


[Dashboard](#) > [Training](#) > [...](#) > [Object Boot Camp](#) > [Introduction](#)

Search

ThoughtWorks®

Training

Welcome [Luca Minudel](#) | [History](#) | [Preferences](#) | [Log Out](#) 

Introduction

[View](#)[Attachments \(1\)](#)[Info](#)[Review](#) [Browse Space](#)Added by [Rolf Russell](#), last edited by [Elizabeth Keogh](#) on Jan 15, 2008 ([view change](#))Labels: (None) [EDIT](#)

Introduction

Session Objectives

By the end of this session students should ...

- Understand that Object Boot Camp is taught in the Socratic Method and focuses on hands-on labs.
- Decide a class contract to follow throughout OBC
- Understand basic OO terminology and coding standards
- Be able to define the job of a class
- Understand encapsulation

Session Overview

Activity

[Learning Objectives](#)
[Introductions](#)
[Brainstorm Class](#)
[Contract](#)
[Shape of the Class](#)
[Vocabulary](#)
[Overview of the Class](#)
[Object Principles](#)
[How Object Oriented Programming Evolved](#)
[Job](#)
[Lab - Rectangle](#)
[Encapsulation in Rectangle](#)
[TDD Conversation / Demo](#)
[Pairing Conversation](#)

Session Notes

Learning Objectives

Welcome to the first day of Object Bootcamp!

Over the course of the next X weeks, we'll discuss and code a number of different object-oriented programming principles and practices. Our goal is to work together to make ourselves better programmers who build systems that are simpler, more maintainable and more defect-free. Object Bootcamp teaches a specific way to think about objects that may be very different that what you are used to. Through the numerous labs and discussions we hope that you will see value of this perspective and the value of the practices we introduce.

Introductions

Have each person introduce themselves: name, background, programming experience, project, role.

Have each person express what they hope to achieve in Object Bootcamp.

Brainstorm Class Contract

Explain to the students that we'll need a contract for behaving in this learning environment. How can we make sure that everyone has the opportunity to get the most from their time here?

Some examples of line items on the contract are the following:

- Be punctual, from the start of the day through the breaks. We have lots of ground to cover.
- Turn mobile phones off as soon as you enter the room. Keep the distractions away!
- Wait your turn to speak, no interruptions or side conversations. Respect your peers and their interest in the learning.
- Focus on the conversation at hand. No email, internet, laptop games or fiddling during class.

(NB: [Kinesthetic learners](#) are natural fiddlers. One way of dealing with this is to have eg: stress balls which will be less distracting than other impromptu toys.)

Tell the students that we'll post the contract every day to remind ourselves of our commitments to the group.

Shape of the Class

Object Bootcamp was developed by Fred George who works for ThoughtWorks. The ideas and principle we will talk about have a long history before the bootcamp. Object thinking can generally be characterized into 2 schools: *Rational* and *Tektronics*.

- The Rational school emphasizes extensive modeling, UML and formal methods. The originators were people who wrote military software. They programmed in ADA, not a complex language. Their software that needed to be proved correct - you don't want your howitzer to stop working in the middle of a battle.
- The Tektronics school came from an oscilloscope manufacturer. The hardware they built was plug and play - different boards could be plugged in and unplugged from the oscilloscopes. They chose a new language - smalltalk - because it was a flexible and seemed to fit the need. The team was composed of Ward Cunningham, Kent Beck, Sam Adams and others you may have heard of from the agile world. With smalltalk they developed a style of moving quickly back and forth between design and coding that would form the basis of agile today. The Tektronics team ended in the mid-80s and Sam Adams went on to the KSC consulting firm. Fred George (the Twer who started this bootcamp) was introduced to the Tektronics school when he worked for IBM and owned the relationship with KSC. So this class is based on the Tektronics school of object thinking as opposed to the Rational school.

The class is taught with the Socratic method which is used in law schools. The style is all about letting you teach yourselves. We will lay out problems for you to solve and then ask you questions about your solutions to help direct you. However we won't lead you by the hand to the right answer. You will struggle at times, but struggle is good. As Fred Brooks said "Good judgement comes from experience, and experience comes from bad judgement." Over the next few days, when you find yourself struggling, remember this. Struggle is not bad, it will lead to experience and good judgement.

The class will be largely hands on programming labs, at least 3/4 of the time. If you like coding it will be a lot of fun. Our discussions will look at the details of your solutions to these programming labs. We will put people's code up on the projector and discuss it. This may be frightening at first, but we find after a couple days people start begging to have their code shown. That is how the most learning happens.

Vocabulary

Elicit vocabulary related to object oriented systems. Write them on a flip chart. Expect 30-40 words.

Ask a student to pick a word and another student to define it, and then repeat. The eventual goal (this will take multiple sessions) is a common understanding of the concepts throughout the class.

The facilitator should use this section to start assessing the class:

- What are their levels?
- What are their perspectives & prejudices?
- Who are opinion leaders? These people should be targeted because if you can convince them, you can convince the whole class.

Overview of the Class

Be cautious of giving out too much information as it could impede the Socratic Method.

The bootcamp is divided into 2 parts.

In Part 1 we will go over Object Principles and Agile Practices. This includes things like:

- Encapsulation
- XP Practices
- Object Relationships
- Delegation
- Inheritance
- Cloning
- Polymorphism
- Interfaces, Abstract Classes and Concrete Classes
- Collections
- Collaboration and Recursion
- Refactoring

In Part 2 we will go into details of Design Patterns. This includes things like:

- Strategy
- NullObject
- Observer
- Composite
- Template Method
- Visitor
- and possibly more

Object Principles

This is a quick exercise to further the facilitator's understanding of the class. Don't spend a long time defining terms or convincing the class of 70/20/10. Use the exercise to learn where the class is coming from.

Conduct a poll of the students: what are the defining principles of object oriented programming? There are 3: inheritance, encapsulation & polymorphism.

Elicit definitions from the class:

- *Inheritance*: A way to form new classes using classes that have already been defined. The former, known as derived classes, take over (or inherit) attributes and behavior of the latter, which are referred to as base classes.
- *Encapsulation*: The hiding of design decisions in a computer program that are most likely to change, thus protecting other parts of the program from change if the design decision is changed. Means of enforcing encapsulation include using the access modifier with least privilege (public, protected, private, package), avoiding accessor / mutators, and programming against interfaces instead of concrete classes.

- **Polymorphism:** The ability of objects belonging to different types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behavior.

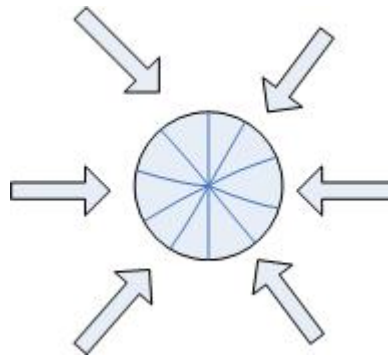
Feelings pie chart

Give everyone in the class a sheet of paper or note card. Ask them to draw a pie chart with 3 sections, one for each principle, showing how important each principle is. (Draw an example on the board). Once everyone is done, sample around the room. Tell that students that this class will come from the perspective of 70% encapsulation, 20% polymorphism & 10% inheritance. Keeping firm on encapsulation helps keep the code well organized and make it possible to refactor easily as you go. Inheritance should be used for *is a* relationship. However we'll see that often it is used to achieve code reuse. There are other ways to reuse code than inheritance.

Unable to render embedded object: File (feelings pie [chart.jpg]) not found.

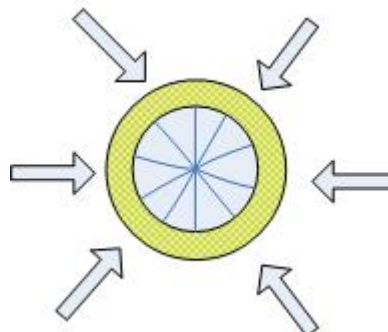
How Object Oriented Programming Evolved

The roots of Object Oriented Programming can be traced to Fred Brooks back in the 1960's. You might know him as the person who wrote the Mythical Man Month. He worked for IBM and taught at UNC. He suggested that finding the right data structure is the most important thing to do - if the data is right, the code will just about write itself. Once the data structure is correctly identified, then writing a set of behaviors around it is essentially what OOP is all about. (Draw the *data-code* picture on the right on the board). Data structures (the pie) are at the center - the most important thing. Code (arrows pointing to pie) is wrapped around the data.



As people worked with this idea, an issue arose - if you change the data structure, then the code that references that data structure is suspect. It will probably need to change as well. So the code using the data is tightly coupled particular data structures implemented.

Along came David Parnas. He was Fred Brooks's graduate student and introduced *data hiding*: the idea of hiding the data from the world by surrounding it by code. This means introducing a stable interface between the data and the code. (Draw the interface as in the *data-hiding* picture on the right). This interface hides the data from the code. The code is now dependent on the



interface, no longer directly dependent on the data. This means they can change independently, and everyone's lives became easier. This concept of *data hiding* is now referred to as *encapsulation*. Note that this history implies that objects are good for programs that change. Objects may not be the right solution if you have a program that will not change. But how often do we see that?

Job

Ok, let's get started with the first problem. Object Oriented programming starts with design. Contrary to all the rumours, agile does not throw design out with the trash. Yes, there is little upfront design as in a traditional waterfall model, but still is an important step where objects are identified, although for a very negligible time period. So let's start there. Good OO design requires conceptualizing. When you are designing your objects you should identify your classes and a *Job* for each class. A Job defines what the responsibility of a class is. Identifying it helps to clarify your thinking about the role of the class, what it might encompass and where its boundaries may lie.

Ask the students what the job of a rectangle is. Expect them to answer something like to hold the length/width or position of a rectangle. Point out that saying a class's job is the data it holds is like saying a person's job is the tools they use: stethoscope, scalpel, heart monitor. Those tools don't describe a doctor's job well. Tell the students to anthropomorphize - think of themselves as the rectangle. What do you do?

Eventually the class should arrive at a definition like *understands a 4-sided figure with sides at right angle*.

Best Practices:

- Should be crisp (not an essay)
- Should not contain the word "and"
- Should not contain the class name (no circularity)
- Classes start with *Understands...*
- Tests start with *Ensures...*

Common Mistakes:

- Too general - "understands a four sided polygon" (could be a square or parallelogram)
- List of methods or data - "understands area and perimeter" or "understands length and width"
 - What about when you want to add more methods or more data? Shouldn't have to change the job.

Lab - Rectangle

Ask the students to implement a geometric rectangle. (You may want to point out that it should be able to answer `area()` and `perimeter()`. However there is a trade-off in that you really want them to make the mistake of implementing getters and setters, which they may not do with too much direction.)

Two main lessons will come out of this lab:

1. Encapsulation (see discussion below)
2. TDD (see discussion below)

You may want to cover encapsulation first as it ties to the data hiding conversation above.

There will also be other things to look for

- Job of the class
- Names of variables, methods, parameters
- Unnecessary stuff - diagonal, etc.
- Duplication - especially in the test eg: new Rectangle(3,5) twice?
- Commented out code
- Tests

Encapsulation in Rectangle

Ask the students what public methods they defined on the Rectangle class?
Expect them to answer *getLength()* & *getWidth()*.

When this happens (or even if it doesn't if everyone has been paying attention, point out that doing so breaks encapsulation.

One way to bring this up is to say

What part of **encapsulation** didn't you understand? You gave up width and length. This is your data. Getters and setters break encapsulation. They expose the data of a class to the world. Don't do it. When outside classes want your data, say "hell no".

Another way is with the following set of questions, with sample answers in italics:

- What is the intention of make the length of width of the rectangle private? *to enable the calculation of area/perimeter*
- Looking at Rectangle's Job, where should area calculation be? *in rectangle*

In either case, this should be followed up with

If the object exposes all of its fields, what is the purpose of the object? How can you refactor the object if everyone uses its fields directly? If the user doesn't directly need a method, make it private (or internal)

When we design objects, we should think of them as having a conversation. Suppose a user of Rectangle class asks the rectangle for the length, the question that we should ask is why? What are you going to do with that? What is the real question you are trying to answer? Probably to calculate area or perimeter or to compare. In that case, that behavior has to be present in the Rectangle class itself. These are the public methods that you will define on the Rectangle class. Again we anthropomorphized. We had a conversation with a rectangle and it helped us to better understand and design the rectangle.

Review code smells related to encapsulation:

- **Law of Demeter** - 2 dots
- **Feature envy**
 - 2 classes: A & B. Instead of asking B to do things, A gets information from B and does them itself. For example: Person and TaxCalculator classes. The TaxCalculator wants to know the Person's age. It could ask the Person what his birthday is and subtract it from today's date. But that calculation belongs on Person. The Person should implement age().
- -er or -or classes like manager or configurator or calculator
 - manager classes are like real world managers - they take credit for what they dont do.
- God objects
 - classes that know everything - 10,000 lines long, 100 methods, etc.
 - we can't understand that much at once
 - the law of 7s (7 digit phone numbers, 7 line methods, 7 public methods in a class, 7 classes in a package, etc)
 - maintenance nightmare
 - Fred once worked on a system of 79 giant classes. They rewrote to 1400 classes with

distinct jobs

- a bug in the 1400 class system means something isn't doing its job right
- a bug in the 79 class system means you have to figure out who is using what data, etc.
- A lot of frameworks are dependant on get/set (eg. struts). Keep them at the edge.

We will at times violate encapsulation, but be very reluctant. If we do, it is probably related to 'programs that change' (which was the reason for objects in the first place).

TDD Conversation / Demo



As an example of the TDD cycle for the first test, consider

- [Assert.AreEqual](15, new Rectangle(3,5).Area())
- Get a compile error
- Using ReSharper to fix it (create the required class and method)
- Getting a red bar
- Fixing it by writing the simplest thing that could possibly work - hard coding a value
- Write more tests to zero in on the answer



Instructors can expect to field some questions like

- I've done the first test, what should I do now? (write the next one)
- What is the next test? (you and your pair decide that)
- How do I test the constructor (Test only the things you think will fail. Do you expect object construction to fail?)

Define the 15 minute green bar coding cycle of TDD:

1. Write a failing test
2. Write the code to make the test pass
3. Refactor

Ask the students to define refactoring: The process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

Ask the students what color the bar should be when starting each step:

1. Write a failing test - green
2. Write the code to make the test pass - red
3. Refactor - green

Ask why we refactor:

- Remove duplication
- More clearly express intent
- Make more maintainable
- Broken windows. If the students are unfamiliar with the theory of broken windows, introduce it.

Pairing Conversation







People new to pairing may think it is only for mentor/mentee pairs where the mentor can teach the mentee through pairing, but this is not the case. It is valuable for equals as well. Ask if programming is more similar to typing or solving crossword puzzles? Well programming involves a lot thinking and problem solving and a little typing so it is much more similar to solving crossword puzzles. When you have simple problems one person is very efficient at solving them, but when you have complex problems like in programming then two heads are much more powerful than one.

Pairing requires discipline. You need social skills - you can't just tell your pair they are an idiot. Pairing forces focus/attention - you can't look out window or surf the web. Pairing is intense! It is good practice to take some breaks over the day in order to catch up on email and other tasks.

Pairing styles:

- Least experienced person drives
- Ping pong: one person writes a failing test, other fixes it by writing code and writes the next test, repeat
- Ball and board: one person runs mouse the other runs the keyboard. advanced technique. forces/requires pair to be utterly in sync. very powerful

Children [Hide Children](#) | [View in hierarchy](#)

-  [Encapsulation in Rectangle](#) (Training)
-  [How Object Oriented Programming Evolved](#) (Training)
-  [Job](#) (Training)
-  [Lab - Rectangle](#) (Training)
-  [Object Principles](#) (Training)
-  [Vocabulary](#) (Training)

 0 comments |  [Add Comment](#)