Antipatterns

Mohammad Sanaye
Mehran Salmani
Amirhossein Asadi

Antipattern

A commonly occurring solution to a problem that generates decidedly negative consequences

Antipatterns clarify the negative patterns that cause development roadblocks

What an antipattern describes?

- ❏ The general form
- ❏ The primary causes
- ❏ Symptoms
- ❏ The consequences
- ❏ Refactored solution

Root causes

- Haste
- Apathy
- Narrow-mindness
- Sloth
- Avarice
- Ignorance
- Pride

Formal Refactoring Transformations

- Superclass abstraction
- Conditional elimination
- Aggregate abstraction

```
class Bird {
  // ...
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return getBaseSpeed();
      case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * numOfCoconuts;
      case NORWEGIAN_BLUE:
        return (isNailed) ? 0 : getBaseSpeed(voltage);
    }
    throw new RuntimeException("Should be unreachable");
  }
}
```

```
abstract class Bird {
  // ...
  abstract double getSpeed();
}

class European extends Bird {
  double getSpeed() {
    return getBaseSpeed();
  }
}
class African extends Bird {
  double getSpeed() {
    return getBaseSpeed() - getLoadFactor() *numberOfCoconuts;
  }
}
class NorwegianBlue extends Bird {
  double getSpeed() {
    return (isNailed) ? 0 : getBaseSpeed(voltage);
  }
 }
```

conditional elimination example from https://refactoring.guru/

Development Antipatterns

- God Class (blob)
- Functional Decomposition
- Poltergeists (gypsy)
- Cut and paste programming
- Call Super
- Lava Flow
- Spaghetti Code
- Golden Hammer

God Class (aka Blob)

The God class is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data

★ Root Causes: Sloth, Haste

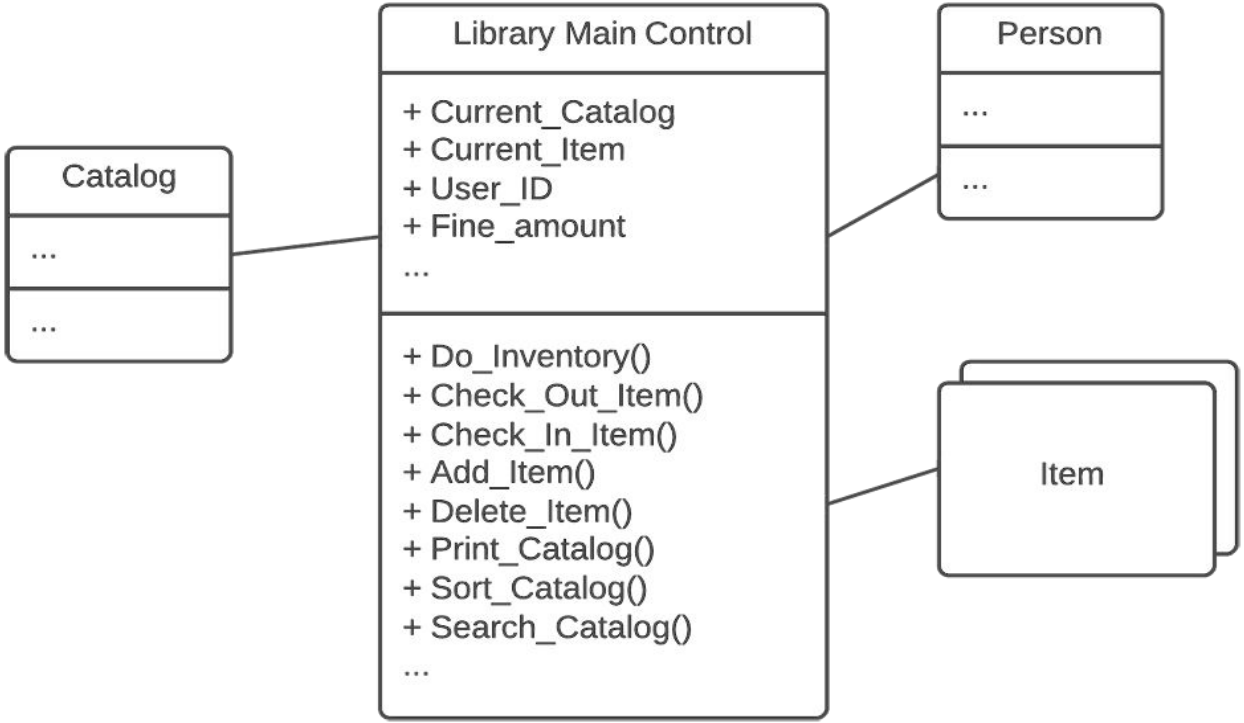
Symptoms:

- A single controller class with associated simple data-object classes
- Single class with a large number of attributes, operation, or both
- Lack of cohesiveness of the attributes and operations

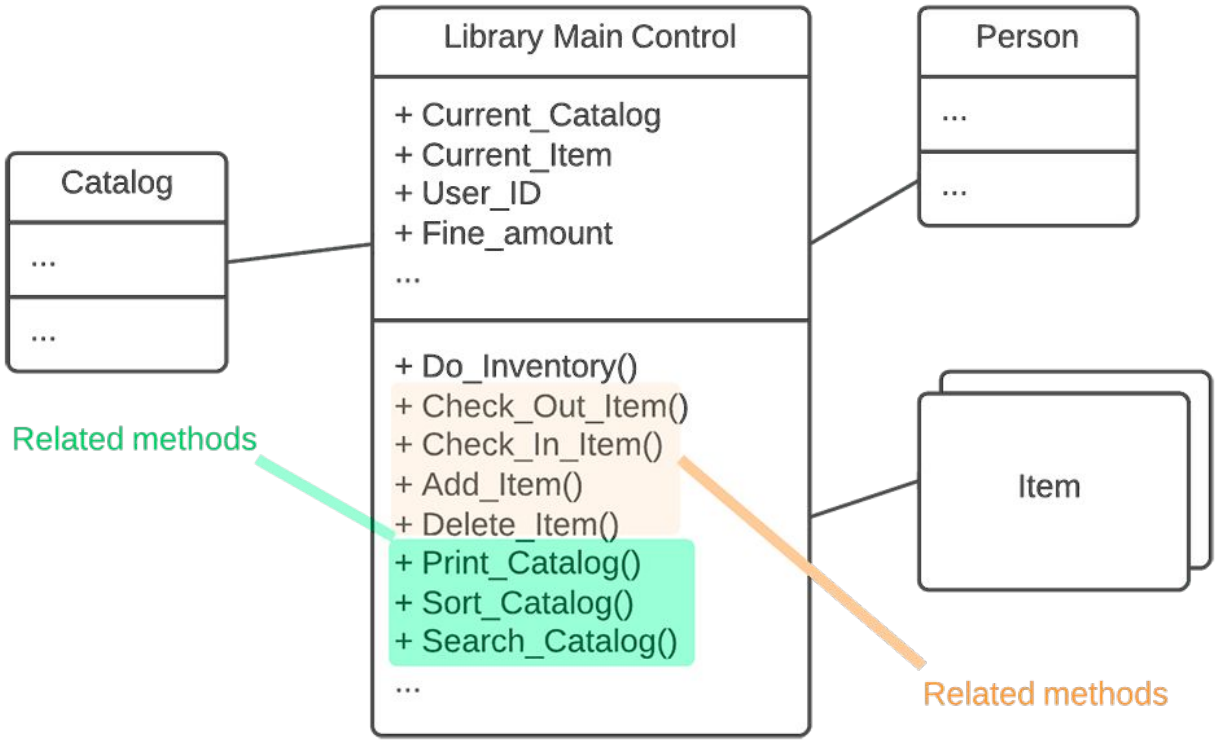
Consequences:

- Too complex for reuse and testing.
- The Blob limits the ability to modify the system without affecting the functionality of other encapsulated objects.

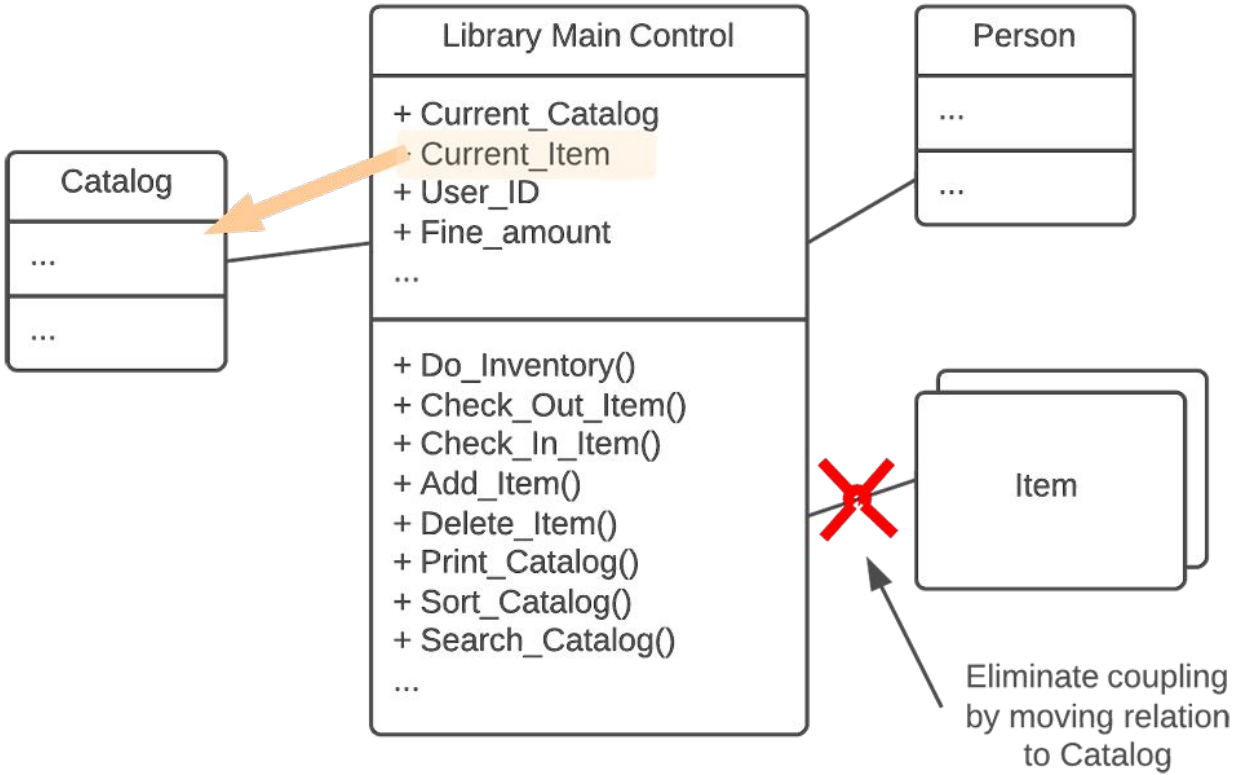
Solution:


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

Solution(contd):



Solution(contd):



functional decomposition

This AntiPattern is the result of experienced, nonobject-oriented developers who design and implement an application in an object-oriented language.

★ Root Causes: Sloth

Symptoms:

- Classes with function names such as Calculate_Interest or Display_Table
- Classes with a single action
- no leveraging of object-oriented principles

Consequences:

- hard to maintain
- No hope of ever obtaining software reuse
- Frustration on the part of testers.

Solution:

- If the class has a single method, try to better model it as part of an existing class.
- Attempt to combine several classes into a new class that satisfies a design objective.
- If the class does not contain state information of any kind, consider rewriting it as a function.

Poltergeists(aka gypsy)

Poltergeists are classes with limited roles to play in the system; therefore, their effective life cycle is quite brief.

★ Root Causes: Ignorance

Symptoms:

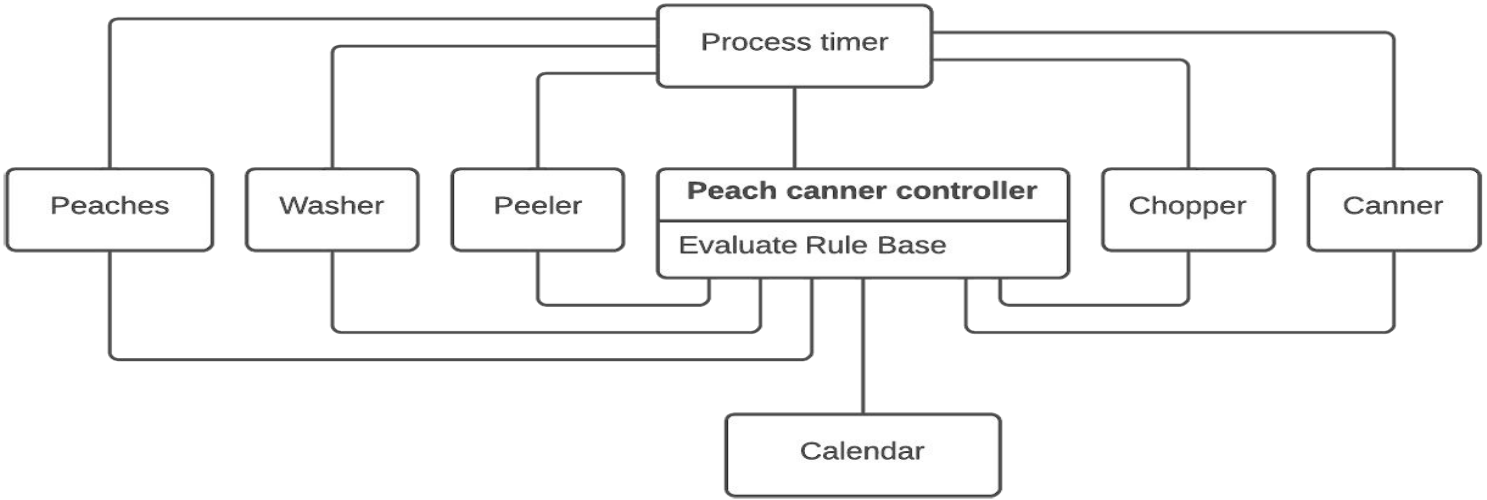
- Transient associations
- Stateless classes.
- Classes with control-like operation names such as start_process_alpha

Consequences:

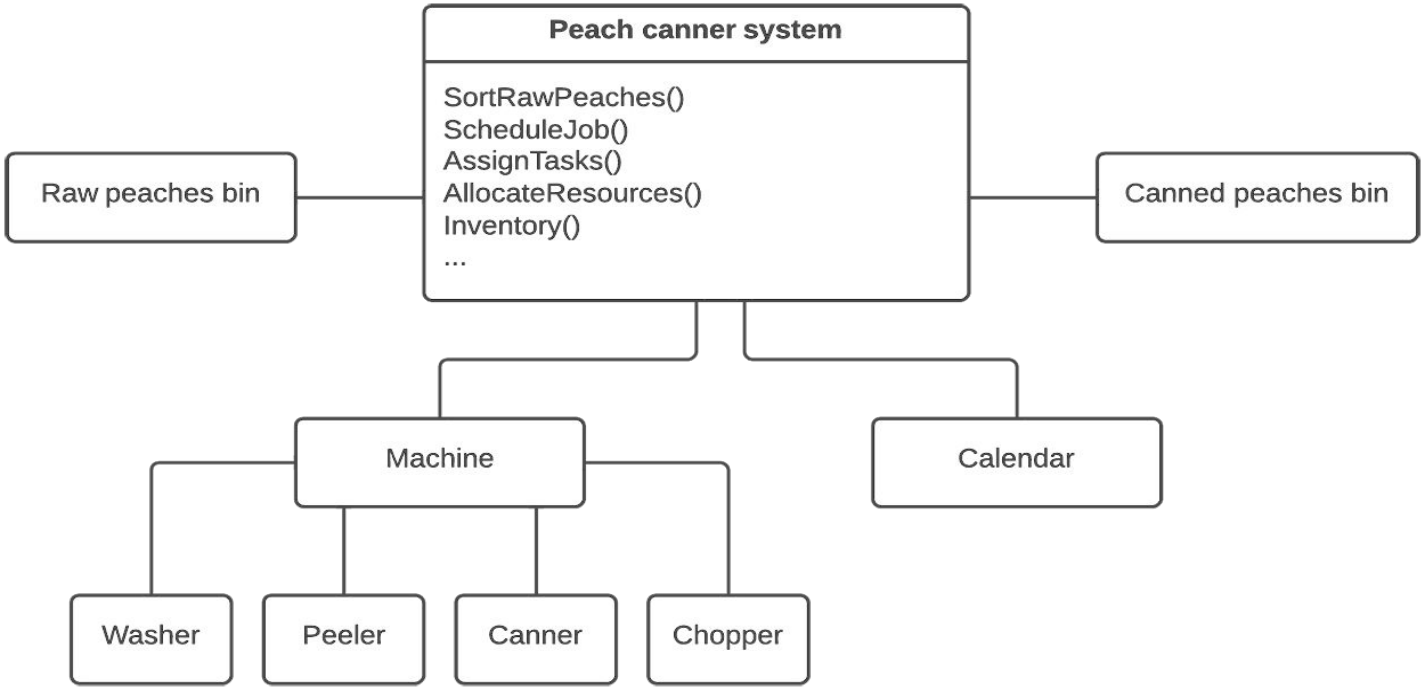
- they waste resources
- needlessly cluttering the object model.

Solution:

The key is to move the controlling actions initially encapsulated in the Poltergeist into the related classes that they invoked.



Solution:


Peach canner system
SortRawPeaches()
ScheduleJob()
AssignTasks()
AllocateResources()
Inventory()
...
Raw peaches bin
Canned peaches bin
Machine
Calendar
Washer
Peeler
Canner
Chopper

Cut and paste programming

This AntiPattern is identified by the presence of several similar segments of code interspersed throughout the software project.

★ Root Causes: Sloth

Symptoms and consequences:

- The same software bug reoccurs throughout software despite many local fixes.
- Software defects are replicated through the system.
- Lines of code increase without adding to overall productivity.
- This AntiPattern leads to excessive software maintenance costs.
- It becomes difficult to locate and fix all instances of a particular mistake.

Typical causes:

- The organization does not advocate or reward reusable components, and development speed overshadows all other evaluation factors.
- Reusable components, once created, are not sufficiently documented or made readily available to developers.
- There is a lack of forethought or forward thinking among the development teams.
- There is a lack of abstraction among developers, often accompanied by a poor understanding of OO principles.

Solution:

refactor the code base into reusable libraries or components that focus on black-box reuse of functionality.

stages:

- ❏ code mining
- ❏ refactoring
- ❏ configuration management

Call super

A particular class stipulates that in a derived subclass, the user is required to override a method and call back the overridden function itself at a particular point.

```
public class EventHandler ...
  public void handle (BankingEvent e) {
    housekeeping(e);
}
public class TransferEventHandler extends
EventHandler...
  public void handle(BankingEvent e) {
    super.handle(e);
    initiateTransfer(e);
}
```

Solution:

```
public class EventHandler … {
public void handle (BankingEvent e) {
    housekeeping(e);
    doHandle(e);
}
   protected void doHandle(BankingEvent e) {}
}

public class TransferEventHandler extends EventHandler … {
   protected void doHandle(BankingEvent e) {
                initiateTransfer(e);
   }
}
```

Lava Flow (aka Dead Code)

immovable, generally useless mass of code that no one can remember much, if anything,about.

★ Root Causes: Avarice, Sloth

```
// This class was written by someone earlier (Alex?) to manager 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);
  }
  // -------------------------
```

Symptoms and consequences:

- Undocumented complex, important-looking functions, classes, or segments that don't clearly relate to the system architecture.
- Lots of "in flux" or "to be replaced" code areas.
- As the flows compound and harden, it rapidly becomes impossible to document the code or understand its architecture enough to make improvements.

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.
- Repetitive change of project

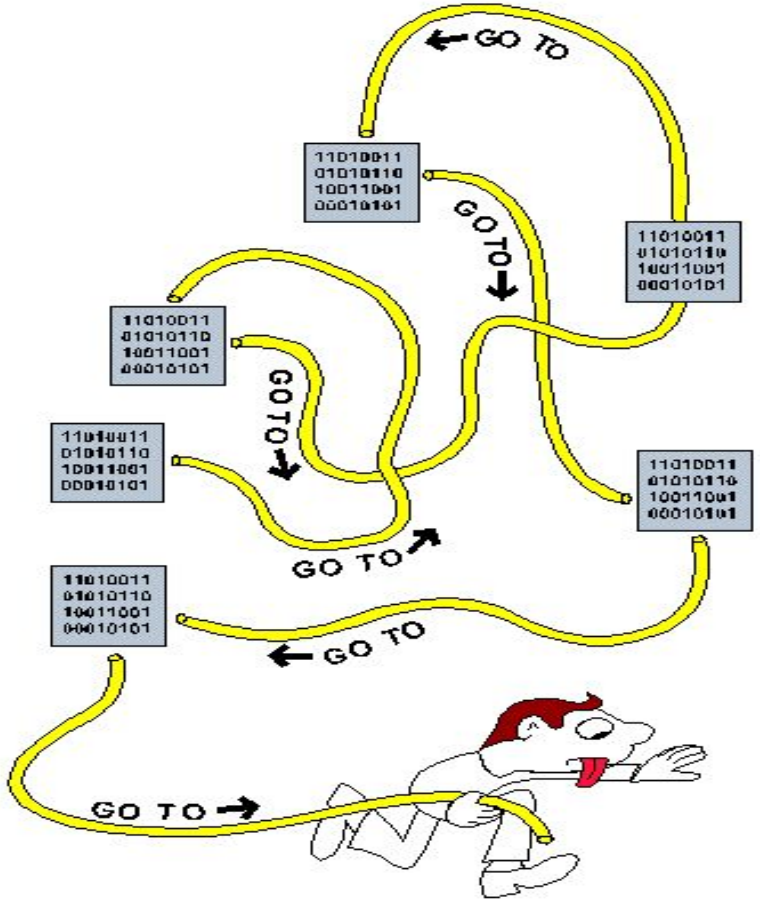
Solution:

- ensure that a sound architecture precedes production code
- this architecture is backed up by a configuration management process that can handle requirement changes
- establish system-level software interfaces that are stable, well-defined, and clearly documented

Spaghetti Code

Code which is so difficult to read or change that it becomes nearly impossible to maintain.

★ Root Causes: Ignorance, Sloth



Symptoms and consequences:

- Minimal relationships exist between objects.
- Many object methods have no parameters, and utilize class or global variables for processing.
- Benefits of object orientation are lost; inheritance is not used to extend the system; polymorphism is not used.

Typical causes:

- Inexperience with object-oriented design technologies.
- No mentoring in place; ineffective code reviews.
- No design prior to implementation.
- Frequently the result of developers working in isolation.

Golden Hammer

This AntiPattern results in the misapplication of a favored tool or concept

- ★ Root Causes: Ignorance, Pride, Narrow-Mindedness

Symptoms and consequences:

- Identical tools and products are used for wide array of conceptually diverse products.
- System architecture is best described by a particular product, application suite, or vendor tool set.
- Existing products dictate design and system architecture.
- Requirements are not fully met, in an attempt to leverage existing investment.

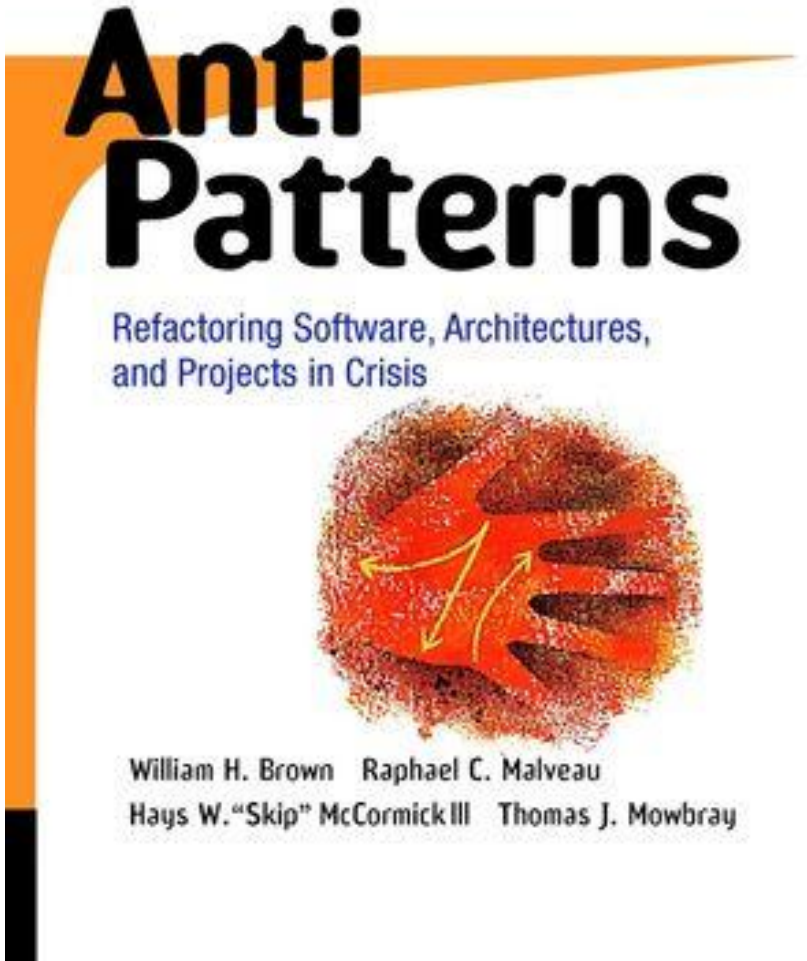
Typical causes:

- Several successes have used a particular approach.
- Large investment has been made in training and/or gaining experience in a product or technology.
- Reliance on proprietary product features that aren't readily available in other industry products.

Solution:

- organizations need to develop a commitment to an exploration of new technologies.
- software developers need to be up to date on technology trends
- encourage the hiring of people from different areas and from different backgrounds

reference:


Anti
Patterns
Refactoring Software, Architectures,
and Projects in Crisis
William H. Brown Raphael C. Malveau
Hays W. "Skip" McCormick III Thomas J. Mowbray

Thanks!