Software Reuse

- Building software from reusable components.

Objectives

- To discuss the advantages and disadvantages of software reuse
- To describe development with and for reuse
- To discuss the characteristics of generic reusable components
- To describe methods of developing portable application systems

Topics covered

- Software development with reuse
- Software development for reuse
- Generator-based reuse
- Application system portability

Reusable component types

- Application system reuse
  - The whole of an application system may be reused on a different machine. Usually referred to as program portability
- Sub-system reuse
  - Major sub-systems such as a pattern-matching system may be reused
- Modules or object reuse
  - The reusable component is a collection of functions or procedures
- Function reuse
  - The reusable component is a single function

Reuse practice

- Application system reuse
  - Widespread. It is common practice for developers of systems (e.g. Microsoft) to make their products available on several platforms
- Sub-system and module reuse
  - Practiced informally in that individual engineers reuse previous work. Little systematic reuse but increasing reuse awareness
- Function reuse
  - Common in some application domains (e.g. engineering) where domain-specific libraries of reusable functions have been established. Reuse is the principal reason why languages such as FORTRAN are still used

Four aspects of reuse

- Software development with reuse
  - Developing software given a base of reusable components
- Software development for reuse
  - How to design generic software components for reuse
- Generator-based reuse
  - Domain-specific reuse through application generation
- Application system reuse
  - How to write application systems so that they may be readily ported from one platform to another

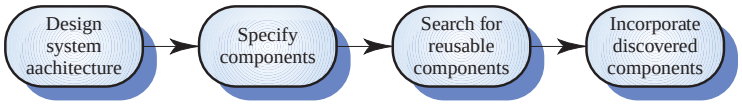
Software development with reuse

- Attempts to maximise the use of existing components
- These components may have to be adapted in a new application
- Fewer components need be specified, designed and coded
- Overall development costs should therefore be reduced

Further advantages

- System reliability is increased
- Overall risk is reduced
- Effective use can be made of specialists
- Organizational standards can be embodied in reusable components
- Software development time can be reduced

Development with reuse process



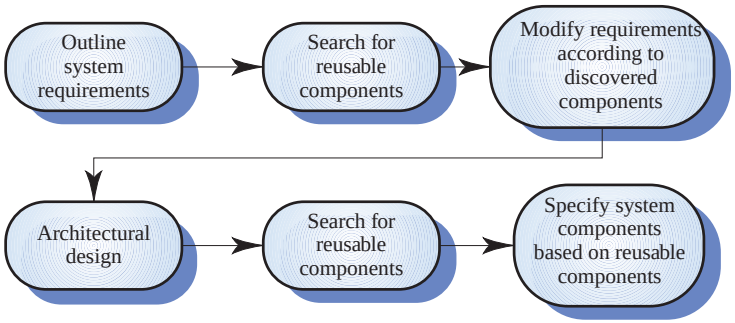
Requirements for reuse

- It must be possible to find appropriate reusable components in a component data base
- Component reusers must be able to understand components and must have confidence that they will meet their needs
- The components must have associated documentation discussing HOW they can be reused and the potential costs of reuse

Reuse-driven development

- Rather than reuse being considered after the software has been specified, the specification takes into account the existence of reusable components
- This approach is commonplace in the design of electronic, electrical and mechanical systems.
- If adopted for software, should significantly increase the proportion of components reused

Reuse-driven development



Reuse problems

- Difficult to quantift costs and benefits of development with reuse
- CASE toolsets do not support development with reuse. They cannot be integrated with a component library systems
- Some software engineers prefer to rewrite rather than reuse components
- Current techniques for component classification, cataloging and retrieval are immature. The cost of finding suitable components is high

Software development for reuse

- Software components are not automatically reusable. They must be modified to make them usable across a range of applications
- Software development for reuse is a development process which takes existing components and aims to generalise and document them for reuse across a range of applications

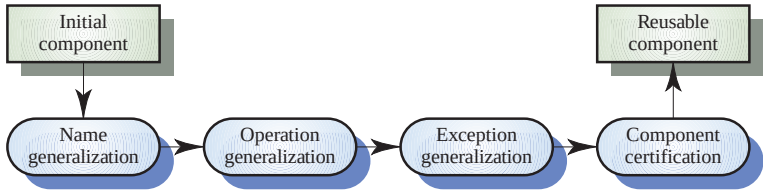
Development for reuse

- The development cost of reusable components is higher than the cost of specific equivalents. This extra reusability enhancement cost should be an organization rather than a project cost
- Generic components may be less space-efficient and may have longer execution times than their specific equivalents

Reusability enhancement

- Name generalisation
  - Names in a component may be modified so that they are not a direct reflection of a specific application entity
- Operation generalisation
  - Operations may be added to provide extra functionality and application specific operations may be removed
- Exception generalisation
  - Application specific exceptions are removed and exception management added to increase the robustness of the component
- Component certification
  - Component is certified as reusable

Reusability enhancement process



Domain-specific reuse

- Components can mostly be reused in the application domain for which they were originally developed as they reflect domain concepts and relationships
- Domain analysis is concerned with studying domains to discover their elementary characteristics
- With this knowledge, components can be generalised for reuse in that domain

Domain-specific reuse

- Reusable components should encapsulate a domain abstraction
- In order to be reusable, an abstraction has to be complete
- The abstraction must be parameterised (at least to some extent) to allow for instantiation in different systems with specific requirements

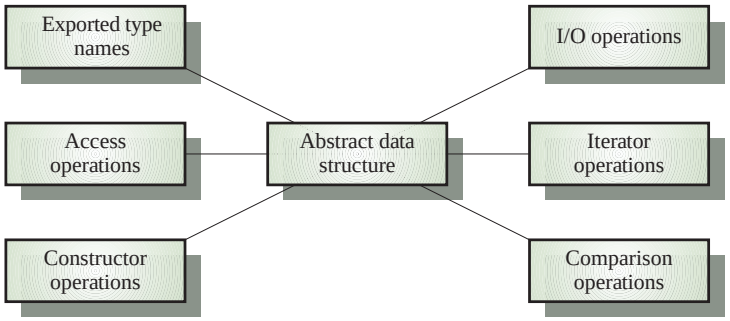
The abstract data structures domain

- Well-understood application domain
- Important as a foundation for many types of software system
- The requirements for reusable abstract data structures have been published by several authors (e.g. Booch)
- A classification scheme for such components has been invented

ADS generalisation

- Involves adding operations to a component to ensure domain coverage
- Operations required include
  - Access operations
  - Constructor operations
  - I/O operations
  - Comparison operations
  - Iterator operations, if the component is a collection of components

Model of a reusable ADS



Reuse guidelines

- Implement data structures as generic packages
- Provide operations to create and assign instances
- Provide a mechanism to indicate whether or not operations have been successful
- Minimise the amount of information defined in the component specification

Reuse guidelines

- Implement operations which can fail as procedures and return an error indicator as an out parameter.
- Provide an equality operation to compare structures.
- Provide an iterator which allows each element ina collection to be visited efficiently without modification to that element

Reusable component example

- Linked list of elements where each element maintains a pointer to the next element in the list
- Commonly implemented in application systems but application-specific components are rarely generic as their operations reflect specific application needs
- Linked list operations are usually independent of the type of element in the list

Linked list generic package

- See portrait slide

Access operations

```
-- true if the list has no elements
function Is_empty (L: LIST) return BOOLEAN ;
-- returns the number of elements in the list
function Size_of (L: LIST ) return NATURAL ;
--  true if a list element is the same as E
function Contains (E: ELEMENT;  L: LIST )
                    return BOOLEAN ;
-- returns the first list element
procedure Head (L: LIST; E: in out ELEMENT ;
                     Error_level: out STATUS )  ;
-- removes the first list element and returns the remaining list
procedure Tail (L: LIST;  Outlist: in out LIST ;
                    Error_level: out STATUS )  ;
```


Constructor operations

- See portrait slides

I/O procedures

```
-- print onto standard output

procedure Print_list (L: LIST; Error_level: out STATUS ) ;
procedure Write_list (F: TEXT_IO.FILE_TYPE ; L: LIST;
                                        Error_level: out STATUS ) ;
 procedure Read_list (F: TEXT_IO.FILE_TYPE ;
                        Outlist: out LIST ; Error_level: out STATUS )  ;
```


Iterator operations

```
procedure Iterator_initialise (L: LIST; Iter: in out ITERATOR;
            Error_status: in out STATUS) ;
procedure Go_next (L: LIST; Iter: in out ITERATOR;
            Error_status: in out STATUS) ;
procedure Eval (L: List; Iter: in out ITERATOR;
            Val: out ELEMENT;  Error_status: in out STATUS) ;
function At_end (L: LIST; Iter: ITERATOR) return BOOLEAN ;
```

C++ linked list component

- See portrait slides

Language-dependent reuse

- Reuse guidelines for domain abstractions are independent of the implementation language
- However, some reuse guidelines may be language independent
  - In Ada, do not pass array sizeas a parameter to reusable components which operate on arrays. Use the built-in attribute to determine the array size
  - In C++, always pass the array size as a parameter to reusable components which operate on arrays

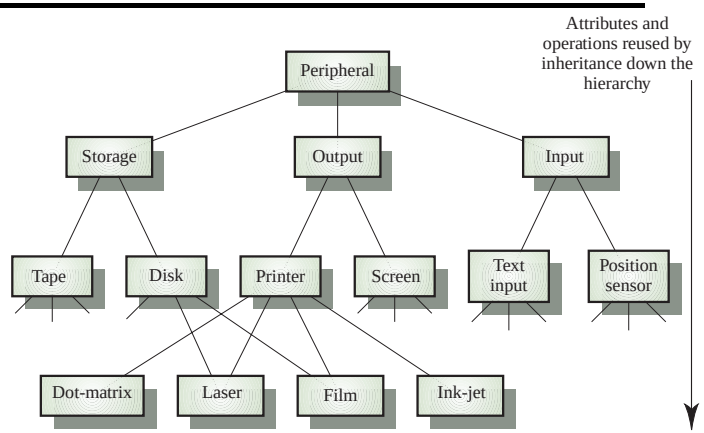
Component adaptation

- Extra functionality may have to be added to a component. When this has been added, the new component may be made available for reuse
- Unneeded functionality may be removed from a component to improve its performance or reduce its space requirements
- The implementation of some component operations may have to be modified. This suggests that the original generalisation decisions may be incorrect

Reuse and inheritance

- Objects are inherently reusable because they package state and associated operations. they can be self-contained with no external dependencies
- Inheritance means that a class inherits attributes and operations from a super-class. Essentially, these are being reused
- Multiple inheritance allows several objects to act as a base class so attributes and operations from several sources are reused

A class lattice



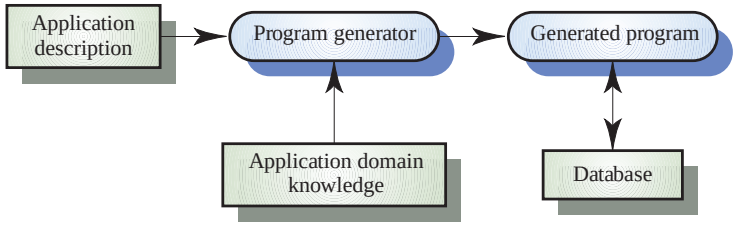
Problems with inheritance

- As component classes are developed, the inheritance lattice becomes very complex with duplications across the lattice. Regular rationalisation is required.
- To understand a component, many classes in the hierarchy may have to be examined and understood
- In many cases, it may be impossible to avoid inheriting unneeded functionality

Generator-based reuse

- Program generators involve the reuse of standard patterns and algorithms
- These are embedded in the generator and parameterised by user commands. A program is then automatically generated
- Compilers are program generators where the reusable patterns are object code fragments corresponding to high-level language commands

Reuse through program generation



Types of program generator

- Types of program generator
  - Application generators for business data processing
  - Parser and lexical analyser generators for language processing
  - Code generators in CASE tools
- Generator-based reuse is very cost-effective but its applicability is limited to a relatively small number of application domains

Application system portability

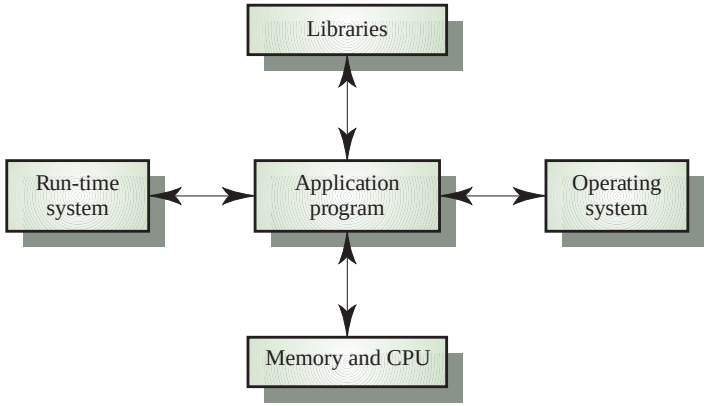
- Portability is a special case of reuse where an entire application is reused on a different platform
- The portability of a program is a measure of the amount of work required to make that program work in a new environment

Aspects of system portability

- Transportation
  - The physical movement of the program code and associated data from one environment to another

    This is a less significant problem than it used to be as electronic interchange of programs through networks avoids media incompatibility
- Adaptation
  - The changes required to make a program work in a different environment

Application program interfaces



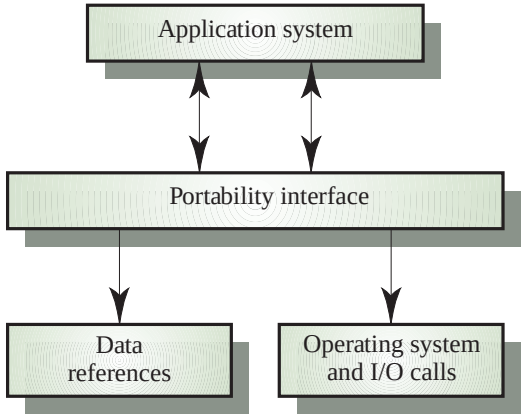
Portability dependencies

- Machine architecture dependencies
  - Dependencies on information representation and organisation
- Operating system dependencies
  - Dependencies on operating system characteristics
- Run-time system problems
  - Dependencies ona particular run-time support system
- Library problems
  - Dependencies on a specific set of libraries

Development for portability

- Isolate parts of the system which are dependent on the external program interfaces. These interfaces should be implemented as a set of abstract data types or objects
- Define a portability interface to hide machine architecture and operating system characteristics
- To port the program, only the code behind the portability interface need be rewritten

A portability interface



Machine architecture dependencies

- The program must rely on the information representation scheme supported by a particular machine architecture
- Common problems are:
  - The precision of real numbers
  - Bit ordering in number representation
- Can be tackled by the use of abstract data types. Different representations can be supported

A portable counter component

- Replace with portrait slide

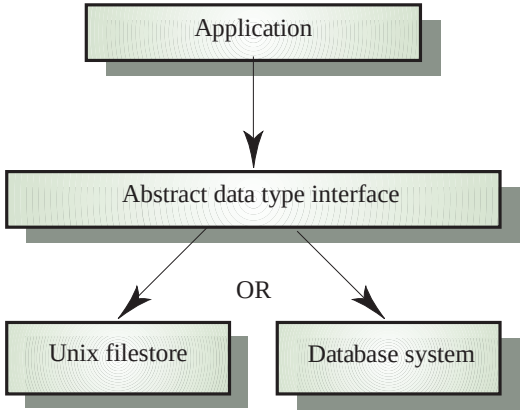
Operating system dependencies

- The program relies on the use of specific operating system calls such as facilities to support process management
- The program depends on a specific file system organisation supported by the operating system

Portable process management

- Replace with portrait slide

Portability interface implementation



Standards

- Standards are an agreement across the community which reduces the amount of variablity in software systems
- The development of standards in the 1980s means that program portability is now much simpler than before
- In principle, as standards are further developed, heterogeneous systems may be developed where parts of a program may run on completely different machines

Existing standards

- Programming language standards
  - Ada, Pascal, C, C++, FORTRAN.
- Operating system standards
  - UNIX, MS-DOS (de-facto standard), MS Windows
- Networking standards
  - TCP/IP protocols, X400, X500, Sun NFS, OSI layered model. HTML, WWW
- Window system standards
  - X-windows. Motif toolkit

Key points

- Software reuse involves using components developed in some application in a different application
- Systematic reuse can reduce costs, reduce management risk and improve software reliability
- Development with reuse must be based on a library of reusable components
- Components must be generalised for reuse

Key points

- Abstract data types and objects are encapsulations of reusable components
- Generator-based reuse depends on using standard domain-specific patterns
- Application portability is a form of reuse where an entire application is reused on a different platform
- Portablity is achieved by developing according to standards and isolating platform dependencies