Proceedings of the 21st International Conference on Software Engineering, IEEE Computer Society, 1999, pp. 97-106

Avoiding Packaging Mismatch with Flexible Packaging

Robert DeLine
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh PA 15213
412-268-2582
rdeline@cs.cmu.edu

abstract

To integrate a software component into a system, it must interact properly with the system's other components. Unfortunately, the decisions about how a component is to interact with other components are typically committed long before the moment of integration and are difficult to change. This paper introduces the Flexible Packaging method, which allows a component developer to defer some decisions about component interaction until system integration time. The method divides the component's source into two pieces: the ware, which encapsulates the component's functionality; and the packager, which encapsulates the details of interaction. Both the ware and the packager are independently reusable. A ware, as a reusable part, allows a given piece of functionality to be employed in systems in different architectural styles. A packager, as a reusable part, encapsulates conformance to a component standard, like an ActiveX control or an odbc database accessor. Because the packager's source code is often formulaic, a tool is provided to generate the packager's source from a high-level description of the intended interaction, a description written in the architectural description language UniCon. The method and tools are evaluated with two case studies, an image viewer and a database updater.

1 introduction

In order to reuse a software component, not only must a developer consider what the component computes, but also how it makes that computation available to other components. A component that expects to interact with other components through procedure calls, for example, is difficult to reuse in a system where components interact by exchanging messages, or by raising and listening for events, or by accessing data in shared memory. The assumptions a component makes about how it interacts with other components constitutes its *packaging*. Today, reusing a component in a new system requires attention to both its functionality and its packaging.

The motivation for selecting a component for reuse is typically its functionality. Often the only preference a developer has about packaging is that it be appropriate to the system in which the component is to be integrated. A Windows developer, for example, will shop for an ActiveX control with the desired functionality for his application, whereas a Unix developer will look for a filter. Today's off-the-shelf components come *prepackaged*: decisions about the component's packaging are made at development time, before the component is made available for reuse. When the packaging decisions encapsulated in a reused component are unsuitable in the context of a new system, the condition is called *packaging mismatch* [6][12].

When packaging mismatch occurs, the system integrator must undo or circumvent the unsuitable packaging decisions, which is often an expensive proposition. In the source code of a conventional software component, the code that accomplishes the interactions tends to be interspersed with the code that accomplishes the component's functionality. This makes it difficult to identify the code related to packaging. When changing the source code is infeasible or overly expensive, the system integrator typically overcomes packaging mismatch by introducing "glue code" in the form of wrappers or mediators [10]. For example, if a component packaged to interact through procedure calls is to be used in an event-based system, the system integrator might place a wrapper around the component that receives events and that makes the appropriate procedure calls on the wrapped component. This glue code becomes another part of the system to test and maintain.

The heart of the packaging mismatch problem is that engineering decisions are being made too early, when too little of the relevant information is known – a violation of Parnas' widely accepted information hiding principle [11]. Since packaging decisions are largely about system integration, they should be deferred until the information about the integration context is known. This paper introduces a method, called *Flexible Packaging*, for structuring a software component's source code to defer decisions about interaction until integration time. Of course, not all decisions about interaction can be deferred: a component's functionality cannot be expressed without mentioning some aspects of interaction. Flexible Packaging provides a mechanism for specifying those aspects of interaction that are essential to the functionality, while

(*a*)

```
for each student S
    total = 0
    count = 0
    for each score s of student S
        total = total + s
        count = count + 1
    report S’s mean is total/count
```

(*b*)

```
#define cellref(C,R) ( sprintf(cell, "%c%d", C, R), cell )

void main(int argc, char** argv) {
    _ApplicationPtr app;
    _WorksheetPtr sheet;
    char cell[10], col;  float total;  int row, count;

    app.CreateInstance(L"Excel.Application.8");
    app→Visible = VARIANT_TRUE;
    sheet = app→Workbooks→Open(argv[1])→ActiveSheet;
    for (row=2; sheet→Range[cellref('A', row)]→Value.vt != 0; row++) {
        total = 0.0;  count =0;
        for (col='B'; sheet→Range[cellref(col, row)]→Value.vt != 0; col++) {
            _bstr_t value = (_bstr_t)sheet→Range[cell]→Value;
            total += atof((char*)value); count++;
        }
        sheet→Range[cellref(col, row)]→Value = total/count;
    }
    app→Quit();
}
```

(*c*)

```
name: John Smith
id: 123456789
score 1: 86.0
score 2: 90.5
score 3: 88.0
name: Sally Jones
id: 987654321
score 1: 92.0
score 2: 91.0
score 3: 80.0
```

(*d*)

```
%{
float total;
int count;
char* name;
}%

%token STRING
%token NL /* newline */
%token INT
%token FLOAT

%%
start: student start | student ;
student: name id scores { printf("%s %f\n", name, total/count); } ;
name: "name" ":" STRING NL  { name = $3; total = 0.0; count = 0; } ;
id:    "id:" INT NL ;
scores:score scores | score;
score:"score" INT ":" FLOAT NL  { total += $4; count++; } ;
%%

main() { yyparse(); }
```

Figure 1. The source code for a component that computes the means of students’ scores, when implemented (a) in pseudocode, (b) as a spreadsheet update, and (d) as a filter that processes records formatted like those in (c).

deferring the incidental details.

As one would expect, every component cannot be packaged in every way; the fact that the functional and interactive concerns can be separated does not imply that arbitrary mix-and-match between them is feasible [5]. This paper focuses on the mechanism that achieves the separation between these concerns. Describing abstract patterns of interaction based on this mechanism, which would allow compatibility checks between the functionality and packaging, is the next phase of the research.

The remainder of this paper discusses Flexible Packaging in more detail. Section 2 uses a simple example to contrast the current practice of packaging components with Flexible Packaging. Section 3 explains the technology and tools behind Flexible Packaging. Section 4 discusses our use of case studies to evaluate the method. Section 5 reviews related research, and Section 6 concludes.

2 flexible packaging in action

To illustrate the problems with today’s component development and integration and to contrast current practice with the Flexible Packaging method, this section uses an example component that is small enough for complete source code to be shown but representative of a larger class of computations. The example component’s function is to compute the arithmetic mean of all the scores of each student enrolled in a course. In pseudocode, this could expressed as in Figure 1(a). Suppose that this averaging component needs to be packaged two ways. For the

(*a*)
```
channel in stream char* StudentNames;
channel in stream float Scores;
channel out stream (char*, float) Means;
char* name;
float score;

while (more(StudentNames)) {
    float total = 0.0, mean;
    int count = 0;
    in(StudentNames, name);
    while (more(Scores)) {
        in(Scores, score);
        total += score;
        count++;
    }
    mean = total/count;
    out(Means, name, mean);
}
```

(*b*)
```
INTERFACE SpreadsheetAccessor WITH
    Names: PLAYER SpreadsheetRead WITH
        range: ( (Col("A"), Row("2")), (Col("A"), FirstEmptyCell) );
        Scores: PLAYER SpreadsheetRead WITH
            range: ( (Col("B"), Current), (FirstEmptyCell, Current) );
        END
        Mean: PLAYER CellWrite WITH
            cell: (FirstEmptyCell, Current);
        END
    END
END
```

(*c*)
```
INTERFACE Filter WITH
    In: PLAYER StreamIn WITH
        format: Seq [
            Lit("name:"), Named("Name", Plus(InSet("0-9"))), EndOfLine,
            Lit("id:"), Plus(InSet("0-9"))), EndofLine,
            Star(Seq [
                Lit("score "), Plus(InSet("0-9")), Lit(": "),
                    Named("Score", Star(InSet("0-9."))), EndOfLine ]) ] ;
    END
    Out: PLAYER StreamOut WITH
        format: [ PrintVar("Name"), Print(" "), PrintVar("Mean") ];
    END
END
```

Figure 2. The component from Figure 1, re-implemented using Flexible Packaging.

first way, both the student scores and the means to be reported are stored as cells in a Microsoft Excel spreadsheet, which the component accesses through ActiveX. For the second, the component is a filter which reads student records, formatted as in Figure 1(c), from standard input and reports the means to standard output.

Current practice: mixing concerns

The source code for the spreadsheet version of the averaging component is shown in Figure 1(b); the source code for the filter version, a script that is input to the parser generator Yacc, is shown in Figure 1(d). Several important aspects of today's practice can be readily seen in this example. First, the code that implements the packaging and the code that implements the functionality are completely intermingled. Indeed, even visually spotting the key lines from the pseudocode version is difficult. (The change bars at the right of the figure highlight those lines of code.) Second, the implementation of the packaging can cause the expression of the component's functionality to be obscured. For example, the arrangement of the code that Yacc induces in Figure 1(d) has little relation to the pseudocode in Figure 1(a). Third, the two versions of the code are very different from one another. Changing the implementation of the averaging component from one version to the other would be challenging.

Separating the concerns

In contrast to current practice, the Flexible Packaging method advocates and supports the separation of a component's functionality and its packaging into distinct software artifacts. The component's functionality is encapsulated in a reusable part called a *ware*; its packaging, in a reusable part called a *packager*. The ware and packager, when compiled together, form a complete component.

Figure 2(a) shows the source code for the ware for our averaging component. It is written in a language called Ciao, which is the C programming language supplemented with high-level constructs for describing the intended interaction with other components. Packagers are also written in Ciao. Rather than being implemented by hand, a packager's source code is typically automatically generated from a high-level description of the packaging, written in the architectural description language UniCon. Figure 2(b) shows a UniCon description of the packaging for the version of our averaging component that accesses the Excel spreadsheet; Figure 2(c) shows a description of the packaging for the filter version. When a system integrator gives the ware code in Figure 2(a) and the packaging description in Figure 2(b) to the Flexible Packaging tools, these tools automatically produce a software component whose behavior is the same as the handmade component from Figure 1(b). Alternatively, he could give the ware and the packaging description in Figure 2(c) to the tools, which would create a component like that in Figure 1(d). By coupling the component's functionality with a description of its packaging at system integration time, the component's packaging can be tailored to the context of integration.

```
         ware code                     packager code

(a)      void compute() {              void output_name(char* n) { begin_transaction(); update_field("NAME", n); }
            output_name(n);            void output_mean(float m) { update_field("MEAN", m); }
            ... compute the mean ...   void output_grade(char g) { update_field("GRADE", g); end_transaction(); }
            output_mean(m);
            ... compute the grade ...
            output_grade(g);
         }

(b)      void compute() {              void begin_output() { begin_transaction(); }
            begin_output();            void output_name(char* n) { update_field("NAME", n); }
            output_name(n);            void output_mean(float m) { update_field("MEAN", m); }
            ... compute the mean ...   void output_grade(char g) { update_field("GRADE", g); }
            output_mean(m);            void end_output() { end_transaction(); }
            ... compute the grade ...
            output_grade(g);
            end_output();
         }
```

Figure 3. Using a procedural interface between the packager and the ware causes either (a) the packager's computation to be awkwardly decomposed or (b) the ware's interface to be poorly abstracted.

3 the flexible packaging method and tools

This section describes in detail the method and tools that allow the system integration scenario sketched in the previous section. This section covers design choices at the heart of any software method: what commitments are to be made by whom, what knowledge is needed to make them, when the commitments are made, and how the commitments are expressed.

Setting for the technology

The Flexible Packaging method recognizes three distinct roles in the development and deployment of a software component: (1) the ware developer, whose expertise covers the problem domain that the component's functionality is solving; (2) the packaging specialist, who expertise covers a particular architectural style or component interface standard (such a pipes and filters, ActiveX and com, or relational databases); and (3) the system integrator, whose expertise is in assembling components in a given architectural style. To make these distinctions clear, consider an ActiveX component that analytically solves differential equations. The expertise behind the functionality of this component (the convergence properties of various numerical methods, the effects of floating-point round-off, and so on) is obviously distinct from the knowledge of what it takes for a given piece of software to be an ActiveX component. But notice also that second and third roles are distinct: there are many more Visual Basic programmers who use ActiveX components in their vb applications than developers who create ActiveX components. Given the diversity of the expertise behind these three roles, it would not be unusual for three different people to play them and at different times. As such, the required coordination among the people playing these roles should be kept to a minimum.

The Ciao language

As was illustrated with the example in Section 2, each of the packager and the ware has computation associated with it. The packager's computation achieves the interaction (for example, the calls to ActiveX interfaces in Figure 1); the ware's computation achieves the functionality (the calculation of the mean scores). Given that these two computations must be combined to achieve the component's total behavior, what mechanism should be used to coordinate the two computations, i.e. to allow them to exchange control and data? Given that the packager and the ware are to be produced independently and that the ware should be usable with a variety of packagers (and vice versa), whatever mechanism is chosen must support this independence.

The most ready choice is the mechanism that today's tools support best and that is best understood in practice, namely the procedure call (or its relatives, method call and higher-order function call). Using this mechanism, the packager and the ware could each be encapsulated in its own module, with procedure calls between them for exchanging control and data. Unfortunately, this choice of mechanism violates our desire to make the packager and ware independently reusable pieces. To see this, we'll explore several design alternatives for a small example.

Consider a ware that needs to output three values and a packager that transactionally writes those three values to a database. Figure 3(a) illustrates a procedural interface between the ware and packager that is designed for the convenience of the ware. Although this interface presents a clean abstraction for the ware (there is one procedure to call to accomplish each of the outputs), the packager's implementation of this interface must awkwardly fit the begin and end transactions calls into the implementation of these procedures. Figure 3(b) shows the alternative where the interface is designed to clean up the structure of the packager's computation, but at the price of adding two new procedures to the interface that are fundamentally not part of the ware's abstraction. Alternatively, we could place the transaction brackets and database updates together in the same procedure body in the packager, but this would constrain the ware to perform its output in one batch, rather than incrementally.

In general, a procedural interface serves two distinct purposes: (1) it provides an abstraction of a computational service to the interface caller; and (2) it provides a preliminary decomposition of the interface definer's computation. These two purposes may be at odds. With a procedural interface, one of the two modules must be written to deal with the constraints imposed by the other, which reduces their independence.

To promote independence, the Flexible Packaging method provides its own mechanisms for tying together the packager and the ware: channels, for exchanging data; and coroutines, for exchanging control. Packagers and wares are written in a language called Ciao, which is the C programming language with a few additional constructs:

channel [in|out|inout] [stream] <*type*> <*cname*>;
in(<*cname*>, <*varname*>);
out(<*cname*>, <*expression*>);
alt { (<*in*> : <*statements*>)+ }

The first of these constructs declares a channel, which is a communication medium between the packager and ware and is reminiscent of channels in occam [8]. If one of the two declares the channel as in and the other as out, then data flows unidirectional across the channel from the out declarer to the in declarer. If both declare the channel as inout, then dataflow is bidirectional. (Other combinations of in, out, and inout are reported as erroneous.) The in statement is used for receiving data from a channel; the out statement, for sending data to a channel. Both the dataflow direction and the type of the data specified in an in or out statement must be consistent with the channel declaration. It is erroneous, for example, for an out statement to name a channel declared as in, and it is illegal for an out statement to send a floating point value on an out channel of type int. Stream channels are used to communicate multiple values between the ware and packager; they support a function to express that no more values will be written to the channel ("close") and another to test for this condition ("more"). The alt construct, like its namesake in occam, allows an input to occur from any one of a set of in statements for which input is ready.

The ware and packager computations are run in a style that is analogous to coroutines. The component as a whole has a single thread of control that is passed back and forth between the ware's computation and the packager's. The thread is switched between computations whenever the currently running computation performs an out statement or performs an in statement on an empty channel. This allows one of the computations to produce a value that the other may immediately consume. Because the packager governs the component's interaction and because exchanging a thread of control among components is a form of interaction, the packager's computation always gets the thread of control first.

```
void ware() {                          void packager() {
    channel out char* Name;                channel in char* Name;
    channel out float Mean;                channel in float Mean;
    channel out char Grade;                channel in char Grade;

    out(Name, name);                       begin_transaction();
    ... calculate the mean ...             in(Name, n);
    out(Mean, mean);                       update_field("NAME", n);
    ... calculate the grade ...            in(Mean, m);
    out(Grade, grade);                     update_field("MEAN", m);
}                                          in(Grade, g);
                                           update_field("GRADE", g);
                                           end_transaction();
                                       }
```

Figure 4. The ware and packager from Figure 3 rewritten in Ciao, which allows each to have a clean functional decomposition and a clean interface between them.

Figure 4 shows the example from Figure 3 re-written in Ciao. The thread of control begins in the packager. The packager executes until it reaches the first in statement. Because no value has been yet sent along the Name channel, the thread of control switches to the ware, which executes its first out statement. The thread is then given back to the packager which executes until the next in statement, and so forth. In this way, the ware's computation and packager's computations are effectively interleaved. Notice that this channel mechanism avoids the pitfall discussed before with procedure calls: both the ware and packager may be structured into procedures in whatever way is natural.

The Flexible Packaging toolset includes a Ciao compiler, which translates Ciao files into C code. The channels are implemented with dynamic arrays; the coroutining, with the Windows nt Fiber library (lightweight threads). Ciao was implemented as a language extension rather than a C library to allow channels to be type-checked.

Mismatch-in-the-small

Although the Flexible Packaging method addresses mismatches in interaction among components, there is still the opportunity for mismatch between the packager and the ware. Given that the packager and the ware are independently written in Ciao – typically by two different developers who never meet – there is no reason to assume that their use of channels will be consistent. In particular, the ware and packager may be inconsistent from one another in four aspects of their channel use:

- They may use different names for the same channel (name mismatch). For example, the ware could call a channel "Init;" whereas the packager calls it "Begin."

- They may differently represent the data on a channel (datatype mismatch). For example, the ware may send an ascii string on a channel; whereas the packager expects a Unicode string.
- They may differ in the order in which they use the channels (ordering mismatch). For example, the ware may do an in on channel A then an in on channel B; whereas the packager may do an out on channel B then an out on channel A.
- They may use different numbers of channels to interact (aggregation mismatch). For example, the ware may send one integer apiece on two channels; whereas the packager expects to receive a pair of integers on one channel.

To accommodate name and datatype mismatch, which are anticipated to be quite common when the ware and packager are separately taken "off the shelf" for reuse, the Ciao compiler accepts an explicit map between ware and packager channel names. For each pair of names, the map may also contain a small Ciao program to overcome datatype mismatch. For example, if the ware contains this channel declaration

```
channel in double Grade;
```

and the packager contains this channel declaration

```
channel out char* Score;
```

a map entry that unifies these two channels would look like this

```
(Grade, Score, TypeFixupCode("
    channel in char* String;
    channel out double Real;
    char* s;
    double r;
    in(String, s);
    r = atof(s);
    out(Real, r); "))
```

The fix-up Ciao code must contain exactly one in and one out channel declaration. The Ciao compiler then unifies the fixup code's out channel with the mismatched code's in channel and vice versa (e.g. Score and String are unified, and Grade and Real are unified). Given these name associations, the Ciao compiler then inlines the fix-up code wherever an out statement appears in the original mismatched code (e.g. all out statements on the channel Score). Datatype mismatches on inout channels, which would require a bidirectional conversion, are not currently supported.

In order to accommodate ordering mismatch, the meaning of out statements has been made looser than the equivalent in occam. In occam, out statements block until the corresponding in statement is ready to execute. Although this semantics could have been chosen for Ciao, it would mean that the ware and packager would have to agree exactly on the order in which they use channels. Instead, out statements buffer their data until the corresponding in statement occurs; an out statement never blocks. (Clearly, in statements still block until their corresponding out statements happen; otherwise, there would be no value to assign to the variable in the in statement. If both the ware and packager are blocked on in statements, the resulting deadlock is detected at runtime and reported.)

Although this semantics does accommodate a certain amount of ordering mismatch, there is a price for this looseness: a computation committing an out statement has no guarantee about when, if ever, its sister computation will "react" to the output it has given. Indeed, under this looser semantics, the Ciao compiler does not even insist that there be an in statement corresponding to every out statement. This design trade-off between accomodating ordering mismatch and providing consumption guarantees may need to be revisited as more experienced is gained with Flexible Packaging. With the case studies discussed in Section 4, the tolerance for ordering mismatch proved to be useful; whereas no need for a consumption guarantee was encountered.

The Ciao tools currently do not accomodate aggregation mismatch. A variation on the approach taken for datatype mismatch could be added.

Generating packagers

Although both the packager and the ware are written in Ciao, there are two important ways in which the packager's source code differs from the ware's. First, the packager's source code is both tediously detailed and formulaic in nature. This makes the task of writing the packager's source code both unrewarding and error-prone. Second, whether a given packager's source is directly reusable varies a lot from packaging to packaging. Such packagings as Netscape plug-ins are exactly the same from component to component. No matter what functionality a given Netscape plug-in provides, its interface to Netscape is always the same; thus, a single packager can encapsulate Netscape-plugin-ness and be reused directly with different wares. Other packagings, however, vary from component to component. For example, a packager that is used to access a particular database will contain details specific to that database, like queries that reflect a given schema. This packager could not be used to access a database with a different schema. Hence reusing a particular ware with different packagers is often more plausible than reusing a particular packager with different wares.

Software generation addresses both the formulaic and situated nature of packager source code. Given a high-level description of a component's packaging in the architecture description language UniCon [13], a Flexible Packaging tool, called a packager maker, generates the packager's source code. Examples of such packaging descriptions were shown previously in Figures 2(b) and 2(c). Consider the description of the filter packaging shown in Figure 2(c). It describes the intended packaging in high-level terms: how many input streams and output streams the filter has and, for each stream, the format, given as a regular expression, of the ascii text flowing on the stream. The filter packager maker reads this description and from it generates both the Yacc script needed to parse the input and the print statements needed to produce the output. The result (in this case after Yacc has been run) is a Ciao source file that uses channels to output the results of parsing and to get the input to the print statements. The names of the channels are derived from the filter's UniCon description.

Another important observation about packaging is that for a piece of software to achieve a given packaging is often not just a question of the content of its source code but also the software construction steps used to process that source code. Consider the list of requirements a component must meet to be a Netscape plug-in: it must implement sixteen particular functions and use memory management functions that Netscape provides; it must be compiled into a dynamically linked library (dll) that exports three particular functions and that contains a resource fork with two particular text resources; the dll's name must be in dos 8.3 format and begin with the letters *NP*; the dll must appear in a particular directory. Of these, the first is about the content of the component's source code; the rest, about its construction. Here, too, software generation is helpful. The Flexible Packaging toolset comes with a set of tools, called experts [13]. For a given component packaging, a packaging expert produces construction instructions (in the form of a Makefile) that perform the necessary steps to process the component's source code.

In summary, there are three forms in which a packaging specialist can capture his knowledge. If the packaging does not vary from component to component, the knowledge is encapsulated as packager source code written in the Ciao language. If the packaging does vary from component to component, then the knowledge is encapsulated in the form of a packager maker. This packager maker reads a UniCon description (which captures the dimensions along which the packaging varies) and generates the packager source code. Finally, in either case, the knowledge of the construction steps necessary to achieve the packaging is encapsulated in a packaging expert. Given the variety of packagings in the world and that new packagings appear over time, the Flexible Packaging toolset provides a framework for easing the job of creating new packager makers and packaging experts.

Flexible Packaging at a Glance

Figure 5 summarizes the major tools and files associated with the Flexible Packaging method. The developers playing each of the three roles makes his own independent contribution. The products they create are shown with thicker lines.

The ware developer programs the component's functionality as a Ciao file (or a set of Ciao files), labeled (a) in Figure 5, and places the file(s) on the shelf for reuse.

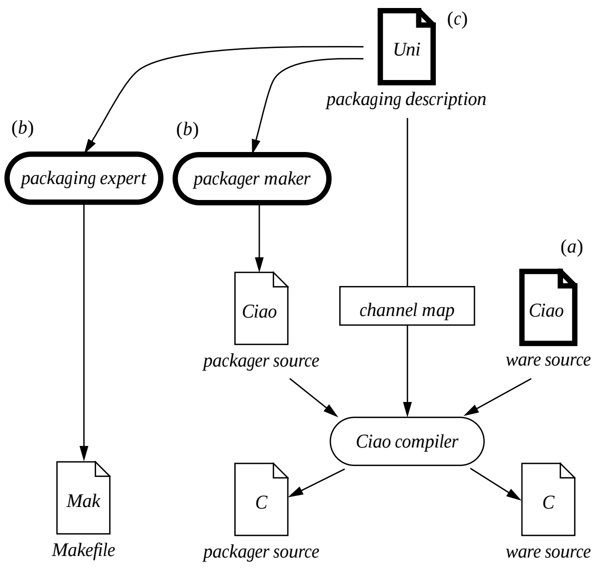

Figure 5. The major tools associated with Flexible Packaging and the files that these tools produce and consume.

The packaging expert, independently, uses a framework provided with the Flexible Packaging toolset to encapsulate her knowledge of a particular packaging in the form of a packager maker and a packaging expert, labeled (b). The packaging expert then makes these tools available for reuse.

The system integrator decides on the required functionality and packaging for a component to be integrated into his system. He acquires a ware that achieves the desired functionality and a set of packaging tools that achieve the desired packaging. He then produces a UniCon description of the desired component packaging, labeled (c). He runs the packager maker on the description, which automatically produces a packager (one or more Ciao files). With a packager and a ware now in hand, the system integrator edits the UniCon description to add a channel map to show the associations between the ware and packager channels and to overcome any datatype mismatch. Finally, he feeds the UniCon description to UniCon, which automatically does the rest. UniCon invokes the Ciao compiler on the Ciao sources to produce standard C source files and invokes the packaging expert on the component's description to produce a Makefile. UniCon then invokes Make on the C sources and Makefile to produce the final component.

4 case studies

To test the feasibility of using Flexible Packaging to develop to "real-world" components, I performed two case studies. Each case study consisted of developing one ware and packaging it three different ways. The emphasis of these initial case studies is on the packagings: Can Flexible Packaging handle the complexity of packagings used in practice today? As such, while the six packagings are all drawn from current practice, the two wares in the study are relatively simple; an on-going case study is testing whether the method works for more complex wares. All of the materials associated with these case studies, including complete source code, are publicly available at *www.cs.cmu.edu/~Compose/packaging*.

Case study 1: Area code converter

In order to accommodate an ever increasing need for new telephone numbers in western Pennsylvania (usa), the 412 telephone area code was recently split into two area codes, 412 and 724. Whether a given phone number remained in the 412 area code or switched to the new 724 area code was determined by its exchange (first three digits). One of the effects of this change is that phone numbers must be updated in many databases and other electronic artifacts. The variety of artifacts to be updated is staggering: traditional databases from a number of vendors, spreadsheets, formatted text files, text documents and document templates, web pages, electronic business cards, address books in contact managers, and many others.

Such a problem provides a natural opportunity to use Flexible Packaging. The goal is to create a family of tools that all share a common function – the ability to update the area code of a phone number – but differ in the kind of database each updates. This case study involved creating three batch programs, each of which accesses a different kind of database: a Microsoft Access database (accessed via odbc); a Microsoft Excel spreadsheet (accessed via com); and a text file formatted with one record of comma-delimited values per line (accessed via ascii streams). The database and spreadsheet programs perform in-place updates of the data, while the text file program acts as a filter. All three programs share an identical ware but use different packagers, generated from UniCon descriptions. All three components run on a Pentium running Windows nt.

Case study 2: PNG image viewer

The Portable Network Graphics (png) image standard was recently designed to be a successor to the popular gif standard. One of the reasons the gif standard still prevails is that many different kinds of software need to display images – drawing programs, document editors, stand-alone image viewers, user interface design tools, web browsers – and each imposes its own packaging requirements on the image-handling component. Creating a png viewing component for each of these niches takes time. Here, too, is a natural opportunity for Flexible Packaging. We would like to capture the functionality of parsing and displaying a png image once and reuse it in many different contexts.

The second case study involved creating three different components for displaying png images: a Netscape (version 4) plug-in; an ActiveX control; and a stand-alone Windows application. As with the previous example, all three programs share an identical ware but use different packagers, generated from UniCon descriptions.

Observations

The primary result of the case studies is an initial validation that Flexible Packaging can be used to develop “real-world” components. Beyond this, several observations can be drawn from the experience.

Non-code artifacts and software construction

Getting a component to have a particular packaging is not always merely a question of calling the right i/o routines in the component’s source code. As was previously mentioned, it can also involve a component’s construction steps, including the creation of non-code artifacts. The two case studies differ greatly in this regard. With the area code case study, achieving the desired packaging *was* a question of calling the right i/o routines: the standard C i/o routines, for the filter; the odbc library, for the database accessor; and Excel’s exported com interface, for the spreadsheet accessor.

In contrast, the png case study involved software construction steps, not i/o libraries. Section 3 sketched the construction steps needed to package a component as a Netscape plug-in. The UniCon Netscape plug-in expert, created for this case study, automates all these construction steps, including the generation the two non-code artifacts involved: a file that the linker uses to guide dll construction and a file that describes the resources (name/value pairs) associated with the dll. Unlike the analogous Microsoft wizard, this expert completely hides the existence of these non-code artifacts. Similarly, the Windows application packaging involves the creation of a resource file. The ActiveX packaging involves the creation of four non-code artifacts: a linker file and a resource file, like those needed for Netscape plug-ins; a description of the component’s com interfaces in the Interface Definition Language (idl); and a file that instructs Windows on how to place the ActiveX component in the system registry.

Internal versus external control

The example averaging component introduced in Section 2 is similar to the area code component and can be used as a surrogate for making an observation. Notice the difference between the pseudocode in Figure 1(a) and its implementation as a Yacc script in Figure 1(d). The former is expressed using what is often called “internal control,” where the algorithm itself determines the order in which the computational steps proceed. The latter is expressed using “external control,” where the content of the data being parsed determines the order in which the steps proceed. In the case of the grading component, except for the breaks between student records, we know exactly what data to expect from input to input, hence expressing the algorithm using internal control is a natural fit; this is simply not a situation where the data drives the computation. The nature of the Yacc tool – the fact that it supports the description of variable data – artificially induces the expression of the algorithm using external control in Figure 1(d). In contrast, as Figure 2(a) shows, the use of channels in the interface between the packager and the ware allows the algorithm to be expressed using internal control, even when the tool used to produce the packager, like Yacc, induces the use of external control.

On the other hand, the nature of the png component is to offer two services – the parsing and painting of png files – that may be used in whatever order and as many times as the client desires. In this case, external control is endemic to the component and is expressed in the ware's main routine in the form of a loop:

```
while ( ! done ) alt { ... }
```

This kind of alt loop is typical of many different kinds of service-based components: rpc-based servers; Unix socket-based servers; components that listen for events or that receive messages; components with user interfaces; and command interpreters. In short, the use of in, out, and alt statements allows the ware developer explicitly and directly to express his expectations about the order and variability of interaction, regardless of how the packager is expressed.

Packaging abstractions

Given the tedious, detailed nature of typical packager source code, the Flexible Packaging method asks the packaging specialist to create two important abstractions. The first abstraction is a set of UniCon definitions, based on which the system integrator will describe his component's packaging. A Netscape plug-in specialist, for example, must decide what it means for a system integrator to describe a component in UniCon as being packaged as a Netscape plug-in – for example, what properties must appear in the description. Because the system integrator's packaging description is the input to the packager maker, the needs of the packager generation process influence the creation of this abstraction.

The second abstraction for which the packaging specialist is responsible is how the packager appears to the ware in form of channels. The design tension here is to hide unnecessary details from the ware without preemptively concealing information that a ware might find useful. As an example, while the sixteen required Netscape plug-in operations provide a strong hint about what information to out to the ware, only some of the operation's parameters are included in the out statements.

Performance

One of the costs of using Flexible Packaging is the run-time overhead that the channel mechanism imposes. This is not a fixed cost, but varies depending on the number of channel communications and the amount of computation performed between those communications. The more channel communications there are between the ware and the packager, the more run-time overhead the component will experience; the fewer computations performed between communications, the more run-time overhead the component will experience.

To measure this overhead, I re-implemented the three components from the area code case study to combine the code from the packager and ware into one module, removing the use of channels. I then measured the difference in execution time between the original and hand-altered versions of the components. Based of these measurements, each component experiences the following percentage of run-time overhead due to channels: 8% for the filter; 2% for the odbc database accessor; 1% for the Excel spreadsheet accessor. The variation is due to the different execution times of the three packagers. Taking consistent measurements for the three png components is infeasible because these components interact with the user.

5 related work

The observation that a component's interaction should be separated from its functionality is not new. Gelertner argued for this separation in his work on Linda [7]. Although Linda does allow a component's interactive and functional concerns to be separated, it does so at the cost of making interaction a second-class concern: the only interaction mechanism between components that he supports is sharing a Linda "tuplespace."

Closer in spirit to this work is that of Callahan and Purtilo [2]. Like Flexible Packaging, their system Nimble also allows a component to be accessed through different interaction mechanisms, restricted to members of a family: procedure call, cross-language procedure call, and remote procedure call. This restriction allows them both to develop their equivalent of wares in standard programming languages and to infer and generate the packager to be used. This work and its later extension to handle mismatches in event systems [3] inspired the channel maps used in Flexible Packaging.

Contemporary projects have also influenced the research on Flexible Packaging. The Flexible Packaging framework for creating packager makers and packaging experts is a locally implemented variation on Batory, Lofaso, and Smaragdakis' Jakarta Tool Suite [1], supporting C rather than Java.

The Aspect-Oriented Programming project [9] directly influenced the expression of a flexibly packaged component as the combination of a ware ("component" in the aop lexicon) and a high-level, declarative description of its packaging (“aspect” in the aop lexicon). While similar, aop and Flexible Packaging have taken different design paths: with aop, “aspects” are always specified relative to particular “components;” with Flexible Packaging, packaging descriptions are specified independently of any ware. As a consequence, the source code generated from an “aspect” can be interleaved (“woven”) with the “component” source code at compile-time; whereas, a packager and ware’s computations are interleaved at runtime via channels.

6 conclusion

This paper introduces the Flexible Packaging method, which allows a component’s functional and interactive concerns to be separated. A component’s functionality is captured in a ware. A ware’s use of channels allows it to specify enough about interaction to express its functionality, while leaving most details unspecified. A component’s interaction is captured in a packager, which may either be reused directly or automatically generated from a high-level description of the component’s packaging. This method supports the following reuse scenario: a system integrator takes a ware “off the shelf” with functionality that she needs, describes the packaging it must have to be compatible with the system’s architectural style, and then uses the Flexible Packaging tools to turn the ware and the packaging description into a software component with the specified packaging. This tailored software component can then be directly integrated into the system.

What is missing from this scenario is a means to tell whether the ware and the packager are compatible. Each of them advertises a list of channels – their names, types, and directionality. Through the process of matching up these channels, the system integrator can tell something about their compatibility; documentation about the ware and packager would have to provide the rest. (This is not very different from reusing a procedure or class library today, where a combination of the procedure/method signatures and the library documentation informs the developer about the library’s use.) The next phase of this research is to create interaction abstractions, namely patterns of channel use that clarify the intentions of the packager or ware’s creator and that support compatibility checking.

ACKNOWLEDGMENTS

I am grateful to Mary Shaw for her timely, helpful feedback on this paper and for suggesting the area code case study. The Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, usaf, and the Advanced Research Projects Agency have supported this work under grant f33615-93-1-1330. This paper represents the views of the author and not of Carnegie Mellon University nor of any of the sponsoring agencies.

REFERENCES

[1] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. “JTS: A tool suite for building GenVoca generators.” In *Proc. International Conf. on Software Reuse*, 1998.

[2] John R. Callahan and James M. Purtilo. “A packaging system for heterogeneous execution environments.” University of Maryland Technical Report cs-tr-2542, 1990.

[3] Chen Chen and James M. Purtilo. “Event adaptation for integrating distributed applications.” In *Proc. International Conf. on Software Engineering and Knowledge Engineering*, 1995.

[4] Melvin E. Conway. “Design of a separable transition-diagram compiler.” *Communications of the ACM* 6(7): 396–408, 1963.

[5] Robert DeLine, Gregory Zelesnik, and Mary Shaw. “Lessons on converting batch systems to support interaction.” In *Proc. International Conf. on Software Engineering*, 1997.

[6] David Garlan, Robert Allen, and John Ockerbloom. “Architectural mismatch, or Why it’s hard to build systems out of existing parts.” In *Proc. International Conf. on Software Engineering*, 1995.

[7] David Gelertner and Nicholas Carriero. “Coordination Languages and their Significance.” *Communications of the ACM* 35(2): 97–107, 1992.

[8] Inmos Ltd. *occam2 reference manual*. Prentice-Hall International Series in Computer Science, 1988.

[9] Gregor Kiczales, John Lamping, Anurag Mandhekar, Chris Maeda, Christina Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-Oriented Programming.” Xerox parc Technical Report spl-97-008, 1997.

[10] Diane E. Mularz. “Pattern-based integration architectures.” Chapter 7 in James O. Coplein and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, 1995. Addison-Wesley.

[11] D. L. Parnas. “On the criteria to be used in decomposing systems into modules.” *Communications of the ACM* 15(12).

[12] Mary Shaw. “Architectural issues in software reuse: It’s not just the functionality, it’s the packaging.” In *Symposium on Software Reusabilty*, 1995.

[13] Mary Shaw, Robert DeLine, and Gregory Zelesnik. “Abstractions and implementations for architectural connections.” In *Proc. Conf. on Configurable Distributed Systems*, 1996.