Hierarchical object-oriented design

• Overview of HOOD
  – abstraction
  – polymorphism
  – encapsulation

• HOOD theory
  1. objects
  2. classes
  3. inheritance

• HOOD representation

• HOOD process

• HOOD example
Hierarchical object-oriented design

An **object** is an identifiable entity that plays a visible role in providing a service that a client can request.

- **Objects:**
  - embody an *abstraction* that is meaningful to their clients (users or other systems)
  - provide corresponding services, which can access and modify data within the object or call other objects
  - are encapsulated so that clients cannot directly access data within objects
  - perform requested operations through their services
Object orientation and design

• First OO procedure (Booch 1983):

  1. Produce a written description of a design solution
  2. Analyse this in terms of nouns and verbs
     – nouns become objects,
     – verbs provide operations associated with objects
  3. Formalise the descriptions of objects to produce a complete design

• Hierarchical Object-Oriented Design (HOOD) was developed from this by European Space Agency (ESA)

• The hierarchy described in HOOD takes two forms:

  1. *Uses*: dependence of one object on another’s services
  2. *Functional decomposition*: object split into child objects, to give functionality of the parent object
10.1 HOOD theory

- Combination of **data structures** and **functionality**
  - Object-oriented (OO) design was first used with OO programming (e.g., Smalltalk)
  - Based on concept of **objects**, **classes** and **inheritance**:
    \[ \text{OO} = \text{Objects} + \text{Classes} + \text{Inheritance} \]

**Abstraction, Polymorphism, Encapsulation, Inheritance**

1. **Objects** — *abstraction*

   - *operation* is procedure provided by an object that can access the internal data
   - *object* is an information hiding module that contains both data and operations on the data
   - object is an *instance* of a Class of objects
2. **Classes** — *modularity*

- A class is an *object template*, like abstract data type
- Consists of *encapsulated data types* and *operations*

3. **Inheritance** — *hierarchy*

- A method of sharing and *reusing* code between classes:
  - child class can adapt contents (data structure) and behaviour (operations) of parent class by *adding* operations and variables, or by *redefining* them
  - new class is based on definition of existing class; no need to copy actual code manually
  - e.g., for designing two similar modules
- parent class is the *super-class*; child is the *sub-class*
- forms an “*is a kind of*” relationship: if class C inherits from class B, then C “is a kind of” B
Reduce, recycle, re-use your computer code!

Objects can:

- have a common implementation so that they share code, making implementation more efficient
- have different implementations of generic operations to provide a consistent service — *polymorphism*
- share partial implementations (when selected elements of the object’s behaviour are to be shared)
- be identified in a request, so when a client issues a request it is clear which object should service it
- support parallel operations and time-varying behaviour of a system
Information hiding — *encapsulation*

- modules contain ‘hidden’ information and procedures to access it:
  - update data by object operation calls
  - eliminate the need for global variables
  - hide details of data structure implementation

- *Virtual interface*: makes modules like black boxes, external access to information via object’s operations

- *Data abstraction*: changes to internal structure does not affect other modules using information

- design guided by what info other objects need to see

- hidden data cannot be modified by clients (users) of the object that provides the service
Active and passive objects

- **Active** objects run concurrent tasks and have a “life” of their own:
  - contribute to the dynamic behaviour of a system
  - encapsulate one or more execution threads

- **Passive** objects act as information-hiding modules:
  - does not contribute to system dynamics
  - thread-empty passive code
  - its operations are called by an active object

- objects communicate by passing a *message* to request an operation from the receiving object:
  - message passing can be synchronous (objects are passive and inactive when message received)
  - otherwise active object can be busy and message must be queued
Language Support

- **object-based** languages (object support only): Ada and Modula2

- **class-based** languages (classes and objects): Clu

- **object-oriented** languages (classes, objects, inheritance): Smalltalk, C++, Java

Summary

Object-oriented design = Objects + Classes + Inheritance
10.2 **HOOD** representation

**HOOD** notation for an object and its components.

Example of a stack object.
(a) Example of the uses relationship between several passive objects.
(b) An active object receiving execution requests in HOOD notation.
(c) Parent-child decomposition in HOOD with implemented by links.

Hierarchical relationship between two object classes.
Flow diagram of the HOOD process.
10.3 HOOD process

- HOOD process starts from textual description of design

- Overall strategy is top-down:
  1. definition and analysis of the problem
  2. elaboration of an informal strategy
  3. formalisation of the strategy
  4. formalisation of the solution

- For each step, the HOOD process identifies:
  1. activities the designer should perform
  2. input ‘documents’ required
  3. output ‘documents’ to be generated
  4. validation procedures to be used
Step 1 — Statement of requirements

The program is to take the source code of a Pascal program as input and perform dependence tracing on it. For each procedure declared in the input Pascal code, the program will need to print out a list of the procedures it calls, from which we can then construct a Structure Chart that represents the organisation of the Pascal program.
Step 2 — Elaboration of an informal strategy

First attempt at describing the solution might be:

The program first reads through the input Pascal source file and makes a list of the procedures that are declared in that program. It then reads the input file again and this time makes a note of each call that is made to any of the procedures that were previously identified and also records the identity of the calling component. From these records it constructs a tree that shows the interdependencies of the different procedures and finally prints out a table which represents the tree, by listing the called procedures and calling procedures in a suitable way.
Step 3 — Formalisation of the strategy

Need to identify:

1. the objects

2. the operations associated with each object

3. inter-relations between the objects

list of **objects** found from *nouns* in outline design

list of **operations** found from *verbs* in outline design

To clarify their role, nouns can be categorised:

(a) *common* noun: abstract data types (and instances),
    i.e., a ‘class’ name, such as a *book, program, file*, etc.

(b) *mass* noun: a unit of measure

(c) *proper* noun: a specific entity in the system
Finding the objects

By going through the **nouns** in the solution:

The **program** first reads through the **input Pascal source file** and makes a **list** of the procedures that are declared in that **program**. It then reads the **input file** again and this time makes a **note** of each call that is made to any of the **procedures** that were previously identified and also records the **identity** of the calling component. From these **records** it constructs a **tree** that shows the **interdependencies** of the different procedures and finally prints out a **table** which represents the **tree**, by listing the **called procedures** and **calling procedures** in a suitable way.
(1) Program can be discarded — reference to solution.
(2) Input Pascal source file — identifies source file as an object that the program will operate on (open, read, write, close)
(3,4) List of procedures seems a good candidate for an object — program will add items to it and search for entries, etc.
(5) Program — this time a synonym for source file
(6) Input file — another synonym for source file
(7,8) Note (of each call) might be an object, but can be replaced by an action — notes each call, should be discarded.
(9) Procedures could be a candidate for an object — procedure identifier (operations are a bit vague though)
(10,11) Identity (of a calling component) a synonym for procedure identifier — bearing in mind notes each call above suggests that calling and caller might be attributes
(12) Records is a synonym for list
(13) Tree suggests presence of new data structure, maybe constructed from existing objects — review later.
(14,15) Interdependencies is really another reference to attributes of procedure identifier
(16) Table could be an object, but seems to relate to tree
(17) Tree is repeat of (13)
(18) Called (procedure) an attribute of procedure identifier
(19) Calling (procedure) also an attribute of procedure identifier
**Identified objects**

Examining the survivors we find:

(a) common (class) nouns: *procedure identifier*

(b) mass nouns: none

(c) proper nouns:

   i. *source file*

   ii. *list of procedures*

   iii. *tree of relations*
Finding the operations

We now identify the **verbs**:

The program first **reads** through the input Pascal source file and **makes** a list of the procedures that **are declared** in that program. It then **reads** the input file again and this time **makes** a note of each call that **is made** to any of the procedures that **were previously identified** and also **records** the identity of the calling component. From these records it **constructs** a tree that **shows** the interdependencies of the different procedures and finally **prints out** a table which **represents** the tree, by **listing** the called procedures and calling procedures in a suitable way.
Then we link each operation to an object:

(1) **Reads** operation on *source file* (ReadItem)

(2) **Makes** (a list) adds an entry to *list of procedures* (AddEntry)

(3) **Are declared** is an ‘existence’ verb discard unless an attribute

(4) **Reads** (again) same as (1)

(5) **Makes** (a note) can be regarded as adding/changing an attribute to *procedure identifier*

(6) **Is made** simply identifies the attribute to be modified (5)

(7) **Were previously identified** is an existence reference

(8) **Records** identifies the attribute of *list of procedures* that is to be modified

(9) **Constructs** is an operation (or set of) on *tree of relations*

(10) **Shows** is another existence reference

(11) **Prints out** is an operation (or set) applied to *tree of relations*

(12) **Represents** is another existence reference

(13) **Listing** is an operation on object not yet considered — *output file*
Operations and associated objects

<table>
<thead>
<tr>
<th>Object</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure Identifier</td>
<td>Add_Caller_Identity, Add_Calling_Identity</td>
</tr>
<tr>
<td>Source File</td>
<td>Read_Item</td>
</tr>
<tr>
<td>Output File</td>
<td>Print_Item</td>
</tr>
<tr>
<td>List of Procedures</td>
<td>Add_Entry</td>
</tr>
<tr>
<td>Tree of Relations</td>
<td>Add_Relation, Read_Node</td>
</tr>
</tbody>
</table>

Relation between classes and methods. Note that last two objects are transforms of each other and can be merged.

Step 4 — Formalisation of the solution

Yielding a more complete solution by:

- providing formal definitions of object interfaces
- defining the parent object of the whole system
<table>
<thead>
<tr>
<th><strong>Object</strong></th>
<th><strong>Attribute</strong></th>
<th><strong>Operation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedure identity</td>
<td>Procedure identifier, Link to list of calling procs, Link to list of called procs</td>
<td>create_proc_identifier, add_caller_link, add_called_link</td>
</tr>
<tr>
<td>Source file</td>
<td>File identifier, Open/close, End of file</td>
<td>read_token, open_file, close_file</td>
</tr>
<tr>
<td>Output file</td>
<td>File identifier, Open/close</td>
<td>print_string, open_file, close_file</td>
</tr>
<tr>
<td>Table of procedures</td>
<td>Entries, Entry relations</td>
<td>add_new_entry, add_link_to_entry, determine_tree_level, traverse_tree</td>
</tr>
</tbody>
</table>

Revised relation of classes and methods via their attributes.

**HOOD diagram of the above dependence-tracing example.**
Summary of HOOD

- HOOD theory
  1. objects
  2. classes
  3. inheritance

- HOOD representation

- HOOD process

- HOOD example
  1. statement of requirements
  2. outline of informal solution
  3. conversion into objects
  4. formalisation of interfaces