A bi-directional builder

Based on an idea from Allen Holub

Can we create objects without getters/setters?

Imagine that you want to write a class representing a Contact in a contact book, but you don’t want to expose the implementation details via getters and setters.

At the same time, you want your Contact objects to be able to be presented in a user interface (text-based or GUI-based).

How could you do that, without at least getters?

The UI component must query the object for its members somehow, right?

Provide a builders for construction and export

You can achieve this behavior by declaring two interfaces in Contact.

One Importer and one Exporter:

```
public interface Exporter{    // Lets you export this contact to a builder
  void addName(String name);
  void addEmail(String email);
  void addPhone(String phone);
}

public interface Importer{    // Lets you create a Contact from a builder
  String provideName();
  String provideEmail();
  String providePhone();
  void   open();
  void   close();
}
```

The constructor of Contact now becomes...

```
  // Constructs a Contact from the builder provided
  public Contact(Importer builder){
    builder.open();
    this.name  = new Name(builder.provideName());
    this.email = new Email(builder.provideEmail());
    this.phone = new Phone(builder.providePhone());
    builder.close();
  }

// We have decoupled the way a Contact is created!
```

Example of text-based importer

```
import java.util.Scanner;
public class ContactTextImporter
       implements Contact.Importer{
  public String provideName();
    return askFor("Name: ");
  }
  public String provideEmail(){
    return askFor("Email: ");
  }
  public String providePhone(){
    return askFor("Phone: ");
  }
  public void open(){
    showHeader();
  }
  public void close(){
    showBye();
  }
  private void showHeader(){
    System.out.println("===New Contact Form===");
  }
  private void showBye(){
    System.out.println("Bye!");
  }
  private String askFor(String prompt){
    try{
      System.out.print(prompt);
      return new Scanner(System.in).nextLine();
    }catch(Exception e){
      throw new RuntimeException("Error getting " +
                                 prompt + " " + e);
    }
  }
}
```

Example of Swing importer

```
public class ContactSwingImporter implements Contact.Importer{
  // lots of components
  public String provideName(){
    return nameTextField.getText();
  }
  public String provideEmail(){
    return emailTextField.getText();
  }
  public String providePhone(){
    return phoneTextField.getText();
  }

  // more stuff
}
```

Layout of the ContactSwingImporter

```
private void initializeComponents(){
  parent = new JFrame();
  frame = new JDialog(parent, "New Contact Form", true);
  panel = new JPanel();
  // 4 rows, 2 cols
  panel.setLayout(new GridLayout(4,2));
  nameTextField  = new JTextField(20);
  emailTextField = new JTextField(20);
  phoneTextField = new JTextField(20);
  nameLabel   = new JLabel("Name:");
  emailLabel  = new JLabel("Email:");
  phoneLabel  = new JLabel("Phone:");
  empty       = new JLabel(""); // left of the button
  okButton    = new JButton("OK");
  okButton.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent ae){
        parent.dispose();
      }
    });
}
```

Layout of the ContactSwingImporter

```
private void layoutComponents(){
  panel.add(nameLabel); panel.add(nameTextField);
  panel.add(emailLabel); panel.add(emailTextField);
  panel.add(phoneLabel); panel.add(phoneTextField);
  panel.add(empty); panel.add(okButton);
  frame.add(panel, BorderLayout.CENTER);
  frame.pack();
}
@Override // from the interface
public void open(){
  initializeComponents();
  layoutComponents();
  frame.setVisible(true);
}
@Override // from the interface
public void close(){
  // possible cleanup
}
```

Example usage in test program

```
public class TestContact{
  public static void main(String[] args){
    Contact c=null;
    if(args.length!=0){
      if(args[0].equalsIgnoreCase("text")){
        c = new Contact(new ContactTextImporter());
      }else if(args[0].equalsIgnoreCase("gui")){
        c = new Contact(new ContactSwingImporter());
      }
    }
    System.out.println("Contact: " + c);
  }
}
```

What about exporting *to* the UI?

Let’s now take a look at how a UI could use the Contact class.

The Contact.Exporter interface defines the following methods:

```
void addName(String name);
void addEmail(String email);
void addPhone(String phone);
```

We provide the builder we'd like to use in the UI

So, if a GUI would like to display a Contact, it supplies a builder to the Contact's export method.

The export method of the Contact class does the following:

```
// Exports this contact to the builder provided as
// argument
public void export(Exporter builder){
  builder.addName(name.toString());
  builder.addEmail(email.toString());
  builder.addPhone(phone.toString());
}
```

When the builder has all information, use it

When the builder has been given all information about the Contact, the UI can simply use it as it sees fit.

Note that there is still a coupling between the Contact and the builder, so if the Contact changes, there is a chance that the builder too has to change.

But the UI is protected from this change.

Good thing about the Exporter builder

The Exporter builder can be used by a UI like a Swing GUI.

But more importantly, it can be used to export to a number of formats!

Let’s say that we’d like to create a line of CSV from a Contact for some system, and a HTML table for another system.

The Exporter interface let’s us add new formats without having to affect the Contact class. We’d just create a new Exporter implementation.

HTMLExporter example

```
public class ContactHTMLExporter implements Contact.Exporter{
  private String name;
  private String email;
  private String phone;
  public void addName(String name){
    this.name = name;
  }
  public void addEmail(String email){
    this.email = email;
  }
  public void addPhone(String phone){
    this.phone = phone;
  }
  @Override
  public String toString(){
    // return name, email and phone as html
    // we should probably return a row with cells, not a complete table
    // depending on the usage
  }
}
```

Using the HTMLExporter

```
public class TestContact{
  public static void main(String[] args){
    Contact c=null;
    if(args.length!=0){
      if(args[0].equalsIgnoreCase("text")){
        c = new Contact(new ContactTextImporter());
      }else if(args[0].equalsIgnoreCase("gui")){
        c = new Contact(new ContactSwingImporter());
      }
    }
    System.out.println("Contact: " + c);
    ContactHTMLExporter html = new ContactHTMLExporter();
    c.export(html);
    System.out.println("HTML:");
    System.out.println(html);
  }
}
```

What about exporting to a GUI?

We’d just write an exporter which creates and offers a JPanel (for instance) with the contact information.

```
import javax.swing.*;
import java.awt.*;
public class GUIExporter implements Contact.Exporter{
  private String name, email, phone;
  public void addName(String name)  { this.name  = name;  }
  public void addEmail(String email){ this.email = email; }
  public void addPhone(String phone){ this.phone = phone; }

  /* A getter! */
  public JPanel getJPanel(){
    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout(3,2));
    panel.add(new JLabel("Name:"));
    panel.add(new JTextField(name));
    panel.add(new JLabel("Email:"));
    panel.add(new JTextField(email));
    panel.add(new JLabel("Phone:"));
    panel.add(new JTextField(phone));
    return panel;
  }
}
```

What about the actual GUI?

```
// The test class does:
GUIExporter gui = new GUIExporter();
c.export(gui);
new SwingDisplay(gui.getJPanel());
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class SwingDisplay{
  // Components... frame, panel, menus
  public SwingDisplay(JPanel contactPanel){
    this.panel = contactPanel;
    initComponents();
    layoutComponents();
    show();
  }
  private void show(){ frame.setVisible(true); }
  private void initComponents(){   //create the components}
  private void layoutComponents(){ //layout the components}
}
```

```
public class TestContact{
  public static void main(String[] args){
    Contact c=null;
    if(args.length!=0){
      if(args[0].equalsIgnoreCase("text")){
        c = new Contact(new ContactTextImporter());
      }else if(args[0].equalsIgnoreCase("gui")){
        c = new Contact(new ContactSwingImporter());
      }
    }
    System.out.println("Contact: " + c);
    ContactHTMLExporter html = new ContactHTMLExporter();
    c.export(html);
    System.out.println("HTML:\n" + html);
    ContactCSVExporter csv = new ContactCSVExporter();
    c.export(csv);
    System.out.println("CSV:\n"+csv);
    System.out.println("Starting gui display...");
    GUIExporter gui = new GUIExporter();
    c.export(gui);
    new SwingDisplay(gui.getJPanel());
  }
}
```

What's the point of all this?

- Show you an interesting use of the Builder pattern
- Show you tell-don't-ask (almost)
- Show you a possible solution to "getting fields without getters"
- Show you how to decouple exporting to different formats
- Make you think about object orientation in a new way
  - Objects are not simply data holders
  - We can treat them as first-class citizens and ask them to work for us
  - Show you a way out of the getters/setters misery (you don't have to agree)