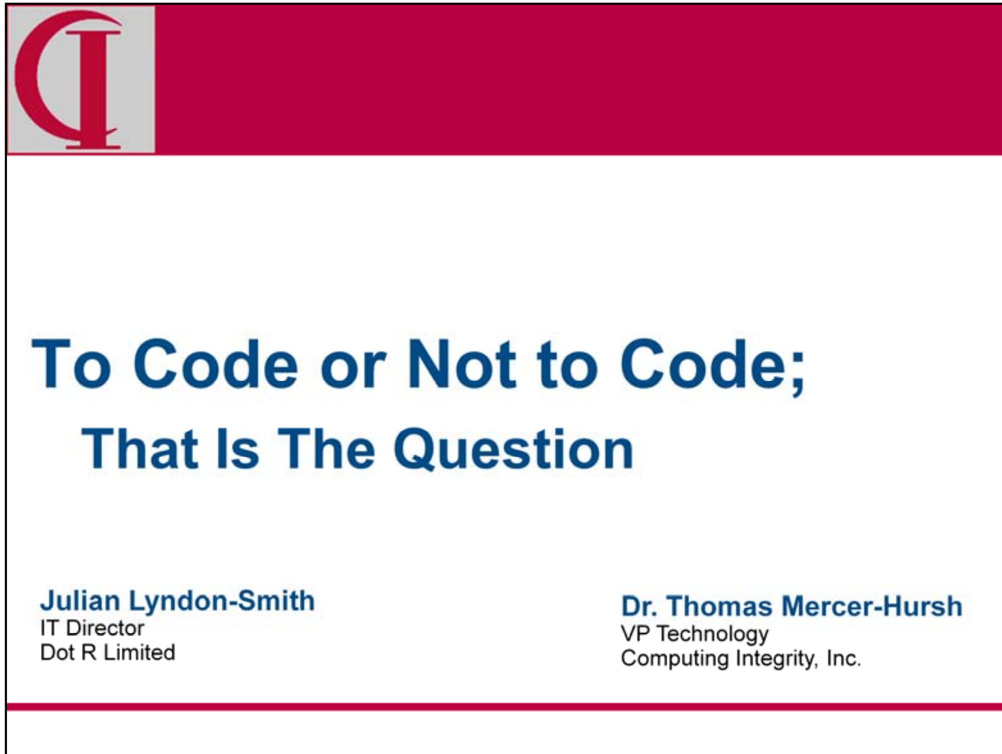


Let us begin by introducing ourselves.

This talk arose out of exchanges between us on the PSDN forums and in private communications related to Julian's exploration of simple techniques to avoid writing routine code, particularly code which was evolving and thus would potentially need the same change to all existing code modules of the same type. Thomas shared his perspective based on both his current work on model-to-code generation and on his historical work with a substantial ABL code generation effort. We are here today to share our … sometimes differing … insights with you so that you can consider the role of code generation in your own shop.

I, Thomas, began working with Progress in 1984 and I have been a Progress Application Partner since 1986. For many years I was the architect and chief developer for our ERP application. In recent years, I have refocused on the problems of transforming and modernizing legacy ABL applications. I have been working with various code generation efforts since the early 1980s and produced a tool from which over 1 million lines of production ABL code have been created. More recently, I have been exploring model-to-code technologies for dramatically increasing productivity in creating and revising ABL applications.

Dot R and Me

- Started using Progress v3 1987
- Dot R created 1991
  - version control systems
    - VSS Open source plugins for Appbuilder
  - Wrote xcode decrypter
- Acquired by Tessera in 1996
- 2011 Independent company again

I, Julian …

Agenda

- Introduction – Why I Started Code Generation
- Quick & Dirty Code Generation for Standard Components
- Specification-Driven Development
- Model-to-Code Translation
- Summary

 © 2011 Computing Integrity

Here's our agenda for today. First we are going to talk briefly about what motivated each of us to explore code generation. Then, Julian will talk about his recent work with developing code generation for some basic components in his application. Then Thomas will talk about his historical tool, Specification-Driven Development and what it was possible to accomplish with that. Then Thomas will talk about his current efforts in Model-to-Code translation. Finally, we will sum up and send you off to make your own explorations.

Agenda

- Introduction – Why I Started Code Generation
- Quick & Dirty Code Generation for Standard Components
- Specification-Driven Development
- Model-to-Code Translation
- Summary

© 2011 Computing Integrity

First, let's each talk a bit about why each of us got interested in code generation and what role we see it playing in overall development.

Why Code generation ?

- Had started writing code in a new way
  - Using objects as parameters wherever possible
  - Using unit tests as basis for development
  - Wanted to split the model from the database
  - Wanted to do it "right"
  - Not have the "same" code, but with different table names
  - Not have lots and lots of similar programs / classes etc

Why Code generation ?

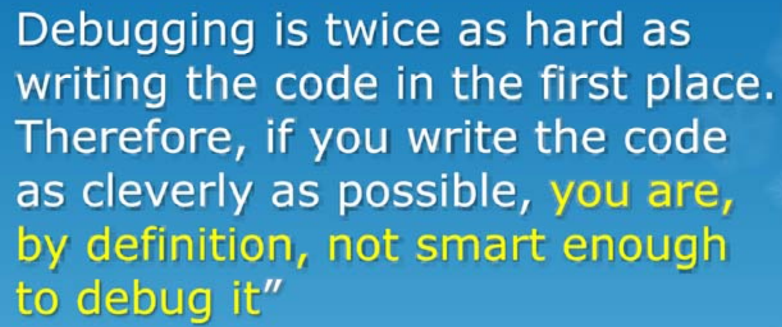
- Wanted the code to be modern
  - Object based
  - No include files
  - Just add a new table, and it will just work
- Dynamic code to the rescue !
- Lazy instantiation to the rescue !

Why Code generation ?

- Hey ! I Got it "right" ☺
  - A single base class
  - Handles all crud operations for any model
  - Inheriting class just contained table name
- Took a weekend of hard work
- Sat back and congratulated myself on how clever I was
  - Several beers in celebration

Things fall apart

- On the Monday, wanted to make a change
  - Who wrote that ** code ?. It's horrible.
  - DYNAMIC-INVOKES
  - DYNAMIC everything else
  - Not a pleasant read. Rated XXX
  - Several beers to drown the sorrows
- Never mind. Document it. That will fix it.
- Then, a well-known saying came into my mind


Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it”
Brian Kernighan

On Reflection

- Why was I so hell-bent on doing it this way ?
  - Trying to save memory
  - Trying to make it as fast as possible
  - Trying to show off how smart (dumb) I was
- Did some analysis
  - "old fashioned" way used 2x the memory, but was 30% faster …

Enter code generation

- So, what I needed was lots of code that was basically the same
- What I didn't want was to write that code
  - I am, by nature, an inherently lazy coder ;)
- Get the computer to write it for me
  - If code needs fixing, fix the template, regenerate

Enter code generation

- "robust" discussion on PSDN
  - http://communities.progress.com/pcom/message/123716
- Had to overcome some severe issues
  - Custom code
  - Different models
  - Dr Mercer-Hursh demanding^h^h^h^encouraging me to do things "the right way"

Enter code generation

- Needed to be in the ABL
- Simple to use
- Simple to fix
- Simple to enhance
- Provide for majority of code

Presenting #Code

- Templates define which classes to build
- Properties automatically created from DB
- Extra properties can be defined
- Point, click, generate
  - Model, Crud, Fixtures, data source and test units
  - Lookups, UI generation
  - Extendable for futures

Why I Started Code Generation

- Origins in 1st 4GL
- Enhancement in Oasis Toolkit
- One-Time Generator
- Rationale behind Specification-Driven Development (SDD)
- Current shift to Model-to-Code Translation

 © 2011 Computing Integrity

Thomas: For me, the rationale for code generation is closely tied to the motivation for working in a 4GL, i.e., to spend less time in the mechanics of writing code with less susceptibility for errors from hard to find details, and thus to be more productive at creating and changing code. I created my first 4GL in 1979. It compiled into AlphaBASIC, the standard language on the AlphaMicro where I was developing a new module. It was intended just to allow a simple, short syntax to compile reliably into a larger number of predictable commands for doing basic, repeated tasks. In the early 80s I developed a code generation tool called the Oasis Toolkit which was more of a true program generator to create all the predictable structure for routine parts of new applications.

In the late 80s I used a generator which came with the ABL framework I was using at the time, but it was not very satisfactory since it was a one time generator, i.e., one fed in some parameters and out came a code skeleton which one would then edit to produce the working code. That saved a little work, but, if one changed one's mind about what the basic pattern should be, one was stuck with manual modification of all programs created thus far. This got me to thinking about code generation which was regenerable, ie.., where one could change the underlying pattern and simply turn it on and regenerate all code created thus far, preserving all the modifications. To me, this is a night and day difference in code generation. Without it, one merely saves a little time. With it, one has a system which one can evolve.

This lead to the development of Specification-Driven Development (SDD) in 1990 and the years immediately following. I'm going to talk about SDD more in a little while, but let me just say that this was a system with a couple of key design goals. One was that it be regenerable. Another is that it be "no compromise", i.e., one should never compromise what one wanted the program to do in order to make it work under the generator. We had a good shop consciousness in which programmers would notice when they needed something multiple times as custom code and we would then add that to the capabilities of the generator as a standard feature so that it rapidly covered more and more capabilities just by selecting specification options. In the end, we created over a million lines of code with it and I would probably still be using it were I still maintaining this application. Its big limitation relative to modern needs is that it needs a major overhaul to conform to modern architectural standards.

In recent years I have been looking at Model-to-Code translation, called MDA or Model-Driven Architecture in UML terminology. To me, this is the next generation beyond 4GL because it means being able to work in the model and not write code manually at all. This may sound unlikely, but I will explain later why it is less surprising than it might seem.

Why I Started Code Generation

Evolving Theme:

- Let the computer write the routine parts.
- Drive the process by specifications or characteristics independent of the implementation of each feature.
- Abstract the specification from the architecture.
- Work as much as possible with the specification, not the result.

© 2011 Computing Integrity

In this progression, there has been an evolving theme.

First, let the computer do the boring, routine parts. It is good at boring, routine things and doing them myself means introducing errors and dulling my mind for solving the hard problems.

Second, make whatever mechanism is used to select among alternatives and features independent of the implementation of that feature or characteristic. Leave room to change the implementation without having to change the specification.

Third, make the specification as a whole abstracted from the implemented architecture. Make the specifications about the problem space, not the computing space. Next year I might change my mind about how to structure the solution.

Finally, work toward a system in which I can work as much as possible with the specification, not the code. Just like a 4GL abstracts further from the computing details than a 3GL, I want to work to abstract as far from the implementation as possible.

Agenda

- Introduction – Why I Started Code Generation
- Quick & Dirty Code Generation for Standard Components
- Specification-Driven Development
- Model-to-Code Translation
- Summary

© 2011 Computing Integrity

Let's start with Julian's current work.

#Code demos

- Defining a configuration
- Using Templates to generate the code
- Entering custom data
- Running the generated code
- Customise UI layout
- Code walkthroughs

Agenda

- Introduction – Why I Started Code Generation
- Quick & Dirty Code Generation for Standard Components
- Specification-Driven Development
- Model-to-Code Translation
- Summary

 © 2011 Computing Integrity

And now a bit about Thomas’ historical SDD tool.

Specification-Driven Development

- Evolved from dissatisfaction with one time generator in framework.
- Early decision not to use ABL because of string processing speed.
- Selection of m4 macro processor – because it was there.

 © 2011 Computing Integrity

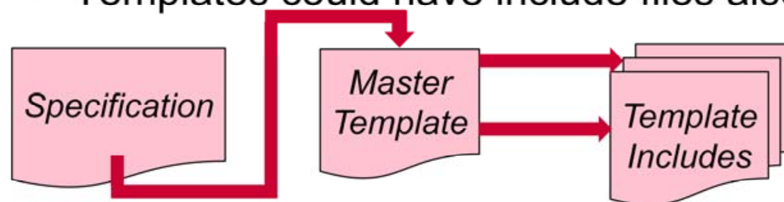
As I said previously, in the late 80s we made some use of a one-time generator in the framework of the product we were selling, but were very dissatisfied with it because it had almost no options and the one-time generation meant we would be doing lots of edits in the future as our architectural model changed. Frankly, it wasn't a whole lot better than just copying an existing program and doing substitutions. Based on the experience of the early 80s, I really wanted a regenerable tool so that existing code could continue to evolve with our ideas about architecture and I wanted something that got me much closer to a final program with less manual coding.

I really wanted to write something in the ABL, but our experience at that point was that ABL was not very fast when it came to string handling and while it might be tolerable for generating a single program, I saw us regenerating large numbers of programs where performance would matter. Poking around for options, I found m4 which then and now comes with every Unix system and was even used in a few Unix configuration tools.

m4 is a macro processor, i.e., it copies a string of text from the input, makes substitutions, and produces an output stream of text. It includes a number of simple and yet powerful logical and pattern facilities.

Specification-Driven Development

- m4 dictated the form of the specification and template files.
- “Specification” file was a list of definitions and the include of a master template.
- Templates included substitutions and decisions.
- Templates could have include files also.



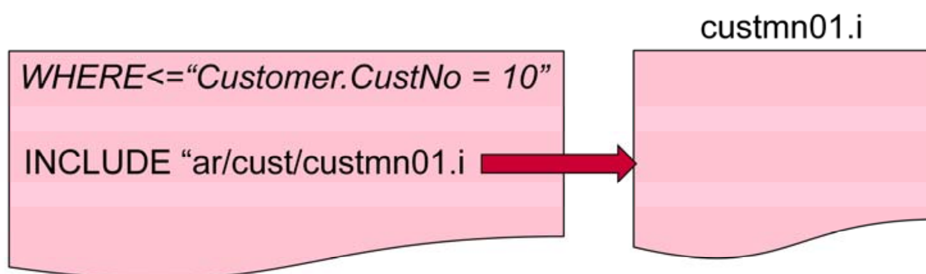
 © 2011 Computing Integrity

What quickly evolved worked as follows:

For every target program unit, we would create a “specification” file. A specification file was a list of definitions which finished with an include of the master template for the desired function. m4 would read the specification file and then start reading the template. While reading the template, it would make substitutions based on the definitions in the specification. Some of the definitions in the specification were definitions of options, so m4 would test whether or not an option was defined or what value it had and pick different code to include or different structures based on that choice. The template itself could have includes so that one could create a pretty clean structure isolating one function per file for easy maintenance. An option might determine whether or not an include was used or which of several includes to use.

Specification-Driven Development

- Custom code could be included by “hooks”.
- Code could be in-line for simple things like a WHERE clause or could reference an include file.

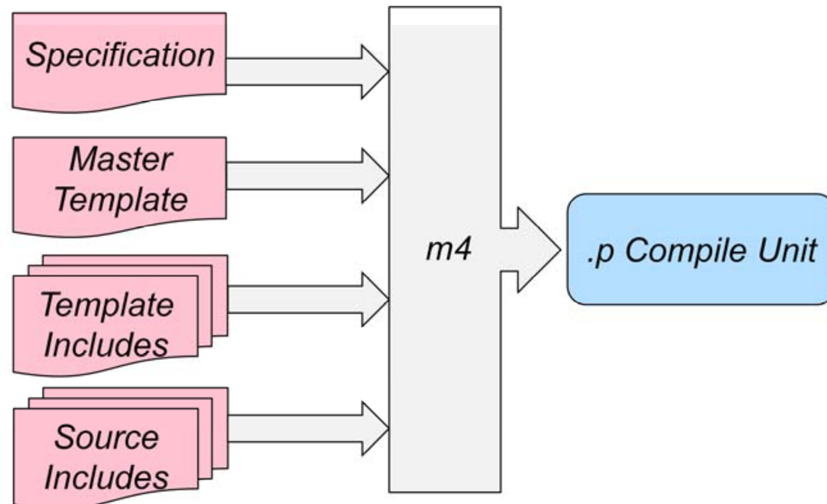


22 To Code Or Not To Code © 2011 Computing Integrity

We could include custom code in the generated program by the use of “hooks”. Anywhere in the template that we thought we might want to include custom code, we would include a “hook”, i.e., a macro name which would be substituted at that point if a definition existed. By simply defining a value for that macro name, the code provided with the definition would be included in the generated program. For short code fragments like WHERE clauses, we would just put the code right in the definition. For longer code blocks we found it more readable if we put the code in an include file and made the definition a reference to that file. For the old-timers in the audience, this was similar to the practice in FastTrack of putting the form definition in a “.f” file and then including it in the program. That was almost the only idea we took from Fastrack….

Specification-Driven Development

The result was new compile unit, i.e., a dot P.



23 To Code Or Not To Code © 2011 Computing Integrity

The definitions, master template, any include templates, and the source code includes pointed to by the specifications are then combined by m4 into a single compile unit, i.e., a .p.

Specification-Driven Development

- Also built shell scripts to generate the basic specification files for any given function type.
- Shell script provided a dialog with basic questions (e.g. file / field names) and primary desired options, then output specification files which were edited for details.

© 2011 Computing Integrity

We also built some shell scripts to generate the basic specification files for any given function type. Eventually we had about 6 basic program types which were covered by this system.

The shell script provided a dialog with some basic questions like file and field names and some primary desired options and then would output the specification files which were then edited for details. The shell script was not a regenerable generator — it only covered the most basic issues.

Specification-Driven Development

- Over a couple of years, we evolved the templates significantly.
- Some evolution was new functionality or new implementation.
- Much was recognizing repeated patterns in the custom code, then providing new options to generate that code (no longer custom).

 © 2011 Computing Integrity

We used the SDD tool fairly extensively for several years. As ABL and our ideas about architecture evolved, we would rework the implementation in the templates and then regenerate all the programs which used that template, typically providing the change in all impacted programs in minutes. When the change was something selective, not universal, we had to edit the appropriate specification files to use the new feature, but this was often just adding a single line.

We also had a good culture in the group in which programmers would recognize that the function they were about to provide with custom code was something that would be used many times. They would then bring this to me and I would add the capability to the template so that functionality could be provided by a simple definition instead of custom code in each case. Thus, as time went on, less and less custom code was required to produce finished programs.

Specification-Driven Development

How well did this work?

- Over 1 million lines of ABL created.
- In one project, over 300,000 lines of ABL in 300 programmer hours.
- Resulting programs were more feature rich than one would typically write by hand.
- Code was extremely stable and uniform leading to very low support costs.
- It was extremely fast to add new features and functions to existing code.

 © 2011 Computing Integrity

How well did this work?

In the end, we created over 1 million lines of ABL. Probably less than 3% of that was custom code.

We did one project creating two new modules which totaled something over 300,000 lines of ABL code. We went from my having built the schema and written brief descriptions of the issues in each function to code ready for integrated testing in just about 300 programmer hours … yes, 1000 lines of ABL per programmer per hour. Integrated testing was also very fast because the code was so stable.

Resulting programs were more feature rich than one would typically write by hand, at least for the time. Expectations are higher today so one might be forced to include more in even basic code, but using this approach allowed “the best we could do” to be the standard for everything we did.

The code was extremely stable and uniform leading to very low support costs. Users could count on functions behaving in consistent and uniform ways so they had few questions. The code just worked. And when there was a problem, it was very easy to diagnose because 99 times out of 100 the issue would be in the hand-written code, not the generated part.

It was extremely fast and inexpensive to add new features and functions to existing code. This meant very low costs to our customers for new features and functionality. ROI decisions were routinely rapid because the cost was so low. How well did this work?

In the end, over 1 million lines of ABL created.

In one project, over 300,000 lines of ABL in 300 programmer hours.

Resulting programs were more feature rich than one would typically write by hand.

Code was extremely stable and uniform leading to very low support costs.

Specification-Driven Development

Some sample specifications – Customer Maintenance

- Before persistent procedure structure!
- Control program ar/cust/custmn.p.
- Key solicitation ar/cust/custmn11.p.
- Find, create, audit ar/cust/custmn20.p
- Multiple tabs of data, e.g. custmn31.p
- Control program calls key solicitation programs in sequence and then presents a strip menu with multiple frames.

 © 2011 Computing Integrity

So, let's take a look at some sample specifications, in this case customer maintenance. Note, this architecture dates from before persistent procedures so it is a long way from what one would do today. This is originally a V6 architecture. The menu system runs a main control program which does no real work except to run other programs. It runs one or more key solicitation programs in sequence, each of which can display validation and provide cursor pointing, inquiry, etc. When a unique record has been identified, then the control program runs a small program which obtains an existing record or creates a new one and provides the before image for the audit trail. It may also run a program to display additional information in the header frame where the keys were solicited. It then runs one or more programs which display current values and presents a strip menu which allows one to navigate among the tabs of data. This supports other table maintenance programs for sibling tables and parent child relationships for things like bills of materials.

Specification-Driven Development

```
# ar/cust/custmn.p_m   Macro definitions for custmn.p
#
__DEFINE(`_PGMNAME',`custmn.p')
__DEFINE(`_APPLCODE',`ar')
__DEFINE(`_SUBDIR',`cust')
__DEFINE(`_FILE',`customer')
__DEFINE(`_FILETITLE',`CUSTOMER')
__DEFINE(`_MANDA',4)
__DEFINE(`_MANDMSG1',`General or Common')
__DEFINE(`_MANDMSG2',`A/R or Common')
__DEFINE(`_MANDMSG3',`O/P or Common')
__DEFINE(`_MANDMSG4',`Addr or Common')
…
```

28 To Code Or Not To Code © 2011 Computing Integrity

Here's part of the specifications for the control program. This shows you the style of the definitions. The MAND options relate to a requirement that certain tabs must be visited to fully populate a new record. This can happen in any order.

Specification-Driven Development

```
__DEFINE(`_MAINCNT',`12')
__DEFINE(`_MAINOPT1',`ar/cust/custmn11.p')
__DEFINE(`_MAINOPT2',`ar/cust/custmn20.p')
__DEFINE(`_MAINOPT3',`ar/cust/custmn21.p')
__DEFINE(`_MAINOPT4',`ar/cust/custmn31.p')
…
__DEFINE(`_MAINOPT10',`ar/csgr/mbgpmn.p')
…
__DEFINE(`_STRPOPT1',`Name')
__DEFINE(`_STRPEXP1',`Update Name')
__DEFINE(`_STRPOPT2',`Common') …
…
```

29 To Code Or Not To Code © 2011 Computing Integrity

Here you see the definitions for the key, create, and some of the tab programs, including one which is a maintenance program for a separate sibling table which allows putting the customer in one or more customer groups. In that case the "header" frame is actually just above the strip menu and there is a frame above that which shows existing memberships. This illustrates the flexibility of the templates covering very different layout requirements.

The bottom part is the list of labels for the strip menu and a long description which shows when that menu item is current.

Specification-Driven Development

```
# ar/cust/custmn11.p_m  Macro definitions for custmn11.p
…
# Key actions available
__DEFINE(`_HASSORT',`Yes')
__DEFINE(`_HASCOPY',`Yes')
__DEFINE(`_HASBRWS',`Yes')
__DEFINE(`_HASEXIT',`Yes')
…
# Specifications for file for cursor pointing
__DEFINE(`_SPECVALD',`{ar/cust/custmn02.i}')
__DEFINE(`_FLDVALD',`Name')
__DEFINE(`_SORTIDX',`2')
#
__INCLUDE(__HOME/m/mnt/mnt1.m)
```

 © 2011 Computing Integrity

This is the specification for the key solicition program, in this case the customer number. Note the selection of options in the key option, each of which triggers a whole set of code to be available in the resulting program.

The _SPECVALD definition illustrates including a small include file with custom code.

At the bottom you see the include of the template file.

Specification-Driven Development

```
# ar/cust/custmn31.p_m  Macro definitions for custmn31.p
…
__DEFINE(`_MAINFORM',`custmn2')
__DEFINE(`_ROW',4)
__DEFINE(`_COL',1)
__DEFINE(`_DOWN',1)
#
# Fields in frame
#
__DEFINE(`_MAINFLD01',`Sort-Name')
__DEFINE(`_MAINFLD02',`Company-Code')
__DEFINE(`_MAINFLD03',`Charge-Cust')
__DEFINE(`_MAINFLD04',`Stats-Cust')
__DEFINE(`_WORKFLD05',`ch_Address[1]')
```

 © 2011 Computing Integrity

Here is a quick look at the specifications for one of the main tabs, showing the form, position, and the list of fields in the frame. Note that the last one is a field not in the customer table since addresses are stored in a separate table in this application.

Specification-Driven Development

Overall:

- Less than 300 short lines of specification, much of that originally generated.
- Over 2000 lines of ABL.
- 11 programs in the set.
- Bunch of small includes for custom code, frame definitions, etc.

Ancient architecture, but highly functional and visually pleasing for ChUI and very easily enhanced and maintained.

 © 2011 Computing Integrity

So, overall, for this customer maintenance function we have:

- Less than 300 short lines of specification, much of that originally generated;
- Over 2000 lines of ABL;
- 11 programs in the set; and
- A bunch of small includes for custom code, frame definitions, etc.

It is an ancient architecture, but highly functional and visually pleasing for ChUI and very easily enhanced and maintained.

Specification-Driven Development

So, where is this tool today?

- Still works.
- Code is ChUI.
- The architecture in the templates is from 1990.
- The specification file is compact, but not very pretty.

A meaningful accomplishment for the time and one with a pretty modest development investment, but we can do better today…

 © 2011 Computing Integrity

So, where is this tool today?

Still works.

Code is ChUI.

The architecture in the templates is from 1990.

The specification file is compact, but not very pretty.

A meaningful accomplishment for the time and one with a pretty modest development investment, but we can do better.

Agenda

- Introduction – Why I Started Code Generation
- Quick & Dirty Code Generation for Standard Components
- Specification-Driven Development
- Model-to-Code Translation
- Summary

 © 2011 Computing Integrity

Having looked at ancient history, let’s look where I am headed today.

Model-to-Code Translation

- Modern development strategies highly diverse:
  - Some emphasize rapid coding and putting partial products in front of the user as fast as possible.
  - Others emphasize the importance of good analysis and design.
- But, everyone needs to deliver new applications quickly and respond nimbly to changing requirements.

 © 2011 Computing Integrity

Modern development strategies are highly diverse.

Some emphasize rapid coding and putting partial products in front of the user as fast as possible.

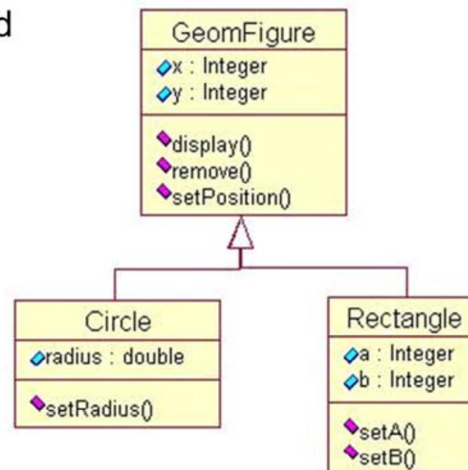
Others emphasize the importance of good analysis and design.

But, everyone needs to deliver new applications quickly and respond nimbly to changing requirements.

Model-to-Code Translation

UML (Unified Modeling Language)

- Predominant method used
- Used differently: as a sketch, a detailed analysis, or to generate code
- Created in mid-1990s to unify a diverse set of modeling languages



36 To Code Or Not To Code © 2011 Computing Integrity

Among those emphasizing analysis and design, the strongly predominant way of expressing that design is UML (Unified Modeling Language).

UML was created in the mid-1990s to unify a diverse set of modeling languages which had grown up, primarily for OO development. A standards body, the OMG or Object Management Group was created to oversee this and other standards and UML has undergone considerable expansion and development since the original version.

Different people use UML in different ways. Some use it simply as a sketching tool, something to put on a white board or in a document to facilitate discussion. Some will use it more completely to do a detailed analysis of a system and then write code from that design. They may or may not keep the design in sync with the code as the system evolves, although it is usually regrettable if they don't. And, there are those … which is what interests us today … who actually generate the working code directly from the model.

Model-to-Code Translation

- Generating code from the model is called MDA or Model-Driven Architecture. More clear to call Model-to-Code Translation.
- MDA very common in some industries (telecom and RTE), used more sporadically in other 3GL OO development.
- Forms of Model-to-Code becoming norm in some product types, e.g.
  - Savvion has process definition modeled in tool supplemented by a small amount of hand .
  - Apama has what to do provided by a tool.

37 To Code Or Not To Code © 2011 Computing Integrity

The term used by OMG for generating code from the model is MDA, Model-Driven Architecture. This term actually covers more than just generating code, which I will get to in a bit. Some of us tend to refer to the concept more as Model-to-Code Translation since historically this sort of going from one form to another form is called Translation.

Such model-based code generation is very common in some industries. Indeed, in order to compete there, you pretty much have to do it because your competition is and you can't be as productive without it. That itself is a powerful endorsement. It is particularly common in Real Time Environments like complex control systems. That it succeeds well there is further testimony since these are very time critical systems where nothing but the best code is good enough. Usage in other 3GL OO development is more sporadic, in part because the competition is more diffuse.

Some form of Model-to-Code is becoming the norm in some product types.

Savvion is an example where the process definition is done in a modeling tool and is supplemented by a small amount of hand coding and then executed.

Apama is another example where the definition of what to do is provided by a tool which is more model-like than it is regular programming.

Model-to-Code Translation

- Two different approaches to completeness:
  - Generate whatever possible from the model and fill in the rest by hand, reverse engineering added code back into the model.
  - Aim for 100% code generated from model.
- People more easily believe in partial generation, but this has many hazards including:
  - Model and code become out of sync.
  - Programmer creates code inconsistent with model, e.g. coupled to implementation specifics.

© 2011 Computing Integrity

There are two very different approaches to completeness.

One approach is to generate what one can from the model and fill in the rest by hand, reverse engineering the added code back into the model.

The other approach is to aim for 100% generation.

People have an easier time understanding the partial generation and reverse engineering approach.

But, this approach is filled with hazards including the model and code becoming out of sync and the programmer creating code which is inconsistent with how one would like it in the model, e.g., coupled to implementation specifics.

Model-to-Code Translation

- 100% generation seems desirable goal, since:
  - Model remains architecturally neutral.
  - Model independent of implementation specifics.
- Use of an "Action Language" enables specifying everything in the model, including algorithms and complex business logic.
- Use ABL as Action Language?
  - Means hard coding current implementation.
  - Tempting to embed solutions in the code which belonged elsewhere in the model.

39 To Code Or Not To Code © 2011 Computing Integrity

100% generation seems like an obviously desirable goal since the model remains architecturally neutral and independent of implementation specifics.

The secret to being able to specify everything in the model is "Action Language".

Action Language is how one specifies algorithms and complex business logic in the context of the model.

One might wonder why one wouldn't use ABL as the Action Language.

Using ABL would mean hard coding current implementation requiring hand modification to take advantage of future language improvements.

Using ABL would make it easy and tempting to embed solutions in the code which belonged elsewhere in the model.

Model-to-Code Translation

What about a non-ABL UI?

- Good Model-to-Code fosters language independence from the target language.
- Visual parts of UI likely to need manual tweaking, no matter what language or technology used.

Treat UI as its own subsystem of “realized code”.

 © 2011 Computing Integrity

What about a non-ABL UI?

One of the pluses of good Model-to-Code should be language independence from the target language.

Visual parts of the UI are likely to require a lot of manual tweaking, no matter what language or technology is used.

The answer is to treat the UI as its own subsystem of “realized code”.

Model-to-Code Translation

"Realized Code" and Subsystems

- Good modern design implies the use of loosely coupled subsystems.
- Treat a subsystem as a black box when dealing with another subsystem.
- Any given subsystem may be created by Model-to-Code or by manual coding or a combination as works best.

© 2011 Computing Integrity

"Realized Code" and Subsystems

Good modern design implies the use of loosely coupled subsystems.

One should be able to treat a subsystem as a black box when dealing with another subsystem.

Any given subsystem may be created by Model-to-Code or by manual coding or a combination as works best.

Model-to-Code Translation

Frameworks

- Good candidate for realized code.
- Some/all created by Model-to-Code or manual – doesn't matter to any other subsystem.
- Can include both common infrastructure frameworks and components used like collection classes.

42 To Code Or Not To Code © 2011 Computing Integrity

Frameworks are another good example of a candidate for realized code.

Some or all of the framework may be created by Model-to-Code or it may be manual – doesn't matter to any other subsystem.

Frameworks can include both common infrastructure frameworks and components used throughout the application like collection classes.

Model-to-Code Translation

Multiple Frameworks & Implementation Approaches

- The Model-to-Code engine is independent of the specific framework or implementation approach.
- Connecting the model to the framework is a question of the translation rules used.
- Switching frameworks or implementation is just a question of changing the translation.
- This reinforces not using ABL as an Action Language.

 © 2011 Computing Integrity

Multiple Frameworks & Implementation Approaches

The Model-to-Code engine is independent of the specific framework or implementation approach.

Connecting the model to the framework is a question of the translation rules used.

Switching frameworks or implementation is just a question of changing the translation.

This reinforces not using ABL as an Action Language.

Model-to-Code Translation

Model-to-Code isn't the only translation.

- Model-to-Model transformations are using during analysis and design, e.g., transforming domain classes to classes in the design.
- Code-to-Model translations can be used to begin the transformation of legacy systems.
- Full transformation would be a cycle of Code-to-Model, Model-to-Model, and Model-to-Code.
- ABL2UML is an example of Code-to-Model.

© 2011 Computing Integrity

Model-to-Code isn't the only translation.

Model-to-Model transformations are using during analysis and design, e.g., transforming domain classes to classes in the design.

Code-to-Model translations can be used to begin the transformation of legacy systems.

Full transformation would be a cycle of Code-to-Model, Model-to-Model, and Model-to-Code.

ABL2UML is an example of Code-to-Model.

Model-to-Code Translation

Where are we today?

- Can’t buy it today.
- The translation engine is possibly identified.
- Working on interesting PSC in the project.
- Looking for companies that might fund the work.
- Preliminary estimate is 6 months to have fairly complete translation and 12 months to initial product ready for FCS.

 © 2011 Computing Integrity

Where are we today?

Can’t buy it today.

The translation engine is possibly identified.

Working on interesting PSC in the project.

Looking for companies that might fund the work.

Preliminary estimate is 6 months to have fairly complete translation and 12 months to initial product ready for FCS.

Agenda

- Introduction – Why I Started Code Generation
- Quick & Dirty Code Generation for Standard Components
- Specification-Driven Development
- Model-to-Code Translation
- Summary

© 2011 Computing Integrity

To summarize …

Summary

- Code generators
  - Help keep the code readable
  - Makes bug fixing easy
  - Makes unit testing easy
  - Allow for rapid changes
  - Allows for automatic documentation
    - System and User
  - ABL generation is *fast*
    - Don’t believe all that you hear ;)

Summary

I have talked about two efforts:

- SDD illustrates a fairly simple technology with a modest investment that provided a huge boost in productivity while producing highly functional, maintainable, stable code.
- Model-to-Code represents a larger investment, but one likely to yield 5-10X improvements in productivity for creating original applications while improving analysis and design and providing highly nimble response to changed requirements.

© 2011 Computing Integrity

I have talked about two efforts:

SDD illustrates a fairly simple technology with a modest investment that can provide a huge boost in productivity while producing highly functional, maintainable, stable code.

Model-to-Code represents a larger investment, but one likely to yield 5-10X improvements in productivity for creating original applications while improving analysis and design and providing highly nimble response to changed requirements.

For More Information, go to…

- Rapid Business Change and ABL Productivity
  - http://cintegrity.com/content/Rapid-Business-Change-and-ABL-Productivity
- A Path to Model-To-Code Translation in ABL
  - http://cintegrity.com/content/Path-Model-Code-Translation-ABL
- OERA Open Source Initiative
  - http://www.oehive.org/OERAOSI

© 2011 Computing Integrity

Thank You!

© 2011 Computing Integrity

Thank you.

Questions?

For more information:

Julian Lyndon-Smith
http://www.dotr.com/
julian@dotr.com
+441702444711

Dr. Thomas Mercer-Hursh
http://www.cintegrity.com
thomas@cintegrity.com
510-233-5400

 © 2011 Computing Integrity

And now for questions.