REVISED MARCH 2020

Best Practice Embedding

This paper is a year 2020, revision and synthesis of three papers written originally in 2014, while the technique, *now standard practice*, was still under development and discussion.

This embedding technique which today, we've come to call **CHT BEST-PRACTICE EMBEDDING** is used everywhere in CHT's HNDAPPS collection, so there are plenty of examples and video to review if you need to see more.

Best Practice Embedding: Point One

Best Practice Embedding *increases structure to reduce the tangle.*

When an event such as a button push or a field alert needs to trigger some hand-written code, the most obvious thing we all do is put the code into the relevant event slot for that control. Fine, if an embed is one or two lines of code.

But what if it takes several hundred lines of code? Such large, often structureless embeds are messy, and difficult to debug once they're in place. And the more of them there are, the messier your embedded procedure becomes. It all gets to be a bit too much to untangle for someone having to work after the fact with your applications.

When that "someone" is you, two or three years down the line, and you have to fix, advance, improve, and debug, reams of embed code, the practice of procedure-enhancement through embedding calls out for a *Best Practice* that helps to solve once and for all, some of the counter-productive issues that arise with procedure code-embedding.

Best Practice Embedding: Point Two

Best Practice Embedding *reduces the chances of significant code loss through embed orphaning.*

Embed orphaning is almost unavoidable in "dynamic" applications that are maintained over a long period of time, as the interface evolves and functionality changes with end-user requirements. But it stands to reason that the shorter your average piece of embedded code, the less code you'll stand to lose should the inevitable happen, and some embeds become orphaned due to the removal of a control or a template.

If your average in-line embed is one or two lines of code that merely *calls* into a local class function where the "actual" code resides, then you really haven't lost any functional code at all. Only your event-hook -- *that button or menu you intentionally or unintentionally removed* -- is lost and can be restored with one line of function-evoking code placed under some other control or in some other event slot.

Best Practice Embedding: Point Three

Best Practice Embedding *improves code design, reliability and re-use, while reducing the amount for repetetive code.*

When functional code is placed in-line into an ACCEPTED event slot under a window control or into an ALERT or WINDOW event slot it becomes difficult to call that code from some other event slot or from elsewhere in your code. Yes, you can POST an event to the window or to the control harbouring this in-line code, but in a busy window there's no absolute guarantee that your event code is going to execute in some required order, before or after other code that also must run, due to an interdependency.

Such is the nature of the beast. Inevitably, for this reason, when hand-embedding, highly similar in-line event code with slight variations, tends to be repeated from one event-slot to the next.

You may tend to counter this particular point with the notion that you always place reusable code into a local routine and then abstract your routine code enough to handle minor functional variances based upon which event invokes the routine.

That's certainly an improvement over in-line embed code from the standpoint of code reuse, but it's far from perfect, especially when your routine is operating on procedure-wide variables.

And most routines do this, because routines dont take input parameters. So the same reasoning surrounding the difficulty of controlling the order of window events, can be used to diminish the suitability of routines, because you're not able to *absolutely control* the values in your procedure-wide input variables in the context of multiple-cascading events.

Best Practice Embedding:
Point Four

***Best Practice Embedding** improves code legibility by sponsoring the use of meaningful, consistent entity names, without having to compulsively hunt through myriad template interfaces to rename template-generated objects and controls.*

To ensure uniqueness, templates tend to name things generically with template instance numbers attached. A classic example of this is BRW3, BRW5...BRW12. And of course, we've all silently cursed QUEUE:Browse:3, QUEUE:Browse:5 under our collective breaths.

But let's be realistic. As long as only template-generated code is referencing and using these names, there's absolutely nothing wrong with these names. The code will always work because the template knows which object or queue or control it's referring to. There's no real motivation to rename these things meaningfully, unless you like to spend a lot of time reading template-generated code over which you have little or no control.

Unfortunately, as soon as you begin having to address these generically named OOP objects, data entities and window controls, inside your embed code, the guess-work starts. Since *YOU DO* have to be able to read embed code, both now as you're writing it, and in the future as you're revising it for the third or fourth time, names *DO MATTER*.

Many Clarion developers, myself included, counter this particular point by changing template generated generic entity names where possible. Some do this passionately, religiously, obsessive-compulsively, *everywhere*. I tend to do it only where it impacts my ability to easily read and understand my hand-embedded code. When only template generated code is present, I tend to *LET IT BE*.

Why so little passion on my part, with this issue? Well, keep in mind, the prime motivation for our having derived a need to rename things more meaningfully in the first place, is to make OUR embed-code more readable and ultimately more maintainable. Generated code can take care of itself.

Best Practice Embedding:
Point Five

***Best Practice Embedding** improves code development speed.*

It stands to reason, that a major benefit deriving from points 1 through 4, is increased code development speed. So this fifth point, while true and valid in its own right, really derives from my previous four points.

In-line embed code is kept short -- often only one line of code -- and can be placed into a given embed point in seconds.

Critical embed code is less likely to be lost to orphaning, and can be re-attached into the procedure event structure with no great effort or expenditure of time.

When embed code can more readily be reused by different event triggers inside the procedure it stands to reason you'll be writing less code, reading less code and in the long run, maintaining less code.

When embed code is clear, concise, structured and uses good naming methodology that eliminates frequent name-hunting trips to the window structure and to the various template interfaces attached to the procedure, coding will proceed more smoothly and quickly.

SO WHAT'S THE TRICK?

No trick at all, really.

Best Practice Class-Based embedding uses a generated, intra-procedure class in a slightly new way than you've seen it used in the past.

Let me first define by what I mean by *generated, intra-procedure class*. Aren't the classes attached to my procedures by various ABC and third-party templates also generated intra-procedure classes? Well, no, not really, not exactly, anyway.

ABC-compliant templates generate intra-procedure *derivatives* of external, pre-existing, classes into your procedure. External classes -- or Library classes, if you will -- like the ABC classes in \libsrc\win\, the HND classes in \accessory\libsrc\win\ and numerous others, exist with pre-built functionality in them. The code in these library clases is pre-written and often highly abstracted. *properties* and *methods* are latched into your procedure

Contact Us

Click the link below to contact us by email.
It will start your email client with our email address inserted:

Click To Contact Us