LOYOLA
UNIVERSITY
CHICAGO

COMP 413: Intermediate Object-Oriented Programming

Dr. Robert Yacobellis
Advanced Lecturer
Department of Computer Science

Announcements & Reminders

- Any remaining questions about Project 3?
  - It's due this Friday, October 21
- Reminders
  - Quiz 2 on October 25 (2 SE Radio Podcasts)
    - # 167, Unit Testing, and #46, Refactoring
    - You will be allowed to bring a page of your own notes to the Quiz
  - Test 2 on November 1 (study guide to follow)

Week 8 Topics

- **Principles of OO Design – SOLID (APPP Chapters 8-12)**
  - **S – Single Responsibility Principle**
  - **O - Open Closed Principle**
  - **L - Liskov Substitution Principle**
  - **I - Interface Segregation Principle**
  - **D - Dependency Inversion Principle**
- Agile Development Overview
- More Design Patterns (as time permits)
- Android Examples (only if time)

Principles of OO Design – SOLID

Jenga Game

The Single Responsibility Principle

- **Every object should have a single responsibility, and all of its services should be aligned with that. A "responsibility" is defined as a "reason to change."**

Split classes, use abstraction

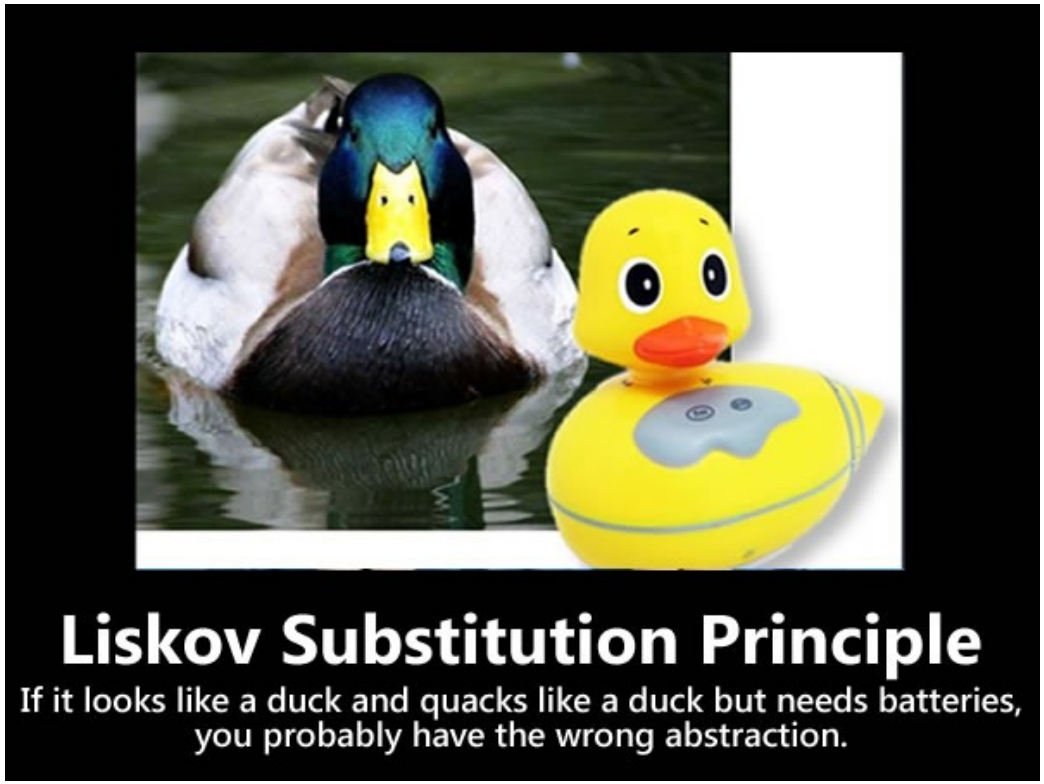
The Open-Closed Principle

- **Software entities should be <u>open for extension but closed for modification</u> – they should be extensible, but extension should not require changing code.**

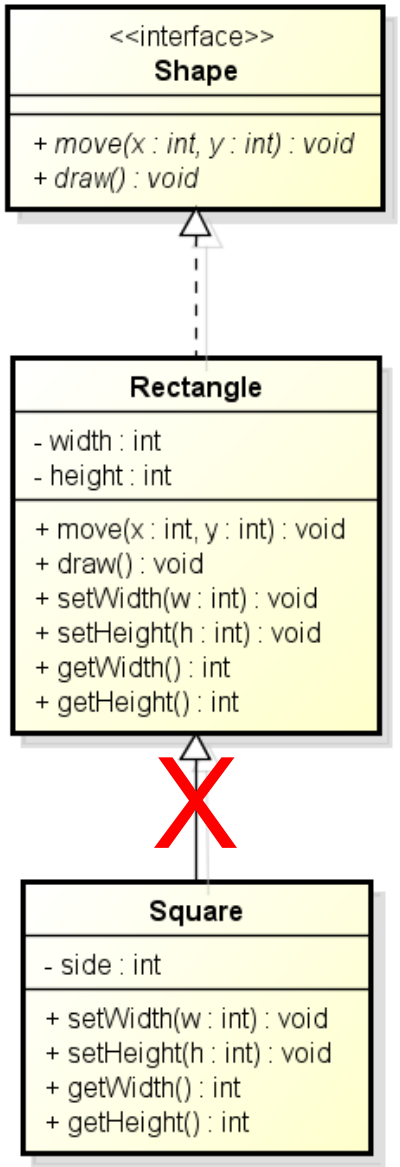
Use interfaces, Visitors, etc.

The Liskov Substitution Principle

- **Subclasses should be <u>exactly substitutable</u> for the classes from which they were derived (they should implement the same "contract" as their base class).**



The Liskov Substitution Principle



A square is <u>not</u> a rectangle!

The Interface Segregation Principle

- **<u>Clients should not be forced to depend on methods they don't use</u> – break members down into groups, and expose the groups as separate interfaces.**

The Collection Interface Breaks ISP (it may also break SRP)

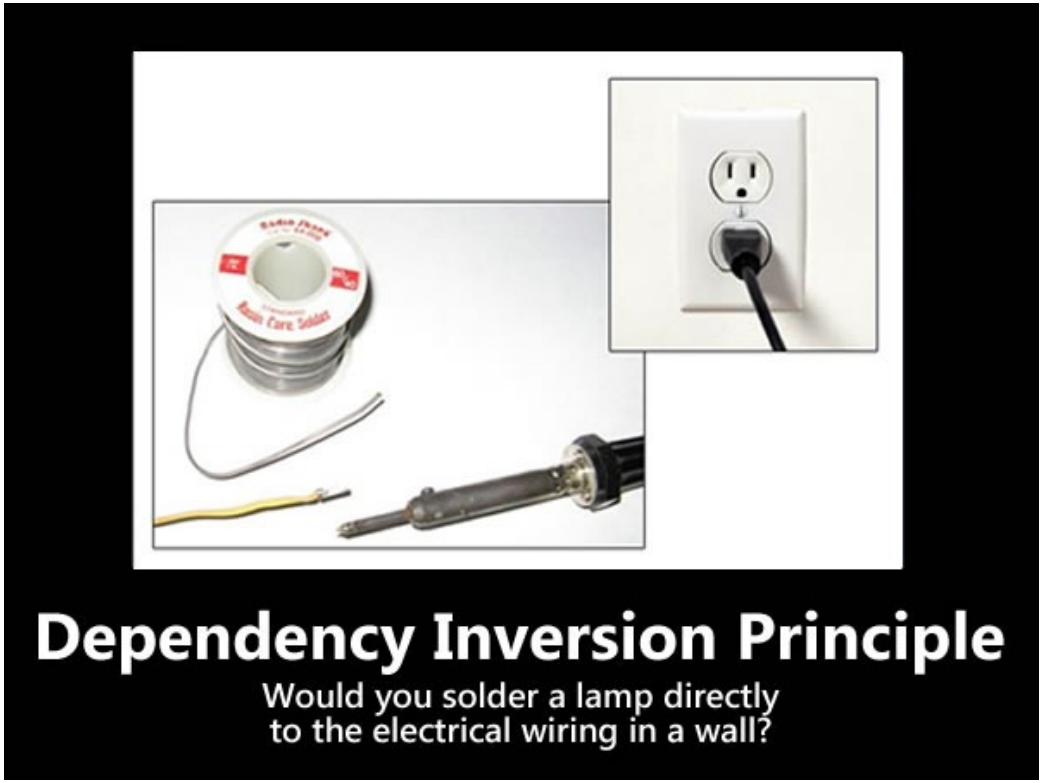
- Standard Java Collection operations (methods):

```
public interface Collection<E> extends Iterable<E> {
  // Basic Collection operations
   int size();
   boolean isEmpty(), contains(Object element);
   boolean add(E element), remove(Object element); // optional
   Iterator<E> iterator();
  // Bulk Collection operations
   boolean containsAll(Collection<?> c);
   boolean addAll(Collection<? extends E> c); // optional
   boolean removeAll(Collection<?> c), retainAll(Collection<?> c); //
    optional
   void clear(); // optional
  // Array operations
   Object[] toArray();
   <T> T[] toArray(T[] a);
} // "optional" methods must be implemented, even if via thrown
    exceptions
```

http://java.sun.com/docs/books/tutorial/collections/interfaces/

The Dependency Inversion Principle

- **<u>High-level modules should not depend on low-level ones</u>, both should depend on shared abstractions. Also, abstractions should not depend on details.**



Benefits of SOLID

- **Low coupling – interfaces are used to abstract away implementation needs, and responsibilities separated**
- **High cohesion – many small pieces can be stacked together like building blocks to create something larger and more complex; pieces can be tied together by depending on abstractions**
- **Encapsulation – implementation details are hidden from external objects by depending on abstractions and their expected behaviors**

Week 8 Topics

- Principles of OO Design - SOLID
- **Agile Development Overview (APPP Chapters 1-3)**
- More Design Patterns (as time permits)
- Android Examples (only if time)

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?

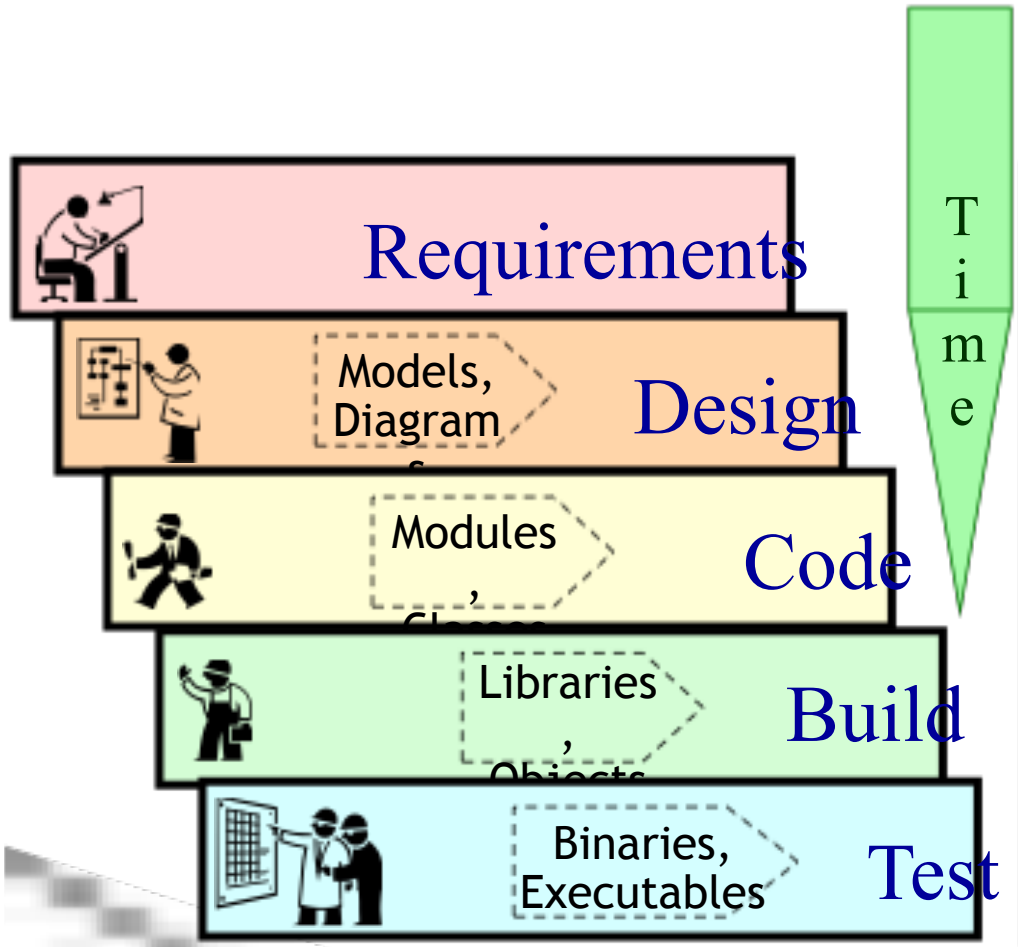
- *Today:*
  - Often used for Software Development
- *Tomorrow:*
  - Systems Development
  - Program Management
  - Hardware Development

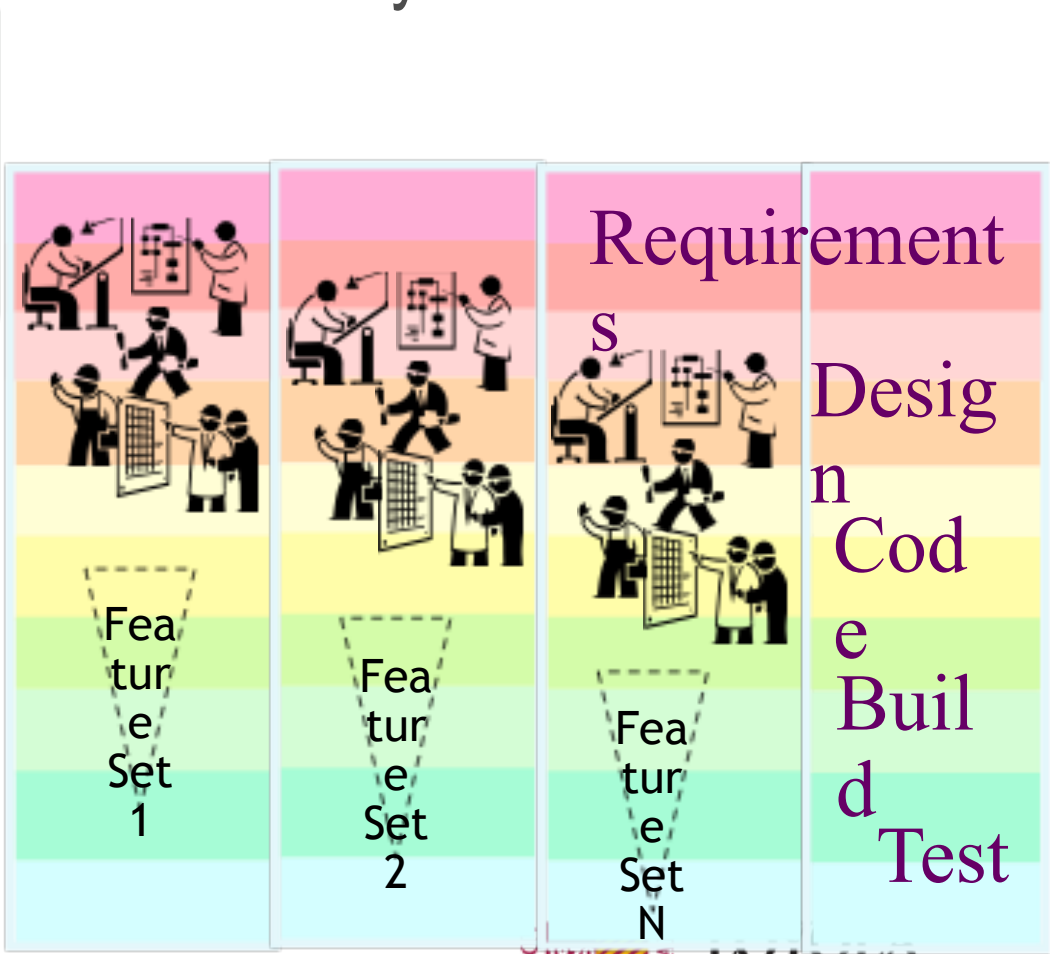
Waterfall Lifecycle

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]

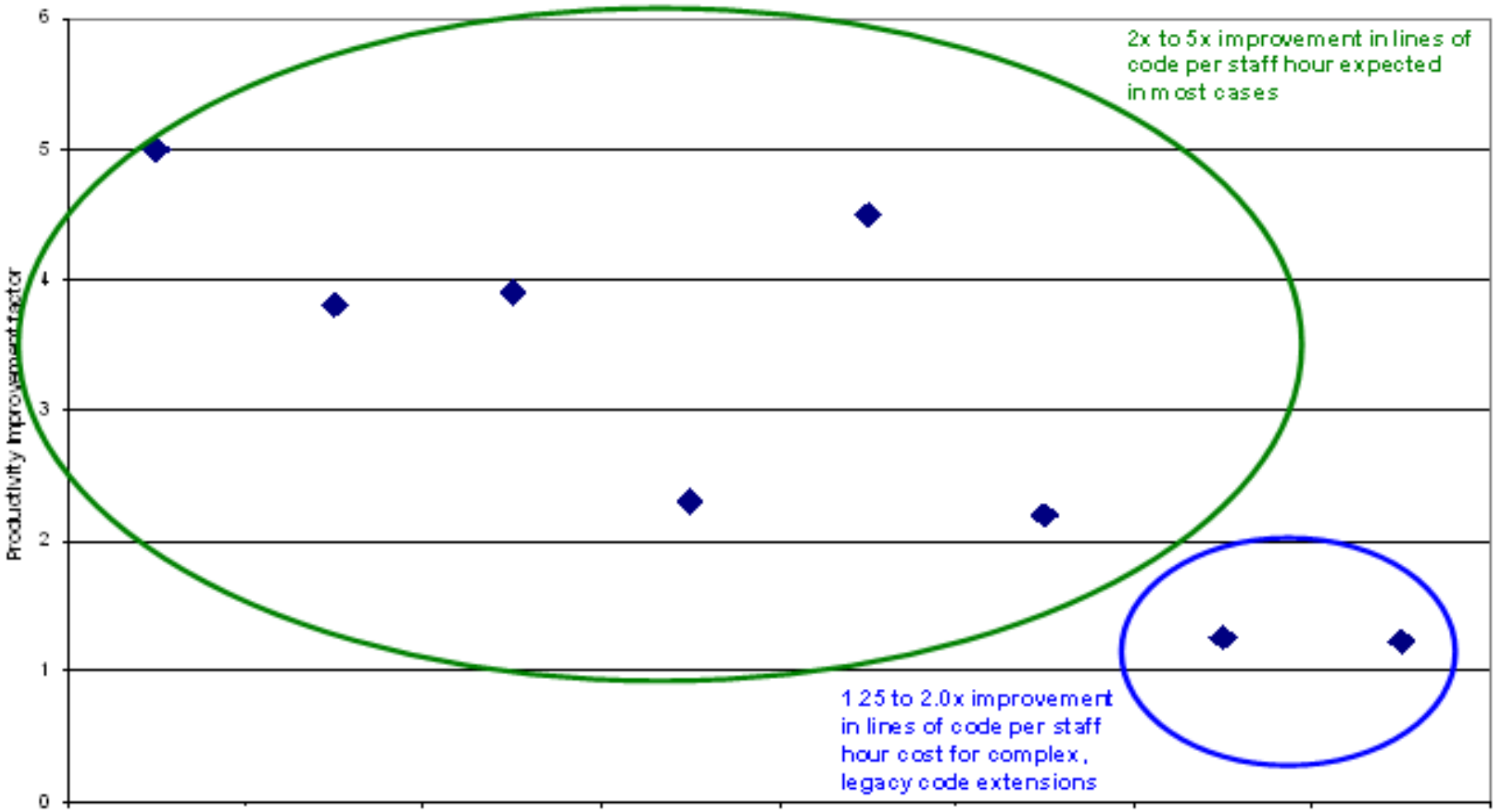
1. Constant Integration
   - Merge+build+test new changes at least once a day
2. Daily Standup
   - Quickly share information and address risks
3. Refactoring
   - “Improving the design of existing code”
4. Customer Proxy
   - Representing the voice of the customer
5. Agile Project Management
   - Managing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

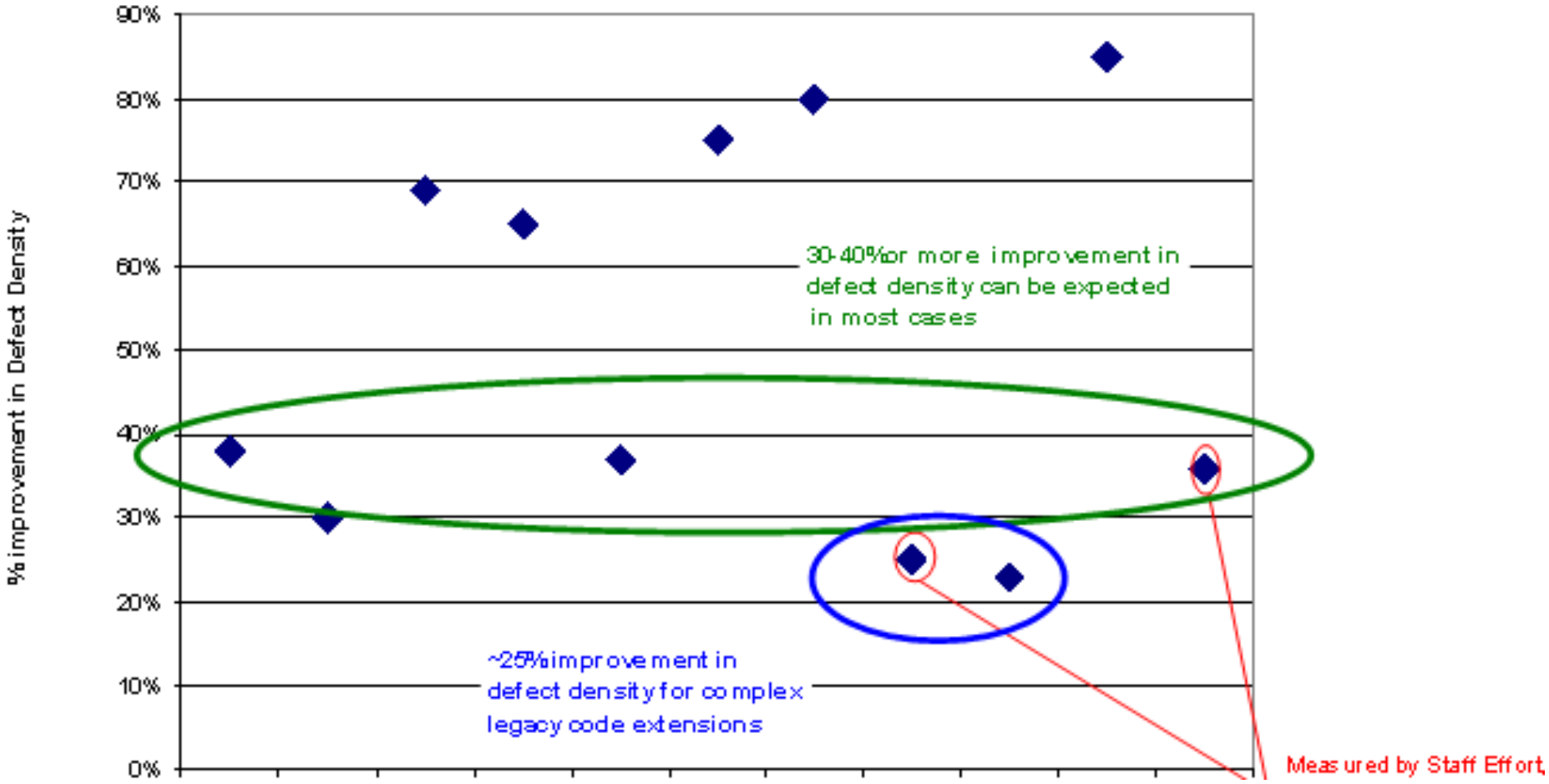
Agile – Demonstrated Productivity Increases

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 8 Topics

- Principles of OO Design – SOLID
- Agile Development Overview
- **More Design Patterns (as time permits)**
  - **Adapter, Façade, Observer, State, Command**
- Android Examples (only if time)

Design Pattern Classification

- The Bob Tarr slides and the "Gang of Four" (GoF) book classify patterns based on their *purpose*:
  - **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

- For more information:
  http://www.programcreek.com/java-design-patterns-in-stories/

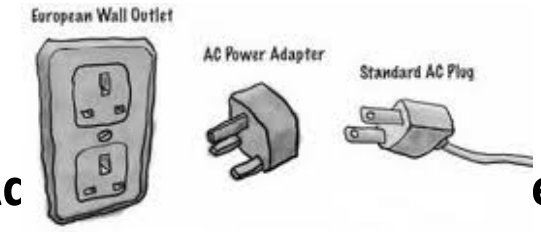
Design Pattern Overview Lectures

Pattern	GoF Classification	Week Covered
Command*	**Behavioral**	**Week 8**
Iterator	Behavioral	TBD
Observer	**Behavioral**	**Week 8**
State	**Behavioral**	**Week 8**
Strategy	Behavioral	Week 4
Visitor	Behavioral	Weeks 4 & 5
Abstract Factory	Creational	TBD
Builder	Creational	TBD
Factory Method	Creational	Week 4
Prototype	Creational	TBD
Singleton	Creational	TBD
Adapter	**Structural**	**Week 8**
Composite	Structural	Week 5
Decorator	Structural	Week 5
Façade	**Structural**	**Week 8**

* Not GoF

The Adapter Pattern (Structural)

From Wikipedia, the free encyclopedia

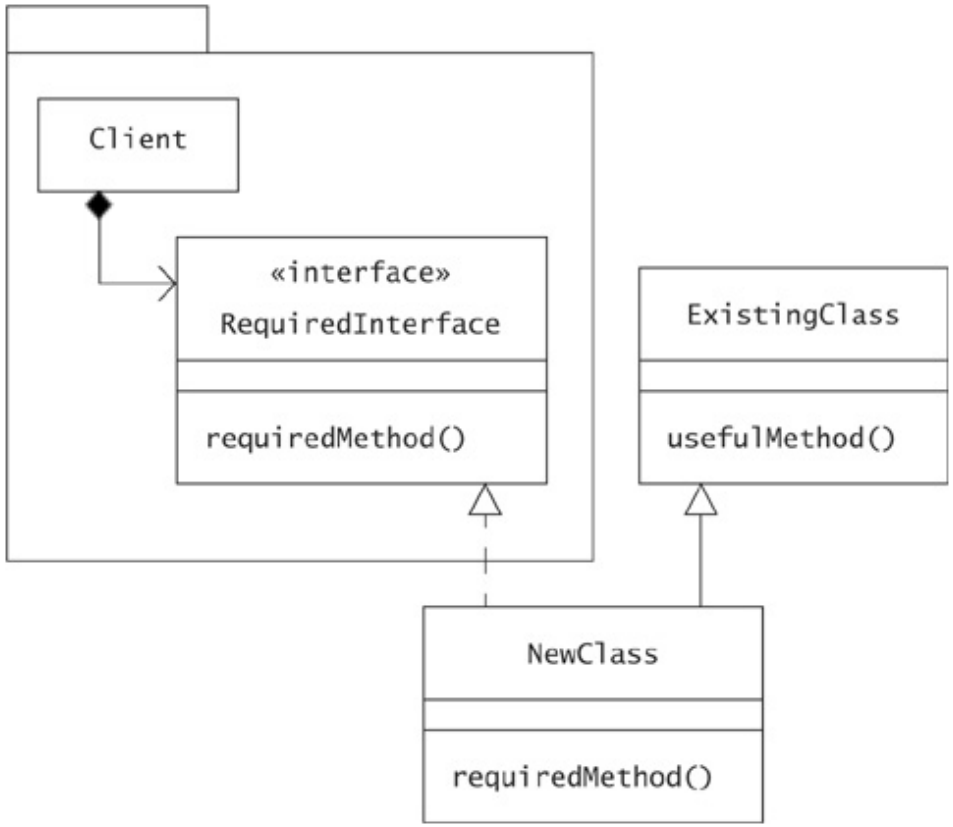
- In computer programming, the **adapter design pattern** (often referred to as the **wrapper pattern** or simply a **wrapper**) **translates one interface for a class into a compatible interface**. **An *adapter* allows classes to work together that normally could not because of incompatible interfaces**, by providing its interface to clients while using the original interface. The adapter translates calls to its interface into calls to the original interface, and the amount of code necessary to do this is typically small.
- **The adapter is also responsible for transforming data into appropriate forms.** For instance, if multiple boolean values are stored as a single integer but your consumer requires a 'true'/'false', the adapter would be responsible for extracting the appropriate values from the integer value.
- **There are two types of Ac** 

ect

DPiJ Interface Patterns (Ch. 2) and the Adapter Pattern (Ch. 3)

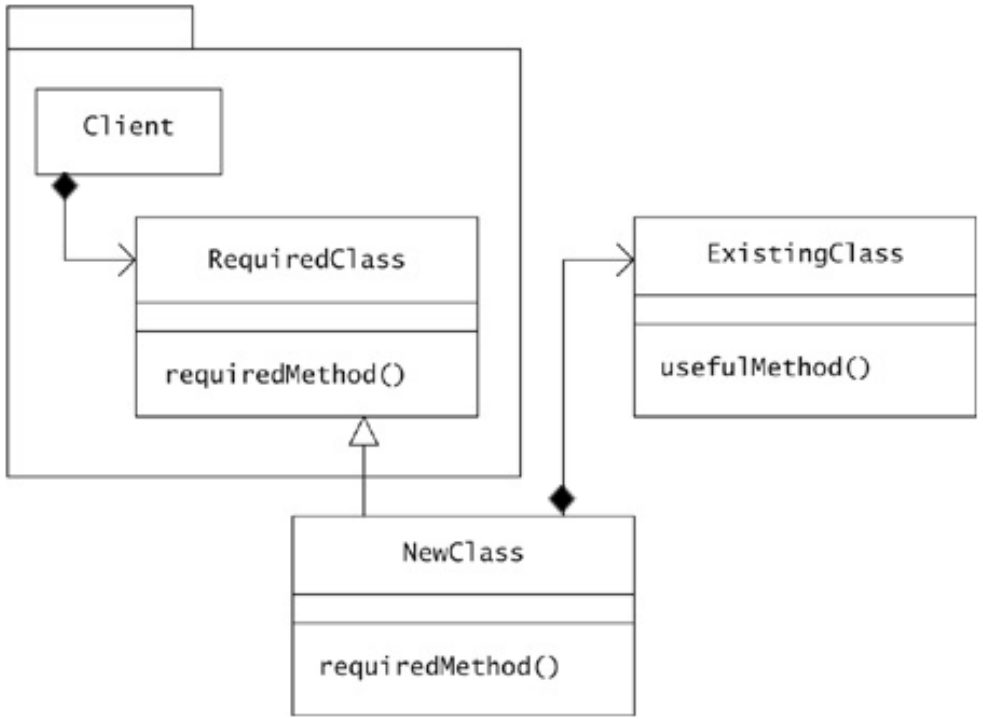
- DPiJ says that Interface patterns like Adapter allow you to define or redefine access to the methods of a class or group of classes, for example to match an existing class' service(s) to the method names or interface expected by a client
- The intent of the Adapter pattern is to provide the interface that a client expects while using the services of a class with a different interface – to meet the client's needs while using an existing, but mismatching, class
  - When a class specifies its requirements via an interface, a ***class adapter*** adapts through subclassing by implementing the required interface and extending (subclassing) the existing class
  - Otherwise an ***object adapter*** may be able to adapt by extending (subclassing) the required class, overriding its methods, and delegating to a composed instance of the existing class

Class Adapter Pattern: Most Common, where the Client Class Uses an Interf



- In this case the Adapter implements the interface required by the client class and extends the Adaptee class in order to access its methods. It may also delegate to an Adaptee object via composition.

Object Adapter Pattern – if the Client Class Doesn't Use an Interface



- In this case the Adapter extends the required class, overrides its required methods, and delegates to an Adaptee object via composition.

http://www.tutorialspoint.com/design_pattern/adapter_pattern.htm
http://my.safaribooksonline.com/book/programming/java/9780321630483/interface-patterns/ch03 - DPIJ free Chapter 3, Adapter
APPP Chapter 33, pp 498-506
→ Bob Tarr Adapter slides 2-17, 21-27

The Façade Pattern (Structural)

From Wikipedia, the free encyclopedia

The **façade pattern** is a software engineering design pattern commonly used with Object-oriented programming. (The name is by analogy to an architectural facade.)

A facade is an object that provides a simplified interface to a larger body of code, such as a class library. A facade can:

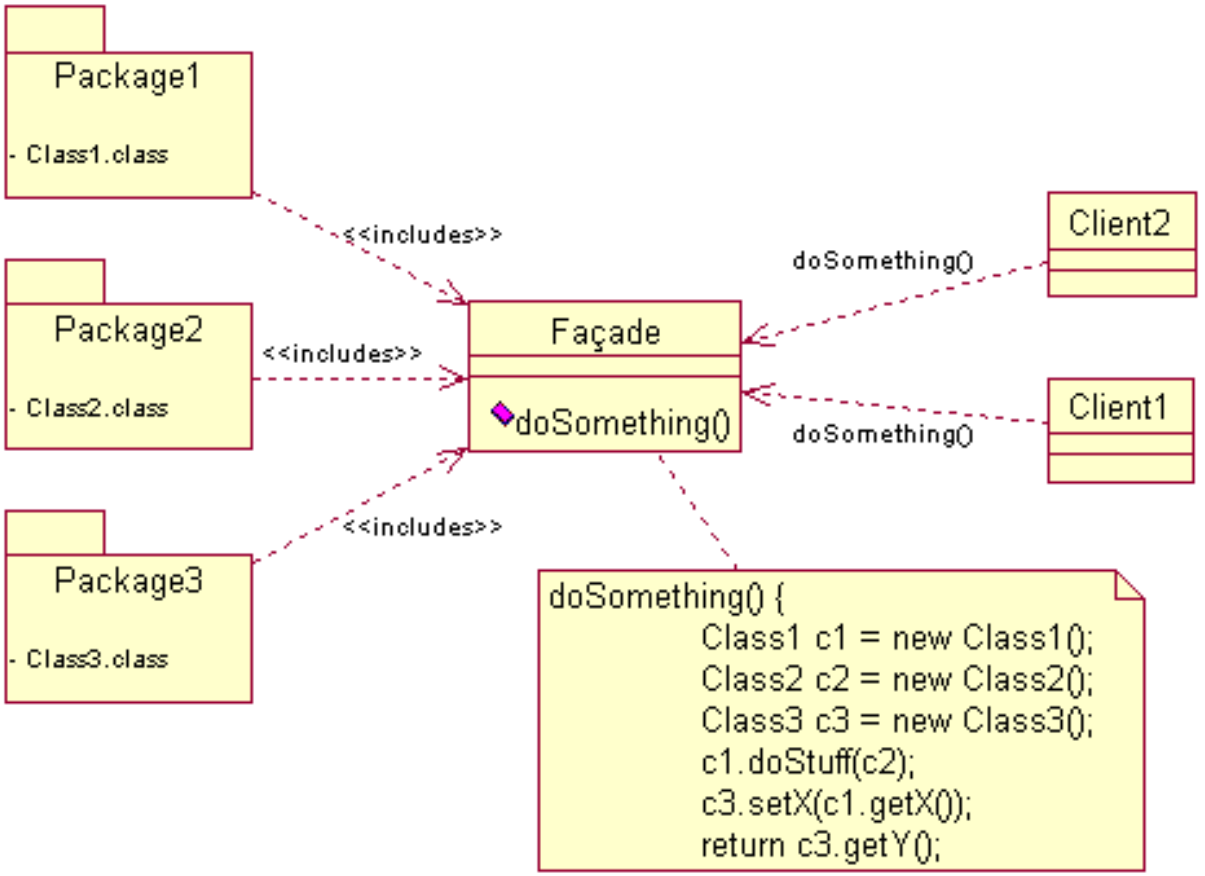
- make a software library easier to use and understand, since the facade has convenient methods for common tasks;
- make code that uses the library more readable, for the same reason;
- reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system;
- wrap a poorly-designed collection of APIs with a single well-designed API (as per task needs).

An Adapter is used when the wrapper must respect a particular interface and must support a polymorphic behavior. On the other hand, a façade is used when one wants an easier or simpler interface to work with: Software library / API collection accessed through the Façade Class.

DPiJ Interface Patterns (Ch. 2) and the Façade Pattern (Ch. 4)

- DPiJ says that Interface patterns like Composite allow you to define or redefine access to the methods of a class or group of classes; in particular, **the Façade Pattern provides an interface that makes a subsystem easy to use**.
  - In an OO system, an application is a minimal class that knits together the behaviors from reusable toolkits / subsystems of other classes
  - Since these toolkits can be used in a wide variety of domain-specific applications, there may be an diverse and extensive set of options that are difficult to manage
  - **The Façade pattern simplifies the use of a toolkit by providing a typical, no-frills usage of the classes in a class library** – a façade is a configurable and reusable class with a level of functionality that lies between a toolkit and a complete application, offering a simplified way to use the classes in one or more packages or subsystems
  - A façade may come about by factoring out a simplifying access class

The Façade Pattern



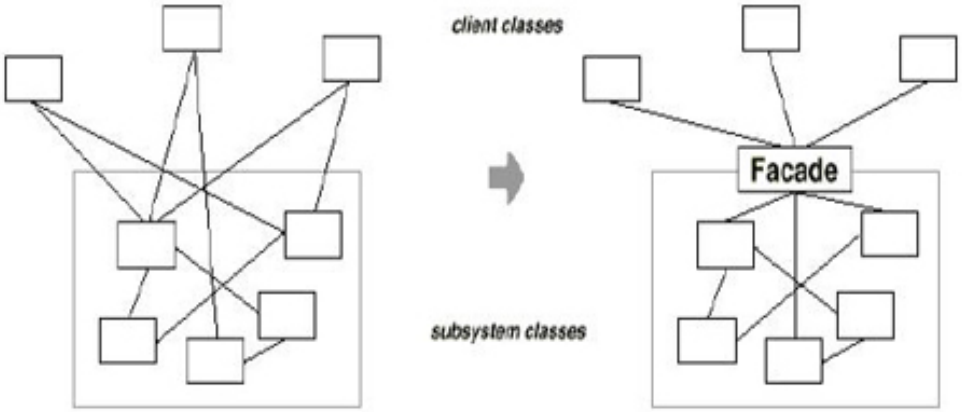
Façade

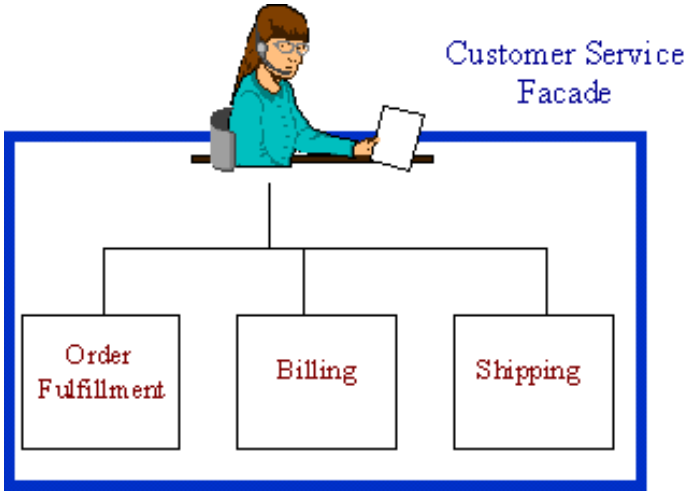
The facade class abstracts Packages 1, 2, and 3 from the rest of the application.

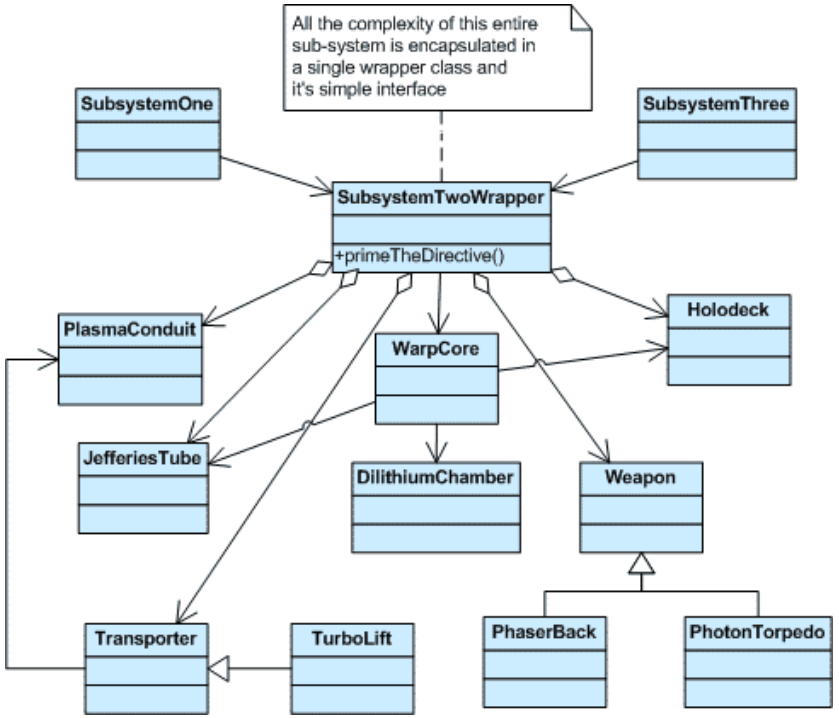
Clients

The objects using the Façade Pattern to access resources from the Packages.

Façade Pattern Examples







Façade Pattern Java Example

```
/* Complex parts */
class CPU {
   public void freeze() { ... }
   public void jump(long position) { ... }
   public void execute() { ... }
}
class Memory { public void load(long position, byte[] data) { ... } }
class HardDrive { public byte[] read(long lba, int size) { ... } }
/* Façade */ // giving you a nicer interface to a complex computer
class Computer {
   public void startComputer() {
         cpu.freeze();
         memory.load(BOOT_ADDRESS, hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
       cpu.jump(BOOT_ADDRESS);
       cpu.execute();
   }
}
/* Client */
class You {
   public static void main(String[] args) {
       Computer facade = new Computer();
       facade.startComputer();
   }
}
```

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

APPP Chapter 23, pp 325-326

→ Bob Tarr Slides 1-7

Observer Design Pattern (Behavioral)

From Wikipedia, the free encyclopedia

- The **observer pattern** (a subset of the asynchronous publish/subscribe pattern) is a software design pattern in which **an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes**, usually by calling one of their methods. **It is mainly used to implement distributed event handling systems.**
- Simply, the Observer pattern allows one object (the observer) to watch another (the subject). **The Observer pattern allows the subject and observer to form a publish-subscribe relationship.** Through the Observer pattern, **observers can register to receive events** from the subject. When the subject needs to inform its observers of an event, it simply sends the event to each observer.
- For example, you might have a spreadsheet that has an underlying data model. Whenever the data model changes, the spreadsheet will need to update the spreadsheet screen and an embedded graph. In this example, the subject is the data model and the observers are the screen and graph. When the observers receive notification that the model has changes, they can update themselves.
- **The benefit: it decouples the observer(s) from the subject.** The subject doesn't need to know anything special about its observers. Instead, the subject simply allows observers to subscribe. When the subject generates an event, it simply passes it to each of its observers.

DPiJ Responsibility Patterns (Ch. 7) and the Observer Pattern (Ch. 9)

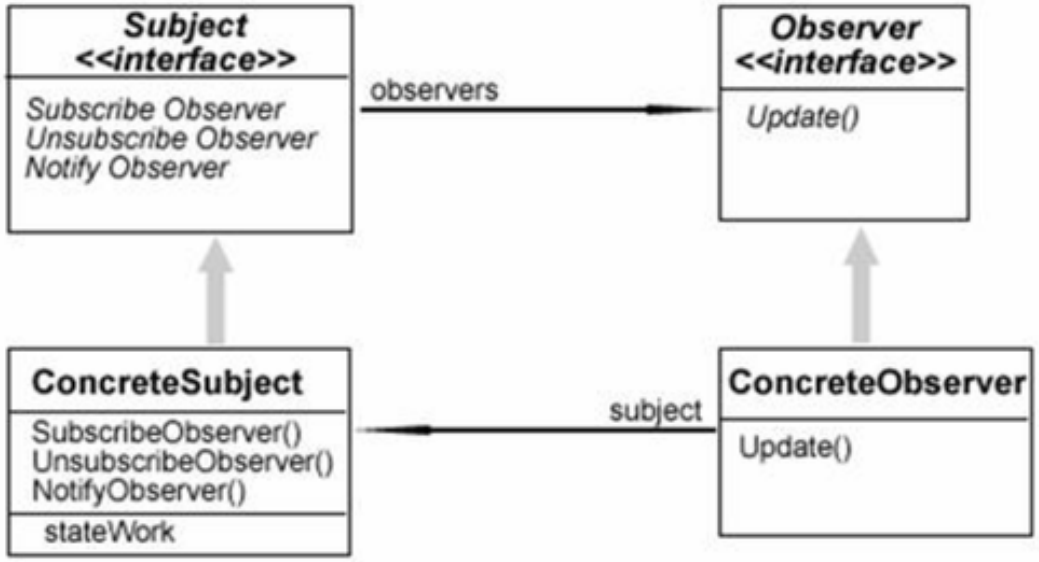
- DPiJ says Responsibility patterns like Observer allow us to centralize, escalate, and/or limit ordinary object responsibility – OO responsibility is normally distributed
- In some designs, objects are responsible for informing clients when there is an "interesting" change in some aspects of those objects. **The Observer design pattern allows clients to be informed about all object changes, leaving it to the clients to follow up as appropriate.**
- **The intent of the Observer pattern is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified so they can react to the change.**

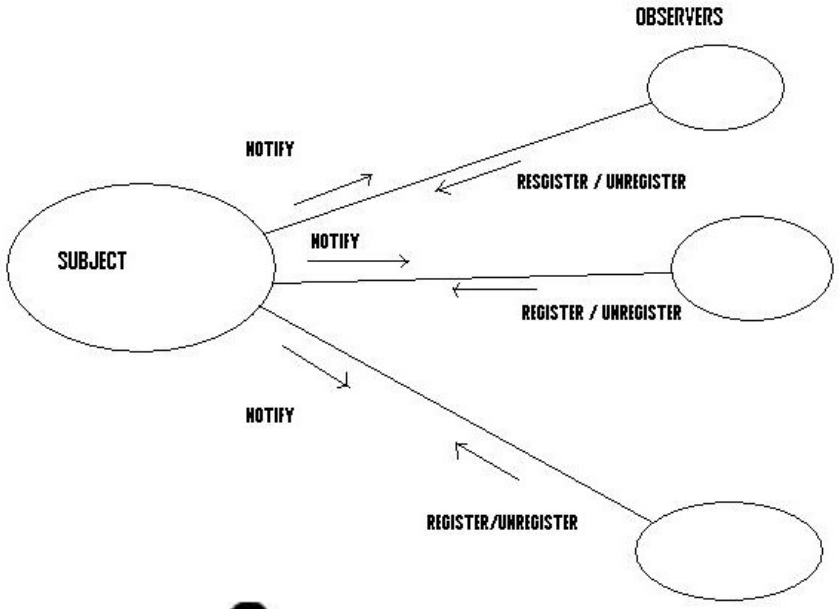
http://www.tutorialspoint.com/design_pattern/observer_pattern.htm

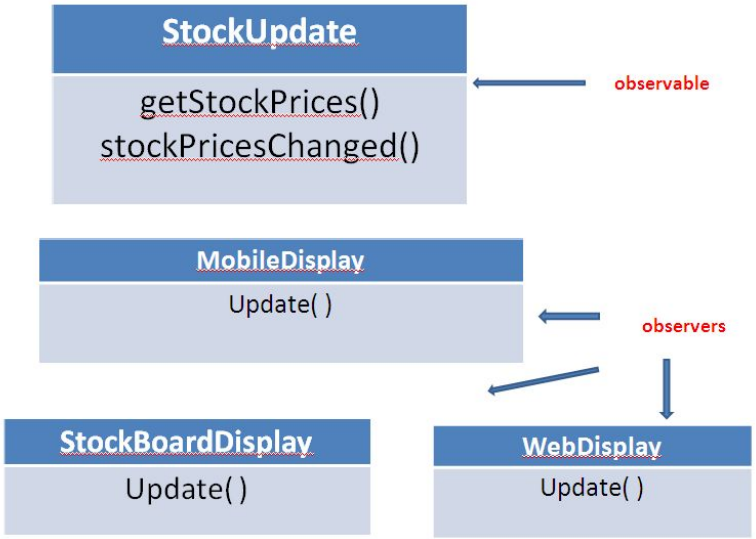
APPP Chapter 32

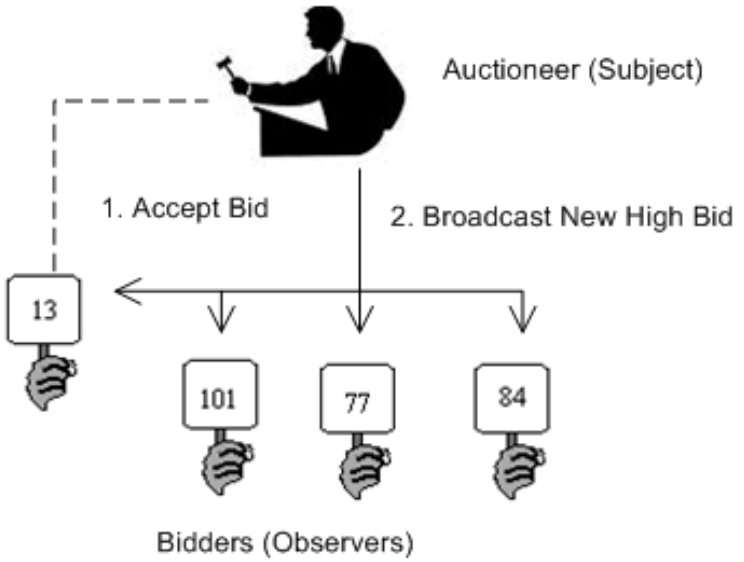
→ Bob Tarr Slides 1-62

Observer Pattern Examples









LOYOLA UNIVERSITY CHICAGO

Bloch on the Observer Pattern – #67

The following class implements an observable set wrapper that allows clients to subscribe to notifications when elements are added to the set – it uses the ForwardingSet from Item #16.

```
public class ObservableSet<E> extends ForwardingSet<E> {
    public ObservableSet(Set<E> set) { super(set); }

    private final List<SetObserver<E>> observers =
        new ArrayList<SetObserver<E>>();

    public void addObserver(SetObserver<E> observer) {
        synchronized(observers) {
            observers.add(observer);
        }
    }
    public boolean removeObserver(SetObserver<E> observer) {
        synchronized(observers) {
            return observers.remove(observer);
        }
    }
    private void notifyElementAdded(E element) {
        synchronized(observers) {
            for (SetObserver<E> observer : observers)
                observer.added(this, element);
        }
    }
    @Override public boolean add(E element) {
        boolean added = super.add(element);
        if (added)
            notifyElementAdded(element);
        return added;
    }

    @Override public boolean addAll(Collection<? extends E> c) {
        boolean result = false;
        for (E element : c)
            result |= add(element); // calls notifyElementAdded
        return result;
    }
}
```

Observers subscribe to notifications this way:

```
public interface SetObserver<E> {
    // Invoked when an element is added to the observable set
    void added(ObservableSet<E> set, E element);
}
```

Subscribing & printing numbers 0 through 99:

```
public static void main(String[] args) {
    ObservableSet<Integer> set =
        new ObservableSet<Integer>(new HashSet<Integer>());

    set.addObserver(new SetObserver<Integer>() {
        public void added(ObservableSet<Integer> s, Integer e) {
            System.out.println(e);
        }
    });

    for (int i = 0; i < 100; i++)
        set.add(i);
}
```

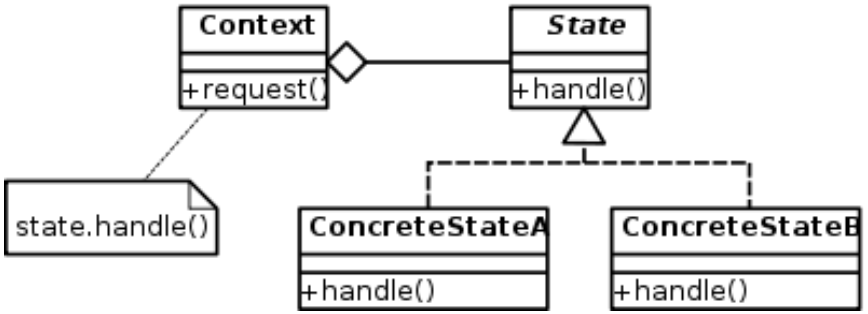
Note: Bloch specifically says this is not a threadsafe implementation!

The State Pattern (Behavioral)

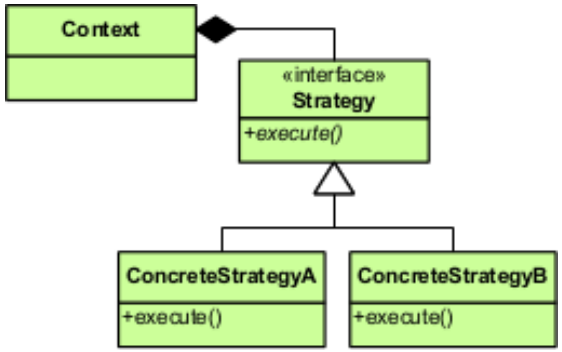
From Wikipedia, the free encyclopedia

- The **state pattern**, which closely resembles Strategy Pattern, is a behavioral software design pattern, also known as the **objects for states pattern**. This pattern is used in computer programming to encapsulate varying behavior for the same routine based on an object's state object. This can be a cleaner way for an object to change its behavior at runtime without resorting to large monolithic conditional statements.
- The **Context** in the State Pattern holds <u>different</u> **state objects** over time.

<u>State Pattern</u>



<u>Strategy Pattern</u>



DPiJ Operations Patterns (Ch. 21)

- DPiJ says that operations patterns like State and Strategy "implement an operation in methods across several classes", and goes on to say that (based on UML concepts) …
  - An **operation** is a specification of a service that can be requested from an instance of a class, eg, **void toString()**
  - A **method** is an implementation of an operation
- An operation is thus a level of abstraction up from the concept of a method
- The concept of an *operation* relates to the idea that methods in different classes can have the same interface but implement that interface in different ways
  - Operations patterns address contexts where you need more than one method with the same interface as part of a design

The State Pattern (DPiJ Ch. 22)

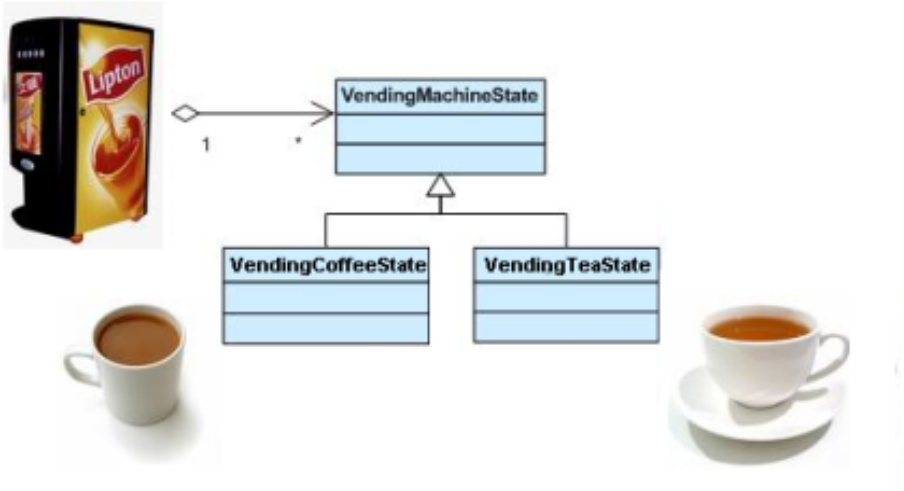
- If you intend to distribute an operation so that each class represents a different state, use the State Pattern
  - If an object has a state that you want to model, you may find that the state is tracked via a variable which appears in complex, cascading *if* statements; the State Pattern offers a simpler approach
  - In this pattern, states are modeled as objects from different classes
  - Communication & dependencies can be managed in different ways

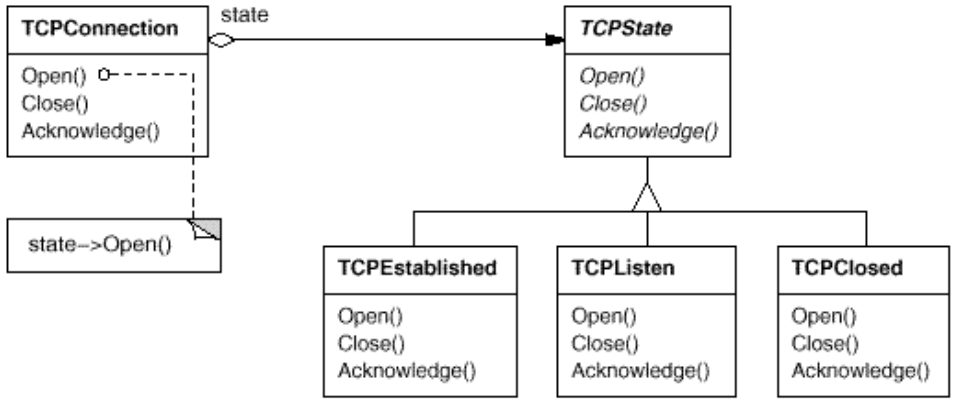
http://www.tutorialspoint.com/design_pattern/state_pattern.htm

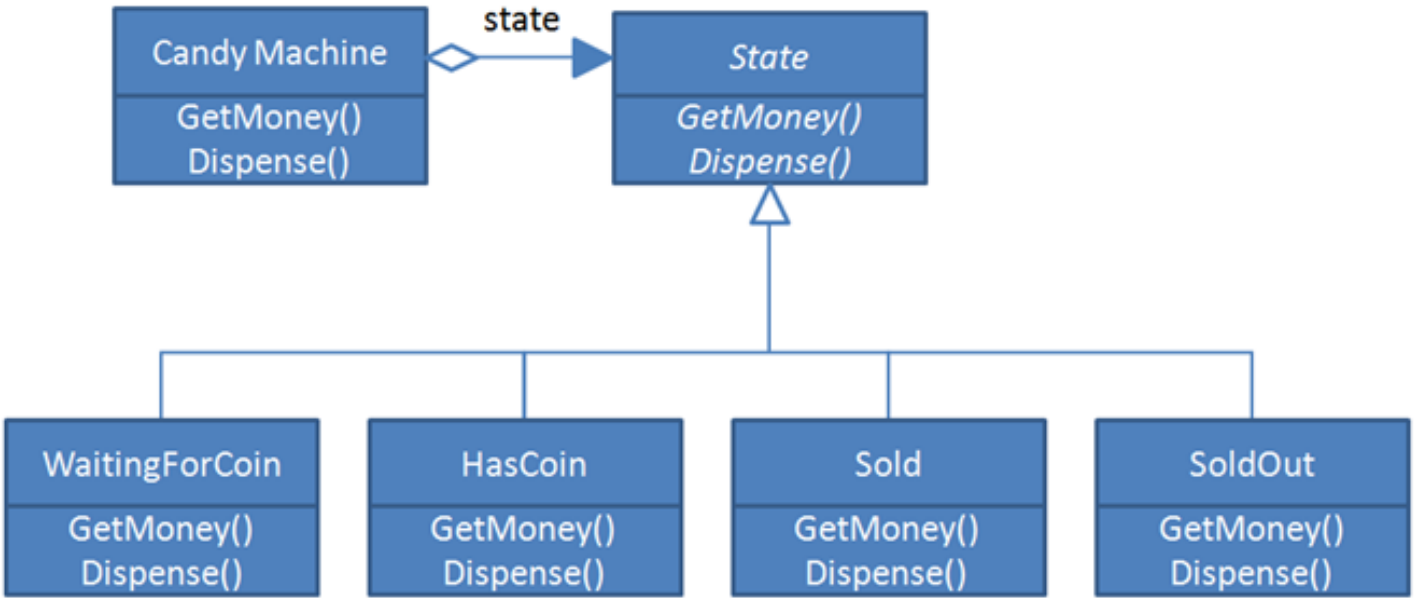
APPP Chapter 36

→ **Bob Tarr State slides 2-4, 13-20**

State Design Pattern Examples







State Design Pattern Observations

What are some potential disadvantages of using the State Pattern, both in general and specifically for this particular example?

- General: can proliferate state objects; some operations may not be needed or may not be appropriate for all states
- Specific: there are so few states in this example that it could easily be handled with a nested *if* or a *switch* statement based on *enums* (but it would not be as extensible)

How are the State and Strategy Patterns similar and how are they different?

- Similar: very similar implementation structures; both based on a common interface
- Different: State intent is to encapsulate state-dependent behavior and transitions; Strategy intent is to encapsulate an algorithm

The Command Pattern (Behavioral)

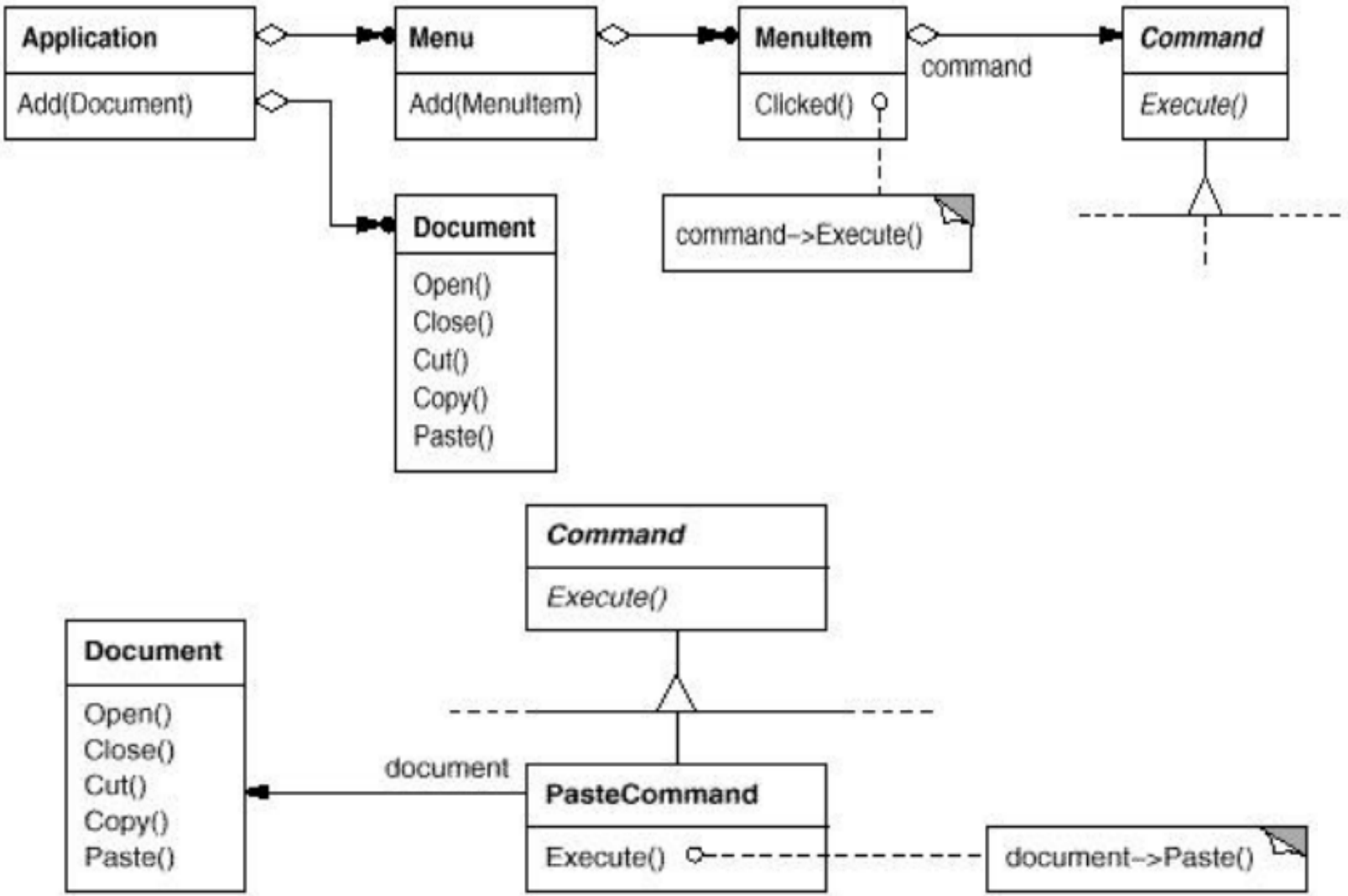
From Wikipedia, the free encyclopedia

- In object-oriented programming, the **command pattern** is a behavioral design pattern in which an object is used to represent and encapsulate all the information needed to call a method at a later time. This information includes the method name, the object that owns the method and values for the method parameters.
- Four terms always associated with the command pattern are ***command***, ***receiver***, ***invoker*** and ***client***. A ***command*** object has a ***receiver*** object and invokes a method of the receiver in a way that is specific to that receiver's class. The ***receiver*** then does the work. A command object is separately passed to an ***invoker*** object, which invokes the command, and optionally does bookkeeping about the command execution. Any command object can be passed to the same invoker object. Both an invoker object and several command objects are held by a ***client*** object. The client contains the decision making about which commands to execute at which points. To execute a command, it passes the command object to the invoker object. See example code below.

Command Pattern Observations

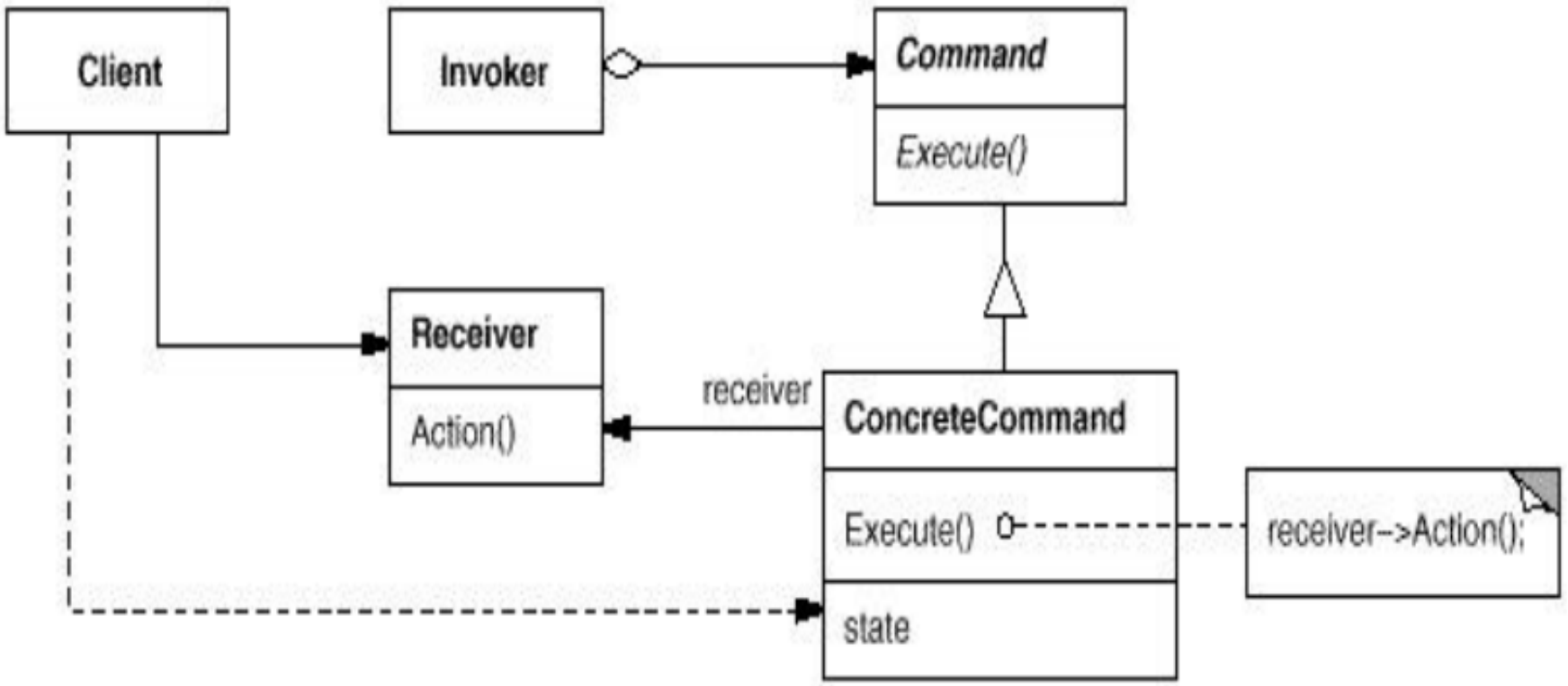
- Using command objects makes it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the class of the method or the method parameters. Using an invoker object allows bookkeeping about command executions to be conveniently performed, as well as implementing different modes for commands, which are managed by the invoker object, without the need for the client to be aware of the existence of bookkeeping or modes.
- Command objects are useful for implementing GUI buttons, menu items, callback functions, mobile code, multi-level undo, thread pools, and transactional behavior, among others

Command Pattern Motivation



LOYOLA UNIVERSITY CHICAGO

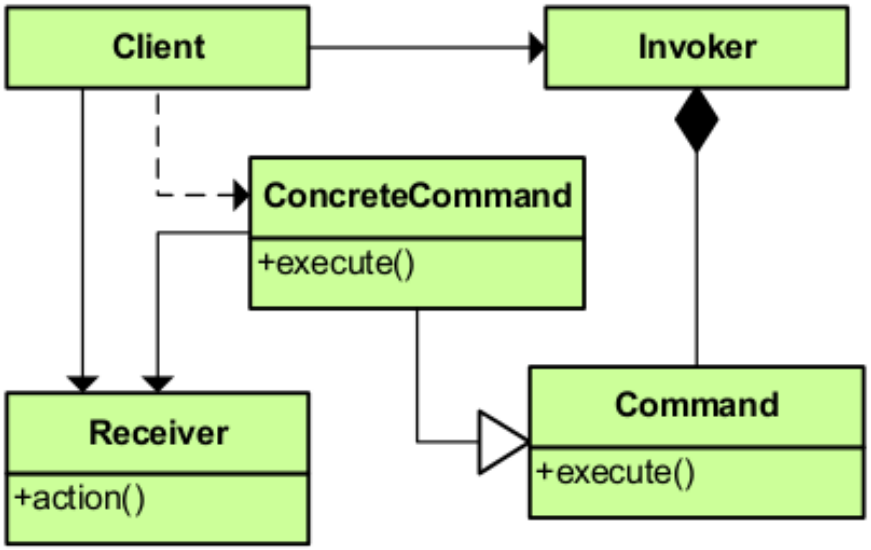
Command Pattern UML Diagram



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

APPP Chapter 21

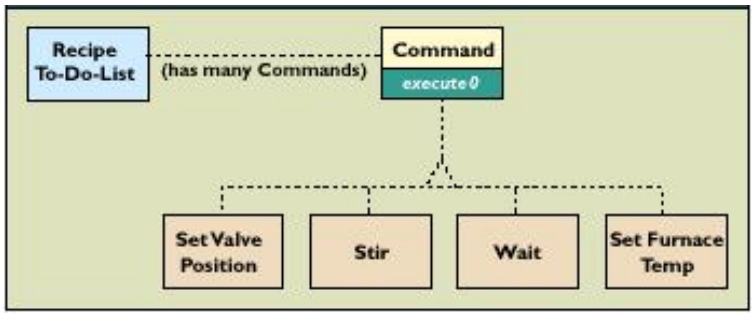
Command Design Pattern Examples

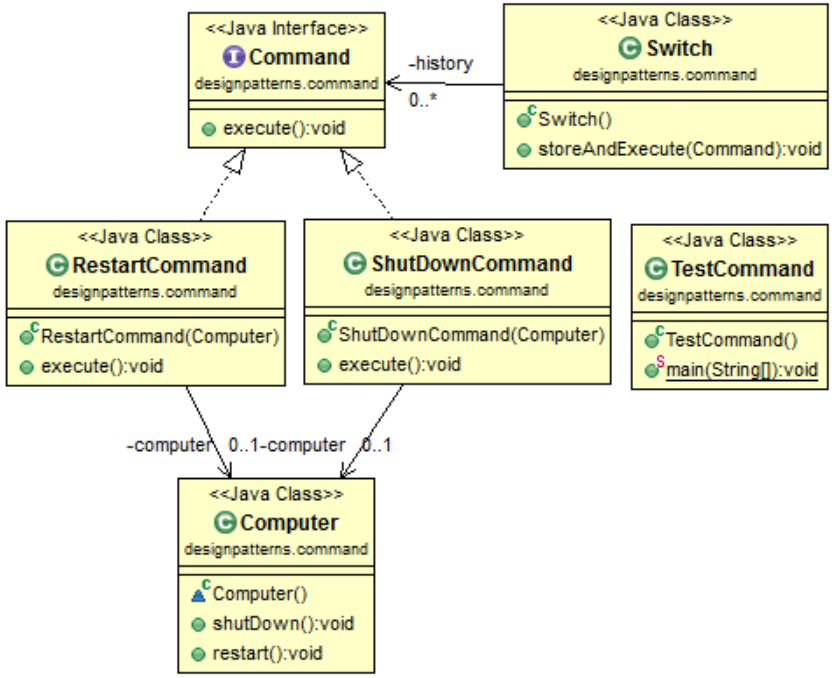


Command

Type: Behavioral

What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



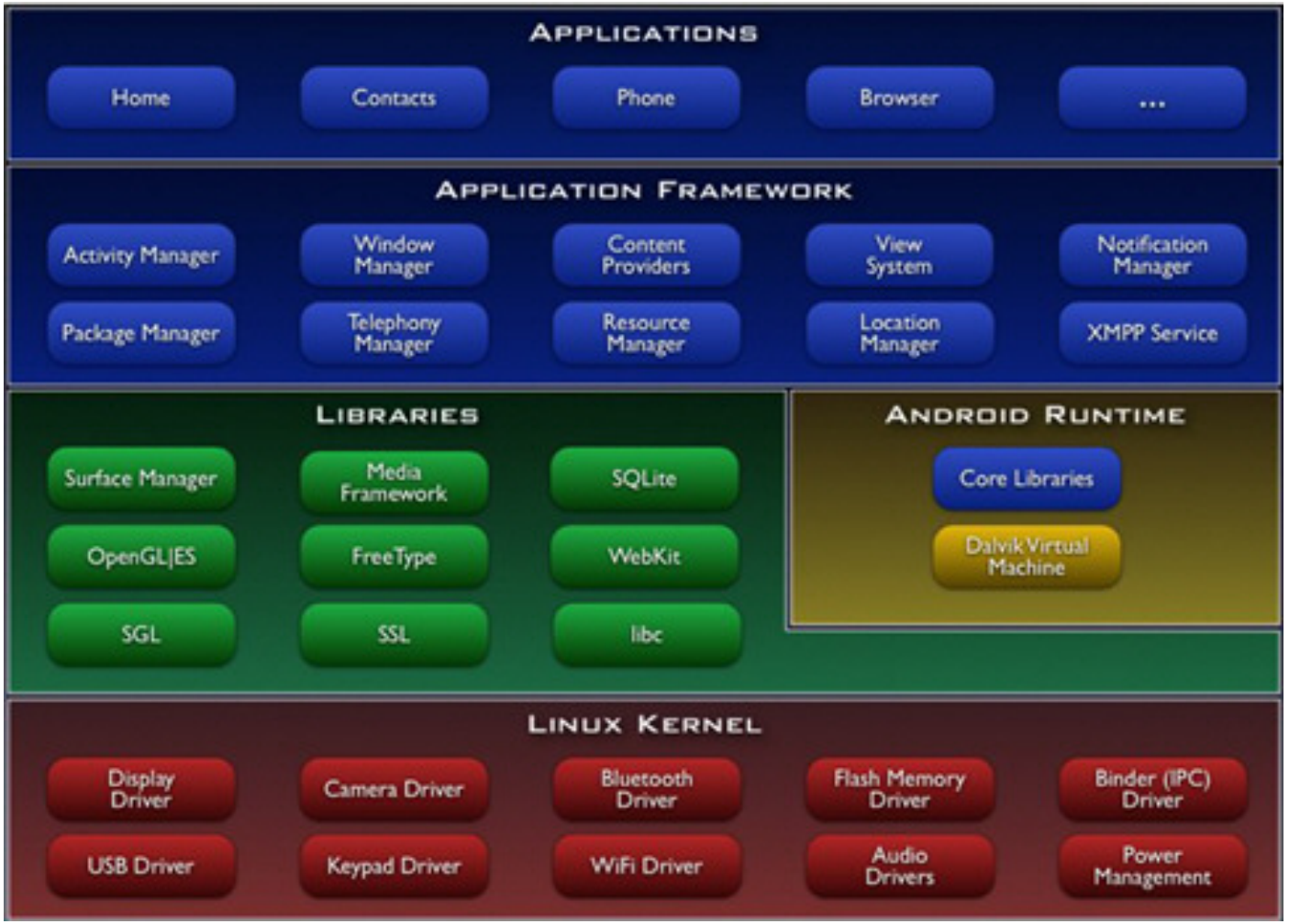


LOYOLA UNIVERSITY CHICAGO

Week 8 Topics

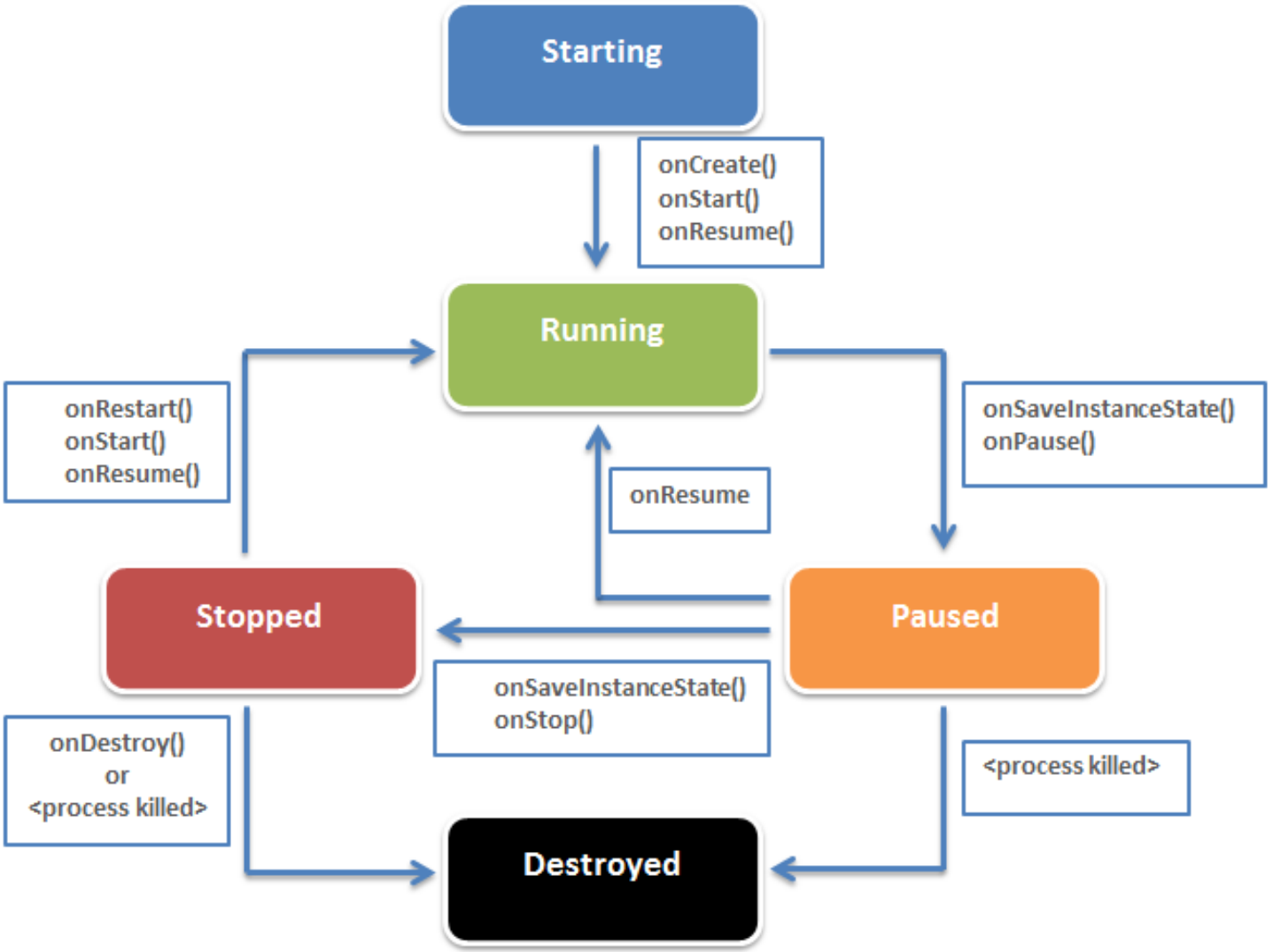
- Principles of OO Design – SOLID
- Agile Development Overview
- More Design Patterns (as time permits)
- **Android Examples (only if time)**

Android Framework/Architecture



http://www.techotopia.com/index.php/An_Overview_of_the_Android_Architecture

Android Activity Life Cycle



http://www.techotopia.com/index.php/Understanding_Android_Application_and_Activity_Lifecycles
http://www.techotopia.com/index.php/Handling_Android_Activity_State_Changes

Example Android Programs

- **hello-android-java – notification**
- **simplebatch-android-java – scrollable text output**
  - **The Android Activity Lifecycle**
- **simpledraw-android-java**

Week 8 Topics

- Principles of OO Design - SOLID
- Agile Development Overview
- More Design Patterns (as time permits)
- Android Examples (only if time)

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.

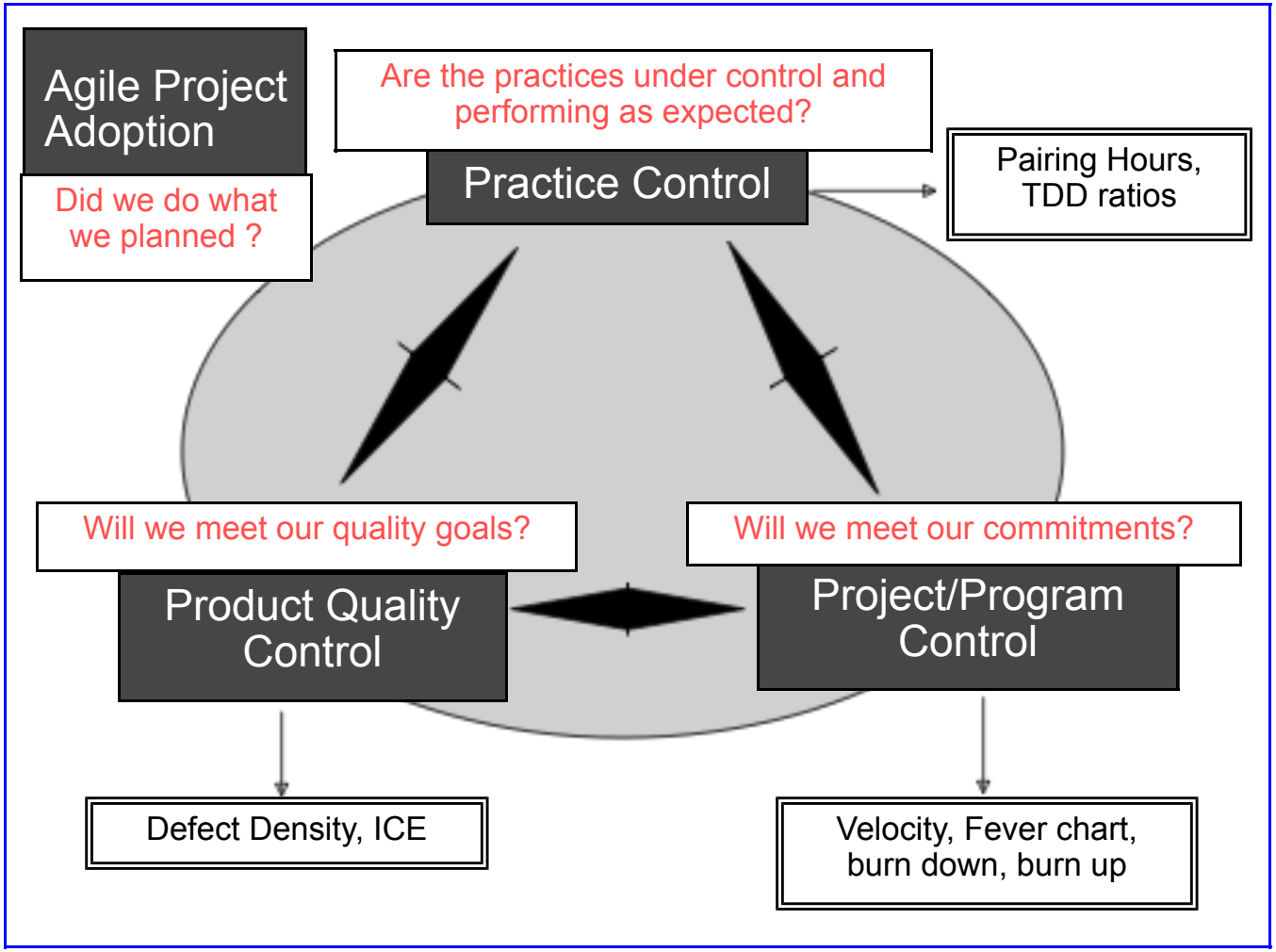
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

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

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

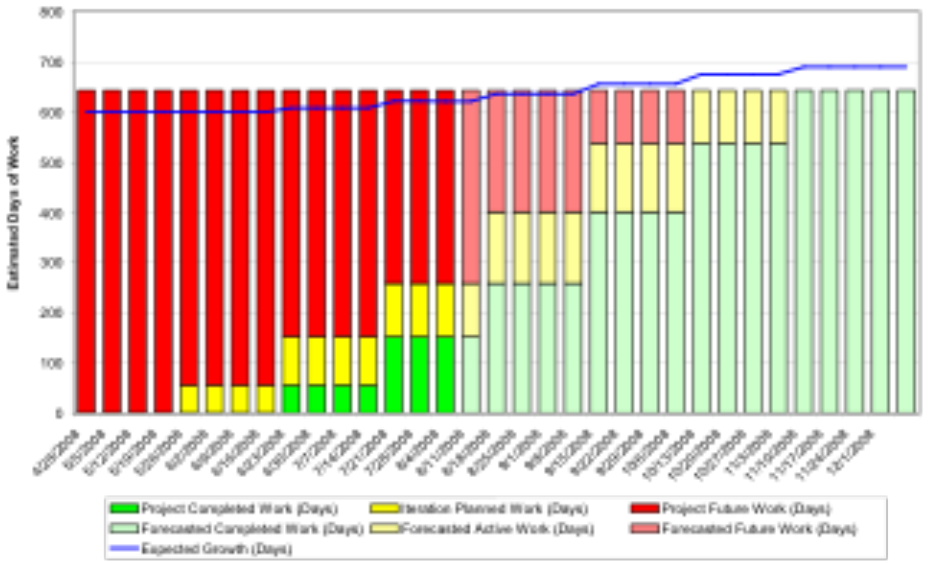
Monitoring and Controlling Agile Projects



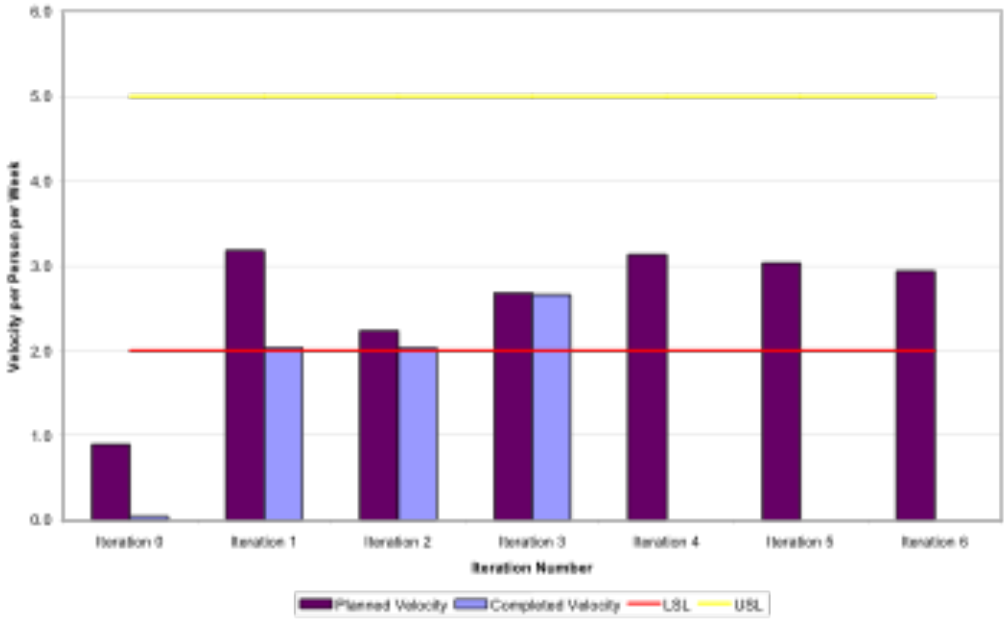
Metrics for tracking and monitoring Agile impact

Agile Project Management Charts

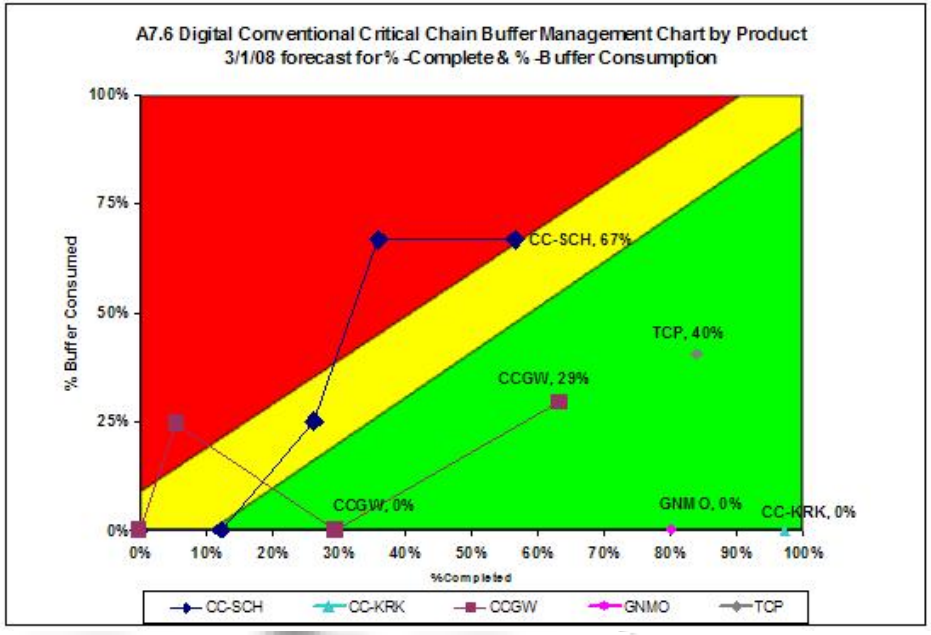
Burn-up Chart



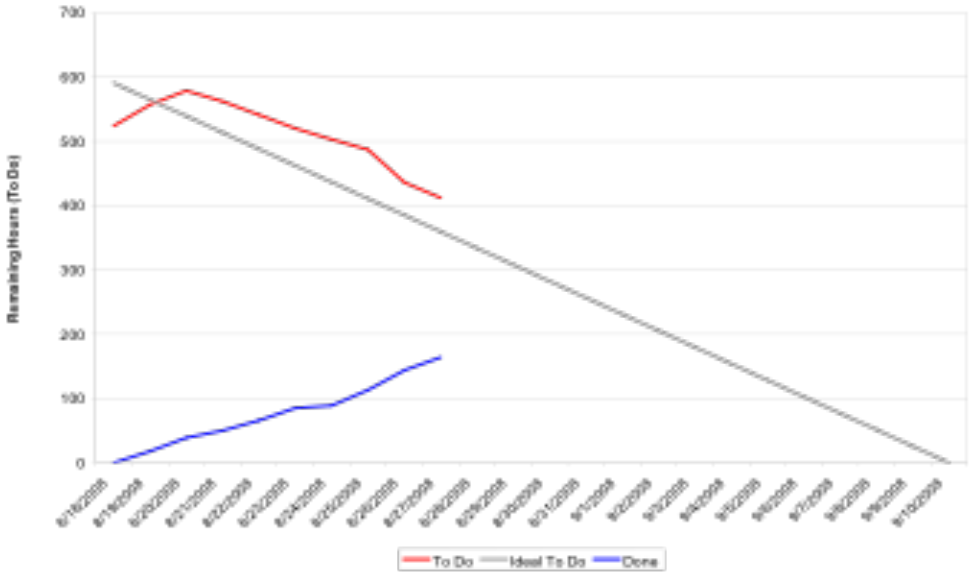
Velocity chart



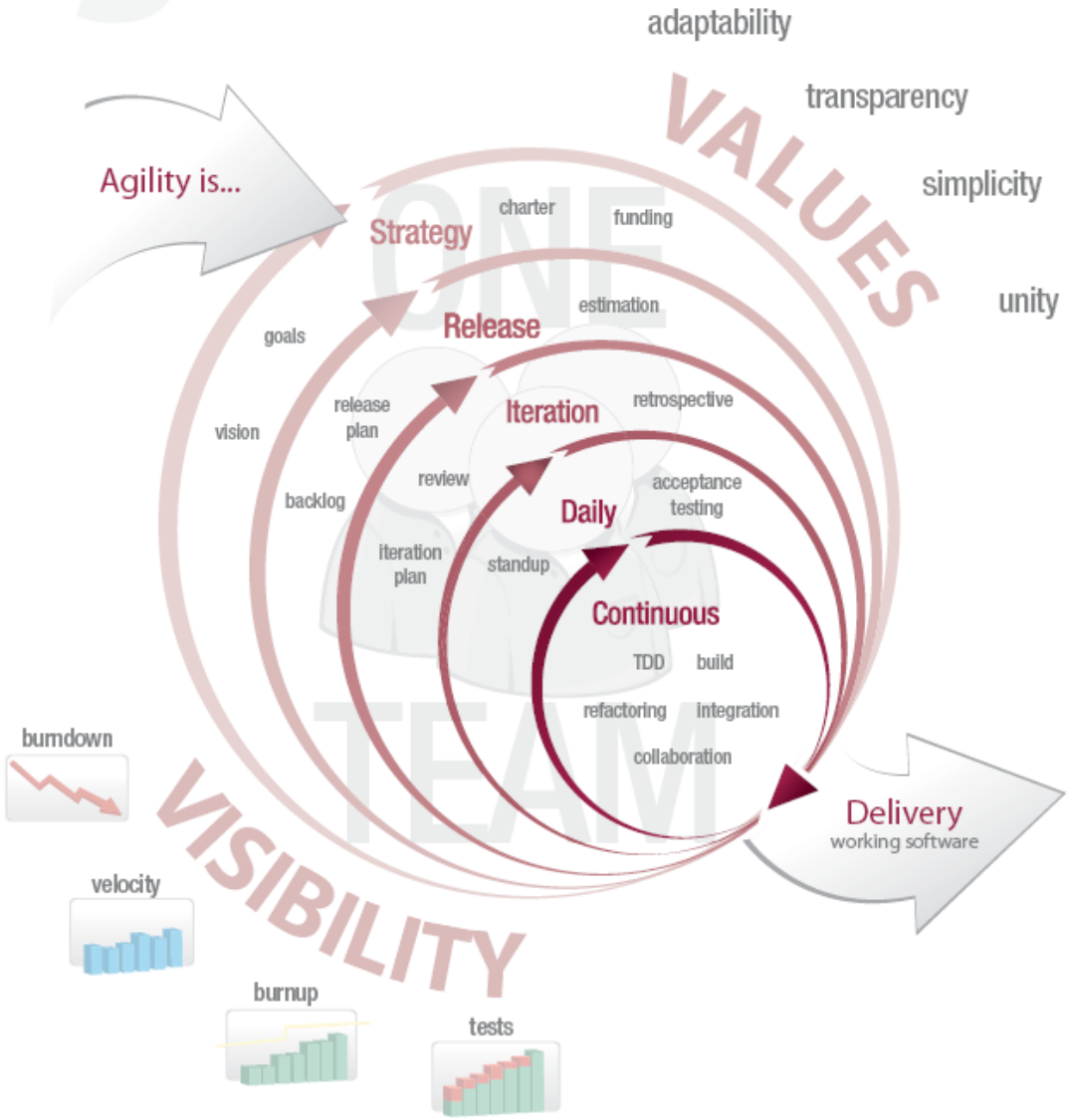
Buffer Consumption Chart



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