
LOYOLA
UNIVERSITY
CHICAGO

# COMP 313 / 413: Intermediate Object-Oriented Programming

Dr. Robert Yacobellis
Advanced Lecturer
Department of Computer Science

# Week 5 Topics

- **Test 1 – 50 minutes + 10 minute break**
- **Final Project 3 team member setup**

# Week 5 Topics

- **More Design Patterns**
  - **Decorator**, Composite, Visitor revisited
- Project 3 and Agile Development
- Android Application Development – continued Week 6
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps

# Design Pattern Classification

- **Creational patterns:** relate to the process of object creation
- **Structural patterns:** deal with composition of classes/objects
- **Behavioral patterns:** deal with interactions of classes/objects

# The Decorator Pattern (Structural)

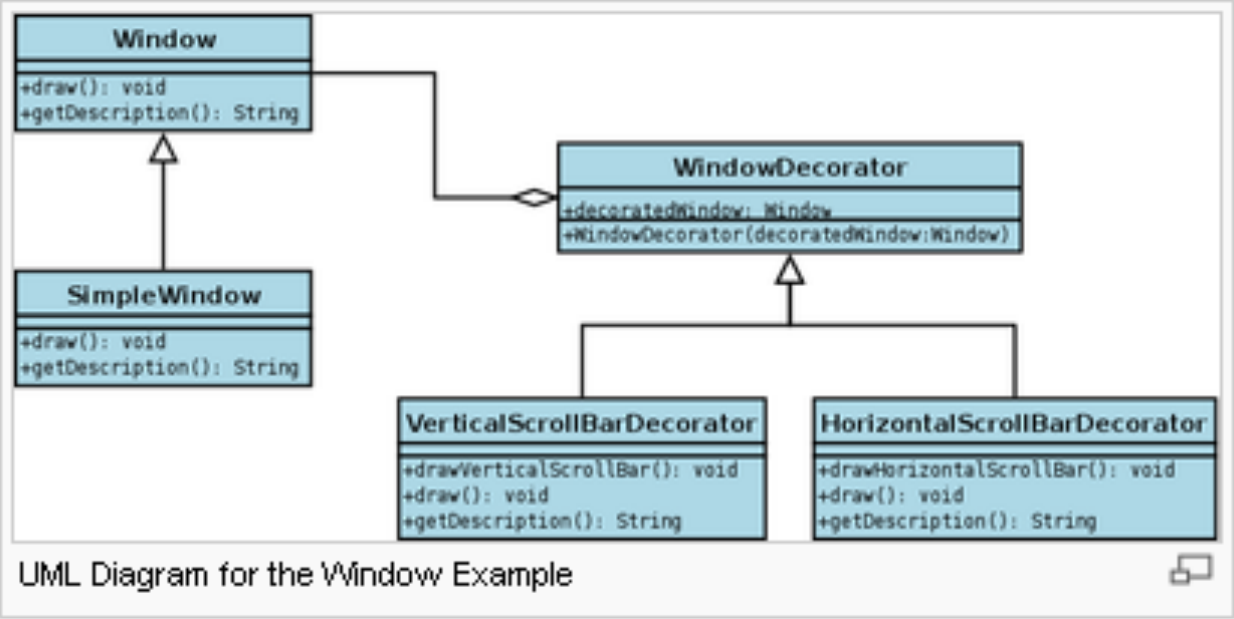
From Wikipedia, the free encyclopedia

- In object-oriented programming, the **decorator pattern** is a design pattern that **allows new/additional behavior to be added to an existing object dynamically**. The decorator pattern can be used to make it possible to extend (decorate) the functionality of a certain object at runtime, independently of other instances of the same class, provided some groundwork is done at design time. **This is achieved by designing a new *decorator* class that wraps the original class.** This wrapping could be achieved by the following sequence of steps:
  1. Subclass the original "Component" class into a "Decorator" class (see UML diagram);
  2. In the Decorator class, add a Component pointer as a field (composition or aggregation);
  3. Pass a Component to the Decorator constructor to initialize the Component pointer;
  4. In the Decorator class, redirect all "Component" methods to the "Component" pointer; and
  5. In the Decorator class, override any Component method(s) whose behavior needs to be modified.
- **This pattern is designed so that multiple decorators can be stacked on top of each other, each time adding new functionality to the overridden method(s). (This is like pipelining in Unix.)**
- **The decorator pattern is an alternative to subclassing.** Subclassing adds behavior at compile time, and the change affects all instances of the original class; decorating can provide new behavior at runtime for individual objects. This difference becomes most important when there are several *independent* ways of extending functionality. In some object-oriented programming languages, classes cannot be created at runtime, and it is typically not possible to predict what combinations of extensions will be needed at design time. This would mean that a new class would have to be made for every possible combination. By contrast, decorators are objects, created at runtime, and can be combined on a per-use basis.
- **An example of the decorator pattern is the Java I/O Streams implementation.**

# DPiJ Extension Patterns (Ch. 26) and the Decorator Pattern (Ch. 27)

- Remember that DPiJ says Extension patterns like Decorator allow you to add new behavior to existing or legacy classes
- In particular, Decorator allows you to compose an object's behavior dynamically by "wrapping" the object in one or more layers of compatible class objects
  - In this case the wrapping object's constructor takes a wrapped object as a parameter; used with Java I/O streams or functions
  - Both the wrapped and the wrapping objects "act as" (provide instantiations for) a common superclass
  - Because of this, wrapped objects can be further wrapped
  - The wrapping object(s) can provide complex processing that the superclass does not need to define or even know about
- Patterns related to Decorator: State, Strategy, Interpreter, Composite, and Proxy (DPiJ page 303)

# Decorator Pattern Windowing System UML Example



UML Diagram for the Window Example

- The Decorator must be a subclass of the object to be decorated

http://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

→ Bob Tarr Decorator slides 2-19

→ APPP Chapter 35, pp 560-565

# Week 5 Topics

- **More Design Patterns**
  - Decorator, **Composite**, Visitor revisited
- Project 3 and Agile Development
- Android Application Development – continued Week 6
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps

# Composite Design Pattern (Structural)

From Wikipedia, the free encyclopedia

- In computer science, the **composite pattern** is a *partitioning* design pattern. **Composite allows a group of objects to be treated in the same way as a single instance of an object.** The intent of composite is to "compose" objects into tree structures to represent part-whole hierarchies. **Composite lets clients treat individual objects and compositions uniformly.**
- When dealing with tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone. The solution is an interface that allows treating complex and primitive objects uniformly. In object-oriented programming, a composite is an object (e.g., a shape) designed as a composition of one-or-more similar objects (other kinds of shapes/geometries), all exhibiting similar functionality. This is known as a "has-a" relationship between objects.
- The key concept is that you can manipulate a single instance of the object just as you would manipulate a group of them. The operations you can perform on all the composite objects often have a least common denominator relationship.
  - For example, if defining a system to portray grouped shapes on a screen, it would be useful to define resizing a group of shapes to have the same effect (in some sense) as resizing a single shape.

# Composite Design Pattern Structure

**Component**

- is **the abstraction for all components, including composite ones**
- **declares the interface** for objects in the composition
- implements default behavior for the interface common to all classes, as appropriate
- declares an interface for accessing and managing its child components
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate

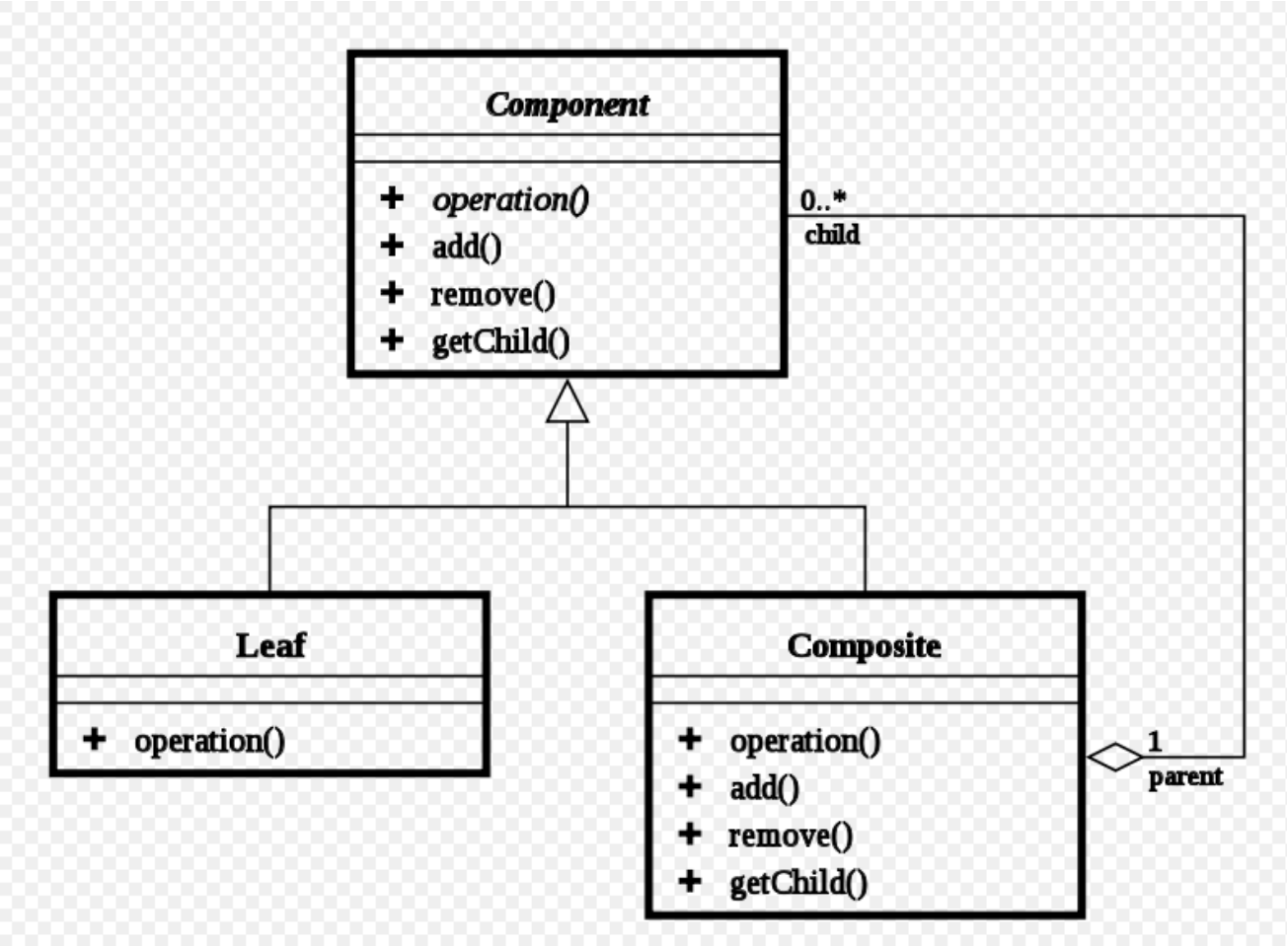
**Leaf**

- represents leaf objects in the composition
- implements all Component methods

**Composite**

- represents a composite Component (component having children)
- implements methods to manipulate children
- implements all Component methods, generally by delegating them to its children

# Composite UML Class Diagram




http://www.tutorialspoint.com/design_pattern/composite_pattern.htm

→ Bob Tarr Composite Slides 1-16

→ APPP Chapter 31

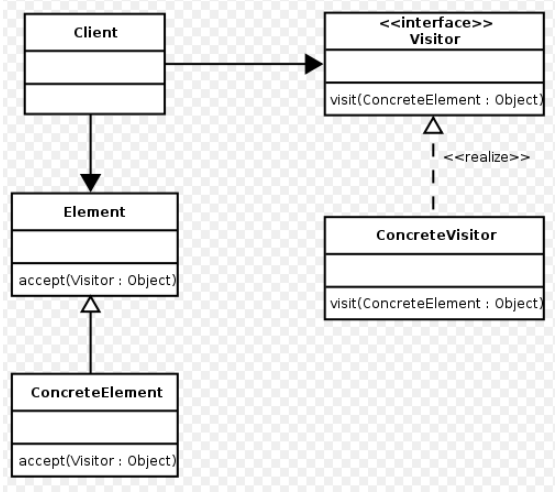
# Week 5 Topics

- **More Design Patterns**
  - Decorator, Composite, **Visitor revisited**
- Project 3 and Agile Development
- Android Application Development – continued Week 6
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps

# Visitor Design Pattern (Behavioral)

From Wikipedia, the free encyclopedia

- In object-oriented programming and software engineering, **the visitor design pattern is a way of separating an algorithm from an object structure upon which it operates.** A practical result of this separation is **the ability to add new operations to existing object structures without modifying those structures**. Thus, using the visitor pattern helps conformance with the open/closed principle.
- In essence, **the visitor allows one to add new virtual functions (abstract methods in Java) to a family of classes without modifying the classes themselves**; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.
- While powerful, the visitor pattern is more limited than conventional virtual functions. It is not possible to create visitors for objects in Java without adding a small callback method inside each class and the callback method in each of the classes is not inheritable.



# The Visitor Pattern Together with Decorator and Composite

- The process of traversing a composite structure can often be handled more effectively by giving each of the composite and leaf nodes an accept() method (via an interface that they all implement), and having one or more visitors "walk" the structure to do things
  - This is how the various visitors in Project 3 process composite Shape components that are drawn on an Android Canvas
- In addition, a visitor can "traverse" a decorated object by extracting and possibly using any extra decorated information, and also extracting and "walking" the wrapped Shape object to as many levels as necessary
  - This is how decorated Project 3 Shapes are visited/processed

# The Visitor Pattern "visit()" Method

- I mentioned last time that each class to be visited must have an accept() method that accepts any object that implements the common Visitor interface
- In turn that accept(Visitor v) method commonly turns around and calls v.visit(this)
  - The method chosen is based on the type of "this"
  - APPP Chapter 35, pp 543-559, demonstrates this approach
- However, an alternative, equivalent implementation for accept(Visitor v) is for it to invoke a visit method like so: v.on*typename*(this); where *typename* is the name of the class being visited, eg, v.onCircle(this);
  - This is the approach taken in Project 3's accept() methods, and it is also the one demonstrated in the Bob Tarr slides

# Week 5 Topics

- More Design Patterns
  - Decorator, Composite, Visitor revisited
- **Project 3 and Agile Development**
- Android Application Development – continued Week 6
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps

# Project 3 Overview

- You'll work in 2-person Project 3 teams to implement the Visitor, Composite, and Decorator design patterns in order to draw diagrams on an Android Canvas
  - The initial, base implementation does not run in Android
- I'll review the following areas in Android Studio:
  - The Visitor<Result> generic interface
  - TODOs plus Outline and Stroke (decorators!), and Polygon accepting classes; Point + Location (a decorator); other "model" classes; Draw, Size, and Bounding Box Visitor classes
  - The Android Canvas and Paint classes: online documentation
  - Unit tests run using Gradle; the Fixtures class; Mockito
- If time, I'll briefly look at Agile development – APPP Chapters 1-3

# Project 3 Teams -TBD

# What is Agile?

- *"Ultimately, Agility is about:*
  - *- Embracing change rather than attempting to resist it*
  - *- Focusing on talent and skills of individuals and teams"*
- *-- Jim Highsmith, Cutter Consortium*

- The Agile Manifesto (2001) establishes a set of values that are *people-centric* and *results-driven*; it emphasizes:
  - ***"Individuals and interactions** over processes and tools*
    - *Working software over comprehensive documentation*
    - *Responding to change over following the plan*
    - *Customer Collaboration over contract negotiation*
  - ***That is, while there is value in the items on the right, we value the items on the left more."***
- ***-- Manifesto for Agile Software Development***

*(www.agilemanifesto.org)*

# The Essence of Agile

- Iterative Lifecycle
  - Rapid feedback & learning in short cycles
- Collaborative Teaming

  - Teams produce better results than individuals
  - Stakeholder involvement yields better decisions
- Development Continuously Validates Quality
  - Automated regression testing
  - Continuous integration
  - Test-Driven Development (TDD)

# Agile Principles – Guiding Themes

- Face-to-face conversations are most effective in communicating
  - Written word / Models leave too much open to interpretation
  - Co-location is critical for team coherence, quality, & productivity
- Working software is the primary measure of progress
  - Nothing is complete until we have working software
- Self-Organizing Teams
  - The team actively participates in managing sprints (time-boxed iterations)
  - The team develops low level plans to achieve the goals of each sprint
- Just Enough Documentation
  - Just enough to implement and maintain the product
- Collective Ownership
  - Everyone is responsible for the finished product
- Welcome change (even changing requirements)
  - Our project is constantly changing
  - Optimize the project around that environment

# Where Does Agile Fit?

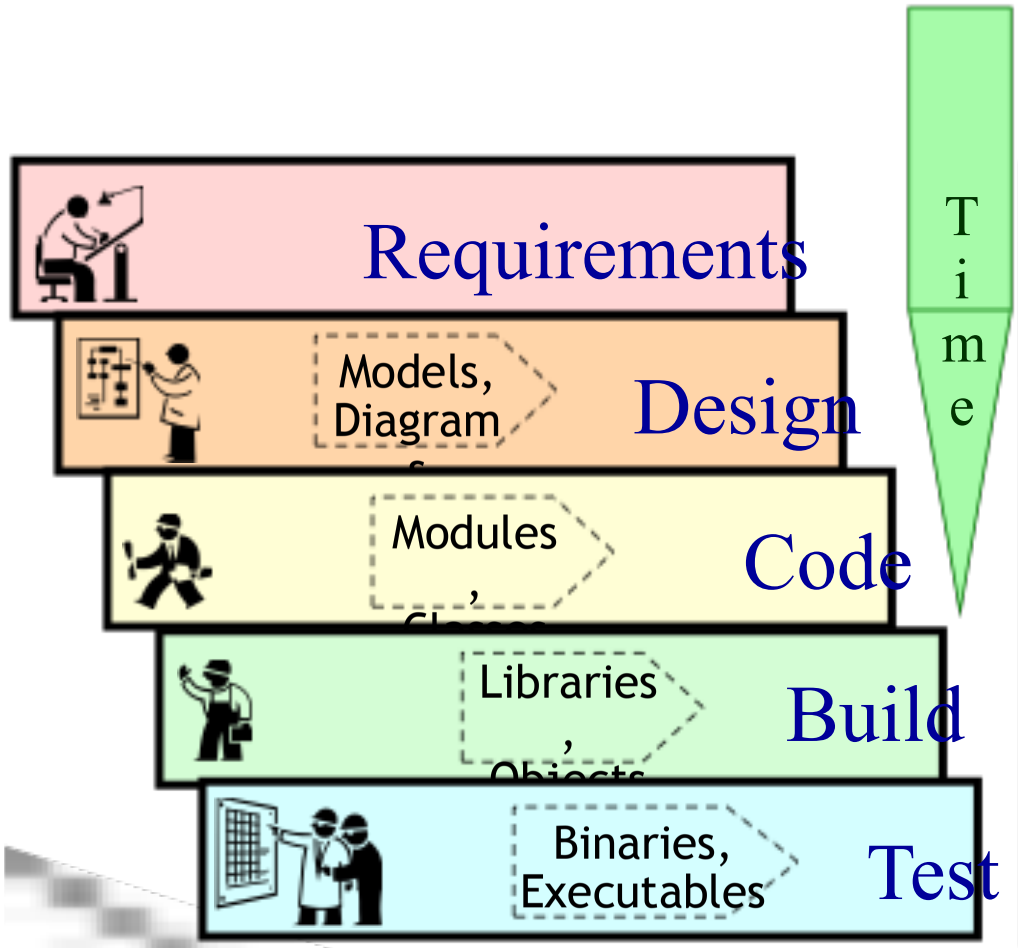
- *Today:*
  - Often used for Software Development
- *Tomorrow:*
  - Systems Development
  - Program Management
  - Hardware Development

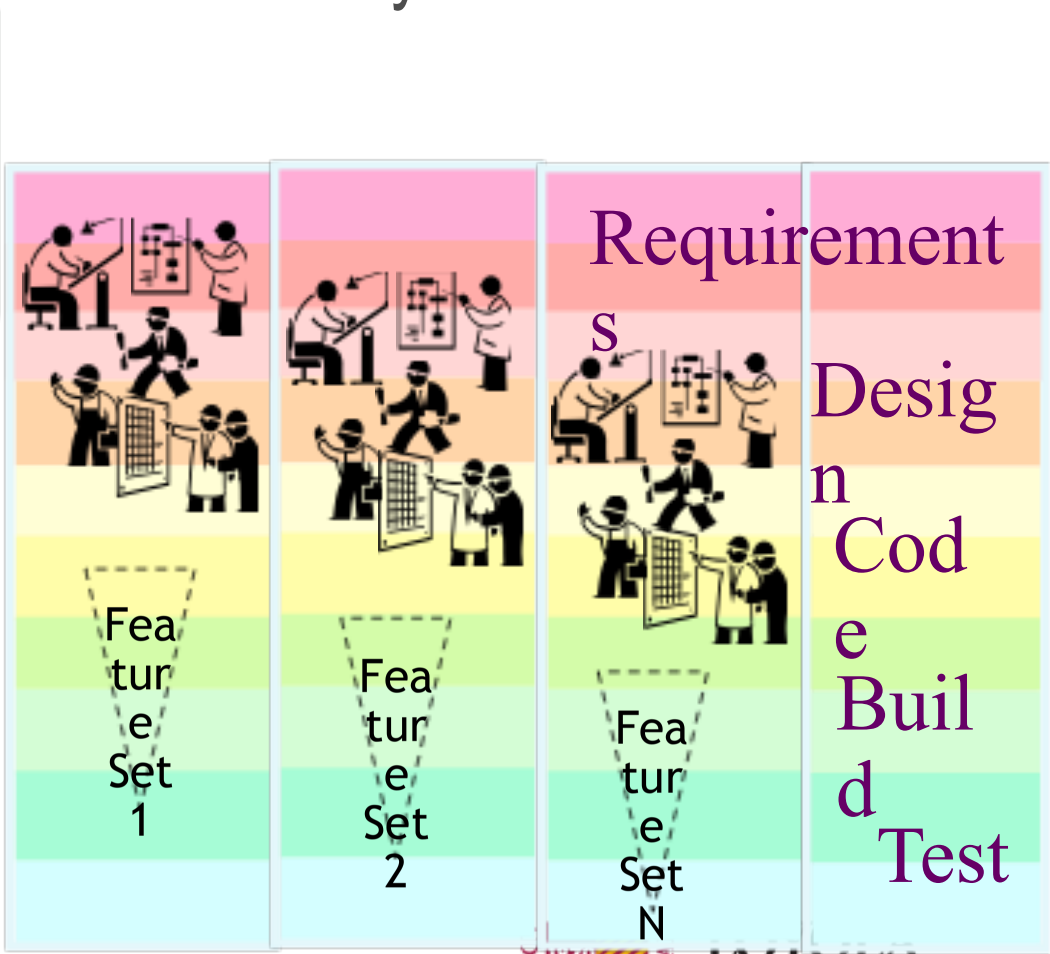
## Waterfall Lifecycle

Breadth-First Delivery
Phase-Based Development
End-of-Phase Handoffs

## Iterative and Incremental Development Lifecycle

Depth-First Delivery
Feature Set-Based Development
Full-Lifecycle Collaboration






LOYOLA UNIVERSITY CHICAGO

# Incremental Development vs. Iterative Development

Incrementa
l

1 

2 

3 

Iterative


1 

2 

3 

# Key Agile Practices To Deploy

These management and engineering practices work well with existing software process definitions:

- Iterative Development
- Test Driven Development
- Continuous Integration
- Agile Project Management
- Pair (or Paired) Development
- Retrospectives
- Customer Proxy
- Refactoring
- Daily Stand-up
- Automated Testing

# Agile Practices

1. Iterative/Incremental Development
   - Between 2 and 6 weeks in length (time-boxed)
2. Retrospectives *(like in-process post mortems)*
   - Looking at the past to improve the future
   - Take advantage of cycles of learning
3. Pair Development *(two developers work on one task)*
   - Two heads are better than one
4. Test Driven Development *(write tests; code to pass them)*
   - Code a little, test a little…
5. Automated Testing
   - Collect all test cases into push-button test suites

# Agile Practices [continued]

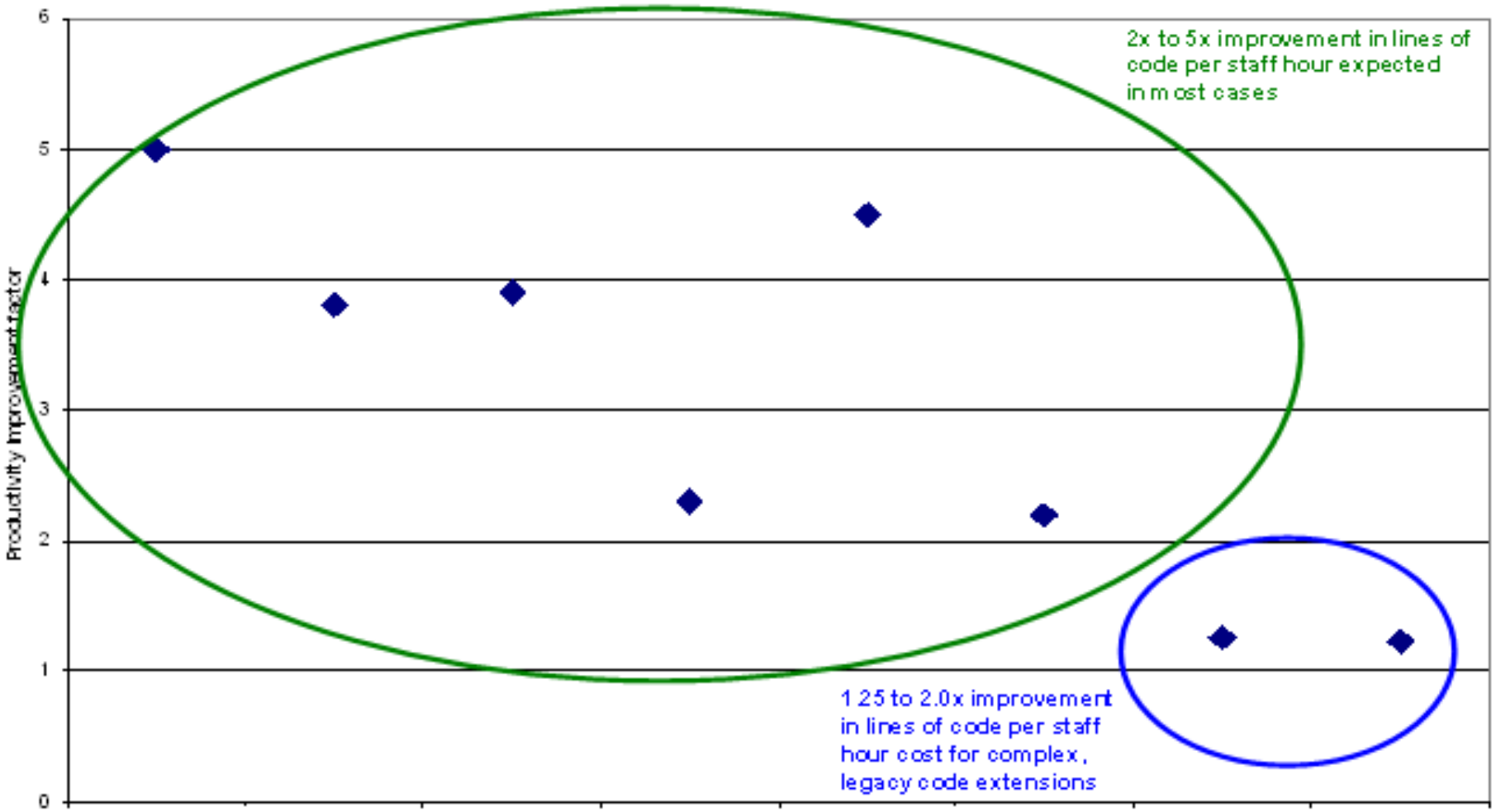
1. Constant Integration
   - Merge+build+test new changes at least once a day
2. Daily Standup
   - Quickly share information and address risks
3. Refactoring
   - “Improving the design of existing code”
4. Customer Proxy
   - Representing the voice of the customer
5. Agile Project Management
   - Managing iterations, velocity (team capability), buffer time

# Reported Results from Agile Teams

- Less excessive overtime toward the end of a project
  - Quality improvement implies fewer issues in test
  - Constant focus on time-boxed work product prevents unnoticed slippage/issues early in the program
  - Focus on work progress instead of process adherence helps track progress to value-added deliverables
- More empowerment of team members
  - How to run the project
  - Taking responsibility for the quality of their own product
- Managers can focus on their own highest value
  - Removing roadblocks
  - Setting vision and priorities

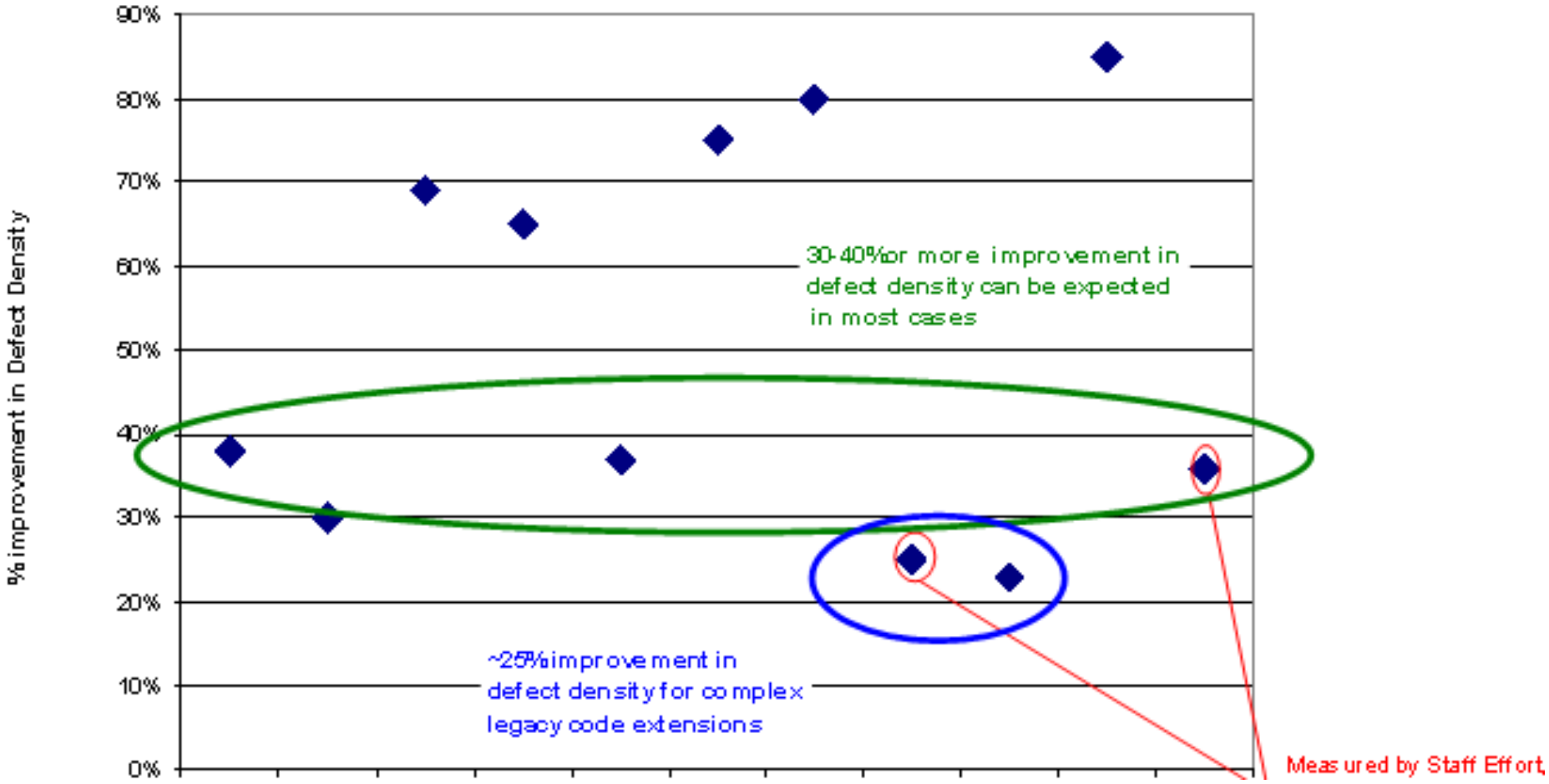
# Agile – Demonstrated Productivity Increases

## 1.25x productivity worst case, 2x or better normally



# Agile – Demonstrated Quality* Improvements

## 25% improvement worst case, 30-40% normally




** Quality as measured by Defect Density, defects per thousand lines of code (KLOC)*

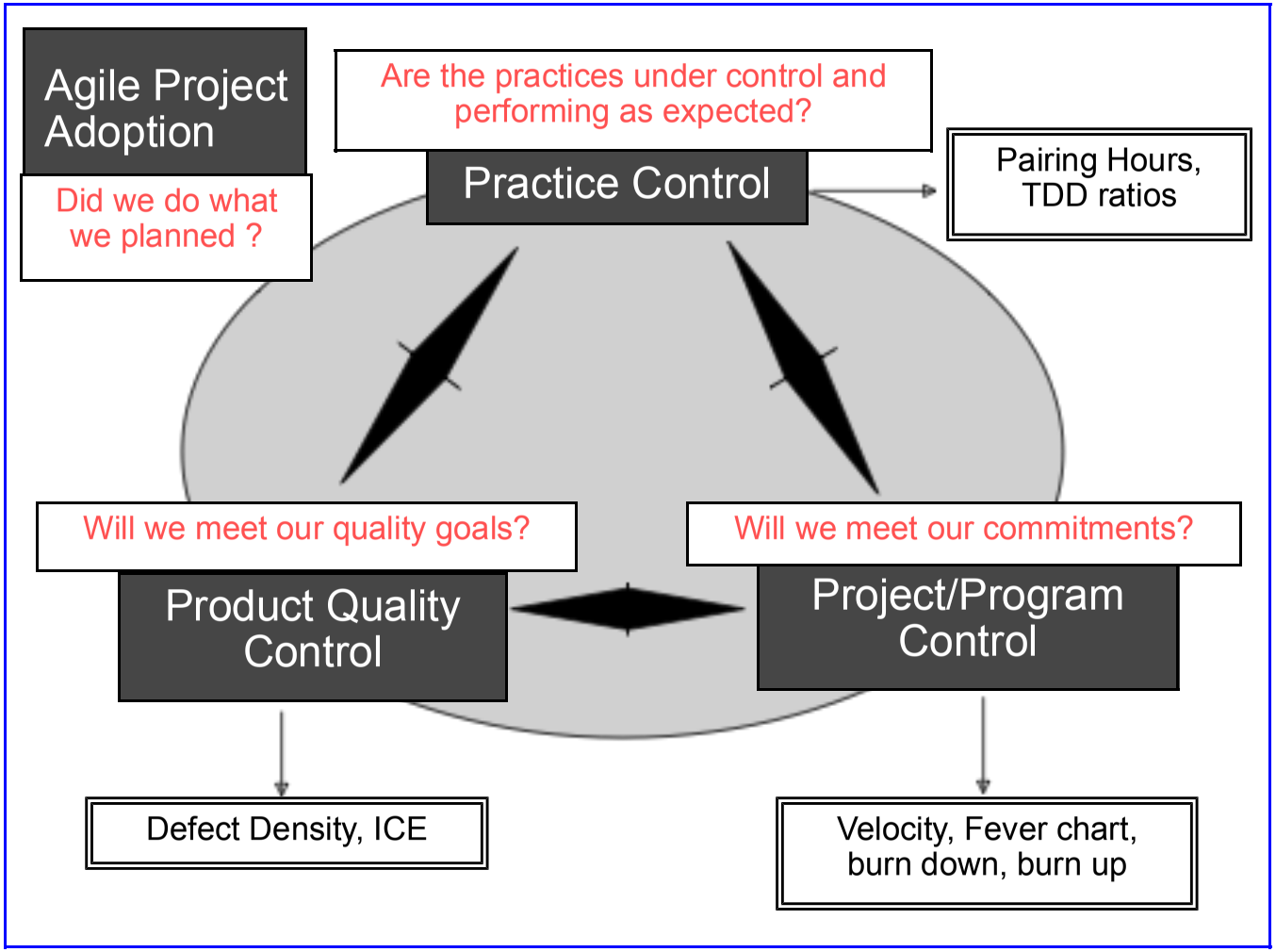
# Week 5 Topics

- More Design Patterns
  - Decorator, Composite, Visitor revisited
- Project 3 and Agile Development
- Android Application Development – continued Week 6
  - Running some simple apps
  - Android framework & activity life cycle
  - More complex apps

# Resources Needed for Agile Deployment

- Agile Deployment Team responsibilities
  - Develop Agile Infrastructure + metrics
  - Coaching of Agile Projects
    - Help tailor Agile practices to any existing development environment
    - Help apply Agile practices through training and mentoring
  - Follow up on Agile data/demonstrated benefits
  - Provide Agile "project governance" (oversight – is it working?)
- Required Business support (eg, within industry)
  - Agile deployment Champion
    - Project leader to run day to day development activities
    - Credibility in the business
    - Comes from the development team
    - Set up to be a future Agile coach
  - Agile deployment should not impact project resource needs
    - Assume no significant change in resource cost
    - Allocation/distribution of the resources will change

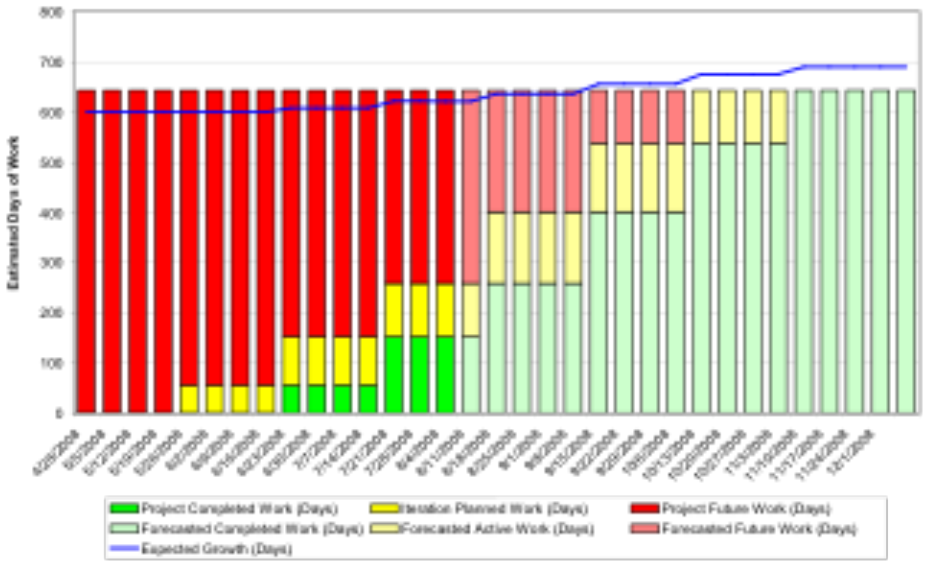
# Monitoring and Controlling Agile Projects



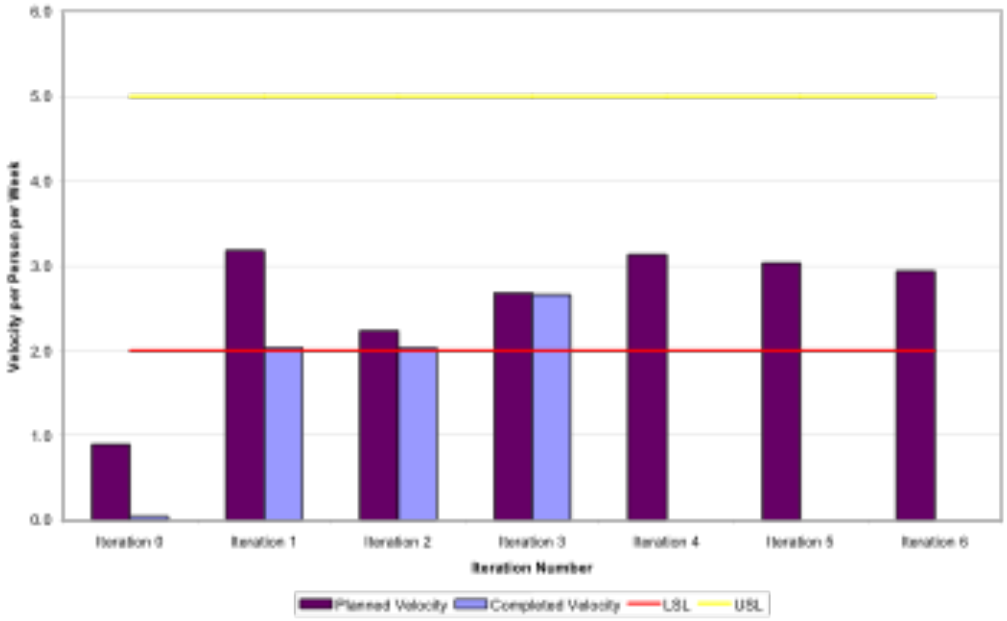

Metrics for tracking and monitoring Agile impact

# Agile Project Management Charts

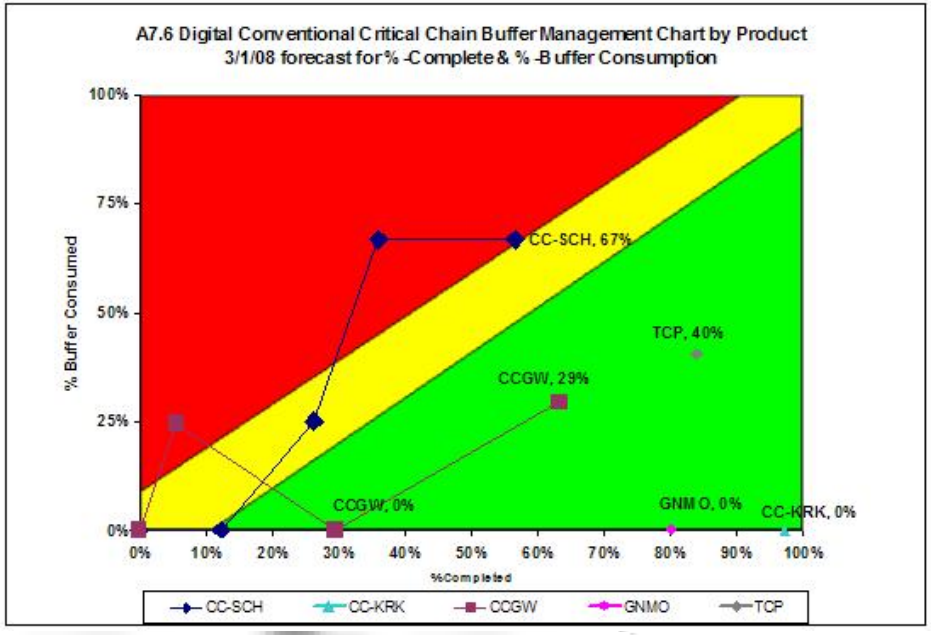
## Burn-up Chart



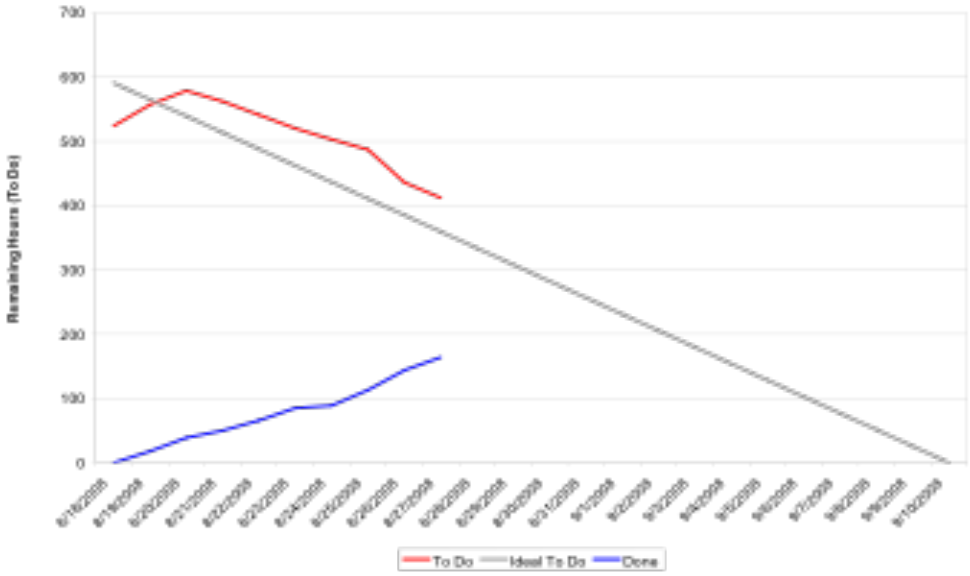

## Velocity chart



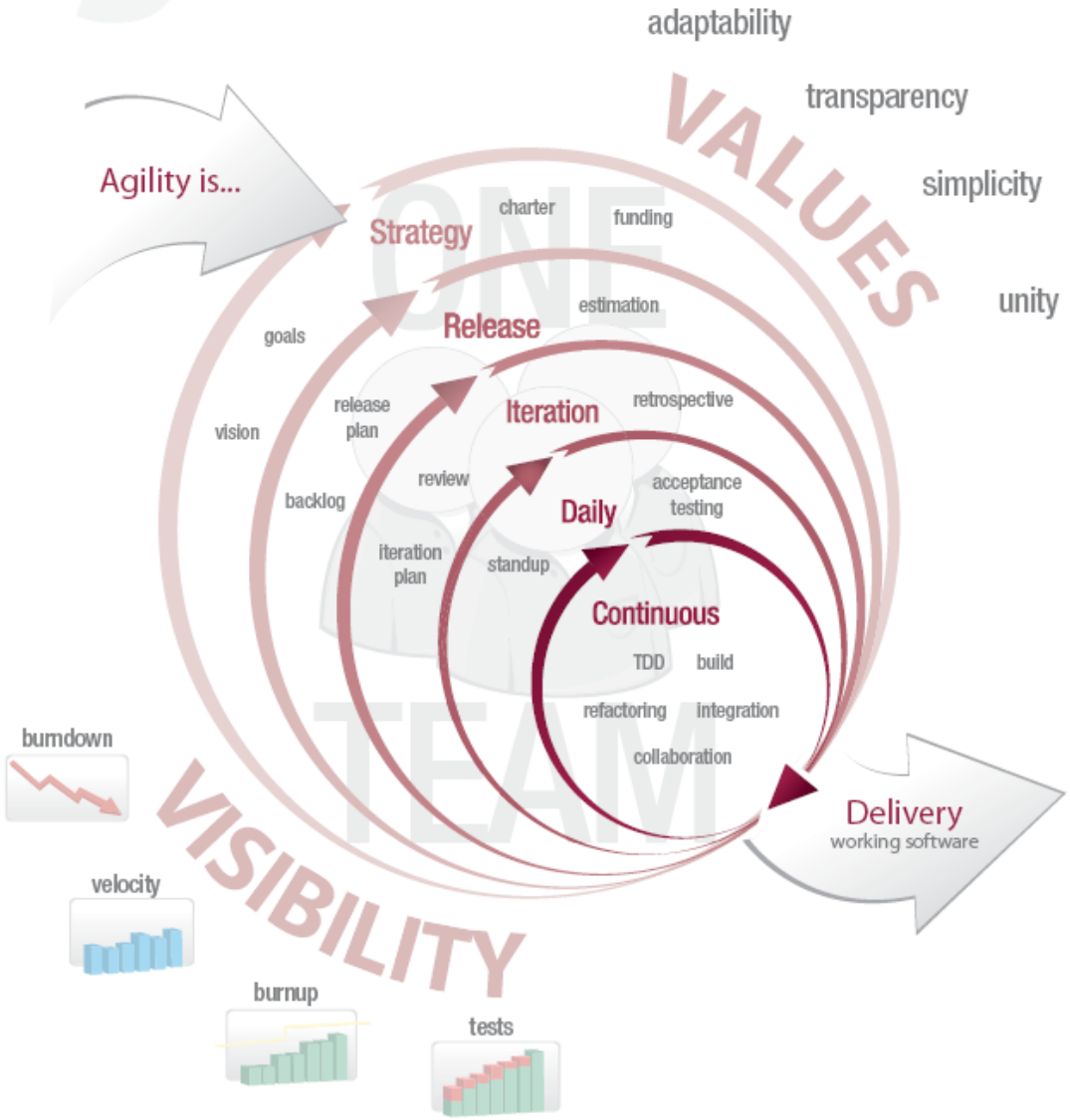

## Buffer Consumption Chart




## Burndown Chart per Iteration



# Agile Development




# Accelerate Success

# Agile Practice Details (1 of 2)

- Iterative Development
  - Iterations measured in weeks
  - Integrated, running, and tested software at the end of each iteration
  - Ends with a retrospective to improve the next iteration
- Test Driven Development
  - Tests are written before design/code
  - Design/Code are just enough to pass the tests
  - When the tests pass, refactor as needed
- Continuous Integration
  - Code is integrated, built and tested continually
  - No “Big Bang”
- Agile Project Management
  - Time-Boxing (delivering working software in a short, 4-week fixed time-box)
  - Multi-Level Planning (Program/Release, Project/Feature, Iteration, Daily/Task)
  - Backlog Management & Prioritization
  - Velocity Tracking
  - Buffer Management (from Critical Chain Project Management - CCPM)
  - Buffer Iterations
- Pair (Paired) Development
  - Collocated developers work in pairs on all development activities
  - Self-checking, self-correcting

# Agile Practice Details (2 of 2)

- Retrospectives
  - Regular reviews at the end of each iteration
  - Opportunity for immediate feedback and quick mid-course correction
  - The team reflects on how to become more effective adjusts its behaviour
  - Emphasis in on taking action in the next iteration
- Customer Proxy
  - Customer or customer proxy is part of the team
  - On-going, intentional involvement throughout the project
  - Clarifies requirements/user stories
  - Helps prioritize backlog based on business value
- Refactoring
  - Changes to code that do not change external behaviour
  - Simplify the design/code
  - Enables incremental/emergent design
- Daily Stand-up (brief daily meeting for the whole team)
  - What was accomplished
  - What roadblocks are preventing progress
  - What is planned for today

# Agile Manifesto Details (1 of 2)

*"The agile methodology movement is not anti-methodology; in fact, many of us want to restore credibility to the word. We also want to restore a balance: We embrace modeling, but not merely to file some diagram in a dusty corporate repository. We embrace documentation, but not to waste reams of paper in never-maintained and rarely-used tomes. We plan, but recognize the limits of planning in a turbulent environment. Those who brand proponents of XP, SCRUM or any of the other agile methodologies as "hackers" are ignorant of both the methodologies and the original definition of the term (a "hacker" was first defined as a programmer who enjoys solving complex programming problems, rather than someone who practices ad hoc development or destruction)."*

**The Agile Manifesto: Principles**

**1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.**

**2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.**

*"The growing unpredictability of the future is one of the most challenging aspects of the new economy. Turbulence-in both business and technology-causes change, which can be viewed either as a threat to be guarded against or as an opportunity to be embraced."*

**3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.**

*"However, remember that deliver is not the same as release. The business people may have valid reasons for not putting code into production every couple of weeks. We've seen projects that haven't achieved releasable functionality for a year or more. But that doesn't exempt them from the rapid cycle of internal deliveries that allows everyone to evaluate and learn from the growing product."*

**4. Business people and developers work together daily throughout the project.**

*"For a start, we don't expect a detailed set of requirements to be signed off at the beginning of the project; rather, we see a high-level view of requirements that is subject to frequent change. Clearly, this is not enough to design and code, so the gap is closed with frequent interaction between the business people and the developers. The frequency of this contact often surprises people. We put "daily" in the principle to emphasize the software customer's continuing commitment to actively take part in, and indeed take joint responsibility for, the software project."*

**5. Build projects around motivated individuals, give them the environment and support they need and trust them to get the job done.**

# Agile Manifesto Details (2 of 2)

**6. The most efficient and effective method of conveying information with and within a development team is face-to-face conversation.**

*"Inevitably, when discussing agile methodologies, the topic of documentation arises. Our opponents appear apoplectic at times, deriding our "lack" of documentation. It's enough to make us scream, "the issue is not documentation-the issue is understanding!" Yes, physical documentation has heft and substance, but the real measure of success is abstract: Will the people involved gain the understanding they need? Many of us are writers, but despite our awards and book sales, we know that writing is a difficult and inefficient communication medium. We use it because we have to, but most project teams can and should use more direct communication techniques."*

**7. Working software is the primary measure of progress.**

*"Too often, we've seen project teams who don't realize they're in trouble until a short time before delivery. They did the requirements on time, the design on time, maybe even the code on time, but testing and integration took much longer than they thought. We favor iterative development primarily because it provides milestones that can't be fudged, which imparts an accurate measure of the progress and a deeper understanding of the risks involved in any given project."*

**8. Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.**

*"Our industry is characterized by long nights and weekends, during which people try to undo the errors of unresponsive planning. Ironically, these long hours don't actually lead to greater productivity."*

*"Agility relies upon people who are alert and creative, and can maintain that alertness and creativity for the full length of a software development project. Sustainable development means finding a working pace (40 or so hours a week) that the team can sustain over time and remain healthy."*

**9. Continuous attention to technical excellence and good design enhances agility.**

**10. Simplicity-the art of maximizing the amount of work not done-is essential.**

**11. The best architectures, requirements and designs emerge from self-organizing teams.**

**12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.**

**Source: http://www.sdmagazine.com/documents/s=844/sdm0108a/0108a.htm**