



# Agile Development Concepts

Steve Bohlen

Senior Software Engineer

Skiff, LLC

“Courage is the power to let  
go of the familiar.”  
*-Raymond Lindquist*

# The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

- Individuals and interactions** over processes and tools
- Working software** over comprehensive documentation
- Customer collaboration** over contract negotiation
- Responding to change** over following a plan

That is, while there is value in the items on the **right**, we value the items on the **left** more.

## Thanks for Coming this Morning

Don't forget to complete an evaluation on  
your way out the door!

(kidding!)

## Some Aspects of Good OO Software Design

- Object-Oriented Principles
  - Encapsulation (data hiding)
  - Decoupled objects
  - Orthogonal class structure
  - S.O.L.I.D. principles (etc.)
- Layered Architecture
  - Should be separated into logical layers
  - These may or may not represent physical layers
- (certainly many others)

# Traditional Building of an Application

User Interface Layer

BV = 100%

Iteration 5

(whatever)

BV = 0%

Iteration 4

Business Logic Layer

BV = 0%

Iteration 3

Data Access

BV = 0%

Iteration 2

use

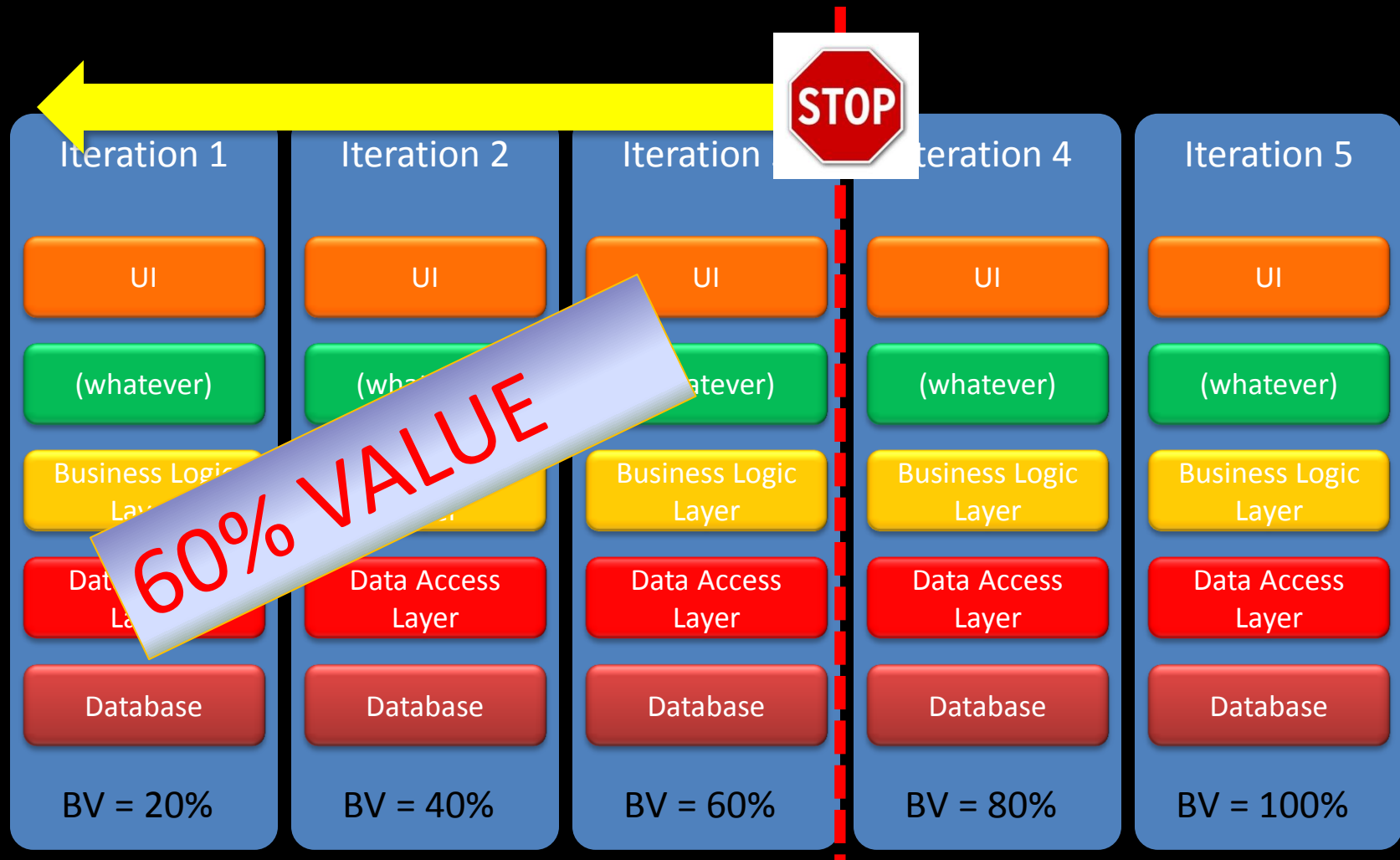
BV = 0%

Iteration 1

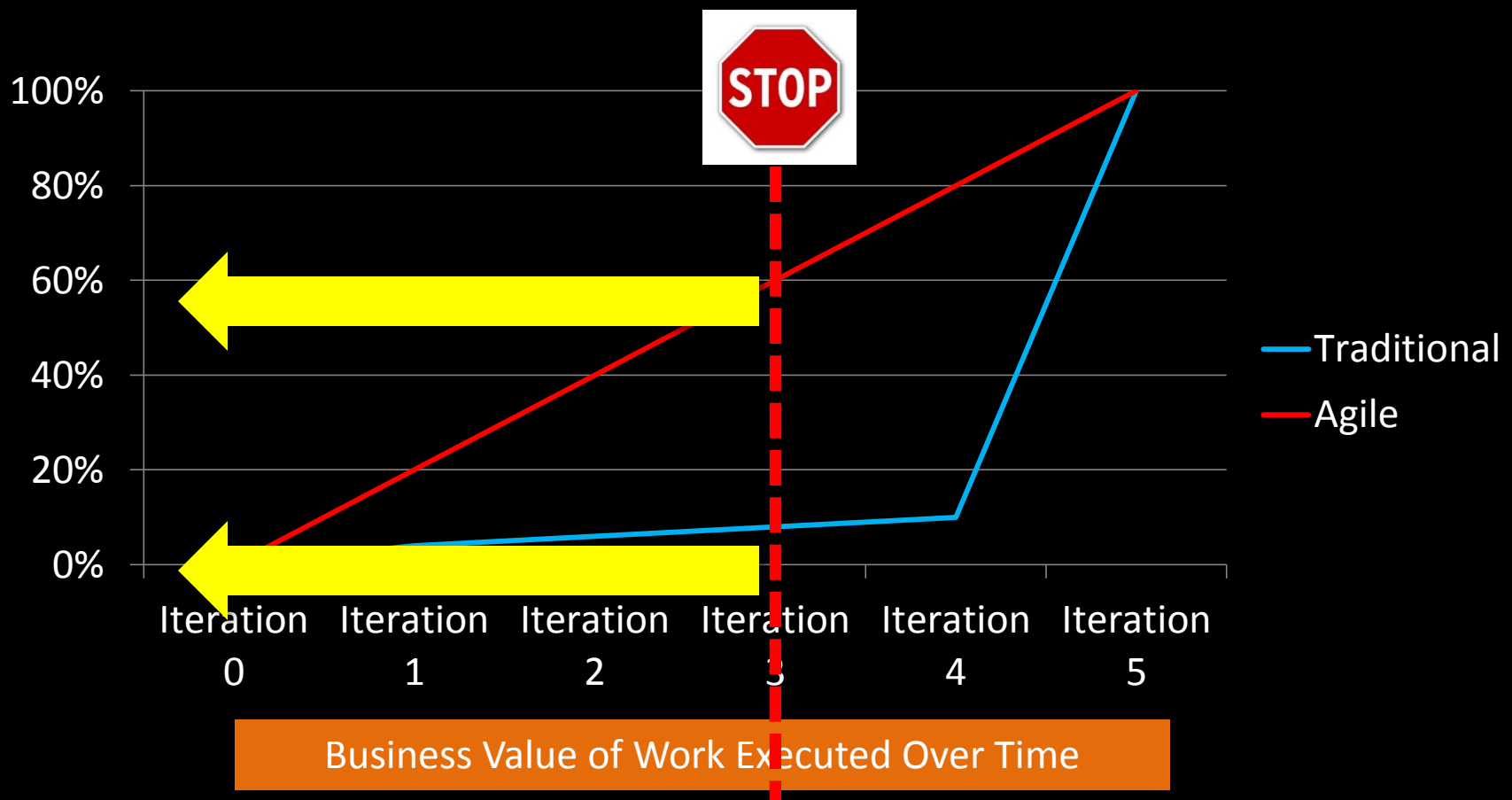
**0% VALUE**



# Agile Building of an Application



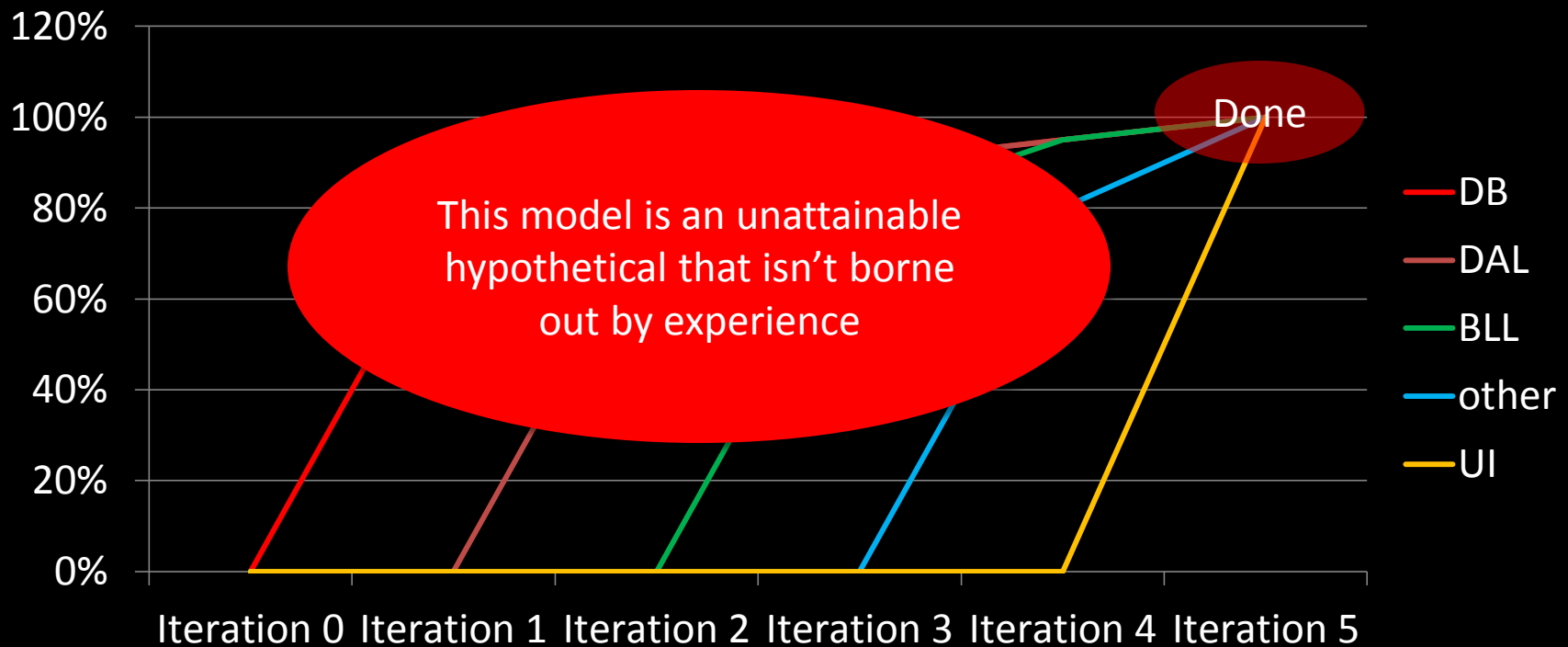
## Another way to Visualize this Relationship



## Aspects of Traditional Application Development

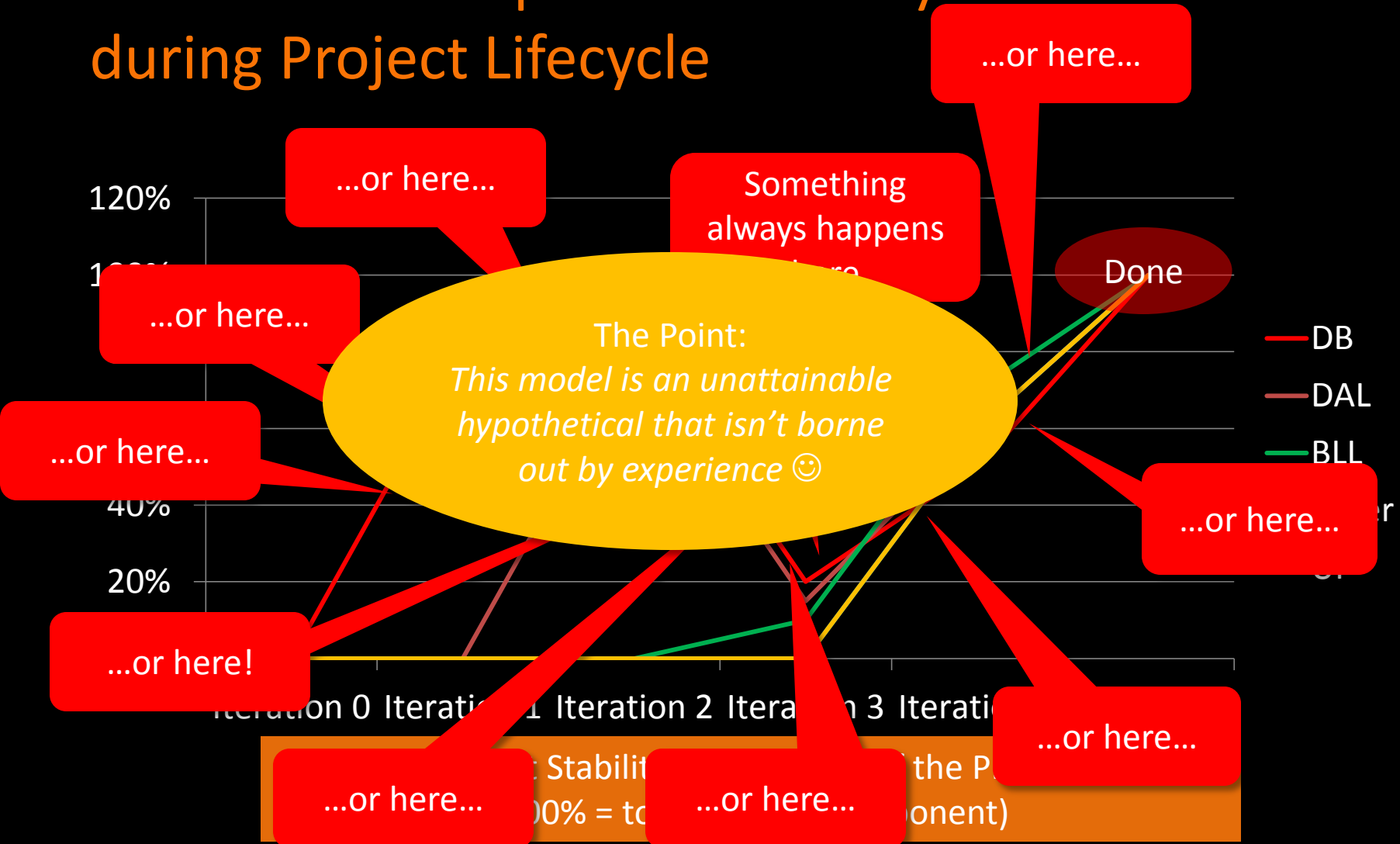
- Building successive **HORIZONTAL** layers of functionality
  - Like building a building (foundation, then walls, then roof)
- Once a layer is sufficiently ‘baked’, move on to the next layer in the stack
- After a layer is baked, its typically very costly (relatively) to change much of it except for minor tweaking
- No ‘penalty’ for tightly-coupled and non-orthogonal designs
  - Foundation layers of application are not in high-flux once ‘baked’
- BDUF approach to design and development is a necessity
  - a poor long-range design choice early-on is expensive to redress
- Treats software like hardware (once its built, its hard to change)
  - Significant change is considered something that doesn’t happen until v2.0
- Zero (nearly) business value until 100% complete

# Traditional Component Stability during Project Lifecycle



Percent Stability over the Life of the Project  
(100% = totally stable component)

# Traditional Component Stability during Project Lifecycle



# Aspects of Agile Application Development

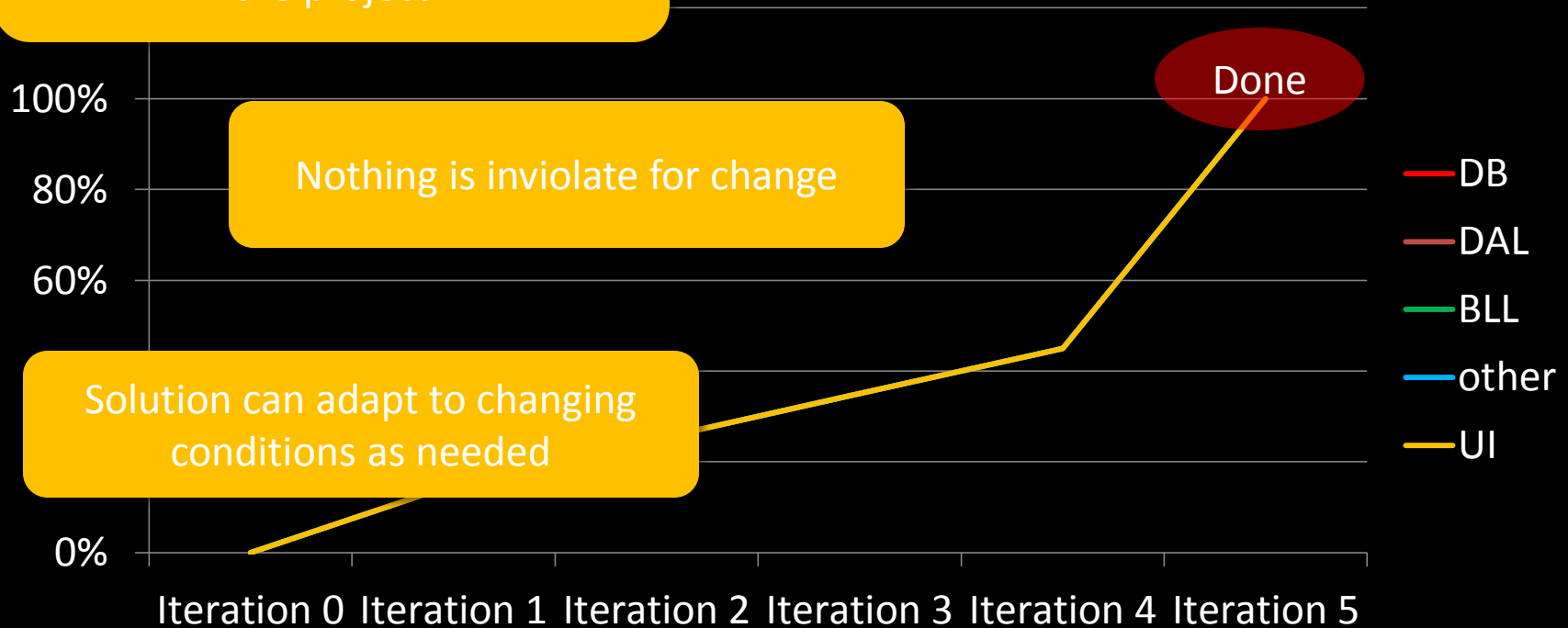
- Building successive **VERTICAL** slices of complete functionality
- All 'layers' of the application stack are fair-game for complete re-write at any time
  - As additional functionality is added (regular progression)
  - As new requirements are uncovered (irregular progression)
- *Huge* 'penalty' for tightly-coupled and non-orthogonal designs
  - This makes redesigning the layers extremely difficult/inefficient
- No BDUF needed beyond general architectural overview-level
- Business Value scales (nearly) linearly with effort expended
  - 55.227% complete = 55.227% business value delivered

# Agile Component Stability

Everything maintains a similar level of stability until the end of the project

Nothing is inviolate for change

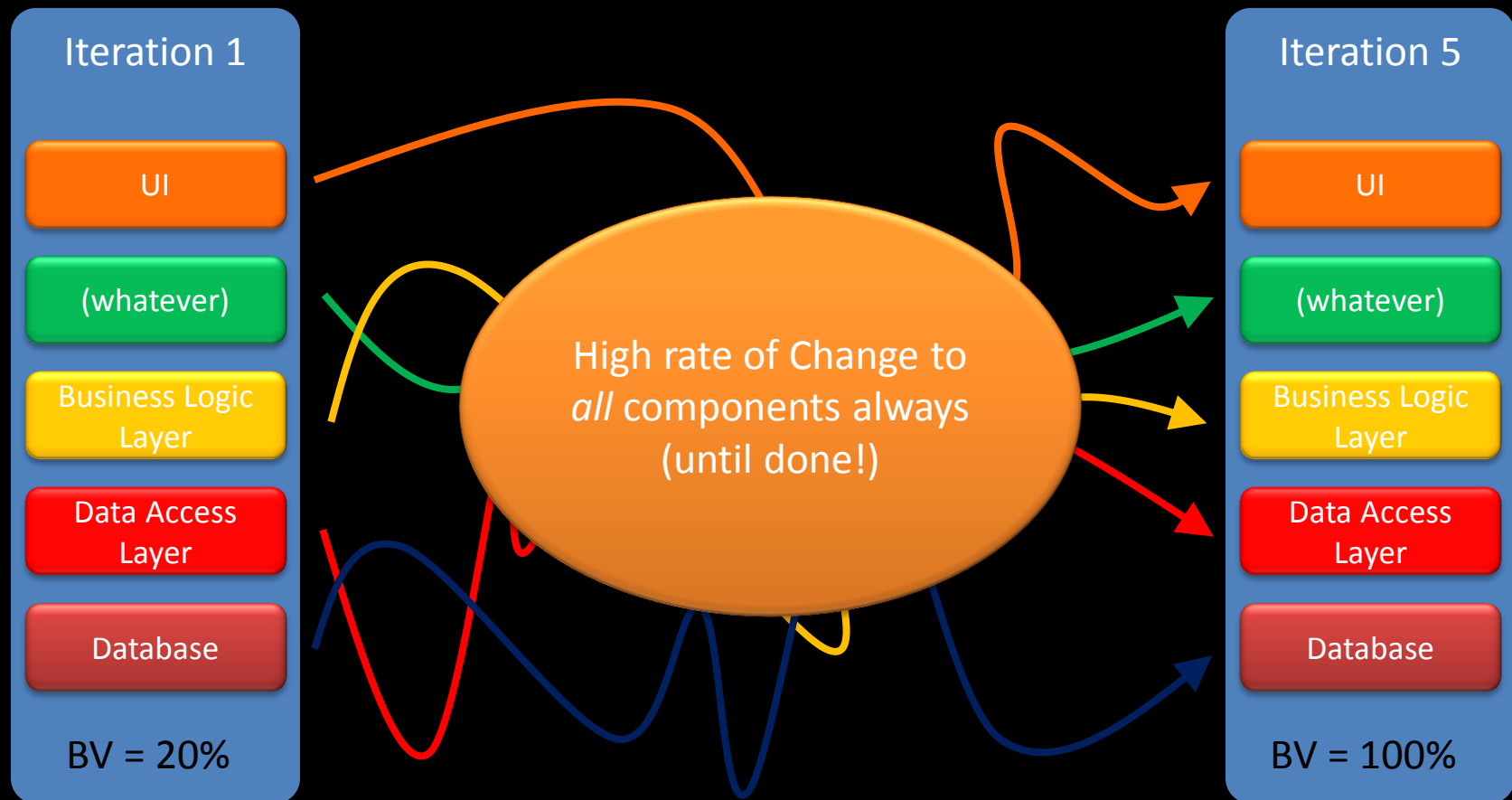
Solution can adapt to changing conditions as needed



# Evaluation of Traditional Development Methodology

- Q: if Agile offers greater flexibility, why Traditional Software Practices?
- A: Without tools and practices to manage constant change to all aspects of a solution, chaos is the result
- A Traditional approach allows you to ‘lock in’ to a high-degree of stability the ‘foundational layers’ of your solution
  - Only one component is in high-flux (under construction) at any one time
- Works (to at least some degree) when coupled with high-fidelity requirements gathering and 100% hard-locked scopes-of-work
  - Q: When was the last time you had one of those...? 😊
- Doesn’t work (at all) when stakeholders don’t know what they really want/need
  - Leads to *“Building the thing right”* instead of *“Building the right thing”*

# Why the Focus on Tests?



## Surviving the Chaos

- All components in high-flux for the entire life of the project is chaos
  - (without a means to mitigate the chaos!)
- Adhering to two core philosophical principles ensures that the chaos is survivable and manageable:
  1. ***THE DESIGN SHOULD SUPPORT EASILY MAKING SWEEPING CHANGES TO THE ENTIRE SYSTEM***
  2. ***RAPID FEEDBACK ON THE AFFECT OF ANY CHANGE ON THE ENTIRE SYSTEM***
- If I can know the moment I have made a change what the consequences of that change are going to be, then I can:
  - freely change anything at any time in response to the *business* needs of the project
  - Rapidly evaluate the consequences of that change on the rest of the system
  - Roll back the change if it fails the cost/benefit evaluation for the system as a whole

# Nearly All Agile Techniques Support Those Two Core Principles

- Adherence to core OO design principles
  - Decoupling, encapsulation, orthogonality, etc.
    - Maximizes ability to make sweeping changes to the system with minimum of 'collateral damage'
- Unit Tests
  - Rapid feedback on the impact of my own changes
    - If manual testing is req'd to validate every change, then the cost of change becomes too high to consider making it
- Continuous Integration
  - Rapid feedback on the impact of everyone else's changes
    - Duration between breaking changes and awareness of issue is zero!
- Iterations/Sprints/Intervals/Whatever
  - Stakeholder feedback on the impact of changes
    - Confirm/Verify/Validate direction of solution

# Different Solutions to The Same Problem: *Change*

- Traditional Software Development Approach
  - Assumes change is avoidable
  - Tries to manage change by sufficient pre-planning and design to avoid change altogether
    - BDUF approach
  - Treats the process of constructing software as if it's a building construction project
    - **Wrong metaphor**
- Agile Software Development Approach
  - Assumes change is inevitable and unavoidable
  - Assumes its impossible (or at least impractical) to attempt to plan-around or design-around change
  - Tries to manage change by ensuring that the software remains flexible enough to respond to the change
  - Ensures that sufficient tooling, process, and methods are in place to allow response to change within the context of an incredibly tight feedback loop

# Software Development



# Discussion / Q+A

- Thoughts
- Impressions
- Experiences
- Questions
- Challenges
- Opportunities