new TaskManager();
        manager.manageTask("Sample Task");
    }
}
```

```
class TaskInitializer {
    public void initializeTask(String name) {
        Task task = new Task(name);
        TaskExecutor executor = new TaskExecutor();
        executor.executeTask(task);
    }
}
class TaskExecutor {
    public void executeTask(Task task) {
        task.execute();
    }
}
class TaskManager {
    public void manageTask(String name) {
        Task task=new Task(name);
        task.execute();
    }
}
                                    public class Main {
                                        public static void main(String[] args) {
                                            TaskManager manager = new TaskManager();
                                            manager.manageTask("Sample Task");
                                        }
                                    }
```

```
class TaskExecutor {   //NOT a poltergeist anymore
    private Queue<Task> taskQueue = new LinkedList<>();

    public void addTask(Task task) {
        taskQueue.add(task);
    }

    public void executeTasks() {
        while (!taskQueue.isEmpty()) {
            Task task = taskQueue.poll();
            task.execute();
        }
    }
}
```

- maintains a queue of tasks and has methods to add and execute tasks.
- a long-lived object with clear responsibilities.

```
class TaskExecutor {   //NOT a poltergeist anymore
    private Queue<Task> taskQueue = new LinkedList<>();

    public void addTask(Task task) {
        taskQueue.add(task);
    }

    public void executeTasks() {
        while (!taskQueue.isEmpty()) {
            Task task = taskQueue.poll();
            task.execute();
        }
    }
}
```

```
class TaskManager {
    private TaskExecutor executor = new TaskExecutor();
    public void manageTask(String name) {
        Task task = new Task(name);
        executor.addTask(task);
    }
    public void executeAllTasks() {
        executor.executeTasks();
    }
}
```

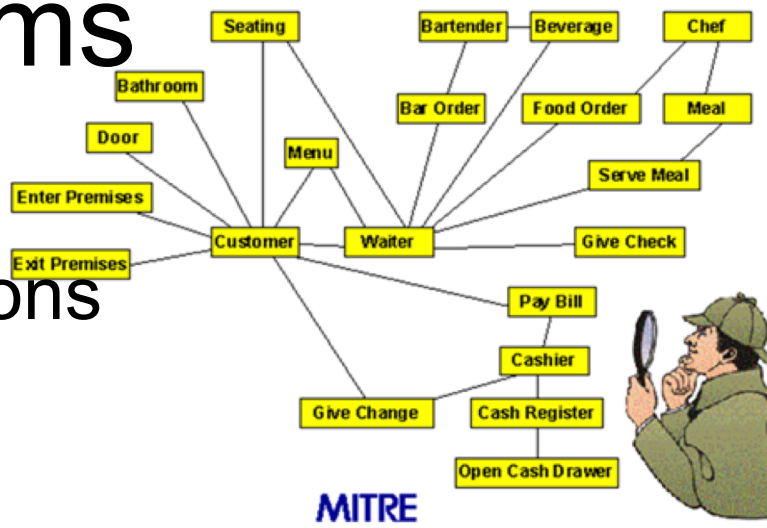
Causes

- When designers familiar with *process modeling* but new to define architectures
- **Poor object design:** Creating short-lived, stateless objects that only serve to invoke methods in other classes.
- **Over-engineering:** Anticipating future needs that may never materialize.
- **Misuse of design patterns:** Applying patterns like ***Command*** or ***Factory*** without a real need.

Question

- Proxy vs poltergeist?
  - Is there an added value?
- Façade vs poltergeist?
  - What is the added value?

Poltergeist: Symptoms



- Redundant navigation paths.
- Spurious classes & associations
  - Transient associations.
  - Stateless classes.
- Temporary, short-duration objects and classes.
- Classes with few responsibilities
  - Single-operation classes that exist only to "seed" or "invoke" other classes through temporary associations.
- Classes with "control-like" operation names such as start_process_alpha.

Poltergeist: Consequences

- Poltergeists clutter software designs, creating unnecessary abstractions
- Increases complexity and maintenance costs.
  - Excessive complexity, hard to understand, and hard to maintain.
- Makes the system harder to understand and debug.
- Wastes resources and impacts performance.
  - They are unnecessary, so they waste resources every time they "appear."
- inefficient because they utilize several redundant navigation paths.

Refactoring Poltergeist

Delete them

1. Refactor classes with short lifecycles and no responsibilities
   - Move them into collaborating classes
   - Regroup to form a cohesive class

2. Delete the poltergeist and insert its functionality in the invoked class

How to avoid

- YAGNI Principle:

  "You Aren't Gonna Need It"

  - don't add complexity for hypothetical future requirements.

- Identify and refactor poltergeist objects early
  - regular code reviews or refactor often

2) Lava flow

What do you do when you see this?
What are your *real* options?

```
…
// This class was written by someone earlier (Alex?) to manage the indexing
// or something (maybe).  It's probably important.  Don't delete.  I don't
// think it's used anywhere - at least not in the new MacroINdexer module which
// may actually replace whatever this was used for…
class IndexFrame extends Frame
{
    // IndexFrame constructor
    //-----------------------------------------------------------------------
    public IndexFrame(String index_parameter_1)
    {
    // Note: need to add additional stuff here…
    super (str);
    }
    //-----------------------------------------------------------------------
…
```

Painful to cure: not easy to throw away unnecessary parts

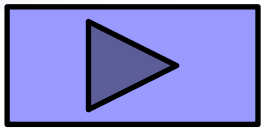
Lava flow

- **Lava Flow** antipattern:

  Dead code that is too scary to remove

- Dead code or forgotten design information are frozen in an ever-changing design.
  - e.g. Legacy code no one dares to touch
  - e.g. Quick fixes that became permanent
- Code becomes expensive to analyze, verify and test.
- **Common Causes:**
  - R&D code or "abandoned experiments" left in production.
  - Lack of code reviews.
  - Lack of version control

Lava flow: Symptoms



- Frequent unjustifiable variables and code fragments
- **Code that seem important**, but no one can really explain what they do or why they exist
- Undocumented complex, important-looking classes or segments which don't clearly relate to the architecture.
- **Whole blocks of commented-out code** with no explanation or documentation
  - The famous comment:

"// Don't delete this, not sure why it's here but it breaks things."

- Unused (dead) code, just left in. Unused, inexplicable or obsolete interfaces

Typical causes

- R&D code placed into production without thought toward configuration management.
- Uncontrolled distribution of unfinished code. Implementation of several trial approaches toward implementing some functionality.
- Lack of versioning system, configuration management or compliance with process management policies.
- Lack of architecture, or non-architecture-driven development.
- Architectural scars.
  - Some architectural commitments are found not to work later in the development. The system architecture may be reconfigured, but these inline mistakes are seldom removed.
  - especially hard in a development environments with hundreds of source code individual files

Lava Flow – Consequences

- Increased maintenance costs
- Higher risk of bugs and errors
- Expensive to analyze, verify and test.
- Consumes memory and resources, impacting performance.
- Proliferates **if not removed**, as code is reused.
- If processes causing to lava flow is not checked, it will lead to geometric growth of issues.
  - produce new, secondary flows as they try to "work around the original ones.
- As the flow hardens, understanding the architecture to make documentation and improvements becomes impossible.

Cure: prevention

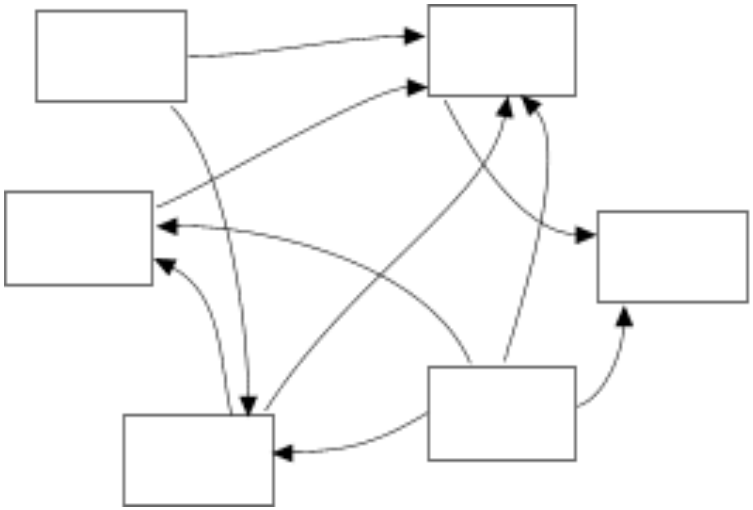
- Use versioning and analysis tools.
  - Version control: git blame will tell who wrote it when
  - Static analysis to tell if anything still calls this class.
- **Code reviews**.
  - stops new Lava Flow from forming.
- **Regular Refactoring**: Continuously clean up and refactor code to remove obsolete or unused parts.
- Left uncured, leads to **"Boat Anchor "** antipattern
  - a piece of code that does nothing but hold the project down.
  - The same cure: *Have the courage to delete dead code.*

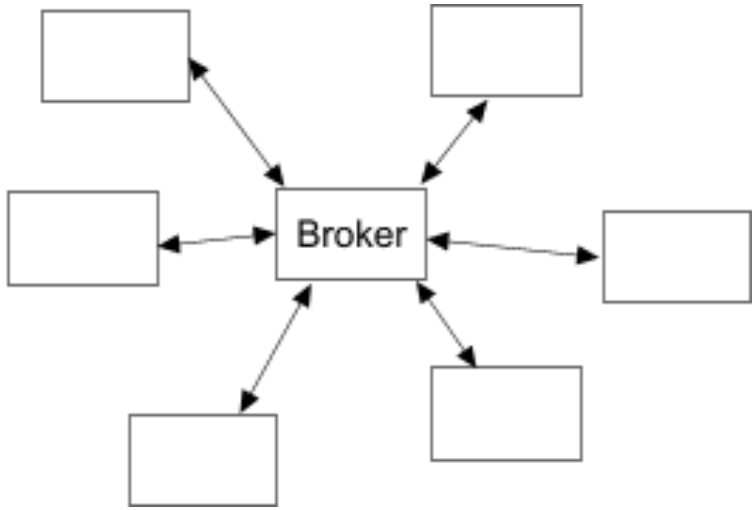
Anectodes…

- "Ugh! What a mess!"
- "You *do* realize that the language supports more than one function, right?"
- "It's easier to rewrite this code than to attempt to modify it."
- "Software engineers don't write spaghetti code."
- "The quality of your software structure is an investment for future modification and extension.

3) Spaghetti design

- Unstructured code or design that is hard to maintain





working with a spaghetti

Activity: you got two tickets. Resolve them.

```
double calculateInterestRate(){
  if(maturity==0){  //new customer
     double capital= calculateAssets()+montlyIncome;
      return riskFactor(capital)*1.75;
  }else if(maturity<5){  //fairly new customer
    double capital= calculateAssets()+montlyIncome;
     return riskFactor(capital)*0.85;
  }else if(maturity>10){ //mature customer
    if(stable) return currentInterestRate*0.8
    else return currentInterestRate;
  }
  return 0.0;
}
```

- **Bug Report:** A customer with 7 years maturity. Their interest rate is 0.0!
- **New Feature:** Add a 'Super-Mature' customer type for maturity > 20. Their rate is currentInterestRate * 0.7

Questions

- Where is the bug?
- Where do you add the `else if` for the new feature?
- How "brittle" does this code feel?

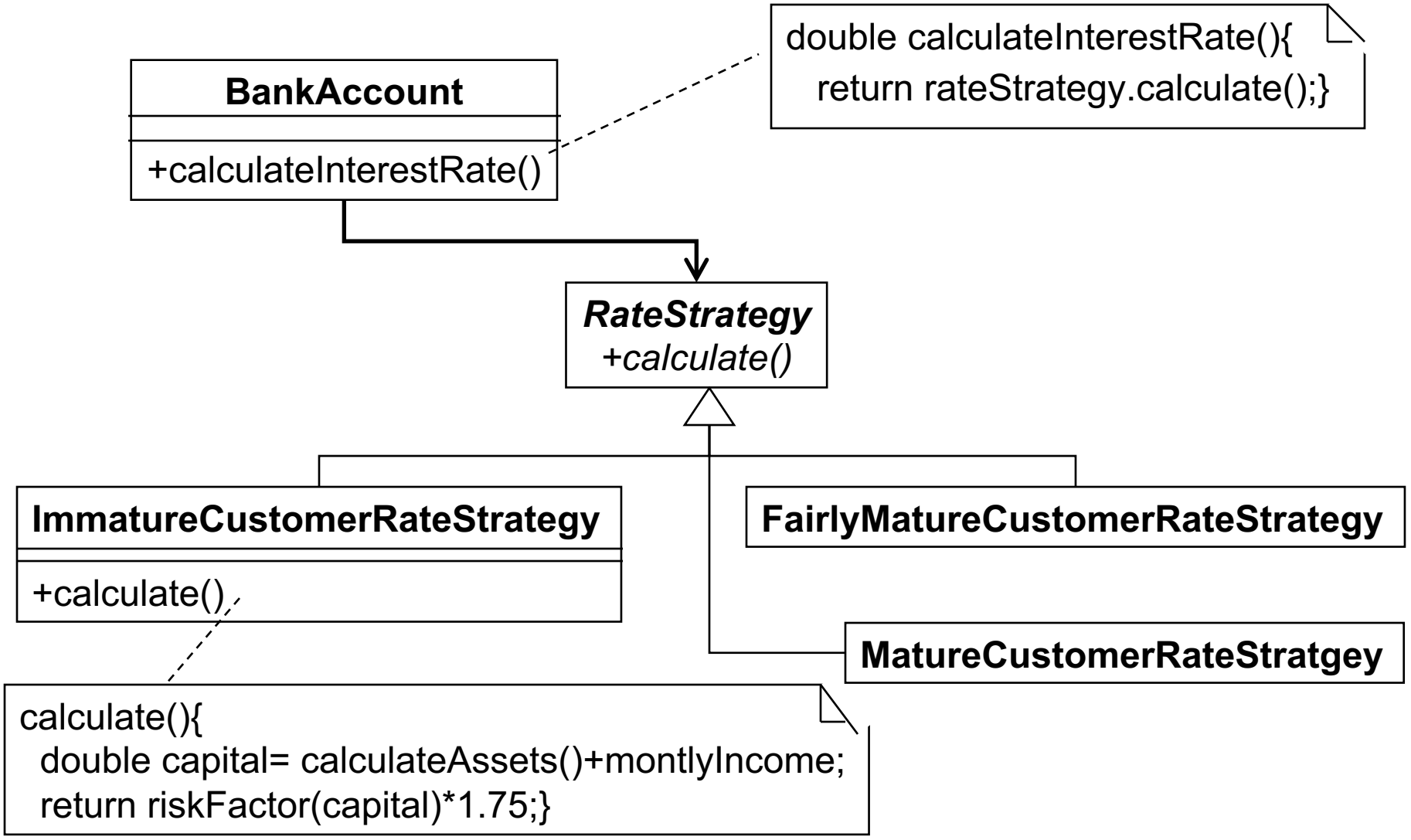
Complex Conditionals

Solution:

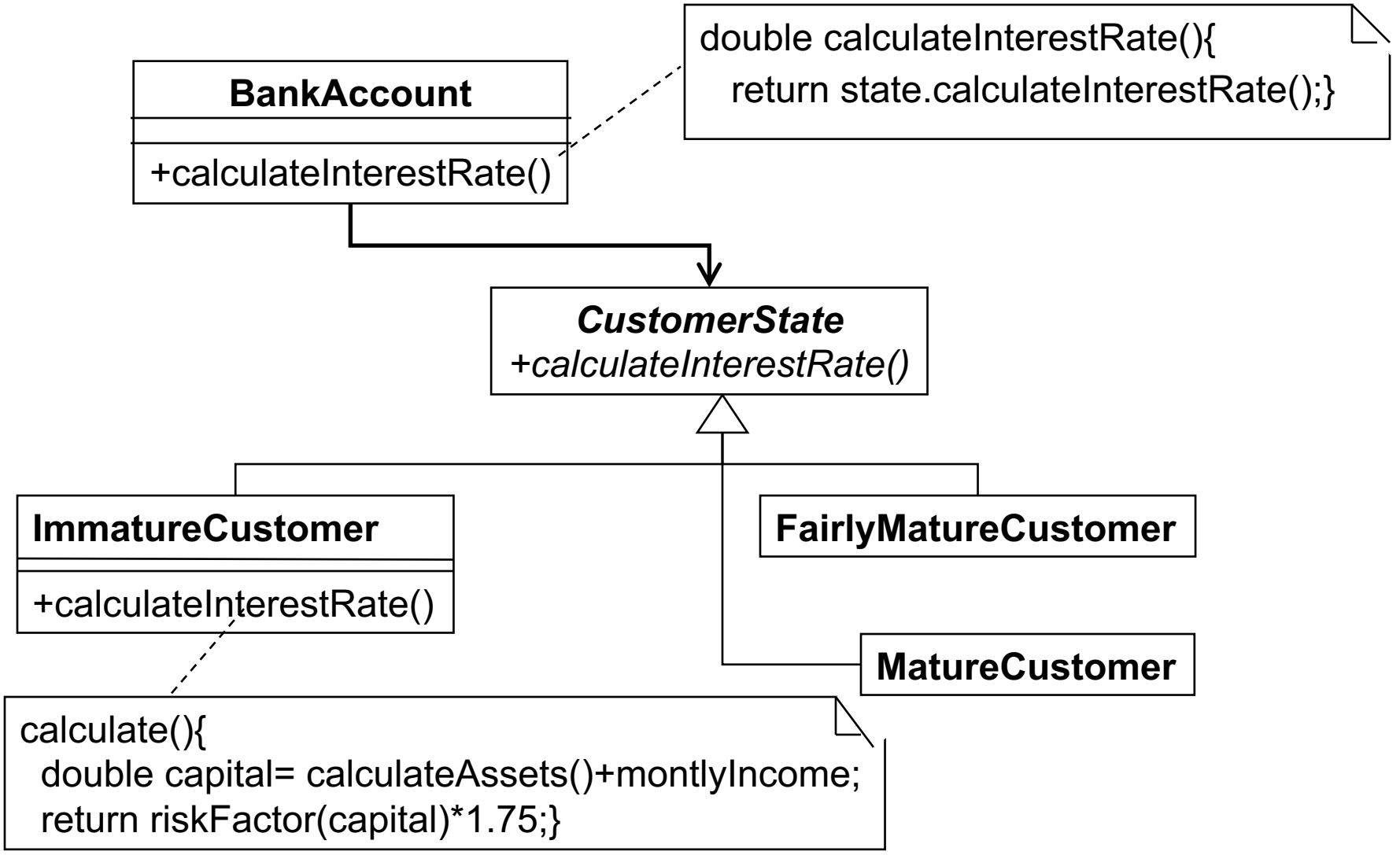
- Clean it first, then add or fix
- applies to all spaghetti code

```
double calculateInterestRate(){
  if(maturity==0){  //new customer
     double capital= calculateAssets()+montlyIncome;
     return riskFactor(capital)*1.75;
  }else if(maturity<5){  //fairly new customer
     double capital= calculateAssets()+montlyIncome;
      return riskFactor(capital)*0.85;
  }else if(maturity>10){ //mature customer
       if(stable) return currentInterestRate*0.8
     else return currentInterestRate;
  }}
```

Example Refactored-1



Example Refactored -2



Refactoring Spaghetti code

- Combine: capture aggregations and components
  - Move members to where they belong
- Generalize: Abstract Superclass
  - Make subclass method signatures compatible
  - Add method signatures to the superclass
  - Migrate common code to superclass
- **Specialize: simplify conditionals**
  - For each condition, make a subclass with matching invariant
  - State and Strategy patterns may help
  - Command pattern?

Spaghetti code: Symptoms

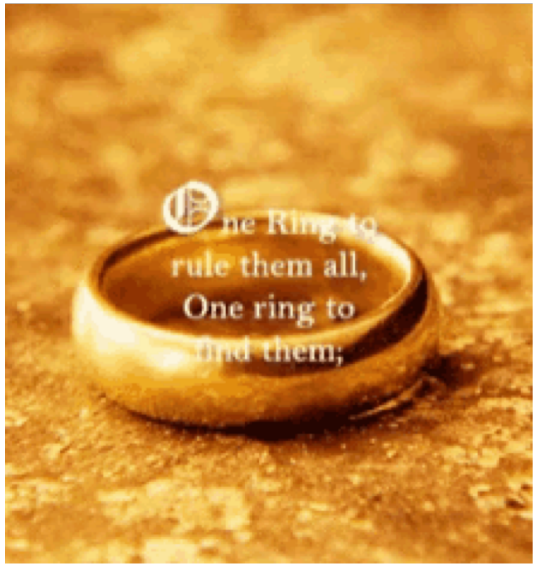
- Many object methods with no attribute
- Tight coupling between many classes
- Suspicious class or global variables
- Methods use global variables to perform computations
- Unforeseen relationships between objects
- Inheritance cannot be used to extend the system
- Polymorphism is not effective

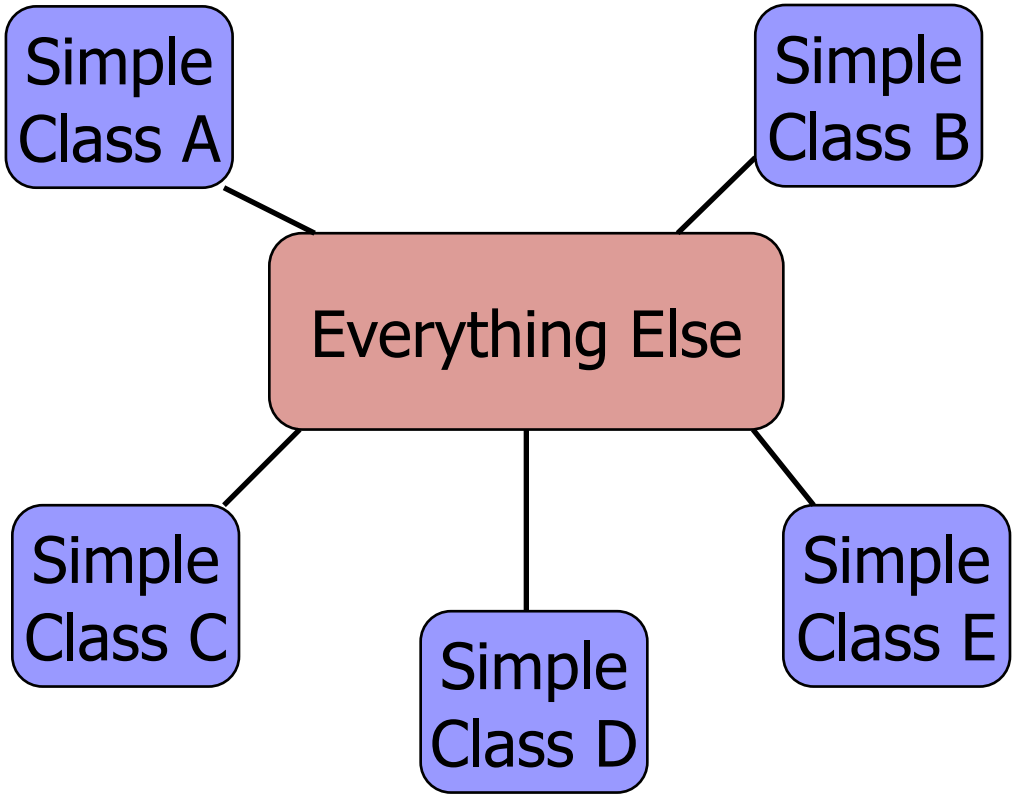
Spaghetti code/Design

- Causes:
  - Quick demo code that became operational
  - Lack of modular design
  - continuous, unplanned changes and poor coding practices, quick fixes.
  - Poor separation of concern
  - no SOLID
- Consequences
  - Most of maintenance time spent on **rediscovery**
  - Undocumented source code that cannot be extended or modified without extreme difficulty due to its structure
  - Difficult to understand and debug.
  - Hard to reuse and test
  - Increases risk of bugs and errors.

Blob (God Class)

- one class monopolizes the processing, other classes primarily encapsulate data





procedural design disguised in objects and OOP

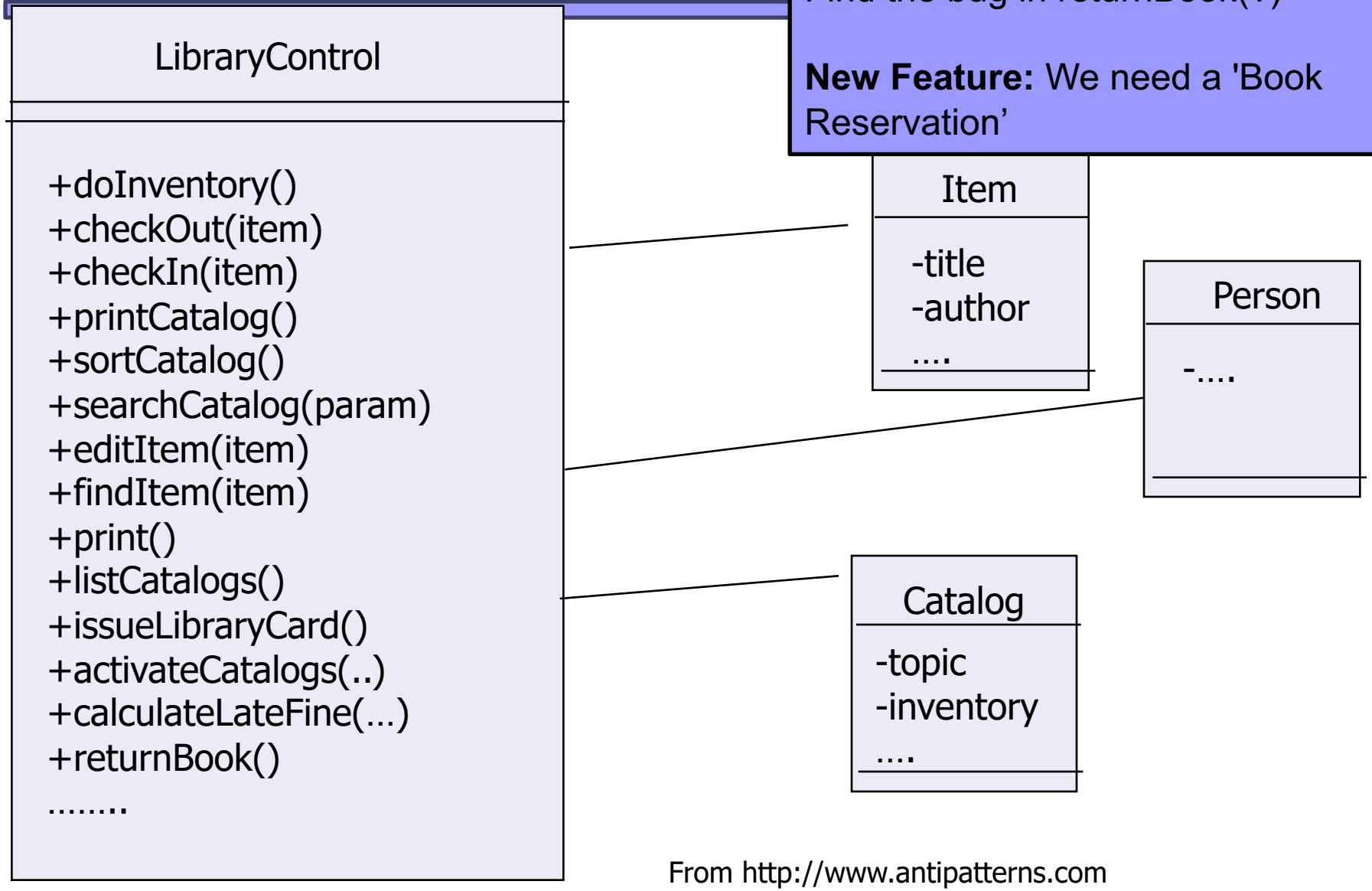
Characteristics of Blob

- **Overly Complex Class:** Handles multiple unrelated tasks.
- **High Coupling:** Interacts with many other classes.
- **Difficult to Maintain:** Hard to understand and modify.
- **Single Point of Failure:** Changes can have widespread impact.
- cohesion? coupling? SRP?

Blob Example

Bug Fix: A 'Premium' user is being charged the 'Standard' user fine. Find the bug in returnBook(?)

New Feature: We need a 'Book Reservation'



From http://www.antipatterns.com

```
public class LibraryControl { //15 atttributes
    public LibraryControl(){…}  //...a huge constructor that initializes everything...
    //….methods for user management: book registration, inventory, etc. ...
    public void issueBook(int userID, int bookID) {   // ... 50 lines of code ...
        // ... logic to check user status, book availability ...
        // ... logic to update database and send email notification ...
    }
    public void returnBook(int userID, int bookID) {    // ... 60 lines of code ...
         if (isOverdue(bookID)) {      // ... logic to check if late ...
            double fine = 0;
            if (getUser(userID).isPremiumMember())  fine = 2.50;
            else  fine = 2.50
            // ... more logic ...
            paymentProcessor.chargeFine(userID, fine);
            notifier.sendFineEmail(userID, fine);
        } //…logic to update the database
    }
    public void generateOverdueReport(String reportType) {       // ... 80 lines of code ...
        // ... complex database query ... // ... logic to format and email report to managers
}}
```

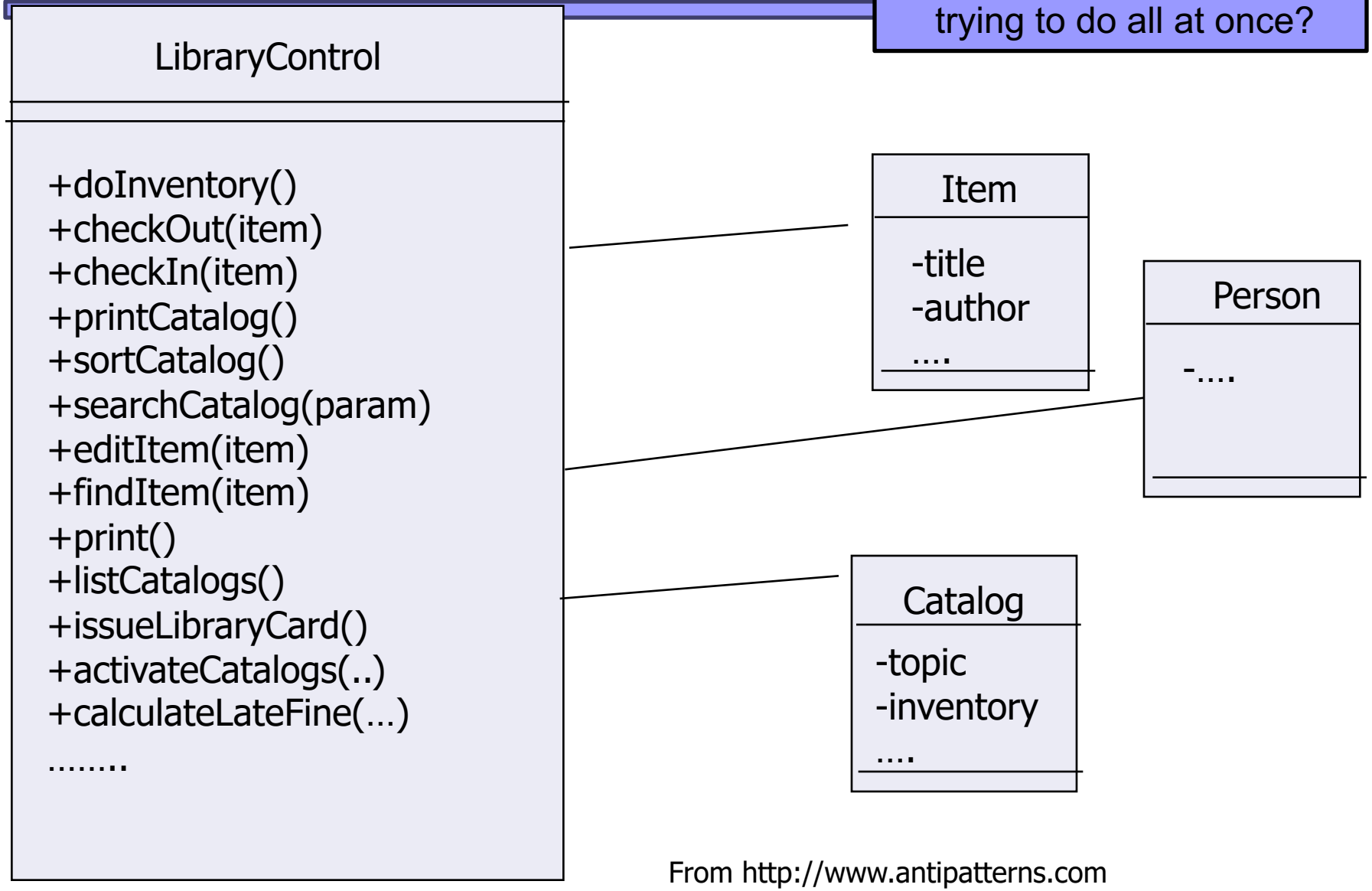
Bug Fix: A 'Premium' user is being charged the 'Standard' user fine. Find the bug in returnBook(?)

New Feature: We need a 'Book Reservation'

Blob Example

Fix Blob before adding new code.

- What are the *different jobs* LibraryControl is trying to do all at once?

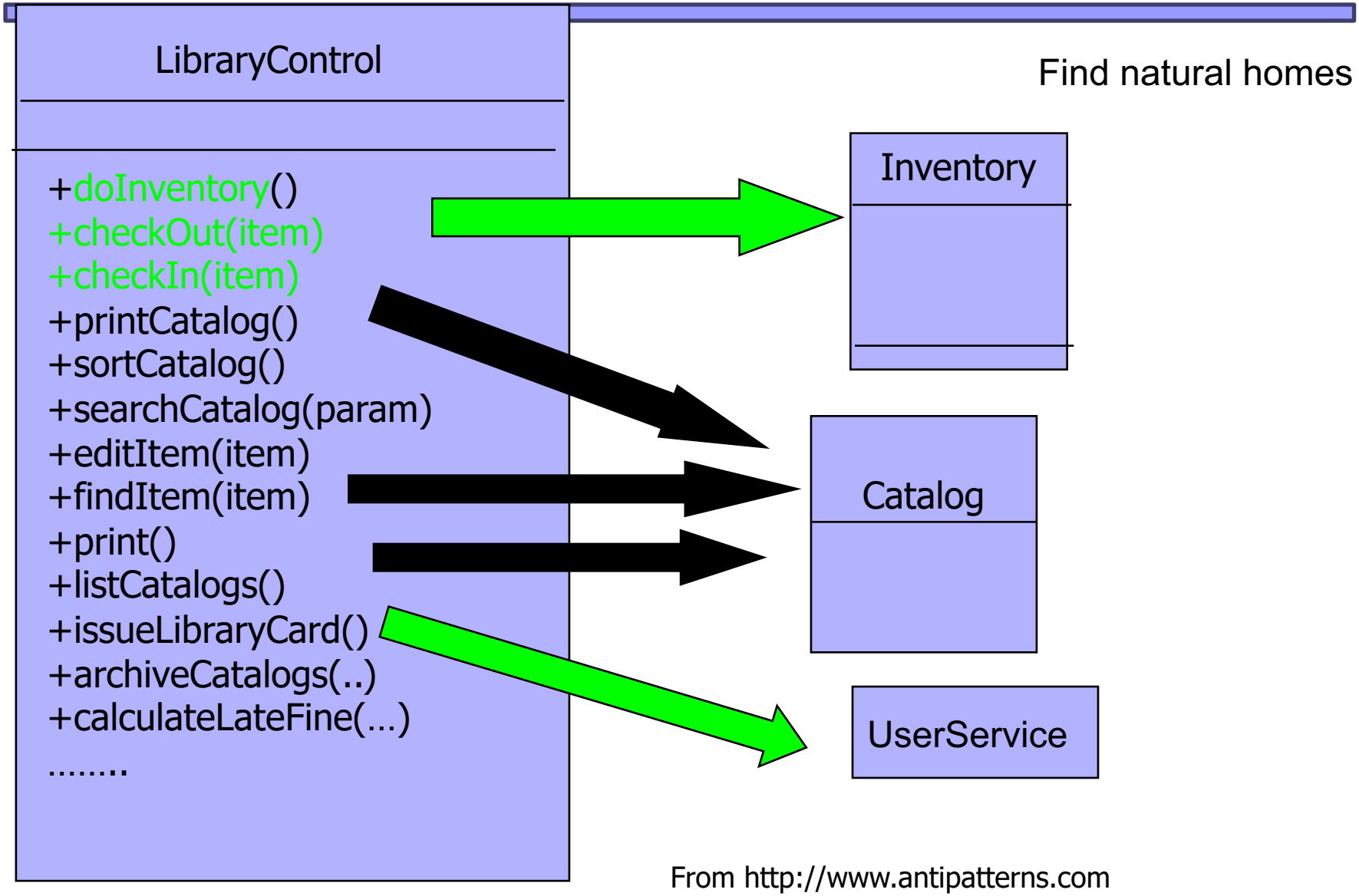


From http://www.antipatterns.com

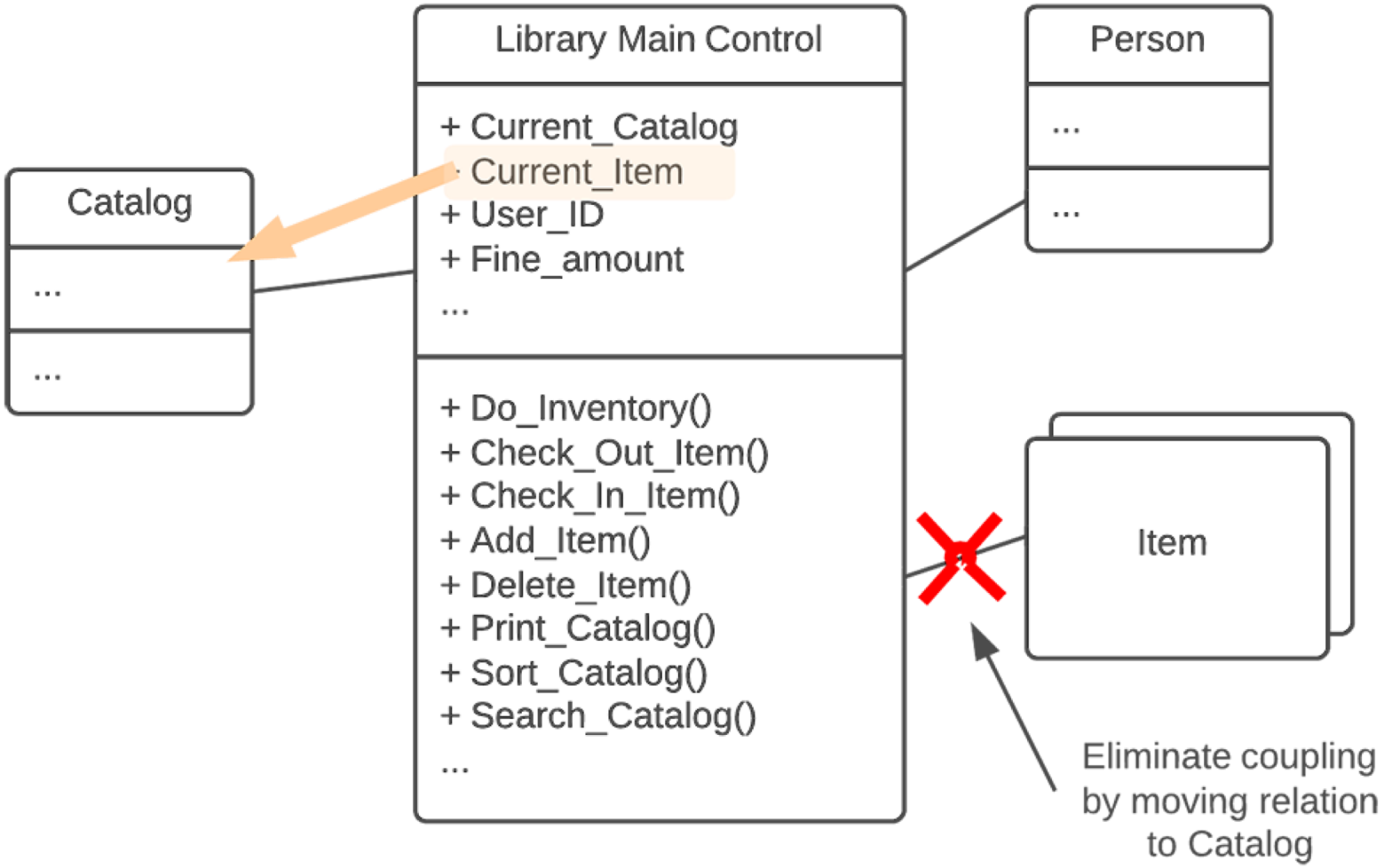
Refactoring Blob

- Break down into smaller focused classes
  - Each class should handle one responsibility
- Distribute the responsibility and isolate the effect of changes
  - Categorize related attributes and operations
    - Consider cohesion
  - Find natural hosts for these collections and migrate these code there
  - Extract methods, subclasses, abstract classes
  - Correct the associations/dependencies

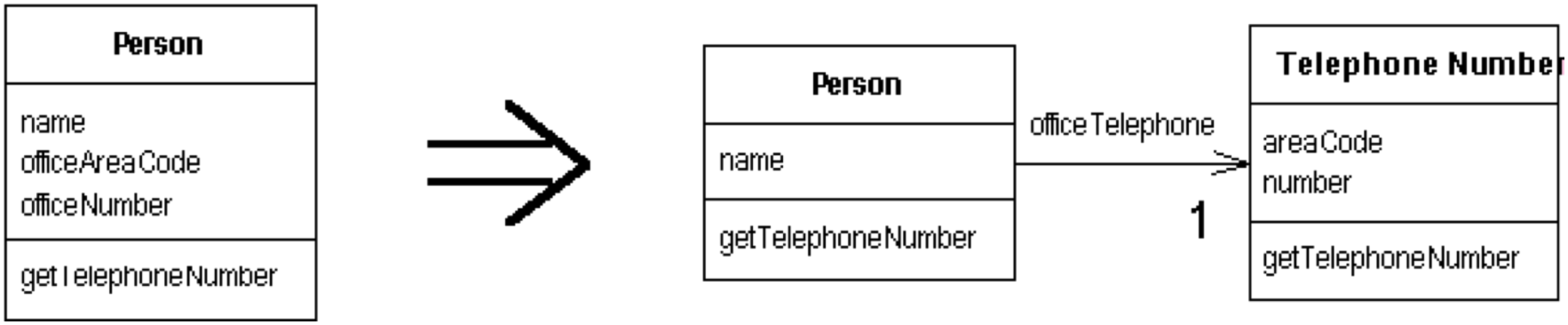
Breaking down the Blob



From http://www.antipatterns.com


Library Main Control
+ Current_Catalog
+ Current_Item
+ User_ID
+ Fine_amount
...
+ Do_Inventory()
+ Check_Out_Item()
+ Check_In_Item()
+ Add_Item()
+ Delete_Item()
+ Print_Catalog()
+ Sort_Catalog()
+ Search_Catalog()
...
Catalog
...
...
Person
...
...
Item
Eliminate coupling
by moving relation
to Catalog

Breaking down into classes



If after moving to natural homes the blob is still huge, break it down

Mediator vs Blob

- Be careful with Mediator pattern.
  - Mediator can become a Blob
  - If the implementation gone wrong, consider Observer
    - Pro: Colleagues notify other colleagues; often easier to make reusable Subjects/Observers.
    - Con: In N-to-N dependencies, Observer creates a web of cascading, untraceable updates.
  - Safety-Critical Systems (like aircraft/control tower), the decentralized, non-deterministic nature of the Observer pattern is dangerous

Blob…

- Complex, error-prone, and unmanageable code.
- Refactor into smaller, focused classes improves maintainability, testability, and readability.
- Adhering to the Single Responsibility Principle is key to avoiding the Blob anti-pattern.

Related Antipatterns

- The Blob is often accompanied by unnecessary code
  - making it hard to differentiate between the useful functionality of the Blob Class and no-longer-used code
  - Lava Flow
  - spaghetti code
  - poltergeist

SW Development Antipatterns

- Blob
- Lava flow
- Functional decomposition
- Poltergeist
- Golden hammer
- Spaghetti code
- Cut and paste programming

Mini antipatterns

- Boat anchor
- Input kludge
- Magic Numbers
- Continuous obsolescence
- Premature optimization

Development Antipatterns(1)

- Blob (God Class)
  - One big class with too many responsibilities
- Proliferation of classes (Poltergeist)
  - Small classes with short life cycles
- Spaghetti code
  - Ad-hoc code structure that is hard to extend and modify
- Lava flow
  - Dead code or forgotten design information that are frozen and no one understands
- Cut and paste programming
  - Duplication of code, leads to many errors like duplicates errors

Development Antipatterns (2)

- Golden hammer
  - Belief that a particular technology solves everything
    - Misapplication of a favored tool or concept
    - Gained high level of confidence on a solution, then every problem is viewed as something that is best solved with it
- Mini antipatterns
  - Input kludge
    - unexpected combinations of user-accessible features crashes your system –race conditions on concurrent input; validation;
  - Boat anchor
    - Keeping something that is no longer used in the current system
  - Magic Number
    - unexplained numerical values in code instead of named constants