Design and Software Architecture
What is design?

- Activity that provides structure to any artifact:
  - **Different domains**: construction, mechanical systems, chemical systems, electrical systems
  - **Different aspects**: system design, hardware design, and software design

- 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
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
  - Requirements specification document must be designed
  - Must be given a structure that makes it easy to understand and evolve
The software 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
Software architecture

- Shows gross structure and organization of the system to be defined

- Its description includes description of:
  - Main components of a system
  - Rationale for decomposition into its components
  - Relationships among those components
  - Constraints that must be respected by any design of the components

- Guides the development of the design
Software architecture: an attempt to solve the problems of large systems

<table>
<thead>
<tr>
<th>Problems</th>
<th>Solution provided by Software Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Human’s inability in comprehending all the details of a project</td>
<td>• Separation of Concerns, and high-level view of system</td>
</tr>
<tr>
<td>• The lack of standard software design and development techniques</td>
<td>• Design patterns, design for reuse, architectural styles</td>
</tr>
<tr>
<td>• Legacy systems maintenance and evolution</td>
<td>• Software architecture recovery</td>
</tr>
</tbody>
</table>
Software Architecture definitions

• A generally accepted definition:

“The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time” [SEI 1994]

• However, software architecture is more than “components and connectors”, or “major elements of a system”. It is a collection of views, patterns, stakeholders, and roles [SEI].

• Therefore, Software architecture provides the necessary means to formalize and interpret the properties of a system.
Software architecture terminology

- **Components**: encapsulation of the system’s computation
  - Filter, layer, client, etc.

- **Connectors**: encapsulation of interactions among components
  - RPC, event broadcast, pipe, etc.

- **Styles**: definition of components & connectors, their properties, and configuration constraints that apply to all instances of a family of closely related systems
  - Pipe/filter, implicit invocation, client/server, etc.
Component Diagram for ABM

- Meaningful naming for services
- Correct direction for the dependency arrows.
See the brief guideline on how to design Component Diagram and Statechart at the architectural level of a system in Lab 4 description.
Important Aspects of Software Architecture

- Architectural Styles
- Architectural views
- Architecture Description Languages
- Architectural Analysis and Evaluation
- Architecture Recovery
Architectural Styles

• **Styles**: definition of components & connectors, their properties, and configuration constraints that apply to all instances of a family of closely related systems
  – Pipe/filter, implicit invocation, client/server, etc.

• Assist in the construction of reliable, cost efficient, and understandable systems

• Support reusability by capturing the common properties of a family of programs.

• Styles can be implemented by the use of design patterns
  – **Design patterns**: are collections of structural and behavioral guidelines and rules for modeling frequently occurring design decisions in large systems.
Sample Architectural Styles

- **Pipe and filter**: UNIX shell
- **Client and server**: distributed systems
- **Implicit invocation (Broker)**: CORBA, HP SoftBench
- **Layered**: ISO/OSI reference model
- **Data repository (Blackboard)**: modern compilers, databases
- **Object-oriented**: Aesop
- **Interpreter**: programming languages
- **State transition**: reactive systems
- **Main program and procedure**: traditional systems
- **DSSA**: avionics, C2, vehicle management
**Example: Pipes and Filters**

- **Filters**: components that receive a stream of input data, manipulate data, and produce a stream of output data.
- **Pipes**: a sequential connector (file) that acts as a buffer of data between two filters and can provide parallel operations of filters.
- **Invariants**: filters (components) must be independent entities; filters do not know the identity of their upstream and downstream filters.
- **Common specialization**: pipelines which restrict the topologies to linear sequences; bounded pipes which restrict the amount of data that can reside in a pipe; typed pipes which require the data passed between two filters have a well-defined type.
- **A degenerate case**: One filter processes all of its input data as a single entity.
- **Examples**: Unix shell programs; compilers.
- **Advantages**: clean design; support reuse; easy to maintain; permit specialized analysis (e.g. deadlock detection) support concurrency.
- **Disadvantages**: batch organization of processing (not interactive); difficult to synchronize pipes. Ch. 4
Different Architectural Styles

Pipe & Filter

Client & Server

Implicit Invocation (Broker)

Object Oriented

State Transition

Blackboard

Layered

Main Program & Procedure
Software Architecture Views
Software Architecture views

Views are the result of applying separation of concern on the development process to categorize the related knowledge about the system into manageable and understandable forms.

- Architectural views assist engineers in understanding, developing, and communicating different aspects of a software system.
- Different groups of researchers have developed their own set of views.
- Two important examples: 4+1 views and Zachman framework.
Software Architecture views
4+1 views

- **4+1 View Model**
  1) **Logical**: functional requirement
  2) **Process**: concurrency, distribution of system services
  3) **Deployment**: planning, monitoring, reuse
  4) **Physical**: network topology

+1) **Scenario**: represents the sequence of system operations; it shows the relation among the elements of the 4 views:
- Scenarios are represented by: object-interaction diagram, sequence diagram, collaboration diagram, statecharts.
- Sometimes referred to as: use-case, or work flow.
4 + 1 Architectural Views

Logical View
- End-user
  - Functionality

Deployment View
- Programmers
  - Configuration management

Process View
- System integrators
  - Performance
  - Scalability
  - Throughput

Physical View
- System engineering
  - System topology
  - Communication

Scenario or Use Case View

Conceptual
- Physical
Scenario for telephone conversation in a PBX system

Figure shows a fragment of a scenario for the small PBX:

1. The controller of Joe’s phone detects and validate the transition from on-hook to off-hook and sends a message to wake up the corresponding terminal object.
2. The terminal allocates some resources, and tells the controller to emit some dial-tone.
3. The controller receives digits and transmits them to the terminal.
4. The terminal uses the numbering plan to analyze the digit flow.
5. When a valid sequence of digits has been entered, the terminal opens a conversation.
Software Architecture views: Zachman’s framework (Views & Perspectives)

Zachman proposes a framework of *views* and *perspectives* that allows to map the knowledge about the system into non-overlapping representations provided by the framework.

*Views* categorize the knowledge about the system into manageable and understandable forms. They answer to questions on: *What, How, Where*

<table>
<thead>
<tr>
<th>Views:</th>
<th>Data view</th>
<th>Function view</th>
<th>Network view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Question:</td>
<td>What the software is made of?</td>
<td>How the software works?</td>
<td>Where the connections exist?</td>
</tr>
<tr>
<td>Concern:</td>
<td>Material</td>
<td>Functionality</td>
<td>Location</td>
</tr>
<tr>
<td>Focus:</td>
<td>Structure of data</td>
<td>Data transformation</td>
<td>Flow</td>
</tr>
</tbody>
</table>
Zachman's framework (views & perspectives)

**Perspectives:**
Each perspective is a system documentation that reflects the interests of a different stakeholder in software development.

- **General scope:** (Owner & business planer)
  - Description of gross estimation of the product’s features
- **Owner** (Owner):
  - Representation of owner’s desires from the final product
- **Designer** (architect):
  - Translation of owner’s representation to a technical plan
- **Developer** (contractor):
  - Translation of designer’s plan to a feasible plan
- **Programmer** (builder):
  - Actual production from a feasible plan
<table>
<thead>
<tr>
<th>Perspectives</th>
<th>Framework</th>
<th>Data view</th>
<th>Function view</th>
<th>Network view</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General scope</strong></td>
<td></td>
<td>List of entities important to business</td>
<td>List of functions the business performs</td>
<td>List of locations the business operates</td>
</tr>
<tr>
<td>(Ballpark)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Owner's perspective</strong></td>
<td></td>
<td>Entity-relation diagram</td>
<td>Function flow diagram</td>
<td>Logistic network</td>
</tr>
<tr>
<td>(Architect’s plan)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Designer's perspective</strong></td>
<td></td>
<td>Data model</td>
<td>Data flow diagram</td>
<td>Distributed system architecture</td>
</tr>
<tr>
<td>(Contractor’s plan)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Developer's perspective</strong></td>
<td></td>
<td>Data design</td>
<td>Structure chart</td>
<td>System architecture</td>
</tr>
<tr>
<td>(Contractor’s plan)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Programmer's perspective</strong></td>
<td></td>
<td>Data description</td>
<td>Program</td>
<td>Network architecture</td>
</tr>
<tr>
<td>(Builder’s product)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design Example:

System Specification:

This system controls different activities in a typical Fast-Food Restaurant, and consists of five communicating units:

- Order-Taking unit.
- Assembling unit.
- Preparation unit.
- Inventory unit.
- Management unit.
Restaurant system with mixture of different views
# Framework of different representations of restaurant sys.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Data view</th>
<th>Functional view &amp; Process view</th>
<th>Network view</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General scope</strong> (&lt;Ballpark&gt;)</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>Owner’s perspective</strong></td>
<td><img src="image4.png" alt="Diagram" /></td>
<td><img src="image5.png" alt="Diagram" /></td>
<td><img src="image6.png" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>Designer’s perspective</strong> (&lt;Architect’s plan&gt;)</td>
<td><img src="image7.png" alt="Diagram" /></td>
<td><img src="image8.png" alt="Diagram" /></td>
<td><img src="image9.png" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>Developer’s perspective</strong> (&lt;Contractor’s plan&gt;)</td>
<td><img src="image10.png" alt="Diagram" /></td>
<td><img src="image11.png" alt="Diagram" /></td>
<td><img src="image12.png" alt="Diagram" /></td>
</tr>
<tr>
<td><strong>Programmer’s perspective</strong> (&lt;Builder’s product&gt;)</td>
<td><img src="image13.png" alt="Diagram" /></td>
<td><img src="image14.png" alt="Diagram" /></td>
<td><img src="image15.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
General Scope (Perspective)

Zachman’s Framework of Views and Perspectives
Restaurant-menu, menu-items, orders, raw-materials, and recipes are the main data entities

- Each order consists of menu-items.
- Menu-items are selected from the restaurant-menu
- Each menu-item consists of raw-materials and the recipe

Fast-food restaurant system:
- General Scope
- Data view
Setting-up orders from restaurant-menu
Handling order payment
Distributing menu-items to be prepared
Assembling orders from the prepared menu-items
Keeping track of raw-material consumption
Setting-up different tables, prices, and recipes
Network View (General Scope)

Manager station

Food-preparation stations

Order-taking stations

Inventory station

Food-assembly stations

Customers

Fast-food restaurant system:
- General scope
- Network view
Owner’s Perspective

Zachman’s Framework of Views and Perspectives
Restaurant-menu, menu-item (tables), order (tables), raw-material (table), and are the main data entities

- Each order consists of menu-items.
- Menu-items are selected from the restaurant-menu
- Each menu-item consists of raw-materials and the recipe
Function View (Owner’s Perspective)

- Setting-up orders from restaurant-menu
- Handling order payment
- Distributing menu-items to be prepared
- Assembling orders from the prepared menu-items
- Keeping track of raw-material consumption
- Setting-up different tables, prices, and recipes

Top-level functional flow diagram.

Second-level

- Take orders & payments
  - 4.1 Set "order in OT station" from "restaurant menu"
  - 4.2 Retrieve "order in OT station" from "stored orders"
  - 4.3 Add/delete "menu-item" from "order in OT station"
  - 4.4 Put "order in OT station" in "stored orders"
  - 4.5 Handle "cash" payment
  - 4.6 Put "order in OT station" in "Completed orders"

Second-level

- Assemble orders
  - 7.1 Put items of new order into "menu-items-to-be-prepared"
  - 7.2 Determine "ready orders"
  - 7.3 Distribute "ready orders" among active Asm stations
  - 7.4 Report assembly of "order in Asm station" to the system
  - 7.5 Report the shortage of "chute-items" to the system
  - 7.6 Delete newly paid order form "unpaid orders"
  - 7.7 Put "order in Asm station" in "unpaid orders"

Fast-food restaurant system:
- Owner’s perspective
- Functional & Process views
Network View (Owner’s Perspective)

- Order-taking stations have touch-screen display.
- Food-preparation & assembly stations have keypad and display.
- Manager and Inventory stations have keyboard and display.

Fast-food restaurant system:
- Owner’s perspective
- Network view
Aspects of Design
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
  - Design the core part of a family of system
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)
  - $\approx 17\%$ of maintenance costs attributed to data representation changes (Lientz and Swanson, 1980)
Example
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
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)
Design goal for family

• Design the whole family as one system, not each individual member of the family separately
Sequential completion: the wrong way

- Design first member of product family
- Modify existing software to get next member products
Sequential completion: a graphical view

Requirements

Version 1

Version 2

Version 3

intermediate design

final product

Ch. 4
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
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
Questions

• How to define the structure of a modular system?
• What are the desirable properties of that structure?
Modules and relations

• Let $S$ be a set of modules
  \[ S = \{M_1, M_2, \ldots, M_n\} \]

• A binary relation $r$ on $S$ is a subset of
  \[ S \times S \text{ that is } r \subseteq S \times S \]

• If $M_i$ and $M_j$ are in $S$, then:
  \[ <M_i, M_j> \in r \text{ can be written as } M_i \mathbin{r} M_j \]
Relations

• Transitive closure $r^+$ of $r$
  
  $M_i r^+ M_j$ iff
  
  $M_i r M_j$ or $\exists M_k$ in $S$ s.t. $M_i r M_k$
  and $M_k r^+ M_j$

  (We assume our relations to be irreflexive)

• $r$ is a hierarchy iff there are no two elements $M_i, M_j$ s.t. $M_i r^+ M_j \land M_j r^+ M_i$
Relations

- Relations can be represented as graphs
- A hierarchy is a DAG (directed acyclic graph)
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
- In a modular system:
  - \(|r| \ll n^2\) where \(n = |S|\)
  - High fan-in and low fan-out
Component Diagram for ABM

- Meaningful naming for services
- Correct direction for the dependency arrows.
Desirable property

- USES should be a hierarchy: e.g., Layered Style

- 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
Hierarchy

- Organizes the modular structure through *levels of abstraction*

- Each level defines an *abstract (virtual) machine* for the next level (OS architecture)
  - *level* can be defined precisely
    - $M_i$ has level 0 if no $M_j$ exists s.t. $M_i \triangleright M_j$
    - let $k$ be the maximum level of all nodes $M_j$ s.t. $M_i \triangleright M_j$. Then $M_i$ has level $k+1$
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

• $M_{S,i} = \{M_k \mid M_k \in S \land M_k \text{ IS_COMPONENT_OF } M_i\}$
  we say that $M_{S,i}$ IMPLEMENTS $M_i$
A graphical view

They are a hierarchy
Product families

- Careful recording of (hierarchical) USES relation and IS_COMPONENT_OF supports design of program families
Interface vs. implementation (1)

- USES and IS_COMPONENT_OF partially describe an architecture

- 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.
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
Interface vs. implementation (3)

interface is like the tip of the iceberg
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 design decisions as implementation secrets within module implementations
How to design module interfaces?

• Example: design of an interpreter for language MINI
  – We introduce a SYMBOL_TABLE module
    • provides operations to
      – CREATE an entry for a new variable
      – GET the value associated with a variable
      – PUT a new value for a given variable

  – the module hides the internal data structure of the symbol table

  – the data structure may freely change without affecting clients
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
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
More on likely changes

an example

- Policies may be separated from mechanisms

- Mechanism (technique, method)
  - ability to suspend and resume tasks in a concurrent system

- Policy (strategy, scheduling, protocol)
  - 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
Design notations

• Notations allow designs to be described precisely

• They can be textual or graphic

• We illustrate two sample notations
  – TDN (Textual Design Notation)
  – GDN (Graphical Design Notation)

• We discuss the notations provided by UML
TDN & GDN

- 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, ...
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
Comments in TDN

- May be used to specify the *protocol* to be followed by the clients so that exported services are correctly provided.

- Examples of protocols:
  - A certain operation which does the initialization of the module should be called before any other operation
  - An insert operation cannot be called if the table is full
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
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
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
X's decomposition

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

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
Code Compilation and Execution

Source from: http://www.technocage.com/~ray/notespage.jsp?pageName=iki

- -- This doesn't write hello world
- begin -- My first program
- var x ; var y;
- -- lousy indentation and SPACing

  while y - 5 loop
  var y;
  read x ,y;
  x = 2 * (3+y);
  end;
  write 5; -- Not sure why five, but okay.
  end

```
begin var ID(x) ; var ID(y) ; while ID(y) - INTLIT(5) loop var ID(y) ;
read ID(x) , ID(y) ; ID(x) = INTLIT(2) * ( INTLIT(3) + ID(y) ) ; end ;
write INTLIT(5) ; end
```

Tokens generated by Lexical Scanner
Code Compilation and Execution ...

1. Generate Tokens

```
begin var ID(x) ; var ID(y) ; while ID(y) = INTLIT(5) loop var ID(y) ;
read ID(x) ; ID(y) ; ID(x) = INTLIT(2) ; ( INTLIT(3) = ID(y) ) ; end ;
write INTLIT(5) ; end
```

2. Use grammar to parse Tokens (don’t panic!)

```
PROGRAM → begin BLOCK end
BLOCK → (DEC ';')* (STMT ';')+
DEC → var ID
STMT → ID '=' EXP
  | read ID (',' ID)*
  | write EXP (',' EXP)*
  | while EXP loop BLOCK end
EXP → TERM (ADDOp TERM)*
TERM → FACTOR (MULop FACTOR)*
FACTOR → INTLIT | ID | (' EXP ')
ADDOp → '+' | '-'
MULop → '*' | '/'
```

3. Generate Syntax Tree From Tokens
3. Generate Syntax Tree From Tokens

5. Traverse tree
   To generate Assembly Program

6. Generate Machine Language Code for CPU

7. Execute Machine Language On the computer

LD     A, 4EH
MOV  B, A
LD  C, (addr)
ADD  A, B, C
LD  (addr+4), A
Example: a compiler

module COMPILER

exports procedure MINI (PROG: in file of char;
                       CODE: out file of char);
MINI is called to compile the program stored in PROG
and produce the object code in file CODE

implementation

A conventional compiler implementation.
ANALYZER performs both lexical and syntactic analysis
and produces an abstract tree, as well as entries in the
symbol table; CODE_GENERATOR generates code
starting from the abstract tree and information stored
in the symbol table. MAIN acts as a job coordinator.

is composed of ANALYZER, SYMBOL_TABLE,
ABSTRACT_TREE_HANDLER, CODE_GENERATOR, MAIN

end COMPILER
module MAIN
uses ANALYZER, CODE_GENERATOR
exports procedure MINI (PROG: in file of char;
                           CODE: out file of char);
...
end MAIN

module ANALYZER
uses SYMBOL_TABLE, ABSTRACT_TREE_HANDLER
exports procedure ANALYZE (SOURCE: in file of char);
  SOURCE is analyzed; an abstract tree is produced
  by using the services provided by the tree handler;
  and recognized entities, with their attributes, are
  stored in the symbol table.
  ...
end ANALYZER
Other modules

module CODE_GENERATOR

uses SYMBOL_TABLE, ABSTRACT_TREE_HANDLER

exports procedure CODE (OBJECT: out file of char);
   The abstract tree is traversed by using the operations exported by the
   ABSTRACT_TREE_HANDLER and accessing the information stored in the symbol table
   in order to generate code in the output file.

   ...

end CODE_GENERATOR

Draw the GDN of the Compiler Program. GDN is discussed in the next slide
Example of Software Architecture

http://www.turing.toronto.edu/~brewste/bkshelf/
Usage of relations in Software Engineering: Architecture of Apache

Modularity Principle: Decomposing a system based on Low-coupling & High-cohesion

Color-coded Links represent the level of dependency of a file to another file.

NOT IN EXAM
Representing the architecture using graph visualizer (Rigi)

- Different types of links between boxes:
  - Association-links
  - Entity-usage links

- Association-links with different strengths to simplify the view

- Viewing the locus of interaction among entities to evaluate the recovery process

- Insight into the system before starting the recovery

- Manual recovery

File-level analysis

Function-level analysis

NOT IN EXAM
Categories of modules

• Functional modules
  – traditional form of modularization
  – provide a procedural abstraction
  – encapsulate an algorithm
    • e.g. sorting module, fast Fourier transform module, ...
Categories of modules (cont.)

- **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
Categories of modules (cont.)

- **Abstract objects**
  - Objects manipulated via interface functions
  - Data structure hidden to clients
  - Example: Symbol Table

- **Abstract data types**
  - Many instances of abstract objects may be generated
  - Example: Stack
Abstract objects: an example

- A calculator of expressions expressed in Polish postfix form
  \[ a*(b+c) \rightarrow abc+* \]

- A module implements a stack where the values of operands are shifted until an operator is encountered in the expression (assume only binary operators)
Example (cont.)

Interface of the abstract object STACK

exports:
procedure PUSH (VAL: in integer);
procedure POP_2 (VAL1, VAL2: out integer);
Design assessment

• How does the design anticipate change in type of expressions to be evaluated?
  – e.g., it does not adapt to unary operators
Abstract data types (ADTs)

• A stack ADT allows to have more than one instance of a stack module

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
ADTs

- An ADT module is a module that exports a type, along with the operations needed to access and manipulate objects of that type.

- An ADT hides the representation of the type and the algorithms used in the operations. The details of the type is dealt with in the implementation of the ADT.

- 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
An example: simulation of a gas station

module FIFO_CARS
uses CARS
exports

  type QUEUE : ? ;
  procedure ENQUEUE (Q: in out QUEUE ; C: in CARS);
  procedure DEQUEUE (Q: in out QUEUE ; C: out CARS);
  function IS_EMPTY (Q: in QUEUE) : BOOLEAN;
  function LENGTH (Q: in QUEUE) : NATURAL;
  procedure MERGE (Q1, Q2 : in QUEUE ; Q : out QUEUE);

This is an abstract data-type module representing queues of cars, handled in a strict FIFO way; queues are not assignable or checkable for equality, since ":=" and "=" are not exported.

end FIFO_CARS

Module GAS-STATION
uses FIFO_CARS
gasoline_1, gasoline_2, gasoline_3: QUEUE
car_wash: QUEUE; that_car: CARS;
ENQUEUE (car_wash, that_car);
MERGE (gasoline_1, gasoline_2, gasoline_3);
end GAS_STATION
Example of Modular Programming Using C

Modular Programming in C
By John R. Hayes, Courtesy of Embedded Systems Programming
Nov 30 2001 (8:24 AM)
URL: http://www.embedded.com/showArticle.jhtml?articleID=9900399

DEFINITION MODULE foo
EXPORT list of function declarations of exported functions and data
END foo

IMPLEMENTATION MODULE foo
IMPORT list of modules used
... code ...
END foo
Listing 1: foo.c (implementation)

/* foo.c */
/* Import needed interfaces: */
#include "x.h"
#include "y.h"

/* Implements this interface; used for compiler checks: */
#include "foo.h"

int var1;
static int var2;

void Fun1(int *p) { ... }
static void Fun2(void) { ... }

**Static** keyword is used to limit the scope of functions and data to a single file (information hiding)

NOT IN EXAM
Listing 2: foo.h (interface)

/* foo.h */
#define var1 Foo_var1
/* prefixing function name with module name */
#define Fun1 FooFun1

extern int var1;
extern void Fun1(int *);

**Extern** keyword is used to announce the publicly accessible parts of a module (interface definition). A client module can access var1 and Fun1.
Missing Proper Header File

#include <stdio.h>
main() { printf("e = %f\n", exp(1)); }

May produce e=0.0000 instead of e=2.718282

Program has a simple mistake. It is missing
#include <math.h>. The use of function prototypes
prevents such a problem. C compiler assumes that
exp(1) expected and integer argument and
returned an integer result.
Listing 3: client.c  (implementation)

/* client.c */
/* Import needed interfaces: */
#include "z.h"

/* Implements this interface: */
#include "client.h"
static void ClientFun(void)
{
    int z;
    ...
    Fun1(z);
    ...
}

An integer passed to Fun1 instead of pointer to integer. 
#include “foo.h” is missing. Client.c will compile but linker Can not find Fun1 and will send an error message
Listing 4: priqueue.h (interface)

/* priqueue.h */
#define Enqueue PriEnqueue
#define Dequeue PriDequeue
#define CreateQueue PriCreateQueue

typedef struct priority_queue_struct * Priority_queue;

extern void Enqueue(Priority_queue, int priority, void *data);
extern void *Dequeue(Priority_queue);
extern Priority_queue CreateQueue(void);

int x;
float f;
void *p = &x;  // p points to x
*(int*)p = 2;
p = &r;     // p points to r
*(float*)p = 1.1;
• Modular programming consists of separating implementation from interface and hiding information in the implementation.

• In C this is achieved by placing the interface definition in a header file and the implementation in a source file.

• Disciplined use of `static` is used to hide implementation details. The interface definition forms the link between a module and its clients.

• The module includes its own interface definition to confirm that it implements the advertised interface; a client module imports/includes the interface definition to verify that it is using the interface correctly.

• Put the constant definitions and data structure declarations in the header for inclusion by its corresponding C source file. Now, if a constant or data structure is only used by one source file, place them in that source file. If a constant or data type is used throughout an application, they belong in a traditional header file.