Towards Reusable Spreadsheet Code: Experiments to Create and Interchange Encapsulated Excel Modules

Abstract
The ability to reuse code is an important element of productivity when using traditional programming languages. Code reuse is rarely done with spreadsheet programming languages. A necessary but not sufficient condition for code reuse is the ability to apply the principles of encapsulation and interchange. We present the results of experiments using different protocols for establishing encapsulated spreadsheet code modules, and for interchanging a revised encapsulated spreadsheet code module from a sandbox location into a production location. We discovered that Excel worksheet-scope names are unstable under worksheet interchange. We find that encapsulation and interchange are feasible with spreadsheet code, with caveats. Highly disciplined programming practices are mandatory. Scalability to large numbers of code modules is an issue that requires further study. We explain why source code management systems might be feasible with spreadsheet code modules, and have potential to take care of the scaling challenge. Although much work remains to be done, this is an important step towards the possibility of reusable spreadsheet code.
Introduction

Spreadsheet programming languages (notably Microsoft Excel) are famous for their flexibility and accessibility to unskilled programmers. However, spreadsheets are not known for their ease of maintenance and updating, nor for their suitability to reuse existing code without substantial rewriting. For this reason many expert spreadsheet programmers prefer to rewrite code from scratch than try to reuse it [Grossman and Özlük 2010].

This is unfortunate, because code reuse is a central element of professional programming using traditional programming languages (i.e., 3rd-generation or object-oriented). Code reuse is a significant contributor to productivity of professional programmers. Spreadsheet code reuse is an important prize, especially for important spreadsheets that are carefully designed and built. Our focus in this paper is native-spreadsheet approaches (no VBA) that don’t use proprietary tools.

There is essentially no literature on spreadsheet code reuse. Paine 2005 is a notable exception; his sophisticated and innovative approach is to abstract the underlying code, manipulate the abstraction, and then generate new code. However, it is not a direct code reuse approach, such as has proven so effective with traditional languages.

In any programming language, code reuse requires that a code module be designed with two properties, “encapsulation” and “interchangeability”. The property of encapsulation allows a module of code used in one program to be used in another program with little or no editing of the internal logic of the code, while restricting external access from other code modules. The property of interchangeability means that a reliable and reasonably efficient mechanism exists to “merge” an encapsulated module of code into a program, and to “swap out” an old module of code and “swap in” a new module of code. (For those versed in structured programming, encapsulation roughly corresponds to the idea of a “subroutine”, and “interchangeability” roughly corresponds to the idea of external link management.)

Encapsulation is a concept from object-oriented programming. In the spreadsheet context, encapsulation means that a code module accepts a precisely defined set of inputs, and generates a precisely-defined set of outputs, and that an encapsulated module has “high cohesion” and “low coupling”. High cohesion means that a code module has limited scope. Low coupling means that a code module requires minimal or no access to data that is not specified in the list of inputs, and that other modules do not access cells that are not specified in the list of outputs. More formal definitions can be found in [McConnell 2004, Wallen 2005, Encapsulation 2015, Freeman and Bates 2004, Gamma and Johnson 1994].

This paper explores some experiments we conducted on how spreadsheet modules can be encapsulated and interchanged, with the eventual aim of making reusable spreadsheet code a reality. The goal was to determine how the principle of code reuse can be used in the context of Excel, and discover the limitations and potential opportunities, and the likelihood that unsophisticated users could easily reuse code.

The benefits of encapsulated, interchangeable spreadsheet code modules

The spreadsheet developer who writes encapsulated, interchangeable spreadsheet code modules will accrue many productivity benefits. These include the ability to revise a code module in isolation from the rest of the program; the ability to revise a code module in a “sandbox” separate from the main
program, and then “merge” the revised code module back into the main program with minimal effort; the ability to have other parts of the program access code module outputs without knowledge of what is in the code module (i.e., treat the code module as a “black box”); the ability to take a code module from one program and reuse it, with no or minimal modification, in another program; the ability to have one programmer to work on one code module while other programmers work on another code module. It might even be possible to use a source code management system to manage spreadsheet code modules, or to establish “libraries” of spreadsheet code modules as is done for code in traditional languages.

**Approach**

We experimented with techniques for code encapsulation, and for interchanging encapsulated code modules using the dominant spreadsheet, Microsoft Excel. We seek to understand the feasibility of these concepts in the spreadsheet; to understand the level of discipline and standardized procedures necessary to implement them; obtain a sense of the time and effort required; explore how these techniques might scale to sizable numbers of code modules; and see if there were any unexpected behaviors by Excel (there were). The experiments are described in the next sections.

**Experiments Creating Encapsulated Excel Code Modules**

In the spreadsheet context, encapsulation means that a code module accepts a precisely defined set of inputs, and generates a precisely-defined set of outputs, and that an encapsulated module has “high cohesion” and “low coupling”.

To our knowledge, there has been no empirical research on the cohesion or coupling of spreadsheet modules. Our observation of spreadsheets in industry, and spreadsheets built by students, is that high-cohesion (a code module does one thing) is a standard practice that is self-discovered or cheerfully adopted with a modicum of training.

In contrast, the degree of coupling among spreadsheet modules can be high. Although there is no empirical research on this aspect of spreadsheets, our observation is that coupling can be high or low. Traditional programming languages have features that support or even compel low coupling. Trained professional programmers eschew global variables (which are inherently highly-coupled) in favor of local variables, and provide access only to designated variables in a way that is visible in the code. In contrast, the spreadsheet makes all cells global variables, and it is convenient for a programmer to access any cell in any module. In a spreadsheet, low-coupling must be enforced by disciplined coding practices.

We performed an experiment to encapsulate a code module, utilizing the following five principles:

1. **Designate a set of cells as input parameters.** This is easily done by defining a sub-module to contain inputs. This inputs sub-module can contain references to other code modules, and can contain hard-coded data values.

2. **Cell formulas access external references only via the input parameters.** Cell formulas in the code module reference only cells contained in the code module. Cell formulas that require data external to the logic of the code module obtain that data only from cells designated as input parameters. No references to cells contained in external code module are permissible, except by cells designated as input parameters.
3. **Designate a set of cells as output parameters.** This is easily done by defining a sub-module to contain outputs, which are cells that will be referenced by external code modules.

4. **External code modules reference cells only via the output parameters.** Cells in external code modules may reference any cell designated as an output parameter. Cells in external code modules are forbidden from referencing any cell that is not designated as an output parameter.

5. **The cell address of an output parameter cell must not change when the code module is updated.** Cells in external code modules will reference an output parameter cell. Should the cell address change, for example by inserting a row above it without the knowledge of the external module, the external reference may point to the wrong cell, thereby breaking the code. Care must be taken that the address of an output parameter cell address never changes as the code module is updated over time.

We experimented with two mechanisms, worksheet-level and workbook-level.

**Worksheet-Level Encapsulation**

Worksheet-level encapsulation places all the code of a module on a single worksheet (or “tab”). It is appropriate to design the tab with three distinct regions (one might call them sub-modules), for the input parameters, the output parameters, and the code that performs model calculations.

The location of the output parameters poses a challenge. As identified in principle 5, a programmer needs to be able to insert new rows into a code module. If new rows are inserted above the output parameters, external code references will break. Therefore, the output parameters must be put near the top of the tab, and the programming team must have the discipline never to insert rows above the output parameters. If columns are inserted, the programming team must take care to edit the output parameters to preserve cell addresses.

The location of the input parameters is flexible. With worksheet-level encapsulation, the input parameters are only referenced by cells in the same tab, which update automatically should an input parameter cell be moved, so they can safely receive new cell addresses. Therefore, the input parameters can go anywhere in the tab. (Normal spreadsheet programming practice seems to place input parameters near the top of a tab; this practice can continue, provided the input parameters reside beneath the output parameters.)
We illustrate the three-region design in Figure 1. We place the output parameters at the top, the input parameters below, and the code at the bottom. Other code modules shall reference only cells in the Output Parameters region, in this case cell B2. Note that the contents of the code itself are not displayed, because the actual calculation is hidden from other code modules that access this module.

**Workbook-Level Encapsulation**

Workbook-level encapsulation places all the code of a module in a single workbook. Multiple worksheets within the workbook can be used. A tab of the workbook can house the input parameters, and another tab can house the output parameters. We illustrate this in Figure 2. In Figure 2, we show a single tab for Inputs, Calculations and Outputs. Any number of tabs containing calculations is permissible and may in fact be desirable. Likewise, multiple Inputs tabs and Outputs tabs may be used.

In order to insure that external code modules can reliably access the Output Parameters, the programmer must discipline himself never to change the name of an Outputs tab nor change the cell address of any of the Output Parameter cells on an Outputs tab. The content of the Calculations tab can be modified freely.

**Results of Experiments with Encapsulation**

We find that encapsulation is possible in Excel. It is important to note that the encapsulation of a spreadsheet code module is established by disciplined practice and protocol. This is in contrast to traditional programming languages, where encapsulation is supported or even compelled by the language or development environment.

Our experiment used direct cell references. It is also possible to use Excel names to designate input parameters and output parameters, but this has implications (discussed below) for interchangeability and the use of Excel names was not pursued at this time.

After a programmer has created encapsulated code modules, he needs the ability to interchange them.

**Experiments Interchanging an Encapsulated Code Module**

In this section, we describe experiments interchanging an encapsulated code module. Following standard software engineering practice, we established a “production” module and a “sandbox” module. A production module contains working code. A sandbox module contains code that is being
prototyped and revised. When work in the sandbox is complete, the sandbox module is interchanged into the production module.

Our goal is to understand the feasibility and effort required to make the interchanged production code fully functional. We did experiments using encapsulated modules at the worksheet level, and at the workbook level.

**Interchange of Worksheet-Level Code Modules Using Cell References**

We created modules at the “worksheet level” where a module is housed on a single worksheet. We created a production workbook that has several encapsulated worksheet-level code modules, where each code module is contained on a single tab.

We created a sandbox workbook by making a copy of the production workbook. In the sandbox workbook, we edited the “Demo” tab. At the same time, other programmers were working on other tabs in their own private sandboxes. At some point, it will be necessary to interchange our revised Demo tab into the production workbook, and other programmers will interchange their revised tabs into the production workbook as well. How can this interchange be performed.

We interchanged the sandbox “Demo” worksheet tab into the production worksheet using different approaches.

**Copy-Paste Approach:**

1) In the sandbox workbook, select the entire “Demo” tab and perform a Copy operation.
2) In the production workbook, select the entire “Demo” tab and perform a Paste operation.

This seems to work. However, this approach is vulnerable to user error in the copy and paste operation. In addition, the unit of operation is at the worksheet, and to be moving a mass of individual cells feels like the wrong level of abstraction. Some spreadsheet programmers are uncomfortable using this approach. For us, the risk of pasting onto the wrong worksheet is worrisome. It might become more risky if there are multiple worksheets to be interchanged at once. So it is a strange situation: An approach that seems to work, that we are reluctant to recommend.

**Direct Exchange Approach:**

1) In the production workbook, delete the existing “Demo” tab
2) Copy the “Demo” tab from the prototype workbook into the production workbook

We examined the worksheet and noticed the following:

1) Any output cells that referred to the original “Demo” worksheet now had “#REF” displayed in them. For example, “=$E$3*#REF!B5”.
2) When the “Demo” worksheet was deleted from the production worksheet, Excel “forgot” all references to the original “Demo” worksheet, and didn’t restore the references when the prototype “Demo” worksheet.

In order to fix the problem, we had to manually update every cell that depended on the “Code” worksheet. This is easily fixed by performing a single Find/Replace operation on the worksheet. On very large or complex spreadsheets, the issues we discovered may require more time to resolve. This
approach doesn’t provide the necessary encapsulation and interchangeable functionality that we’re seeking, so we don’t recommend this approach.

**Worksheet Renaming Approach.** To try to avoid the problems with references we experienced with the Direct Exchange Approach, we did the following:

1) Copy the “Demo” tab from the sandbox workbook into the production workbook. We now have a tab named “Demo (2)”.
2) For every cell in the production workbook that references the “Demo” tab, change the worksheet reference from “Demo” to “Demo (2)”. This can safely be done using a global change on each worksheet in the workbook.
3) Delete the existing “Demo” tab
4) Rename the “Demo (2)” tab to “Demo”

We were able to successfully exchange a worksheet, but this did require additional work for changing the cell references and deleting the original worksheet. The effort required was modest, but could become tedious or risky if the production workbook had a large number of Calculations tabs. It is notable that the effort involved was much higher than the trivial effort required for performing a similar action using traditional software. It is likely this process could be automated by a VBA macro.

**Worksheet-Level Interchange Using Excel Names**

We continue using the same tabs as in the previous section. Now, we use Excel names rather than cell references. The Excel “name” feature allows a descriptive name to be assigned to a cell or range of cells, and that name can be referenced just like an A1 reference. We seek to understand the feasibility and effort required when a worksheet-level code module is interchanged. We used the same three approaches as above.

**Copy-Paste Approach:**

When there are Excel names in the worksheet, an intimidating series of dialog boxes opens.

![Microsoft Excel error message](image)

By clicking “Yes” to each, the results seem to work. However, this approach is intimidating. The possibility of clicking through the name warnings, and perhaps missing a dialog box that should not be clicked through is also a source of concern. More extensive and careful experimentation should be done before this approach can be recommended.

**Direct Exchange Approach:**

1) Delete the existing “Code” worksheet in the Production workbook
2) Copy the “Code” worksheet from the Sandbox workbook into the Production workbook.
We were able to successfully exchange the worksheet, but we noticed the following upon closer inspection:

1) Name Manager showed an intimidating set of name conflicts and broken references, as follows:

![Name Manager screenshot]

2) When the original “Code” worksheet was deleted, the named ranges weren’t deleted as we expected. This resulted in duplicate names that must be removed.

3) The names in the Sandbox worksheet had the default workbook-scope. After copying the “Code” worksheet, the scope for each name was changed to the “Code” worksheet, which was unexpected. We expected the scope to remain unchanged. In this case, the scope changed from global to local.
4) The output cell in the “Output” worksheet now refers to broken cell reference that must be resolved manually:

![Image of Excel worksheet showing broken cell reference]

We conclude that Excel names should not be used for interchange. Further research is merited to understand how Excel names respond when sheet-level interchange is performed, as it is desirable to be able to make this work satisfactorily.

**Worksheet Renaming Approach.**

To try to avoid the problems with references we experienced with the Direct Exchange Approach, we did the following:

1) Copy the “Demo” tab from the sandbox workbook into the production workbook. We now have a tab named “Demo (2)”.
2) For every cell in the production workbook that references the “Demo” tab, change the worksheet reference from “Demo” to “Demo (2)”. This can safely be done using a global change on each worksheet in the workbook.
3) Delete the existing “Demo” tab
4) Rename the “Demo (2)” tab to “Demo”

Our results were the same as with the Direct Exchange Approach. The scope changed from global to local, which is problematic.

**An Unpleasant Surprise: Worksheet-Level Interchange Causes Worksheet-Scope Excel Names to be Unstable**

A standard practice in traditional programming languages is to avoid the use of “global variable” which are variables that can be accessed by any code module, and instead use “local variables” that are accessible only within a code module. Excel cells are automatically global. Excel names default to be global, which Excel calls “workbook scope”. Excel provides a limited ability to have local variables, by setting a name to be worksheet-scope, meaning that only cells on that worksheet can use the name. This has the benefit that when coding on a different worksheet, the name in question cannot be referenced, and is not visible in the name box.

To our surprise, we discovered that Excel names with worksheet scope names are unstable when interchanged. More specifically, worksheet-scope names are unstable when the worksheet is moved or copied to a new workbook.
In our experiment, both the production workbook and the sandbox workbook have a tab called “Demo”, containing defined Excel names that had their scope limited to the worksheet. As would be normal practice, the two worksheets had the same names defined, using the same cells for each definition. The sandbox worksheet receives some minor modification to the code that does not affect the names. The task now is to interchange the tab from the sandbox into the production workbook. We use the following procedure:

1) In the production workbook, delete the existing “Demo” tab.
2) Drag the “Demo” tab from the sandbox workbook into the production workbook.

We encountered the following surprising challenges:

1) Name Manager showed an intimidating set of name conflicts and broken references, as follows:

![Name Manager screenshot]

2) When the original “Code” worksheet was deleted, the named ranges weren’t deleted as we expected. This resulted in duplicate names that must be removed manually.
3) The names in the Sandbox worksheet had worksheet scope. After copying the “Code” worksheet, we noticed that an extra name was added with worksheet scope, which was unexpected. To us, it seems like an undesirable language feature or even a bug in Excel. This behavior requires individual editing of every sheet-level name to reset its scope to worksheet level.
4) The output cell in the “Output” worksheet now refers to broken cell reference that must be resolved manually:

![Image showing a cell with a broken reference in the Output worksheet]

5) The cell in the “Demo” sheet referred to in the Output Parameters on the Inputs sheet now refers to an absolute path of the original Sandbox file. This will result in a “#REF” displayed in the Output Parameters sheet.

![Image showing a cell with an absolute path reference in the Demo sheet]

Sheet-level names are unstable when interchanged. Individual manual editing of each name is required. This is tedious and risky. It does seem amenable to automation.

We conclude that names with sheet-level scope should be avoided when interchanging encapsulated (or unencapsulated) code modules. This is unfortunate.

Worksheet Renaming Approach.

To try to avoid the problems with references we experienced with the Direct Exchange Approach above, we did the following:

1) Copy the “Demo” tab from the sandbox workbook into the production workbook. We now have a tab named “Demo (2)”.

2) For every cell in the production workbook that references the “Demo” tab, change the worksheet reference from “Demo” to “Demo (2)”. This can safely be done using a global change on each worksheet in the workbook.

3) Delete the existing “Demo” tab

4) Rename the “Demo (2)” tab to “Demo”
We were able to successfully exchange a worksheet, but this did require additional work for changing the cell references and deleting the original worksheet. The effort required was modest, but could become tedious or risky if the production workbook had a large number of Calculations tabs. It is notable that the effort involved was much higher than the trivial effort required for performing a similar action using traditional software. It is likely this process could be automated by a VBA macro.

**Workbook-Level Interchange**

A code module can reside in a workbook. To connect the code module to other code modules requires establishing Excel links. This is done as follows:

1. In the source Excel worksheet that you wish to link to, select a cell and copy it
2. In the destination Excel worksheet, select a destination cell and select Paste Special->Paste Link
   a. This will display something similar to the following in the formula bar:
   
   
   \[ \text{"=[WORKSHEET.XLSX]Sheet1!$B$3", where B3 is the cell in the source worksheet} \]

Links provide an easy way for code module to take an input from another code module. To insure encapsulation, the programmer must be careful that only the input parameters of a code module can contain a link to bring in data from another code module, and only the output parameters of a code module can be referenced by a link from another code module.

Links can be challenging to manage and hence are perceived to be risky. Excel hardcodes the absolute file path into the link, rather than using a relative file path. In the example above, the formula bar would display something similar to the following: “=[C:\Users\enik\Excel\WORKSHEET.XLSX]Sheet1!$B$3”. If a workbook is moved to a different folder, including being emailed to someone, the link will break. This can be fixed, but it requires individually editing each broken link, which is problematic. Therefore, any use of links must take due consideration of the challenge of link editing and management.

If the workbooks are managed such that the file locations rarely or never change (for example, they are located on a shared server), the challenge of altered file locations is greatly simplified. In this case, there is an easy way to perform interchange that seems to be stable, low risk, and low effort. The procedure is as follows:

1. Make a copy of the production code module workbook, which will be used as a sandbox.
2. Modify the sandbox workbook, without altering the input or output parameters.
3. Move the existing production workbook into an archive location for safekeeping.
4. Rename the sandbox workbook so it has the exact filename and path as the original production workbook.

This approach is simple, effective, and in our experiments reliable. It is important to not change the location of any of the files because that will break the links. This approach should be low risk provided the number of workbook-modules is modest. Traditional programmers have abandoned this approach in favor of source code management systems, in part because it scales poorly; when the number of files gets large, it is easy to make mistakes, lose files and corrupt data.

We conclude that workbook-level encapsulation works well, provided that the directory locations are stable. If the directory changes (e.g., the code is emailed) manual link updates will be necessary.
It would be a valuable contribution to identify or develop tools to simplify or automate the process of archiving an old production workbook and replacing it with a sandbox workbook. It would also be valuable to have tools to automate the updating of workbook links, e.g., the ability to dynamically redefine links.

Source Code Management Systems Have Potential to Be Used with Spreadsheets

Source Code Management systems (SCMs) are utilized by software engineers to help manage, organize and control changes to source code files, and are integral to the widespread practice of code reuse with traditional programming languages. SCMs allow the programmer to have multiple versions of each source code file, and specify which version shall be used in the production code. SCMs provide a high level of automation such that the mechanics of moving/renaming files, managing multiple versions of the same file and so forth are handled instantly and invisibly to the programmer. SCMs greatly increase productivity and are standard equipment in all traditional programming shops. Although they’re primarily targeted towards source code for Java, C++ and other languages, SCMs can be used for any kind of file, including Excel workbooks. [Mitchell 2013, SVN 2015, BetterExplained 2015].

In the context of Excel, SCMs can only work with individual workbook files, not individual worksheets, so workbook-level modules are required. In the approach we described above, interchangeability requires that revision control be done using stable file names for the production workbooks, in order to avoid the challenge of link management. Therefore, if an SCM could be configured to maintain version control of workbooks while maintaining unchanging filenames for the production modules (e.g., through a predefined code module hierarchy), the issue of link management would not arise and the SCM would be effective with spreadsheet code modules.

This hypothesis merits further investigation and testing. There are many SCMs to choose from, and deeper awareness of how each of them functions is required. This is an area that is ripe for further research.

Alternatively, an SCM that changes file names or locations, and therefore requires links to be updated, could be a viable option if a highly-automated means of updating the links was integrated into the SCM. This possibility also merits further research.

Conclusions

We conclude that the foundation elements of code reuse is feasible for spreadsheet code modules, with some caveats. The foundation of code reuse is encapsulation and interchange. The use of worksheet-level encapsulated code modules raises challenges with regard to Excel names. We recommend the use of workbook-level encapsulated code modules, where each code module resides in a workbook.

Encapsulation requires careful definition of input parameters (which reference cells in other workbooks) and output parameters (which are referenced by cells in other workbooks). The location (sheet name and cell address) of output parameters in a workbook must never change. Cells in a module other than
the input parameters and output parameters must never be referenced from or to by cells in another module.

Workbook-level code modules can be interchanged by placing the code modules in a stable, unchanging directory location (such as on a shared server), and by carefully renaming a new module to the filename of the old module. We see potential for source code management systems, that are widely used in traditional programming, to be used for managing Excel code modules, although further investigation is needed.

We do not conclude that spreadsheet code reuse is practical. That remains to be shown, and there are issues to be more thoroughly explored and understood. However, we have established the basic mechanics of encapsulating and interchanging spreadsheet code modules, which is a necessary but not sufficient condition for code reuse.

**Disciplined practices are essential**

With a traditional programming language, it is impossible for another routine to access intermediate calculations of a subroutine. The only values that are accessible to another routine are the output parameters of the subroutine, and of course any global variables that are modified by the subroutine. This restriction is instrumental to code replacement and reuse.

In contrast, the spreadsheet imposes no such restrictions. The programmer who desires to replace or reuse code must discipline himself to allow a similar level of limited access. The choice to follow such a discipline might seem restrictive, unaesthetic, or counter-intuitive. However, the programmer with the intent to create spreadsheet code module that can easily be replaced or reused must have the wisdom to do what serves his goal rather than what seems intuitive; this is the essence of technical acumen.

**Code reuse is not practical for unsophisticated users**

The experiments done in this paper show that there many roadblocks to code reuse. There can undoubtedly be overcome. However, we conclude that unsophisticated users will not be able to reuse spreadsheet code without outside assistance.

**There are opportunities for new tools and functionality**

There exists a clear need and opportunity for spreadsheet engineering development tools. These would help increase productivity by reduce time spent on rework and automating tasks that would otherwise take significant resources and time to complete. These might be in the form of add-ins, plugins or libraries that can be easily installed.

There is a clear opportunity for a “Link Manager” tool that can update and redefine links between Excel workbooks from an intuitive interface. Without such a tool one must manually edit the links, which is very time consuming and error prone. This would be particularly useful when moving a code module from a sandbox into a production environment. Several such tools exist, but the strengths and weaknesses of each are outside the scope of this research.

We identified an opportunity to use source code management systems with spreadsheets, but they can only operate effectively with entire Excel workbooks. To maximize the value of SCMs, we would need a tool that can access the individual worksheets and link them to an SCM. SCMs have potential to be valuable tools to spreadsheet engineers.
In order to provide the higher level of encapsulation that is available with traditional programming languages, Excel would need to support the use of “access modifiers” to directly control access to spreadsheet cells [Access Modifiers 2015].

Further research
Further work should continue this line of work to demonstrate an example where code reuse works in a practical way. This should be done using workbook-level modules, followed by a demonstration of the use of a source code management system to manage the modules.

This paper examined code reuse in different, independent spreadsheet programs. One might seek to use the same code in a single spreadsheet program; with luck this will turn out to be straightforward. The commercial product BPMModules [BPMModules 2015] has a name-based approach to creating modules that might be further explored.
References
Gamma, H., Johnson, V. 1994. “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley Professional, 1994