Homework 9

Due 1 May

Handout 21
CSCI 334: Spring, 2012

Reading

1. **(Required)** Mitchell, Chapter 13.
2. **(Recommended)** Sun's descriptions of new features in Java 1.5:
   - `http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html`
   - `http://java.sun.com/developer/technicalArticles/releases/j2se15/`
3. **(Skim)** Scala Resources as needed for the programming.

Problems

1. (*40 points*) ..... Undoable Commands

The goal of this problem is to implement the core data structures of a text editor using the *Command Design Pattern*.

Text Editor In essence, a text editor manages a character sequence that can be changed in response to commands issued by the user, such as inserting new text or deleting text. Typically, these commands operate on the underlying character buffer at the current position of the cursor. Thus, if the cursor is positioned at the beginning of the buffer, typing the string "moo" will cause those letters to be inserted at the start of the buffer, and so on. This question explores the internal design of a simple editor.

Most text editors involve a GUI and the user issues commands to the editor by keyboard and mouse events. For us, however, the most interesting part of a text editor's is what happens behind the scenes. Therefore, our text editor will just be a simple command line program that prompts you for edit commands. The program will print the contents of the text editor's buffer, including a "^" to indicate the current cursor position, print the prompt "?", and then wait for you to enter a command. At one point in time, this was in fact how many text editors worked — look up "ed text editor" in Wikipedia, for example (or run it on our lab machines...). The following shows one run of our editor:

Sample Execution	Description
`Buffer:` `       ^`	*Buffer is initially empty, cursor at start*

`? I This is a test.`
`Buffer: This is a test.`
`                    ^`
	Insert "This is a test." and move cursor to immediately after inserted text

`? < 9`
`Buffer: This is a test.`
`             ^`
	Move cursor 9 characters left

`? >`
`Buffer: This is a test.`
`              ^`
	Move cursor 1 character right

`? I n't`
`Buffer: This isn't a test.`
`                ^`
	Insert "n't"

`? > 3`
`Buffer: This isn't a test.`
`                   ^`
	Move cursor 3 characters right

`? D 4`
`Buffer: This isn't a .`
`                   ^`
	Delete 4 characters.

`? I cow`
`Buffer: This isn't a cow.`
`                       ^`
	Insert "cow"

`? Q`	*Quit*

Here is a summary of all available editor commands (including some described below). The term *[num]* indicates an optional number.

Command	Description
`I` *text*	Insert *text* at the current cursor, moving cursor to after the new text.
`D` *[num]*	Delete *num* characters to the right of cursor position. (If *num* is missing, delete 1 character.)
`<` *[num]*	Move the cursor *num* characters to the left. (If *num* is missing, move 1 character.)
`>` *[num]*	Move the cursor *num* characters to the right. (If *num* is missing, move 1 character.)
`Q`	Quit
`U`	Undo the previous edit command
`P`	Print the history of edit commands
`R`	Redo an undone edit command

I have provided a working program for all but the last three commands. Your job is to change `TextEditor` to support multiple levels of undo and redo using the *Command Design Pattern*.

Figure 1 shows an example that uses "`U`" (undo) and "`P`" (print history). (We'll look at "`R`" (redo) at the very end of the problem.) Notice that you can undo multiple edits, not simply the last one. To support this, the text editor must keep track of an edit command history that permits you to undo as many commands from the history as you like. Undoing all commands will lead you all the way back to the original empty buffer.

The starter code for this problem is divided into two classes:

- `Buffer`: This class manages the internal state of the editor's buffer (ie, character sequence and current cursor location), and it supports commands for getting/setting the cursor location and for inserting/deleting text. Refer to the javadoc on the handouts page for more details. *You should not change this class.*

Sample Execution	Description

`Buffer:`
`        ^`
`? I Hello`
`Buffer: Hello`
`             ^`
`? < 2`
`Buffer: Hello`
`          ^`
`? D 2`
`Buffer: Hel`
`          ^`
`? I p`
`Buffer: Help`
`           ^`	

`? U`
`Buffer: Hel`
`          ^`
	Undo the previous command.

`? I ium`
`Buffer: Helium`
`              ^`	

`? P`
`History:`
`    [Insert "Hello"]`
`    [Move to 3]`
`    [Delete 2]`
`    [Insert "ium"]`
`Buffer: Helium`
`              ^`
	Print the command history.

`? U`
`Buffer: Hel`
`          ^`
	Undo the last command (`[Insert "ium"]`*).*

`? U`
`Buffer: Hello`
`          ^`
	Undo the last command (`[Delete 2]`*).*

`? P`
`History:`
`    [Insert "Hello"]`
`    [Move to 3]`
`Buffer: Hello`
`          ^`
	Print the command history.

`? Q`	

Figure 1: Sample run of the text editor with undo.

- `TextEditor`: This class stores a `Buffer` named `buffer`. The `processOneCommand()` method reads in a command from the user and performs the appropriate operation on `buffer` by invoking one of the following methods:
  - `protected def setCursor(loc: Int): Unit`
  - `protected def insert(text: String): Unit`
  - `protected def delete(count: Int): Unit`
  - `protected def undo(): Unit`
  - `protected def redo(): Unit`
  - `protected def printHistory(): Unit`

  These methods are all quite simple. For example, the `insert` method simply inserts the text into `buffer` and repositions the cursor:

```
protected def insert(text: String) = {
    buffer.insert(text);
    buffer.setCursor(buffer.getCursor() + text.length());
}
```

The `EditCommand` Class To support undo, we first change the way the `TextEditor` operates on the underlying `buffer`. Rather than changing it directly, the `TextEditor` constructs `EditCommand` objects that know how to perform the desired operations and — more importantly — know how to undo those operations. All `EditCommand` objects will be derived from the `EditCommand` abstract class:

```
abstract class EditCommand(val target: Buffer) {

  /** Perform the command on the target buffer */
  def execute(): Unit;

  /** Undo the command on the target buffer */
  def undo(): Unit;

  /** Print out what this command represents */
  def toString(): String;
}
```

Here, the `execute()` method carries out the desired operation on the `target` buffer, and `undo()` would perform the inverse operation. For example, to make insert undoable, the first step would be to define an `InsertCommand` class in a new file `InsertCommand.scala`:

```
class InsertCommand(b: Buffer, val text: String) extends EditCommand(b) {
  override def execute(): Unit = { ... }
  override def undo(): Unit = { ... }
  override def toString(): String = { ... }
}
```

The `TextEditor` would then perform code like the following inside `insert`:

```
protected def insert(text: String) = {
    val command = new InsertCommand(buffer, text);
    command.execute();
    ...
}
```

Assuming `InsertCommand` is implemented properly, the insertion would happen as before. However, the `TextEditor` can now remember that the last operation performed was the `InsertCommand` we created, and we can undo it simply by calling that object's `undo()` method. In essence, an `EditCommand` object describes one modification to a `Buffer`'s state and how to undo that modification. Supporting undo is then as simple as writing a new kind of `EditCommand` object for each type of buffer modification you support.

And of course, to implement multiple levels of undo, you need to keep track of more than just the last command object created...

Implementation Strategy I suggest tackling the implementation the following steps:

(a) Download the starter code from the handouts page. Compile the Scala files with the command `fsc *.scala` as usual. I have added some `assert` statements to the `Buffer` class to aid in debugging. The general form is

```
assert(condition, { "message" })
```

You may find it useful to add similar asserts to your own code as well.

(b) Implement `InsertCommand`, `DeleteCommand`, and `MoveCommand` subclasses of `EditCommand`. For each one, you must define: 1) `execute()`, (2) `undo()`, and (3) `toString()`. I recommend holding off on `undo()` for the moment. Change `TextEditor` to create and `execute` edit command objects appropriately.

(c) Extend `TextEditor` to remember the last command executed, and change `TextEditor`'s `undo()` method to undo that command. Go back and implement `undo` for each type of `EditCommand`.

(d) Once a single level of undo is working, extend `TextEditor` to support undoing multiple previous commands. Specifically, change `TextEditor` to maintain a history of commands that have been executed and not undone. Also implement the `printHistory()` method to aid in debugging. Your program should simply ignore undo requests if there are no commands are in the history. You are free to use any Scala libraries you like in your implementation (ie, any immutable or mutable collection class).

(e) The last task is to implement redo. Specifically, if you undo one or more commands but have not yet performed any new operations on the buffer, you can redo the commands you undid:

Sample Execution	Description
`Buffer: hello` `  ^`	

`? D 1`
`Buffer: helo`
`  ^`	

`? D 1`
`Buffer: hel`
`  ^`	

`? U`
`Buffer: helo`
`  ^`
	Undo delete of "o"

`? U`
`Buffer: hello`
`  ^`
	Undo delete of "l"

`? R`
`Buffer: helo`
`  ^`
	Redo delete of "l"

`? R`
`Buffer: hel`
`  ^`
	Redo delete of "o"

`? I p`
`Buffer: help`
`   ^`	

`? U`
`Buffer: hel`
`  ^`
	Undo insert of "p"

`? U`
`Buffer: helo`
`  ^`
	Undo redone delete of "o"

Note that redoing undone commands is no longer possible if the buffer is changed in any way. For example, if you insert text after undoing some command *E*, you should no longer be able to redo command *E*:

Sample Execution	Description
`Buffer:` `  ^`	

`? I moo`
`Buffer: moo`
`  ^`	

`? U`
`Buffer:`
`  ^`	

`? I hello`
`Buffer: hello`
`   ^`
	Change buffer after undo

`? R`
`Buffer: hello`
`   ^`
	Redo will have no effect

Also, redone commands should be able to be subsequently undone:

Sample Execution	Description
`Buffer:` `        ^`	

`? I 334`
`Buffer: 334`
`           ^`	

`? I cow`
`Buffer: 334cow`
`                 ^`	

`? I moo`
`Buffer: 334cowmoo`
`                       ^`	

`? U`
`Buffer: 334cow`
`                 ^`
	Undo insert of "moo"

`? U`
`Buffer: 334`
`           ^`
	Undo insert of "cow"

`? R`
`Buffer: 334cow`
`                 ^`
	Redo insert of "cow"

`? R`
`Buffer: 334cowmoo`
`                       ^`
	Redo insert of "moo"

`? U`
`Buffer: 334cow`
`                 ^`
	Undo insert of "moo"

`? U`
`Buffer: 334`
`           ^`
	Undo insert of "cow"

`? U`
`Buffer:`
`        ^`
	Undo insert of "334"

`? R`
`Buffer: 334`
`           ^`
	Redo insert of "334"

`? R`
`Buffer: 334cow`
`                 ^`
	Redo insert of "cow"

`? U`
`Buffer: 334`
`           ^`
	Undo insert of "cow"

Extend `TextEditor` to support multiple levels of redo. You should not need to change any class other than `TextEditor` to implement this feature.

(f) Turn in your code using `turnin` as in the previous homeworks.

There are many extensions that would make our editor more "realistic". One idea is listed below as an extra credit problem. It should not require more than a few additional lines of code and really highlights the elegance and simplicity of adopting this design pattern.

2. (*10 points*) ..... Subtyping and Exceptions

Mitchell, Problem 13.3

3. (*10 points*) ..... Array Covariance in Java

Mitchell, Problem 13.5

4. (*15 points*) ..... Java Interfaces and Multiple Inheritance

In C++, a derived class may have multiple base classes. The designers of Java chose not to allow multiple inheritance. Therefore, a Java derived class may only have one base class. However, Java programs may contain interfaces (roughly, classes without an implementation) and a class may be declared to implement more than one interface. This question asks you to compare these two language designs.

This question asks you to consider the following kinds of movies.

Movie: a class describing all kinds of movies

Action: a movie containing lots of explosions

Romance: a movie where romantic interest drives the plot

Comedy: a movie with largely humorous content

Mystery: a who-dunnit movie

Rescue: a hybrid action-romance movie, where the main character attempts to save his or her romantic interest from almost certain doom

Romantic Comedy: a hybrid romance-comedy with large amounts of both humorous and romantic content

Hollywood Blockbuster: an action-romance-comedy-mystery movie designed to please crowds

(a) Draw a C++ class hierarchy with multiple inheritance for the above set of classes.

(b) If you were to implement these classes in C++ for some kind of movie database, what kind of potential conflicts associated with multiple inheritance might you have to resolve?

(c) If you were to represent this hierarchy in Java, what interfaces and classes would you use? Write your answer by carefully drawing a class/interface hierarchy, identifying which nodes are classes and which are interfaces. Note that there must be a class for each of the movie genres, but you may use any interfaces you require to preserve the relationships between genres. For example, one way of doing this would be to have the **Comedy** and **Romantic-Comedy** genres both implement some kind of **IComedy** interface.

(d) Give an advantage of C++ multiple inheritance over Java classes and interfaces and one advantage of the Java design over C++.

(e) Assuming you will need to create objects of every movie type (other than `Movie`), do Scala traits offer a better solution in this case? Why or why not?

5. (*10 points*) ..... Adding Pointers to Java

Java does not have general pointer types. More specifically, Java only has primitive types (Booleans, integers, floating point numbers, ...) and reference types (objects and arrays). If a variable has a primitive type, then the variable is associated with a location, and a value of that type is stored in that location. When a variable of primitive type is assigned a new value, a new value is copied into the location. In contrast, variables of reference types are implemented as pointers. When

a reference-type variable is assigned, Java copies a pointer to the appropriate object into the location associated with the variable.

Imagine that you were part of the Java design team and you believe strongly in pointers. You want to add pointers to Java, so that for every type `A`, there is a type `A*` of pointers to values of type `A`. Gosling is strongly opposed to adding an "address of" operator (like `&` in `C`), but you think there is a useful way of adding pointers without adding address-of.

One way of designing a pointer type for Java is to consider `A*` equivalent to the following class:

```
class A* {
      private A data;

      A*() {
           data = null;
      }

      public void assign(A x) {
             data=x;
      }
      public A deref() {
             return data;
      }
}
```

Intuitively, a pointer is an object with two methods, one assigning a value to the pointer and the other dereferencing a pointer to get the object it points to. One pointer, `p`, can be assigned the object reached by another, `q`, by writing `p.assign(q.deref())`.

(a) If `A` is a reference type, do `A*` objects seem like pointers to you? More specifically, suppose `A` is a Java class with method `m` that has a side effect on the object. Consider the following code:

```
A x = new A(...);
A* p = new A*();
p.assign(x);
(p.deref()).m();
```

Here, pointer `p` points to the object named by `x` and `p` is used to invoke a method. Does this modify the object named by `x`? Answer in one or two sentences.

(b) What if `A` is a primitive type, such as `int`? Do `A*` objects seem like pointers to you? (Hint: Think about code similar to that in part (a).) Answer in one or two sentences.

(c) If `A <: B`, should `A* <: B*`? Answer this question by completing the following chart and explaining the relationship:

Method	Type
A*.assign	
B*.assign	
A*.deref	
B*.deref	

(d) Can you generalize the issue discussed in part (c) to Java generics? More specifically, What might happen if you had a pointer generic? Based on the `Ptr` example, do you think it is correct to assume that for every generic class `Template`, if `A <: B` then `T<A> <: T<B>`? Explain briefly.

6. (*20 points*) ..... Upper and Lower Bounds on Types

Type parameters in Scala (and Java) can be given an upper and lower bounds to restrict how they can be instantiated. Specificically, the type `List[_ <: C]` describes a list that stores some data of some type that is a subtype of of class `C`. In other words, the type parameter has an upper bound `C`. For example, an object of `List[_ <: Point]` is a list that contains objects which extend the `Point` class. For example, the list could be `List[Point]` or `List[ColorPoint]`, etc. Reading an element from such a list is guaranteed to return a `Point`, but writing to the list is not generally allowed. This sort of bounded type is often called an existential type because it can be interpreted as "there exists some type T such that T `<: C`.

Existential types can also have lower bound constraints. A constraint `[_ >: C]` means that the existential type must be a supertype of class `C`. For example, an object of `List[_ >: Point]` could be a `List[Point]` or `List[Any]`. (`Any` is the supertype of all types in Scala, and serves similar purposes as `Object` in Java). Reading from such a list returns objects of type `Any`, but any object of type `Point` can be added to the list.

This question asks about generic versions of a simple function that reads elements from one list and adds them to another. Here is sample non-generic code, in case this is useful reference in looking at the generic code below.

```
def addAllNonGeneric(src: MutableList, dest: MutableList) : Unit = {
  for (o <- src) {
    dest += o;
  }
}
val listOfPoints = new MutableList();
val listOfColorPoints = new MutableList();
...
addAllNonGeneric(listOfColorPoints, listOfPoints);
```

It will not compile in Scala, but gives you an idea of what we're trying to do.

(a) The simplest generic version of the `addAll` method uses an unconstrained type parameter and no bounds.

```
def addAll0[T](src: MutableList[T], dest: MutableList[T]) : Unit = {
  for (o <- src) {
    dest += o;
  }
}
```

Suppose that we declare

```
val listOfPoints : MutableList[Point] = new MutableList[Point]();
val listOfColorPoints : MutableList[ColorPoint] = new MutableList[ColorPoint]();
```

and call

```
addAll0(listOfColorPoints, listOfPoints).
```

Will this call compile or will a type error be reported at compile time? Explain briefly.

(b) With `listOfColorPoints` and `listOfPoints` defined as in the previous part of this question, will the call

```
addAll1(listOfColorPoints, listOfPoints)
```

compile, where `addAll1` is defined as follows:

```
def addAll1[T](src: MutableList[_ <: T], dest: MutableList[T]) : Unit = {
  for (o <- src) {
    dest += o;
  }
}
```

Explain briefly.

(c) With `listOfColorPoints` and `listOfPoints` defined as in the previous part of this question, will the call

```
addAll2[ColorPoint](listOfColorPoints, listOfPoints)
```

compile, where `addAll2` is defined as follows:

```
def addAll2[T](src: MutableList[T], dest: MutableList[_ >: T]) : Unit = {
  for (o <- src) {
    dest += o;
  }
}
```

Explain briefly. (The explicit instantation of `T` in the call to `addAll2` is needed because of how Scala infers types.)

(d) Suppose we want to change our function so that in addition to adding elements of one list to another list, we also return the last element added. Code for this static method is written below. Fill in each of the blanks with type parameters, wildcards, and/or constraints so that the code will type-check at compile time and will have the most flexible type possible.

```
def addAll[T](src: MutableList[ _______ ],  dest: MutableList[ _______ ]) : T = {
  var last : T = null;
  for (o <- src) {
    dest += o;
    last = o;
  }
  return last;
}
```

(e) In the code in part d) above, suppose `src` has type `MutableList[R]` and `dest` has type `MutableList[S]`. What subtype/supertype relationships between `R`, `S`, and `T` are needed for the body of the method to be type-correct?

(f) In your solution to part d) above, how do the constraints you wrote guarantee the subtype/-supertype relationships between `R`, `S`, and `T` you listed in part e)?

(g) Suppose that your friend comes across the following general programming suggestion on the web and asks you to explain. What can you tell your friend to explain the connection between the programming advice and principles of subtyping, showing off your understanding gained from CS 334?

> **Get and Put Principle:** If you pass a generic structure to a function:
>
> - Use an existential type with an upper bound (ie, `[_ <: T]`) when you only GET values out of the structure.
> - Use an existential type with a lower bound (ie, `[_ >: T]`) when you only PUT values into the structure.
> - Don't use an existential type when you both GET and PUT values out of / into the structure.

7. (20 points) ..... Template Expansion versus Type Erasure

Java and C# implement generics in two different ways. Java generics are implemented using an erasure technique discussed in class that inserts casts and uses one run-time copy for all instances of the generic. Although C# is similar to Java in some respects, the C# implementation of generics may produce separate copies if several instances of a generic are used in a program. The C# implementation has some general similarities to the C++ implementation of templates.

(a) Java chose the erasure and cast implementation of generics for several reasons. One was the large number of users running old Java VMs and a huge amount of legacy code. Why does the type erasure and casts implementation support both legacy code and old VMs better than an implementation that uses expansion.

(b) You are sitting in Tunnel City Coffee and overhear a conversation between our old friends Rocky and Bullwinkle. Rocky says:

> "I think C# made the wrong choice when adding generics to their language: they've invalidated all their old code and VMs. Java does it right."

Bullwinkle replies:

> "But what if your old code starts adding `Strings` onto your `Vector<Integer>`? This is wrong, but the error is not detected until you use the inserted `String`. It should be detected when the old code inserts a `String` into a `Vector<Integer>`."

Using that fact that `Vector<Integer> <: Vector`, write a short program that exhibits the problem Bullwinkle describes.

(c) Rocky replies:

> "I have a solution to this problem. I just use this special `MyVector`:
>
> ```
> class MyVector<T> extends Vector {
>   void add (Object a) {
>     try {
>       T tmp = (T)a;
>       super.add(tmp);
>     } catch (ClassCastException c) {
>       System.out.println("Error Detected");
>     }
>   }
> }
> ```
>
> It will catch errors at run time right when a program tries to add a `String` into a `Vector<Integer>`, instead of waiting until the `String` is used as an `int`.

Bullwinkle rolls his eyes and tells Rocky that his scheme would never detect those insertion errors when they happen, even at run time. Who is right? Explain why.

(d) Translate the following code into the code that could run on an older VM without generics (i.e. Java 1.4 or earlier). Remember that `ints` are value types that are not subtypes of `Object`.

```
MyVector<Integer> a = new MyVector<Integer>();
a.add (5);
int j = a.lastElement();
```

Fill in the following code:

```
MyVector a = new MyVector();
a.add (_________________________________5_______)
int j = ____________________ a.lastElement() ____________________ ;
```

(e) Sometimes Java's use of erasure and casting has very bizarre effects. In this case, write the console output of the following program.

```
class Container<T> {
  public T internal;
  public static Container lastInstance;
  Container(T value){
    internal = value;
    lastInstance = this;
  }
}

class Main {
  public static void main(String args[]) {
    Container<String> str = new Container<String>("Cow");
    Container<Integer> myint=new Container<Integer>(31337);
    System.out.println("Last Instance: " + str.lastInstance.internal);
  }
}
```

Explain why it behaves this way in terms of generics and Java erasure.

(f) C++ uses the expansion method for its templates instead of Erasure and inserted casts. What does the output of the same program translated into C++ look like. Write the output and explain why it is different or why it remains similar even with a different template expansion mechanism.

(g) It is sometimes useful to create "mixins."* A mixin class is a class that inherits from a template parameter like so:

```
class MyClass <T> extends T {
  ...
}
```

Why is this construct ineffective when used with erasure and casting?

8. (*15 points*) ..... (Bonus Question) Composable Commands

Here is one interesting extension to the basic Text Editor.

Most of the time, two consecutive commands of the same type are lumped together into a single command. Thus, if I type "hello" followed immediately by " there" into an editor (such as emacs), the editor lumps them together into a single insertion command that removes all of "hello there" from the buffer when undone. Similarly, if I perform two cursor movement commands in a row, that is recorded in the undo history as a single command. Here is an example:

*The name was inspired by Steve's Ice Cream Parlors. (No relationship to the instructor, though I ate a lot of Steve's Ice Cream back in the day.) The ice cream shop had several flavors of ice cream and blended in extra items (chocolate chips, nuts, cookies, etc.) upon request. The result was a "Mix-In."

Sample Execution	Description
`Buffer:` `        ^`	

`? I hel`
`Buffer: hel`
`            ^`	

`? I ium`	*second insert composed with first*
`Buffer: helium` `                ^`	

`? P`
`History:`
`    [Insert "helium"]`
`Buffer: helium`
`                ^`	

`? U`
`Buffer:`
`        ^`	

`? R`
`Buffer: helium`
`               ^`	

`? <`
`Buffer: helium`
`              ^`	

`? < 2`	*second move composed with first*
`Buffer: helium` `            ^`	

`? D 2`
`Buffer: helm`
`            ^`	

`? D 1`	*second delete composed with first*
`Buffer: hel` `            ^`	

`? P`
`History:`
`    [Insert "helium"]`
`    [Move to 3]`
`    [Delete 3]`
`Buffer: hel`
`            ^`	

`? I p`
`Buffer: help`
`             ^`	

`? U`	*undo insert of "p"*
`Buffer: hel` `            ^`	
`? U`	*undo composed delete command*
`Buffer: helium` `            ^`	
`? U`	*undo composed move command*
`Buffer: helium` `               ^`	
`? U`	*undo composed insert of "helium"*
`Buffer:` `        ^`	

Implement composable commands. A good way to start is to extend the `EditCommand` class and its subclasses to define the following method:

```
def compose(other : EditCommand) : Option[EditCommand]
```

This method either:

- returns `None` if the current command cannot be composed with `other`.
- returns a new command if the current command can be composed with `other`, because, for example, they are both insert commands. In this case, the method should also change the current command to be the composed command.

For example,

```
val c1 = new InsertCommand(target, "hel");
val c2 = new InsertCommand(target, "lo"));
c1.compose(c2) match {
  case None     => // can't combine them
  case Some(c3) => c3.execute();
}
```

would create the command `c3` that inserts "hello" into the target. If we changed `c2` to be a `DeleteCommand`, the `compose` operation would return `None`. You may find it useful to test whether an object has a certain type, which can be done in Scala with pattern matching, as in:

```
x match {
  case i : InsertCommand  => ... // x is an InsertCommand, now bound to i
  case i : DeleteCommand  => ... // x is an DeleteCommand, now bound to i
  case i                  => ... // match all other types
}
```

9. (*20 points*) ..... (Bonus Question) Compiling Scala Traits

Scala `traits` strike a compromise between Java's interfaces and full multiple inheritance. How are they implemented in the Scala run time? That is, what get's generated when you compile a program using traits, and how does the language feature and translation circumvent the complexities of multiple inheritance:

- How does it ensure subtypes have a uniform representation, as in C++ and Java?
- What happens to programs exhibiting the diamond pattern?
- How are ambiguous method invocations resolved?

You can generate small examples and see how they are translated using the following tools:

- "`scalap -private -classpath . A`" prints a "Scala" summary of the compiled class `A` in the current directory.
- "`javap -c -private -classpath . A`" prints a "Java" summary of the compiled class `A`, including the bytecode instructions.

You may notice that compiling a program with traits results in additional "helper" classes being generated...

This is intentionally a vague question. Explore a bit and see what you can learn about trait generation (and Scala compilation in general).