Testing in Rails

All Rails projects start with a `test` directory. It contains subdirectories for various kinds of tests.

- *Unit tests* are used to test a particular class. They call methods of the class and check whether the expected response is received.
- *Functional tests* are used to test individual requests made over the web. They test for conditions such as …
    - was the web request successful?
    - was the user redirected to the right page?
    - was the user successfully authenticated?
    - was the correct object stored in the response template?
    - was the appropriate message displayed to the user in the view?
- *Integration tests* test how different parts of the application interact. They can be used to test use cases.
- *Performance tests* are designed for benchmarking and profiling the code. Like functional tests, they can test individual requests. Like integration tests, they can test multiple parts of the application.

What kinds of tests are these?

Philosophy: Put as much as possible in the model. This avoids dependencies between business logic and presentation.

The view can be as complicated as you want, as long as the logic is only to display information to the user.

Fixtures

In order to run tests, you need data to test with. You could execute code to set up the objects used in your tests at the beginning of each test. And this is what factories do. But if objects remain the same between tests, it isn't necessary.

Just like we can use .erb files to specify how a view looks, we can use *.yml files* to specify objects that exist when a test starts.

.yml is the extension for YAML files (YAML stands for "YAML ain't markup language").

In a Rails project, the fixtures are stored in the test/fixtures directory. Fixtures can refer to each other. Which lines in categories.yml and recipes.yml refer to other fixtures?

Some fixtures are generated automatically.

In this directory, which of the lines in categories.yml and recipes.yml do you think were autogenerated, and which were inserted manually? Why?

The idea is that you can autogenerate a few fixtures, which are instances of objects of the class, and then write ERB code to generate a lot of others that have the same basic format.

When are the autogenerated lines actually generated?

Rails loads fixtures automatically when tests are run.

- It removes any existing objects from the database table that the fixture is an instance of.
- It loads the fixture data into the database table.
- It allows the program to refer to the fixture by name.

What concept that we introduced last week are fixtures an instance of? ActiveRecord

How long do we want the data to last when we load it into a testing database? As long as the tests run.

Running tests

We need to set up a test environment explicitly. Let's look at config/database.yml.

How many databases does it reference?

It's important to have a separate test db for a reason we mentioned above. What reason is that?

Unit tests

Unit tests are typically used to test models. Why are they suitable for models?

Let's look at `recipe_test`, which has two tests to determine whether it is possible to create a recipe with all fields blank.

It's good practice to have a unit test for each method in a model.

Each test must include at least one assertion. The assertions should test everything that is likely to break.

Which of the following scenarios should be tested by unit tests?

What else could you test about recipes?

Many kinds of assertions are available.[1]

Assertion	Purpose
`assert( test, [msg] )`	Ensures that `test` is true.
`assert_not( test, [msg] )`	Ensures that `test` is false.
`assert_equal( expected, actual, [msg] )`	Ensures that `expected == actual` is true.
`assert_same( expected, actual, [msg] )`	Ensures that `expected.equal?(actual)` is true.
`assert_nil( obj, [msg] )`	Ensures that `obj.nil?` is true.
`assert_match( regexp, string, [msg] )`	Ensures that a string matches the regular expression.
`assert_raises( exception1, exception2, ... ) { block }`	Ensures that the given block raises one of the given exceptions.
`assert_instance_of( class, obj, [bmsg] )`	Ensures that `obj` is an instance of `class`.
`flunk( [msg] )`	Ensures failure. This is useful to explicitly mark a test that isn't finished yet.

[1] From http://guides.rubyonrails.org/testing.html

Almost all of these also have negative versions, e.g., **`assert_not_equal(…)`**.

Functional tests

Let's run a functional test, e.g., `category_creation_flow_test.rb`.

- One visits the new category page, fills in the form and checks to see if the new category can be displayed
- One visits the categories page and checks whether the application is displaying the correct title.
- One visits the new recipe page, fills in the form and checks to see if the new recipe can be displayed

Let's run a functional test, e.g., `category_creation_flow_test.rb`.

```
class RecipesControllerTest < ActionController::TestCase
  setup do
    @recipe = recipes(:one)
  end
  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:recipes)  Make sure something is assigned
  end

  test "should get new" do
    get :new
    assert_response :success
  end

  test "should create recipe" do
    assert_difference('Recipe.count') do  Make sure count changes
      post :create, recipe: { description: @recipe.description,
instructions: @recipe.instructions, title: @recipe.title }
    end

    assert_redirected_to recipe_path(assigns(:recipe))
  end

  test "should show recipe" do
    get :show, id: @recipe
    assert_response :success
  end

  test "should get edit" do
    get :edit, id: @recipe
    assert_response :success
  end

  test "should update recipe" do
    put :update, id: @recipe, recipe: { description:
@recipe.description, instructions: @recipe.instructions, title:
@recipe.title }
    assert_redirected_to recipe_path(assigns(:recipe))
  end

  test "should destroy recipe" do
    assert_difference('Recipe.count', -1) do
      delete :destroy, id: @recipe
```

```
    end

    assert_redirected_to recipes_path
  end
end
```

Let's look at a functional test, e.g., `recipe_creation_flow_test.rb`. This test uses the following commands provided by the Capybara gem [SaaS §7.5]:

- **`visit`**: navigate the Capybara driver to that particular page. Note that now the application is being tested from a browser perspective
- **`fill_in`**: fill in the particular form field
- **`select`**: for the recipe select a particular category from the category dropdown
- **`click_button`**: actuate a button

What kinds of functional tests would be good to have?

Integration tests

Integration tests are used to test interactions among controllers.

No integration tests are auto-generated. But, Rails provides a generator to get you started:

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  # test "the truth" do
  #   assert true
  # end
end
```

Look at the "**`should use layout`**" test in `categories_controller_integration_test.rb`. This makes sure you can get a page without a 404 error, etc. We can change the title of the

page in `application.html.erb` to be something other than "Cookbook", and see that the test fails.

(When this test runs, it fails because of a wrong title on the cookbook.)

Notice that almost every test requires `test/test_helper.rb`. This is included as a mixin, so that the functionality is available to every test.

A good description of almost everything we have covered today is in "A Guide to Testing Rails Applications." Mocks and stubs are an alternate way of setting up tests, which allows things to be tested that do not yet exist, or are too expensive or destructive to access.

- Stubs are objects where, if someone calls this method on you, this is what you're going to respond with — and that's it. It's a stand-in for some other object. Just returns canned data.
- Mocks are things like supporting services: if you are testing an emailer, it would normally send email, but in this case it doesn't, and acts like it did. "Let's not, & say we did." They verify that a particular method was called.

A readable description of mocks and stubs can be found at Code with Jason; Jesus Castello has a good video description.

The SaaS text covers them in Sections 8.3 and 8.4. It also covers fixtures in Section 8.6. Fixtures can be brittle; if you change the schema of a table, you have to change the associated fixtures—and possibly other fixtures that reference the fixtures you changed.

Now, test your knowledge of the different kinds of tests by filling out this form

Design

What do we mean by program *design*?
Deciding on the relationship between major entities in the program.

Why do we worry about design when writing a program? Why isn't it enough that the program works?

Outline for Week 7

Suppose the code is never intended to be read by anyone else, or used again?

O-o design: The CRC-card method

In writing object-oriented software, it is very important to get the design right.

If the design is wrong,

- Objects of one class may need to make extensive use of features of another class ("high coupling").

  It's OK if objects of one class merely *use* public features of another class. But if you find your code depending on the implementation of the other class, your code becomes unmaintainable.

- Methods and instance variables grouped in one class have little relationship with each other (low cohesion).

To get the design right, we should be careful to choose our classes.

The goals of this process are to—

- Discover classes.
- Determine the responsibilities of each class.
- Describe the relationships among the classes.

To discover the classes, we can look for the *nouns* in the task description (sometimes called the "requirements document").

For example, if I say,

> The function of the system is to allow bus riders to plan a route from origin to destination,

what might be the classes? Route, Location

When choosing classes, make sure that what you identify …

- is a singular noun,
- does not really have the same functionality as some other class,
- is not simply a primitive type or a library object,

Now let's consider a sample system.

Example 1. Flight reservation

Requirements for the Flight Reservation System

- The mission is to allow round-trip airline *tickets* to be bought over the Web.
- Each *customer* specifies an origination *airport*, a destination airport, and dates for outbound and return *flights*.
- The customer reserves one *outbound flight* and one *return flight* from a menu presented by the system.
- Each choice that the system presents consists of one or more flight *segments* (there may be a stop or a change of planes).
- The customer may buy tickets for one or more *passengers*.
- No more tickets can be sold for a flight than there are *seats* on the *plane*.
- Each passenger is assigned to a specific seat.
- The system calculates the total *cost* of the tickets by adding the cost of the individual segments.
- If dissatisfied with the cost, the customer may select *alternate flights*.

- After a customer has bought a ticket, (s)he will be e-mailed a *confirmation*

Take a couple of minutes working with your group to identify the classes. Then enter your class names here.

Also name some nouns that are not classes.

(*Note:* Be sure to avoid this common misconception: Something that is an *attribute* of another class may be a class itself!)

Example 2. Address book

Here is a very complete example of an address book.

We will work our way from the requirements statement, through use cases to CRC cards.

Responsibilities and collaborators

Finding the classes is only the first step in the design process.

Next, we need to look for *responsibilities*, which are usually *verbs* in the task description.

For each responsibility, there may be one or more *collaborators*—classes that need to be called to help fulfill the responsibility.

In summary, we have—

- *Classes:* To find the objects, look for the nouns.
- *Responsibilities:* Things a class knows or can do.
- *Collaborators:* Other classes that are *directly* involved in fulfilling these responsibilities.

Now let's consider some responsibilities of the Customer class in the Flight Reservation system. Which collaborator(s) does each one have? Enter responsibilities and collaborators here.

CRC cards

A common design practice is to write information for each class on a separate card. A card has the form …

Class Name	
Responsibility 1	Collaborator(s) 1
Responsibility 2	Collaborator(s) 2
…	…
Responsibility *n*	Collaborator(s) *n*

We don't have a good way for you to share entire CRC cards with the rest of the class, but you can simulate a CRC card by filling out this class/responsibility/collaborator form repeatedly.

Common errors in CRC-card design

In designs created by students, certain errors keep coming up over and over. Here are some examples.

1. *Using a class name that is not a singular noun.*

   "Customers", "Segments", "Buy"

2. *Naming a system class as a key abstraction of the program.*

   "String", "Date"

3. *Defining a new class where an existing (usually primitive) object would suffice.*

   "Cost", "Time"

4. *Thinking that something can't be a key abstraction because it is part of a larger abstraction.*

   "Seat" can't be a key abstraction, because it's part of the plane.

   "Wheel" can't be a key abstraction, because it's an attribute of the plane.

5. *Confusing inheritance with composition.*

   "Seat" inherits from "Plane"

6. *Confusing an object with an aggregation of such objects.*

   Responsibilities of Seat include knowing the available number of window, aisle, and exit-row seats

7. *Confusing ambiguity with synonyms.*

   Ambiguous: 1 term, 2 meanings

   Synonyms: 2 terms, 1 meaning

   "Segment" and "leg" are synonyms with regard to flights, because they mean the same thing.

8. *Treating collaboration as a transitive relationship.*

   Class: Customer
   Responsibility: Buy ticket
   Collaborators: Passenger, Flight, Segment, Airport

Example 3. Invitations

Consider inviting someone to join your team in Expertiza. What should be the responsibilities of an Invitation class?

Send an invitation (be sent)
Accept an invitation (be accepted)
Decline an invitation (be declined)
Retract an invitation (be retracted)

What does the code need to do when an invitation is sent?

- Verify that there is room on your team.

(Should you be allowed to issue more invitations than you have places on your team? Probably not.)

- Verify that the invitee is a participant in the assignment.
- Set the reply status to W.

What does the code need to do when someone accepts an invitation?

  - Verify that there is room on the team.
  - Add the invitee to the team.
  - Set the reply status to A.
  - [Email the inviter]
- What do you need to do when you *decline* an invitation?
  - Set the reply status to D.
  - [Notify the sender.]

Now let's see how the current Invitation class is implemented in Expertiza.

Does it give us any hints on what else needs to be done when an invitation is accepted?

What else does the code need to do when someone accepts an invitation?

Now let's design the `ProjectTopic` class. What are the responsibilities of `ProjectTopic`?

- Create a topic.
- Select a topic.
- Acquire a topic (if you were waiting for one).
- Relinquish a topic.

Now let's look at the current SignupTopic class.

Do we need an `invitations_controller`?

Design Smells

"Uncle Bob" Martin, the architect of the SOLID principles, identifies several "design smells" that are symptomatic of "rotting software."

Outline for Week 8

Rigidity

The system is hard to change because every change forces changes to other parts of the system.

You start to make what seems to be a simple change, but as you get into it, you find that it impacts more code than you expected. And when you fix the code in the other places, it affects still more modules. What principle or guideline from past weeks does this violate?

Fragility

A single change tends to "break" the program in many places.

Often those places have little apparent relation to the place where the code is changed. Patching those modules may make the problem worse later on.

Immobility

Parts of the code could be useful in other systems, but it is easier to rewrite them than to extricate them and reuse them.

Viscosity

When changes need to be made, the design is hard to preserve.

It is easier to hack a change into the code than to make it in a way that follows the principles of the design. *Example:* Instead of subclassing a base class again, use case statements to add new functionality.

Needless complexity

The design includes elements that aren't currently useful. Maybe the designer *expects* them to be useful later on …

Needless repetition

What's another name for this problem?

Opacity

A module is difficult to understand, not written in a clear and direct manner.

Code tends to become more opaque as it ages, because no one is intimately familiar with it any longer.

SOLID Principles

The SOLID principles are an acronym for five design principles that are not patterns, but just rules to be followed when designing programs.

The Single-Responsibility Principle

[SaaS §11.3] The Single-Responsibility Principle is

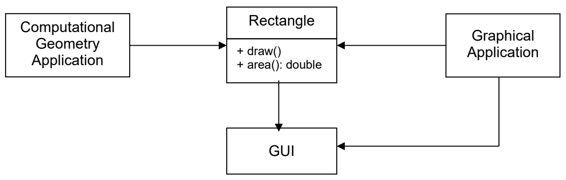
A class should have only one reason to change.

We have already seen this principle. Good cohesion dictates that, "Every class should be responsible for doing one thing only and doing it well."

But what does "doing one thing only" mean? Martin says it means that a class should have only one reason to change.

One bad example, that is common in student code, is a controller class that does calculations related to the application's business logic.

Another of Martin's examples is a Rectangle class that has two responsibilities: calculate area and draw itself.



Two applications use Rectangles. Only one needs to draw the rectangle.

In a static language, the GUI class would have to be included in both applications. Changes to the `draw()` method would force the computational-geometry application to be recompiled, even though it doesn't use the method.

A dynamic language doesn't have these problems, but still in order to change the view, the model would have to change.

Now, it is possible to go overboard with this principle. Too many classes are bad, too. The ideal number of methods for a class ≠ 1.

Martin clarifies what he means by a single reason for change: "*Gather together the things that change for the same reasons. Separate those things that change for different reasons.*"

Here's an example of a `CityMap` class.

"Bad" Example

Main class: `CityMap`

In this example, the `CityMap` class represents a map consisting of a list of cities with various attributes. Although this represents a single logical object, the `CityMap` class takes on several very separate pieces of functionality which should, according to this principle, be divided into several classes. Those functionalities include managing the list of cities (add and remove), drawing the map on the screen, and calculating the total population.

"Good" Example

The good example simply splits the `CityMap` class into two classes, `Map` and `CityList`. `CityList` maintains the `ArrayList` of cities and also allows calculating the total population. The Map class focuses solely on drawing the map on a screen. This fixes the issues with the "bad" example, as each class now focuses solely on operations related to one set of data.

First, say which components of the "bad" example should go into each class in the "good" example.

Then, fill in the blanks in the Ruby or Java "good" example.

The Open-Closed Principle

The open-closed principle can be expressed as follows:

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Why is it harder to add new functionality to a program when an existing class needs to change? Unfortunately, this approach may require other classes that depend on the changed classes to change, which in turn may require still other changes. These changes may introduce new errors into the code. Better to minimize the amount of change to working code and instead to extend that code by adding new classes that incorporate the changes.

Keeping the data of a class private helps assure that the class is closed for modification.

Here is a bad and a good example of the Open-Closed principle.

Bad Example

Main class - **ProgramRunner**

In this example, ProgramRunner.java is responsible for running programs from several programming languages. Two classes (`PythonProgram` and `RubyProgram`) implement the `Program` interface, which has a `getCode()` and a `getType()` method.

`ProgramRunner` has to figure out which type of program it has been given in order to run it, and therefore has an ugly if statement followed by a separate method to run every type of program.

In production, such a system would quickly grow unwieldy, as adding any type of program requires adding to the if statement and adding new methods to the `ProgramRunner`, breaking the Open/Closed Principle.

Good Example

The Good example corrects the above issues by simply adding a `runProgram`" method to the Program interface.

This renders the entire `ProgramRunner` class obsolete, and allows each class to handle its own execution, rather than being tightly coupled to a `ProgramRunner` class.

As a result of this change, new Program types can be added without needing to also update the `ProgramRunner` class.

Fill in the rest of the code for the good example in Java (or in Ruby).

If classes are not to change, then you need to be careful to design them so they don't need to. This suggests …

The Liskov Substitution Principle

[SaaS §11.5] A short statement of the Liskov Substitution Principle is,

> Subtypes must be substitutable for their base types.

If they are not, you have to be careful in writing code that uses the base type. One example from StackExchange:

Suppose you have a class `Task` and a subclass `ProjectTask`. `Task` has a `close()` method that doesn't work for `ProjectTask`.

Here is some code that uses `close()`.

```
public void processTaskAndClose(Task taskToProcess)
{
    taskToProcess.execute();
    taskToProcess.dateProcessed = DateTime.now;
```

```
    taskToProcess.close();
}
```

You can't be sure this code will work if a `ProjectTask` is passed to `processTaskAndClose`. So you need to put some kind of **if** statement or **case** statement around the call to `close()`.

Here's an exercise involving the LSP.

"Bad" Example

In this example, a `Computer` object keeps track of its amount of RAM and its OS version. It also has methods for upgrading the RAM and updating the OS. Two classes, a `DesktopComputer` and a `Phone`, extend this class and implement its methods.

A `ComputerUpgrader` object claims to be able to upgrade any `Computer` (that is, add more RAM and update the OS), but it really can't add more RAM to a phone, so it must check to make sure the `Computer` object it has been given isn't a `Phone`.

This violates the LSP, as a `Phone` cannot fully be substituted for a `Computer`.

"Good" Example

The most straightforward method of solving the above problems is to add a new interface `HardwareUpgradable`, which is only implemented by `Computer`s which can have their hardware upgraded (`DesktopComputer` can, Phone cannot).

Next, by splitting the upgrade method in `ComputerUpgrader` into `upgradeRAM` (which accepts `HardwareUpgradable`) and `upgradeOS` (which accepts any computer), the issue can be resolved. No type-checking is necessary.

Fill in the blanks to finish this example in Java (or in Ruby).

The Interface-Segregation Principle

The Single-Responsibility Principle tells us that classes that are too big are no good. The Interface-Segregation Principle says the same thing about interfaces. It is,

> Clients should not be forced to depend on methods that they do not use.

If you know Java, you are probably familiar with the `MouseListener` and `MouseMotionListener` interfaces. Both of them handle `MouseEvents`. Why are two listeners needed for `MouseEvents`, when all other kinds of events have only one listener interface?

Because many programs don't track movement of the mouse, and thus, they can get by with many fewer events.

This video describes the issue of read streams vs. read-and-write streams.

OK, you might say, this makes sense for Java, but how about Ruby? Ruby doesn't even have interfaces!

Indeed, dynamically typed languages don't need interfaces. Why?

They have duck typing!

The issue, then, is how to give a Ruby class access to the methods it needs from another class, rather than giving it access to *all* the methods, which it would get if it inherited from the class.

The **forwardable** mixin has a **def_delegator** method that allows one Ruby class to use some, but not all, of the methods of the class it delegates to. This video shows how **forwardable** can be used to create a **Moderator** class that can edit posts, but not create or delete them.

Here is an exercise involving the ISP.

"Bad" Example

In this example, a single interface, `Game` is created, for two classes, `SingleplayerGame` and `MultiplayerGame`.

This is on the surface a logical structure. However, in this case, the methods `getServerList` and `pauseGame`, published in the `Game` interface, are not used by both clients (as a `MultiplayerGame`

cannot be paused, and a `SingleplayerGame` does not have servers).

Because of this mismatch, the `SingleplayerGame` is forced to throw an `UnsupportedOperationException` when `getServerList` is called on it, and `MultiplayerGame` is forced to throw an `UnsupportedOperationException` when `pauseGame` is called on it.

This demonstrates a violation of the Interface Segregation principle, as a single, logical but ill-fitting interface is used by several clients, despite clear incompatibilities.

"Good" Example

This example is derived from the "bad" example. In this case, the single interface `Game` was split into three interfaces with a single method each: `BasicGame`, `OnlineGame`, and `PausableGame`.

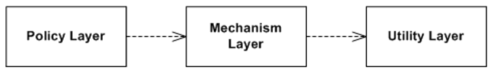
With this split, `MultiplayerGame` can implement `BasicGame` and `OnlineGame`, but it does not need to implement `PausableGame` (as it is not pausable), and `SingleplayerGame` can implement `BasicGame` and `PausableGame` (as it can be paused but is not online). This corrects the need to throw `UnsupportedOperationExceptions`, and follows the ISP by dividing one general purpose interface into several smaller interfaces.

Fill in the blanks in the "good" Java code (or Ruby code).

The Dependency-Inversion principle

[SaaS §11.6] We have just seen an instance where code depends on abstractions to decide what kind of object is to be created. Now here is another situation where code is clearer if it depends on abstractions.

Any object that uses another object to carry out its work is said to *depend* on the other object. A very common layered architecture has higher-level modules depending on lower-level modules, like this.



However, the Dependency-Inversion Principle says this is not good. It says,

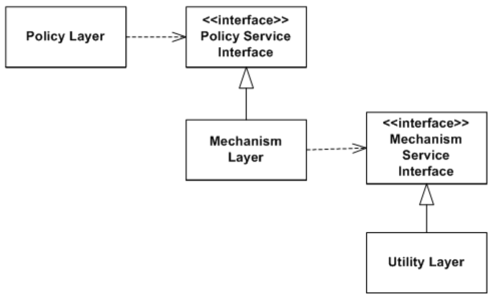
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

The reason that it's bad for high-level modules to depend on low-level modules is that a change to a low-level module can require a change to a high-level module.

This makes it hard to contain the damage when editing low-level modules.

The reason that this is called an "inversion" principle is that it goes against the advice of other software-development methodologies, such as Structured Analysis and Design.

Interposing interfaces between the various levels allows either level to change without affecting the other, assuming that the same interface is still implemented.


Policy Layer
<<interface>> Policy Service Interface
Mechanism Layer
<<interface>> Mechanism Service Interface
Utility Layer

The second part of the principle says, essentially, that high-level modules should not get involved in performing low-level functions.

This v ideo explains that

- the Coca Cola CEO should not deliver products to 7/11, and
- the Ruby ActiveRecord class should not say how an application's **users** table is to be structured.

Just as with the Single-Responsibility and Interface-Segregation Principles, it is possible to go overboard with Dependency Inversion.

The system should not be filled with single-method classes and interfaces, nor should there be an interface between every two classes.

Interfaces should be reserved for the boundaries between layers, or collections of classes that are otherwise cohesive.

Here is an exercise involving the Dependency-Inversion Principle.

"Bad" Example

Main Class: **Bank**

In this example, a **Bank** class is a high-level class with complex functionality (redacted for this example). One piece of that functionality is handling transactions between various accounts (simple, low-level classes).

The **Bank**, however, is very tightly coupled with the **CheckingAccount** and **SavingsAccount** classes. Adding additional account types (such as a **MoneyMarketAccount**, **InvestmentAccount**, or **RetirementAccount**) would exponentially increase the complexity of Bank.

"Good" Example

Main Class: **Bank** (high level) / **Accounts** (low level)

To correct the above issues, a layer of abstraction between the various types of accounts and the high-level **Bank** class is added. In this case, a simple **BasicAccount** interface is added between the layers.

Although each account may process transactions in different ways (for example, Federal Reserve Regulation D requires that savings accounts limit the number of transactions per month, but this does not apply to Checking accounts), a simple, uniform interface can be provided. This dramatically simplifies the **Bank** class, and will allow for new types of accounts to be added easily.

An argument could be made to make **BasicAccount** an abstract class. In this limited example, this would actually simplify the codebase, by allowing the repeated code found in various versions of **deposit()** and **withdraw()** to be moved to a single location. In that setup, classes which require checks/validation before accepting a deposit or withdraw could make those and then delegate to the abstract class.

However, in a more complete system, there may very well be scenarios where this structure would not work (for example, HSA/IRA/Business/Credit accounts may have very different deposit or withdrawal structures). Therefore, the class is left as an interface, as that structure makes the example clearer (it is essential to clarify

the interface as existing primarily as a layer between **Bank** and the **Account** classes).

Fill in the blanks to complete the code.

Now, to test your knowledge of the SOLID Principles, take this quiz.

Readability

“Programs must be written for people to read, and only incidentally for machines to execute.”—Abelson & Sussman

Outline for Week 9

Guideline: Give a variable the narrowest scope that you can.

Give an example of this principle. Index variables in loops.

Why is this a good principle?

Guideline: Using standard idioms, make code as concise as possible.

Example: In the following statement, **b** is a boolean variable:

```
if (b == true)
  return true;
else
  return false;
```

This statement is far too verbose. An equivalent and much more readable statement is—

```
return b;
```

In most cases, this can be made even more readable. How? Use a better name for the variable.

Guideline: Variable names should be neither too short nor too long.

Consider a variable that controls whether a **while**-loop is exited.

```
while (variable) {
…
}
```

What is a good name for this variable? notFound

Which should be shorter, in general? Variable names or names of constants?

In general, names that are used less frequently can be longer.

Guideline: Names should be descriptive of the entity they apply to. They should not be vague or overly general.

Give an example of a bad (variable, method, etc.) name you have encountered in code that you were refactoring or interfacing to.

Here are some examples from Expertiza.

Guideline: Names should not be redundant.

Suppose `course_controller.rb` contains a method called **`create_course`**. What should it be? **`create`**

Suppose it contains a method called **`create_section`**. What should it be? **`Section.new`** (or **`Section.create`**, if it is to be immediately saved to the db).

An excellent discussion of variable naming can be found in *Code Complete*, by Steve McConnell, on electronic reserve for this course.

Guideline: Factor out duplicated code.

If a program has two places where the same sequence of instructions is being executed, it is almost always beneficial to move the duplicated code into a separate procedure.

Example: Suppose you are developing a class of objects one of whose responsibilities is to parse an input string, such as a complicated mathematical expression.

Part of the process of parsing involves checking that the input is valid. So the class might have a method like this:

```
public void parse(String expression)
{
  ...do some parsing...
  if( ! nextToken.equals("+") )  {
    //error
    System.out.println
      ("Expected +, but found " + nextToken);
    System.exit(1);
  }
  ...do some more parsing...
  if( ! nextToken.equals("*") )  {
    //error
    System.out.println
      ("Expected *, but found " + nextToken);
    System.exit(1);
  }
  ...
}
```

How can we clean this code up?

```
private void handleError(String message) {
  System.out.println(message);
  System.exit(1);
}

public void parse(String expression)
{
  ...do some parsing...
  if( ! nextToken.equals("+") )
   unexpectedToken
     ("Expected '+', but found " + nextToken);
  ...do some more parsing...
  if( ! nextToken.equals("*") )
   unexpectedToken
     ("Expected '*', but found " + nextToken);
  ...
}
```

Besides being more readable, this code has another advantage. What? It is also much easier to change the error handling. For

example, if you decide later that your parser needs to throw an exception instead of printing a message and quitting, you only need to change the code in one place, namely inside the body of the **`unexpectedToken`** method.

Guideline: A method should do only one thing and do it well.

Here's an example of a method to avoid:

```
void doThisOrThat(boolean flag) {
  if( flag ) {
    ...twenty lines of code to do this...
  }
  else {
    ...twenty lines of code to do that...
  }
}
```

How should we change it?

```
void doThisOrThat(boolean flag) {
  if( flag )
    doThis();
  else
    doThat();
}
```

Inheritance vs. delegation

Delegation—where one object passes a message on to another object—can often achieve the same effect as inheritance. Let’s look at an example.

Consider the **`java.util.Stack`** class. How many operations does it have? 50 or more

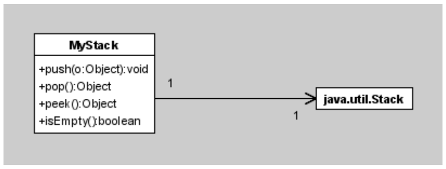
Suppose in a program you want a “pure” stack class—one that can only be manipulated via **`push(…)`** and **`pop()`**.

Why would you want such a class, when Java already gives you that and more?

What is the "simplest" way to get a pure **Stack** class? Override the methods that you don't want will null implementations. But this violates the LSP.

Or you could create **Stack** class "from scratch." What's wrong with doing this? Violates DRY

Another option is to create your own **Stack** class, but have it *include* a **java.util.Stack**.



What is the name for the approach are we using here? Delegation

Here's what this class might look like.

```
public class MyStack
{
   private java.util.Stack stack;
   public MyStack(){stack = new java.util.Stack();}
   public void push(Object o) { stack.push(o); }
   public Object pop() { return stack.pop(); }
   public object peek() { return stack.peek(); }
   public boolean isEmpty(){return stack.empty();}
}
```

Delegation is particularly useful where objects might need to "change state"—think of a student becoming an employee. Both Student and Employee can delegate to Person.

Exercise: Delegation in a sorted list

This exercise is an example of creating a sorted **ArrayList** of **String**s by delegating to Java's **ArrayList** class. Every time an element is added to the list, the **sort** method of **Collections** is called.

This exercise asks you to fill in the blanks so that the list stays sorted.

Polymorphism

Unbounded vs. subtype polymorphism

In a statically typed o-o language like Java or C++, you can declare a variable in a superclass, then assign a subclass object to that type:

```
public class Bicycle {
  protected int gear;
  public void setGear(int nextGear) {
    gear = nextGear;
  }
}
public class MountainBike extends Bicycle {
  protected int seatHeight;
  public void setHeight(int newSeatHeight) {
    seatHeight = newSeatHeight;
  }
}
public class BikeSim {
  public static void main() {
    ...
    Bicycle myBike = new MountainBike();
    ...
    myBike.setGear(3);
    myBike.setHeight(5);
    }
}
```

Which statement is illegal in the code above? Why?

In most dynamically typed o-o languages, including Ruby, that statement would be legal. In Ruby, if a method is defined on an object, the object can respond to the message.

It doesn't matter what class the object is declared as … in fact, the object isn't declared!

This is called *unbounded polymorphism*—the polymorphism is not limited by the declared class of the object.

In contrast, statically typed o-o languages usually have *subtype polymorphism*—the compiler checks that the invoked method is defined in the type that the object is declared as.

Unbounded polymorphism is related to *duck typing*, which was discussed in the Week 3 online lectures [§2.4 of the textbook].

Dynamic method invocation

A call to an inherited method works just as if the inherited method had been defined in the caller's class.

But suppose the subclass (e.g., **MySpiffyLabel**) overrides a method of the superclass (e.g., **JLabel**).

```
JLabel label = new MySpiffyLabel("A label");
label.paint(g);  //for some Graphics object g
```

MySpiffyLabel

- inherits a **paint** method from **JLabel**, and
- implements its own version of **paint**.

Which of those two implementations of **paint** will be executed in the second line of above example?

- The paint defined in **JLabel**?
- The paint defined in **MySpiffyLabel**?

Dynamic method invocation: To invoke a method on an object, the JRE looks at the class of the receiving *object* to choose which version to execute.

For example, when asked to execute **label.paint(g)**, the Java environment does not look in the *declared class* of **label** (namely, **JLabel**).

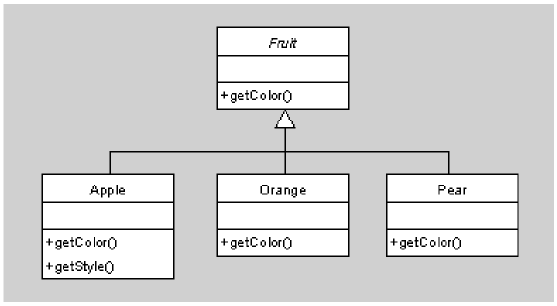
Instead it chooses the **paint** method in the *actual class* of the object referred to by **label** (namely, **MySpiffyLabel**).

When a method is called on an object of a subclass that overrides a superclass method, the *overriding* version of the method is always called.

Let us consider a rather tricky, but illustrative, example.

Abstract class **Fruit** has subclasses **Apple**, **Orange**, and **Pear**.

Since it is an abstract class, its name is shown in italics in the class diagram.



Note that Apple has a **getStyle()** method to return the kind of apple (**Delicious**, **McIntosh**, etc.).

Because of subtype polymorphism, it is legal to declare a variable as being of some class and then assign an object of a subclass to it:

```
Fruit fruit = new Apple("McIntosh");
```

Suppose that we have several fruits, and want to print out the colors of each. This code will do the trick:

```
Fruit[] A = new Fruit[3];
```

```
A[0] = new Apple("Granny Smith");
A[1] = new Orange();
A[2] = new Pear();
for( int i = 0; i < A.length; i++ ) {
  if( A[i] instanceof Apple )
    System.out.println(
      ((Apple) A[i]).getColor());
  else if( A[i] instanceof Orange )
    System.out.println(
      ((Orange) A[i]).getColor());
  else if( A[i] instanceof Pear )
    System.out.println(((Pear)
A[i]).getColor());
  else
    System.out.println(A[i].getColor());
```

What's wrong with this? If you add new classes, you have to modify every such test to add to the options in the **if** statement.

How can we simplify it?

```
for (int i = 0; i < A.length; i++ )
  System.out.println(A[i].getColor()); // or …

for (Fruit a: A) System.out.println(a.getColor());
```

What would happen if no `getColor` method were defined in `Fruit`? A compilation error; can't compile a getColor call to a Fruit if no getColor is declared for fruit.

Overloading vs. overriding

Two methods are *overloaded* if they are in the same class, but have different parameter lists.

When a method is *overridden*, one of its *subclasses* declares a method of the same name, with the same signature.

Consider this example. All of our `Fruits` inherit an `equals` method from class `Object`. Suppose that `Fruit` declares its own `equals` method:

```
Object>>public boolean equals(Object obj)      (1)

Fruit>>public boolean equals(Fruit fruit)      (2)
```

Has `Fruit` overridden the `equals` method? The parameter types are different, so it can't be overriding; it must be overloading.

Which `equals` method is called in each case below?

```
Object o = new Object();
Fruit f = new Fruit();
Object of = new Fruit();
f.equals(o);
f.equals(f);
f.equals(of);
```

What about these calls, using the same variables?

```
o.equals(o);
o.equals(f);
o.equals(of);
of.equals(o);
of.equals(f);
of.equals(of);
```

Now, let's throw overriding into the picture and declare, in class `Fruit`—

```
Object>>public boolean equals(Object obj)      (1)
Fruit>>public boolean equals(Fruit fruit)      (2)
Fruit>>public boolean equals(Object obj)       (3)
```

Which methods are called now?

```
Object o = new Object();
Fruit f = new Fruit();
Object of = new Fruit();
f.equals(o);
f.equals(f);
f.equals(of);
o.equals(o);
```

```
o.equals(f);
o.equals(of);
of.equals(o);
of.equals(f);
of.equals(of);
```

In summary, the compiler decides which overloaded method to call by looking at the declared type of

- the object being sent the message and
- the declared types of the arguments to the method call.

The particular version of the overloaded method is chosen at runtime by dynamic method invocation using the actual type of the object being sent the message.

The actual classes of the arguments to the method call do not play a role.

This is very different from a language like CLOS, which uses the actual types of the arguments to decide which method to execute.

Exercise: Singleton pattern

In the Week 5 video lecture, we saw the Singleton pattern defined in Ruby.

```
require 'singleton'
class Registry
  include Singleton
  attr_accessor :val
end
r = Registry.new #throws a NoMethodError
r = Registry.instance
r.val = 5
s = Registry.instance
puts s.val   >> 5
s.val = 6
puts r.val  >> 6
s.dup  >> TypeError: can't duplicate instance of singleton Registry
```

The idea is to prevent more than one object of the class from being defined, and to return the single instance by using a class method.

Here is an exercise with another Singleton pattern, except blanks are left in the code. You need to fill in the blanks to get the code to run.

```
class Balance
  attr_reader __________(1)__________

  def __________(2)__________(balance)
    @balance = balance
    __________(3)__________ = nil
  end

  def __________(4)__________.instance
    @first_instance = __________(5)__________(100)
                          if @first_instance.nil?
    __________(6)__________
  end

  def withdraw(amount)
    @balance > amount ? (@balance -= amount) :
           (puts 'Insufficient balance')
  end

  def deposit(amount)
    @balance += amount
  end
end

class FamilyMember
  def initialize(name)
    @name = name
    @balance = Balance.__________(7)__________
  end

  def withdraw(amount)
    __________(8)__________(amount)
  end

  def deposit(amount)
    __________(9)__________(amount)
  end

  def balance
    __________(10)__________
  end
end
```

Fill in the blanks in the `Singleton` class and the `FamilyMember` class. Note that Singleton is not implemented as a mixin, though it could be.

Exercise: Adapter Pattern

An *adapter* allows classes to work together that normally could not because of incompatible interfaces.

- It "wraps" its own interface around the interface of a pre-existing class. What does this mean?

- It may also translate data formats from the caller to a form needed by the callee.

One can implement the Adapter Pattern using delegation in Ruby. Consider the following contrived example.

- We want to put a `SquarePeg` into a `RoundHole` by passing it to the hole's `peg_fits?` method.
- The `peg_fits?` method checks the `radius` attribute of the peg, but a `SquarePeg` does not have a radius.
- Therefore we need to adapt the interface of the `SquarePeg` to meet the requirements of the `RoundHole`.

```
class SquarePeg                     class RoundPeg
   attr_reader :width                  attr_reader :radius
   def initialize(width)               def initialize(radius)
      @width = width                      @radius = radius
   end                                 end
end                                 end

class RoundHole
    attr_reader :radius

    def initialize(r)
```

```
        @radius = r
    end

    def peg_fits?(peg)
        peg.radius <= radius
    end
end
```

Here is the Adapter class:

```
class SquarePegAdapter
    def initialize(square_peg)
        @peg = square_peg
    end

    def radius
        Math.sqrt(((@peg.width/2) ** 2)*2)
    end
end

hole = RoundHole.new(4.0)
4.upto(7) do |i|
    peg = SquarePegAdapter.new( SquarePeg.new(i.to_f) )
    if hole.peg_fits?( peg )
        puts "peg #{peg} fits in hole #{hole}"
    else
        puts "peg #{peg} does not fit in hole #{hole}"
    end
end
```

```
>>peg #<SquarePegAdapter:0xa038b10> fits in hole
#<RoundHole:0xa038bd0>
>>peg #<SquarePegAdapter:0xa038990> fits in hole
#<RoundHole:0xa038bd0>
>>peg #<SquarePegAdapter:0xa0388a0> does not fit in hole
#<RoundHole:0xa038bd0>
>>peg #<SquarePegAdapter:0xa038720> does not fit in hole
#<RoundHole:0xa038bd0>
```

Here is an exercise on the Adapter pattern. Fill in the blanks.

```
interface Bird
{
    // birds implement Bird interface that allows
    // them to fly and make sounds adaptee interface
    public void fly();
```

```
    public void ______(2)______();
}

class Sparrow implements ___(1)___
{
    // a concrete implementation of bird
    public void ____(4)___()
    {
        System.out.println("Flying");
    }
    public void makeSound()
    {
        System.out.println("Chirp Chirp");
    }
}

interface ToyDuck
{
    // target interface
    // toyducks dont fly they just make
    // squeaking sound
    public void squeak();
}

class PlasticToyDuck implements ToyDuck
{
    public void _____(3)____()
    {
        System.out.println("Squeak");
    }
}

class BirdAdapter implements ToyDuck
{
    // You need to implement the interface your
    // client expects to use.
    Bird bird;
    public BirdAdapter(Bird bird)
    {
        this.bird = bird;
```

```
    }

    public void squeak()
    {
        bird.______(5)_______();
    }
}

class Main
{
    public static void main(String args[])
    {
        Sparrow sparrow = new Sparrow();
        ToyDuck toyDuck = new PlasticToyDuck();

        // Wrap a bird in a birdAdapter so that it
        // behaves like toy duck
        ToyDuck birdAdapter = new BirdAdapter(sparrow);

        System.out.println("Sparrow...");
        sparrow.fly();
        sparrow.makeSound();

        System.out.println("ToyDuck...");
        toyDuck.squeak();

        // toy duck behaving like a bird
        System.out.println("BirdAdapter...");
        birdAdapter.squeak();
    }
}
```