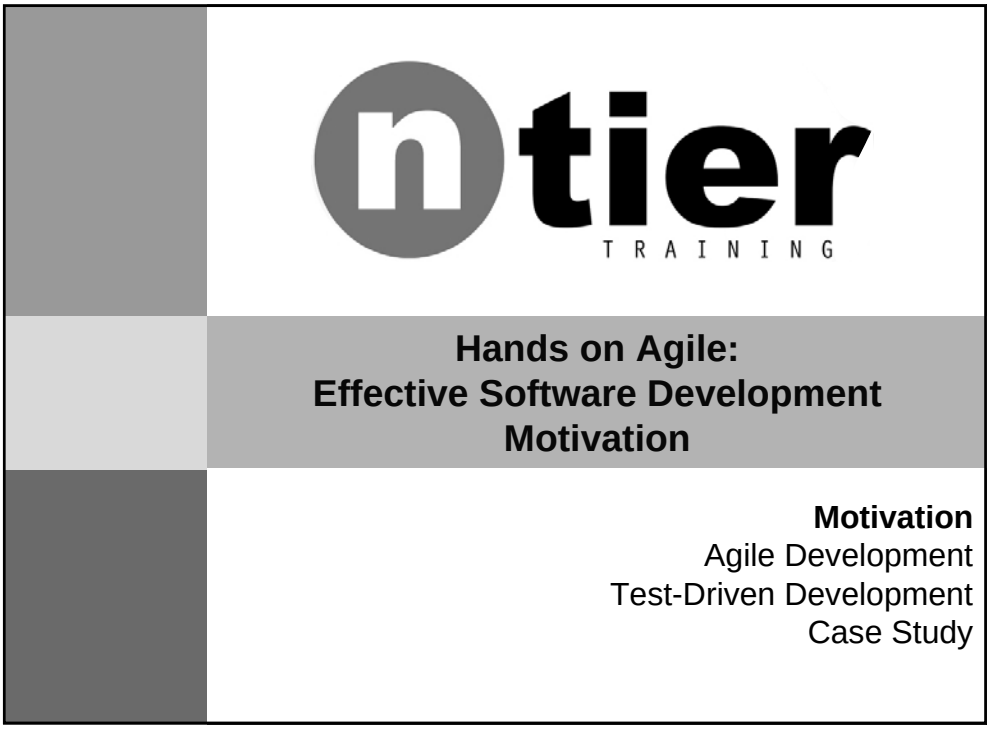
Objectives

- Understand the motivation for improving our development practice
- Learn the basic principles and practices of agile development
- Learn about test-driven development and outline its benefits
- Learn a technique for integrating agile and TDD in your daily practice
- Examine a case study which brings together the OO First Principles, UML, design patterns, TDD, and agile

 3



Facing up to Our Failures

- **The Chaos Report (1995)**
  - The Standish Group research shows a staggering **31.1%** of projects will be cancelled before they ever get completed
    - An estimated $81 BILLION for cancelled software projects (still blame them for off-shoring?)
    - Further results indicate that 52.7% of projects will cost over 189% of their original estimates
  - Only **16.2%** of software projects are completed on-time and on-budget
    - In larger companies, the news is even worse: only **9%** of their projects come in on-time and on-budget
    - Even when successful projects are completed, many are no more than a mere shadow of their original specification requirements

 5

Reasons for Project Cancellation

1. **Incomplete requirements**
2. **Lack of user involvement**
3. Lack of resources
4. Unrealistic expectations
5. Lack of executive support
6. Changing requirements and specifications
7. Lack of planning
8. Didn't need it any longer
9. Lack of IT management
10. Technology illiteracy
11. Other

Topping our list: **incomplete requirements** and lack of user involvement

http://www.spinroot.com/spin/Doc/course/Standish_Survey.htm

 6

Does OOP Solve Our Problems?

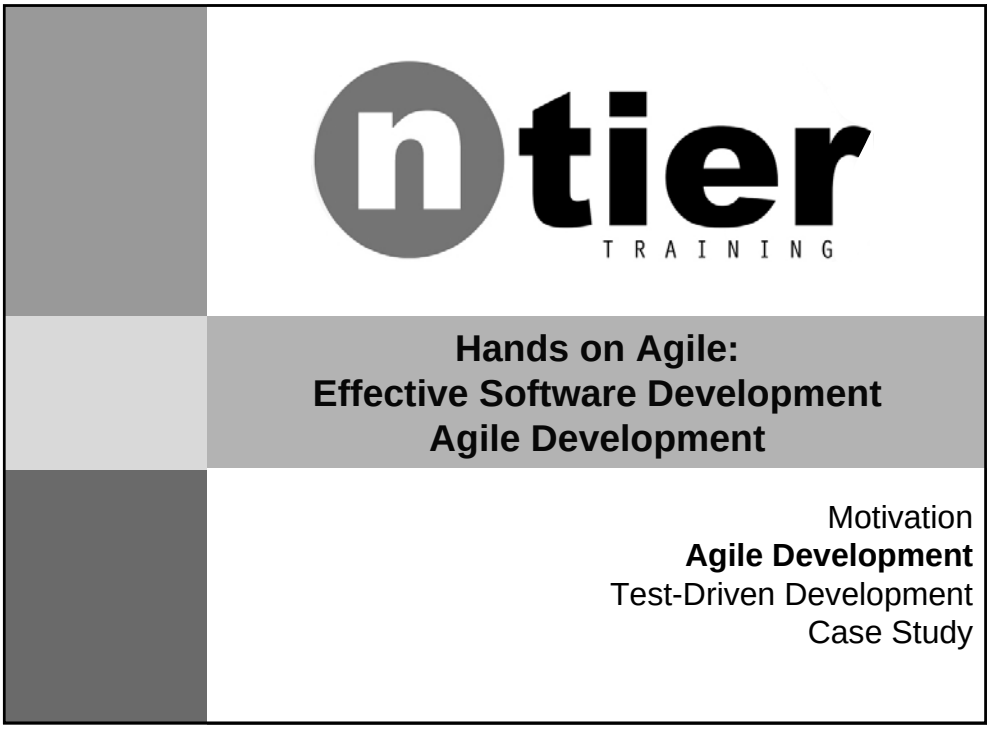
- What percentage of OOP projects fail?
- How many OOP books are there?
- How long does it take to learn OOP?
- How many people have you met who are really good at it?
- OO languages do not require OO code
  - Give you the ability to write better software
  - Don't force you to do it
  - When historians look back, a lot of us will get credit for inventing "procedural Java"

 7

OOP Can Yield Competitive Advantage

- As Thomas Friedman describes in his (must read) book, "The World is Flat," IT is about competitive advantage
- "The ROI for OO comes when you can:
  - ***quickly respond to your customer's changes*** and
  - ***lower their total cost of ownership***, thereby giving them a
  - ***competitive advantage*** in the marketplace."
    Ed Lance © 2001
- ***KEY: quickly respond to requirements changes***
- Okay, I buy it – but what do I **do**?

 8



Process Control - Defined vs. Empirical

- It is typical to adopt a plan-driven, well-defined approach when the underlying mechanisms are well understood
- What happens if we think they are well understood and later it turns out that we did not understand them at the outset at all?
- What happens when requirements start changing on us?
  - "Hey, that wasn't in the plan..."

Origins of Agile Software Development

- Agile developed as a reaction to process-driven methodology
  - For many, the appeal is the rebuttal to the bureaucracy involved
- Compromise between no process and too much process
  - Just enough process to gain a reasonable payoff
- Significant differences in emphasis from engineering methods
  - **Less document-oriented**
  - In many ways, rather code-oriented

 11

Characteristics of Agile Software Development

- **Adaptive** rather than **predictive**
  - Regular adaptation to changing circumstances
- Two common denominators:
  - **Time-boxed**
  - **Belief that all requirements can't be captured up front**
- **People-oriented** rather than **process-oriented**
  - Close, daily cooperation between business people and developers
  - Face-to-face conversation is best
  - Projects built around motivated individuals – who should be trusted
  - Self-organizing teams

 12

Project Vision Is the Driver

- Waterfall approach
  - Estimate cost and schedule
  - Fix "features"
- Plan-driven

- Agile approach
  - Fix costs and schedule
  - Estimate "features"
- Vision-driven

The **plan** creates costs/schedule estimates

The **vision** creates feature estimates

 13

Agile Development Methods

- There are many agile development methods – Scrum, XP, etc.
- Most minimize risk by developing software in **short intervals**
  - A time box, or ***iteration***, typically lasts one to four weeks
- Each iteration includes **planning**, **requirements analysis**, **design**, **coding**, **testing**, and **documentation**
  - An iteration may not add enough functionality to warrant releasing the product to market
    - Goal is to have an available release (without bugs) after an iteration
- After each iteration, team reevaluates project priorities
  - Usually with the customer

 14

The Agile Manifesto

- In 2001, a group got together in Snowbird, Utah to discuss the growing field of what were then called "lightweight methods"
- The term ***agile*** was used for this new breed of methods
- They wrote the ***Manifesto for Agile Software Development***, setting out the values and principles of these agile processes
- ***http://agilemanifesto.org***

 15

Agile Manifesto Mission Statement

- We are uncovering better ways of developing software by doing it and helping others to do it
- Through this work we have come to value:

1. Individuals and interactions	over processes and tools
2. Working software	over comprehensive documentation
3. Customer collaboration	over contract negotiation
4. Responding to change	over following a plan

- While there is value in the items on the right, we value more the items on the left (the ones in bold)

http://agilemanifesto.org

 16

Agile Manifesto Principles

- Highest priority is to satisfy customer through **early and continuous delivery** of valuable software
- **Welcome changing requirements**, even late in development
  - Agile processes harness change for the customer's competitive advantage
- **Deliver working software frequently**, from weeks to months, with preference to shorter timescales

http://agilemanifesto.org

 17

Agile Manifesto Principles

- **Working software is the primary measure of progress**
- Promote **sustainable development**
  - Sponsors, developers, and users should be able to maintain a constant pace indefinitely
- Continuous attention to **technical excellence and good design**
- Simplicity, the art of **maximizing the amount of work NOT done**

http://agilemanifesto.org

 18

It's All about Communication

- The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**
- Business people and developers must **work together daily** throughout the project

Even Journalism Is Becoming Agile

- Blog-style journalism is contributing to the decline of the printed newspaper
  - It's agile journalism!
- Old school paper journalism:
  - Assess need/desire for story
  - Hit the streets, research it
  - Write it up
  - Submit to editor
  - Revise
  - Print
  - Story is "done" (in the past)
- Blog-style online journalism:
  - Hit streets, research story
  - Write it up and push it out
    - Faster time to market
  - Reader comments come in
    - Adding relevant info
    - Refuting facts or evidence
    - Comments on comments
  - Learn and adjust as you go
  - Story is ongoing, "alive"

A Few Final Points

- Agile development is **not**:
  - Sloppy
  - "Cowboy coding"
  - "Winging it"
  - Undocumented chaos
- Agile can fail – anything new requires UAS:
  - **U**nderstanding
  - **A**cceptance
  - **S**upport
- Which form of Agile is best for me?
  - Yacht sales metaphor

 21

Hands on Agile: Effective Software Development Test-Driven Development

Motivation
Agile Development
Test-Driven Development
Case Study

Test-Driven Development - Broadly Defined

- ***Test-driven development*** (TDD) is a technique in which testing the production code is of primary importance
- Typically, the tests are written **before** the production code
- Code is not considered done until all tests pass
- All requirements are captured in tests
  - When the code passes the tests, you have satisfied the requirements
  - However, you may (**will**) still refactor your code to make it better

 23

What Does TDD Really Mean?

- **Test-Oriented** Development
  - All design up front
  - You have lots of tests, but you don't always write them first
- **Test-Driven** Development
  - All design up front
  - You write tests before you write code
- Test-Driven **Design** (XP)
  - Test-first approach as a design technique – **no** design up front
  - The tests are a bonus, but the idea is to design as you go

 24

What Does TDD Really Mean?

- Test-Driven **Development and Design** (nTier's philosophy)
  - Some initial design up front
  - Test-first approach to drive new code and changes
  - Design will evolve, because you learn from the tests, the tests can point out various code smells, etc.
    - It may also change to make the code more testable
- What does TDD mean at your organization?

 25

TDD Toolset

- Test-driven development relies on an automated testing tool
  - Since tests run so often, need rapid response to small changes
- Testing framework needs to make writing tests easy
- Most Java developers use ***JUnit***
- **Kent Beck** was one of the TDD pioneers, and created JUnit
  - He also wrote a book: ***Test-Driven Development by Example***

 26

Kent Beck's TDD Metaphors

- Driving
  - Driving is not about getting the car pointed in the right direction
  - Driving is about constant adjustment...a little this way, a little that way, forever...you always pay attention...you are always prepared to change things
- Pulling a bucket of water from a well
  - When the bucket is small, a free-spinning crank is fine
  - When the bucket is big and gets full, you get tired on the way up
  - You need a ratchet mechanism that enables you to rest between bouts of cranking
    - The heavier the bucket, the closer the teeth need to be on the ratchet
  - The tests in TDD are the teeth of the ratchet
    - The tougher the programming problem, the smaller the steps

 27

Test-Driven Development Cycle - Overview

1. Add a test
2. Run all tests and see the new one fail **RED**
3. Write some code – **"make it work"**
4. Run all tests and see them all succeed **GREEN**
5. Refactor the code – **"make it right"** **REFACTOR**
   Sometimes your refactoring breaks something ~~**REFACTOR**~~
6. Repeat
   Sometimes you repeat only step 5

 28

Benefits of TDD

- Provides concrete evidence that the software works
  - ***"If the bar is green, the code is clean"***
- Shortens the programming loop
  - Enables (and encourages) you to take small steps when coding
  - Provides rapid, fine-grained, concrete feedback (every few mins.)
    - If you break something, find out immediately
    - Encourages experimentation
- Helps you write cohesive code – small steps means small methods – that are **easily callable and testable**
  - By focusing on the tests first, you must imagine how the functionality will be used by clients (think like a client)
  - **Focus first on the interface**, then the implementation

 29

Benefits of TDD

- You are forced to think of the software in terms of **small units** that can be written and tested **independently**
  - This leads to smaller, more focused classes (cohesion), better coupling, and cleaner interfaces
- Your designs must consist of **highly cohesive, loosely coupled** components, just to make testing easier
  - Great news! This also makes evolution/maintenance of the system easier, e.g., when handling new/changed requirements
- Many TDD developers claim to have reduced their use of the debugger to zero (or close to it)
  - Debugging is the antithesis of automated testing – requires extensive manual inspection of variables, call stacks, etc.

 30

Benefits of TDD

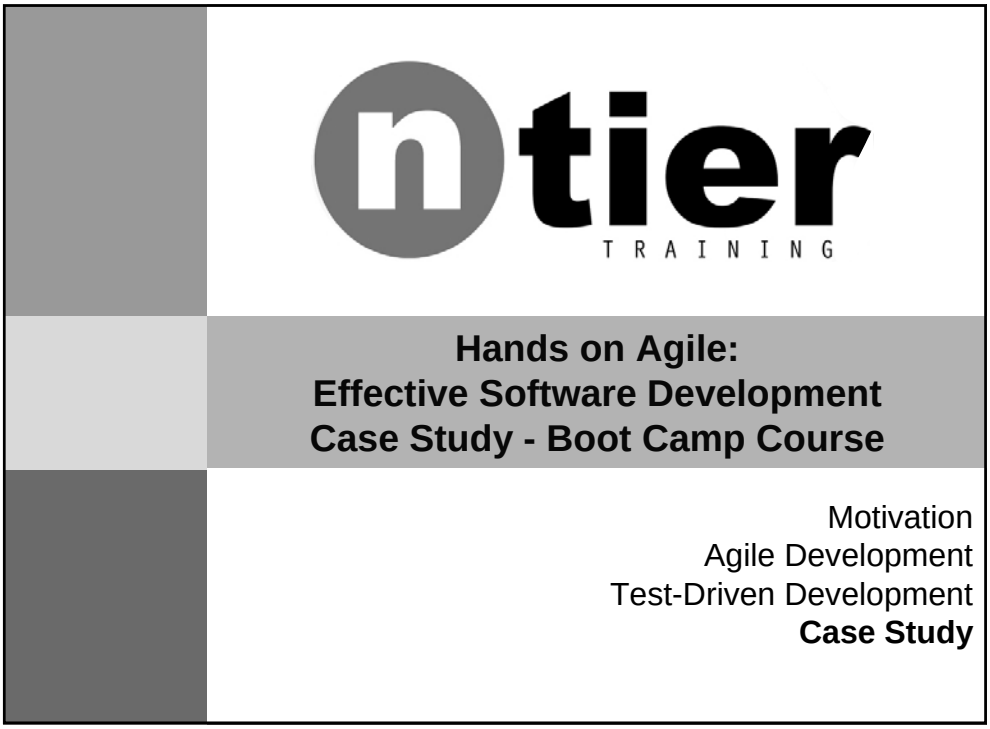
- Writing tests first implies **100% code coverage**
- When you have to write tests first for everything you do, you are far **less likely to add extraneous capabilities**
  - Helps to control developer-driven scope creep
  - *"Maximize the amount of work NOT done"*
- Code-based documentation
  - Tests automatically provide **sample client code**
    - Most programmers prefer to learn the basics of a class or operation from sample code than from written documentation
  - Tests not meant to provide complete documentation
    - But they do form an important part of it

 31

Benefits of TDD

- Taking small steps (get to green as quickly as possible) points out redundancies and other code smells
  - This allows you to have a clear initial direction as to what to refactor
- Shows you ways that your **design should evolve**
  - You see too much coupling in the client
  - You see better APIs than originally designed
  - You see things in the code that indicate when a design pattern(s) might be appropriate
  - Remember Beck: constant adjustment...a little this way, a little that way...you always pay attention...you are always prepared to change things

 32



OO First Principles

Code to the interface, not the implementation

Find what varies and encapsulate it

Hide by convention, reveal by need

What you hide you can change

Favor composition over inheritance

Open-closed principle

The OOP Equation

OOP = Principles + Practices

OO Principles	Patterns	OO Practices
Coupling	GoF	Intentionally couple
Cohesion	J2EE patterns	Code to the interface
Redundancy		Hide by convention
Encapsulation		Separate instantiation from implementation
Testable		Favor composition
		Encapsulate variation
		Refactor regularly
		Code by intention
		Remove redundancy
		Practice testing

Encapsulation

- Hiding: **data** and **type**
- Motivation: easier to change

First Principle

Code to the interface, not the implementation

- An "interface" can mean **public methods**, **superclass**, **abstract class**, or **Java interface**, depending on context

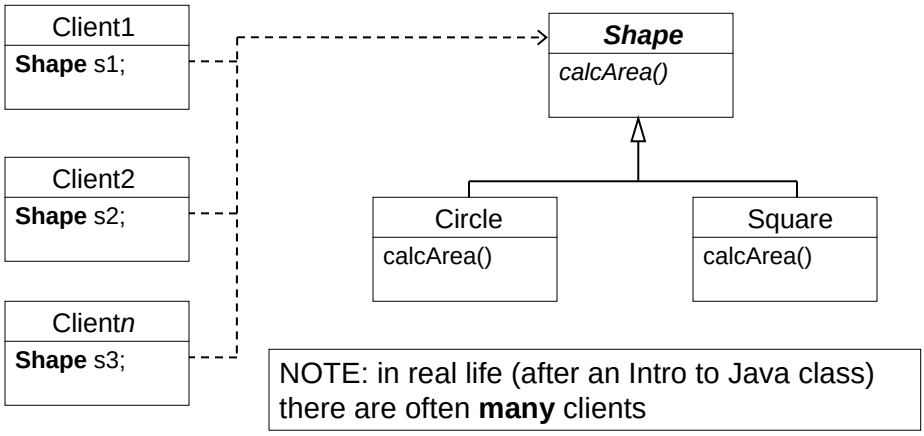
First Principle

Hide by convention, reveal by need

- What you hide, you can change

Polymorphism

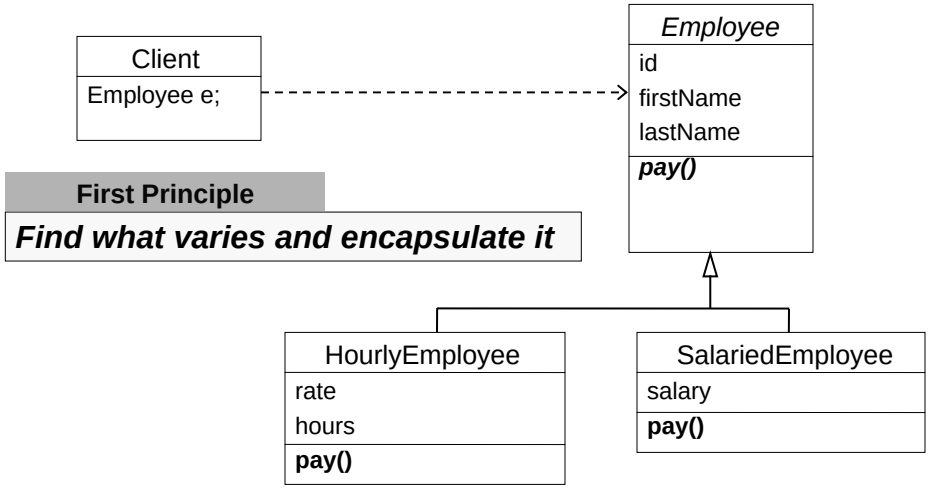
- Definition: treating many things (types) as one thing (type)
- Example: treating Circle and Square as Shape
  - Decouple clients from specific shape types



 37

Inheritance and Encapsulation

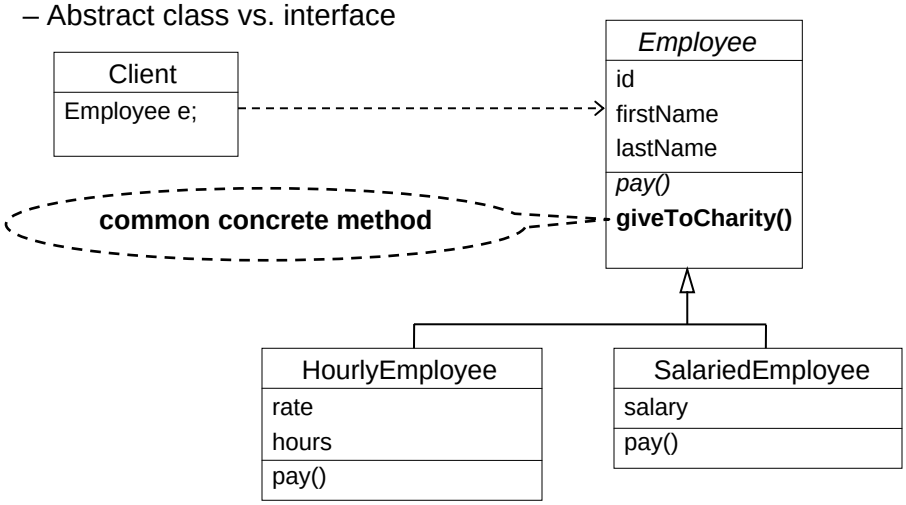
- **Use inheritance to encapsulate what varies**



 38

Inheritance and Commonality

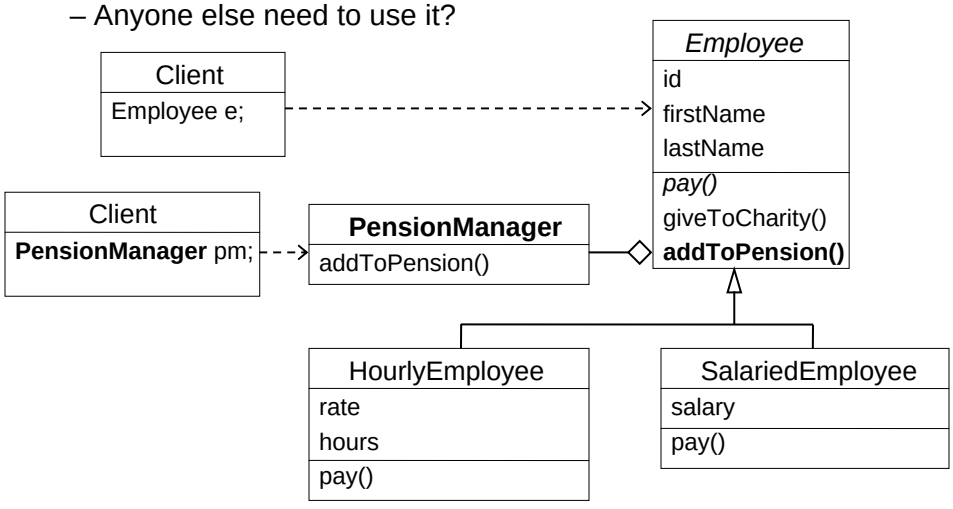
- **Use commonality to avoid redundancy**
  - Abstract class vs. interface



 39

Composition and Delegation

- **HAS-A vs. IS-A**
  - Anyone else need to use it?



 40

Erich Gamma on Learning OO

- I think patterns as a whole can help people learn object-oriented thinking: how you can leverage polymorphism, design for composition, delegation, balance responsibilities, and provide pluggable behavior...
- You really learn about polymorphism when you've understood the patterns. So patterns are good for learning OO and design in general...

http://www.artima.com/lejava/articles/gammadpP.html

 41

Boot Camp Project - Just in Time Shipping (JITS)

- JITS is software for a mail kiosk
- First models will be in lobbies of post offices for use during busy times or when the main part of the post office is closed
- You can mail boxes or letters
- We'll be "discovering" new requirements as we go

 42

Our Approach in the Boot Camp Project

- We will use an agile development technique known as ***test-driven development (TDD)***
  - This involves writing the tests before writing the code
  - But there's more to it than that
- Coding will be done in **short iterative cycles**
  - Forces you to get good code written faster (it really works)
  - Also serves to measure your progress
  - Likely requires an adjustment to how you normally do things
- You will make mistakes...and you will learn from them
- You will learn from us, and each other – and we from you

 43

TDD+Agile as a Coding Practice

- You've been working with TDD
- You've heard about agile development
- How can you put them together in practice when coding?
- The **D-T-C-R** code cycle **D**o **T**he **C**ode **R**ight

Design	not exhaustively, the design will evolve
Test	practice coding by intention
Code	make it work
Refactor	make it right

- This is a **15-20 minute** time box, at the end of which:
  - You should have something working
  - *"Delivery working software frequently"*

 44

Improving Your Estimates

- You will estimate how many cycles you need to do each lab
  - You are not expected to know this number exactly
  - *"Agile is adaptive rather than predictive"*
- We will measure your progress after each cycle
  - *"Working software is the primary measure of progress"*
  - You can revise your estimate after each cycle
- The customer (instructors) is in the room
  - *"Close cooperation between business people and developers"*
- The project should be moving forward at all times
  - *"Promote sustainable development"*

 45

blue page

 46

Erich Gamma on Teaching Design Patterns

- The other question was how we should teach patterns. Not that I know exactly what you should do, but I think what you should *not* do is have a class and just enumerate the 23 patterns. This approach just doesn't bring anything.
- You have to feel the pain of a design which has some problem. I guess you only appreciate a pattern once you have felt this design pain.
- It's an eye opener to realize that oh, actually this pattern, factory or strategy, is a solution to my problem. And I think that's the really interesting way to teach.

http://www.artima.com/lejava/articles/gammadpP.html

 47

Erich Gamma on Teaching Design Patterns

- When I started teaching it was really boring, because I was just enumerating the patterns. I found it far more interesting to try to motivate with real examples how to apply patterns.
- In other words, you really need to present the problem in a realistic context – synthetic examples do not fly.
- This is definitely the way I'd recommend that people use patterns. Do not start immediately throwing patterns into a design, but use them as you go and understand more of the problem.
- Because of this I really like to use patterns after the fact, refactoring to patterns.

http://www.artima.com/lejava/articles/gammadpP.html

 48

Section Review

- OO by itself is not enough
- Good OO is principle-driven

- Agile development:
  - Unites customer and developers
  - Embraces change
  - Focuses on the delivery of software above all else

- Once you go TDD, you won't go back
- TDD + Agile = D T C R
  - Success is daily – but it's also every 20 minutes

- OO principles and design patterns are best taught in context

End

- This slide is intentionally devoid of any useful content

