From Design Model to Implementation

- Creating Language Classes
- Testing
- Refactoring
- Completing the Iteration

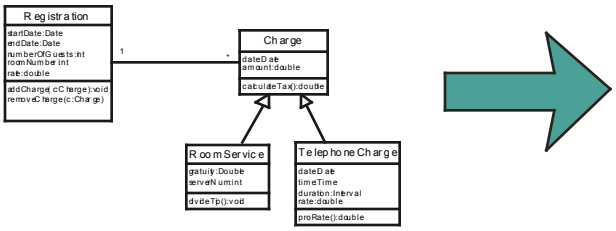
Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

From Design Model to Implementation

- Once you have a fully fleshed-out design model, your next job is to write or generate the code that implements the model

Design-Level
Class Diagram




```
public class Registration
{
   private Date startDate;
   private Date endDate;
   private int numberOfGuests;
   private int roomNumber;
   private double rate;

   private ArrayList charges = new ArrayList();

   public void addCharge (Charge c)
   {
      charges.add(c);
   }

   public void removeCharge (Charge c)
   {
      charges.remove(c);
   }
```

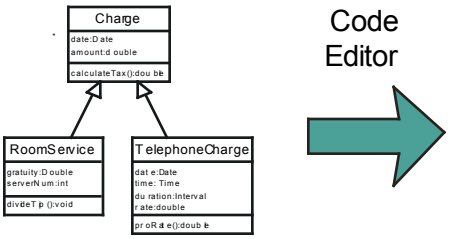
After doing all of the work to create the model, now it's time to create the actual implementation code. Since this course is about the OOA/D portion, we will not spend too much time looking at actual code.

Also note that the implementation language something you can choose, too. We will use Java as an example, but we should emphasize that any other object-oriented language would be OK, too.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Writing Language Classes Manually

- An experienced developer should be able to quickly write skeleton classes from the UML diagram
- The hard part is implementing the methods! (Interaction diagrams help)




```
public class TelephoneCharge extends Charge
{
    private Date date;
    private int durationInMilliseconds;
    private double rate;

    private static final double startRate=2.00;
    private static final double mainRate=0.10;

    private double proRate()
    {
        double proRateAmount = 0;
        if (durationInMilliseconds < 60000)
        {
            proRateAmount =
            durationInMilliseconds/60000 *
            mainRate + startRate;
        }
        else
            proRateAmount =
                durationInMilliseconds/60000 +
                startRate;

        return proRateAmount;
    }
}
```


A developer that is skilled in both their programming language and the UML can convert diagrams (e.g. class and sequence) diagrams to code that implements the model.

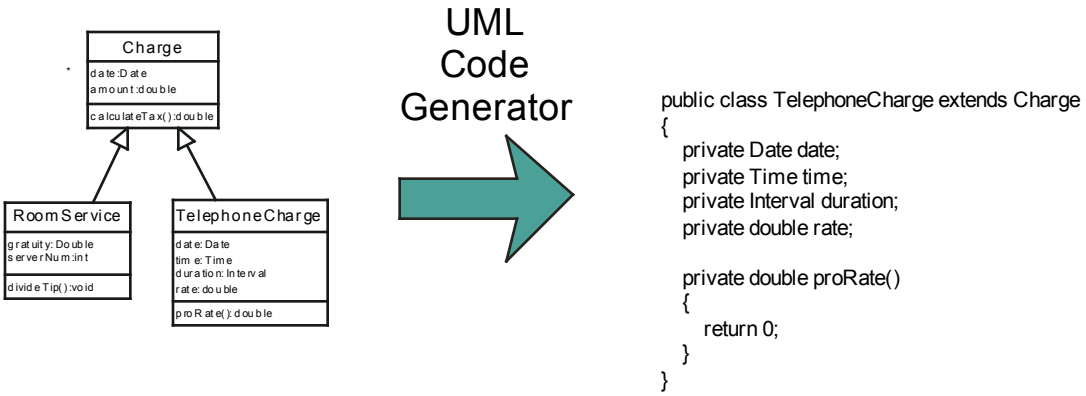
Using a modern Integrated Development Environment (IDE) can help developers perform this task efficiently.

Note however, that if the developer changes the model during implementation to get the code to actually work, the diagrams and the model may get out of sync. You will then need to decide whether it's worth the effort to continually update the model diagrams as the code is being written.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Generating Code from a UML Tool

- Most UML tools will generate the skeletons of classes for you, but developers still must implement the methods

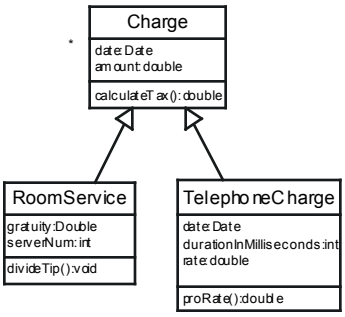


If you have a capable UML tool, such as IBM Rational or ArgoUML, developers don't need to write the skeletons of classes Instead, the tool can generate those skeletons, saving the developer a bit of "grunt" work. However, note that such tools usually cannot generate the logic of the methods.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Forward and Reverse Engineering

- Many UML tools not only generate classes (forward engineering) but also can create or update UML diagrams from existing code (reverse engineering)
- Reverse-engineering helps keep UML diagrams and the actual code synchronized



Forward/
Reverse
Engineering

```
public class TelephoneCharge extends Charge
{
   private Date date;
   private int durationInMilliseconds;
   private double rate;

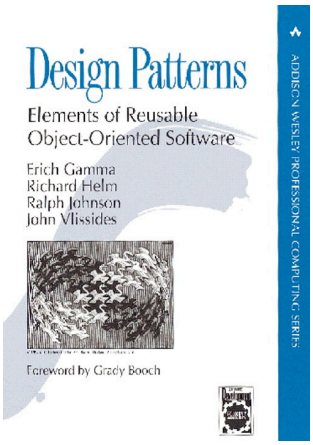
   private double proRate()
   {
     /* BEGIN CODE BLOCK */
      return 0;
     /* END CODE BLOCK */
   }
}
```


If you have a UML tool that supports reverse engineering, then it's easy to keep the code and UML diagrams synchronized. Most such tools emit comments into the code that act as markers that help the tool and show programmers where it's appropriate to edit code.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Design Patterns

- If the design model doesn't use standard design patterns, developers may need to modify the design to account for patterns
- We will cover several popular design patterns in a later chapter



Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using GRASP

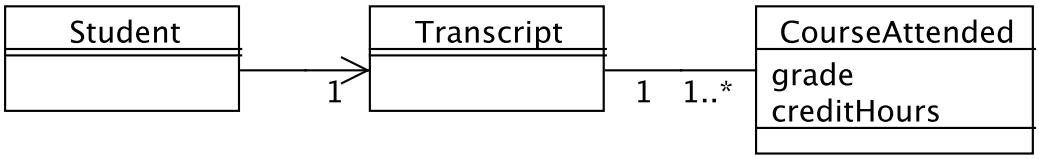
- **General Responsiblity Assignment Software Patterns** (GRASP) provides techniques to help you determine in which class to assign states and behaviors
- GRASP helps in creating maintainable design models
- GRASP includes:
    - Information Expert
    - Creator
    - Low Coupling
    - High Functional Cohesion

See "Applying UML and Patterns" by Craig Larman for more on GRASP.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

GRASP: Information Expert

- **Information Expert** advises placing behavior in the class that has the state that the behavior requires



- In which class would you put a **calcCumulativeGPA()** behavior?

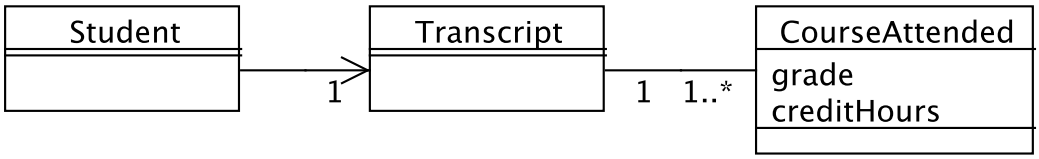
Information Expert minimizes coupling by eliminating unnecessary method calls.

In this case, since Transcript has the information to calculate an average (i.e., the list of CourseAttended), it makes sense to code that behavior in Transcript.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

GRASP: Creator

- **Creator** advises placing the behavior to instantiate objects in the class that aggregates the objects



- In which class would you put a **createCourseAttended()** behavior?

Creator assigns responsibility based on which class keeps track of the objects to be created.

In this case, since Transcript keeps the list of CourseAttended, it makes sense to assign the responsibility of creating CourseAttended to the Transcript class.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

GRASP: High Functional Cohesion

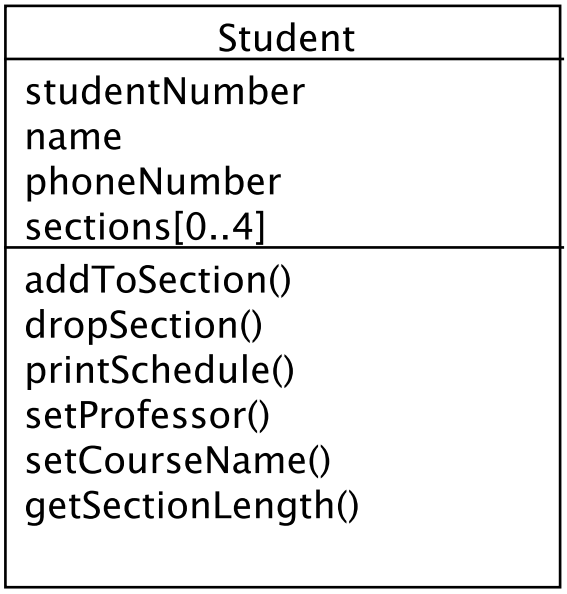
- "A class should have **all** of the methods it needs, but no more"
- High cohesion often reduces coupling (another GRASP) since methods don't need to refer to methods in other objects
- But beware of **king classes** that do too much resulting in poor re-use and improper *separation of concerns*

A highly cohesive class does only one thing but does it well.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

GRASP Exercise

- Given this model, determine which GRASP techniques it misuses and then redesign with proper GRASP



Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Testing Fundamentals

- Testing is a fundamental part of any software development process, but is especially important for an iterative process such as the Unified Process
- Types of tests:
    - Unit Tests
    - Integration Tests
    - Acceptance Tests

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Unit Testing

- A unit test is a self-contained test within a class that verifies the class's methods' basic function
- Unit tests are a type of regression test and are vital in an iterative process to ensure that code added in subsequent iterations doesn't break previously written code
- There are several frameworks available for unit testing, for example JUnit




```
import junit.framework.TestCase;

public class RegistrationBeanTest extends TestCase
{
  public void testCalculateAmountDue()
  {
    RegistrationBean rb = new RegistrationBean();
    rb.setCds(0);
    rb.setAttendees(1);
    rb.setGuildmember(false);
    assertTrue ( "Amount due should be 1000",
      rb.calculateAmountDue() == 1000 );
  }
}
```

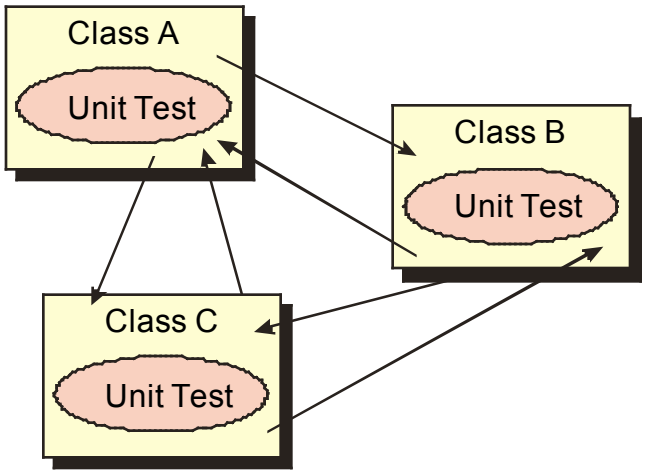

To unit test a class, you write a companion class that contains methods that exercise the methods of the class being tested. Each of the testing methods should essentially deliver a pass/fail indication. For a complex class, there might be dozens of testing methods -- the idea is that all of them must pass for the entire test to pass.

Using a framework like JUnit makes it fairly painless for developers to write and run the unit tests. That's important, since developers generally don't like to spend a lot of time writing test cases.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Integration Testing

- While unit tests verify the standalone function of classes, integration tests verify that the classes work together



Integration testing checks the interactions between the classes in a software system, i.e. that higher-level procedures (e.g. use-cases) execute as desired.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Acceptance Testing

- To ensure that the stakeholders (e.g. end-user) is satisfied, you must do an acceptance test at the end of each iteration
- Remember that you also must make a continue/stop decision at this point!

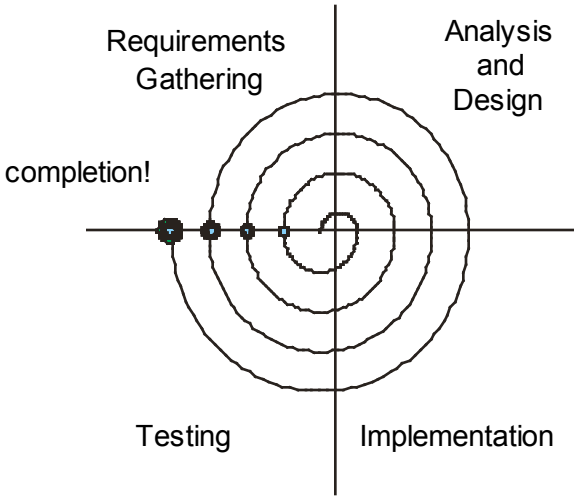


An important part of an iterative process is feedback from the stakeholders at the end of each iteration. In fact, you shouldn't consider the iteration complete until the stakeholders have signed off on the iteration's functionality.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Planning the Next Iteration

- After successfully completing an iteration, your next job is to determine what to accomplish in the next iteration
- Remember that you should do the riskiest parts first



Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Refactoring

- The end of an iteration is also a good time to examine existing working code to determine if it can be improved (e.g. simplified or made faster)
- The act of modifying working code is referred to as refactoring -- the standard book on the topic is "Refactoring: Improving the Design of Existing Code" by Martin Fowler
- Note that unit tests help ensure that refactoring doesn't break existing code

Some Integrated Development Environments (IDE) such as Eclipse support refactoring to make it easier for developers to perform common refactoring, such as extracting duplicate code into a new method, moving methods up to a superclass and so forth.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Review Questions

- Why is unit testing critical in an iterative process?
- What is the difference between forward and reverse engineering?

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- The basics of converting a design model into implementation code
- Testing fundamentals and its importance in an iterative process
- Refactoring fundamentals

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum