NCC9-16

Designing Software for Reuse

JOHNSON
GRANT
IN-61-CR
332286
P.81

Will Tracz

IBM FSD
MD 0210
Owego, NY 13827

OWEGO@IBM.COM
(607) 751-2169

11/7/90

(NASA-CR-187917) DESIGN SOFTWARE FOR REUSE (Houston Univ.) 81 p CSCL 09B
N91-22720
Unclas
G3/61 0332286

SEPEC

Software Engineering Professional Education Center
University of Houston-Clear Lake
2700 Bay Area Blvd., Box 258
Houston, Texas 77058

SOFTWARE ENGINEERING 90
RICIS SYMPOSIUM

Designing Software for Reuse

Will Tracz

IBM FSD
MD 0210
Owego, NY 13827

OWEGO@IBM.COM
(607) 751 – 2169

11/7/90

Designing Software for Reuse

Outline

- Terminology
- Mindset – Maxims
- The Big Picture
- The 3 – C Model
- Science of Programming
- Interface Design Example
- Modularization Example
- Reuse and Implementation Guidelines

Software Reuse
Definitions

- *Use*
- *Reuse*
- *Useability*
- *Reusability*

Software Reuse
Definitions

- *Software Reuse*
- *Software Salvaging*
- *Carry – over Code*
- *Reusable Software*
- *Software Reusability*

Software Reuse

Definitions (Webster)

- **Use:** the act of employing something.
- **Reuse:** further or *repeated* use.
- **Useability:** having utility.
- **Reusability:** a property that supports reuse.

Definition #1

Software Reuse

Using **existing software**

instead of

writing **new software.**

(a broad definition)

Definition #2

Software Reuse

Using existing software

- **across time** – *maintenance*
- **across environments** – *porting*
- **across applications** – *adaptability*

Definition

Software Salvaging

The process of locating, extracting and possible modifying software from an ***existing application*** for use in a ***new application.***

(new application)

Definition

Carry – over Code

Code that is **kept (reused)** from one **version** of an application to another.

(same application)

Definition

Reusable Software

Software that was **designed to be reused.**

(new application)

Definition

Software Reusability

The degree to which software can be reused for **different applications.**

- **Vertical:** within one application domain.
- **Horizontal:** across application domains.

Software Reuse

Terminology

- Black Box vs Clear Box
- Primitive vs Composite
- Simple vs Tailorable
- Template – Skeleton – Frame
- Generic – Macro

Reuse is ..

Something **you** do all the time.

Not something new.

Not something you always **plan** on doing.

Not something you always **plan** on doing **again.**

Not something **we** do all the time.

Definitions

Taxonomy

1. **Unplanned Reuse**
   - Re – hosting
   - Maintenance
   - Salvaging
2. **Planned Reuse**
   - Portability
   - Adaptability
   - Modularity

Software Reuse
Current Approaches

1. Passive – *Composition Technology*
2. Active – *Generation Technology*

A Framework for Reusability Technologies

FEATURES	APPROACHES TO REUSABILITY				
COMPONENT REUSED	BUILDING BLOCKS		PATTERNS		
NATURE OF COMPONENT	ATOMIC AND IMMUTABLE PASSIVE		DIFFUSE AND MALLEABLE ACTIVE		
PRINCIPLE OF REUSE	COMPOSITION		GENERATION		
EMPHASIS	APPLICATION COMPONENT LIBRARIES	ORGANIZATION & COMPOSITION PRINCIPLES	LANGUAGE BASED GENERATORS	APPLICATION GENERATORS	TRANSFORMATION SYSTEMS
TYPICAL SYSTEMS	- LIBRARIES OF SUBROUTINES	- OBJ ORIENTED - PIPE ARCHS	- VHLLs - POLs	- CRT FMTRS - FILE MGMT	- LANGUAGE TRANSFORMERS

Passive Approaches to Reuse
Composition Technology

- Building Blocks
- Subroutine Libraries
- Objects
- Classes
- Modules
- Reusable Components
- Software Repositories/Bases
- **Megaprogramming**

Active Approaches to Reuse

Generation Technology

- Macros
- Generics
- Pre – Processors
- Application Generators
- 4th Generation Languages
- Parameterized Programming
- Frame – Based Programming
- Generic Architectures/Domain Models
- Constructors (Expert Systems)

Formal Approaches to Reuse

Theory

- Type Theory
- Lambda Calculus
- Conceptual Model for Reusable SW Components:
  - Separate Concept from Context.
  - Separate Concept from Content.
  - Isolate change (context) via parameters.

Software Reuse Maxims

A Perspective on Software Reuse

- Motivation
- Inspiration
- Education

Software Reuse Maxims

Golden Rules of Reusability

Before you can reuse something, you need to

1. *find it,*
2. *know what it does, and*
3. *know how to reuse it.*

Software Reuse Maxims

Rules of Three

1. Before you can develop reusable software you need to have used it ***three*** times.

2. Before you can reap the benefits of reuse, you need reuse it at least ***three*** times.

Software Reuse Maxims

Software Reuse will become the

Expert Systems

of the 1990's.

Software Reuse Maxims

Software Reuse is like a savings account

before you can collect any interest,

you need to make a deposit, and

the larger the deposit,

the larger the dividend.

The Reuse Mindset

Problem: design and implement a Stack

FORTRAN Mindset: an array

PASCAL Mindset: a linked list

Basic Ada Mindset: a package

Experienced Ada Mindset: a generic package

Advanced Ada Mindset: a family of generic packages

Software Reuse Maxims

Why is there never enough money

to *do the job right*

BUT

Always enough to *do it over?*

Software Reuse Maxims

Software Reuse:

The Search for Elegance.

Software Reuse Maxims

What sets reusable software *apart*

is

How it is *put together*.

Software Reuse Maxims

When your *object is*

Reusable Software

you need a *methodology*

to support it.

Software Reuse Maxims

When you design your software

Top Down

but implement your software

Bottom Up

sometimes it doesn't meet in the middle.

Software Reuse Maxims

For ***instance***

Reusable Software Engineers

inherently do it with

class.

Software Reuse Maxims

Reusable Software:

it's the type of thing that

makes it most useful.

Evolution of Types

- Mathematical
- Machine Intrinsic
- Built into language
- User Defined (Abstract) Data Types
- Parameterized Types
- Polymorphic Types
- Type Hierarchies
- Derived Types/Subtypes

Software Reuse Maxims

Software Reuse

is the best way to

Reuse Software

again.

Software Reuse Maxims

Reusable Software

has many arguments;

Not Reusable Software

may have too many or too few.

Software Reuse Maxims

It's not easy to make a

good CASE

for

Software Reuse

Software Reuse Maxims

Picture this pipe dream:

Today's Menu: *Reuse — — Made to order*

Today's Special: *Macroni Shell Script Delight*

Software Reuse Maxims

Have Template

Will Trace!

Software Reuse Maxims

Ask not what you can do

for your software

but what your software can do

for you.

Software Reuse Corollary

What is one question that is never answered

NO

more than once in a Japanese Software Factory?

Does a part exist that does this function?

Software Reuse Maxims

The most important quality of

Reusable Software

is that

it is *quality software.*

A Quality Argument

Given a program made up of **n** components

What is the probability (P) that it is correct?

Assume: the probability each component is correct is 95%

If **n** = 10, P =

If **n** = 100, P =

Software Reuse Maxims

Software Reuse,

like **Quality,**

is ***free.***

Software Reuse Maxims

You can make the difference between

Reusable Software

and

Reused Software

Software Reuse Maxims

It's time to move from

Reusable Software

Techniques and Mythodology

to

Technology and Methodology

Software Reuse Maxims

Ad Hack Reuse:

Business as

Re – usual

Software Reuse Maxims

Software Reuse

is a good example of

Software Engineering **discipline.**

The Big Picture

The Programming Process (IBM)

1. Requirements
2. Design
3. Implementation
4. Test
5. Package and Validate
6. Availability

The Big Picture

DOD – STD – 2167

1. Requirements Analysis
2. Preliminary Design
3. Detailed Design
4. Coding and Unit Test
5. System Integration Test
6. Production and Deployment

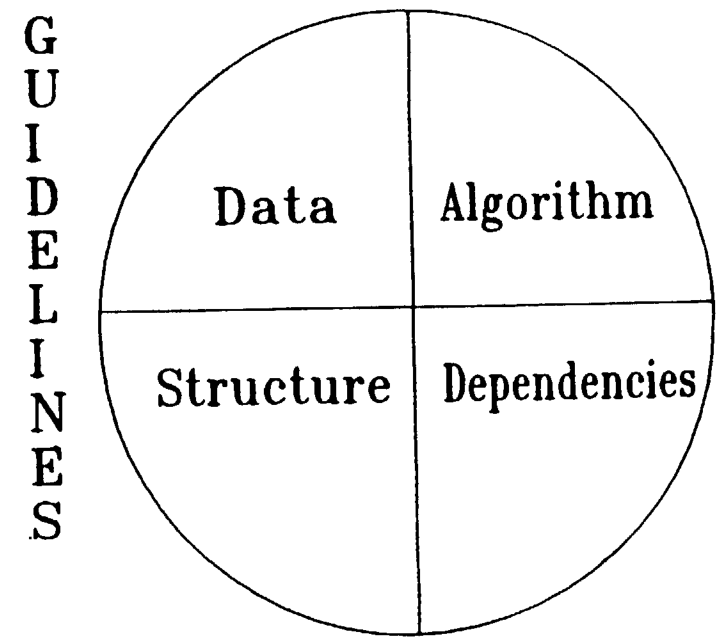
The Big Picture

Software Reuse Technology

1. Domain Analysis
2. Data Encapsulation and Information Hiding
3. Application Generators, 4GLs and gIBIS
4. Object – Oriented Programming Languages
5. Formal/Rigorous Verification
6. Promotion and Sales

The Big Picture

Programming For Reuse



The Big Picture
Programming Taxonomy

- *exploratory*
- *by difference*
- *by analogy*
- *by contracting*
- *by subcontracting*

- for **Reuse**
- with **Reuse**

- in the **Old**
- in the **New**

The Big Picture
Programming Taxonomy

- in the small
- in the large
- at large
- with the large

- for the many by the many
- for the many by the few
- for the few by the many
- for the few by the few

Conceptual Model

Reusable Software Components

- Context
- Concepts
- Content
  - Context
  - Concepts
  - Content

Conceptual Model

Context

- **"Language shapes thought"**
  - Inheritance
  - Genericity/Parameterization
  - Importation
- **Binding time**
  - Compile time
  - Load/Bind time
  - Run Time

Conceptual Model

Concepts

- **Concept:** – *What*
- **Content:** – *How*
- **Context:**

  1. **Conceptual** – *relationship*
  2. **Operational** – *with/to what*
  3. **Implementation** – *trade – offs*

Context: what is needed to complete the definition of a concept or content within an environment. (*Latour*)

Software Components

Formal Foundations

- **Horizontal Structure**

  1. type inheritance
  2. code inheritance

- **Vertical Structure**

  – implementation dependencies

  – virtual interfaces

- **Generic Structure**

  – variations/adaptations

Conceptual Model

Example

- **Concept:** Stack
  - ***Operational Context:*** Element/Type
  - ***Conceptual Context:*** Deque
  - ***Implementation Context:*** Sequence

Conceptual Model

Example

- Stack **Implementation**

  1. Inherit Deque
  2. Use an array
  3. Use a linked – list
     - memory management
     - no memory management
     - concurrent access

Megaprogramming Example

Stack – > Deque

```
make Deque [ Triv ] is
    Stack [ Triv ]
        * ( rename ( Push = > Push_Right )
                ( Pop = > Pop_Right )
                ( Stack = > Deque )
        * ( add Push_Left, Push_Right )
end;
```


Hyperprogramming Example

Make with View

```
make Integer_Set is
    LIL_Set [ Integer_View ]
end;

view Integer_View :: Triv = > Standard is
    types (Element = > Integer);
end;
```

Megaprogramming Example

Make with Vertical Composition

```
make Short_Stack is
      LIL_Stack
            - - horizontal composition
      needs (List_Theory = > List_Array)
            - - vertical composition
end;
```


LILEANNA Example

Package Expressions

```
make New_Ada_Logic_Interface is
    Identifier_Package +
    Clause_Package*(hide Copy) +
    Substitution_Package +
    DataBase_Package +
    Query_Package*(add function Query_Fail (C: Clause;
                                      L: List_Of_Clauses)
                                          return Boolean)
                 *(rename ( Query_Answer => Query_Results ))
end;
```

The Science of Programming

David Gries

Springer – Verlag – 1981

- Propositions and Predicate Calculus
- Programming Principles/Strategies
  - *Developing the proof along with the program*
  - Precondition and Postconditions
  - What to put in formalism or put in English

Programming Principle #1

A Program and its proof should be developed hand – in – hand, with the *proof* usually leading the way.

- It's too hard to prove an existing program correct.
- Need balance between Formality and Common Sense.
  - Formality alone = > incomprehensible detail.
  - Common sense alone = > allows too many errors.

Programming Principle #2

Use theory to provide insight use common sense and intuition where it is suitable, but fall back on the formal theory for support when difficulties and complexities arise.

- Proof versus Test Case Analysis
- Test cases don't always give insight
- ***Don't Program from Example***

Programming Principle #3

Know the properties of the objects that are to be manipulated by the program.

- Data Refinement
- Data Encapsulation

Programming Principle #4

Never dismiss as obvious any fundamental principle, for it is only through *conscious application* of such principles that success will be achieved.

- Recognize a Principle /= Applying a Principle
- QWERTY style of programming
- One programs into a language, not in it.

Programming Principle #5

Programming is a *goal − oriented* activity.

- Insight from postcondition
- Abstraction = Simplification/Layers
- Precondition = Interface check
- Prove/Work *Backward*

Programming Principle #6,#7

Before attempting to solve a problem, make absolutely sure you know what the problem is.

Before developing a program, make precise and refine the pre – and postconditions.

- Specification = What a program is to do.
- Abstraction = Simplification of specification
- Non – Determinism allows greater freedom in design

Programming Principle #8

All other things being equal, make the guards of an alternative command as strong as possible, so that some errors will cause abortion.

- Inline test = *Assertion*
- Test/document if what you think is valid holds.

Programming Principle #9

All other things being equal, make the guards of a loop as weak as possible, so that an error may cause an infinite loop.

- It takes 3 runs to debug a loop
  1. Once too many
  2. Once too few
  3. Just right
- Establish guards (end points)
- Establish loop invariants

Programming Principle #10

Introduce a variable only when there is a good reason for doing so.

- User optimization?
- What are compilers good at?
- Abstraction

Programming Principle #11

Put suitable bounds on each variable introduced.

- Abstract data type
- Static/Dynamic checking

Programming Principle #12

Introduce a name to denote a value that is to be determined.

- Define ***basic concepts*** for notation
- Avoid over – specifying
- Methodology = Top – down and Bottom – up

Programming Principle #13

The more guarded commands and the weaker their guards, the easier it may be to develop a correct program.

- Dual paradigm approach
- Develop proof to gain insight into program.

Programming Principle #14

Program *into* a programming language, not *in* it.

- Procedural Refinement
- Data Refinement
- Use the data structure that matches the problem.

Programming Principle #15

Keep the number of different cases to a minimum.

- Generalize
- Look for other methods of expression.
- Postpone decisions as late as possible.

Programming Principles

Summary

Resolve ambiguities and unknowns at specification time.

Design (and test case analysis) comes before coding.

Program *into* a programming language, not *in* it.

Interface Design

Sorting Example

Requirement:

- Implement a Sort Routine

Sorting Example

Missing Specifications

- **Sort what kind of data?**
  - Predefined (e.g., Integer, Float, ...)
  - User Defined (e.g., record, private)
- **Sort what kind of data structure?**
  - Array
  - Linked List
  - File

Does it matter?

Sorting Example
Missing Specifications

- **What are the environment constraints?**
  - OS dependencies
  - Concurrency
  - Error conventions
  - Size of data/execution speed
- **How is this going to be used?**
  - Function or Procedure call
  - Default Parameters
  - Single/multiple data types

Does it matter?

Sorting Example
It Does Matter

- **Sort what kind of data?**
  - Predefined (e.g., Integer, Float, ...)
    - *can assume availability of ":=" and "<"*
  - User Defined (e.g., record)
    - *need to know ":=" and "<" are available*

Sorting Example
It Does Matter

- **Sort what kind of data structure?**
  - Array
    - *Can use indexing to access list.*
    - *What are the indexes?*
  - Linked List
    - *need way to manipulate data structure*
      - Next
      - End
      - Length
  - File

Sorting Example
It Does Matter

- **What are the environment constraints?**
  - OS dependencies
    - *I/O*
  - Concurrency
    - *Is data shared?*
  - Error conventions
    - *Exceptions?*
    - *Error flag?*
    - *Formal generic procedure?*
    - *State of data on error exit?*
  - Size of data/execution speed
    - *Algorithm*

Sorting Example
It Does Matter

- **How is this going to be used?**
  - Function call
    - *Can't Sort in place*
    - *Could run out of storage*
  - Procedure call
    - *What should the parameter sequence be?*
    - *algorithm selection as a parameter?*
  - Defaults
    - *algorithm?*
    - *what about generic arguments?*
  - Generic
    - *how many formats in package?*
    - *what are the formal parameters?*

Sorting Example
Signatures

type D_Str **is ...** -- **Data Structure**

function Bubble_Sort (X: D_Str) **return** D_Str;

function Quick_Sort (X: D_Str) **return** D_Str;

Sorting Example

Signatures

```
type Algorithm is (Bubble_Sort, Quick_Sort);

function Sort (A: Algorithm; X: D_Str)
          return D_Str;

function Sort(X: D_Str;
          A: Algorithm: = Bubble_Sort)
          return D_Str;

function Sort (X: D_Str) return D_Str;
        - - Heuristic on size of X
```


Sorting Example

Signatures

```
procedure Bubble_Sort(X: in out D_Str);

procedure Bubble_Sort(X_In: in D_Str;
                      X_Out:  out D_Str);

procedure Bubble_Sort(X_In: in D_Str;
                      X_Out:  in out D_Str);

procedure Bubble_Sort (X_Out: out D_Str;
                   X_In: in D_Str);
```

Sorting Example

Signatures

```
procedure Sort ( X: in out D_Str;
                 A: Algorithm := Bubble_Sort );
```

Residual Control

```
procedure Set_Algorithm ( A: Algorithm);

procedure Sort ( X: in out D_Str);
```


Interface Modification

New Requirements

- Add metrics:
  - Number of comparisons
  - Number of swaps

Interface Modification

Alternatives

- Add two new parameters to each signature
- Add one new parameter to each signature (record)
- Replace Algorithm with Options record
- Add a new operation to report results.
- Add a new operations with new parameters

Interface Modification

Alternatives

- Add two new parameters to each signature
  - Have to change all calls
  - No growth potential
  - Signatures become cluttered

Interface Modification

Alternatives

- Add one new parameter to each signature (record)
  - − Have to change all calls
  - + Some growth potential
  - − What order with other options?

Interface Modification

Alternatives

- Replace Algorithm parameter with Options record
  - + Growth potential
  - − Still have to change all calls

Interface Modification

Alternatives

- Add a new operation to report results.
  - + Don't need to change any calls
  - + Good growth path
  - + Keeps interface clean.
  - − Problems in concurrent applications.

Interface Modification

Alternatives

- Add new operations with new parameters
  - + Don't need to change any calls
  - + Good growth path
  - + No problems in concurrent applications.
  - − Clutters interface.

Parameterization Conventions

Style

- No functions
- Limit number of parameters
- Operation/option tradeoff
- Residual control/option list tradeoff
- Categorize operations
  - Selectors
  - Constructors
  - Iterators
  - Control

Parameterization Conventions

Parameter Ordering

- Operands appear before options
- **in out** parameters appear before **in** parameters
- defaults for control parameters
- options organized as an aggregate (record)
- default option under functional control

Parameterization Conventions

Internal versus External Parameters

- Command line options
- Input options
- Option file
- Input option file name
- Explicit default files for options

Parameterization Conventions

Issues Not Addressed

- Documentation
- Fire walls
- Intelligent defaults
- Application generators
- Expert system guidance
- Polymorphism

Modularization:
A Case Study for Reuse in Ada

- **Example:** *deque*
  - Pop/Push/Print
- **Forms**
  - Functions
  - Abstract Data Object
  - Abstract Data Type
  - Parameterized Abstract Data Type
- **Composition Techniques**
  - black box
  - program templates
  - functional composition
  - repackaging

Interface Design

- Naming Conventions
- Use of Global Data
- Types and Number of Parameters
- Use of Functions or Procedures
- Use of Default Values
- Documentation

Functional Decomposition

Procedural Style

```
package Deque_FD_1 is
   .
   . -- declarations for Element_Type and Deque_Type
   .
   procedure Push_Front ( Value : in   Element_Type;  Deque : in out Deque_Type );
   procedure Push_Rear  ( Value : in   Element_Type;  Deque : in out Deque_Type );

   procedure Pop_Front  ( Value : out Element_Type;  Deque : in out Deque_Type );
   procedure Pop_Rear   ( Value : out Element_Type;  Deque : in out Deque_Type );

   procedure Print ( Deque : in Deque_Type );

end Deque_FD_1;
```


Functional Decomposition

Pure Functional Approach

```
package Deque_FD_2 is
   .
   . -- declarations for Element_Type and Deque_Type
   .
   function Push_Front ( Value : in   Element_Type;  Deque : in Deque_Type )
                          return Deque_Type;
   function Push_Rear  ( Value : in   Element_Type;  Deque : in Deque_Type )
                          return Deque_Type;

   function Top_Front  ( Deque : in Deque_Type ) return Deque_Type;
   function Top_Rear   ( Deque : in Deque_Type ) return Deque_Type;

   function Pop_Front  ( Deque : in Deque_Type ) return  Element_Type;
   function Pop_Rear   ( Deque : in Deque_Type ) return  Element_Type;
   procedure Print ( Deque : in Deque_Type );

end Deque_FD_2;
```

Functional Decomposition

Name/Parameter Tradeoffs

```
package Deque_FD_3 is
   .
   . -- declarations for Element_Type and Deque_Type
   .
   type Location_Type is ( Front, Rear );

   procedure Push ( Value     : in   Element_Type; Deque : in out Deque_Type;
                    Direction : in Location_Type );

   procedure Pop  ( Value     : out  Element_Type; Deque : in out Deque_Type;
                    Direction : in Location_Type );

   procedure Print ( Deque : in Deque_Type );

end Deque_FD_3;
```


Functional Decomposition

Side Effects (Operational)

```
package Deque_FD_4 is
   .
   . -- declarations for Element_Type and Deque_Type
   .
   type Location_Type is ( Front, Rear );

   procedure Put_In ( Direction : in Location_Type );
   procedure Push ( Value   : in   Element_Type;  Deque : in out Deque_Type );

   procedure Pop  ( Value   : out  Element_Type;  Deque : in out Deque_Type );

   procedure Print ( Deque : in Deque_Type );

end Deque_FD_4;

package Deque_FD_5 is
   .
   . -- declarations for Element_Type and Deque_Type
   .
   procedure Put_in_Front;
   procedure Put_in_Rear;
   .
   . -- same operations as in the previous example
   .
end Deque_FD_5;
```

Functional Decomposition

Side Effects (Global Data)

```
package  Deque_FD_6 is
   .
   . -- declarations for Element_Type and Deque_Type
   .
   type Location_Type is ( Front, Rear );
   Location : Location__Type;
   .
   . -- same operations as in the previous example
   .
end  Deque_FD_6;
```


Deque Abstract Data Object

Visible Data Representation

```
package Deque_ADO_1  is

   type Element_Type  is new  Natural;
   type Location_Type         is ( Front, Rear );

   The_Deque : array ( 1 .. 100 ) of Element_Type;

   Top,
   Bottom    : Integer range 0 .. The_Deque'Last := 0;

   procedure Push ( Value : in Element_Type; Into : in Location_Type );

   procedure Pop  ( Value   : out Element_Type;  Out_of : in Location_Type );

   procedure Print;

end Deque_ADO_1;
```

Deque Abstract Data Object
Visible Data Representation

Analysis

- *Reuse Potential:*
  -
  -
- *Abuse Potential:*
  -
- *Change Analysis:*
  -
  -

Deque Abstract Data Object
Hidden Data Representation

```
package Deque_ADO_2 is

   type Element_Type is new Natural;

   procedure Push_Front ( Value : in  Element_Type );
   procedure Push_Rear  ( Value : in  Element_Type );

   procedure Pop_Front  ( Value : out Element_Type );
   procedure Pop_Rear   ( Value : out Element_Type );

   procedure Print;

end Deque_ADO_2;
```

Deque Abstract Data Object

Hidden Data Representation

Analysis

- *Reuse Potential:*
  - —
  - —
- *Abuse Potential:*
  - —
- *Change Analysis:*
  - —
  - —

Deque Abstract Data Type

Visible Type Implementation

```
package Deque_ADT_1 is

    Deque_Size : constant := 100;

    type Element_Type       is new  Natural;
    type Deque_Index_Type  is range 1 .. Deque_Size;
    type Location_Type      is ( Front, Rear );

    type The_Deque is array ( Deque_Index_Type ) of Element_Type;

    type Deque_Type  is
      record
        Top,
        Bottom  : Deque_Index_Type;
        Empty   : Boolean := true;
        List    : The_Deque;
      end record;

    procedure Push ( Value : in Element_Type;  Onto  : in out Deque_Type;
                     Direction  : in Location_Type );

    procedure Pop  ( Value : out Element_Type;  From : in out Deque_Type;
                     Direction  : in Location_Type );

    procedure Print ( Value : in Deque_Type );

end Deque_ADT_1;
```

Deque Abstract Data Object

Visible Type Implementation

Analysis

- *Reuse Potential:*
  -
  -
- *Abuse Potential:*
  -
- *Change Analysis:*
  -
  -

Deque Abstract Data Type

Data Encapsulation: Array Implementation

```
package Deque_ADT_A is

    type Element_Type   is new  Natural;
    type Location_Type  is ( Front, Rear );

    type Deque_Type ( Deque_Size : Positive := 100 ) is  private;
     .
     . -- Same operations as in the previous example.
     .
    private

        subtype Deque_Max_Size is Positive range 1 .. 100;

        type The_Deque is array ( Deque_Max_Size range <> ) of Element_Type;

        type Deque_Type( Deque_Size : Positive := 100 ) is
          record
            Top,
            Bottom  : Positive;
            Empty   : Boolean := true;
            List    : The_Deque( 1 .. Deque_Size );
          end record;

end Deque_ADT_A;
```

Deque Abstract Data Object

Data Encapsulation: Array Implementation

Analysis

- *Reuse Potential:*
  - —
  - —
- *Abuse Potential:*
- *Change Analysis:*
  - —
  - —

Deque Abstract Data Type

Data Encapsulation: Linked List Implementation

```
package Deque_ADT_L is
    .
    . -- Same operations and type declarations as in the previous example.
    .
    private

        type Node;
        type Link_Type is access Node;

        type Deque_Type ( Deque_Size : Positive := 100 ) is
            record
                Top,
                Bottom  : Link_Type := null;
            end record;

        type Node is
            record
                Value : Element_Type;
                Previous,
                Next  : Link_Type := null;
        end record;

end Deque_ADT_L;
```

Deque Abstract Data Object

Data Encapsulation: Linked List Implementation

Analysis

- *Reuse Potential:*
  -
  -
- *Abuse Potential:*
- *Change Analysis:*
  -
  -

Deque Abstract Data Type

Data Encapsulation: Heterogeneous Elements

```
package Deque_ADT_H is

   type Natural_Type   is new  Natural;
   type Float_Type     is new  Float;

   type Element_Types  is ( Natural_Kind, Float_Kind );

   type Element_Type  ( Kind : Element_Types := Natural_Kind ) is
     record
       case Kind is
         when Natural_Kind => Natural_Value : Natural_Type;
         when Float_Kind   => Float_Value   : Float_Type;
       end case;
     end record;

   type Location_Type     is ( Front, Rear );

   type Deque_Type ( Deque_Size : Positive := 100 ) is  private;
   .
   . -- Same operations as in the previous example.
   .
   private
     .
     . -- Same type declarations as in the previous example.
     .
end Deque_ADT_H;
```

Deque Abstract Data Object

Data Encapsulation: Heterogeneous Elements

Analysis

- *Reuse Potential:*
  - —
  - —
- *Abuse Potential:*
  - —
- *Change Analysis:*
  - —
  - —

Parameterized (Generic) ADT

Generic Deque: Array Implementation

```
generic

    type Element_Type is private;
    Default_Deque_Size : Positive := 100;

    with procedure Print ( Value : Element_Type );

package Deque_GADT_A is

    type Location_Type      is ( Front, Rear );

    type Deque_Type ( Default_Deque_Size ) is  private;
    .
    . -- Same operations as in the previous example.
    .
    private
        .
        . -- Same as in section 5.1
        .
end Deque_GADT_A;
```

Parameterized (Generic) ADT
Generic Deque: Array Implementation

Analysis

- *Reuse Potential:*
  -
  -
- *Abuse Potential:*
  -
- *Change Analysis:*
  -
  -

Parameterized (Generic) ADT
Deque: Linked List Implementation

```
generic
   .
   . -- Same generic formal parameters
   .
package Deque_GADT_L is
   .
   . -- Same operations and declarations as in the previous example
   .
private
   .
   . -- Same type declarations as example 5.2
   .
end Deque_GADT_L;
```

Parameterized (Generic) ADT

Deque: Linked List Implementation

Analysis

- *Reuse Potential:*
  -
  -
- *Abuse Potential:*
  -
- *Change Analysis:*
  -
  -

Parameterized (Generic) ADT

Generic Deque: Hidden Implementation

```
generic
   .
   . -- Same generic formal parameters
   .
package Deque_GADT_X is
   .
   . -- Same operations and declarations as in the previous example
   .
private

   type Deque;
   type Deque_Pointer is access Deque;

   type Deque_Type ( Deque_Size : Positive := 100 ) is
      record
         Value  : Deque_Pointer;
      end record;

end Deque_GADT_X;
```

Parameterized (Generic) ADT

Generic Deque: Hidden Implementation

Analysis

- *Reuse Potential:*
  - —
  - —
- *Abuse Potential:*
- *Change Analysis:*

Reusability Assessment

Understandability

- selecting the proper operation
- supplying the right actual subprogram parameter
- declaring a variable of a certain type
- specifying actual generic parameters
- modifying the package specification
- modifying the package body

Reusability Assessment

Summary of Interface Styles

- Package Format
  - functional decomposition with no state
  - functional decomposition with residual control
  - abstract data object
  - abstract data type
  - parameterized (generic) ADT/ADO, etc.
- retain data and control state information
- tradeoff: Operations/parameters

Summary of Interface Styles

- State data can be
  - hidden in the package body
  - protected in a private section
  - exposed in the specification
- Data Encapsulation protects integrity of data.
- Data Encapsulation limits reusability.
- Private data is useful for documentation and modification.
- Hiding state and control data requires more effort to modify but makes for a cleaner interface.

Composition

Deque – > Stack/Queue

- **Abstract Data Object – >**
  - Abstract Data Object
- **Abstract Data Type – >**
  - Abstract Data Object
  - Abstract Data Type
- **Generic Abstract Data Type – >**
  - Abstract Data Object
  - Abstract Data Type
  - Generic Abstract Data Type

Ada Reuse Mechanisms

- with clause
- rename statement
- subtype declaration
- derived type declaration

Stack ADO from Deque ADO

Using Derived Types

```
with Deque_ADO_1;
package St_01 is

    package DQ renames Deque_ADO_1;
    type    Element_Type is new  DQ.Element_Type;

    procedure Push ( Value : in Element_Type );
    procedure Pop  ( Value : out Element_Type );
    procedure Print renames DQ.Print;

end St_01;

package body St_01 is

    procedure Push  ( Value :  in Element_Type ) is
    begin
        DQ.Push ( DQ.Element_Type( Value ),  Direction => DQ.Front );
    end Push;

    procedure Pop  ( Value : out Element_Type ) is
    begin
        Pop ( Value,  Direction => DQ.Front );
    end Pop;

    --   procedure Print is taken care of in the specification

end St_01;
```


Stack ADO from Deque ADT

Using Subtypes

```
with Deque_ADO_1;
package St_1 is

    package DQ renames Deque_ADO_1;
    subtype Element_Type is DQ.Element_Type;

    procedure Push ( Value : in Element_Type );
    procedure Pop  ( Value : out Element_Type );
    procedure Print;

end St_1;

package body St_1 is

    Stack1 : DQ.Deque_Type;   -- declare the object

    procedure Push  ( Value :  in Element_Type )  is
    begin
        DQ.Push ( Value,  Stack1,  Into => DQ.Front );
    end Push;

    procedure Pop  ( Value : out Element_Type )  is
    begin
        DQ.Pop ( Value,  Stack1,  Out_Of => DQ.Front );
    end Pop;

    procedure Print is
        begin
            DQ.Print( Stack1 );
        end Print;
end St_1;
```

Queue ADT from Deque ADT

Using Subtypes

```
with Deque_FD_1;
package QADT1_DADT1 is

    package DQ renames Deque_FD_1;
    subtype Element_Type is DQ.Element_Type;
    subtype Queue_Type   is DQ.Deque_Type;

    procedure Push ( Value : in Element_Type;
                     Deque : in out Queue_Type ) renames DQ.Push_Front;

    procedure Pop  ( Value : out Element_Type;
                     Deque : in out Queue_Type ) renames DQ.Pop_Rear;


    procedure Print ( Value : in Stack_Type ) renames DQ.Print;

end QADT1_DADT1;
```


Stack ADO from Generic Deque ADT

```
package SADO1_DGADT1 is

    subtype Element_Type is Integer;

    procedure Push ( Value : in Element_Type );
    procedure Pop  ( Value : out Element_Type );
    procedure Print;

end SADO1_DGADT1;

with Deque_GADT_A;
with Text_IO; use Text_IO;
package body SADO1_DGADT1 is

    package Int_IO is new Integer_IO(Integer);
    procedure Print( N: in Integer );
    package DQ is new Deque_GADT1( Integer, 100, Print );

    Stack1 : DQ.Deque_Type; -- Declare the object.

    procedure Print( N: in Integer) is
    begin
        Put( N );
    end Print;
    .
    . -- See section 8.3.1 for implementation of Push, Pop and Print.
    .
end SADO1_DGADT1;
```

Queue ADT from Generic Deque ADT

```
with Deque_GADT1;

package QADT1_DGADT1 is

    subtype Element_Type is Integer;
    type Queue_Type is private;

    procedure Push ( Value : in Element_Type;  Onto  : in out Queue_Type );
    procedure Pop  ( Value : out Element_Type,  From  : in out Queue_Type ),
    procedure Print ( Value : in Queue_Type ),

    private
        procedure Print ( N : in Element_Type );
        package DQ is new Deque_GADT1(Element_Type, 100,  Print);

        type Stack_Type is new DQ Deque_Type;

end QADT1_DGADT1.

with Text_IO,
package body QADT1_DGADT1 is

        package Int_IO is new Text_IO Integer_IO(Integer),
        procedure Print( N : in Element_Type ) is
            begin
                Int_IO Put( N ),
            end Print,

        procedure Push ( Value : in Element_Type;  Onto : in out Queue_Type ) is
        begin

            DQ.Push ( Value,  DQ Deque_Type( Onto ),  DQ Front )
        end,

        procedure Pop  ( Value : out Element_Type,  From  : in out Queue_Type ) is
        begin
            DQ Pop ( Value,  DQ.Deque_Type( From ),  DQ Rear );
        end;

        procedure Print ( Value   in Queue_Type  ) is
        begin
            DQ.Print ( DQ Deque_Type( Value ) ),
        end;

end QADT1_DGADT1.
```

Stack Generic ADT from Generic Deque ADT

```
with Deque_GADT1;
generic

    type Element_Type is private;
    Default_Stack_Size  :  Positive  := 100;
    with procedure Print( Value  :  Element_Type ) is <>;

package SGADT1_DGADT1 is

    type Stack_Type is private;

    procedure Push ( Value : in Element_Type;  Onto  : in out Stack_Type );
    procedure Pop  ( Value : out Element_Type;  From  : in out Stack_Type );
    procedure Print ( Value : in Stack_Type );

    private
        package DQ is new Deque_GADT1( Element_Type,  Default_Stack_Size,  Print );
        type Stack_Type is new DQ.Deque_Type;

end SGADT1_DGADT1;


package body SGADT1_DGADT1 is
    .
    . -- Same implementation of Push and Print as in last example.
    . -- The implementation of Pop needs to have the last parameter
    . --     changed to DQ.Rear.
    .
end SGADT1_DGADT1;
```

Software Reuse Rules of Thumb

The Questions

1. What software *should* be made reusable?

2. *How* is software made reusable?

Software Reuse Rules of Thumb

The Answers

1. To identify software for reuse, **factor out commonality.**

2. To develop reusable software, **separate context from concept and content.**

Factoring Out Commonality

The Questions

1. What software is ***common*** among most applications?

2. What software is ***common*** within a specific application domain?

Domain Analysis

Biggerstaff's Guidelines

A good domain for reuse is one that

1. encompasses well understood abstractions,

2. has only a few data types,

3. depends on an underlying technology that is stable, and

4. has standards within the problem domain.

Software Reuse Maxim

Before software can be

reusable,

it needs to be

useful.

Separating Context

Sort Example

1. **Data:** (e.g., payroll records, student grades)
2. **Data Structure:** (e.g., linked list, an array, or in a file.
3. **Variations:** (e.g., ascending or descending order)
4. **Hardware Dependencies:**
5. **Operating System/Data Base Dependencies:**
6. **User Interface:**

Software Reuse Maxim

Before software can be

reusable,

it needs to be

useable.

Software Reuse Rules of Thumb

Corollaries

1. ***Separate the interface from the implementation.***

2. ***Isolate dependencies through virtual interfaces.***

Separate Concept from Content

Reuse Checklist

Strategy

Domain Analysis:

factoring out commonality and
factoring in generality.

Design for Reuse:

separating context from content through
modularization and parameterization.

Domain Analysis

What software should be made reusable?

- What is common among software applications?
  - Common implementation language?
  - Written for the same operating system?
  - Uses the same data – base system?
  - Has the same user interface?
  - Works on the same hardware platform?
  - Has the same functionality?

Domain Analysis

Checklist

- What is common among versions of a certain application?
  - Common modularity? Are some of each system's modules the same, or similar?
  - Written for the same operating system?
  - Uses the same data – base system?
  - Has the same user interface?
  - Works on the same hardware platform?
  - Has the same functionality?

Domain Analysis

Checklist

- What is common between **current** versions and **future** applications?
- How many future applications will be similar to current implementations?
- What kind of changes in requirements can be envisioned for future versions of existing applications?

Domain Analysis

Checklist

- What is common between **current** versions and **future** applications?
  - Written for the new operating system?
  - Uses a new data base system?
  - Has a new user interface?
  - Works on a different hardware platform?

Domain Analysis

Checklist

- Can a business case be made to justify the cost of creating a baseline?
- Can a business case be made to justify the cost of creating an application generator?
- Can these questions be answered by in – house experts?
- Is management willing to support the development, documentation, maintenance and training effort to support reuse?

Design for Reuse

(How to make software Reusable?)

- What is the best way to modularize the application for reuse?
  - Can operations be grouped that work on the same kind of data (abstract data type)?
- Can global data be eliminated or encapsulated in modules along with the operations that manipulate it (data encapsulation)?
- Can implementations be separated from interfaces (program families)?

Design for Reuse

Checklist

- Can algorithms be generalized to work on different
  - hardware,
  - operating systems,
  - I/O devices,
  - user interfaces, or
  - data structures/data bases?

Design for Reuse

Checklist

- Can virtual interfaces be defined to separate
  - hardware,
  - operating system,
  - I/O,
  - user interface, or
  - data structure/data – base dependencies?

Design for Reuse

Checklist

- What documentation is necessary to help the user
  - reuse,
  - locate,
  - understand, or
  - modify the software?

Design for Reuse

Checklist

- Can the domain of applicability of a function or module be increased through parameterization?
- Can tests be built in to assure parameters are correct on invocation?
- Can tests be built in to assure parameters are correct on instantiation?

Software Reuse Rules of Thumb

The Answers

1. To identify software for reuse, **factor out commonality.**
2. To develop reusable software, **separate context from concept and content.**

General Reuse Guidelines

Ed Berard

The following *increase* reusability:

- Following standards
- Management encouragement
- Code without language or implementation tricks
- Portable code
- Reliable code
- Functionally cohesive and loosely coupled modules
- Well defined interfaces
- Generality and Robustness
- Conceptual Integrity

Ada Reuse Guidelines

Ed Berard

Reusability is *increased* when using

- meaningful mnemonics
- attributes
- named parameters
- fully qualified names
- precise, concise comments
- subunits and separate compilation
- packages
- generics
- isolated machine dependencies
- isolated application specific dependencies

Ada Reuse Guidelines

Ed Berard

Reusability is *decreased* when using

- literal constants
- use clause
- default values for:
  - discriminants
  - record field values
  - formal parameters
- **optional language features:**
  - pragmas
  - unchecked – deallocation
  - unchecked – conversion

Ada Reuse Guidelines

Ed Berard

Reusability is *decreased* when using

- anonymous types
- pre – defined and implementation – defined types
- attention to underlying implementation
- restrictive modules
- assumptions about garbage collection