Lessons Learned: Architectural Renovation

Jeff Certain
Application Architect
Colorado CustomWare

About Me… About The Talk

- Code monkey
- Microsoft MVP
- Technical expert

- A survivor of architectural renovation

- Here to share my experiences
  - Descriptive, not prescriptive

Why Not Refactoring?

- “Refactoring” implies a certain amount of cleanliness and control
  - Has unit tests to ensure that the functionality doesn’t change
  - Small, tested steps
- Sometimes the changes are more than merely cosmetic
  - Deep structural changes may not be covered by unit tests

The Mental Image

Renovation

- Renovation is rarely clean
  - Often have to remove, wreck or destroy what is currently in place
  - A messy, dusty, dirty process
  - Often, once you've torn out the old, you discover:
    - Something that needs to be fixed
      - Old electrical or plumbing
    - Something that needs to be remediate
      - Mold or asbestos
- Replacing the foundation or structural members takes time, caution and effort

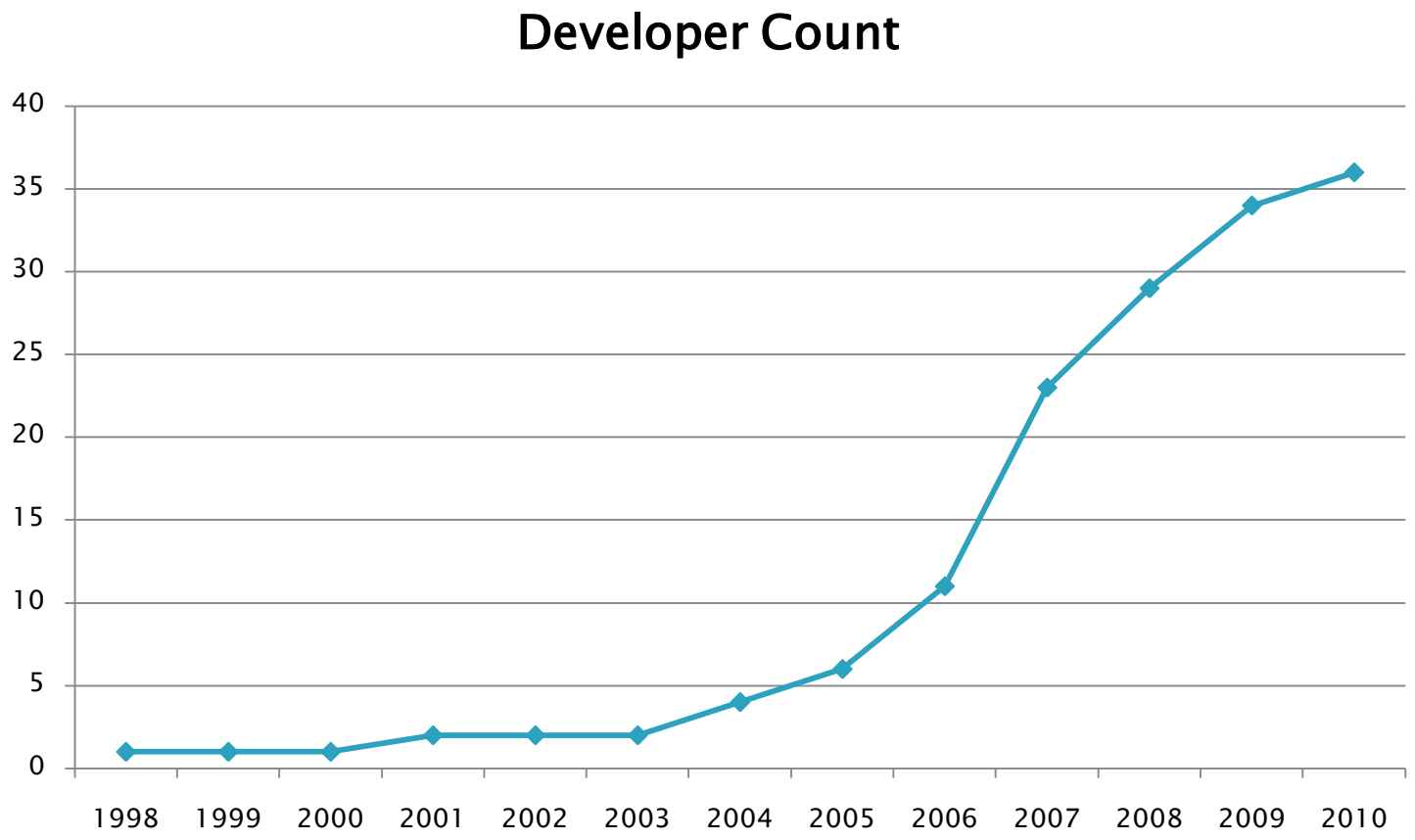
Overview

- Context
- Approaches to Refactoring
- Generating Consensus
- Tactics
- Retrospective

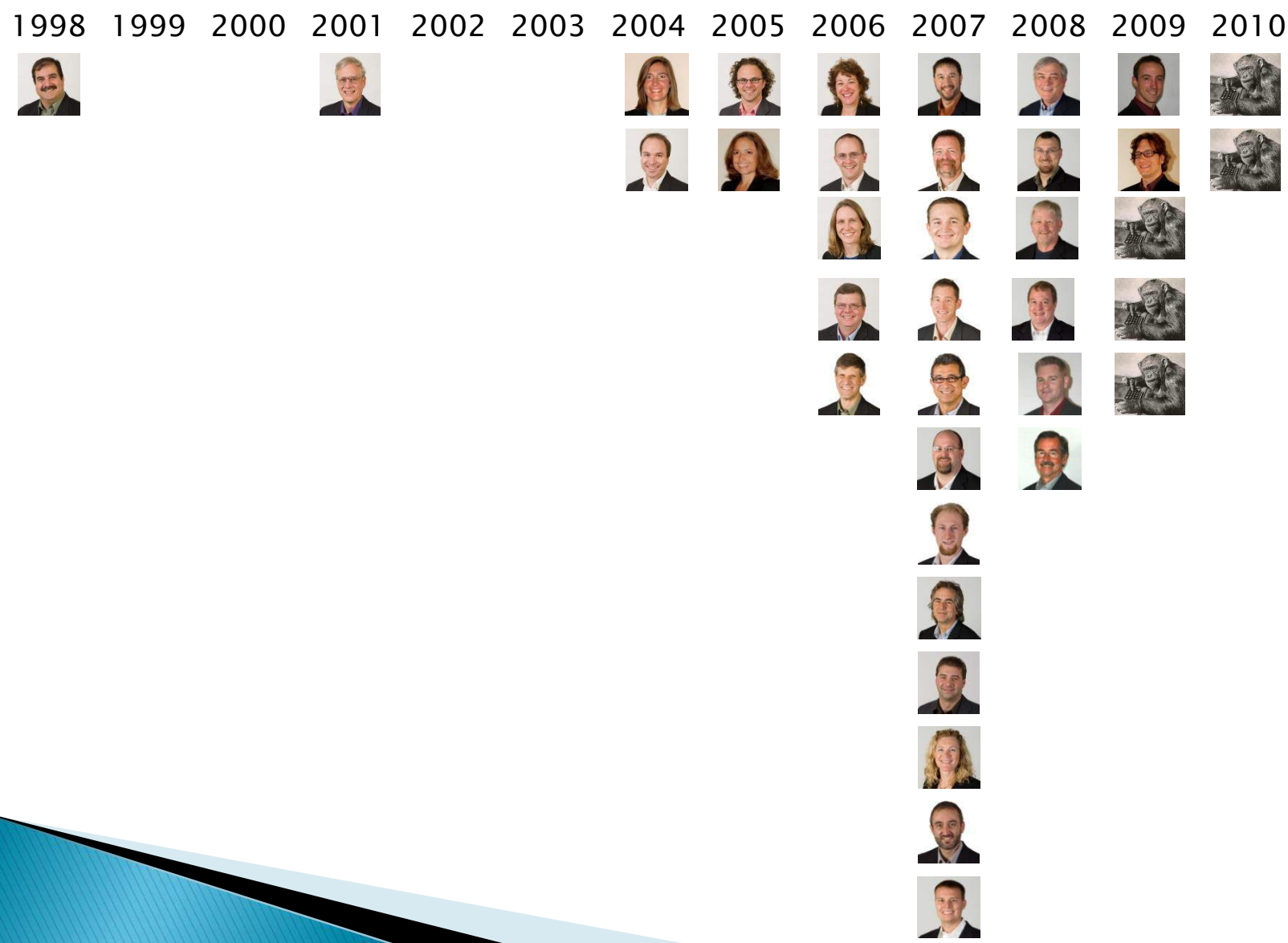
Context: Company

- 20-year-old company created to make our customers' lives better
- 95% retention rate
- Original product targeted at county assessor's office
- Oracle back-end, Access front-end

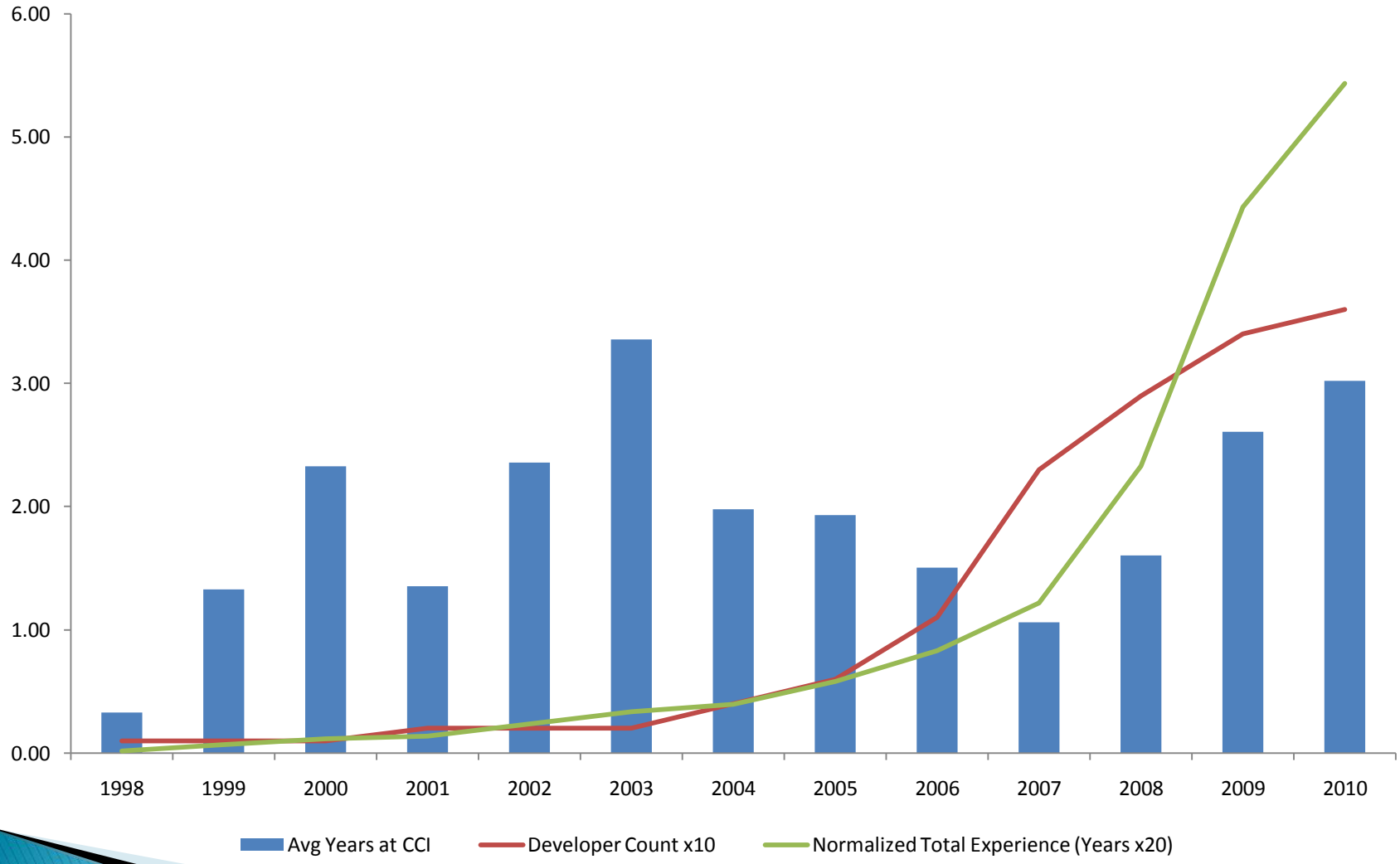
Development Team



Development Team Growth



Development Team



Context: The Idea

- Previous version was the requirement
- Senior developer became architect
- Implemented by developers from previous product
- Estimate: 6 people, 2 years

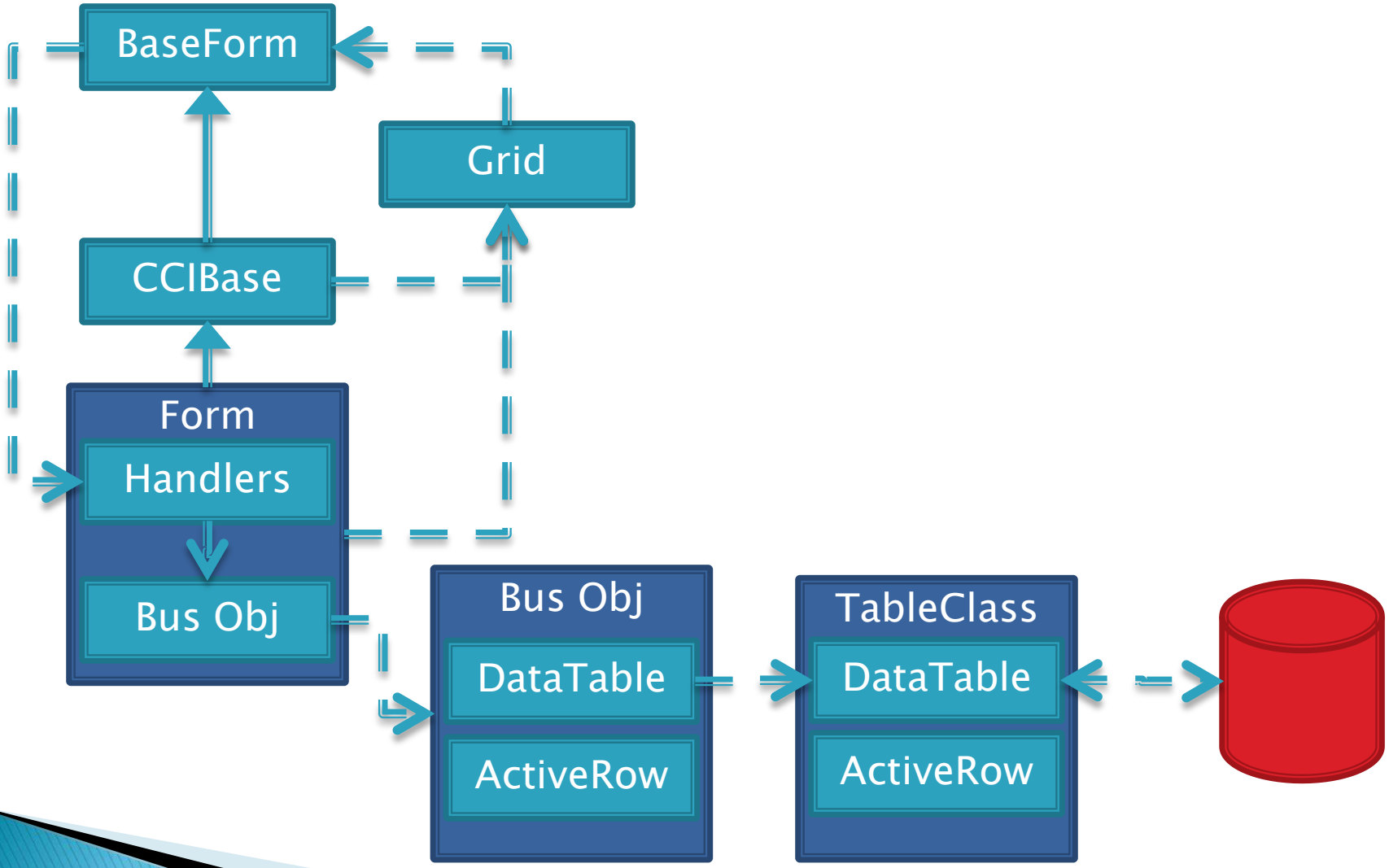
Context: The Reality

- 4+ product-oriented development teams
- 30+ developers, 5 architects
- First release in late 2009
- Over $7M in privately funded development

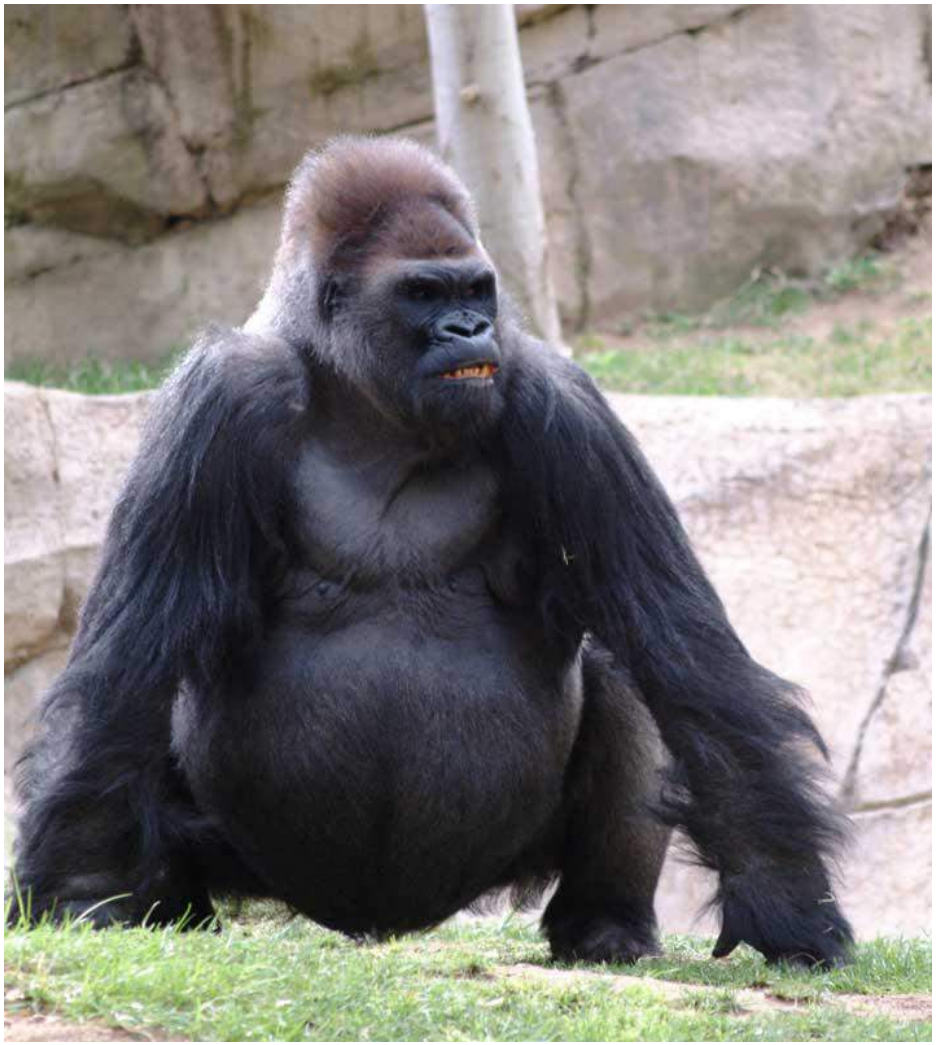
Context: Two Years Ago

- Over 1.5 million lines of code (VB.NET)
- No true business objects
- Data stored in DataTables
- One-to-one mapping between the database schema and tables in memory
- Issues with data binding in .NET 1.x lead to 2 copies of the data being maintained
- Heavy use of reflection calls

The Architecture



Guerilla Warfare



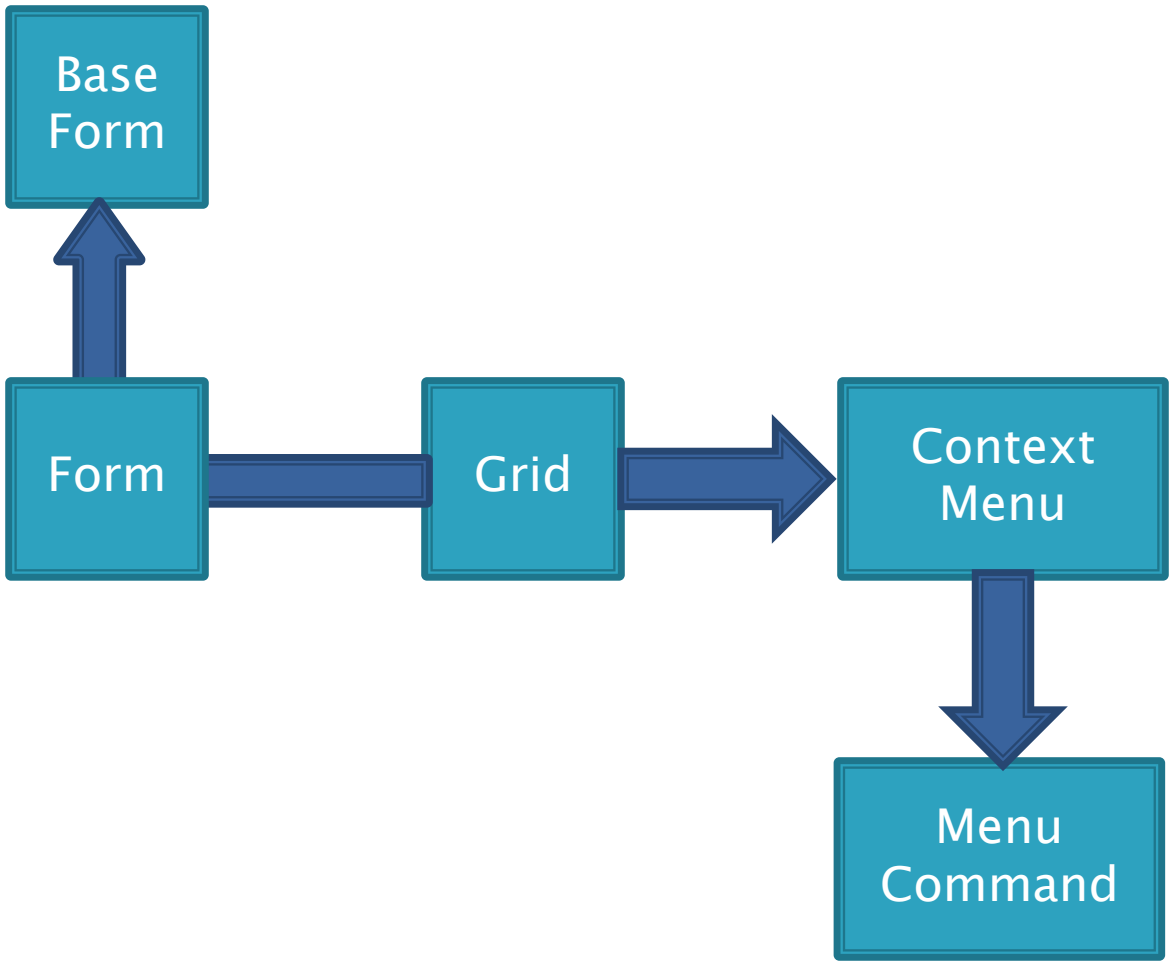
Targets of Opportunity

- Code generation templates
  - Drastic reduction in total code by implementing polymorphism rather than repetitious code
  - Increases velocity due to more dynamic nature of base classes
- Pain points
  - Target accidental complexity
  - Emphasize architectural change
- Introduce design patterns

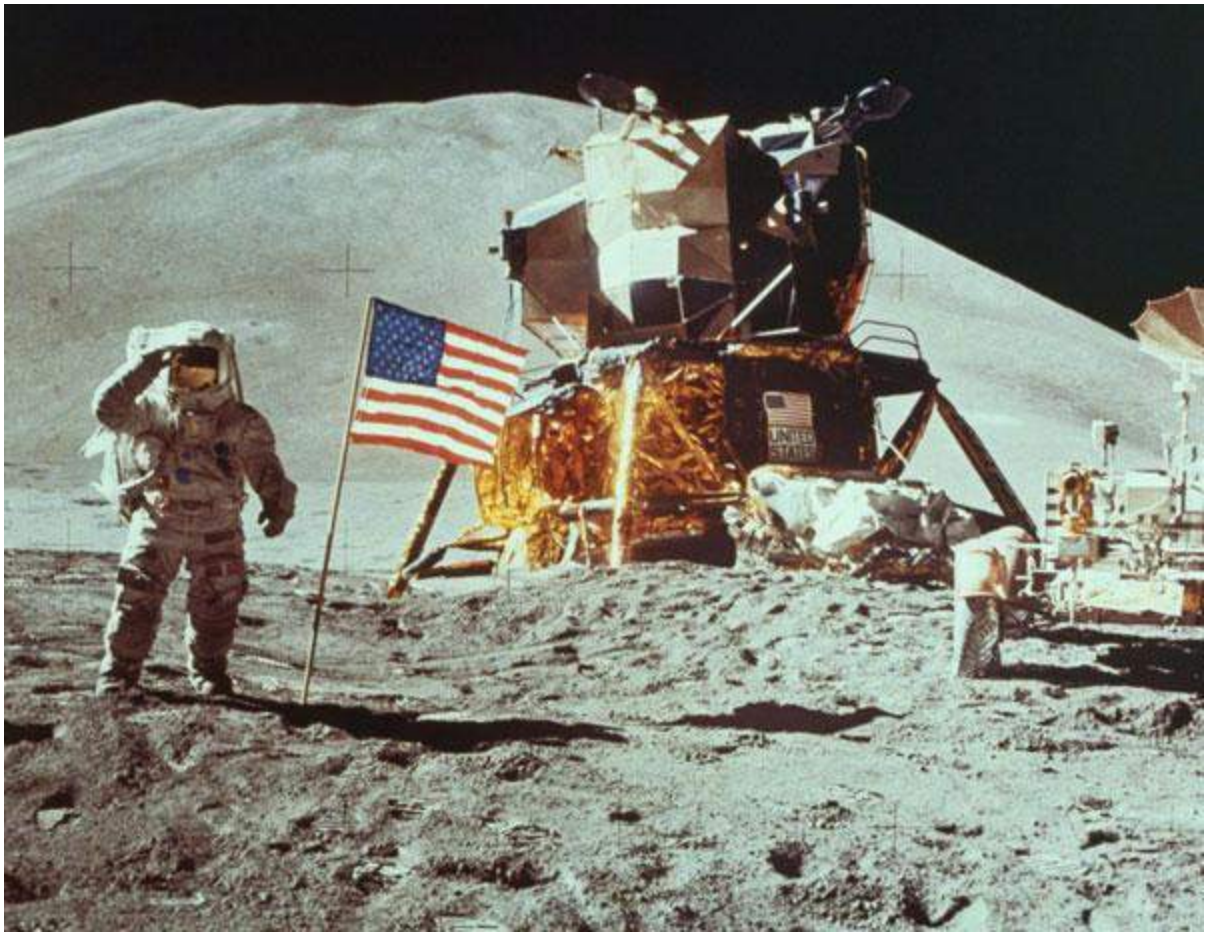
Some Changes

- Modified code generation templates
  - Used base class(es) rather than repeating code in template
  - Increased velocity since architecture can be tweaked without running code generation
  - Drastically reduced amount of code
    - Dropped ~900KLoC compiled code
- When encountering bugs caused by limitations of the architecture, fix the architecture

Context Menus



Some Things Require a Team…



Convincing Management

Management

- Six person leadership team
  - Combined 68+ years at CCI
- Current CEO and president is the founder of the company
- VP of Software has been with the company since Access 2 version

Building Consensus

- Survey circulated among all developers identified a single, common source of pain
- Team leads agreed that change was needed
- Took proposal to leadership team

Identify Concrete Benefits

- Make the business case
- Concrete reasons were given
  - Reduce pain for developers
  - Reduce bug count
  - Increase velocity going forward
  - Easier to realize product roadmap
- “Investment in the future”
  - Reduced maintenance costs

“Technical Debt”

- Originally, Ward Cunningham used this term to refer to suboptimal design or implementation decisions that were *deliberately* made, tied to a specific learning objective
  - Essentially, deferring the final design or implementation until the last responsible moment
- Often, this term is used to mean any bad stuff in the code base

Key Factors: Experience and Trust

- Leadership team understood first-hand what maintenance costs the company in the long run
- Leadership team trusted technical staff enough to let us make recommendations with far-reaching ramifications

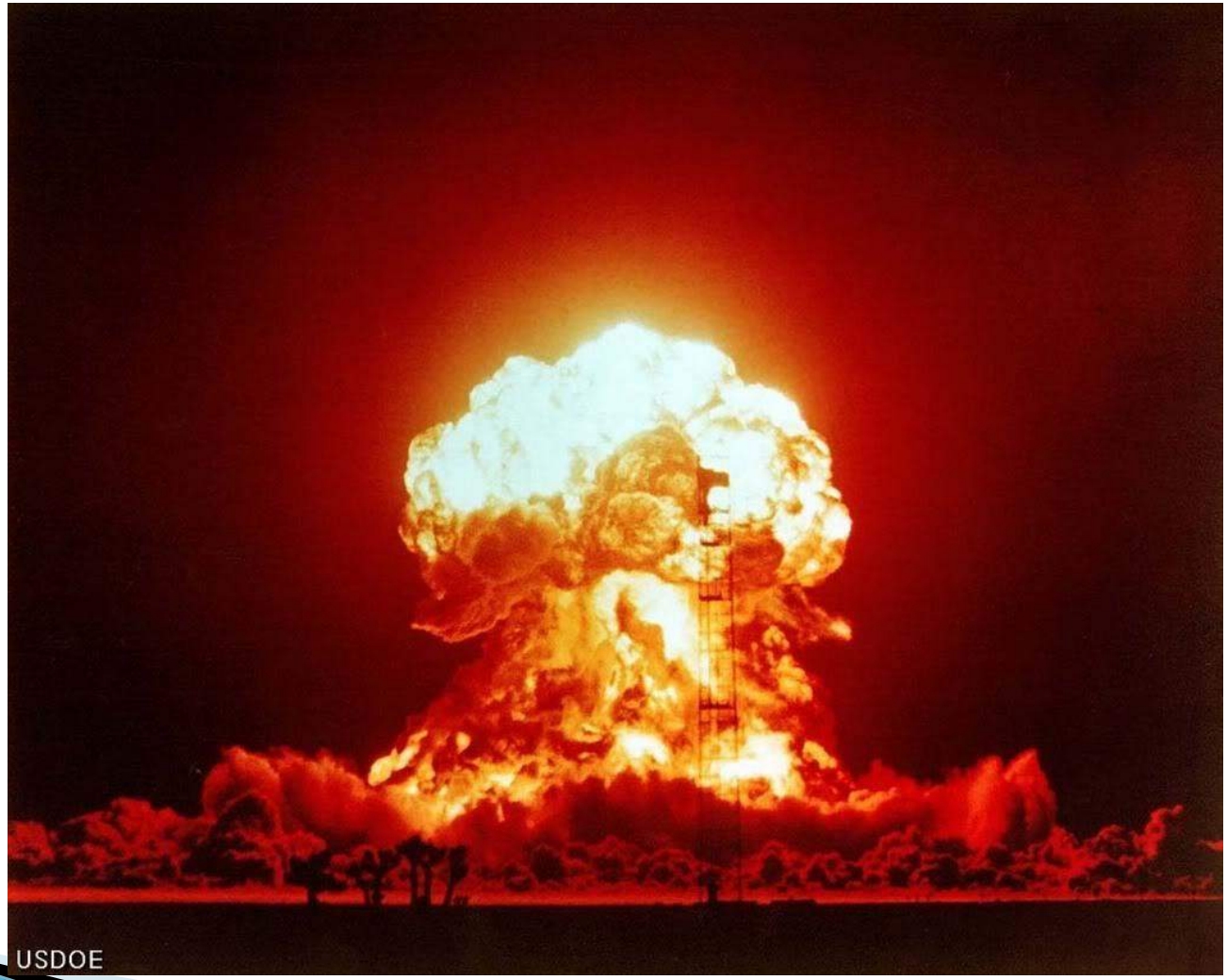
Now What??

- Now that you're convinced management…

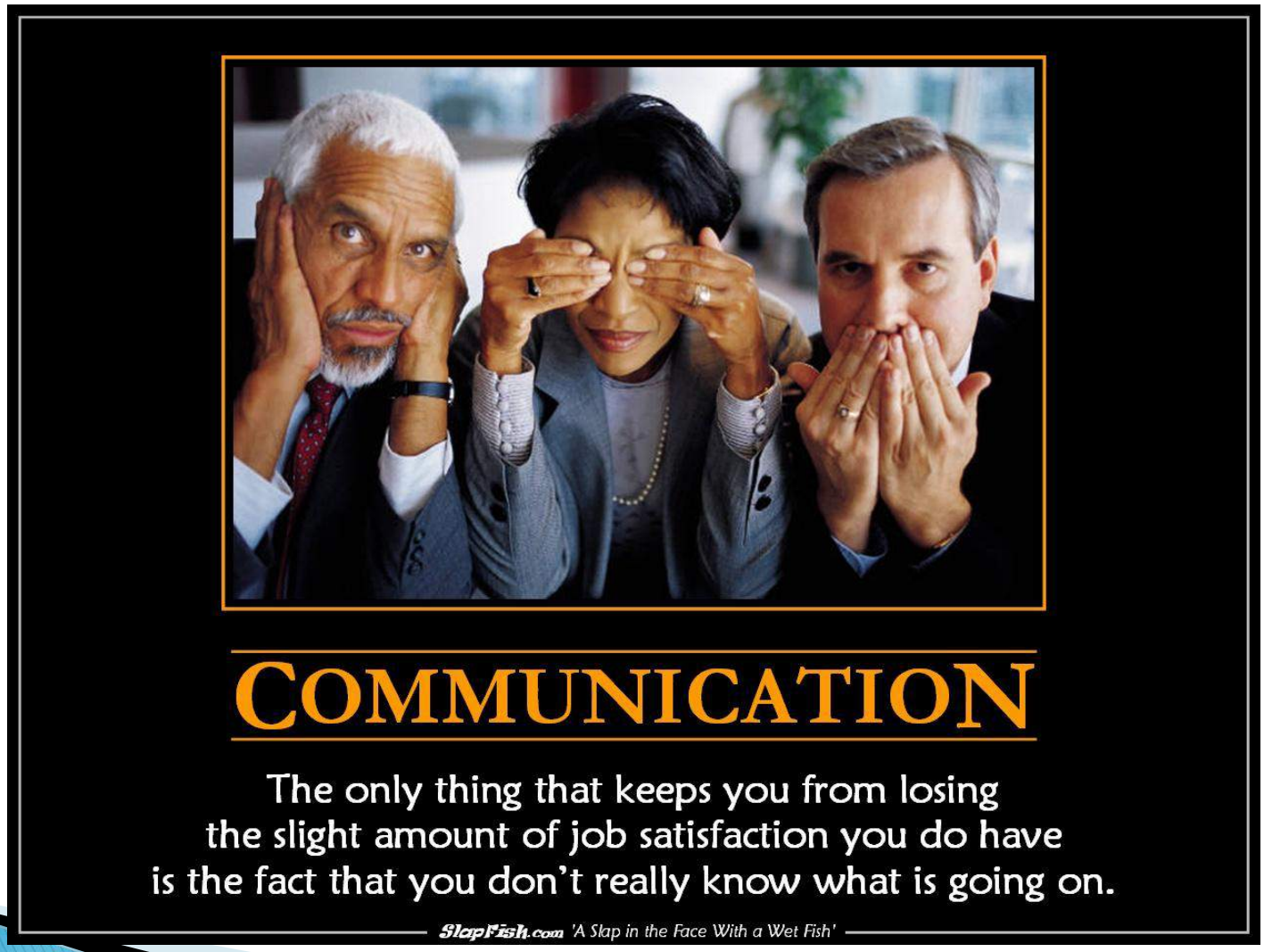
Some Considerations

- Before undertaking a major refactoring effort, here are some things you might want to consider…

Contain The Blast Radius?



Promulgate Change



Support the Team

Determine Change Mechanism

- Tactics
  - How are you going to push out changes?
  - Who needs to do what work?
  - What are the effects?
  - How are you going to deal with the unexpected?

The Change

- “TableClass” became a subclass of DataTable rather than containing a DataTable
  - Allowed better binding story with grids
- Data access separated out into separate class
- A new subclass of grid was created with event handlers wired in
  - Eliminated need for developers to manage active row

Results

- Team effort to remove duplicated data tables reduced memory footprint of application
- Reduced code by ~40%
  - 940KLoc total -> 740KLoC total
- Favorite quotes:
  - “Hey! I can actually *see* bugs in the code!”
  - “For the first time, I believe this code is maintainable.”

A Word about Metrics

- Know what you're measuring!
  - Generated code actually *got worse* from a "Maintainability Index" perspective
  - Removed many pass-through methods that we "highly maintainable" (i.e. few LoC, low CC, etc.)
  - As one developer said, it was like taking all the bright kids out of a class and realizing that you had problem children there after all

Unexpected Result

- Now that we've got a massive refactoring effort under our belt, management and developers are less scared about the next one

Retrospective

- After 2 years of adding features, code base is still smaller than it was
- Average maintainability is up 3 points

Retrospective

- Some unexpected things:
  - Developers used the opportunity to do a lot of cleanup that was outside the scope of the plan
  - Reflection calls reduced the need to developers to understand the event model… and as a result, not all of them did
  - Long history of the development effort meant that there was a lot of code that predated architectural components, and was “unique” or “special”

Takeaways

- Renovation isn't refactoring
  - Different considerations and scope
- Drive from a business value perspective
  - Technical debt is a powerful metaphor
- Significant architectural change *is* possible, even in a large project under active development

Questions?