FormSoul Technology:

Fast, Secure and Easy Form Building

By
Nguyễn, M. Hải - Chief architect

Updated 07/11/2008

THN Solutions (Thiên Hải Ngọc manufacturing & trading company)
website: www.thnsolutions.com

Contents

Executive Summary

"Technology has become the limiting factor in executing a business strategy. We need to find new tools and techniques that allow us to bring technology's capabilities in line with business demands" [1]

As demand for information technology increases, the development costs for business software rises higher than ever due to rapidly increasing complexity and diminishing productivity. Our **FormSoul** technology dramatically improves the productivity in developing business application's user interface.

Even though software is assumed to deliver higher productivity for business, it is the software development that is falling behind. According to a research study of The Standish Group, on average only 16.2% of software projects are completed on-time and on-budget, 31.1% of projects will be cancelled before they ever get completed [2]. Over the years, software development technology has grown by leaps and bounds. The sheer volume of technologies available today is boggling to the mind. Yesterday, a developer is expected to be proficient in C++, VB, Java, Delphi, and a couple of databases. Today, the same developer finds himself struggling to keep up with .Net, SOA, Ajax, Flex, SilverLight, and a lot more of acronyms soup. To keep up with fast market changes, business asks for more complex software. To keep up with complex requirements, technologies are invented. To keep up with technologies, the developer, the human, is either required to learn or she will be brutally replaced.

As the results of our many studies in productivity, we have realized it is the human that is the most valuable asset and the most "optimizable" factor. The key is to have the correct technologies and management to enable the strengths of human. **FormSoul** is one of our technologies that are designed at improving the productivity of programmers. A large chunk of development effort is wasted for building, validating, and testing the user interface. **FormSoul** shrinks hundreds of codelines down to several simple ones, freeing the developer from coding repetitive, error-prone, vulnerable codes. Best of all, **FormSoul** architecture can practically works with any user-interface (UI) paradigms.

In short, **FormSoul** lifts burdens from a developer's soul with:

- No more boring codes
- No more Get/Set/Validate UI
- No more boring tests
- Fast, secure codes
- Simple methodology for both Windows and Web applications

Current State of Building Application User Interface

Unlike art designers who have the luxury of experimenting with colors and images, a user-interface programmer spends most of his time on crafting and testing repetitive codes .

Building a form (screen) typically includes the following tasks:

- Design : create and arrange UI controls
- Binding : retrieve data from UI controls or show existing data to UI controls
- Validate : check whether the data obtained are valid
- Tests : huge amount of testing effort is required to ensure that the form works as intended and users should not be able to crash the application by making mistakes on this form. This involves UI tests, automatic-tests, monkey tests, security tests, etc.

In current state, programmers have several limited choices: manual coding, code generation and auto-binding.

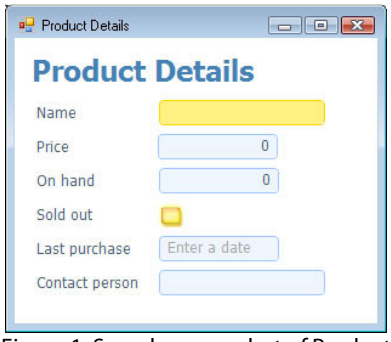

Figure 1: Sample screenshot of Product Details form

Manual Coding

This is the utmost painful solution to any programmer. Sadly, it is also the most common solution selected because it allows the greatest flexibility.

Assuming we are developing a Product Details form like *figure 1*, let's see how this simple form will be coded.

Example codes for Windows Application

Code 1.1: Retrieve data from input controls

```
        /// <summary>
        /// Gets input parameters from form
        /// </summary>
        Product GetInputs()
        {
            Product result = new Product();
            result.Name = txtName.Text;
            result.Price = nbPrice.ValueAsDecimal;
            result.OnHand = nbOnHand.ValueAsInt;
            result.SoldOut = chkSoldOut.Checked;
            result.LastPurchase = dtpLastPurchase.Date;
            result.ContactPerson = txtContactPerson.Text;
            return result;
        }
```

Code 1.2: Ensure the data obtained are valid

```
bool Validate(Product p)
{
    bool valid = true;

    //name must not be null, must be < 20 characters
    if ((p.Name != null) && (p.Name.Length < 20))
        valid &= true;
    else
        valid = false;

    //more validation codes

    return valid;
}
```

Code 1.3: Show existing data to form

```
/// <summary>
/// Show a product to form
/// </summary>
void ShowInfo(Product p)
{
    txtName.Text = p.Name;
    nbPrice.Value = p.Price;
    nbOnHand.Value = p.OnHand;
    chkSoldOut.Checked = p.SoldOut;
    dtpLastPurchase.Date = p.LastPurchase;
    txtContactPerson.Text = p.ContactPerson;
}
```

Example codes for Web Application

Thanks to great engineering, the code used in web application nowadays is pretty much similar to that in windows application. However, since web applications experience much more security vulnerabilities, a typical GetInputs() method needs to be "sanitized". Data must be normalized before any validation logic to deal with attacks such as cross-site scripting attack, SQL injection attack, etc.

Code 1.4: Retrieve data from a web form

```
/// <summary>
/// Gets input parameters from form
/// </summary>
Product GetInputs()
{
    Product result = new Product();
    result.Name =
     DataSanitizer.NormalizeString(txtName.Text);
    result.Price =
     DataSanitizer.NormalizeDecimal(nbPrice.ValueAsDecimal);
    result.OnHand =
     DataSanitizer.NormalizeInt(nbOnHand.ValueAsInteger);
    result.SoldOut =
     DataSanitizer.NormalizeBool(chkSoldOut.Checked);
    result.LastPurchase =
     DataSanitizer.NormalizeDateTime(dtpLastPurchase.SelectedDate);
    result.ContactPerson =
     DataSanitizer.NormalizeString(txtContactPerson.Text);
    return result;
}
```

The manual coding, like its name suggests, is manual, laborious, repetitive, and error-prone. Even though it allows the greatest flexibility and freedom, the programmer must repeat the same code for each and every input controls, form-after-form.

Generated Codes

Since binding and validation codes are repetitive and similar, auto-generated codes quickly reduce the time required for coding. Thus, a plethora of code generators has flourished. Some popular generators are CodeSmith, IronSpeed Designer, JavaGen, OpenXava, .netTiers [3].

"The road to bad code is paved with code generators"
- Assaf Arkin

Code Generator is a great tool. However, "the devil is in the details". Generator always requires careful use and well-planned design.

Template is king
At the heart, code generators use various templates for generating codes. The generated codes are as good as the templates. Be it data-tier or UI-tier, real-life applications cannot be generated using plain-vanilla templates built-in in generators. Someone must create the master template. That someone frequently is some developer whereas this job should be the task of an architect. Only the architect can thoroughly hand-craft a template that uses best-practices and patterns, is flexible, scalable, and secured. Time changes, things change, practices change. The once carefully crafted templates are obsolete. The already generated codes become binary wastes.

Thou must not change
Generated codes generally includes special base classes that are not to be changed. They should be considered as "work of art" because developers can only admire that codes without touching them. Modifications become taboo, or at least will break the established architecture that has been planned and built into the templates.

Maintenance is a nightmare
For repetitive and similar codes, hundreds of man-hour can be saved by just a click on button Generate. This one click is also the sure ticket to hell.

For a hypothetical project, an e-commerce website is generated. The project is almost finished with everything from data, logic to UI and CSS. Suddenly, external security audit reveals that all input forms are vulnerable to SQL injection attack. This leads to review, retest, redo everything from data up to UI layer. The sad news, generators cannot help you anymore. Good commercial generators may help revise the original templates and re-generate new base codes. But be prepared for impact, your codes may be lost, hidden bugs may creep up everywhere.

Five years down the road, things only get worse. The generator used becomes obsolete. The original developers left the boat. The codes stay. No one understands these codes anymore. The logics are buried under thousands of generated codelines. To actually fix something, one must dig up the artifacts by reading these cosmetic codes.

Automatic Binding

Almost all modern User-Interface kits include some forms of automatic binding. The following is a sample data binding code used in WPF (one of the latest presentation layers from Microsoft)

```
<DockPanel
  <DockPanel.Resources>
    <c:MyData x:Key="myDataSource"/>
  </DockPanel.Resources>
  <DockPanel.DataContext>
    <Binding Source="{StaticResource myDataSource}"/>
  </DockPanel.DataContext>
  <Button Background="{Binding Path=ColorName}"
          Width="150" Height="30">I am bound to be RED!</Button>
</DockPanel>
```

The problem of auto-binding is its superficial beauty. Many methodologies have been invented for auto-binding, from simple and naive DataSource property to complex xml-based ones. It is quite ironic that beginners are taught to use auto-binding with UI controls, but seasoned programmers prefer hand coding.

The problem is easy: get property x from object o and show it to textbox y. The solution, unfortunately, is not that simple. With each new auto-binding method, programmers are required to re-learn how to do the same thing in a different way. They also must understand the little limitations such as one-way binding versus two-way binding, single versus multi-level of containment, etc.

Testing

Security tests and UI tests consume considerable amount of efforts. They are slow and they can't be skipped. For each and every input on every form, at least one testcase is required to ensure that users cannot accidentally crash or deliberately hack the software, or simply to check that the form works as intended. The more inputs there are on a form, the more effort is required for testing.

Using FormSoul

FormSoul is a component designed for binding and validating existing forms with minimal amount of coding and testing effort. Given the above example, FormSoul helps achieve the same results with just a couple of codelines.

Automatic Binding

Code 2.1: Retrieve data from input controls

```
        FormSoul soul = new FormSoul(); //FormSoul component

        /// <summary>
        /// Gets input parameters from form
        /// </summary>
        Product GetInputs()
        {
            Product p = new Product();
            p = soul.Get(p) as Product;
            return result;
        }
```

Code 2.2: Show existing data to form

```
        /// <summary>
        /// Show a product to form
        /// </summary>
        void ShowInfo(Product p)
        {
            soul.Set(p);
        }
```

The above codes are not over-simplified for this example, they are exactly the actual codes needed when using FormSoul. It doesn't matter how many properties an object or how many fields a form has, binding data to a form and getting them back require just a single line of Get, Set.

Decorative Validation

Looking back at *Code 2.2*, the line

```
            p = soul.Get(p) as Product;
```

will return a Product instance if all validations are passed. Otherwise, the FormSoul will return a null object.

To specify the validation rules that need to be checked by FormSoul, we can decorate rules directly on the Product class as followed:

Code 2.3: Decorate validation rules

```
public class Product
{
    [Required]
    public string Name;

    [LessThan(1000)]
    [GreaterThan(0)]
    public int OnHand;

    ...
}
```

The above codes specify that a Product is valid only when its Name is not null or empty, and its OnHand must be between 0 and 1000.

For complex validation logics, the programmer can write custom validation rules and register them to FormSoul easily.

Same code, different paradigms

Using FormSoul for Web Application
For web applications, nothing more is required. The above codes can be used for both Windows Form and Web Form applications. This feature greatly reduces the learning curve for any developer. For projects that require both Windows Form client and Web UI, this is even better. Programmers only need to copy and paste the same codes, nothing is changed, no configurations are needed.

Using FormSoul for Different UI Kits
The plain-vanilla UI controls shipped with .Net framework are usually not enough for developing real-world applications. As such, developers frequently have to work with third-party controls or write their own. New UI control library means new learning curves, new codes to do old tasks. Furthermore, switching UI kits in the middle of a development cycle usually brings along headache from changing and testing codes. Using FormSoul saves developers from that problem. FormSoul completely abstracts away the impedance mismatch between libraries. For any new control, developers need to write a simple adapter class to register the class to FormSoul. This is needed only once. The adapter class, on average, requires about 4-10 lines of code.

Emitted Form

For simple forms such as those found in web applications, FormSoul can be used to directly build the form.by dropping a FormSoul component on a blank asp.net page, then put the following code to the page's Load event:

Code 2.4: Auto-generate Asp.Net form

```
protected void Page_Load(object sender, EventArgs e){
{
    soul.ControlContainer = this;
    soul.Build(typeof(Product));
}
```

FormSoul will automatically emit all the necessary input controls.

How It Works?

FormSoul does it magic by knowing nothing about the actual controls it have to work with. In architectural jargons, FormSoul is designed with the principle of "separation of intent" in mind.

Assuming we are not bound to any programming language and free of any specific UI control, how would we work with a paper-based application form?

Typically, a form filler (writer) will do the following tasks:

- Write data to form, item by item
- Double-check that the item are correctly filled

An application approver (form reader) will:

- Read data from form, item by item
- Ensure that data are valid

Now assuming we have the same application form, but printed on color paper, or even imprinted on a sheet of metal, will we do differently to fill out the form? Of course not, we still do the basic steps above. As human, our brain allows us to quickly analyze and validate inputs. When it comes to programming, we are clouded by codes and forced to do the same task in different ways.

As the results of many researches in productivity, FormSoul decomposes form activities down to basic tasks and abstract away the differences of programming languages, of different UI controls. For current version, FormSoul has this architecture:

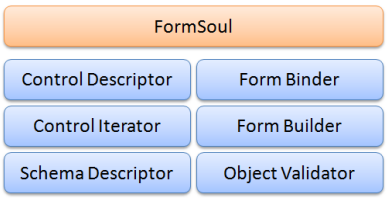

Figure 2: Architecture of FormSoul

- Control Descriptor: describes how to work with a particular control
- Control Iterator: allows working with controls nested in control
- Schema Descriptor: describes how to work with a specific business object
- Form Binder: binds data between business object and UI form
- Form Builder: generates UI form, manages states of UI
- Object Validator: validates business object

Using this architecture, FormSoul is free from working with any specific UI control. It simply concentrates on what it does best: reading, writing, and validating data.

Case Study: Productivity for Numerous Applications

FPT Software and Petronas project

Petronas [4] is an oil & gas corporation and is ranked among Fortune Global 500's largest corporation in the world. In 2006, via its subsidiary iPerintis [5], Petronas selected FPT Software [6] (one of the largest outsourcing IT corporations in Vietnam) to develop a multi-million-dollar project.

The project was to convert roughly 1,300 applications from Lotus Notes to Microsoft technologies. The user base was estimated at about 20,000 users worldwide.

FormSoul as a framework component

Most applications in Petronas project were estimated to take about 3-7 man-month to develop. Bigger ones were about 12-24 man-months. The problem with developing so many applications was a methodology to consistently build high quality products on-time. One of the core requirements was to develop a framework that would be used for developing all applications to ensure high quality and productivity.

THN was one of the consultants of this project and contributed to the core framework architect panel. From our analysis, we came to the conclusion that UI building and testing were one of the productivity bottle necks. As the results, we suggested to apply FormSoul along with other technologies. These suggestions helped reduce the development time from months down to weeks.

Final Thoughts

An application is successful only when users find it essential to their jobs. The costs to build such application keeps rising due to increasingly more complex business requirements, rapid change of technology, and due to the fact that human is slow to learn and to change his mind set.

FormSoul is designed to embrace the productivity of developers while coding for the presentation layer. Using FormSoul, developers are free from doing repetitive tasks. Without the encumbrance of "wiring-codes", routine tests, and security vulnerabilities, developers can enjoy again what human does best: to think.

References

[1] **The Open Group,** *Agent Technologies Discussion Paper,* October 2006
http://www.opengroup.org/

[2] **The Standish Group Report**, *Chaos*, 1995

[3] **Code Generators**
CodeSmith http://www.codesmithtools.com/
IronSpeed Designer http://www.codesmithtools.com/
JavaGen http://www.javagen.com/
.netTiers http://www.nettiers.com/

[4] **Petronas Corporation**
http://www.petronas.com.my/

[5] **iPerintis**
http://www.iperintis.com/

[6] **FPT Software**
http://www.fpt-soft.com/