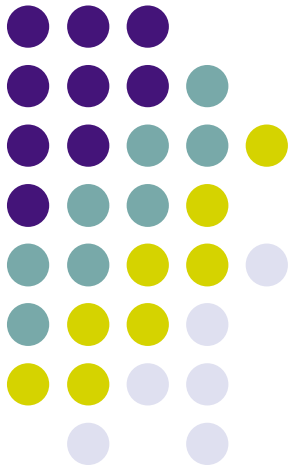
Agile Development and Extreme Programming

CSCI 5828: Foundations of Software Engineering

Lecture 24

Kenneth M. Anderson

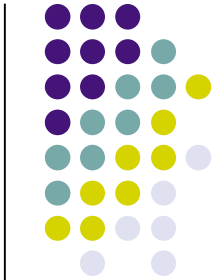


Credit where Credit is Due

- The material for this lecture is based on content from “Agile Software Development: Principles, Patterns, and Practices” by Robert C. Martin
- As such, some of this material is copyright © Prentice Hall, 2003

© University of Colorado, 2007

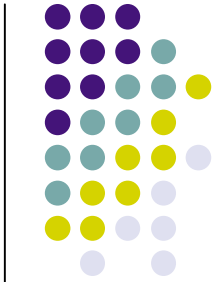
Goals for this lecture



- (Very) Briefly introduce the concepts of Agile Design and Extreme Programming
- Agile Design is a design framework
- Extreme Programming is one way to "implement" agile design
  - Other agile life cycles include SCRUM, Crystal, feature-driven development, and adaptive software development
  - See http://www.agilealliance.org/ for pointers

© University of Colorado, 2007

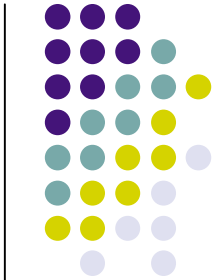
Outline



- **Background on Agile Methods**
- Extreme Programming
- Agile Perspective on Software Design

© University of Colorado, 2007

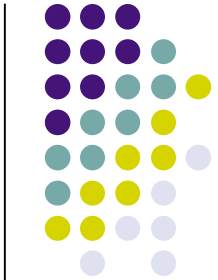
Agile Development (I)



- Agile development is a response to the problems of traditional “heavyweight” software development processes
  - too many artifacts
  - too much documentation
  - inflexible plans
  - late, over budget, and buggy software

© University of Colorado, 2007

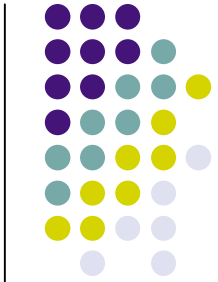
Agile Development (II)



- A manifesto (from the Agile Alliance)
  - “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

© University of Colorado, 2007

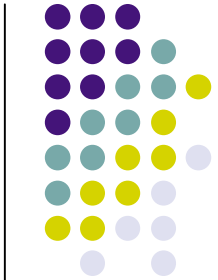
Agile Development (III)



- From this statement of values, agile development has identified twelve principles that distinguish agile practices from traditional software life cycles
- Lets look at five of them
  - Deliver Early and Often to Satisfy Customer
  - Welcome Changing Requirements
  - Face to Face Communication is Best
  - Measure Progress against Working Software
  - Simplicity is Essential

© University of Colorado, 2007

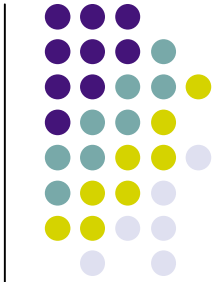
Deliver Early and Often to Satisfy Customer



- MIT Sloan Management Review published an analysis of software development practices in 2001
  - Strong correlation between quality of software system and the early delivery of a partially functioning system
    - the less functional the initial delivery the higher the quality of the final delivery!
  - Strong correlation between final quality of software system and frequent deliveries of increasing functionality
    - the more frequent the deliveries, the higher the final quality!
- Customers may choose to put initial/intermediate systems into production use; or they may simply review functionality and provide feedback

© University of Colorado, 2007

Welcome Changing Requirements



- Welcome change, even late in the project!
- Statement of Attitude
  - Developers in agile projects are not afraid of change; changes are good since it means our understanding of the target domain has increased
  - Plus, agile development practices (such as refactoring) produce systems that are flexible and thus easy to change

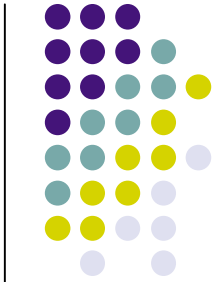
© University of Colorado, 2007

Face to Face Communication is Best

- In an agile project, people talk to each other!
  - The primary mode of communication is conversation
    - there is no attempt to capture all project information in writing
  - artifacts are still created but only if there is an immediate and significant need that they satisfy
    - they may be discarded, after the need has passed

© University of Colorado, 2007

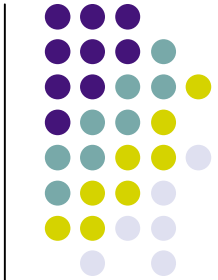
Measure Progress against Working Software



- Agile projects measure progress by the amount of software that is currently meeting customer needs
  - They are 30% done when 30% of required functionality is working AND deployed
- Progress is not measured in terms of phases or creating documents

© University of Colorado, 2007

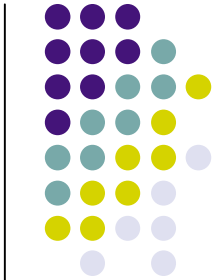
Simplicity is Essential



- This refers to the art of maximizing the amount of work NOT done
  - Agile projects always take the simplest path consistent with their current goals
  - They do not try to anticipate tomorrow's problems; they only solve today's problems
  - High-quality work today should provide a simple and flexible system that will be easy to change tomorrow if the need arises

© University of Colorado, 2007

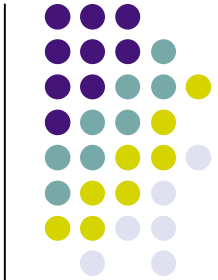
The Other Seven



- The other seven principles are
  - Deliver working software frequently
  - Stakeholders and developers work together daily
  - Build projects around motivated individuals
  - Agile processes promote sustainable development
  - Continuous attention to technical excellence and good design enhances agility
  - Agile team members work on all aspects of the project
  - At regular intervals, the team reflects on how to become more effective

© University of Colorado, 2007

Outline



- Background on Agile Methods
- **Extreme Programming**
- Agile Perspective on Software Design

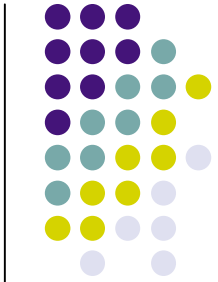
© University of Colorado, 2007

Extreme Programming

- Extreme Programming (XP) takes commonsense software engineering principles and practices to extreme levels
  - For instance
    - “Testing is good?”
    - then
    - “We will test every day” and “We will write test cases before we code”
- As Kent Beck says extreme programming takes certain practices and “sets them at 11 (on a scale of 1 to 10)”

© University of Colorado, 2007

XP Practices

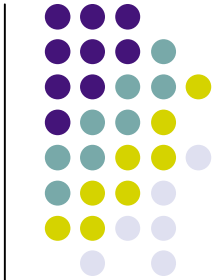


- The best way to describe XP is by looking at some of its practices
  - There are fourteen standard practices

Customer Team Member
User Stories
Short Cycles
Acceptance Tests
Pair Programming
Test-Driven Development
Collective Ownership
Continuous Integration
Sustainable Pace
Open Workspace
The Planning Game
Simple Design
Refactoring
Metaphor

© University of Colorado, 2007

Customer Team Member



- The “customer” is made a member of the development team
  - The customer is the person or group who defines and prioritizes features
  - A customer representative should be “in the same room” or at most 100 feet away from the developers
  - “Release early; Release Often” delivers a working system to the client organization; in between, the customer representative provides continuous feedback to the developers

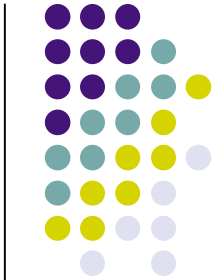
© University of Colorado, 2007

User Stories (I)

- We need to have requirements
- XP requirements come in the form of “user stories” or scenarios
  - We need just enough detail to estimate how long it might take to support this story
    - avoid too much detail, since the requirement will most likely change; start at a high level, deliver working functionality and iterate based on explicit feedback

© University of Colorado, 2007

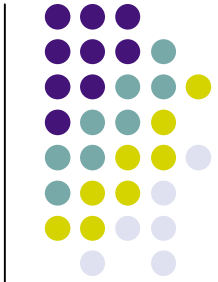
User Stories (II)



- User stories are not documented in detail
  - we work out the scenario with the customer “face-to-face”; we give this scenario a name
  - the name is written on an index card
    - developers then write an estimate on the card based on the detail they got during their conversation with the customer
- The index card becomes a “token” which is then used to drive the implementation of a requirement based on its priority and estimated cost

© University of Colorado, 2007

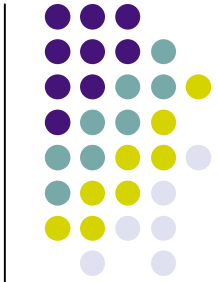
Short Cycles (I)



- An XP project delivers working software every two weeks that addresses some of the needs of the customer
  - At the end of each iteration, the system is demonstrated to the customer in order to get feedback

© University of Colorado, 2007

Short Cycles (II)



- Iteration Plan
  - The collection of user stores that will be implemented during this iteration
  - determined by a “budget” of points
  - the budget is determined by the progress made on the previous iteration
- Release Plan
  - A plan that maps out the next six iterations or so (3 months)
  - A release is a version of the system that can be put into production use

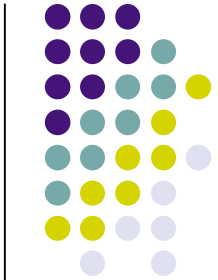
© University of Colorado, 2007

Acceptance Tests

- Details of a user story are captured in the form of acceptance tests specified by the customer
  - The tests are written before a user story is implemented
  - They are written in a scripting language or testing framework that allows them to be run automatically and repeatedly
  - Once a test passes, it is never allowed to fail again (at least for very long)
  - These tests are run several times a day each time the system is built

© University of Colorado, 2007

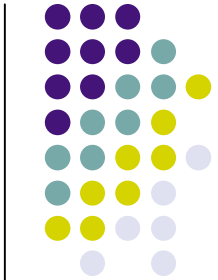
Pair Programming



- All production code is written by pairs of programmers working together at the same workstation
  - One member drives the keyboard and writes code and test cases; the second watches the code, looking for errors and possible improvements
  - The roles will switch between the two frequently
  - Pair membership changes once per day; so that each programmer works in two pairs each day
    - this facilitates distribution of knowledge about the state of the code throughout the entire team
- Studies indicate that pair programming does not impact efficiency of the team, yet it significantly reduces the defect rate!
  - [Laurie Williams, 2000] [Alistair Cockburn, 2001] [J. Nosek, 1998]

© University of Colorado, 2007

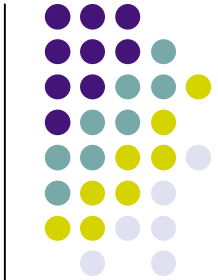
Test-Driven Development



- All production code is written in order to make failing test cases pass
  - First, we write a test case that fails since the required functionality has not yet been implemented
  - Then, we write the code that makes that test case pass
  - Iteration between writing tests and writing code is very short; on the order of minutes
- As a result, a very complete set of test cases is written for the system; not developed after the fact

© University of Colorado, 2007

Collective Ownership



- A pair has the right to check out ANY module and improve it
  - Developers are never individually responsible for a particular module or technology
- Contrast this with Fred Brook's conceptual integrity and the need for a small set of "minds" controlling a system's design
  - Apparent contradiction is resolved when you note that XP is designed for use by small programming teams; I haven't seen work that tries to scale XP to situations that require 100s or 1000s of developers

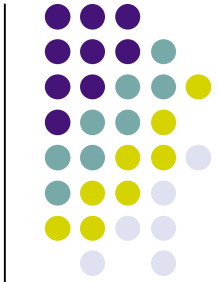
© University of Colorado, 2007

Continuous Integration

- Developers check in code and integrate it into the larger system several times a day
- Simple Rule: first one to check-in “wins”; everyone else merges
- Entire system is built every day; if the final result of a system is a CD, a CD is burned every day; if the final result is a web site, they deploy the web site on a test server, etc.
  - This avoids the problem of cutting integration testing to “save time and money”

© University of Colorado, 2007

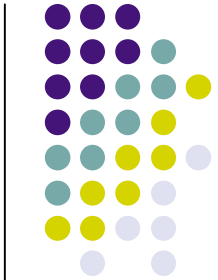
Sustainable Pace



- A software project is not a sprint; it's a marathon
  - A team that leaps off the starting line and races as fast as it can will burn out long before the finish line
  - The team must instead "run" at a sustainable pace
- An XP rule is that a team is not *allowed* to work overtime
  - This is also stated as "40 hour work week"

© University of Colorado, 2007

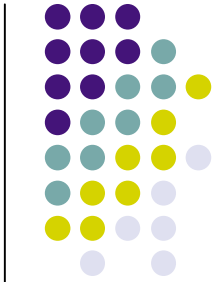
Open Workspace



- The team works together in an open room
  - There are tables with workstations
  - There are whiteboards on the walls for the team members to use for status charts, task tracking, UML diagrams, etc.
- Each pair of programmers are within earshot of each other; information is communicated among the team quickly
  - “War room” environments can double productivity
    - http://www.sciencedaily.com/releases/2000/12/001206144705.htm
- Joel on Software disagrees
  - http://www.joelonsoftware.com/items/2006/07/30.html

© University of Colorado, 2007

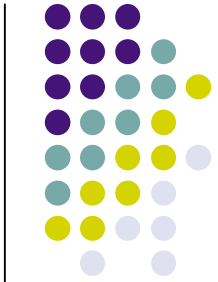
The Planning Game



- Customer decides how important a feature is
- Developers decide how much that feature costs
- At the beginning of each release and/or iteration, developers give customers a budget based on productivity of previous iteration
- Customers choose user stories whose costs total up to but do not exceed the budget
  - The claim is that it won't take long for customer and developers to get used to the system
  - and then the pace can be used to estimate cost and schedule

© University of Colorado, 2007

Simple Design



- An XP team makes their designs as simple and expressive as they can be
  - They narrow focus to current set of stories and build the simplest system that can handle those stories
- Mantras
  - Consider the Simplest Thing That Could Possibly Work
  - You Aren't Going to Need It
  - Once and Only Once (aka Don't Repeat Yourself)

© University of Colorado, 2007

Refactoring

- XP teams fight “code rot” by employing refactoring techniques constantly
  - They have the confidence to do this because they also use test-driven design
  - By “constantly” we mean every few hours versus “at the end of the project”, “at the end of the release”, or “at the end of the iteration”

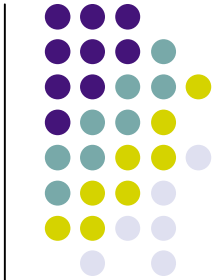
© University of Colorado, 2007

Metaphor (I)

- The big picture that ties the whole system together
  - Vocabulary that crystallizes the design in a team member's head

© University of Colorado, 2007

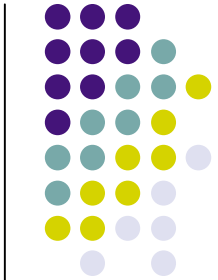
Metaphor (II)



- Example
  - A system that transmits text to a screen at 60 chars per second; programs write to buffer, when buffer full, programs are suspended, when buffer empty, programs are activated
    - Metaphor: Dump Trucks Hauling Garbage
    - Screen = “Garbage Dump”, Buffer = “Dump Truck”, Programs = “Garbage Producer”

© University of Colorado, 2007

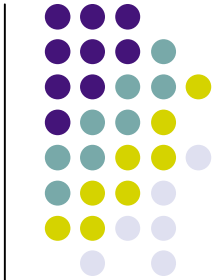
Metaphor (III)



- Example
  - network traffic analyzer, every 30 minutes, system polled dozends of network adapters and acquired monitoring data; Each adaptor provides block of data composed of several variables
    - Metaphor: A toaster toasting bread
    - Data Block = “Slices”
    - Variables = “Crumbs”
    - Network analyzer = “The Toaster”
    - Slices are raw data “cooked” by the toaster

© University of Colorado, 2007

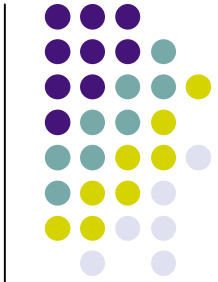
Benefits of XP



- Customer Focus
- Emphasis on teamwork and communication
- Programmer estimates before implementation
- Emphasis on responsibility for quality
- Continuous measurement
- Incremental development
- Simple design
- Frequent redesign via refactoring
- Frequent testing
- Continuous reviews via pair programming

© University of Colorado, 2007

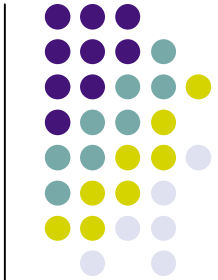
Criticisms of XP



- Code centered vs. Design centered
  - Hurts when developing large systems
- Lack of design documentation
  - Limits XP to small systems
- Producing readable code is hard
- Code not good as serving as documentation (listings can run to 1000s of pages)
- Lack of structured inspection process (can miss defects)
- Limited to narrow segment of software application domains
- Methods are only briefly described
- Difficult to obtain management support
- Lack of transition support (how do you switch from waterfall or other process?)

© University of Colorado, 2007

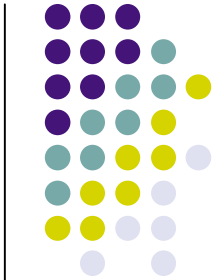
Outline



- Background on Agile Methods
- Extreme Programming
- **Agile Perspective on Software Design**

© University of Colorado, 2007

Agile Methods: Perspective on Design



- Agile methods have an approach to software design that includes
  - identifying aspects of bad design
  - avoiding those aspects via a set of principles

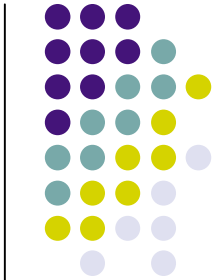
© University of Colorado, 2007

Bad Design

- Rigidity: The design is hard to change
- Fragility: The design is easy to break
- Immobility: The design is hard to reuse
- Viscosity: It is hard to do the right thing
- Needless Complexity: Overdesign
- Needless Repetition: Copy and Paste
- Opacity: Disorganized Expression

© University of Colorado, 2007

Good Design Principles



- The Single Responsibility Principle
- The Open-Closed Principle
- The Liskov Substitution Principle
- The Dependency Inversion Principle
- The Interface Segregation Principle

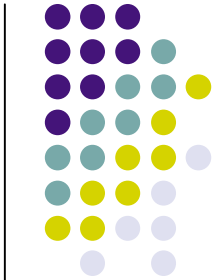
© University of Colorado, 2007

Single Responsibility Principle

- A class should have only one reason to change
  - If a class has more than one responsibility, the responsibilities can become coupled
    - changes to one can impact the other
  - If a class has a single responsibility, you can limit the impact of change with respect to that responsibility to this one class
- Example
  - Class Rectangle with draw() and area() methods
    - draw() is only used by GUI apps, separate these responsibilities into different classes

© University of Colorado, 2007

Open-Closed Principle



- Discussed previously in Lecture 20
- A class should be open to extension but closed to modification
  - Allows clients to code to class without fear of later changes

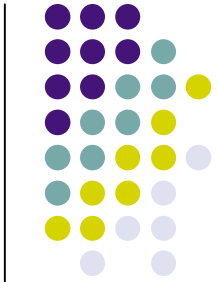
© University of Colorado, 2007

Liskov Substitution Principle

- Also discussed in lecture 20
- Subclasses need to respect the behaviors defined by their superclasses
  - if they do, they can be used in any method that was expecting the superclass
- If a superclass method defines pre and post conditions
  - a subclass can only weaken the superclass pre-condition, and can only strenghen the superclass post-condition

© University of Colorado, 2007

Example (I)

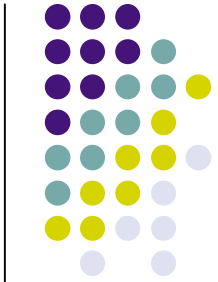


```
class Rectangle {
  private int width;
  private int height;
  public void setHeight(int h) { height = h }
  public void setWidth(int w) { width = w }
}
```

- postcondition of setHeight?
- postcondition of setWidth?

© University of Colorado, 2007

Example (II)

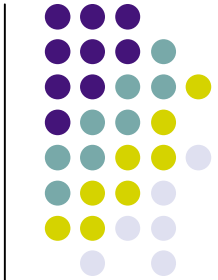


```
class Square extends Rectangle {
   public void setHeight(int h) {
      super.setHeight(h)
      super.setWidth(h)
   }
   public void setWidth(int w) {
      super.setWidth(w)
      super.setHeight(w)
   }
}
```

Does this maintain the postconditions of the superclass?

© University of Colorado, 2007

Example (III)



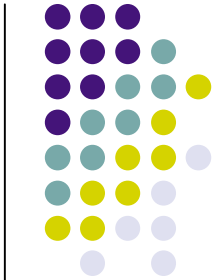
- Let's check

```
public void checkArea(Rectangle r) {
  r.setWidth(5);
  r.setHeight(4);
  assert (r.area() == 20);
  // fails when Square is passed
}
```

- Whoops!

© University of Colorado, 2007

Example (IV)



- In order to achieve the Liskov substitutability principle, Square's methods needed to have equal or stronger postconditions than its superclass
  - For setWidth:
    - Rectangle: width == w && height == old.height
    - Square: width == w && height == w
  - The postcondition for Square is weaker than the postcondition for Rectangle because it does not attempt to enforce the clause (height == old.height)

© University of Colorado, 2007

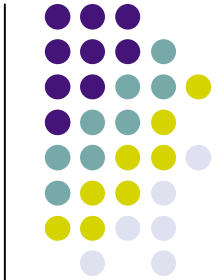
Dependency-Inversion Principle

- High-level modules should not depend on low-level modules; Both should depend on abstractions

- In response to structured analysis and design, in which stepwise refinement leads to the opposite situation
  - high-level modules depend on lower-level modules to get their work done

© University of Colorado, 2007

Why “inversion”?

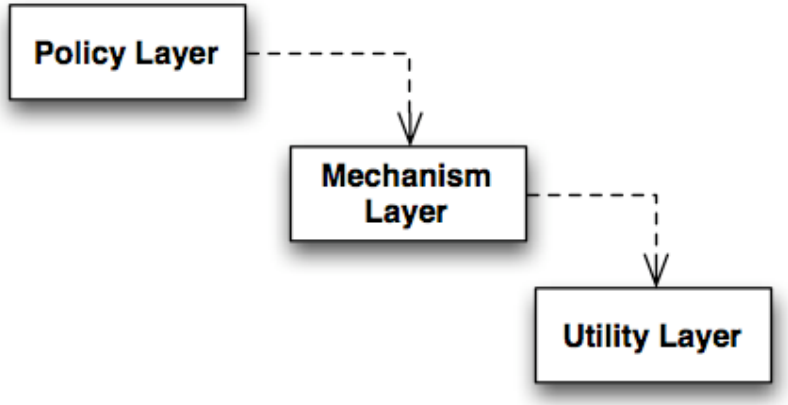


- DIP attempts to “invert” the dependencies that result from a structured analysis and design approach
  - High-Level modules tend to contain important policy decisions and business rules related to an application; they contain the “identity” of an application
    - If they depend on low-level modules, changes in those modules can force the high-level modules to change!
    - High-level modules should not depend on low-level modules in any way

© University of Colorado, 2007

Solution (I)

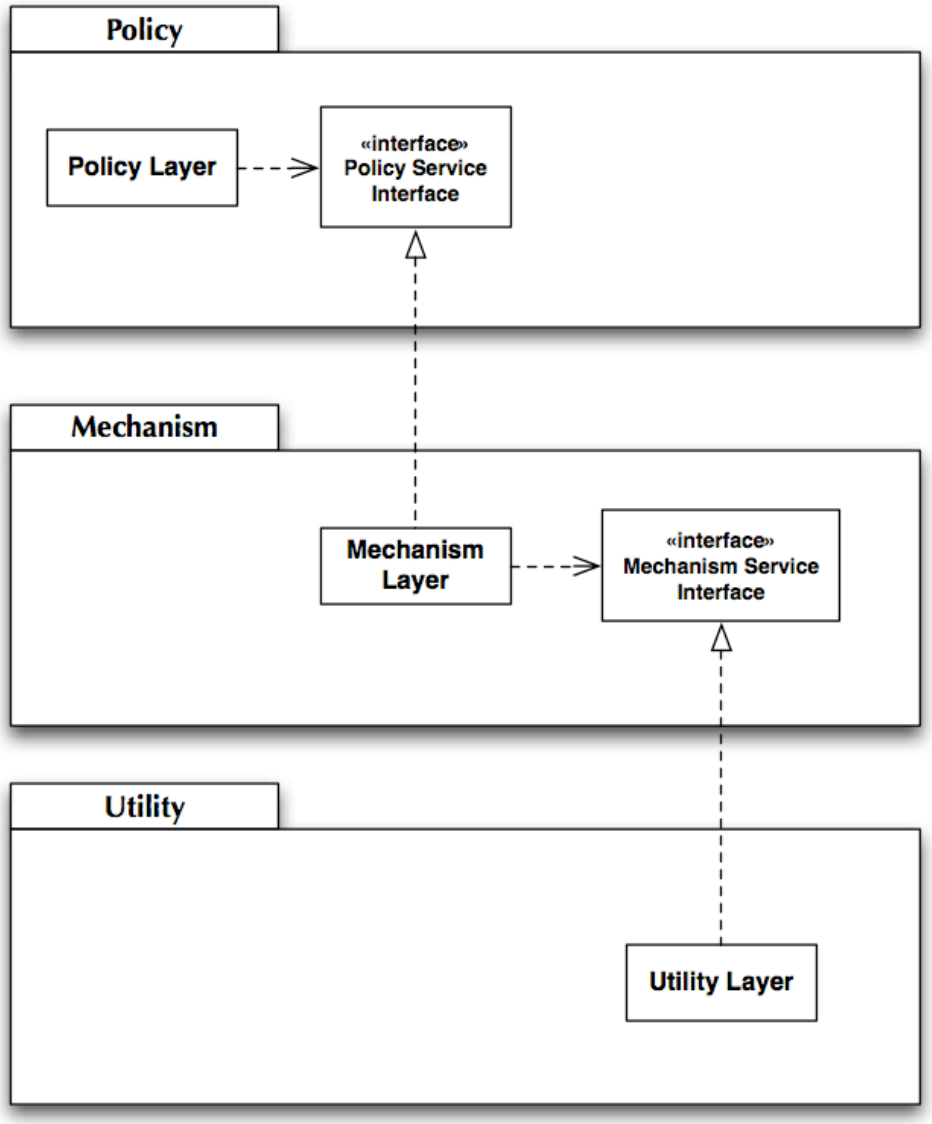
- In a layered system, a “higher” layer defines an interface that lower layers implement
- Typical Layered System



© University of Colorado, 2007

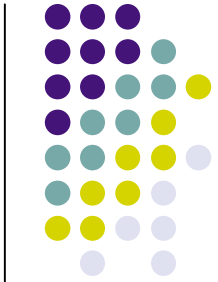
Solution (II)

Layered System
with DIP applied



© University of Colorado, 2007

Inversion of Ownership



- Its not just an inversion of dependency, DIP also inverts ownership
  - Typically a service interface is “owned” or declared by the server, here the client is specifying what they want from the server

© University of Colorado, 2007

Depend on Abstraction

- A simple heuristic based on DIP
  - No variable should reference a concrete class
  - No class should derive from a concrete class
  - No method should override an implemented method of any of its base classes
- Note: this is just a heuristic, not something that can be universally applied
  - but its use can lead to lower coupling within a system

© University of Colorado, 2007

Interface Segregation Principle

- Clients should not be forced to depend on methods that they do not use
- Concerns classes with “fat” interfaces
  - what we’ve been calling a non-cohesive module
- A class with a “fat” interface has groups of methods that each service different clients
  - This coupling is bad however, since a change to one group of methods may impact the clients of some other group

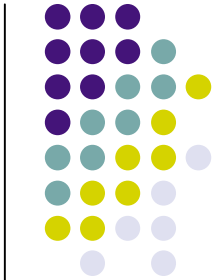
© University of Colorado, 2007

Example (I)

- Security System with Doors that can be locked and unlocked
  - One type of Door is a TimedDoor that needs to sound an alarm if its left open too long
  - Timing is handled in the system via a Timer class that requires its clients to implement an interface called TimerClient
  - TimedDoor needs to make use of Door and TimerClient

© University of Colorado, 2007

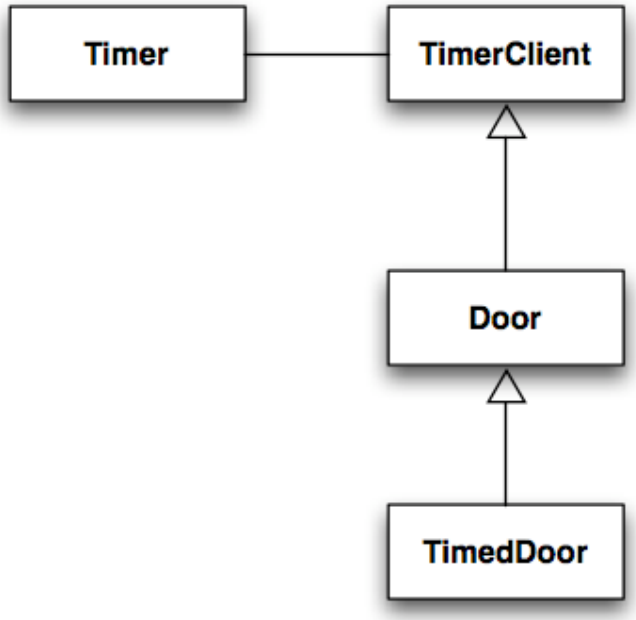
Example (II)



- A naïve solution would be to have the Door interface extend the TimerClient interface
  - This would allow TimedDoor to have access to the routines in TimerClient that would then allow it to interact with the Timer class
- Unfortunately, this “pollutes” the interface of Door with methods that only one of its subclasses needs
  - All other door types get no benefit from the TimerClient methods contained in the Door interface

© University of Colorado, 2007

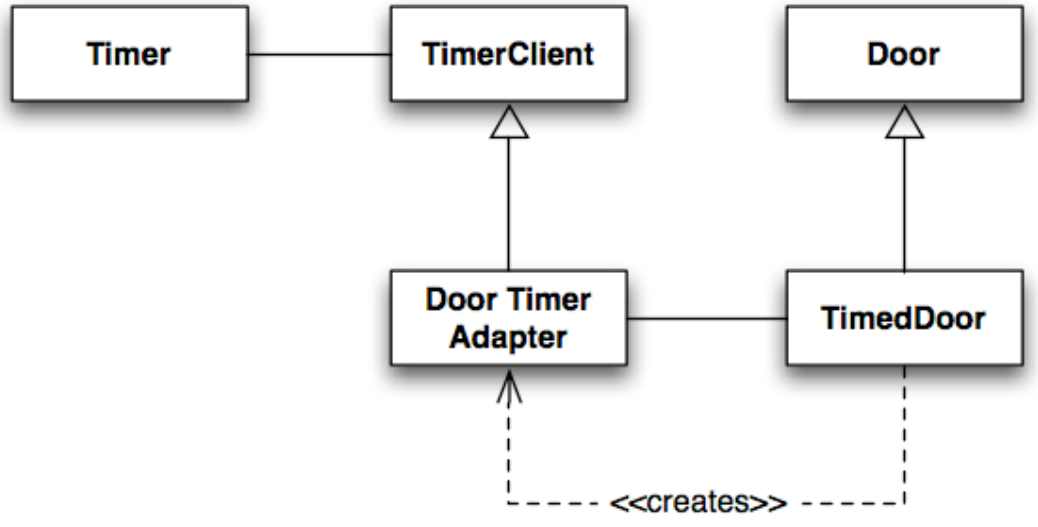
Example (III)



Here, Door is made to extend TimerClient so that TimedDoor can interact with the Timer object. Unfortunately, now all subclasses of Door depend on TimerClient even if they don't need Timing functionality!

© University of Colorado, 2007

Example (IV)



The solution is to keep the interfaces separate and make use of the adapter pattern to allow TimedDoor access to Timer and its functionality; note, here that this solution keeps TimedDoor independent of Timer and TimerClient

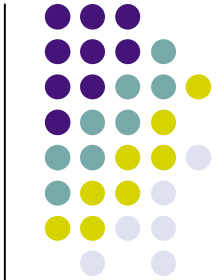
© University of Colorado, 2007

Summary

- Agile design is a process, not an event
  - It's the continuous application of principles, patterns, and practices to improve the structure and readability of software
- Agile Methods (of which XP is one) are a response to traditional software engineering practices
  - They deemphasize documents/processes and instead value people and communication

© University of Colorado, 2007

Coming Up Next



- Test-driven Design
- Refactoring

© University of Colorado, 2007