Software Architecture & Design

- What is design
- How can a system be decomposed into modules
- What is a module's interface
- What are the main relationships among modules
- Prominent software design techniques and information hiding
- The UML collection of design notations
- Design of concurrent and distributed software
- Design patterns
- Architectural styles
- Component based software engineering

131

131

What is design?

- Provides structure to any artifact
- Decomposes system into parts, assigns responsibilities, ensures that parts fit together to achieve a global goal
- Design refers to both an activity and the result of the activity

132

132

Two meanings of "design" activity in our context

- Activity that acts as a bridge between requirements and the implementation of the software
- Activity that gives a structure to the artifact
  - e.g., a requirements specification document must be *designed*
    - must be given a structure that makes it easy to understand and evolve

133

133

Two meanings of "design" activity in our context

- Activity that gives a structure to the artifact

Requirements	**SDD**
Decomposed so domain experts can read & understand it easily	Decomposed so software will be structured for easy maintenance

134

134

The sw design activity

- Defined as system decomposition into modules
- Produces a Software Design Document
  - describes system decomposition into modules
- Often a *software architecture* is produced prior to a software design

135

135

Software architecture

- Shows gross structure and organization of the system to be defined
- Its description includes description of
  - main components of a system
  - relationships among those components
  - rationale for decomposition into its components
  - constraints that must be respected by any design of the components
- Guides the development of the design

136

136

Two important goals

- Design for change (Parnas)
  - designers tend to concentrate on current needs
  - special effort needed to anticipate likely changes
- Product families (Parnas)
  - think of the current system under design as a member of a program family

137

137

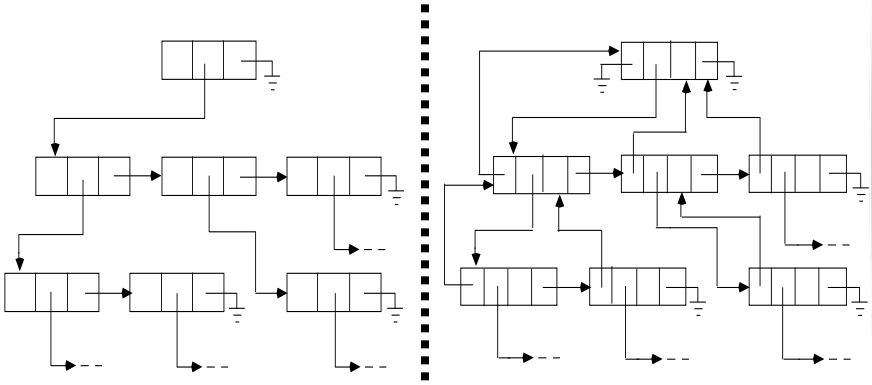
Sample likely changes? (1)

- Algorithms
  - e.g., replace inefficient sorting algorithm with a more efficient one
- Change of data representation
  - e.g., from binary tree to a threaded tree (see example)
  - ≈17% of maintenance costs attributed to data representation changes (Lientz and Swanson, 1980)

138

138

Example



139

139

Sample likely changes? (2)

- Change of underlying abstract machine
  - new release of operating system
  - new optimizing compiler
  - new version of DBMS
  - …
- Change of peripheral devices
- Change of "social" environment
  - new tax regime
  - EURO vs national currency in EU
- Change due to development process (transform prototype into product)

140

140

Product families

- Different versions of the same system
  - e.g. a family of mobile phones
    - members of the family may differ in network standards, end-user interaction languages, …
  - e.g. a facility reservation system
    - for hotels: reserve rooms, restaurant, conference space, …, equipment (video beamers, overhead projectors, …)
    - for a university
      - many functionalities are similar, some are different (e.g., facilities may be free of charge or not)

141

141

Design goal for family

- Design the whole family as one system, not each individual member of the family separately

142

142

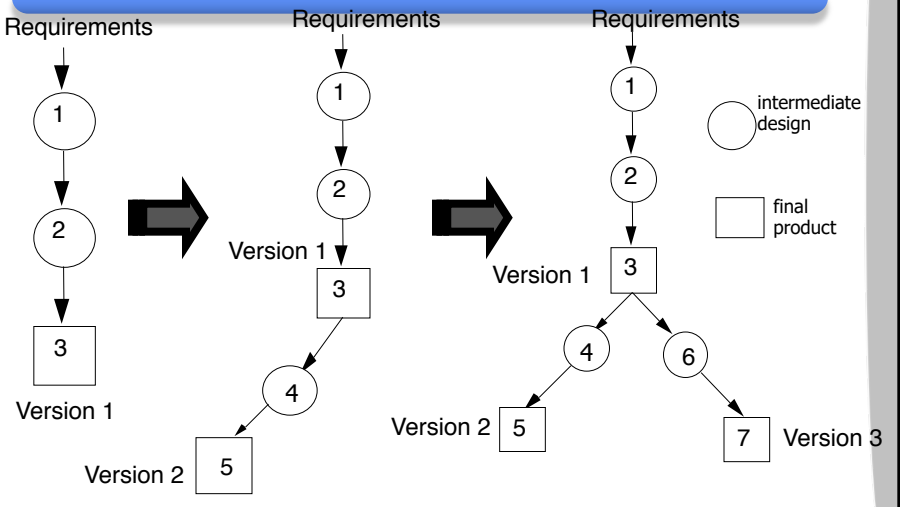
Sequential completion: the wrong way

- Design first member of product family
- Modify existing software to get next member products

143

143

Sequential completion: a graphical view



144

144

How to do better

- Anticipate definition of all family members
- Identify what is common to all family members, delay decisions that differentiate among different members
- We will learn how to manage change in design

145

145

Module

- A well-defined component of a software system
- A part of a system that provides a set of services to other modules
  - Services are computational elements that other modules may use
- A work assignment (Parnas)

146

146

Modularity

- A complex system may be divided into simpler pieces called *modules*
- A system that is composed of modules is called *modular*
- Supports application of separation of concerns
  - when dealing with a module we can ignore details of other modules

147

147

Questions

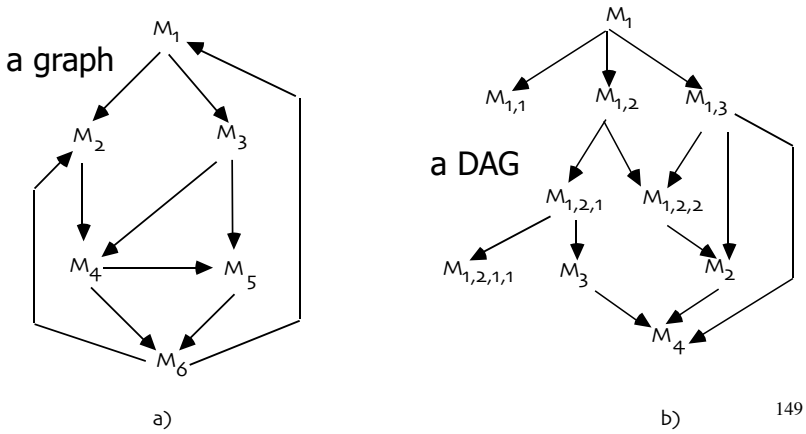
- How to define the structure of a modular system?
- What are the desirable properties of that structure?

148

148

Relations

- A hierarchy is a DAG (directed acyclic graph)



149

149

The USES relation

- A uses B
  - A requires the correct operation of B
  - A can access the services exported by B through its interface
  - it is "statically" defined
  - A depends on B to provide its services
    - example: A calls a routine exported by B
- A is a client of B; B is a server

150

150

Desirable property

- USES should be a hierarchy
- Hierarchy makes software easier to understand
  - we can proceed from leaf nodes (who do not use others) upwards
- They make software easier to build
- They make software easier to test

151

151

Hierarchy

- Organizes the modular structure through *levels of abstraction*
- Each level defines an *abstract (virtual) machine* for the next level
  - *level* can be defined precisely
    - M_i has level 0 if no M_j exists s.t. M_i u M_j
    - let k be the maximum level of all nodes M_j s.t. M_i u M_j. Then M_i has level k+1

152

152

Cohesion and coupling

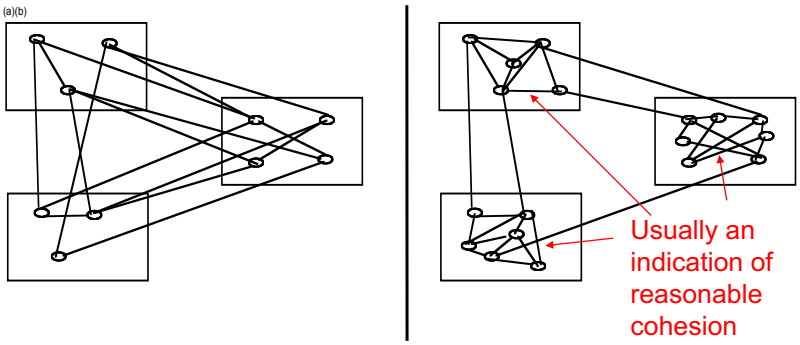
- Each module should be *highly cohesive*
  - module understandable as a meaningful unit
  - Components of a module are closely related to one another
- Modules should exhibit *low coupling*
  - modules have low interactions with others
  - understandable separately

153

153

A visual representation

We can evaluate coupling from an external view (”uses”), but need to see internal details to judge cohesion



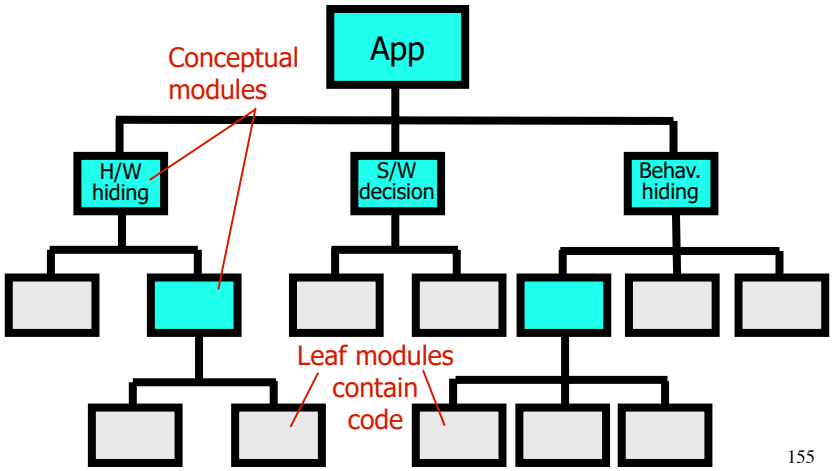
high coupling

low coupling

154

154

Module Decomposition (Parnas)



155

155

Module Fundamentals

Name
Interface
- Types
- Constants
- Access programs

156

156

Module Fundamentals

Name
Interface
Types
Constants
Access programs

Hidden from everyone other than the designers of the module

Implementation
Types
Constants
Variables

Details of access programs & local programs

157

157

Module Fundamentals

Module A
int getS1
int getS2
setValues(int v1,v2)

int S1, S2
....
setValues(int v1,v2)
...
q = B.getV
S1 = v1
S2 = v2*q

Module B
int getV
setValue(int n)

int V
....
setValue(int n)
...
V = n

158

158

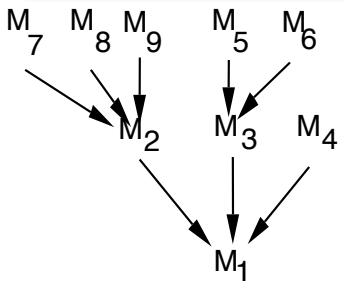
IS_COMPONENT_OF

- Used to describe a higher level module as constituted by a number of lower level modules
- A IS_COMPONENT_OF B
  - B consists of several modules, of which one is A
- B COMPRISES A
- $M_{S,i}=\{M_k | M_k \in S \wedge M_k$ IS_COMPONENT_OF $M_i\}$ we say that $M_{S,i}$ IMPLEMENTS M_i

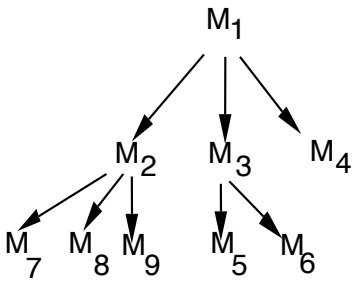
159

159

A graphical view



(IS_COMPONENT_OF)



(COMPRISES)

They are a hierarchy

160

160

Product families

- Careful recording of (hierarchical) USES relation and IS_COMPONENT_OF supports design of program families

161

161

Interface vs. implementation (1)

- To understand the nature of USES, we need to know what a used module *exports* through its *interface*
- The client *imports* the resources that are exported by its servers
- Modules *implement* the exported resources
- Implementation is *hidden* to clients

162

162

Interface vs. implementation (2)

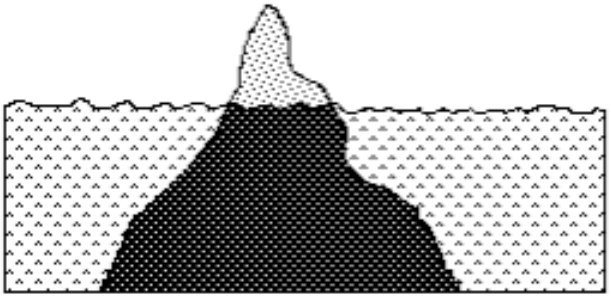
- Clear distinction between interface and implementation is a key design principle
- Supports separation of concerns
  - clients care about resources exported from servers
  - servers care about implementation
- Interface acts as a contract between a module and its clients

163

163

Interface vs. implementation (3)

interface is like the tip of the iceberg

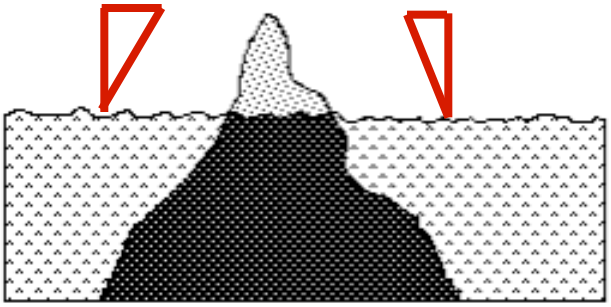


164

164

Interface vs. implementation (3)

interface is like the tip of the iceberg
- need to include "effects" as well



165

165

Interface vs. implementation (4)

- To describe the interface, we need to describe the relationship between outputs (responses) to inputs (stimuli).
- One way is to use a "black-box" model, from Harlan Mills' "Box Structures".
- This is the "history-based" approach we saw earlier.

166

166

Interface vs. implementation (5)

- When we use an infinite history as in the Mills black-box description, we can describe Ri in terms of any Sj and their history.
- We can include statements such as “most recent value of Sk”, for example.
- If we describe the behaviour solely in terms of **S** and $\mathbf{S}_h$, i.e. externally visible behaviour, we will not include any implementation details.

167

167

Interface vs. implementation (6)

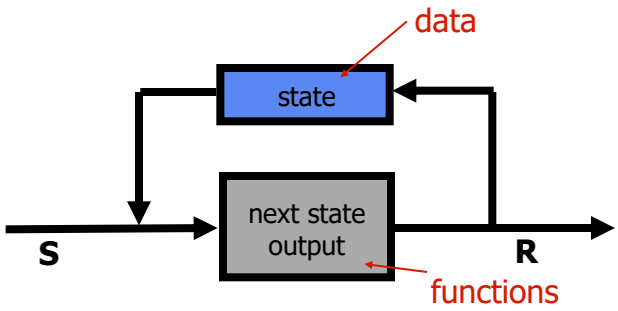
- There are a couple of problems with a history-based approach:
  - Infinite history is not practical when it comes to building an implementation (my opinion - it’s one reason why it is great for describing requirements, and the interface represents requirements for the module).
  - Some people find it difficult to write history-based descriptions.

168

168

Interface vs. implementation (7)

- An alternative to the black-box description is a finite state machine FSM - we saw that earlier too



169

169

Interface vs. implementation (8)

- The “state” in the FSM captures the effect of history in the black-box representation.
- We have to be careful in describing interfaces using a FSM - don’t give away how the state data is stored etc.
- We do this by keeping the state data simple - usually just the previous value of a stimulus or response.

170

170

Remember - Information hiding

- Basis for design (i.e. module decomposition)
- Implementation secrets are hidden to clients
- They can be changed freely if the change does not affect the interface
- Golden design principle
  - *INFORMATION HIDING*
    - *Try to encapsulate changeable (requirements &) design decisions as implementation secrets within module implementations*

171

171

Interface design

- Interface should not reveal what we expect may change later
- It should not reveal unnecessary details
- Interface acts as a firewall preventing access to hidden parts

172

172

Prototyping

- Once an interface is defined, implementation can be done
  - first quickly but inefficiently
  - then progressively turned into the final version
- Initial version acts as a prototype that evolves into the final product

173

173

More on likely changes *an example*

- *Policies* may be separated from *mechanisms*
  - mechanism
    - ability to suspend and resume tasks in a concurrent system
  - policy
    - how do we select the next task to resume?
      - different scheduling policies are available
      - they may be hidden to clients
      - they can be encapsulated as module secrets

174

174

Design notations

- Notations allow designs to be described precisely
- They can be textual or graphic
- We illustrate two sample notations
  - Parnas' Rational Design Process (MG, MIS, MID)
  - We discuss the notations provided by UML

175

175

RDP

- Parnas proposed a Rational Design Process (RDP)
- There are a number of variations - but there are some fundamentals.
- We will look at the fundamentals of documenting an RDP design

176

176

RDP - Module Guide

- When decomposing the system into modules (as per earlier discussions), we need to document the module decomposition so that the developers and other readers can understand the decomposition.
- Parnas proposed a Module Guide (MG) based on the decomposition module tree shown earlier.

177

177

RDP - MG

- The MG consists of a table that documents each module's responsibility and secret.
- Conceptual modules will have broader responsibilities and secrets.
- The leaf modules that represent code will contain much more precise responsibilities and secrets.

178

178

RDP - MG Example

#	Name	Responsibility	Secret
…	…	…	…
1.3	AIs	Interface with the A/I hardware to produce software variables that represent the field data	Hardware representation of data & how to get data from hardware
1.3.1	FlowSensors	Software representation of flow measurements	Relationship between physical flow and software representation
…	…	…	…

179

179

RDP - MIS

- For each leaf module we need to document its interface and its implementation.
- In RDP, the interfaces are documented in the Module Interface Specification (MIS).
- We would aim to use some form of mathematical approach to specify the MIS behaviour - as black-box as possible.

180

180

RDP - MIS Example

- A stack interface specification is an example of what would go into the MIS.
- Parnas is not clear as to whether there is a document that contains all the interfaces of all the modules, or whether each module's documentation has an MIS section - my preference.

181

181

RDP - Views

- As well as the MG, the modular decomposition should be displayed using a variety of views.
- An obvious one is a "Uses Hierarchy" - which can be formed once the MIS for all modules is complete. It is not always obvious how to achieve this - for complex systems graphical representations are not successful without good tool support.

182

182

RDP - MID

- Finally, we can document the Module Internal Design (MID) for each module.
- This provides the implementation of the module, i.e. how we deliver on what we promised in the MIS.
- The more complete this is, the better.
- But - it represents a requirement for the coder - so it is better represented at a higher level of abstraction than the code.

183

183

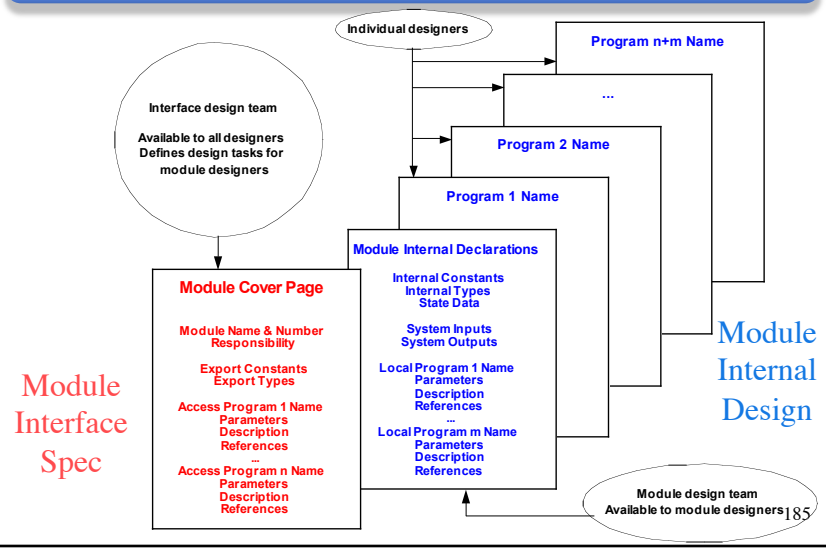
RDP -Module Documents

- MG - shows module decomposition
- Uses hierarchy (produced after all MIS)
- Module Design for each module
  - MIS
  - MID

184

184

Documentation of Modules



185

TDN

- Illustrate how a notation may help in documenting design
- Illustrate what a generic notation may look like
- Are representative of many proposed notations
- TDN inherits from modern languages, like Java, Ada, …

186

186

An example

module X
uses Y, Z
exports var A : integer;
type B : **array** (1. .10) **of** real;
procedure C (D: **in out** B; E: **in** integer; F: **in** real);
Here is an optional natural-language description of what A, B, and C actually are, along with possible constraints or properties that clients need to know; for example, we might specify that objects of type B sent to procedure C should be initialized by the client and should never contain all zeroes.
implementation
If needed, here are general comments about the rationale of the modularization, hints on the implementation, etc.
is composed of R, T
end X

187

187

Comments in TDN

- May be used to specify the *protocol* to be followed by the clients so that exported services are correctly provided
  - e.g., a certain operation which does the initialization of the module should be called before any other operation
  - e.g., an insert operation cannot be called if the table is full

188

188

Example (cont.)

```
module  R
uses Y
exports  var K : record  . . . end;
          type  B : array  (1. .10) of real;
          procedure   C (D: in out  B; E: in  integer; F: in  real);
implementation
          .
          .
          .
end  R

module  T
uses Y, Z, R
exports  var A : integer;
implementation
          .
          .
          .
end  T
```

189

189

Benefits

- Notation helps describe a design precisely
- Design can be assessed for consistency
  - having defined module X, modules R and T must be defined eventually
    - if not → *incompleteness*
  - R, T replace X
    - → either one or both must use Y, Z

190

190

Categories of modules - 1

- Functional modules
  - traditional form of modularization
  - provide a procedural abstraction
  - encapsulate an algorithm
    - e.g. sorting module, fast Fourier transform module, …

191

191

Categories of modules- 2

- Libraries
  - a group of related procedural abstractions
    - e.g., mathematical libraries
      - implemented by routines of programming languages
- Common pools of data
  - data shared by different modules
    - e.g., configuration constants
      - the COMMON FORTRAN construct

192

192

Categories of modules - 3

- Abstract objects
  - Objects manipulated via interface functions
  - Data structure hidden to clients
- Abstract data types
  - Many instances of abstract objects may be generated

193

193

Abstract data types (ADTs)

- A stack ADT

indicates that details of the data structure are hidden to clients

```
module   STACK_HANDLER
exports
        type  STACK = ?;
        This is an abstract data -type module; the data structure
        is a secret hidden in the implementation part.
        procedure   PUSH (S:  in out  STACK ; VAL:  in  integer);
        procedure   POP (S: in out  STACK ; VAL:  out  integer);
        function   EMPTY (S:  in  STACK) : BOOLEAN;
        .
        .
        .
end  STACK_HANDLER
```

194

194

ADTs

- Correspond to Java and C++ classes
- Concept may also be implemented by Ada private types and Modula-2 opaque types
- May add notational details to specify if certain built-in operations are available by default on instance objects of the ADT
  - e.g., type A_TYPE: ? (:=, =) indicates that assignment and equality check are available

Object-oriented design

- One kind of module, ADT, called *class*
- A class exports operations (procedures) to manipulate instance objects
  - often called *methods*
- Instance objects accessible via references

Syntactic changes in TDN

- No need to export opaque types
  - class name used to declare objects
- If a is a reference to an object
  - a.op (params);

197

197

A further relation- inheritance

- ADTs may be organized in a hierarchy
- Class B may specialize class A
  - B inherits from A

  conversely, A generalizes B
- A is a superclass of B
- B is a subclass of A

198

198

An example - 1

```
class EMPLOYEE
exports
        function FIRST_NAME(): string_of_char;
        function LAST_NAME(): string_of_char;
        function AGE(): natural;
        function WHERE(): SITE;
        function SALARY: MONEY;
        procedure HIRE (FIRST_N: string_of_char;
                        LAST_N: string_of_char;
                        INIT_SALARY: MONEY);
        Initializes a new EMPLOYEE, assigning a new identifier.
        procedure FIRE();
        procedure ASSIGN (S: SITE);
        An employee cannot be assigned to a SITE if already assigned to it (i.e., WHERE
        must be different from S). It is the client's responsibility to  ensure this. The effect is to
        delete the employee from those in WHERE, add the employee to those in S, generate
        a new id card with security code to access the site overnight, and update WHERE.
end EMPLOYEE
```

199

199

An example - 2

```
class ADMINISTRATIVE_STAFF inherits EMPLOYEE
exports
        procedure DO_THIS (F: FOLDER);
        This is an additional operation that is specific to
        administrators; other operations may also be added.
end ADMINISTRATIVE_STAFF

class TECHNICAL_STAFF inherits EMPLOYEE
exports
        function GET_SKILL(): SKILL;
        procedure DEF_SKILL (SK: SKILL);
        These are additional operations that are specific to
        technicians; other operations may also be added.
end TECHNICAL_STAFF
```

200

200

Inheritance

- A way of building software incrementally
- A subclass defines a subtype
  - subtype is *substitutable* for parent type
- Polymorphism
  - a variable referring to type A can refer to an object of type B if B is a subclass of A
- Dynamic binding
  - the method invoked through a reference depends on the type of the object associated with the reference at runtime

201

201

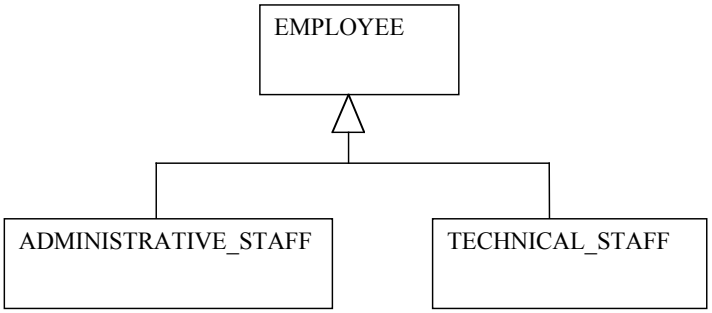
How to represent inheritance

- We start introducing the UML notation
- UML (Unified Modeling Language) is a widely adopted standard notation for representing OO designs
- We introduce the UML class diagram
  - classes are described by boxes

202

202

UML - inheritance

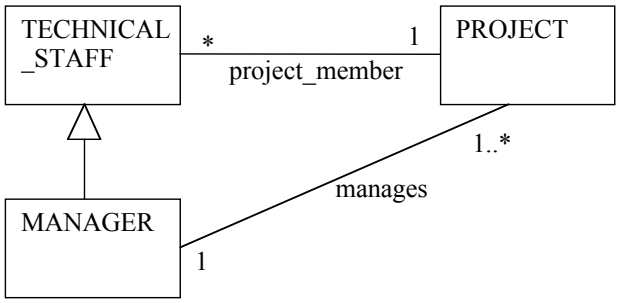


203

203

UML - associations

- Associations are relations that the implementation is required to support
- Can have multiplicity constraints

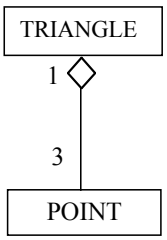


204

204

Aggregation

- Defines a PART_OF relation
  Differs from IS_COMPOSED_OF
  Here TRIANGLE has its own methods
  It implicitly uses POINT to define
  its data attributes

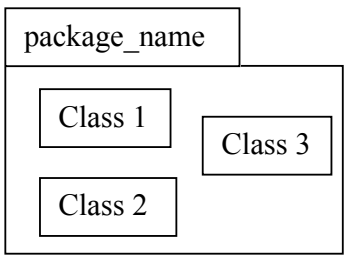


205

205

More on UML

- Representation of IS_COMPONENT_OF via the *package* notation



206

206