HMI automated code generation

Mark Grant, Ken Merrick

HMI automated code generation

A. Mark Grant, B. Ken Merrick

1: SiemensVDO, France

Abstract: In today's automotive infotainment systems a lot of time and effort (= money) is invested in developing the HMI. One of the problems encountered when developing HMI is the necessity to change the spec late on in the development phase which can result in SOP delays. As a result, there is a continuing need to reduce the development time for the HMI. In addition, the OEM's invest in providing their unique branding and have it available across the complete product range offered. Two techniques are invoking interest in this field:

- Cross-platform HMI frameworks
- Multi-platform code generation

This paper highlights worked carried out on the second technique – multi-platform code generation.

1. History of the project

In the beginning of 2000 a development group was created to provide a 'next generation' infotainment system based on Java technology. One of the goals was to provide a flexible HMI framework with support for multi-modality and ease of customisation. It was decided to define the internal format of the HMI specification in XML. Due to budget constraints a design tool that would allow the user to specify the HMI graphically was postponed and so the developers had to 'code' the XML by hand. Once the product was shipped it was decided to provide the design tool and so a few years later a WYSIWYG tool was developed to allow the HMI designer to build the HMI. An additional feature of this tool was to plug in code generators for the different hardware/software target systems.

2. HMI specification as XML

The industry trend for HMI (layout) specification is to use XML as the meta-language, with the different players providing their own schema. Examples of this are Microsoft (WPF), Adobe (Flex) and the latest being Google (Android).
The HMI framework developed for the above system extended the use of XML to include the definition of the overall HMI screen hierarchy and transitions as well as per screen event handling.

As XML is a well formed text based language it lends itself to automated 'data-handling' such as runtime parsing, or in our case multi-target code generation.

The KISS Principle

The KISS principle - Keep It Short and Simple – means make the simple things simple and the complex things possible.
The KISS Principle is sometimes cited on a development project to fend off feature creep - a tendency for product or project requirements to increase during development beyond those originally foreseen, leading to features that weren't originally planned and resulting in risk to product quality or schedule.
This is a very common design mistake. Users ask for lots of features and developers work hard at delivering those features, but often at the expense of a simple to use application.

By simple, we mean "everything should be made as simple as possible, but no simpler" (quote attributed to Albert Einstein). If the interface doesn't make the most common way of performing a task as simple as possible, the interface has failed the user.

Specifying HMI Layout

HMI consists in general, of various graphical objects positioned on the screen in such a way as to convey information to the user in a structured fashion. In most cases this is in the form of Frames, with each frame containing visual elements (widgets). More advanced HMI can provide a better visual appearance using transparent objects, animation and 3D, but generally these can be abstracted to graphical objects laid out on the screen.

If we Take a 1000 feet view of the above we can see that the HMI layout can be defined as having at any one time: one screen consisting of one or more frames, each frame containing one or more widgets. Each widget has several properties that helps describe the visual appearance (colour, transparency, text, icon...).

Specifying Event handling/Data binding

An event describes, in sufficient detail, a particular user action. Rather than the program actively collecting user-generated events, the program is notified when an interesting event occurs. Programs that handle user interaction in this fashion are said to be event driven.

Data binding is the process of tying the data in one object to another object. It provides a convenient way to pass data around in an application. Data binding requires a source property, a destination property, and a triggering event that indicates when to copy the data from the source to the destination.

Specifying Screen hierarchy

An applications HMI usually consists of several screens with each set of screens exposing a set of features. As the use selects a feature from one screen, the next screen can propose one or more additional options which are a subset of the current feature. Viewed in this manner we can see that the HMI is composed of a hierarchy of screens and the user navigates within this hierarchy.

Specifying transitions

Navigating from one screen to another is accomplished using event triggers. The result of an event trigger is referred to as a screen transition and can be as simple as displaying one screen after another to providing various animation effects such as fade-in/fade-out. These transitions help provide a workflow when navigating through the HMI and are well defined and consistent throughout the life of the HMI.

A special case of transitions which breaks the above workflow is the popup transition. This transition interrupts the normal flow and can result in a new level of navigation (navigating through several screens within the popup context). Once the popup context is terminated the system must return to the original workflow. Note that a system can have several layers of popup and so a context stack must be provided.

3. Why code generation?

Code generation is the automated building of high-level code from a description of the requirements of the code.

Code generators are split into two major types, passive and active. Passive generation builds a set of code once and does not maintain the code over the long term. Engineers are encouraged to alter the output of a passive generator to meet their requirements. Active generators build code and maintain it through further generation cycles.

Code generation provides several advantages over its hand-coded equivalent.

Quality

The quality of generated source code is directly related to the quality of the templates. This is a big advantage because as you increase the quality of the templates over time you will also increase the quality of the entire code base. You can start out with a relatively lightweight pass at the templates and then make them more robust in an iterative process as your gain understanding of the framework and proper error handling. Contrast this process with hand-written code, which will have unreliable quality over time as interest in the project ebbs and flows.

There's another quality advantage, bug fixing. Fixing 100 hand-coded classes one by one is a lot harder than fixing one template and generating the 100 classes again. Systemic bugs are much more likely to get fixed in generated code bases. In addition, optional enhancements to increase the stability or functionality of the code base are much more likely to happen. For all of these reasons, a generated code base is much more agile and robust than its hand-coded equivalent.

Consistency

Generated code bases are extremely consistent in interface definition. This makes it easier to hand-write code on top of generated APIs. It also makes it easier to build other layers of generated code on top of them.

The easiest and most obvious way to get maintenance changes incorporated is through code generation. The fundamental here is that maintenance programmers can not adequately repeat complex processes, and that instructions become out of date.

Once code is hand-written by an engineer they turn thrit attention to something else and the old code starts to atrophy until someone gives it some more attention. Generated code is continuously maintained by the generator. Every generation cycle replaces the entire generated code base with fresh code. When bugs are fixed in the templates they are propagated across all of the code consistently. So the output code is always better maintained than its hand-coded equivalent.

Productivity

The productivity of the engineering group is unquestionably enhanced by code generation. The more important productivity increases are tied to the morale improvement you'll see in the engineers working on the project. Code generation applications meet or exceed manually created ones in every aspect. That includes performance, security, source control, and so on.

Code generation using XSLT

XSLT is a template language designed to create any type of text, XML, or Hypertext Markup Language (HTML) output from XML input. Because source code is text output, XSLT can create source code.
The XSLT language presents a new way of thinking because it's explicitly designed from the ground up to apply patterns to information in order to transform information into text. Because metadata is information and source code is text, XSLT does exactly the job presented by code generation.
Generally, XSLT processors produce one output file, although an external utility could break the single output file into multiple files. For example, in the case of source code the output file can delimit the beginning and end of each file with markers not likely to occur as part of the source code.
Furthermore, the location of each file could be indicated as part of the generation.
The external utility then processes the single output file to produce multiple files in multiple locations. Similarly, input to the XSLT processor should preferably be one XML file including other XML files.

Code generation using .Net

The .NET framework is a programming model for developing, deploying, and running XML Web services that targets all types of applications. The .NET framework enables building open applications. Every .NET-aware language is compiled to intermediate language code which is executed in Common Language Runtime (CLR). Custom attributes are one of the most innovative features of the .NET framework. They allow everyone to define information which can be applied to classes, methods, parameters, and so on.

4. Mapping to the target platform

The basic steps to carry out when specifying the code generation are:

- Identify the Layout elements in the target HMI framework
- Identify the widget set and properties
- Identify the event handling mechanism
- Identify the HMI menu structure and display management.
- Identify the screen transition mechanism

Once specified the next step is the task of mapping the source HMI schema to the match the final target framework(s). The mapping steps are:

- Provide template code blocks to represent the creation and layout of the visual elements
- Provide template code blocks to set/get the specific widget properties, including type conversion
- Implement scripts to output the relevant code blocks when parsing the HMI layout to add the visual elements and set the element properties
- Provide template code blocks to map input events to HMI actions (update the screen, send requests to the application...)
- Provide template code blocks to bind application data to HMI elements
- Implement scripts to output the relevant code blocks when parsing the HMI event definitions to handle the input events
- Provide code blocks to interact with the target display management framework
- Implement scripts to output the relevant code blocks when parsing the HMI menu structure and transition definitions

The above steps cover the majority of cases encountered up until now (true 3D environments excluded).

As an example, let's assume we are targeting a simple Flash framework (Flex based) with the following properties:

- One Frame per screen
- The widget toolkit consists of Text and Button elements (so Lists must be built using Buttons)
- Business interaction is achieved via Action Script classes
- Screen navigation is performed by setting the current child of a ViewStack (Flex specific main panel)

The following conversion steps are defined:

- Create a Canvas element for each Frame defined
- Add the relevant element for each widget encountered (including a recursive generator to create multiple Buttons for List widgets)
- Convert the widget properties to the corresponding XML attributes
- Add a function for every event defined per frame which will include the calls to the business logic
- Add the function actions defined for the event handler (including transitions and screen updates)
- Associate a click attribute to the widget defined as triggering the event (name matching)
- Define a function to trigger the screens transitions

With the above defined it is a simple case of adding the necessary 'plumbing' to call the various templates at the correct point in the processing of the XML files. An example sequence is:

- Create the header section of the Flex file
- Read in the transitions definitions and call the template to inject the transition functions
- For each frame:
    - Read in the event definitions and call the template to inject the event handler functions (and add the business calls)
- Create the ViewStack element
- Read in the list of frames and for each frame: add a Canvas element
    - Read the list of widgets for each frame and call the template to inject the element (and set the attributes) for each widget
    - Close the Canvas
- Close the ViewStack
- Append the footer section of the Flex file

The resultant file can then be passed on to the Flex compiler to produce a Flash file which can be loaded and tested in a web page.

4. Testing

The code generation, based on the common XML schema, has so far been tested on the following graphical environments and platforms:

- SDL (C++) on Linux and Windows XP
- Win32 (C++) on Windows XP and Windows CE
- .Net (C#) on Windows XP and Windows CE
- Java on Windows XP (AWT and SWT) and VxWorks
- Flash (Flex+ActionScript) on Windows XP and Linux
- Yahoo Widgets on Windows XP
- Silverlight (XAML) on Windows XP

For each of the above the target widget toolkit had to be developed or an external toolkit used, although a prototype of widget definition using XML is currently being developed.

5. Conclusion

Using a simple XML specification to define the HMI, we can easily use existing technology to generate HMI for varying platforms and frameworks. With the addition of a WYSIWYG tool this moves the development of the HMI back to the HMI designer and frees up the application developers to do what they are best at – coding, not HMI.

6. Acknowledgement

I would like to thank Ken Merrick who works within our group for his expert help in analysing the market trends and sanity-checking my work and our various team members who contributed to the development of the HMI frameworks and tooling used throughout our systems.

7. References

8. Glossary

HMI: Human Machine Interface

XML: eXtensible Markup Language

WYSIWYG: What You See Is What You Get

WPF: Windows Presentation Foundation

SDL: Simple Media Layer is a set of portable libraries originally designed for game development that provide graphics/audio capabilities.

Android: Android is a software stack for mobile devices that includes an operating system, middleware and key applications.

Flex: Adobe® Flex™ is a cross-platform development framework for creating rich Internet applications (RIAs).

.Net: .NET is the Microsoft Web services strategy to connect information, people, systems, and devices through software. Integrated across the Microsoft platform, .NET technology provides the ability to quickly build, deploy, manage, and use connected, security-enhanced solutions with Web services.