Announcements

- SRS Interviews
  - TODAY 1pm CSB 303 Shlomo Zilberstein
  - WED in class Peterson/Zinn (OIT) & Battisti (CCBIT)
- Office Hours
  - Tuesday (3-4)
- VOTE tomorrow!

15-Design

- Readings
  - David Parnas "On the Criteria To Be Used in Decomposing Systems into Modules," Comm. ACM 15, 12 (Dec. 1972), 1053-1058
  - David Parnas"On the design and development of program families" IEEE Trans. On SE., vol. SE-2, pp.1-9, Mar. 1976
  - Davis, A. M. "Software Requirements: Analysis and Specification". Prentice-Hall, 1990.
  - Michael Jackson, Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices (ACM Press Books) Addison-Wesley Pub Co; 1st edition (1995)
  - David Budgen, Software Design (2nd Edition) Pearson Addison Wesley; 2nd edition (2003)

History of Software Design

- 1960s
  - Structured Programming
    - ("Goto Considered Harmful", E.W.Dijkstra)
    - Emerged from considerations of formally specifying the semantics of programming languages, and proving programs satisfy a predicate.
    - Adopted into programming languages because it's a better way to think about programming
- 1970s
  - Structured Design
    - Methodology/guidelines for dividing programs into subroutines.
- 1980s
  - Modular (object-based) programming
    - Ada, Modula, Euclid, …
    - Grouping of sub-routines into modules with data.
- 1990s
  - Object-Oriented Languages started being commonly used
  - Object-Oriented Analysis and Design for guidance.

Key Word In Context

"The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters.

Any line may be 'circularly shifted' by repeatedly removing the first word and appending it at the end of the line.

The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order."

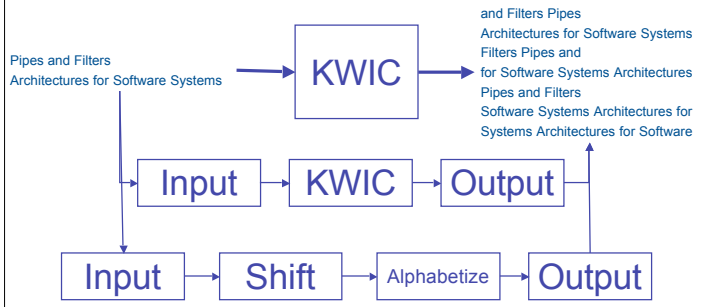
On the Criteria for Decomposing Systems into Modules. David Parnas. *CACM*, 1972

KWIC example "borrowed" from Software Architectures © David Garlan

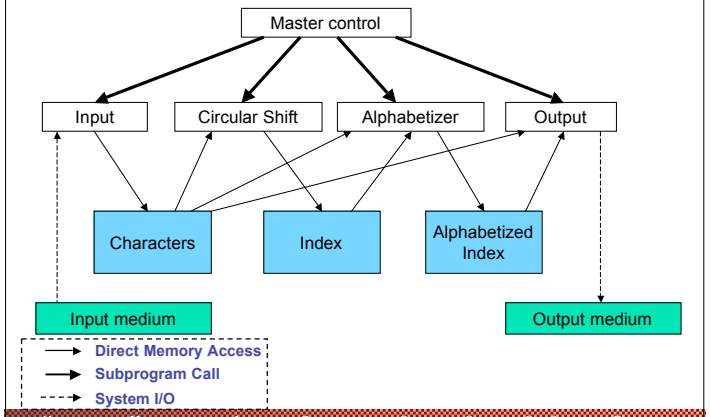
Design Considerations

- Change in Algorithm
  - e.g., batch vs. incremental
- Change in Data Representation
  - e.g., line storage, explicit vs implicit shifts
- Change in Function
  - e.g., eliminate lines starting with trivial words
- Performance
  - e.g., space and time
- Reuse
  - e.g., sorting

Stepwise Refinement Strategy



KWIC Modularization



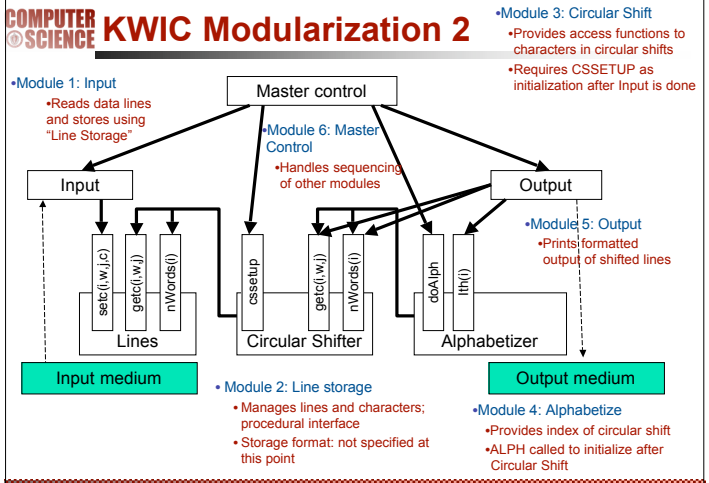
Advantages & Disadvantages

- Advantages
  - computations can share the same storage
  - allow efficient data representation
  - has a certain intuitive appeal
    - distinct computational aspects are isolated in different modules
- Disadvantages
  - serious drawbacks in terms of its ability to handle changes
    - a change in data storage format will affect almost all of the modules
    - changes in algorithm and enhancements to system function are not easily handled
  - reuse is not well-supported because each module of the system is tied tightly to this particular application.

Alternative Modularization

- Each major step in the processing ⇔ module, but now add Information hiding
  - Each module hides design decisions from all others.
    - Lines -- how characters/lines are stored
    - Circular Shifter -- algorithm for shifting, storage for shifts
    - Alphabetizer -- algorithm for alpha, laziness of alpha
- Maintain same flow of control, but organize solution around set of data managers (objects):
  - for initial lines
  - shifted lines
  - alphabetized lines
- Each manager handles the representation of the data & provides a procedural interface for accessing the data

KWIC Modularization 2



Comparisons

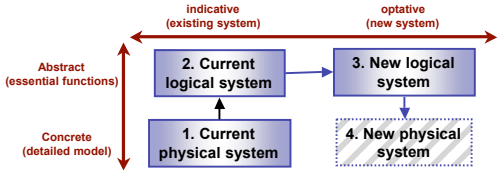
- Change in Algorithm
  - Solution 1: batch algorithm wired into
  - Solution 2: permits several alternatives
- Change in Data Representation
  - Solution 1: Data formats understood by many modules
  - Solution 2: Data representation hidden
- Change in Function
  - Solution 1: Easy if add a new phase of processing
  - Solution 2: Modularization doesn't give particular help
- Independent Development
  - Solution 1: Must design all data structures before parallel work can proceed; complex descriptions needed
  - Solution 2: Must design interfaces before parallel work can begin; simple descriptions only
- Comprehensibility
  - Which is better?

Summary

- Every architect should have a standard set of architectural styles in his/her repertoire
  - it is important to understand the essential aspects of each style: when and when not to use them
  - examples: pipe and filters, objects, event-based systems, blackboards, interpreters, layered systems
- Choice of style can make a big difference in the properties of a system
  - analysis of the differences can lead to principled choices among alternatives

Structured Design & Analysis

- Definition
  - SA&D a data-oriented approach to conceptual modeling
  - DFD central
  - Typically used for information systems, occurs in many legacy systems
- Modeling process:
  - Model of current physical system only useful as basis for the logical model
  - Distinction between indicative and optative models is very important

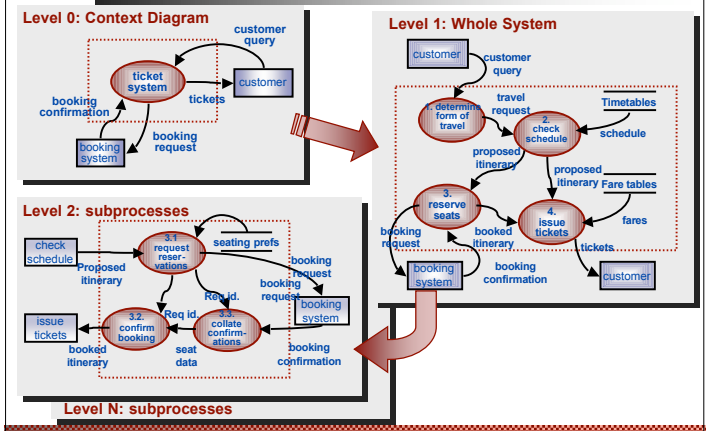


Modeling tools

- Data flow diagram
  - Context diagram ("Level 0")
    - whole system as a single process
  - Intermediate level DFDs decompose each process
  - Functional primitives are processes that cannot be decomposed further
- Data dictionary
  - Defines each data element and data group
  - Use of BNF to define structure of data groups
- Primitive Process Specification
  - Each functional primitive has a "mini-spec"
  - These define its essential procedural steps
  - Expressed in English narrative, or some form of pseudo-code
- Structured Walkthrough

Source: Adapted from Svoboda, 1990, p258-263

Hierarchies of DFDs



Data Dictionary & Process Specs

Example Data Dictionary

Mailing Label =
  customer_name +
  customer_address
customer_name =
  customer_last_name +
  customer_first_name +
  customer_middle_initial
customer_address =
  local_address +
  community_address + zip_code
local_address =
  house_number + street_name +
  (apt_number)
community address =
  city_name + [state_name |
  province_name]

Example Mini-Spec

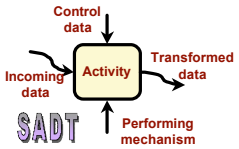
FOR EACH Shipped-order-detail
  GET customer-name + customer-address
  FOR EACH part-shipped
    GET retail-price
    MULTIPLY retail-price by quantity-shipped
    TO OBTAIN total-this-order
CALCULATE shipping-and-handling
ADD shipping-and-handling TO total-this-order
  TO OBTAIN total-this-invoice
PRINT invoice

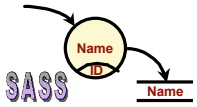
Source: Adapted from Svoboda, 1990, p262-4

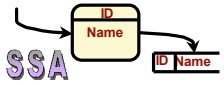
DFD variants

Source: Adapted from Svoboda, 1990, p264-5

- Structured Analysis and Design Technique (SADT)
  - Developed by Doug Ross in the mid-70's
  - Uses activity diagrams rather than dataflow diagrams
  - Distinguishes control data from processing data
- Structured Analysis and System Specification (SASS)
  - Developed by Yourdon and DeMarco in the mid-70's
  - 'classic' structured analysis
- Structured System Analysis (SSA)
  - Developed by Gane and Sarson
  - Notational style slightly different from Yourdon & DeMarco
  - Adds data access diagrams to describe contents of data stores
- Structured Requirements Definition (SRD)
  - Developed by Ken Orr in the mid-70's
  - Introduces the idea of building separate models for each perspective and then merging them







Evaluation of SA&D techniques

- Advantages
  - Facilitate communication.
  - Notations are easy to learn, and don't require software expertise
  - Clear definition of system boundary
  - Use of abstraction and partitioning
  - Automated tool support
    - e.g. CASE tools provide automated consistency checking
- Disadvantages
  - Little use of projection
    - even SRD's 'perspectives' are not really projection
  - Confusion between modeling the problem and modeling the solution
    - most of these techniques arose as design techniques
  - These approaches model the system, but not its application domain
  - Timing & control issues are completely invisible
    - although extensions such as Ward-Mellor attempt to address this

Source: Adapted from Davis, 1990, p174

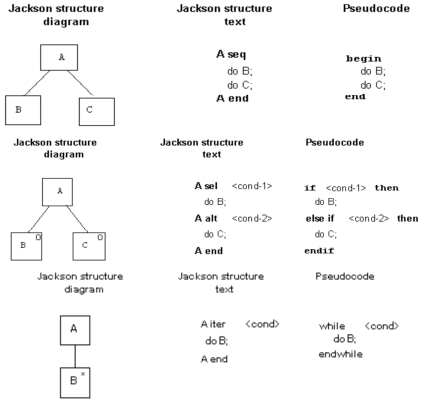
JSP & JSD

- Jackson System Development
  - Emphasis on high-level conceptual design
  - Develops collection of coordinated graphical depictions of system
  - Strong hints about how to carry them to implementation decisions
  - Strong suggestions about how to go about doing this
- Jackson Structured Programming
  - JSD Based on/uses JSP, so let's look at that first

JSP

- Design is about structure, about the relation of parts to the whole.
- Programs consist of the following parts or components:
  - elementary components
  - three types of composite components -- components having one or more parts:
    - **sequence** -- a sequence is a composite component that has two or more parts occurring once each, in order.
    - **selection** -- a composite component that consists of two or more parts, only one of which is selected, once.
    - **iteration** a composite component that consists of one part that repeats zero or more times.

composite components



Basic program design method (JSP)

- system diagram
- input/output structure diagrams
- program structure diagram
- allocation of operations to program structure
  - which part? how many times?
  - "read-ahead rule"
- constructive method of design
  - not top-down, not stepwise refinement

JSP design method

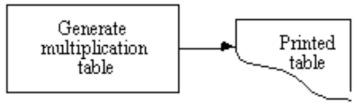
- consists of the following steps:
  1. Draw a system diagram
  2. Draw a data structure for each input and output file
  3. Draw a single data structure based on correspondences between the input and output data structures; this data structure forms the basic program structure
  4. List the operations needed by the program, For each, ask "Where does it belong (in what program part?)" "How many times does it occur?" Allocate the operations to the basic program structure.
  5. Translate the program structure into text, specifying the conditions for iteration and selection

multiplication table example

- The required output is:

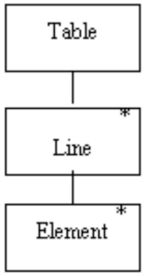
  1
  2 4
  3 6 9
  4 8 12 16
  ... ...
  10 20 30 40 50 60 70 80 90 100

1. Draw system diagram

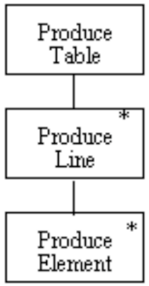


multiplication table example

2. Draw data structures



3. Form program structure based on the data structures from the previous step.



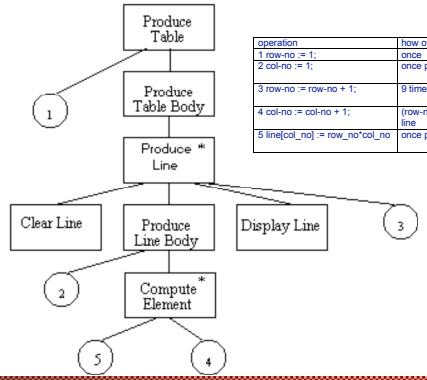
multiplication table example

4. List and allocate operations
   - elementary operations needed to perform the task, and for each operation
     - "How often is it executed?"
     - "In what program component(s) does it belong?"
   - The operations must be elementary statements of some programming language; e.g., Pascal.

operation	how often?	where?
1 row-no := 1;	once	at start of program
2 col-no := 1;	once per line	in part that produces a line, at start
3 row-no := row-no + 1;	9 times	in part that produces a line
4 col-no := col-no + 1;	(row-no)-1 per line	in part that computes an element
5 line[col_no] := row_no*col_no	once per element	in part that computes an element

multiplication table example

5. Code program from structure diagram or structure text



operation	how often?	where?
1 row-no := 1;	once	at start of program
2 col-no := 1;	once per line	in part that produces a line, at start
3 row-no := row-no + 1;	9 times	in part that produces a line
4 col-no := col-no + 1;	(row-no)-1 per line	in part that computes an element
5 line[col_no] := row_no*col_no	once per element	in part that computes an element

Difficulties in applying JSP

- The development procedures of a method should be closely matched to specific properties of the problems it can be used to solve
- basic JSP requires the problem to possess at least these two properties:
  - the data structures of the input and output files, and the correspondences among their data components, are such that a single program structure can embody them all
  - each input file can be unambiguously parsed by looking ahead just one record
- If the file structures do not correspond appropriately it is impossible to design a correct program structure: this difficulty is called a **structure clash**
- If an input file can not be parsed by single look ahead it is impossible to write all the necessary conditions on the program's iterations and selections: this is a **recognition difficulty**

Structure Clashes

- three kinds of structure clash
  - interleaving clash
    - data groups that occur sequentially in one structure correspond functionally to groups that are interleaved in another structure
    - e.g., the input file of a program may consist of chronologically ordered records of calls made at a telephone exchange; the program must produce a printed output report of the same calls arranged chronologically within subscriber. The 'subscriber groups' that occur successively in the printed report are interleaved in the input file
  - ordering clash
    - corresponding data item instances are differently ordered in two structures
    - e.g., an input file contains the elements of a matrix in row order, and the required output file contains the same elements in column order.
  - boundary clash,
    - two structures have corresponding elements occurring in the same order, but the elements are differently grouped in the two structures; the boundaries of the two groupings are not synchronized.

Boundary clashes

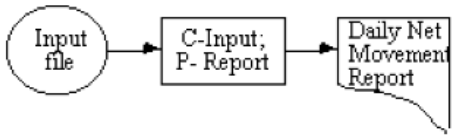
- are surprisingly common
- three well-known examples:
  - The calendar consists of years, each year consisting of a number of days. In one structure the days may be grouped by months, but by weeks in another structure. There is a boundary clash here: the weeks and months can not be synchronized.
  - A chapter of a printed book consists of text lines. In one structure the lines may be grouped by paragraphs, but in another structure by pages. There is a boundary clash because pages and paragraphs can not be synchronized.
  - A file in a low-level file handling system consists of variable-length records, each consisting of between 2 and 2000 bytes. The records must be stored sequentially in fixed blocks of 512 bytes. There is a boundary clash here: the boundaries ofthe records can not be synchronized with the boundaries of the blocks

Program decomposition

- Example of a "structure clash"
  - an inventory transaction file consists of daily transactions sorted by part number
  - each part number may have one or more transactions
    - either a receipt into the warehouse or an order out of the warehouse
  - each transaction contains a transaction code, a part-identifier, and a quantity received or ordered
  - A program is to be written that prints a line for each part number showing the net daily movement for that part number into or out of the warehouse
    - Assumption: the input file is blocked, with each block containing a record count followed by a number of records

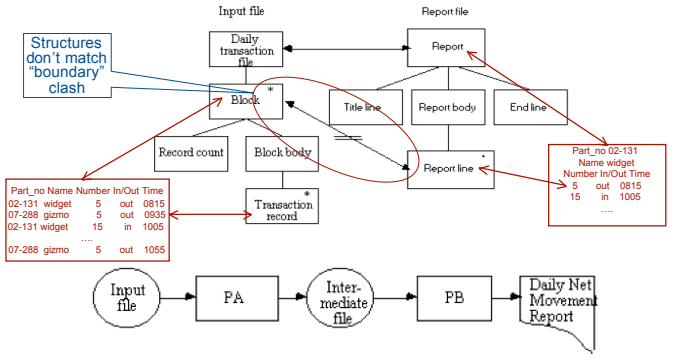
Report generation

1. Draw system diagram



Report generation example

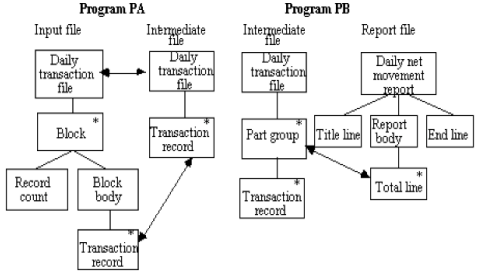
2. Draw data structures



Report generation example

- Program PA consumes the input file of blocks of records and produces an unblocked file of transactions. Program PB consumes the groups of unblocked transaction records and produces the required report.

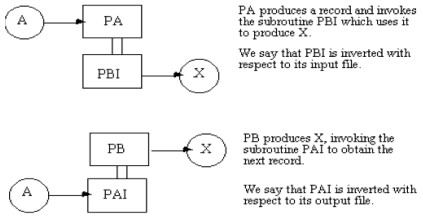
2. Draw data structures



Resolution of the structure clash

- decomposing the program P into two programs, PA and PB as shown below:

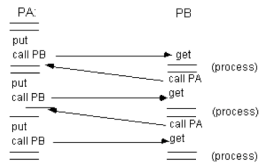
- program inversion



Resolution of the structure clash

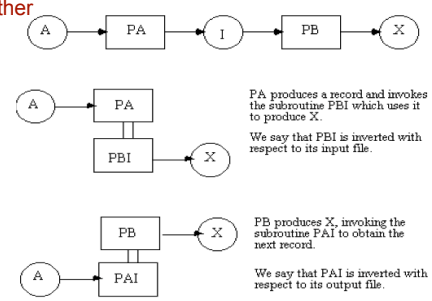
- decomposing the program P into two programs, PA and PB as shown below:

- Implementation
  - Batch processing: PA produces the serial data stream, I, which is then processed by program PB.
  - Parallel processing: cooperating programs or coroutines or as independent tasks under control of a multi-programming task supervisor



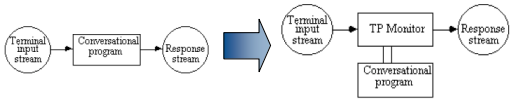
Program Inversion

- solution of a structure clash more cheaply than employing Multi-programming
  - convert one program that it runs as a subroutine of the other

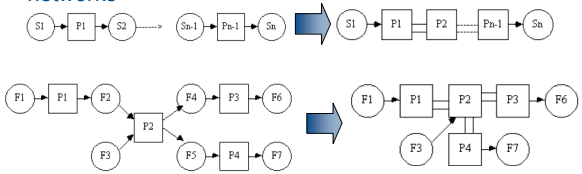


Uses of program inversion

- Interactive conversational programs



- Interrupt handler
- Implementation of pipes & filters and hierarchical networks

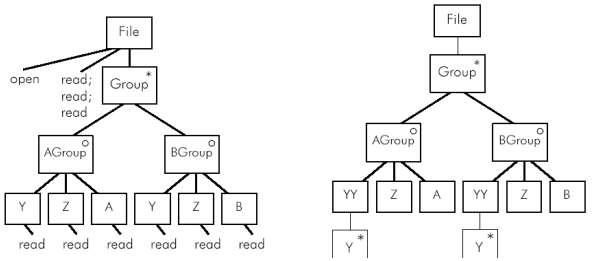


Significance of Inversion

- many situations appearing in their dynamic, piecemeal executable form can be recast in their underlying serial form as a simple program
  - any resumable program--one that is alternately activated and suspended--is an example of inversion
- what is the underlying seriality of its input and output?
  - can recast the problem in serial form, and design a simple program using JSP
  - can optimize the design using inversion
  - inversion preserves program correctness--it is an algorithmic transformation--we can be confident about the design of the inverted (resumable) program
- inversion allows us to extend the range of JSP to many situations that at first glance do not appear to be amenable to it

Recognition Difficulties

- A recognition difficulty is present when an input file can not be unambiguously parsed by single look-ahead
  - sometimes the difficulty can be overcome by looking ahead two or more records
  - sometimes a more powerful technique is necessary

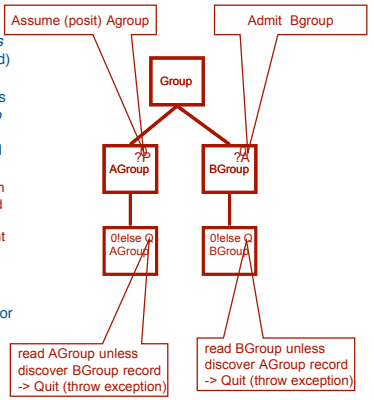


Backtracking technique

1. the recognition difficulty is simply ignored. The program is designed, and the text for the AGroup and BGroup components is written, as usual. No condition is written on the Group selection component. The presence of the difficulty is marked only by using the keywords **posit** and **admit** in place of if and else.
2. a **quit** statement is inserted into the text of the posit AGroup component at each point at which it may be detected that the Group is, in fact, not an AGroup. In this example, the only such point is when the B record is encountered. The quit statement is a tightly constrained form of GO TO: its meaning is that execution of the AGroup component is abandoned and control jumps to the beginning of the admit BGroup component.
3. the program text is modified to take account of side-effects: that is, of the side-effects of operations executed in AGroup before detecting that the Group was in fact a BGroup.

Sketch

- The component *attempt to read Agroup records* **posits** *assume you can read Agroup records* and **admits** that you (cannot & should) *instead read Bgroup records*
- The *attempt to read Agroup records* is a call of the subprogram *read Agroup records* which may cause an implicit **quit** when, in reading, it is discovered the record is a BGroup record
  - an implicit quit is an exception thrown within a subprogram and not handled and so is propagated from the subprogram to its calling environment
- a posit/ admit design must contain a single posit and at least one admit connected at the same level
- it can contain any number of, implicit or explicit, quits within the admit component at any level



Central virtues of JSP

- it provides a strongly systematic and prescriptive method for a clearly defined class of problem
  - independent JSP designers working on the same problem produce the same solution
- JSP keeps the program designer firmly in the world of static structures to the greatest extent possible.
  - only in the last step of the backtracking technique, when dealing with side-effects, is the JSP designer encouraged to consider the dynamic behavior of the program
  - this restriction to designing in terms of static structures is a decisive contribution to program correctness for those problems to which JSP can be applied
  - avoids the dynamic thinking -- the mental stepping through the program execution -- that has always proved so seductive and so fruitful a source of error.
- Hints
  - Don't optimize!!If you have to, do it as the last step, after you have designed the program properly.
  - Use Models not functions

Jackson System Development (JSD)

- Emphasis on high-level conceptual design
- Develops collection of coordinated graphical depictions of system
- Strong hints about how to carry them to implementation decisions
- Strong suggestions about how to go about doing this
- Considerable literature delving into the details of JSD
- Product of a commercial company
- Supported by courses, tools, consultants

JSD Models Focus on Actions

- JSD produces models of the real world and the way in which the system to be built interacts with it
- Primary focus of this is actions (or events)
  - actions can have descriptive attributes
  - set of actions must be organized into set of processes
- Processes describe which actions must be grouped together and what the "legal" sequences of actions are
  - Processes can overlap in various ways
  - Processes are aggregated into an overall system model
  - using two canonical models of inter-process communication
- Data are described in the context of actions
  - in JSD data considerations are subordinate to actions

JSD - Phases

- the modeling phase
  - Entity/action step
  - Entity structure step
  - Model process step
- the network phase
  - connect model processes and functions in a single system specification diagram (SSD)
- implementation phase
  - examine the timing constraints of the system
  - consider possible hardware and software for implementing our system
  - design a system implementation diagram (SID)

Student Loan Example

- Functional requirements:
  - before getting a loan, there is an evaluation process after which agreement is always reached
    - a **TE transaction** records each step of the evaluation process
    - a **TA transaction** records the overall loan agreement
  - a student can take any number of loans, but only one can be active at any time
    - each loan is initiated by a **TI transaction**
  - the student repays the loan with a series of repayment
    - each repayment transaction is recorded by a **TR transaction**
  - a loan is terminated by a **TT transaction**
  - two output functions are desired:
    - an inquiry function that prints out the loan balance for any student,
    - a repayment acknowledgment sent to each student after payment is received by the university
- Non Functional requirements
  - to be implemented on a single processor
  - inquiries should be processed as soon as they are received
  - repayment acknowledgments need only be processed at the end of each day.
  - Note: generates a stream of data over a long-period of time

Step 1: Entity/action step

- Actions have the following characteristics:
  - an action takes place at a point in time
  - an action must take place in the real world outside of the system.
  - an action is atomic, cannot be divided into subactions.
- Entities have the following characteristics:
  - an entity performs or suffers actions in time.
  - an entity must exist in the real world, and not be a construct of a system that models the real world
  - an entity must be capable of being regarded as an individual; and, if there are many entities of the same type, of being uniquely named.

Candidates

- Entities/Description:
  - student
  - system
  - university
  - loan
  - student-loan

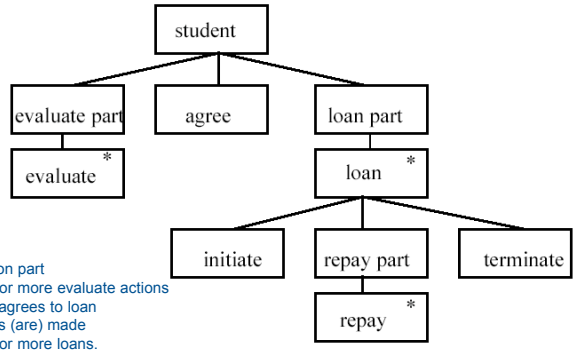
Actions/Attributes:

- **evaluate** -action of university? (university performs the evaluation); action of student? (student is evaluated)
  - attributes: student-id, loan-no, date of evaluation, remarks
- **agree** - action of university? (university agrees to loan); action of student ? (agrees to loan)
  - attributes: student-id, loan-no, date of agreement, amount of loan, interest rate, repayment period)
- **make loan** - action of university
  - attributes: student-id, loan-no, date of loan, loan amount, interest rate, repayment period
- **initiate** - action of university? (university initiates loan); action of student? (student initiates loan); action of loan? (is initiated)
  - attributes: student-id, date initiated
- **repay** - action of loan? (loan is repaid); action of student? (student repays the loan);
  - attributes: student-id, date of repayment, amount of repayment
- **terminate** - action of loan (loan is terminated);
  - attributes: student-id, date of termination, remarks

Focus on:

- Entities/Description:
  - student
- Actions/Attributes:
  - evaluate -action of student; student? (student suffers the action, is evaluated);
    - attributes: student-id, loan-no, date of evaluation, remarks
  - agree - action of student
    - attributes: student-id, loan-no, date of agreement, amount of loan, interest rate, repayment period)
  - initiate - action of student
    - attributes: student-id, date initiated
  - repay - action of student
    - attributes: student-id, date of repayment, amount of repayment
  - terminate - action of student
    - attributes: student-id, date of termination, remarks

Step 2: Entity structure step



(1) evaluation part
- zero or more evaluate actions

(2) student agrees to loan

(3) loan(s) is (are) made
- zero or more loans.
- loan is a sequence of initiate action, iteration of repay actions, a terminate action

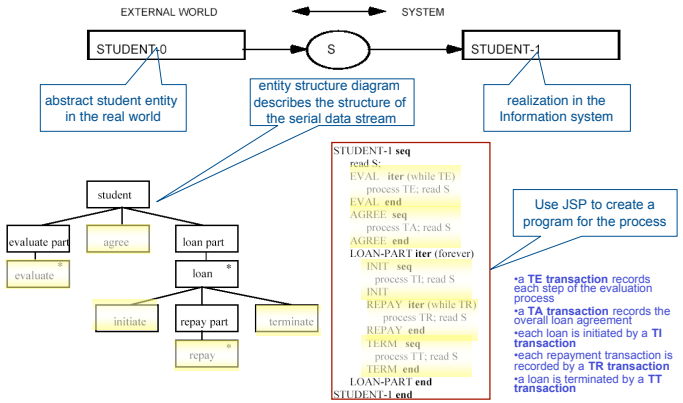
Model Process

- Primary building block of a JSD design
  - contains all actions characterizing a key real-world process
- Actions are structured into a tree
  - only the leaf nodes of the tree are real-world actions
  - interior nodes are conceptual
  - interior nodes can be annotated to show choice or iteration
  - traversals of this tree constitute the only "legal" sequences of actions for this process
- Model process tree defines a regular expression
  - set of traversals is a regular set

Model Processes

- A model process is a particular view of the system
  - various model processes provide different views
  - model process is multiply instantiated for different instances
  - model processes are often annotated with informal specifications and notations
  - same action may appear as part of more than one process
- Model Processes and Data
  - actions on data hang off of model process leaf nodes
  - global data is necessary too
    - for functions that must combine data from >1 model process
    - to assure consistency between model processes
    - to coordinate between different instances of the same model process
    - to coordinate between different models of the same entity

Step 3: Model process step

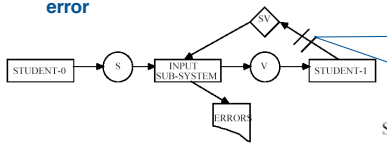


Error handling

- a real-time system (but slow-running) system
- information is collected as it arrives from the real-world
- entity model process is synchronized with the actions of the real world entity
- the state vector of a model process's "program" has a "counter" … and if it "points" to repay component of a student's process, then an 'E' (evaluate), 'A' (agree) or 'I' (initiate) transaction **must be recognized as an error**

```
STUDENT-1 seq
    read S;
    EVAL   iter (while TE)
        process TE; read S
    EVAL  end
    AGREE  seq
        process TA; read S
    AGREE  end
    LOAN-PART iter (forever)
        INIT   seq
            process TI; read S
        INIT
        REPAY  iter (while TR)
            process TR; read S
```

counter



state vector (SV) connection -- one process can examine the SV of a 2nd process -- the double lines indicate that an inquiry process, over its life, will examine many student processes

Total System Model

- At the Network Phase, weave Model Processes together incrementally to form the total system specification
  - also add new processes during this phase: e.g., input, output, user interface, data collection
- Goal is to indicate how model processes communicate with each other, use each other, are connected to user and outside world
- Linkage through two types of communication:
  - Message passing
  - State vector inspection
- Indicates which data moves between which processes
  - and more about synchronization

Model Process Communication

- Fundamental notion is Data Streams
  - can have multiple data streams arriving at an action in a process
  - can model multiple instances entering a data stream or departing from one
- Two types of data stream communication:
  - asynchronous message passing
  - State vector inspection
- These communication mechanisms used to model how data is passed between processes

Message Passing

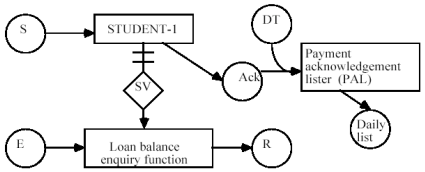
- Data stream carries a message from one process activity to an activity in another process
  - must correlate with output leaf of sending model process
  - must correlate with input leaf of receiving model process
- Data transfer assumed to be asynchronous
  - less restrictive assumption
  - no timing constraints are assumed
  - messages are queued in infinitely long queues
  - messages interleaved non-deterministically when multiple streams arrive at same activity

State Vector Inspection

- Modeling mechanism used when one process needs considerable information about another
- State vector includes
  - values of all internal variables
  - execution text pointer
- Process often needs to control when its state vector can be viewed
  - process may need exclusive access to its vector
- Could be modeled as message passing, but important to underscore characteristic differences

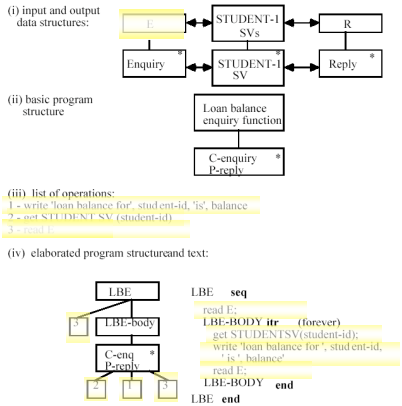
Network Phase -- the SSD

- loan balance inquiry function (LBE) is connected to the Student-1 process by state vector (SV) connection
- The function to produce the student acknowledgments data stream (ACK) is embedded in the student-1 process in the repays component



- DT is an input signal at the end of the day--a daily time marker--that tells the payment acknowledgment lister (PAL) function to begin
- The ACK and DT data streams are rough-merged, that is, we don't know precisely whether a repayment acknowledgment will appear on today's or tomorrow's daily list.

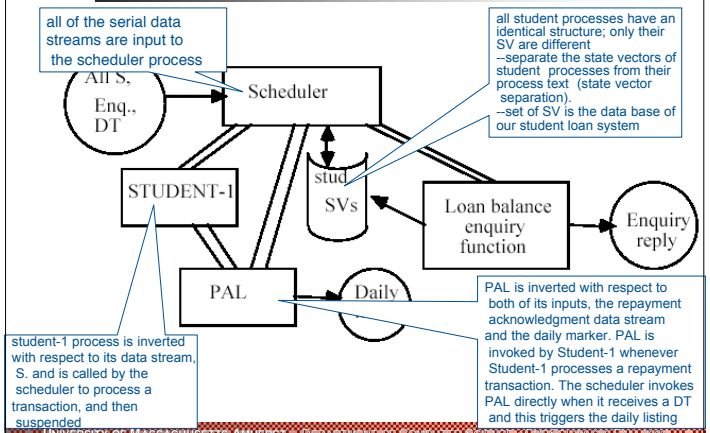
Designing the LBE function w/ JSP



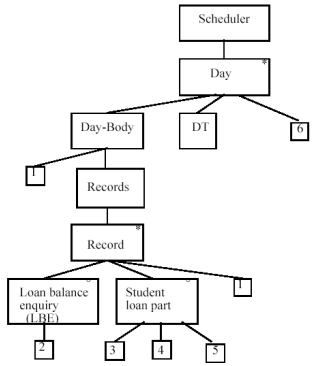
Implementation Phase

- Use of inferences encouraged by understandings gleaned from the network phase
- Network Phase suggests ideal traversal paths through model processes and their local data
  - suggests internal implementation of model processes
  - studying use of model processes suggests internal structure of their data
- Communication by data streams and state vector inspection often suggest real implementations
  - But often not

The SID



Design of the scheduler in JSP



- records from the serial data stream (loan balance inquiries and student loan transactions) are read and processed in real-time
- at the end of the day, a daily time marker--perhaps a signal to the system from the operator--is input

List of operations:

1-read input
2-call LBE(inrec)
3-get SSV(student-id)
4-call student-1(srec, ssv)
5-put SSV(student-id)
6-call PAL(DT)

- PAL is invoked & processes payment acknowledgments that have been previously generated in real-time whenever a student repayment is made and stored in a buffer

JSD and JSP

- In JSD, the principles of JSP are extended into the areas of systems analysis, specification, design and implementation
- In JSP, a simple program describes a sequential process that communicates by means of sequential data streams; its structure is determined by the structure of its input and output data stream
- In JSD, the real world is modeled as a set of sequential model processes that communicate with the real world and with each other by sequential data streams (as well as by a second read-only communication called state vector connection). The structure of a model process is determined by the structure of its inputs and outputs.
- The JSD implementation step embodies the JSP implementation technique, program inversion, in which a program is transformed into a procedure
- Other JSP techniques, such as the single read-ahead rule and backtracking, and principles, such as implementation through transformation, are used in JSD