SE Methods

Software Development Life Cycle (SDLC)

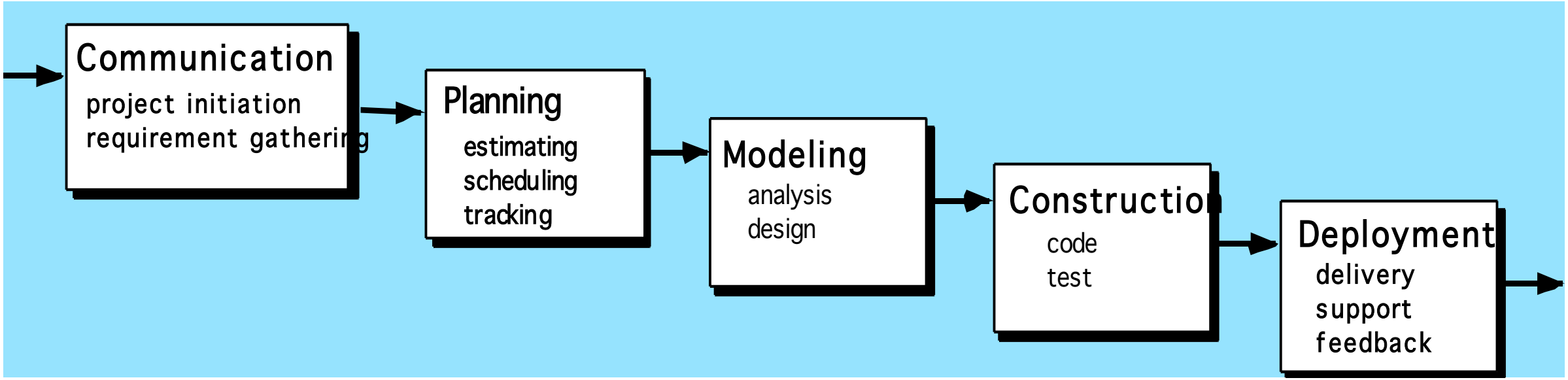
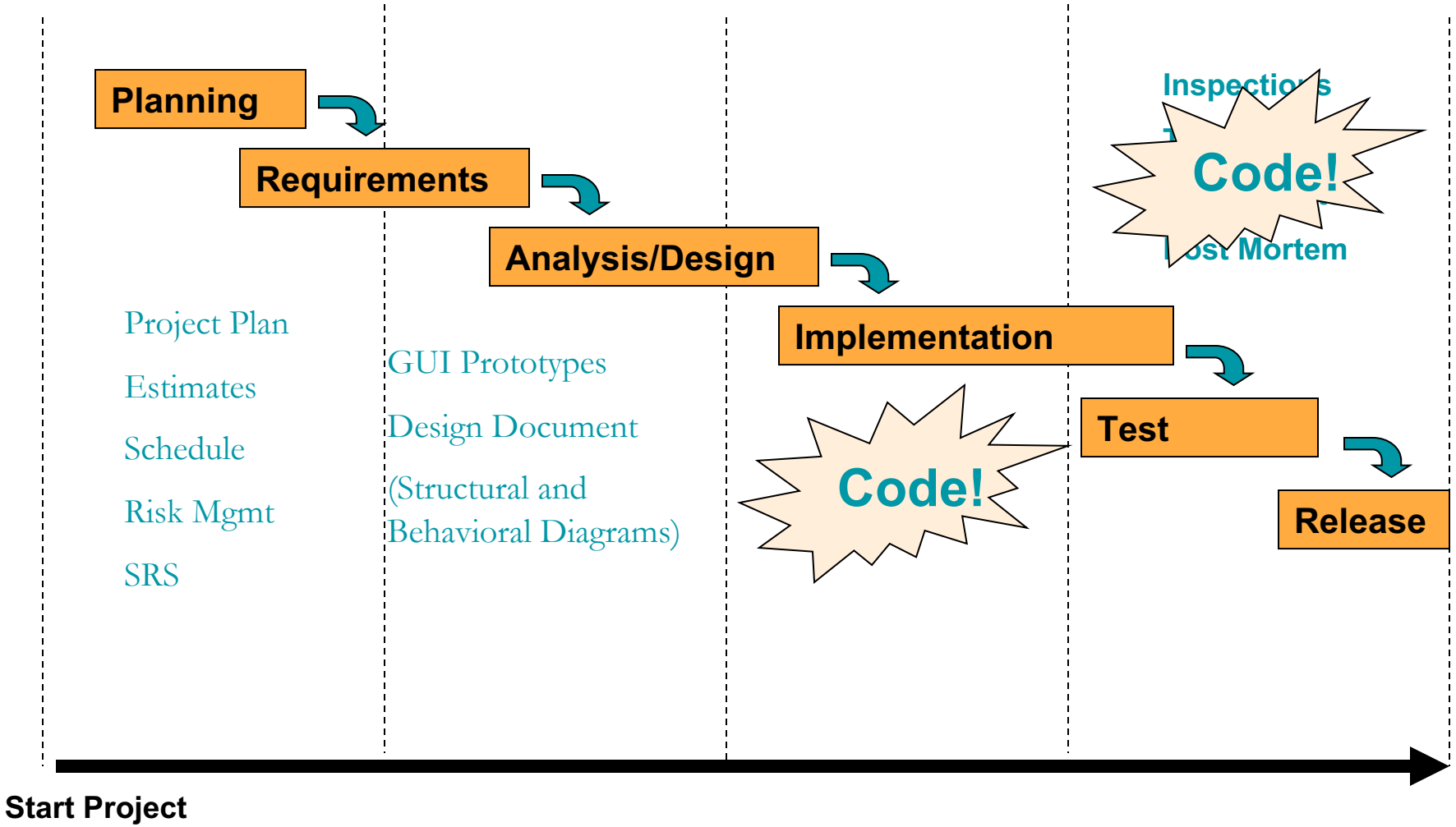
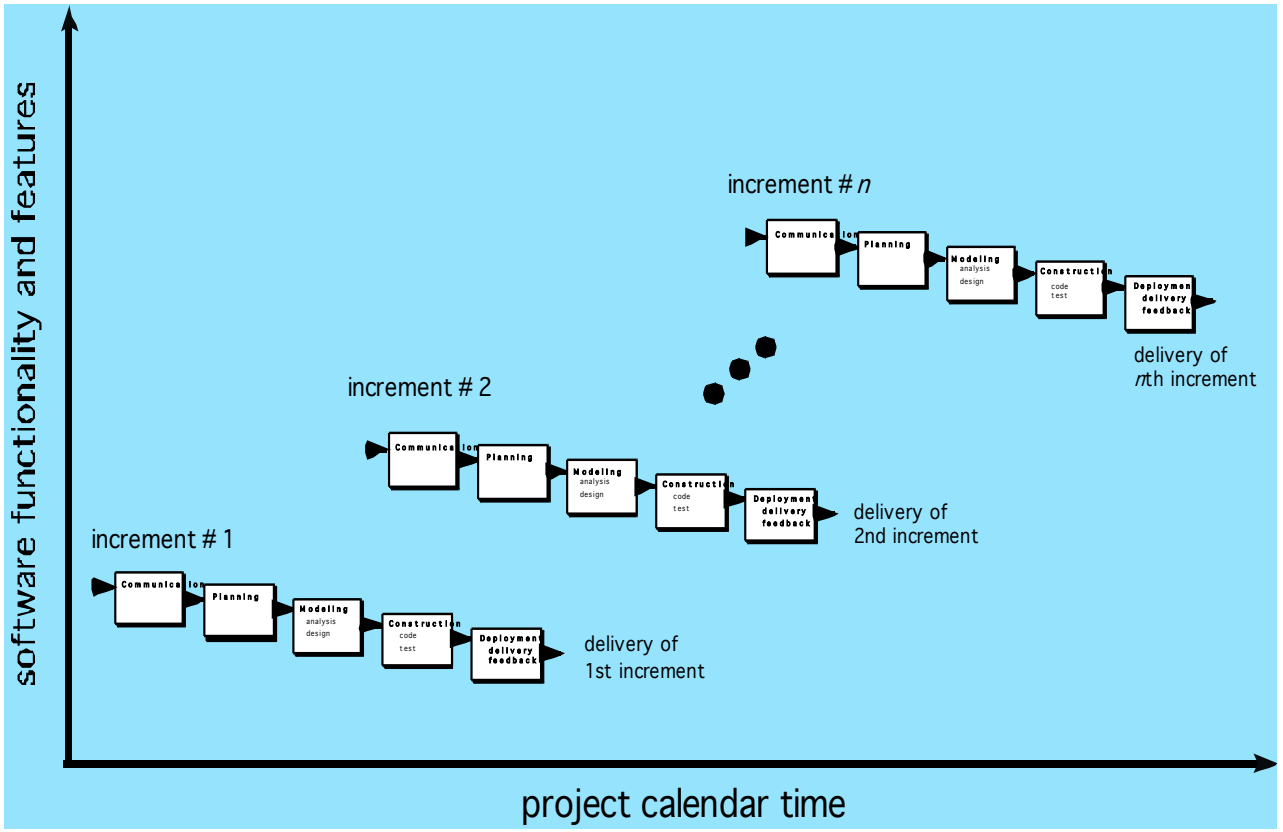


Image Courtesy: Sofware Engineering: A Practioner's approach by Roger Pressman

Traditional SDLC. (e.g. waterfall process)



Software Development Life Cycle – Process models



The Incremental Model

Image Courtesy: Sofware Engineering: A Practioner's approach by Roger Pressman

Challenges with traditional (plan-driven) approaches

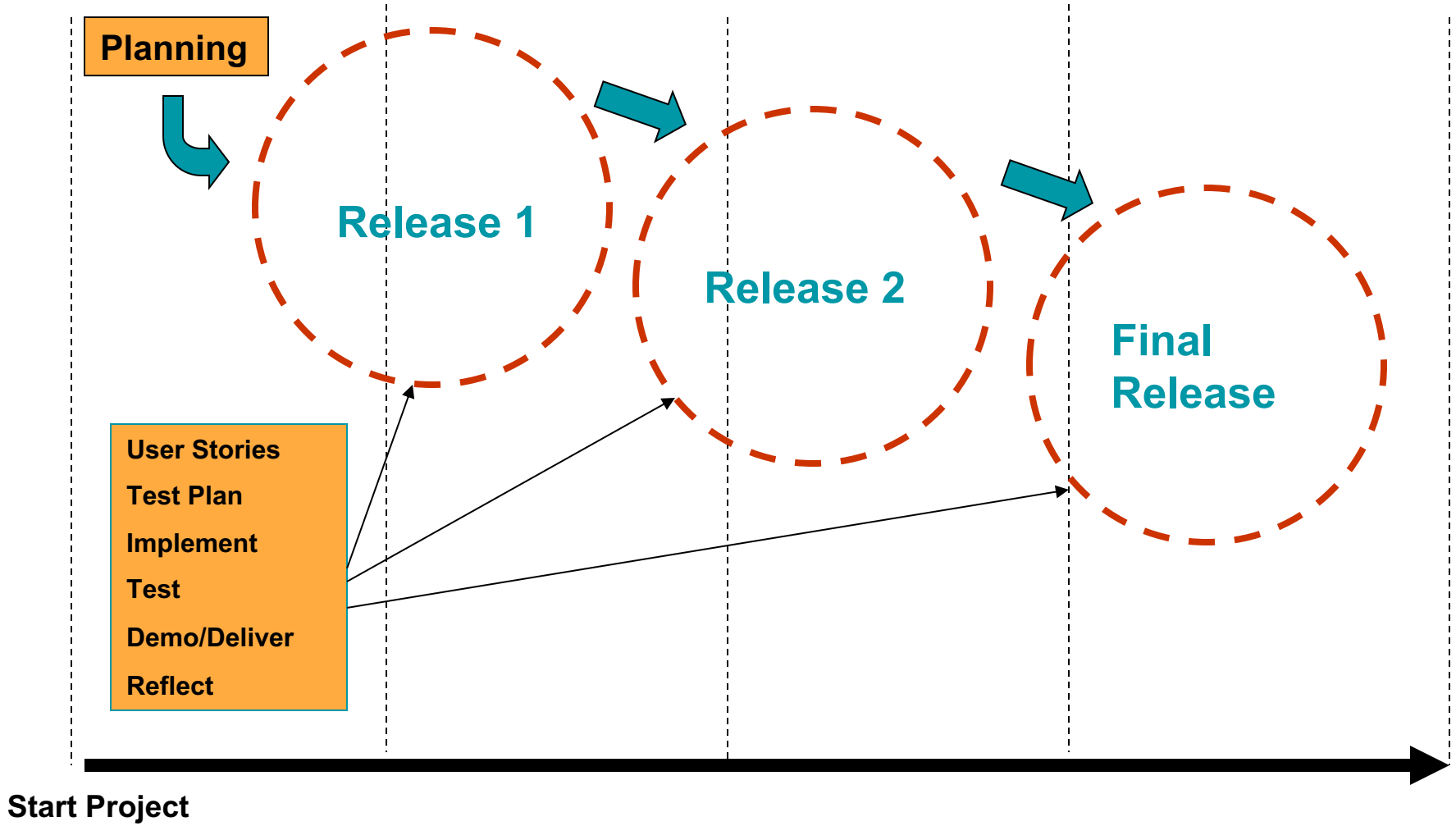
- Lightweight applications/heavyweight process
- Document intensive (perceived)
- Less flexible design
- Big bang approach to coding/integration
- Testing short-shifted
- One-shot delivery opportunity
- Limited opportunity for process improvement

What Is Agile Software Development?

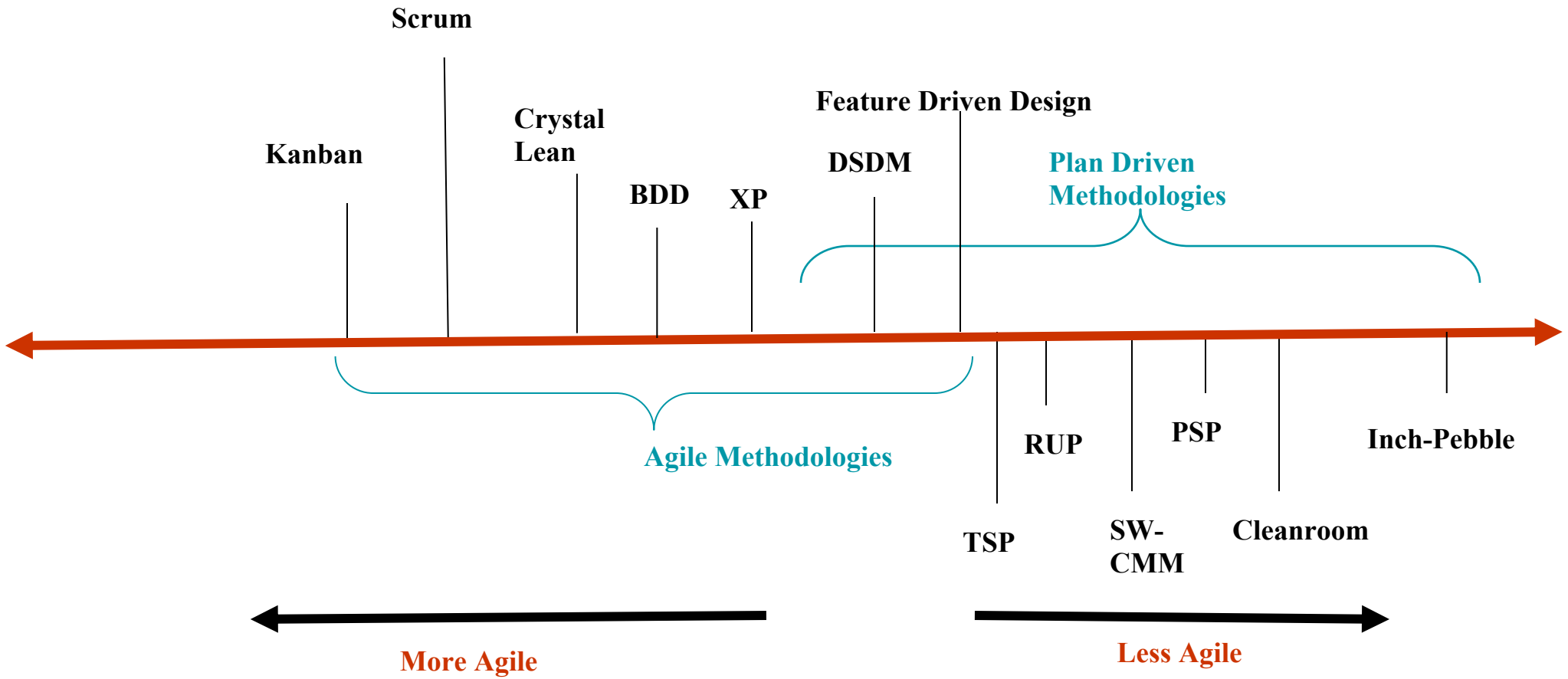
- In the late 1990's several methodologies began to get increasing public attention. All emphasized:
  - Close collaboration between developers and business experts
  - Face-to-face communication (as more efficient than written documentation)
  - Frequent delivery of new deployable business value
  - Tight, self-organizing teams
  - Ways to craft the code and the team such that the inevitable requirements churn was not a crisis.

- 2001 : Workshop in Snowbird, Utah, Practitioners of these methodologies met to figure out just what it was they had in common. They picked the word "agile" for an umbrella term and crafted the
  - Manifesto for Agile Software Development,

Applying Agility



The Process Methodology Spectrum



It's not that black and white. The process spectrum spans a range of grey !

@ (Barry Boehm & Turner)

Agile Characteristics

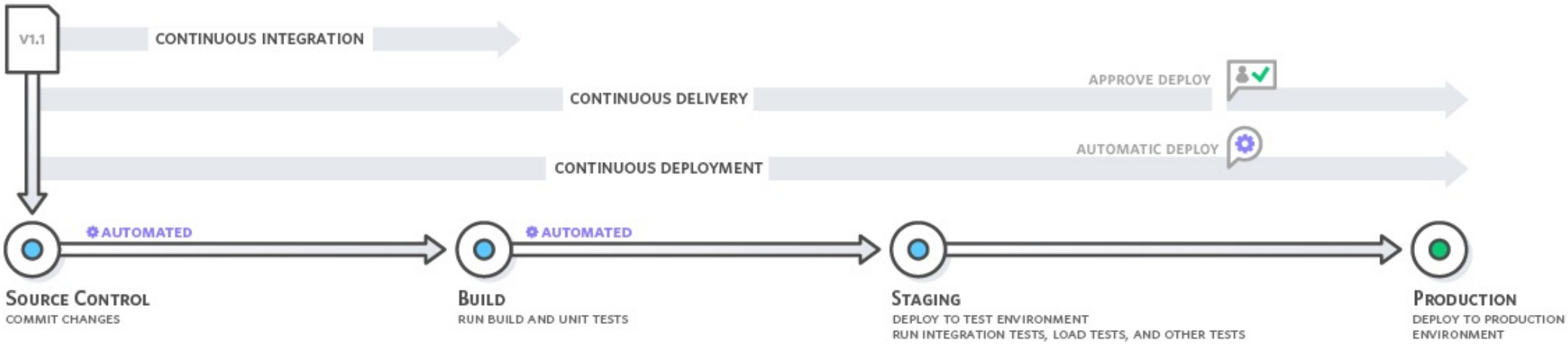
- Incremental development – several releases
- Planning based on user stories
- Each iteration touches all life-cycle activities
- Testing – unit testing for deliverables; acceptance tests for each release
- Flexible Design – evolution vs. big upfront effort
- Reflection after each release cycle
- Several technical and customer focused presentation opportunities

Key Agile Components

- User Stories
  - Requirements elicitation
  - Planning – scope & composition
- Evolutionary Design
  - Opportunity to make mistakes
- Test driven development
  - Dispels notion of testing as an end of cycle activity
- **Continuous Integration**
  - Code (small booms vs big bang)
- **Refactoring**
  - Small changes to code base to maintain design entropy
- Team Skills
  - Collaborative Development (Pair programming)
  - Reflections (process improvement)
- Communication/shared ownership
  - Interacting with customer / team members

Continuous Integration

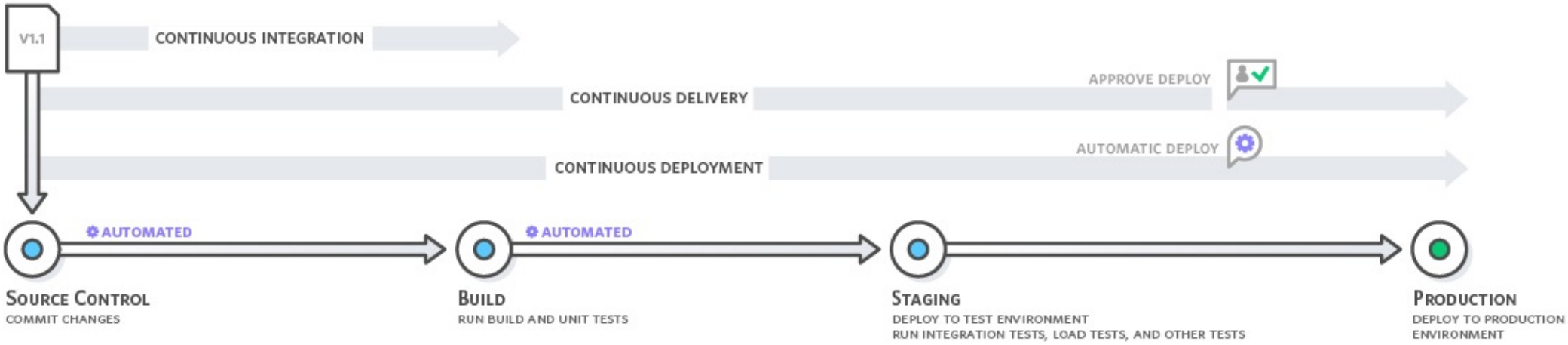
- Developers merge their code into the central repository regularly
- Automated builds and testing



Imagesource:aws.amazon.cor

Continuous Delivery

- Deploys all code changes to a testing and/or production environment after the build stage
- Deployment is manual



Imagesource:aws.amazon.cor

Developers

- Designing
- Coding
- Testing, bug tracking, reviews
- Continuous Integration
- ...

Operations

Managing/Allocating hardware/OS updates/resources, database

Monitoring load spikes, performance, crashes hardware updates

Backups, Rollback releases

etc.

- ✓ Can there be better coordination between Developers and Operators?
- ✓ Reduce issues while moving changes from development to production
- ✓ Configurations as code
- ✓ Automation (Delivery and Monitoring)

DevOps

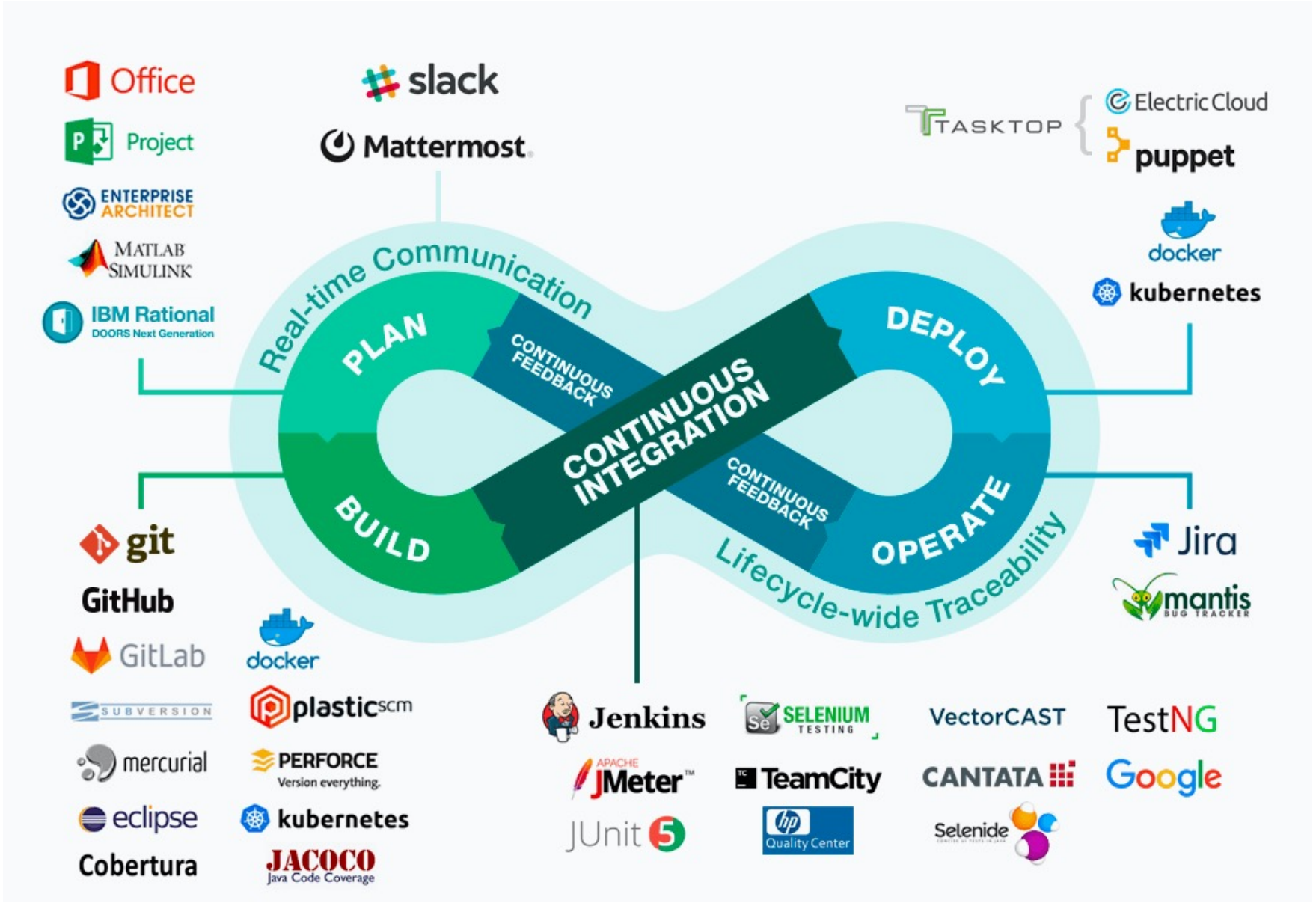
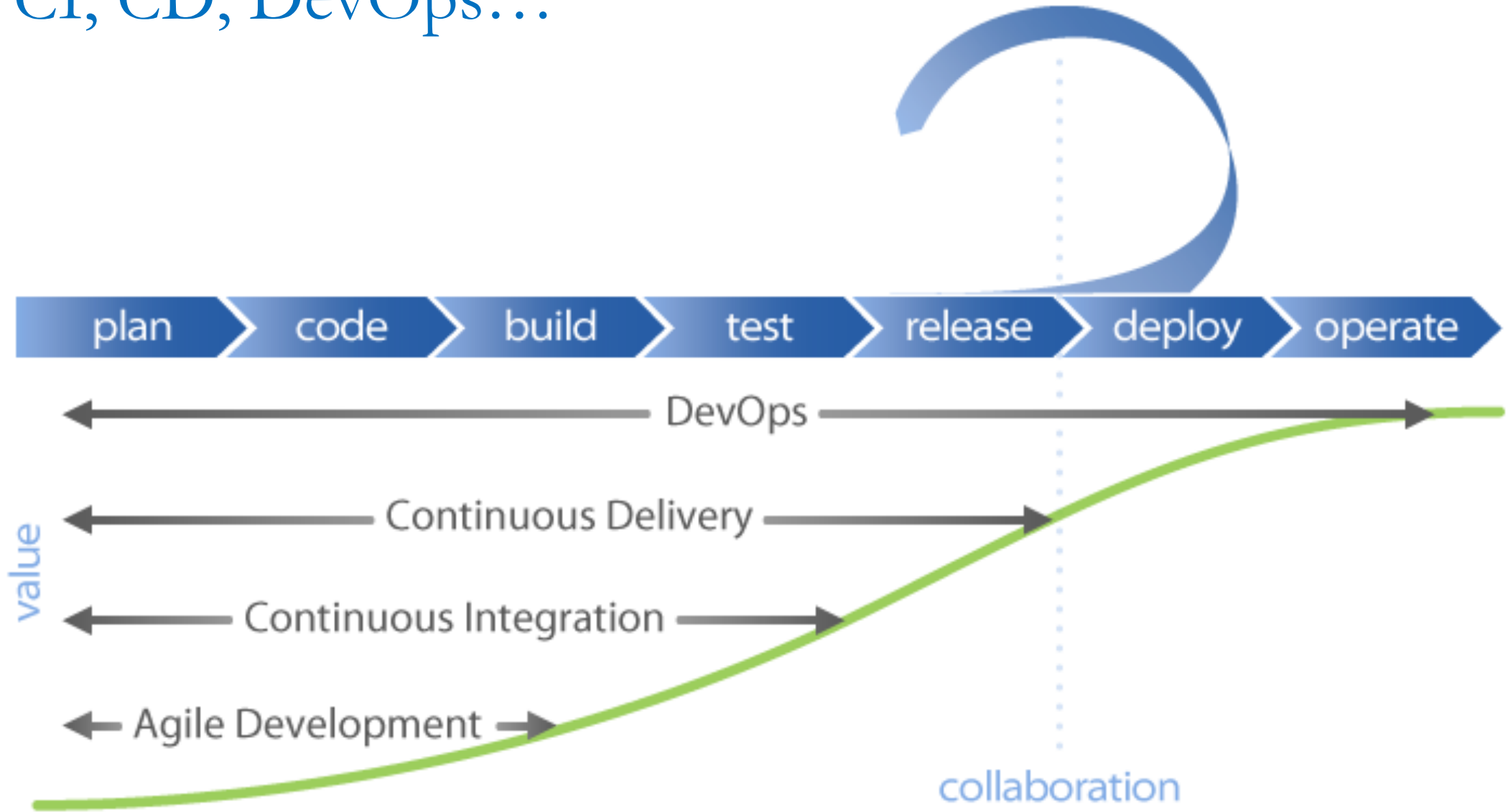


Image Source: https://intland.com/wp-content/uploads/2019/07/devops-infinity-1-1.png

Agile, CI, CD, DevOps…

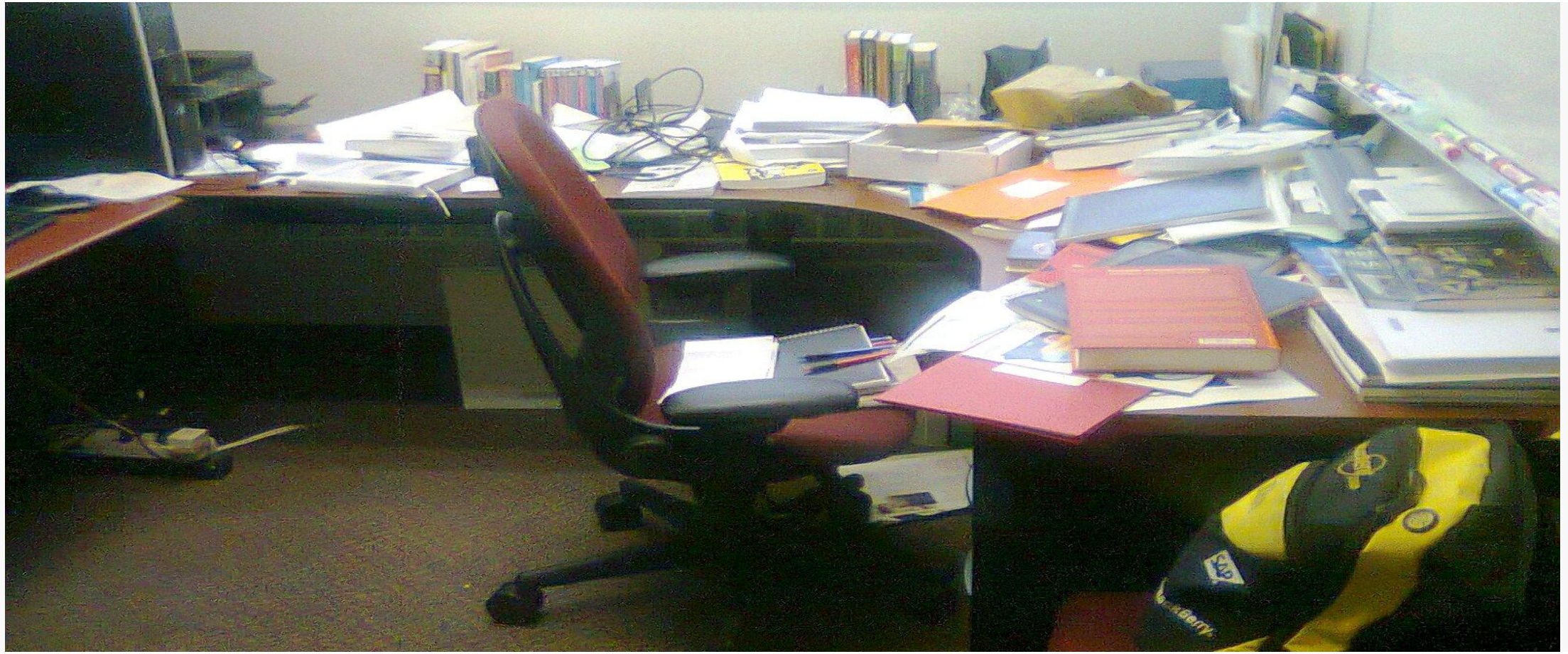


image@CollabNet

Refactoring

(Some material adapted from Martin Fowler's book)

From this…



Source: Mike Lutz, RIT

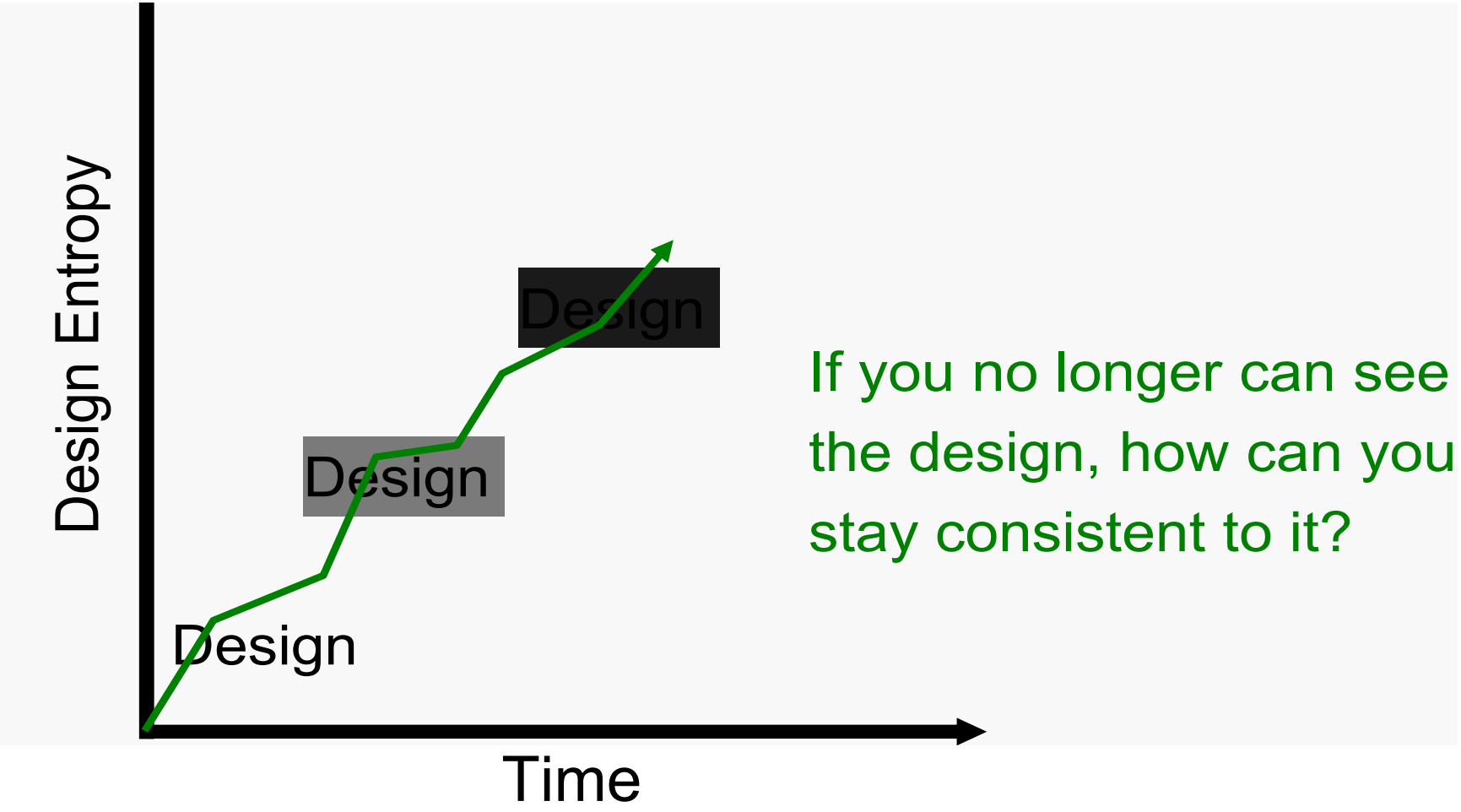
… To this

Source: Mike Lutz, RIT

It is usually hard to counter, “If it ain’t broke, don’t fix it.”

- Generally improves product quality
- Pay today to ease work tomorrow
- May actually accelerate today’s work

Design Entropy Vs Time



What causes Design Smells?

- Violation of design principles
  - java.util.Calender
    - Violates Single responsibility principle (overloaded with date functionality and time functionality)
  - java.util.Stack
    - extends java.util.Vector
    - Do Stack and Vector share a IS-A relationship?
    - Violates principle of hierarchy (Broken Hierarchy – substitutability is broken)

What causes Design Smells?

- Inappropriate use of patterns
- Language limitations
- Procedural thinking in OO
- Viscosity
  - Software viscosity
  - Environment viscosity
- Non-adherence to best practices and processes

Code Smells Within Classes

- **Comments**
  - Are the comments necessary?
  - Do they explain "why" and not "what"?
  - Can you refactor the code so the comments aren't required?
  - Remember, you're writing comments for people, not machines.
- **Long Method**
  - Shorter method is easier to read, easier to understand, and easier to troubleshoot.
  - Refactor long methods into smaller methods if you can
- **Long Parameter List**
  - The more parameters a method has, the more complex it is.
  - Limit the number of parameters you need in a given method, or use an object to combine the parameters.
- **Duplicated code**
  - Stamp out duplication whenever possible.
  - Don't Repeat Yourself!

Code Smells Within Classes

- **Conditional Complexity**
  - large conditional logic blocks, particularly blocks that tend to grow larger or change significantly over time.
  - Consider alternative object-oriented approaches such as decorator, strategy, or state.

- **Combinitorial Explosion**
  - Lots of code that does *almost* the same thing.. but with tiny variations in data or behavior.
  - This can be difficult to refactor-- perhaps using generics or an interpreter?

- **Large Class**
  - Large classes, like long methods, are difficult to read, understand, and troubleshoot.
  - Large class can be restructured or broken into smaller

Code Smells Between Classes

- **Indecent Exposure**
  - Classes that unnecessarily expose their internals.
  - Aggressively refactor classes to minimize their public surface.
  - You should have a compelling reason for every item you make public. If you don't, hide it.
- **Feature Envy**
  - Methods that make extensive use of another class may belong in another class.
  - Move the method to the class it is so envious

Code Smells between Classes

- **Lazy Class**
  - Classes should pull their weight.
  - If a class isn't doing enough to pay for itself, it should be collapsed or combined into another class.
- Solution Sprawl
  - If it takes five classes to do anything useful, you might have solution sprawl.
  - Consider simplifying and consolidating your design.
- **Middle Man**
  - If a class is delegating all its work., then cut out the middleman.
  - Beware classes that are merely wrappers over other classes or existing functionality in the framework.

Refactoring

- As a software system grows, the overall design often suffers
- In the short term, working in the existing design is cheaper than doing a redesign
- In the long term, the redesign decreases total costs
  - Extensions
  - Maintenance
  - Understanding
- Refactoring is a set of techniques that reduce the short-term pain of redesigning
  - Not adding functionality
  - Changing structure to make it easier to understand and extend

The Scope of Refactoring

- Small steps:
  - Rename a method
  - Move a field from one class to another
  - Merge two similar methods in different classes into one common method in a base class
- Each individual step is small, and easily verified/tested
- The composite effect can be a complete transformation of a system

Principles

- Don’t refactor and extend a system at the same time
  - Make a clear separation between the two activities
- Have good tests in place before you begin refactoring
  - Run the tests often
  - Catch defects immediately
- Take small steps
  - Many localized changes result in a larger-scale change
  - Test after each small step

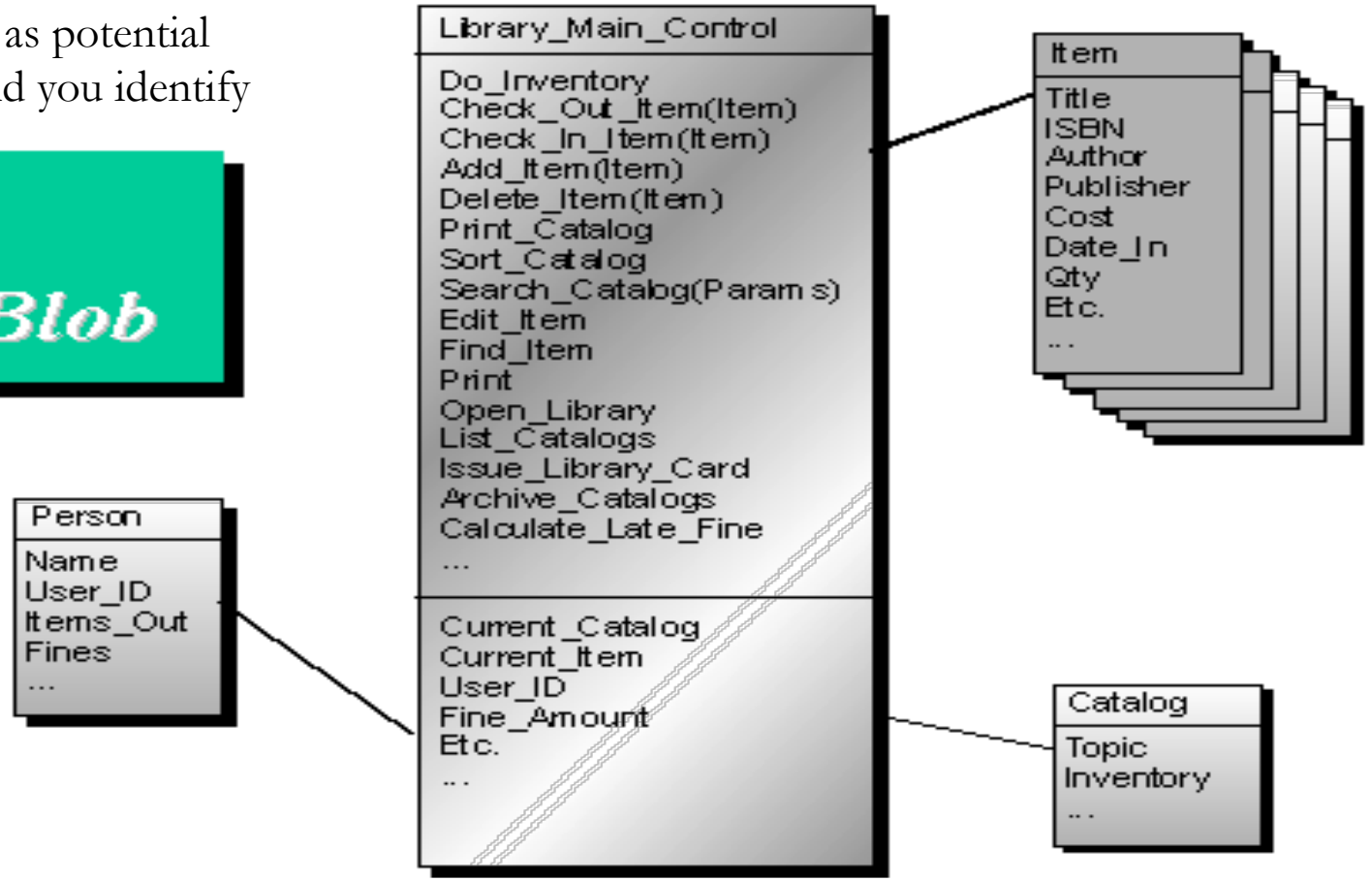
When Should You Refactor?

- You're extending a system, and realize it could be done better by changing the original structure
  - Stop and refactor first
- The code is hard to understand
  - Refactor to gain understanding, and leave the code better than it was

Refactoring Library system – Existing design

What areas do you see as potential problem areas? Why did you identify each of those areas?

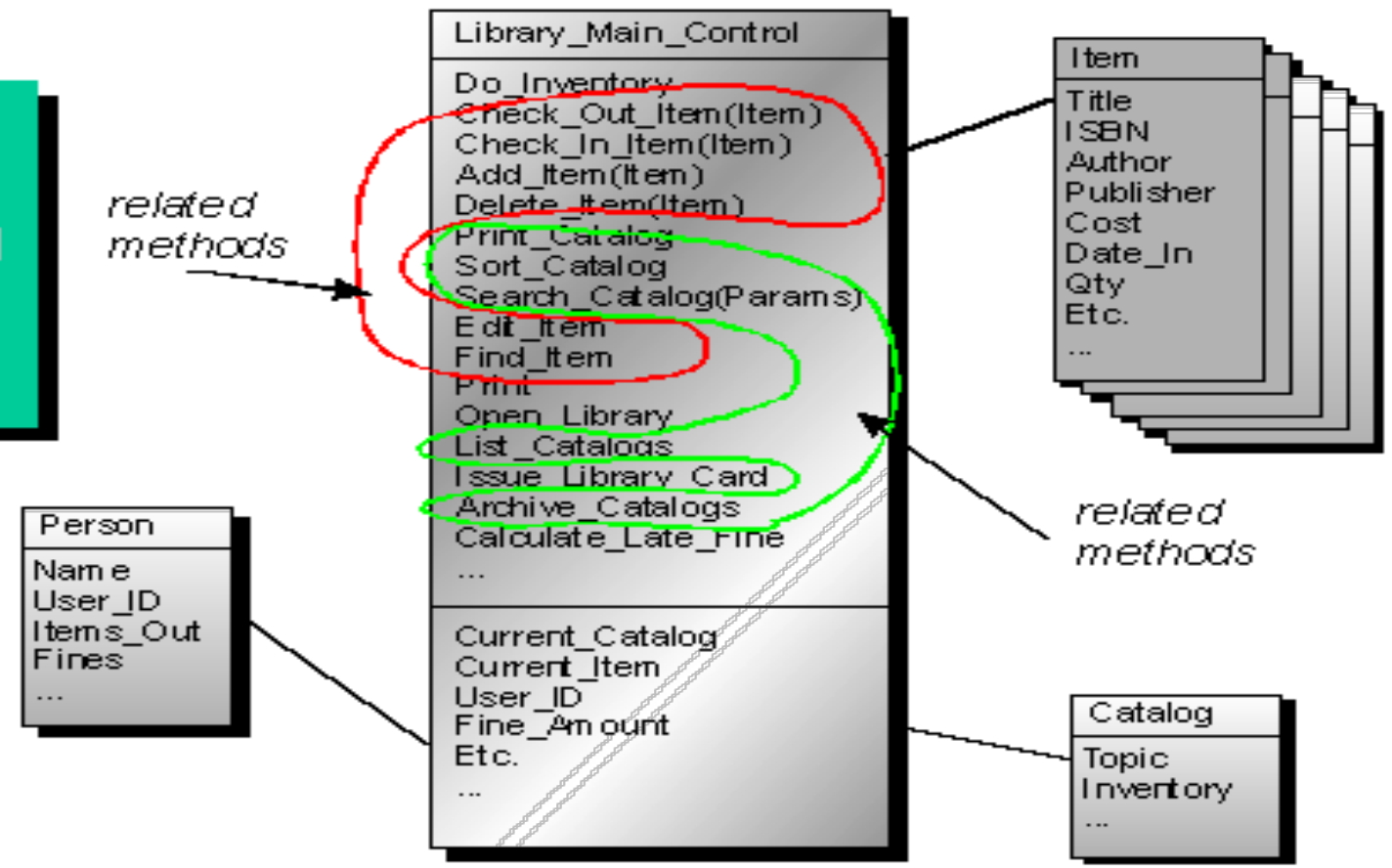
Example:
The Library Blob



Source: MITRE

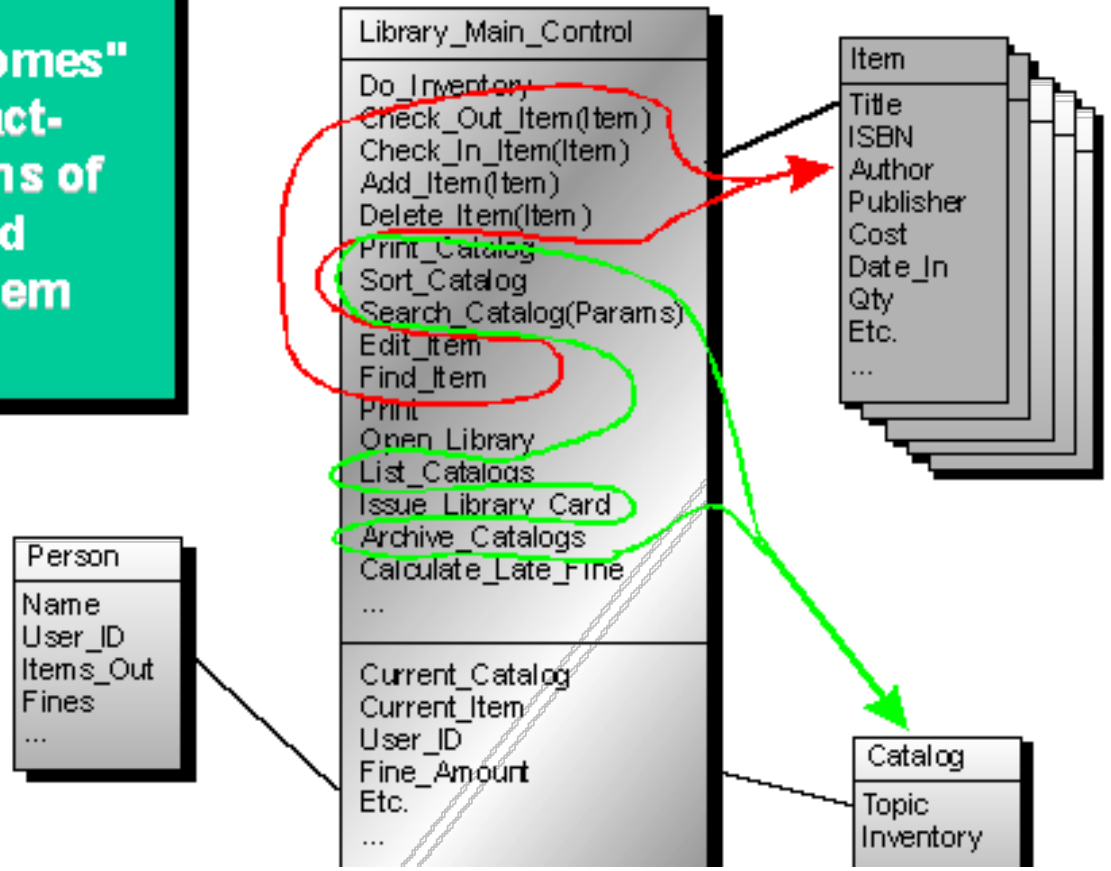
Library system – Changing the design

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

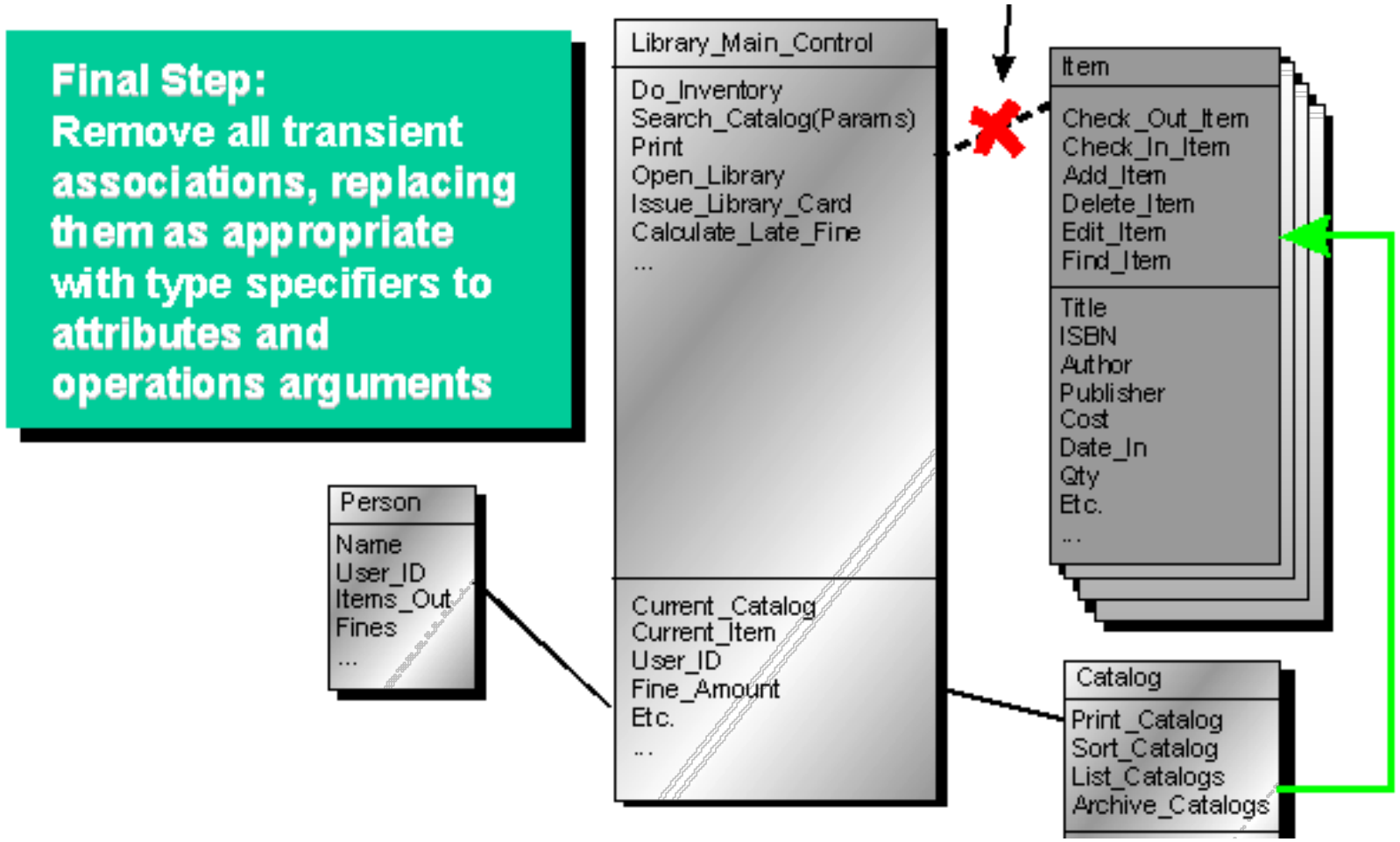


Library system – Changing the design

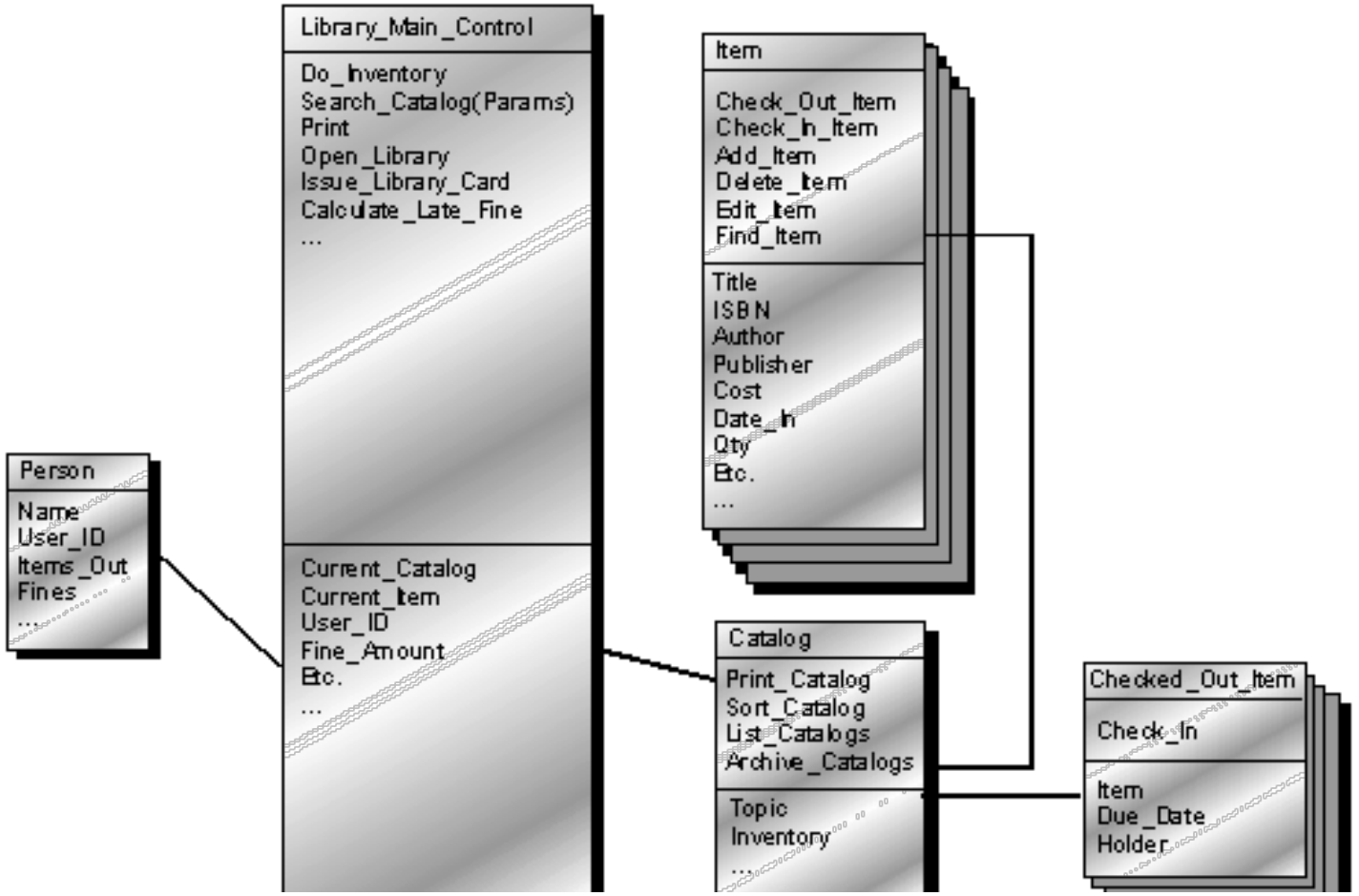
Step 2:
Find "natural homes" for these contract-based collections of functionality and them migrate them there



Library system – Changing the design



Library system – Changing the design



Refactoring Exercise

```
public int getScore()
{
      int result;
      result = (int)(Math.random() * 6) + 1;
      dice[0].setFaceValue(result);

      result = (int)(Math.random() * 6) + 1;
      dice[1].setFaceValue(result);

      int score = dice[0].getFaceValue() +
                  dice[1].getFaceValue();
      return score;
}
```

/* Assume that dice is an array of Die objects and has access to 'faceValue' property */

Writing test cases…

- Prepare the test cases before any/every change made…

For example, a test framework such as JUnit can check the values:

```
assertTrue(diceValue >= 2 && diceValue <=12);
```

Refactoring No. 1 - Self Encapsulate field

```
dice[0].setFaceValue(result)
Gets replaced by

getDice(0).setFaceValue(result)

=====================================
public int getScore()
{
        int result;
        result = (int)(Math.random() * 6) + 1;
        getDice(0).setFaceValue(result);

        result = (int)(Math.random() * 6) + 1;
        getDice(1).setFaceValue(result);

        int score = getDice(0).getFaceValue() + getDice(1).getFaceValue();
        return score;
}
```

/* This refactoring tells us not to directly access an object's fields within its methods, but to use accessor methods */

Refactoring No. 2 - Extract Method

```
// roll the die
result = (int)(Math.random() * 6) + 1;
```

can become

```
result = rollDie();
```

==========

```
   public int getScore()
   {
          int result;
          result = rollDie();
          getDice(0).setFaceValue(result);
          result = rollDie();
          getDice(1).setFaceValue(result);
          int score = getDice(0).getFaceValue() + getDice(1).getFaceValue();
          return score;
   }

   public int rollDie() {
          return (int)(Math.random() * 6) + 1;
   }
```

/* This refactoring tells us to extract lines of code from long method and make it a separate method */

Refactoring No. 3 – Rename method/class/variable/etc.

```
Change getScore to ThrowDice()

It might be confusing if player scores are to be computed
==================================
public int ThrowDice()
{
        int result;
        result = rollDie();
        getDice(0).setFaceValue(result);
        result = rollDie();
        getDice(1).setFaceValue(result);
        int score = getDice(0).getFaceValue() + getDice(1).getFaceValue();
        return score;
}

public int rollDie() {
        return (int)(Math.random() * 6) + 1;
}
```

/* Changing the names in code (of classes, methods, variables etc.) to be more meaningful can make a positive contribution to code readability */

Refactoring No. 4 – Replace Temp with Query

```
public int ThrowDice()
{
	int result;
	result = rollDie();
	getDice(0).setFaceValue(result);
	result = rollDie();
	getDice(1).setFaceValue(result);
	return getDiceValue();
}

public int rollDie() {
	return (int)(Math.random() * 6) + 1;
}

// replace temp variable score with query
public int getDiceValue() {
int score = getDice(0).getFaceValue() + getDice(1).getFaceValue();
return getDice(0).getFaceValue() + getDice(1).getFaceValue();
	return score;
}
```

/* This refactoring encourages us to use methods directly in code rather then storing their results in temporary variables.*/

Refactoring No. 5 – Move Method

```
public void roll() {
  setFaceValue((int)(Math.random() * 6) + 1);
}
==========================================
    public int ThrowDice(){
            int result;
            result = rollDie();
            getDice(0).setFaceValue(result);
            getDice(0).roll();
            result = rollDie();
            getDice(1).setFaceValue(result);
            getDice(1).roll()
            return getDiceValue();
    }

    public int rollDie() {
            return (int)(Math.random() * 6) + 1;
    }
  public void roll() {
            setFaceValue((int)(Math.random() * 6) + 1);
  }
   public int getDiceValue() {
            return getDice(0).getFaceValue() + getDice(1).getFaceValue();
    }
```

/* This refactoring involves moving a method from one class to another, so can potentially be quite difficult because of the possible side effects */

AntiPatterns

Material adapted from "AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis" by William J Brown et. al.

AntiPatterns

- A pattern of practice that is commonly found in use
- A pattern which when practiced usually results in *negative* consequences
- Patterns defined in several categories of software development
  - Design
  - Architecture
  - Project Management

Pattern Vs AntiPattern

- Patterns
  - Usually bottom up
  - Begin with recurring solution
  - Then the forces and context
  - Usually leads to one solution
- AntiPatterns
  - Top down
  - Begin with commonly recurring practice
  - Obvious negative consequences
  - Symptoms are past and present; consequences go into the future

Software Design AntiPatterns

- AntiPatterns
  - The Blob
  - Lava Flow
  - Functional Decomposition
  - Poltergeists
  - Golden Hammer
  - Spaghetti Code
  - Cut-and-Paste Programming
- Mini-AntiPatterns
  - Continuous Obsolescence
  - Ambiguous Viewpoint
  - Boat Anchor
  - Dead End
  - Input Kludge
  - Walking through a Minefield
  - Mushroom Management

The Blob

- AKA
  - Winnebago, The God Class, Kitchen Sink Class
- Causes
  - Sloth, haste
- Unbalanced Forces:
  - Management of Functionality, Performance, Complexity
- Anecdotal Evidence:
  - “This is the class that is really the *heart* of our architecture.”

The Blob (2)

- Like the blob in the movie can consume entire object-oriented architectures
- Symptoms
  - Single controller class, multiple simple data classes
  - No object-oriented design, i.e. all in main
  - Start with a legacy design
- Problems
  - Too complex to test or reuse
  - Expensive to load into system

Causes

- Lack of OO architecture
- Lack of any architecture
- Lack of architecture enforcement
- Limited refactoring intervention
- Iterative development
    - Proof-of-concept to prototype to production
    - Allocation of responsibilities not repartitioned

Solution

- Identify or categorize related attributes and operations
- Migrate functionality to data classes
- Remove far couplings and migrate to data classes

Lava Flow

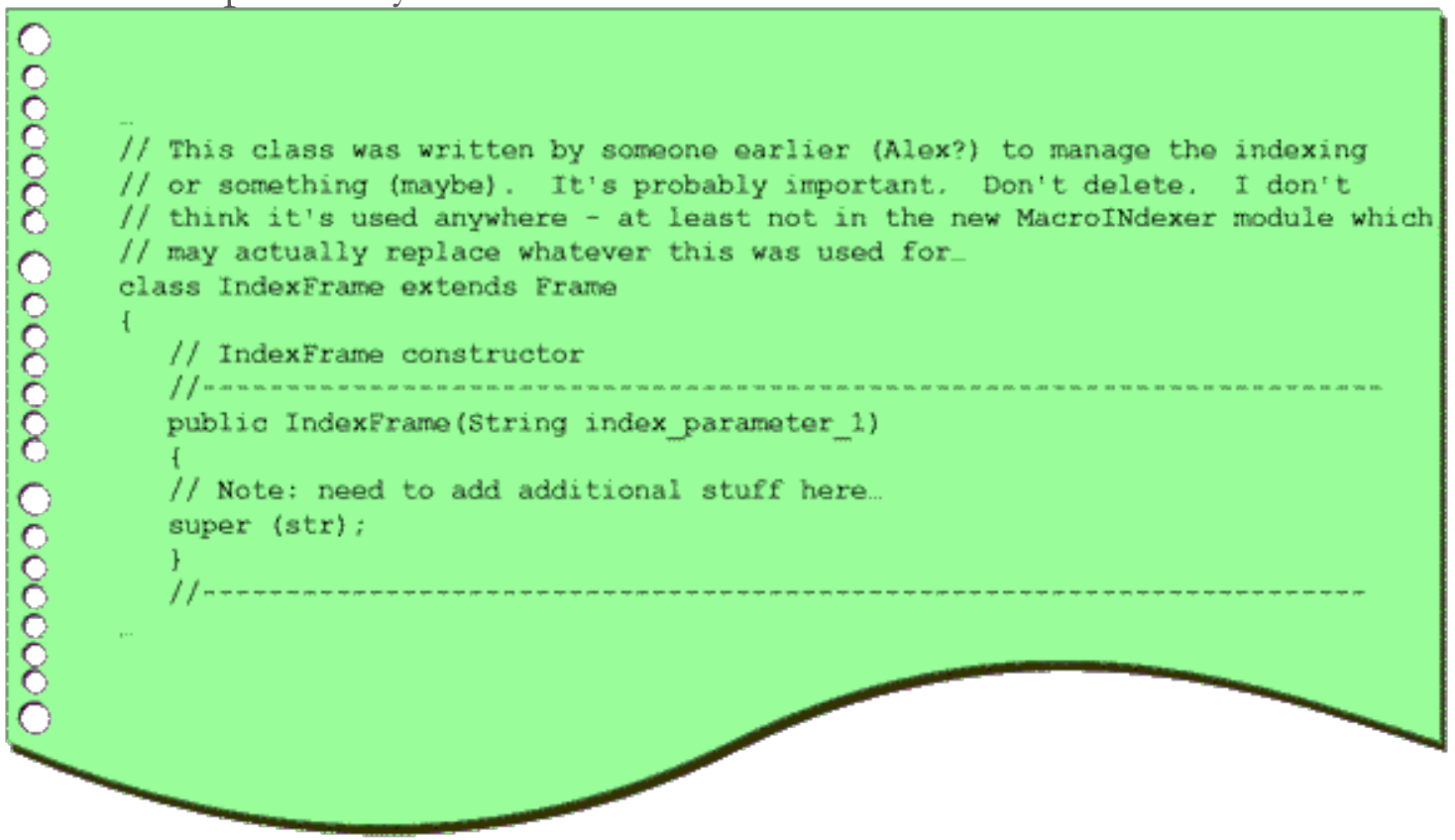
- AKA
  - Dead Code
- Causes
  - Avarice, Greed, Sloth
- Unbalanced Forces
  - Management of Functionality, Performance, Complexity

Symptoms and Consequences

- Unjustifiable variables and code fragments
- Undocumented complex, important-looking functions, classes
- Large commented-out code with no explanations
- Lot’s of “to be replaced” code
- Obsolete interfaces in header files
- Proliferates as code is reused

Causes

- Research code moved into production
- Uncontrolled distribution of unfinished code
- No configuration management in place
- Repetitive development cycle



```
…
// 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);
   }
   //-------------------------------------------------------------------------
…
```

Solution

- Don’t get to that point
- Have stable, well-defined interfaces
- Slowly remove dead code; gain a full understanding of any bugs introduced
- Strong architecture moving forward

Cut-and-Paste Programming

- AKA
  - Clipboard Coding
- Root Causes
  - Sloth
- Unbalanced Forces
  - Management of Resources, Technology Transfer
- Anecdotal Evidence
  - “Hey, I thought you fixed that bug already, so why is it doing this again?” “Man, you guys work fast. Over 400,000 lines of code in three weeks is outstanding progress!”

Symptoms and Consequences

- Same software bug reoccurs
- Code can be reused with a minimum of effort
- Causes excessive maintenance costs
- Multiple unique bug fixes develop
- Inflates LOC without reducing maintenance costs

Causes

- Requires effort to create reusable code; must reward for long-term investment
- Context or intent of module not preserved
- Development speed overshadows all other factors
- “Not-invented-here” reduces reuse
- People unfamiliar with new technology or tools just modify a working example

Solution

- Code mining to find duplicate sections of code
- Refactoring to develop standard version
- Configuration management to assist in prevention of future occurrence

Refactoring summary…

- Designs can deteriorate over a period of time
- Refactoring can help in managing the deterioration of design
  - One small step at a time
  - Don't refactor and add functionality at the same time

Systems are fairly large nowadays. Systems evolve over time and complexity needs to be managed!

Of course it does not make sense to **manually refactor!!!**

As a Software Engineer, one of the most important aspect of your job is to make the life of other developers/programmers/testers easier!!!