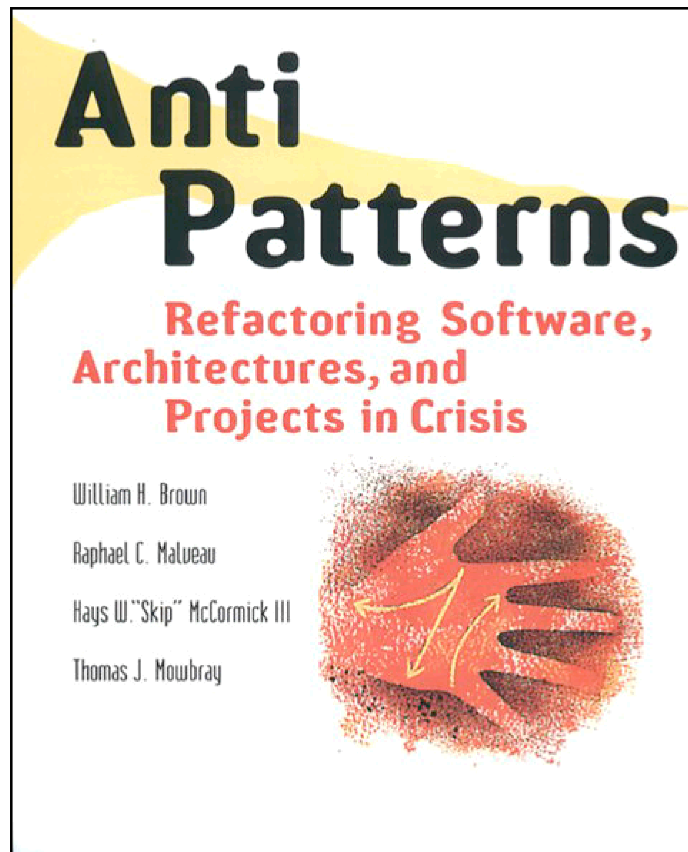
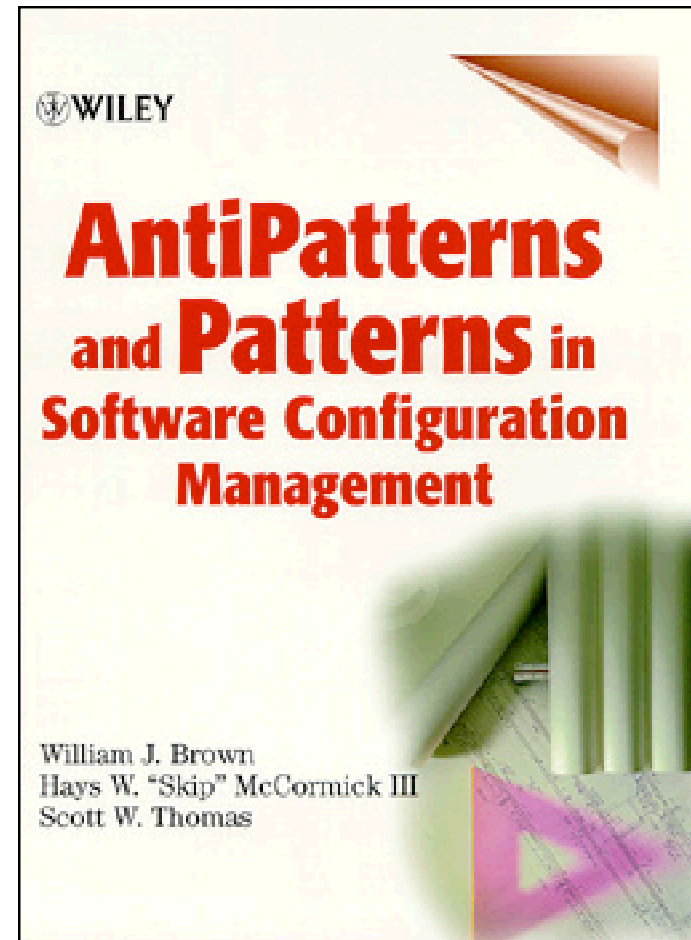
"AntiPatterns"

What are AntiPatterns?

- Recall Design Patterns.
- AntiPatterns are observable phenomena that are signs of development **problems**.
- The purpose of cataloguing antiPatterns is so they can be recognized and remedied ("refactored").

Representative Sources





AntiPatterns

- Begin with a **problematic attempt** to solve a problem.
- Abstract **symptoms** and consequences, similar to the context of a design pattern.
- Once identified, the antiPattern's ***refactored* solution** can be used to resolve or lessen the problem.

refactored?

- “refactored” is a code word meaning:

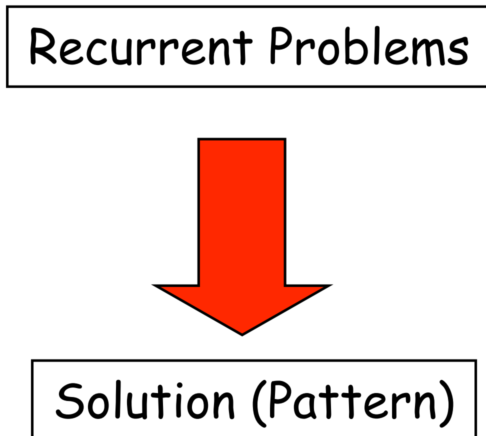
  “changed for the better”,

  “improved”,

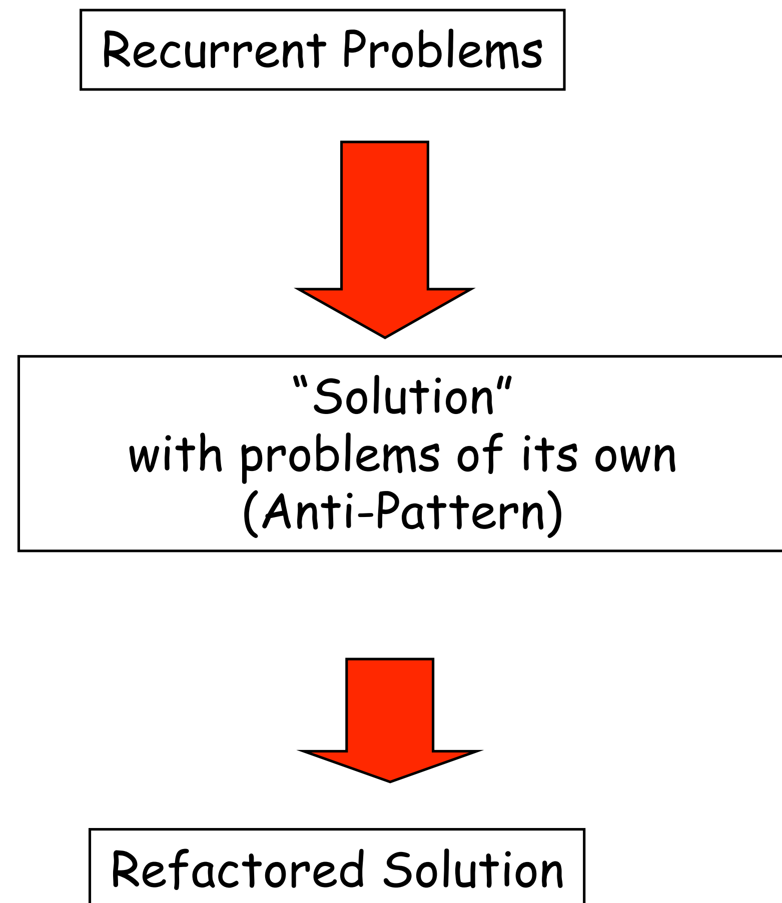
  “re-engineered”

Patterns vs. AntiPatterns

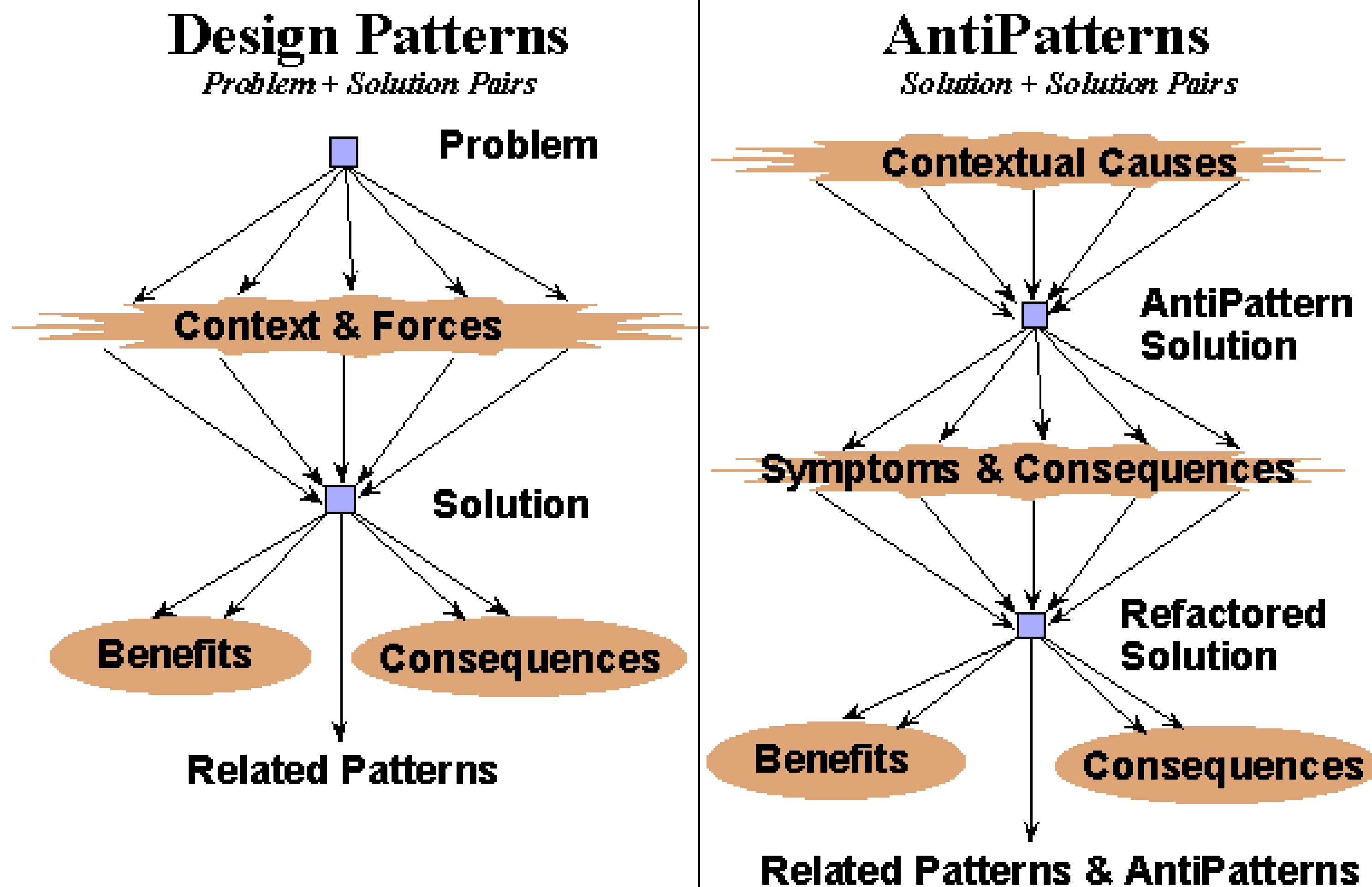
Patterns:



Anti-Patterns:



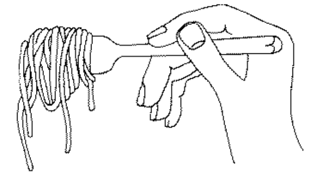
The preceding is my own refactoring of this "cloud diagram"



Software vs. Software Development

- When authors say **"software" anti-Patterns** they often mean "***software development***" antiPatterns.
- Software can have its own antiPatterns, also referred to as "design flaws":
  - fascistic features
  - gratuitous sound effects
  - unnecessary font transformations

AntiPattern: Spaghetti Code



- **Scale**: Application
- **Symptoms**:
  - Single body of code supporting > 1 function
  - Easier to rewrite code than modify
  - Lack of documentation
- **Cause**: sloth, ignorance, time pressure
- **Refactored Solution**: Code cleanup, code factoring, higher-order functions, object hierarchy

From: "How To Write Unmaintainable Code"
http://mindprod.com/jgloss/unmain.html

It almost goes without saying that the larger a function is, the better it is. And the more jumps and GOTOs the better. That way, any change must be analyzed through many scenarios. It snarls the maintenance programmer in the spaghettiness of it all.

And if the function is truly gargantuan, it becomes the Godzilla of the maintenance programmers, stomping them mercilessly to the ground before they have an idea of what's happened.

AntiPattern: The Blob (aka "God Class")

- **Scale**: Application
- **Symptoms**:
  - One **big** class, hundreds of unrelated methods
  - Many methods with no arguments
- **Cause**: lack of design experience
- **Refactored Solution**: Split into smaller classes, avoid transitive associations
- Similar to: Swiss army knife, kitchen sink

AntiPattern: Poltergeists

- **Scale**: Application
- **Symptoms**:
  - Lots of small, non-descript, classes
  - Classes have limited use
  - Classes have overlapping uses
- **Cause**: lack of design
- **Refactored Solution**: Create fewer and more coherent classes

AntiPattern: Cut-and-Paste

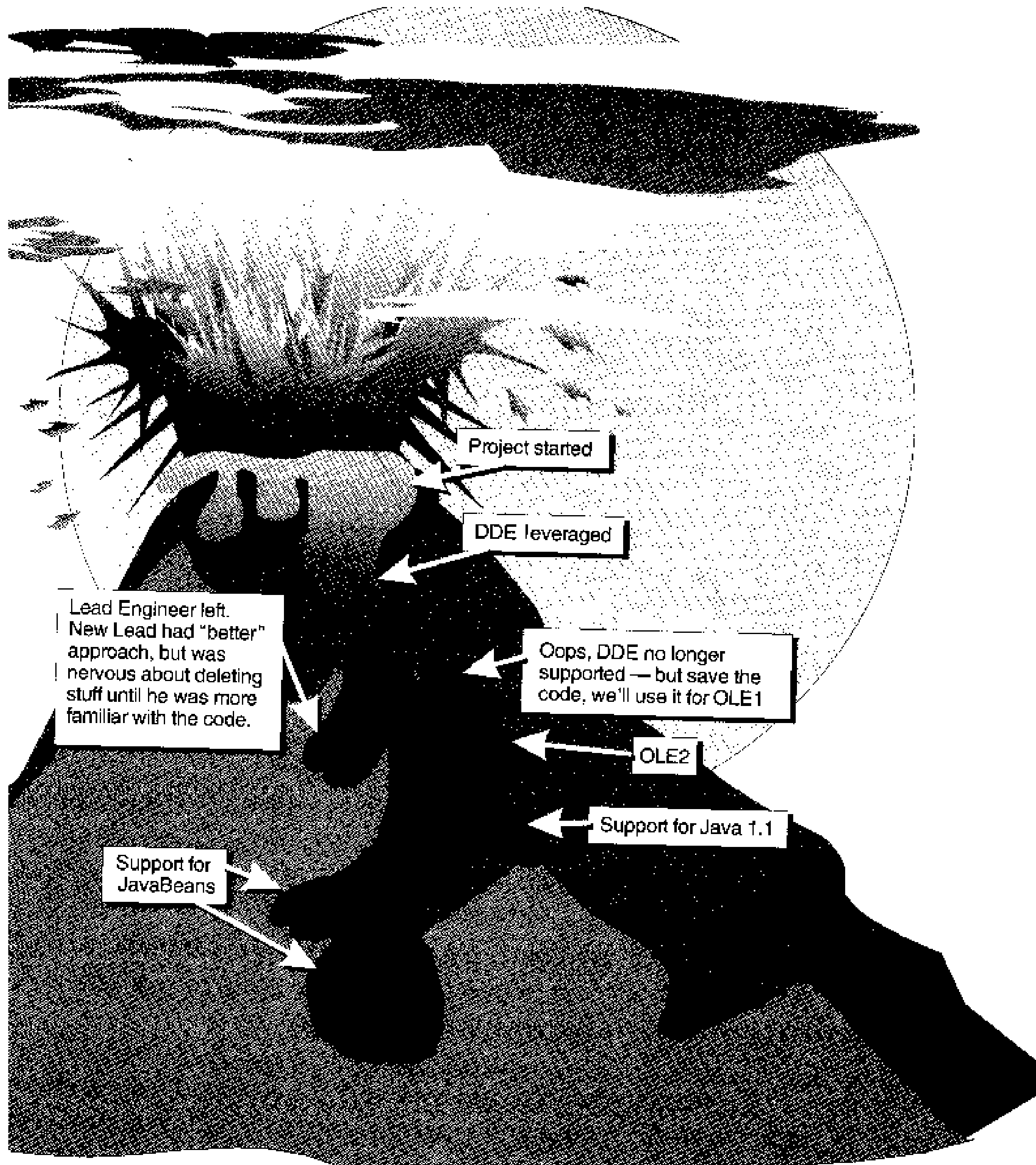
- **Scale**: Application
- **Symptoms**:
  - Over 10000 lines of code in a week
  - Having to make multiple identical edits to correct a single problem
- **Cause**: sloth, ignorance, time pressure
- **Refactored Solution**: Procedures, macros, methods, higher-order functions

AntiPattern: Input Kludge

- **Scale**: Application
- **Symptoms**:
  - Software fails on straightforward input tests
- **Cause**: sloth, ignorance, time pressure
- **Refactored Solution**: Construct a proper parser and error-check the input.

AntiPattern: Lava Flow
(dead code and forgotten design)

- **Scale**: Application
- **Symptoms**:
  - Code that nobody understands
  - Code author long departed
  - Undocumented design
  - Code that can't be tested
- **Cause**: failure to use revision control system
- **Refactored** Solution: Use revision control, get rid of dead code, redesign


Project started
DDE leveraged
Lead Engineer left. New Lead had "better" approach, but was nervous about deleting stuff until he was more familiar with the code.
Oops, DDE no longer supported — but save the code, we'll use it for OLE1
OLE2
Support for Java 1.1
Support for JavaBeans

AntiPattern: Stovepipe System
(irregular system parts hooked together)

- **Scale**: Application
- **Symptoms**:
  - Inordinately large system
  - Many components with similar functions
  - Many different interfaces
- **Cause**: too much reliance on components
- **Refactored Solution**:
  - Use abstraction to coalesce components.
  - Use layered architecture, use common interfaces.

AntiPattern: Vendor Lock-in

- **Scale**: Application
- **Symptoms**:
  - "Our architecture *is* DCOM" (or whatever) (which is code for "We don't have an architecture.")
  - Missing features, because vendor doesn't support them
  - Difficult to upgrade product
- **Cause**: excessive reliance on a vendor
- **Refactored Solution**: layer that isolates vendor-specific modules from other code

AntiPattern: Golden Hammer

- **Scale**: Application
- **Symptoms**:
  - Contrived code
  - Database driving the architecture
  - Unusual language choices (Hypertalk, Excel macros)
- **Cause**: using one tool for everything
- **Refactored Solution**: find tools best suited to the problem

AntiPattern: Reinventing the Wheel

- **Scale**: Application
- **Symptoms**:
  - "Not invented here"
  - "Our problem is unique"
- **Cause**: failure to use the work of others or buy available components
- **Refactored Solution**: Use existing patterns and components

AntiPattern Categories

- Software development
- Software architecture
- Project Management

Some Project *Management* AntiPatterns

- Design by Committee
- Analysis Paralysis
- Death by Planning
- Viewgraph Engineering
- Corncob
- Death March Projects
- Irrational Management
- Throw it over the wall
- Fire Drill

Some Software Team Anti-Patterns

- **Reinvent the wheel**: Ignore existing patterns.
- **Out-to-lunch**: Team member doesn't read email or voice-mail (Particularly bad if it's the leader.)
- **Things-to-do, places-to-go, ...**: Team member doesn't attend meetings.
- **Golden hammer**: Team member too enamored with using specific tool, forgets about the main problem.
- **Client starvation**: Team does not interact with client, ends up with unusable product.
- **I'd rather do it myself**: Team doesn't ask for advice on difficult issue until it's too late.
- **Blackhole**: Team gets committed to an unusable software library.