A Model for the Reuse of Software Design Information

Kevin W. Jameson

June 17, 1999

Abstract

This paper presents a general model for the representation and manipulation of module level software design information, leading to the effective reuse of software design information across different programming languages.

Language independent design documents are represented as ASCII files containing tagged design information sufficient for the construction of a compilable program architecture. The resulting architecture is composed of documented module stubs which describe calling relationships, parameters, functional descriptions, and algorithms characteristic of the architecture. No executable code is included in the compilable modules.

Frameworks of tagged locations in *language dependent* standard module templates are matched against corresponding tags in the ASCII design files, effectively blending language dependent and independent information into a compilable stub architecture. The process is *reversible* in that a language independent design file can be generated from an architecture in the standard tagged format, thus supporting the movement of design information across different programming languages.

Introduction

High software development costs continue to motivate research into reuse of the software development effort represented by software programs, subsystems, and subroutine libraries.

One popular approach to the reuse of design information can be seen in the reuse of existing software modules. This approach attempts to fit previously completed software modules (perhaps from unrelated projects) into the software designs of current projects. The design information in the modules is clearly but implicitly reused, though reuse of the implementation effort appears to be the major motivation of the approach. In the process, the design of the host architecture or of the reused module must often be modified to achieve a suitable fit [2]. A typical implementation of this approach is some form of a subroutine library, possibly augmented with tools for the classification, storage, selection, and retrieval of software modules [3]. The approach can be effective

- if modules are designed for reuse [6] [7],
- if modules are catalogued effectively for retrieval [3], and
- if modules are available in the target programming language.

In contrast to reuse techniques which focus on fitting existing code modules into new designs, the approach taken by this paper is to reuse the old design itself — that is, to fit previously constructed *designs* into the designs of current projects. This method has several advantages:

- It moves the reuse (fitting) attempt to a point much earlier in in the development lifecycle. This saves design phase effort in addition to the normal benefits of reusable software.

- It allows direct utilization of existing programs at the more abstract conceptual levels of architecture (preliminary design) and algorithm (detailed design), instead of just at the detailed code level.

- It supports direct reuse of design work across different programming languages because it provides a mechanism to codify design models in a language independent way.

- Old designs which are incrementally changed and improved in the course of reuse can be directly extracted and preserved in their improved forms.

The reuse method is based upon three software design representations which correspond to the preliminary, detailed, and implemented design stages of the software lifecycle:

1. As *annotated calling trees* which represent the preliminary design (that is, the hierarchical architecture) of a program. This representation is language independent (Figure 1).

2. As *design files* (Figure 3) and *group files* (Figure 4) which represent the detailed (algorithmic) design of a program. This representation is also language independent.

3. As *compilable module stubs* in the target implementation language. This representation is clearly language dependent (Figure 5).

A regular text editor can be used to modify a design in any one of the three representations.

Four software design tools have been designed to transform program designs between the three representations, thus forming a relatively complete, reversible design manipulation and reuse facility:

- **dt**, a design tool for translating *annotated calling trees* into *design file* representations. This tool manipulates and pretty prints a language independent calling hierarchy of the program under design, maintains a list of parameters and data types common throughout the design, and allows module interfaces to be classed by type (subsystem, recursive, utility, system dependent). For each interface in the tree, **dt** generates a minimal design file by placing calling tree information in the appropriate places in a standard design file template. Parameter names are placed under the parameters header, subordinate module names are placed under the external dependencies header, and so forth.

- **gentree**, a tool for translating *design files* back into *annotated calling trees.* This tool constructs an annotated calling tree representation from a set of design files, allowing the developer to manipulate the program design in a hierarchical form. The output format is compatible with the design tool **dt** input file format so as to form a complete "loop" of tools and representations. The module dependency information which determines the structure of the tree is found by **gentree** under the data, external, and system dependencies headers of the design file.

- **stub**, a tool which translates *design files* into *compilable modules*. This tool creates compilable program components from design files and standard module templates. Design information is moved from the design files into the compilable file by matching tags in the design file with identical tags in the template file.

- **extract**, a tool for translating *compilable modules* back into *design files*. This tool extracts preliminary and detailed design information from compilable implementation files. Design information is moved from a compilable file back into a design file by matching tags in the compilable file with identical tags in a standard design file template.

The general idea is for a developer to outline the program structure in the annotated calling tree representation using the **dt** tool, translate the design to design file representation using the **dt** tool, complete the detailed design in the new representation, and finally translate the design into a target implementation language using the **stub** tool. In the reverse direction, the **extract** and **gentree** tools can be used to move a design from the implementation stage back to a language independent design file or annotated calling tree representation, where it can be further manipulated or used to generate a similar implementation in another programming language.

The following sections of this paper describe the representations and procedures of the design reuse method, and conclude with discussions of the current state of the project and the work which remains to be completed.

1 Representations

1.1 Definitions

A *module* is considered to be a logical entity of significance in a software design — for example, a package, function, procedure, include file, data structure template, or definition file for data types or constants. It is usual, but not always the case, for a module entity to correspond to a popular block structured programming language construct.

A *preliminary design* is considered to be either

- A set of compilable module stubs, each of whose contents describe the function, interface, and dependencies of the module, and as a group which adequately describe the characteristics of the final program, or

- A set of non-compilable ASCII design files which can be processed by software tools to generate a set of compilable module stubs as defined immediately above.

The two alternatives are equivalent, as each can be generated from the other using the methods described below. In practice it is easier to meet the ASCII file definition when initially generating a design, as the syntactic and clerical overhead of generating compilable modules is avoided.

A *detailed design* is the same as a preliminary design except that in addition it contains English task comments which define the algorithmic operations to be implemented by executable code. Figures 3 and 5 show detailed designs in the design file and compilable stub representations respectively.

The preliminary design is associated with *architectural design* (the existence and relationships of modules and major data structures), whereas the detailed design is associated with *algorithmic design* (internal module algorithms and internal content of data structures).

Reuse of a design is viewed as any reuse of the architectural or algorithmic information in an existing design. For example, portions of a design are often reused when a program is rewritten or modified for use in a new application or updated version.

1.2 Data File Representations

The design reuse method uses three types of files to store language independent, partially dependent, and fully language dependent information:

- **Design Files** (Figure 3) represent the *language independent* information needed to define program components. Design information is classed and tagged according to a standard set of information categories (that is, according to the headers in the design files) so it can be easily merged into module templates. The information in this file is language independent since it is concerned with purely conceptual entities and dependencies. Simple text strings which identify conceptual entities in the design file are straightforwardly cast into the entity templates (functions, procedures, etc) of the target programming language.

- **Template Files** (Figure 2) represent the *language dependent* information needed to define a program component in the target programming language. Each template file contains

  1. A set of statements which map symbolic data types used in the design file into actual data types in the target programming language. For each conceptual data type appearing in the design file, this file defines a compilable representation of the data type in the target language. The stubbing tool uses these translation rules to translate design file data types into target language data types during the stub generation process. For example, the symbolic type SINT16 (signed integer of 16 bits) is mapped to "integer" in Turbo Pascal, which uses a default signed integer word size of 16 bits.

  2. A set of module templates for each of the possible module entities in the target language (eg. functions, procedures, packages). Each template contains a framework of tagged locations corre-

sponding to a standard set of information categories useful in classifying software design information. All data type translations and templates for a particular programming language are contained in one file.

The template file is normally predefined for each programming language, and remains constant for all designs unless the module templates or data type mappings for that language must be changed or augmented for some reason.

- **Group Files** (Figure 4) have two major functions:

  1. They bind logical module names to internal (language dependent) names and to operating system dependent physical filenames. Group files may thus be both operating system and language dependent (depending on the names they contain).
  2. They define the physical grouping of related modules into one physical file — for example, into one physical compilation unit. The names of files to be grouped are listed in a block structured group statement so a stubbing tool can generate an appropriate output file.

  There is only one group file for any particular program implementation.

1.3 Design Information Representation

This section explains the representation of program designs and discusses the special treatment given to some smaller items of design information.

The representations of module names, return values, parameter data types and external subroutine interface descriptions receive special treatment in design reuse method

Module names and return data types are represented in the template file by *template variables* because they cannot be adequately treated by the general translation rule mechanism mentioned earlier. Specifically, most programming languages use a syntax for the module name and return value specification which cannot be easily treated by bounding the information by headers and applying translation rules. While it is possible to do so, the method grossly distorts the visual appearance of the module. A better solution is to have the stubbing tool replace special *template variables* in the template with the text strings corresponding to the module name and return description (see Figure 2).

Parameter specifications are represented by an identifier/data type pair in the language indendent design file, and are translated by the stubbing tool into suitable target language syntax according to translation rules held in the template file. Similar rules will be used to perform reverse type translation during extraction operations with a design extraction tool.

External subroutine interfaces are also represented by identifier/data type pairs in the design file, where the data type (a special one for external dependencies) indicates to the stubbing tool that the identifier name should be used as a search key into a database of interface descriptions derived from the design files which originally defined the referenced modules. The database (or optionally separate databases) also holds interface definitions for object library routines.

To summarize, the design reuse method uses a language independent file to hold the bulk of the design information, a language dependent file to define data type translations and standard module templates in the target language, and a group definition file to define the physical filenames and grouping of conceptual entities. An auxiliary interface database file is created from the design files to collect interface definitions for all modules for the convenience of the **stub** tool.

The creation of a compilable architecture from these files is accomplished with relatively simple table lookup and text substitution operations, as is the reverse process during extraction and reuse of the design. The most complex part of the process is the translation of symbolic data types into compilable representations.

2 Design Generation

The procedure of creating a preliminary design consists of developing and recording architectural and functional descriptions which characterize the desired program.[1] A design creation process compatible with the design reuse method is outlined below.

1. A normal text editor is used to create a language independent calling hierarchy of modules, annotated with short function descriptions and parameter lists (Figure 1).

2. A translation tool is used to translate the annotated calling tree into a language independent *Design File*. The design file format provides locations for complete descriptions of the functions, interfaces, and dependencies of all program components (Figure 3).

3. A text editor is used to complete the design file description of the program, and to create a *Group File* which defines the language and operating system dependent internal and physical filenames of all the modules defined by the design file (Figure 4). At this point in time, the conceptual program design is complete. It needs only to be translated into a language dependent compilable format.

4. A software stubbing tool is used to create an architecture of compilable module stubs (Figure 5) from the design file, the group file, and the template file. The essence of the stubbing algorithm is to match tags in the design file with tags in the template file, and to copy and translate the associated information as appropriate. In the reverse (reuse) operation, design files can be generated from existing compilable architectures which have been coded in the standard format.

3 Scenarios of Use

The general pattern of tool use is for a developer to outline the program structure in the annotated calling tree representation using the **dt** tool, translate the calling tree to design file representation using the **dt** tool, complete the detailed design in the new repre-

[1] Describing the process of conceptual program design is beyond the scope of this paper, which is limited to describing a method of manipulating such design information *after* it has been conceptualized.

sentation by adding algorithmic comments (visible in the design file of Figure 3), and finally translate the design into a target implementation language using the **stub** tool. In the reverse direction, the developer uses the **extract** tool to move a design from the implementation stage back to the language independent design file representation, where it can be further manipulated or used to generate a similar implementation in another programming language.

In the specific context of the *design phase*, the anticipated iterative pattern of tool usage is **dt**, **gentree**, **dt**, **gentree**, ..., where the developer alternates between the calling tree representation (to abstract or summarize the *architecture* of the evolving design) and the design file representation (to add more detailed *algorithmic* information to the evolving design).

In the context of *implementing the design*, the anticipated linear pattern of tool usage is **dt** to **stub**, wherein the developer generates compilable modules from the design files produced by the **dt** tool.

In the context of *software maintenance*, the anticipated linear pattern of tool usage is **extract** to **gentree** and **dt**, as the developer extracts and abstracts design information from the existing program so that the architecture of the program is made visible and easier to understand.

Finally, in the context of *design reuse*, the simplest anticipated sequence of tools is **extract** to **stub**, wherein the developer extracts the conceptual design from an existing piece of code into design files which are then used to drive the **stub** tool in the generation of a compilable module architecture in another language. If significant changes are to be incorporated during the move to a new language, a more plausible pattern of tool use would allow the developer to move the extracted design back to the **dt**–**gentree** design phase where the changes could be made more easily. Once the desired changes were made, the design information could be carried forward and reimplemented by the **stub** tool in the desired target language.

3.1 Reuse of Designs in Non-Standard Format

The effective reuse of design information according to the model presented here is critically dependent upon a standard module format (such as that in [5]) which explicitly tags interesting design information. Should it become desirable to recover and reuse the design information contained in existing software systems according to the model presented here, the existing software must be modified such that

1. It suitably tags the information with standard headers as discussed above (an operation which will not change the executable code of the system in any way), and

2. A set of conceptual data types are defined for the language and application according to the needs of the data type translation mechanisms in **stub** and **extract**. (This requirement is more difficult.)

4 Conclusion

4.1 Current Progress

Current progress in the project is as follows:

- An initial version of the **dt** design tool has been written. It supports
  - the definition of annotated calling trees,
  - the assignment of module type attributes for different pretty printing treatment (eg. subsystem, recursive, utility, and system dependent types),
  - the direct generation of Pascal module stubs (in Pascal compilation order).
  - the generation of simple design files.

  It does not yet support
  - the production of design files which preserve module type information (subsystem, system dependent, recursive, utility).
  - the production of a group file to accompany the output design files.
  - the use of arbitrary standard module headers.
- An initial version of the **stub** tool has been written. This version
  - supports the generation of compilable architectures in any target language for which suitable data type translation rules and template files are defined.
  - has been used on a daily basis (only by the developers) to generate module stubs and standard architectures in the C programming language for several months.
- Formal testing of the tools for design reuse capabilities must wait until the **extract** and **gentree** tools have been constructed.

4.2 Future Work

The design approach will be tested by implementing several small programs which, as a group, use a variety of different programming language "entity" vehicles (procedures, packages, include files), parameter data types (integers, pointers, structures), and passing mechanisms (input, output, update, return parameters). Each program will be designed and implemented in one language, then extracted and moved to the other languages. The list of test programming languages includes ADA, Assembler, C, Lisp, Modula2, Turbo Pascal, and PL/1. A personal computer will be used for hardware support.

Preliminary experiences with the **dt** and **stub** tools has been very positive.

4.3 Summary

This paper has presented a general method of representing and manipulating software design information. Major aspects of the work include

- the use of textual tags in a standard module structure [5],
- the clear separation of software design information into language dependent and independent forms (Compilable Module Stubs, Template Files, Design Files, and Group Files),
- the use of stubbing and extraction tools to effectively move design information between the language independent and dependent forms,

Positive experiences with the test programs would illustrate both the practicality of reuse of software design information and the associated savings in labor.

Contributions of this work include

- a model of information content for preliminary and detailed software designs,
- a simple model of language dependent and independent representations for preliminary and detailed software designs, and
- an automated transformational process for moving design information between language dependent and independent forms.

Acknowledgements

The author would like to thank Steve Hole for providing many valuable comments on the concepts and tools described in this paper, and for implementing most of the tools in software.

References

[1] Balzer, Robert. *A 15 Year Perspective on Automatic Programming.* IEEE Trans. on Software Engineering Vol 11(11). Nov 1985. pp. 1257-1268.

[2] Basset, Paul G. *Frame Based Software Engineering.* IEEE Software Vol 4(4) July 1987. pp. 9-16.

[3] Burton, Bruce A. et al. *The Reusable Software Library.* IEEE Software Vol 4(4) July 1987. pp. 25-33.

[4] Gargaro, Anthony. and Pappas, T. L. *Reusability Issues and Ada.* IEEE Software Vol 4(4) July 1987. pp. 43-51.

[5] Jameson, Kevin W. *SEE: A Software Engineering Environment.* University of Calgary, Dept. of Computer Science, Research Report 88/316/28. 1988.

[6] Lenz, Manfred., Schmid, Hans A., and Wolf, Peter F. *Software Reuse through Building Blocks.* IEEE Software Vol 4(4), July 1987.

[7] Rich, Charles and Waters, Richard C. *Automatic Programming: Myths and Prospects.* IEEE Computer, Vol 21(8), August 1988. pp. 40-51.

Figure 1: An Annotated Calling Tree

Interface Types

+ subsystem * system dependent ¿ recursive : utility Est. Source Type Interface Filename Lines Function Description

CALLING TREE

stub (argc,argv) sc (stub.c) 30 /* stub main entry */ ceinit (cmd$_{e}nv$) : $intwf(ceinit.c)30/ * initializecommandenvironment * /parse(cmd_{e}nv)$: $intwf(parse.c)200/ * parsecommandlinearguments * /process(cmd_{e}nv)$: $intwf(process.c)100/ * processcommandenvironment * / + get_{t}pl_{d}ir()$: $intwf(get_{t}dir.c)50/ * getthetemplatedirectory * / : query_{u}ser()sc(query.c)60/ *$ $promptandgetastringreponse * / +$ $load_{h}drs() : intwf(ld_{h}drs.c)50/ * loadthevalidseeheaders * / + load_{t}ype_{t}rans() :$ $intwf(ld_{t}ypes.c)50/ * loaddesignfiledatatypetranslations * / + parse_{g}roup() :$ $intwf(pars_{g}rp.c)100/ * */ * fopen()ec(system)50/ * openafilefori/o * / +$ $parse_{d}esign() : intwf(pars_{d}es.c)100/ * parseadesignfilemoduledef * / +$ $gen_{s}tub() :$ $intwf(gen_{s}tub.c)100/ * generateamodulestub * / * fclose()sc(system)50/ *$ $closeafile * / : destroy_{l}ist()sc(list_{a}dt.c)50/ * destroyalist *$ $/cleanup(cmd_{e}nv)sc(cleanup.c)50/ * cleanupthecommandenvironment * /$

... subsystems omitted to reduce publication size

SUBSYSTEM load$_{h}drs$

+ load$_{h}drs()$: $intwf(ld_{h}drs.c)50/ * loadthevalidseeheaders * / :$ $file_{e}xists()ec(fi_{e}xists.c)20/ * checkforfileexistence *$ $/ * fopen()ec(system)50/ * openafilefori/o * / : new_{l}ist()ec(list_{a}dt.c)50/ *$ $createandinitializeanewlist * / * fgets()ec(system)50/ * readalinefromafile *$ $/ : append_{c}ell()ec(list_{a}dt.c)50/ * appendelementinanewcelltolist * /$

UTILITIES

: append$_{c}ell()ec(list_{a}dt.c)50/ * appendelementinanewcelltolist * / :$ $destroy_{l}ist()sc(list_{a}dt.c)50/ *$ $destroyalist * / : file_{e}xists()ec(fi_{e}xists.c)20/ * checkforfileexistence * / :$ $get_{s}ee_{c}f() : intwf(get_{s}cf.c)40/ * getaSEEcontrolfile * / : get_{t}pl_{f}ile() :$ $intwf(get_{t}plf.c)50/ * getapathnametoatemplatefile * /$ Figure 2: A Template

File

– The first part of a template file maps symbolic data types – into the actual data types of the language. – The – the identifier contained in the design file. – SINT8 SINT16 STRING32 – ... – – The second part of a template file contains language specific – templates for symbolic entities in the design file. – (* MODULE function ¡logical$_{m}odule_{n}ame > *)function < internal_{n}ame >$ $(* < shortfunctiondescription > *)($

(* FUNCTION END FUNCTION *)

(* NOTES END NOTES *)

(* PARAMETERS *)) : ¡return type¿; (* ¡return description¿ *) (* END PARAMETERS *)

(* DATA DEPENDENCIES *) (* END DATA DEPENDENCIES *)

(* EXTERNAL DEPENDENCIES *) (* END EXTERNAL DEPENDENCIES *)

... other headers omitted to reduce publication size

var (* MISCELLANEOUS *) (* END MISCELLANEOUS *)

begin

(* ALGORITHM *) (* END ALGORITHM *)

end; (* END$_{M}ODULEfunction < logical_{m}odule_{n}ame > *)$ Figure 3: A

Design File for swapmax

MODULE function swapmax

FUNCTION – swaps two args and returns max The swapmax function will swap its two arguments and return the maximum of the two. END FUNCTION

NOTES Local usage notes go here. END NOTES

PARAMETERS arg1 SINT16 – u: the first integer arg2 SINT16 – u: the second integer return SINT16 – o: the max of the two END PARAMETERS

MISCELLANEOUS temp SINT16 – temp 16bit integer variable END MISCELLANEOUS

ALGORITHM : set temp variable to first argument : set first argument to second argument : set second argument to temp variable : if first arg is ¿= second arg : - return the first arg : else : - return the second arg END ALGORITHM

END$_M ODULEfunctionswapmax$ Figure 4: A Group File for swapmax

– This file: – – * Binds logical module names to internal names – * Groups logical module names into logical groups – * Binds logical groups to group types and – to physical filenames – – Syntax for module definition part: – – module logical$_n ameinternal_n amephysicalfilename - - - - - - - - Syntaxforgroupdefinitionpart : - - - - group < logicalgname >< gtype >< grouprootfilename > -- < Namesofconceptualmodulesinthegroup... > --endgroup--moduleswapmaxinternal_s wap--bindlogicaltointernalname$

group swapgroup function swapfile swapmax – only need logical name here end group Figure 5: The Derived Compilable Stub swapfile.pas

(* MODULE function swapmax *)

FUNCTION internal$_s wap(*swapstwoargsandreturnsmax*)($

(* FUNCTION

The swapmax function will swap its two arguments and return the maximum of the two. END FUNCTION *)

(* PARAMETERS *) arg1 : integer; (* u: the first integer *) arg2 : integer (* u: the second integer *)) : integer; (* o: the max of the two *) (* END PARAMETERS *)

... other headers omitted to reduce publication size var (* MISCELLANEOUS *) temp : integer; (* temp 16bit integer variable *) (* END MISCELLANEOUS *)

begin

(* ALGORITHM *) (* : set temp variable to first argument *) (* : set first argument to second argument *) (* : set second argument to temp variable *) (* : if first arg is ¿= second arg *) (* : - return the first arg *) (* : else *) (* : - return the second arg *) (* END ALGORITHM *)

end; (* END$_M ODULEfunctionswapmax*)$