Module Specifications as Program Family Generators

Grady H. Campbell, Jr.

A fundamental precept of the SCR design method is `localization of change' into information–hiding modules. A module specification describes the common properties of a family of valid implementations. The need for alternative implementations arises because of changing or uncertain needs as well as alternate uses for a module. A set of valid alternative implementations constitute a family because they are indistinguishable relative to the corresponding module specification. The primary differences among the members of this family are indicated by the module's “secrets", a set of design decisions that need to be easy to change or, equivalently, must be resolved differently in order to support different needs. If these design decisions are formally defined, alternative implementations can be made mechanically derivable. The model for this formalization is programming languages, applied to the limited domain of programs as structured artifacts.

The audiences for a module specification are the implementor of the module and the implementors of client modules. The purpose of module specifications is to allow the implementation of a module and of its clients to occur independently. Because client module implementations depend on the interfaces of modules they reference, changes to common properties are expensive and often difficult to make. However, any member of a module's family of implementations is usable in principle as long as the contract implied by the module specification is satisfied.

Normally, module specifications are viewed as a minimal constraint on an acceptable implementation; the result is a (single) implementation that may be changed later. However, it is often actually easier to create multiple implementations than it is to modify one implementation to create another. The reason for this is the difficulty of discovering and accounting for all of the implications of previous design decisions that are changing. When the implementor has an explicit means of expressing each design decision and its implications in, and yet apart from, a specific implementation, the result can be an explicit definition of a family of implementations. The value of this is the leverage that comes from recognizing that common properties described by a module specification imply similarities among alternative implementations and that differences arise as implications of design decisions.

Although the concept of program families motivated the notion of module specifications (in Parnas' paper on program families), subsequent work in this area has focused almost exclusively on how to express the common properties of a family, either formally or at least precisely. Strategies for multiple or alternative implementations have been limited to multiple physically–distinct implementations, simple macro substitution, and conventionally–built special–purpose program generators. The first two strategies are too weak and the third requires substantial effort. An alternative is a general–purpose notation for formalizing the variations that distinguish among the instances of each particular program family. Such a notation is a simple generalization of the concepts of macroprocessors and syntax–directed editors. However, it can provide the power of a special–purpose program generator with the transparency and predictability of a program editor.

Such a notation, and its associated translator, was a key mechanism in both the development and the operation of the Spectrum application generation environment. A derivative notation and

translator supports the Synthesis approach to reuse which advocates the development and use of adaptable modules, documentation, and test scenarios. Future work will relate this idea to the concepts of object–oriented and transformational programming as representations of program families and as mechanisms for reuse and automatic programming.

(Background reading: “On the Design and Development of Program Families" by D. Parnas in *IEEE Transactions on Software Engineering*, March 1976)

Virginia
CENTER of
EXCELLENCE

for Software Reuse and Technology Transfer

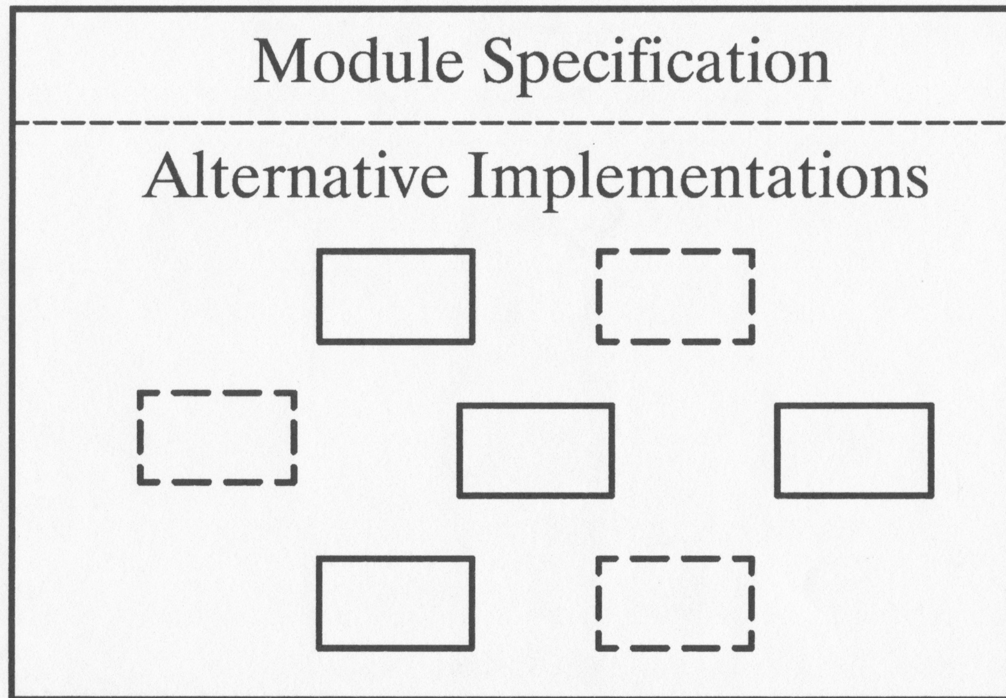
Module Families as Module Generators

September 9, 1993

Grady H. Campbell, Jr.

This material is based in part upon work sponsored by the Advanced Research Projects Agency under Grant # MDA972–92–J–1018. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

A Simple Module Family



Module family: An aggregation of program families

Module specification:

- Module interface ⇒ common properties
- Hidden design decisions (secrets) ⇒ variable properties

Virginia
CENTER of
EXCELLENCE
for Software Reuse and Technology Transfer

Proposal

Secrets should be formalized as variations of a module family, making it constructable in the form of a module generator.

- Design decisions that are represented as variations can be deferred or easily changed.
- Variations are a precise statement of requirements for creating alternative implementations.
- When implemented by a module generator (i.e., metaprogram), variations enable reuse or rapid replacement of alternative implementations
- Variations provide client implementors with specific, meaningful decisions as simplified criteria for choosing among alternative implementations to use.

A Simple Example

Abstraction: Stack of integers

Interface:

- PUSH ...
- POP ...
- TOP ...

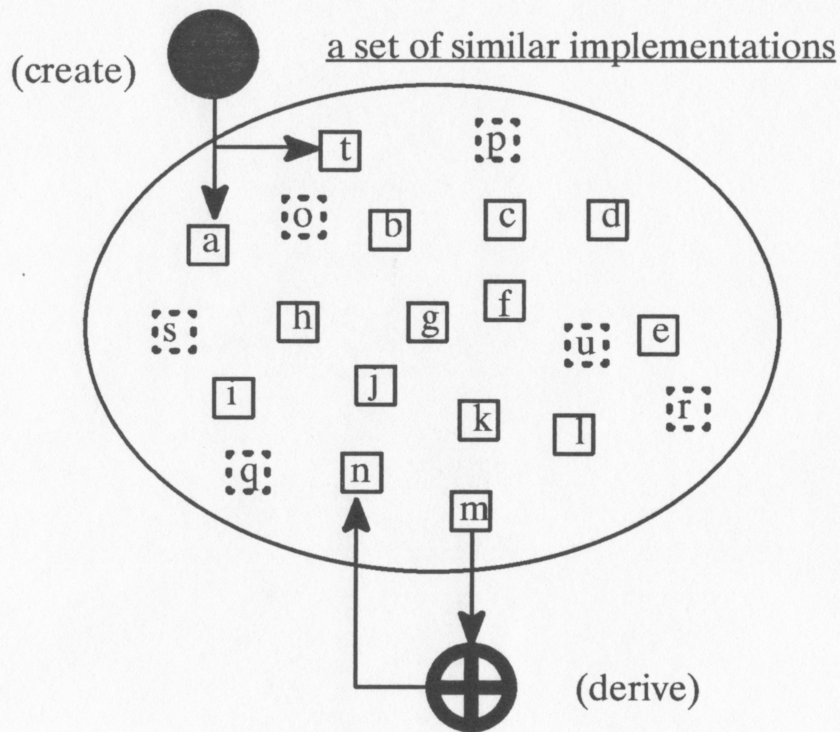
Secrets:

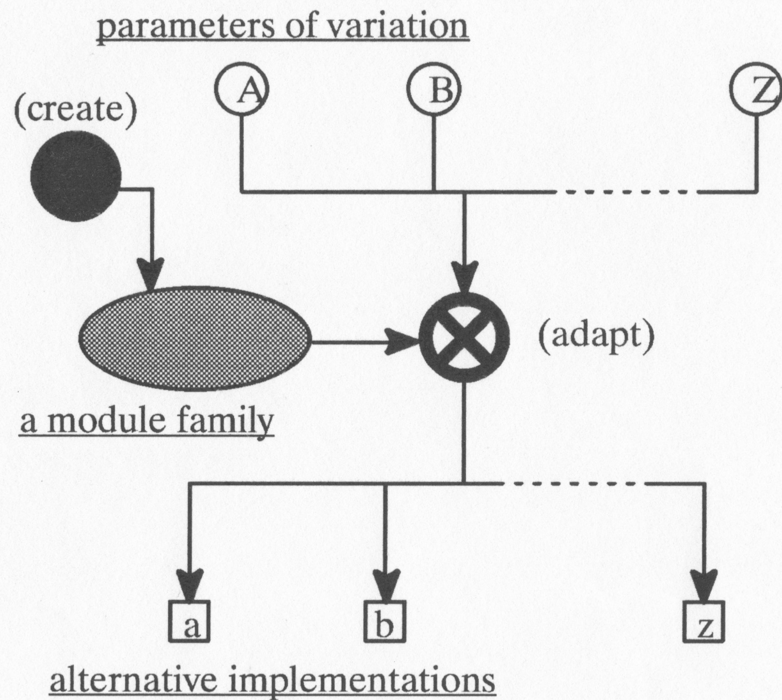
- implementation as array or linked list
  ⇒ at least 2 alternative implementations
- storage capacity if implemented as an array

⇒ Variation:
(array?:bound, list?) where 'bound' is a positive integer

Virginia
CENTER of
EXCELLENCE
for Software Reuse and Technology Transfer

Two Views of a Module Family





Virginia CENTER of EXCELLENCE for Software Reuse and Technology Transfer

Technologies for Implementing Module Families

- Branching on runtime parameter values
- Physically distinct implementations
- Object-oriented language mechanisms (e.g., subclassing)
- Ada generics; C++ templates
- Programming language or word processor macros (compile-time or link-time tailoring)
- Special-purpose generator (conventionally coded)
- Metaprogramming (precompile-time tailoring)
  - Descriptive
  - Prescriptive

Virginia CENTER of EXCELLENCE
for Software Reuse and Technology Transfer

A Simple Metaprogramming Example (1)

```
{ ^ program (stack,
    (name:target, types:list of (type:target),
    size:(unbounded?, bounded?:target) ))}
{ ^ prog_impl (stack, (
   {package {name}_stack is}
     ^ forall (type, types, (
        {type stack is record }
       ^ select (
        ^ defined (size.unbounded) -> (
             {elements: access stack_cell;} )
        ^ defined (size.bounded) -> (
           {elements:array (1..{size.bounded})
              of item; top:0..{size.bounded};})
            )
       . . .
       ))
   ^ buffer_ops ((stack), types)  -{ PUSH, POP, TOP}
   ))}
```

Virginia
CENTER of
EXCELLENCE
for Software Reuse and Technology Transfer

Metaprogramming Concepts

Control constructs:

- Substitution
- Sequencing
- Selection
- Repetition
- Definition
- Instantiation

Data types:

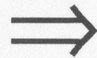
- Target-valued
- Structure-valued
- List-valued

```
{^program (stack,
    (name:target, types:list of (type:target),
      size:(unbounded?, bounded?:target) ))}
{^prog_impl (stack, (
  {package {name}_stack is}
  ^forall (type, types, (
    {type stack is record }
    ^select (
      ^defined (size.unbounded) -> (
        {elements: access stack_cell;} )
      ^defined (size.bounded) -> (
        {elements:array (1..{size.bounded})
          of item; top:0..{size.bounded};})
      )
    ...))
  ^buffer_ops ((stack), types) -{PUSH, etc}
  ))}
```

Virginia
CENTER of
EXCELLENCE
for Software Reuse and Technology Transfer

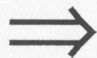
A Simple Metaprogramming Example (2)

```
{ ^ buffers.stack (active_processes, ({process}), (unbounded))}
```



```
package active_processes_stack is
   type stack is record
      elements: access stack_cell;
      . . .
   . . .
```

```
{ ^ buffers.stack (active_processes, ({process}), (bounded:{32}))}
```



```
package active_processes_stack is
    type stack is record
      elements:array (1..32) of item;
       top:0..32;
       . . .
   . . .
```

Virginia
CENTER of
EXCELLENCE
for Software Reuse and Technology Transfer

Experience with Module Families

- Spectrum
  - ~30 module families (TRF)
  - used to create Spectrum and several applications
- Synthesis
  - Rockwell (WordPerfect, TRF2)
  - Boeing (Awk)

Generalized Module Families: Abstraction-Based Programming

- An 'abstraction' (i.e., abstract module) represents a family of modules (the potential for variation in both specification and implementation).
- Variabilities in a module family (i.e., requirement and engineering decisions) imply parameterization of the abstraction.
- System-generation-time application of decisions (e.g., from an Application Model) to an abstract module produces a tailored instance.

Metaprogramming supports defining and using abstract modules.

Research Issues

- Verifying module families
  - instance-level (e.g., extrapolation from statistically representative coverage of variation sampling)
  - family-level (e.g., parameters of variation are akin to free variables in assertions)
- Notations for representing variations in non-textual forms
  - distinguished graphical forms
  - color coding
  - 2-layer (meta and instance level) forms
- Interactive tools for metaprogramming (e.g., editor/instance-viewer/debugger extensions)