18- Architecture, Frameworks, Components, Patterns, Middleware + Design: JSP/JSD

Rick Adrion

Software Architectures

- Architectural taxonomy ("boxology")
- Architectural patterns & idioms
- Design patterns & idioms
- Reuse
  - Class libraries
  - Components
  - Frameworks
  - Middleware

Architectural taxonomy ("boxology")

- dataflow
  - batch sequential
  - data flow network
  - pipes & filter
- call/return
  - main program/subroutines
  - abstract data types
  - objects
  - call based client/server
  - layered
- independent components
  - communicating processes
  - distributed
  - event systems (implicit, explicit)
- virtual machine
  - interpreter
  - rule-based
- data-centered
  - repository
  - blackboard

can decompose into sequential stages
involves transformations on continuous (or on very long streams) streams of data

flexibility, configurability, loose coupling
hierarchies, producer-consumer, tightly connected

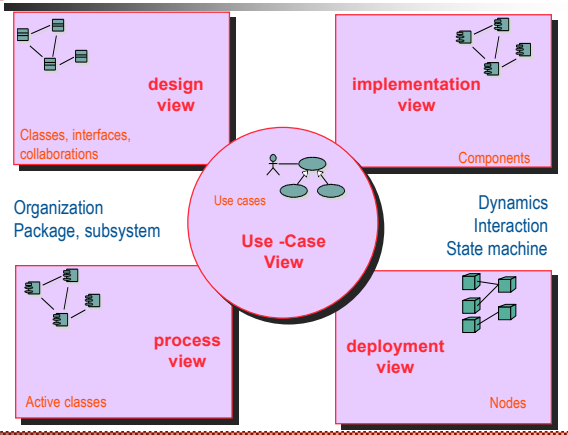
cross-platform
late decision on hardware

focus on management and representation of data
long-lived (persistent) data is focus on repositories
stream of incoming requests to access highly structured data
changing data
"noisy" input data, uncertain execution order can not be predetermined, consider a blackboard

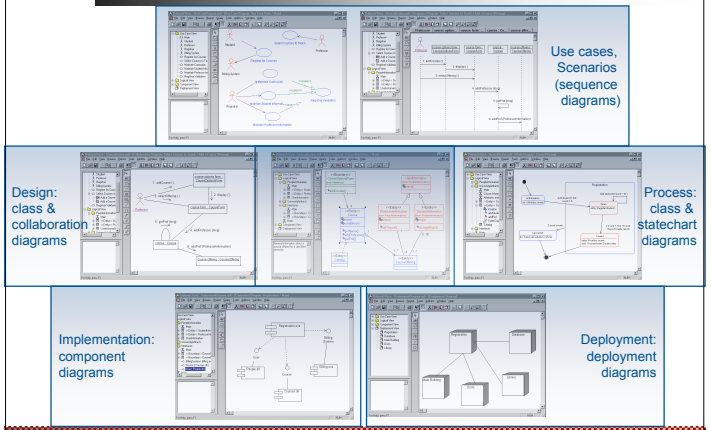
Taxonomy of Patterns & Idioms

Type	Description	Examples
Idioms	**Restricted to a particular language, system, or tool**	**Scoped locking**
Design patterns	**Capture the static & dynamic roles & relationships in solutions that occur repeatedly**	**Active Object, Bridge, Proxy, Wrapper Façade, & Visitor**
Architectural patterns	**Express a fundamental structural organization for software systems that provide a set of predefined subsystems, specify their relationships, & include the rules and guidelines for organizing the relationships between them**	**Half-Sync/Half-Async, Layers, Proactor, Publisher-Subscriber, & Reactor**
Optimization principle patterns	**Document rules for avoiding common design & implementation mistakes that degrade performance**	**Optimize for common case, pass information between layers**

The Rational 4+1 Views



The Rational 4+1 Views



Architecture Description Languages

- formal notations for representing and analyzing architectural designs
- provide both a conceptual framework and a concrete syntax for characterizing software architectures
- tools for parsing, displaying, compiling, analyzing, or simulating architectural descriptions.

ADL Examples

- Adage
  - supports the description of architectural frameworks for avionics navigation and guidance
- Aesop
  - supports the use of architectural styles
- C2
  - supports the description of user interface systems using anevent-based style
- Darwin
  - supports the analysis of distributed message-passing systems
- Meta-H
  - provides guidance for designers of real-time avionics control software;
- Rapide
  - allows architectural designs to be simulated, and has tools for analyzing the results of those simulations;
- SADL
  - provides a formal basis for architectural refinement;
- UniCon
  - has a high-level compiler for architectural designs that supports a mixture of heterogeneous component and connector types;
- Wright
  - supports the formal specification and analysis of interactions between architectural components.

formal architectural specification.

- module interconnection languages
  - static aspects of component interaction
  - definition and use of types, variables, and functions among components
  - examples: INTERCOL, PIC, CORBA/IDL
- process algebras
  - dynamic interplay among components
  - concerned with the protocols by which components communicate
  - examples: Wright (based on CSP), Chemical Abstract Machine (based on term rewriting)
- event languages
  - identification and ordering of events
  - event is a very flexible, abstract notion
  - example: Rapide

Evaluation & analysis

- conduct a formal review with external reviewers
  - time the evaluation to best advantage
  - choose an appropriate evaluation technique
  - create an evaluation contract
  - limit the number of qualities to be evaluated
  - insist on a system architect
- benefits
  - financial
  - increased understanding and documentation of the system
  - detection of problems with the existing architecture
  - clarification and prioritization of requirements
  - organizational learning

Benefits

- examples
  - AT&T
    - 10% reduction in project costs, on projects of 700 staff days or longer, the evaluation pays for itself.
  - consultants
    - reported 80% repeat business, customers recognized sufficientvalue
  - where architecture reviews did not occur
    - customer accounting system estimated to take two years, took seven years, re-implemented three times, performance goals never met
    - large engineering relational database system, performance made integration testing impossible, project was cancelled after twenty million dollars had been spent.

Architecture vs Frameworks

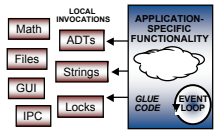
- Frameworks
  - an object-oriented reuse technique
  - used successfully for some time & are an important part of the culture of long-time object-oriented developers,
  - BUT they are not well understood outside the object-oriented community and are often misused
- Question:
  - are frameworks mini-architectures, large-scale patterns, or they are just another kind of component?
- Definitions
  - a framework is a **reusable design** of all or part of a system that is **represented by a set of abstract classes** and the way their instances interact
  - a framework is the skeleton of an application that can be customized by an application developer

Ralph E. Johnson, "Frameworks= (Components+Patterns)."**Communications of the ACM,** October 1997/Vol. 40, No. 10

Frameworks & Class Libraries

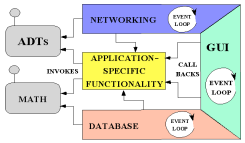
- developers often do not even know they are using a framework, but refer to a "class library"
- frameworks differ from other class libraries by reusing high-level design
  - more to learn before a class can be reused
  - can never be reused in isolation; typically a set of classes must be learned at once
- you can often tell that a class library is a framework if there are dependencies among its components and if programmers who are learning it complain about its complexity.

Frameworks & Class Libraries



Class Library Architecture

- A class is a unit of abstraction & implementation in an OO programming language



Framework Architecture

- A framework is an integrated set of abstract classes that can be customized for instances of a family of applications

Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

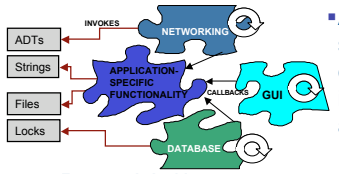
Components & frameworks

- Frameworks
  - were originally intended to be reusable components
    - but reusable O-O components have not found a market
  - are a component in the sense that
    - venders sell them as products
    - an application might use several frameworks.
  - BUT
    - they more customizable than most components
    - have more complex interfaces
      - must be learned before the framework can be used
- a component represents **code reuse**, while frameworks are a form of **design reuse**

Components & frameworks

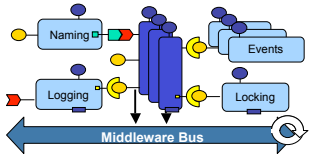
- frameworks
  - provide a reusable context for components
  - provide a standard way for components to handle errors, to exchange data, and to invoke operations on each other
    - "component systems'' such as OLE, OpenDoc, and Beans, are really frameworks that solve standard problems that arise in building compound documents and other composite objects. make it easier to develop new components
  - enable making a new component (such as a user interface) out of smaller components (such as a widget)
  - provide the specifications for new components and a template for implementing them.
- a good framework can reduce the amount of effort to develop customized applications by an order of magnitude

Frameworks & Components



Framework Architecture

- A framework is an integrated set of abstract classes that can be customized for instances of a family of applications



Component Architecture

- A component is an encapsulation unit with one or more interfaces that provide clients with access to its services

Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

Comparison

Class Libraries	Frameworks	Components
Micro-level	Meso-level	Macro-level
Stand-alone language entities	"Semi-complete" applications	Stand-alone composition entities
Domain-independent	Domain-specific	Domain-specific or Domain-independent
Borrow caller's thread	Inversion of control	Borrow caller's thread

Frameworks as Reusable Design

- Are they like other techniques for reusing high-level design, e.g., templates or schemas?
- templates or schemas
  - usually depend on a special purpose design notation
  - require special software tools
- frameworks
  - are expressed in a programming language
  - makes them easier for programmers to learn and to apply
  - no tools except compilers
  - can gradually change an application into a framework
  - because they are specific to a programming language, some design ideas, such as behavioral constraints, cannot be expressed well

Frameworks and domain-specific architectures

- A framework is ultimately an object-oriented design, while a domain-specific architecture might not be.
- A framework can be combined with a domain-specific language by translating programs in the language into a set of objects in a framework
  - window builders associated with GUI frameworks are examples of domain-specific visual programming languages
- Uniformity reduces the cost of maintenance
  - GUI frameworks give a set of applications a similar look and feel
  - using a distributed object framework ensures that all applications can communicate with each other.
  - maintenance programmers can move from one application to the next without having to learn a new design

Overview of Patterns

- Patterns
  - present solutions to common software problems arising within a certain context
  - help resolve key software design issues
    - Flexibility, Extensibility, Dependability, Predictability, Scalability,Efficiency
  - capture recurring structures & dynamics among software participants to facilitate reuse of successful designs
  - codify expert knowledge of design strategies, constraints and best practices

software patterns

- record experience of good designers
  - describe general, recurring design structures in a pattern-like format
  - problem, generic solution, usage
- solutions (mostly) in terms of O-O models
  - crc-cards; object-, event-, state diagrams
  - often not O-O specific
- patterns are generic solutions; they allow for design and implementation variations
  - the solution structure of a pattern must be “adapted” to your problem design
  - map to existing or new classes, methods, ...
    - a pattern is not a concrete reusable piece of software!

qualities of a pattern

- encapsulation and abstraction
  - each pattern encapsulates a well-defined problem and its solution in a particular domain
  - serve as abstractions which embody domain knowledge and experience
- openness and variability
  - open for extension or parametrization by other patterns so that they may work together
- generativity and composability
  - generates a resulting context which matches the initial context of one or more other patterns in a pattern language
  - applying one pattern provides a context for the application of the next pattern.
- equilibrium
  - balance among its forces and constraints

Taxonomy of Patterns & Idioms

Type	Description	Examples
Idioms	**Restricted to a particular language, system, or tool**	**Scoped locking**
Design patterns	**Capture the static & dynamic roles & relationships in solutions that occur repeatedly**	**Active Object, Bridge, Proxy, Wrapper Façade, & Visitor**
Architectural patterns	**Express a fundamental structural organization for software systems that provide a set of predefined subsystems, specify their relationships, & include the rules and guidelines for organizing the relationships between them**	**Half-Sync/Half-Async, Layers, Proactor, Publisher-Subscriber, & Reactor**
Optimization principle patterns	**Document rules for avoiding common design & implementation mistakes that degrade performance**	**Optimize for common case, pass information between layers**

Frameworks and Patterns

- frameworks represent a kind of pattern
  - e.g., Model/View/Controller is a user-interface framework often described as a pattern
  - applications that use frameworks must conform to the frameworks' design and model of collaboration, so the framework causes patterns in the applications that use it.
- frameworks are at a different level of abstraction than patterns
  - frameworks can be embodied in code, but only examples of patterns can be embodied in code.
  - a strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly
  - in contrast, design patterns have to be implemented each time they are used.

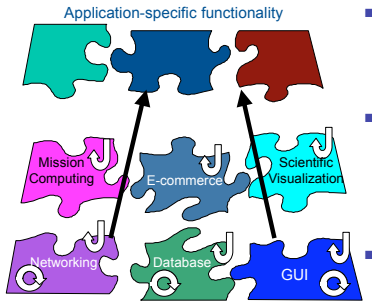
Frameworks and Patterns

- design patterns are smaller architectural elements than frameworks
  - a typical framework contains several design patterns but the reverse is never true
  - design patterns are the micro-architectural elements of frameworks.
    - e.g., Model/View/Controller can be decomposed into three major design patterns, and several less important ones
    - MVC uses the Observer pattern to ensure the view's picture of the model is up-to-date, the Composite pattern to nest views, and the Strategy pattern to cause views to delegate responsibility for handling user events to their controller.
- design patterns are less specialized than frameworks.
  - frameworks always have a particular application domain.
  - design patterns can be used in nearly any kind of application.
  - more specialized design patterns are certainly possible, even these wouldn't dictate an application architecture

Frameworks

- are firmly in the middle of reuse techniques.
- are more abstract and flexible than components,
- are more concrete and easier to reuse than a pure design (but less flexible and less likely to be applicable)
- are more like techniques that reuse both design and code, such as application generators and templates.
- can be thought of as a more concrete form of a pattern
  - patterns are illustrated by programs, but a framework is a program

Framework Characteristics

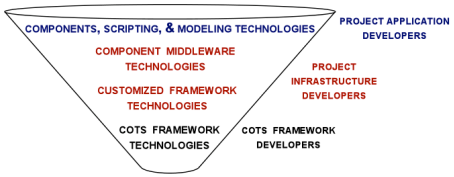


- Frameworks exhibit "inversion of control" at runtime via callbacks
- Frameworks provide integrated domain-specific structures & functionality
- Frameworks are "semi-complete" applications

Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

Using Frameworks Effectively

- Frameworks are powerful, but hard to develop & use effectively by application developers
- It's often better to use & customize COTS frameworks than to develop in-house frameworks
- Components are easier for application developers to use, but aren't as powerful or flexible as frameworks
- Successful projects are often organized using the "funnel" model

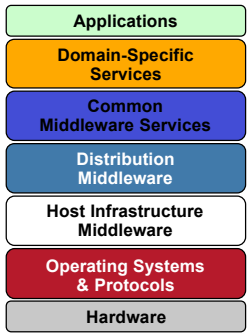


Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

Relation to Middleware

- one of the strengths of frameworks is that they are represented by traditional object-oriented programming languages.
- BUT, this is also a weakness of frameworks, however, and it is one that the other design-oriented reuse techniques do not share.
- Middleware
  - COM, CORBA, etc. address this problem, since they let programs in one language interoperate with programs in another
- Other approaches
  - some frameworks have been implemented twice so that users of two different languages can use them, such as the SEMATECH CIM framework

Evolution of Middleware



- Historically, mission-critical apps were built directly atop hardware & OS
  - tedious, error-prone, & costly over lifecycles
- There are layers of middleware, just like there are layers of networking protocols
- Standards-based COTS middleware helps:
  - Control end-to-end resources & QoS
  - Leverage hardware & software technology advances
  - Evolve to new environments & requirements
  - Provide a wide array of reuseable, off-the-shelf developer-oriented services

Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

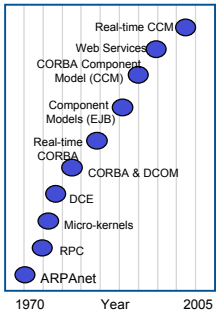
Middleware

- Infrastructure middleware.
  - encapsulates core OS communication and concurrency services to eliminate many tedious, error-prone, and non-portable aspects of developing and maintaining distributed applications using low-level network programming mechanisms, such as sockets
  - Examples: the Java Virtual Machine (JVM) and the ADAPTIVE Communication Environment (ACE).
- Distribution middleware
  - builds upon the lower-level infrastructure middleware to automate common network programming tasks, such as parameter marshaling/demarshaling, socket and request demultiplexing, and fault detection/recovery
  - Examples: Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA), Microsoft's Distributed COM (DCOM), and JavaSoft's Remote Method Invocation (RMI).

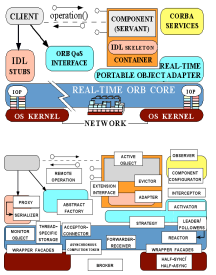
Middleware

- Common middleware services
  - augments the distribution middleware by defining domain-independent services, such as event notifications, logging, multimedia streaming, persistence, security, transactions, fault tolerance, and distributed concurrency control
  - applications can reuse these services to perform common distribution tasks that would otherwise be implemented manually.
- Domain-specific Services
  - tailored to the requirements of particular domains, such as telecommunications, e-commerce, health-care, or process automation
  - are generally reusable, and thus are the least mature of the middleware layers today
  - embody domain-specific knowledge, however, they have the most potential to increase system quality and decrease the cycle-time and effort required to develop particular types of networked applications

Progress



- significant progress in QoS-enabled middleware, stemming in large part from the following trends:
  - years of iteration, refinement, & successful use
  - maturation of middleware standards
    - .NET, J2EE, CCM
    - Real-time CORBA
    - Real-time Java
    - SOAP & Web Services
  - maturation of component middleware frameworks & patterns



Adapted from Douglas C. Schmidt, "Patterns, Frameworks, & Middleware: Their Synergistic Relationships"

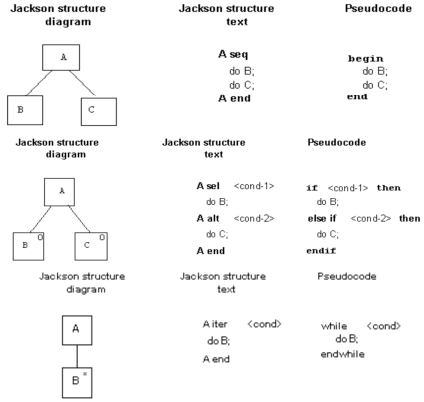
JSP & JSD

- 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

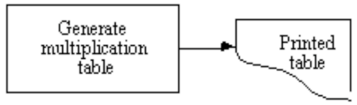
- consists of the following steps:
  1. Draw a system diagram
  2. Draw a data structure for each input and output file
  3. Draw a single data structure based on correspondences between the input and output data structures; this data structure forms the basic program structure
  4. List the operations 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:

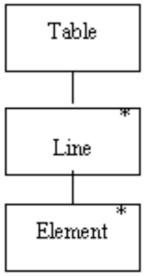
  1
  2 4
  3 6 9
  4 8 12 16
  ... ...
  10 20 30 40 50 60 70 80 90 100

1. Draw system diagram

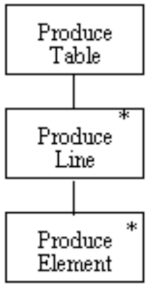


multiplication table example

2. Draw data structures



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



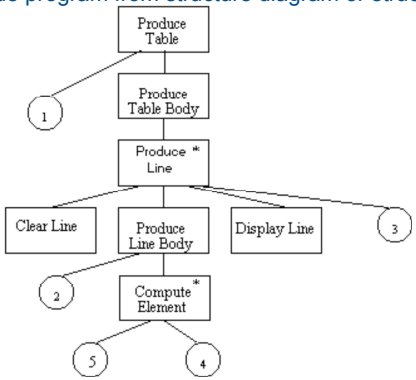
multiplication table example

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



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

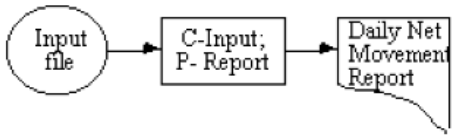
- are surprisingly common
- three well-known examples:
  - The calendar consists of years, each year consisting of a number of days. In one structure the days may be grouped by months, but by weeks in another structure. There is a boundary clash here: the weeks and months can not be synchronized.
  - 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

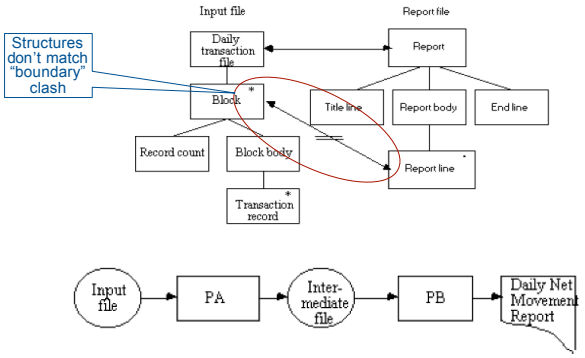
Report generation

1. Draw system diagram



Report generation example

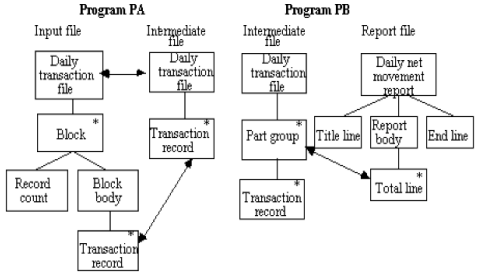
2. Draw data structures



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.

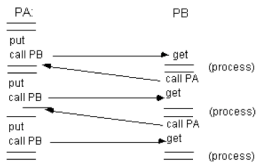
2. Draw data structures



Resolution of the structure clash

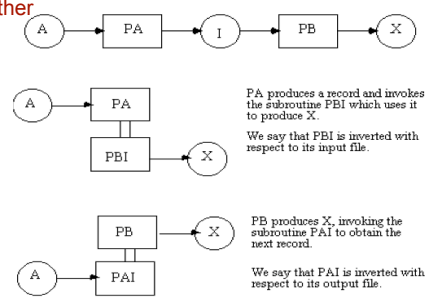
- decomposing the program P into two programs, PA and PB as shown below:

- 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



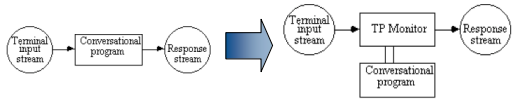
Program Inversion

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

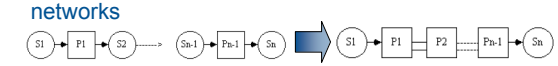


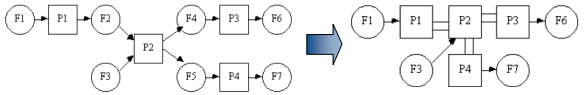
Uses of program inversion

- Interactive conversational programs



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



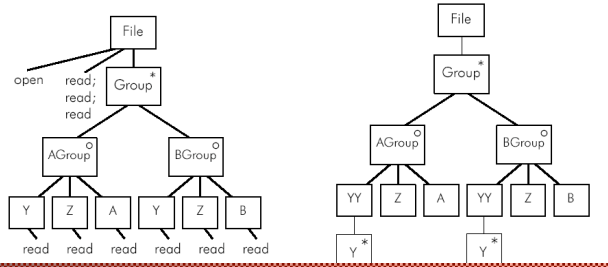


Significance of Inversion

- 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.

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.

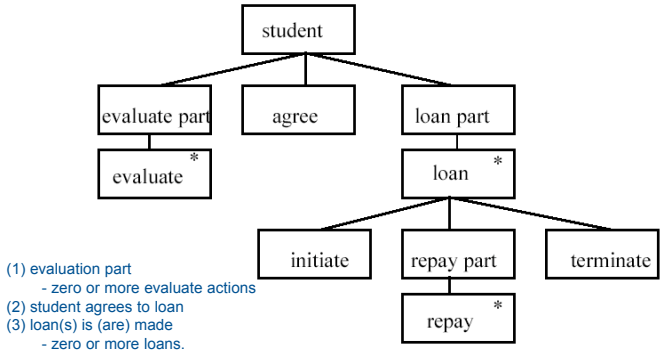
Candidates

- 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

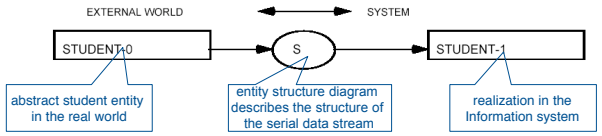
Model Process

- 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



Use JSP to create a program for the process

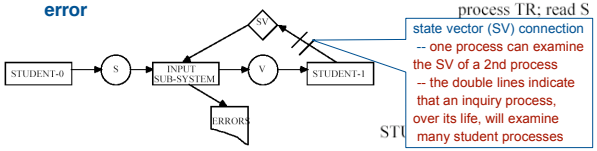
```
STUDENT-1 seq
    read S;
    EVAL iter (while TE)
        process TE; read S
    EVAL end
    AGREE seq
        process TA; read S
    AGREE end
    LOAN-PART iter (forever)
        INIT seq
            process TI; read S
        INIT
        REPAY iter (while TR)
            process TR; read S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

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

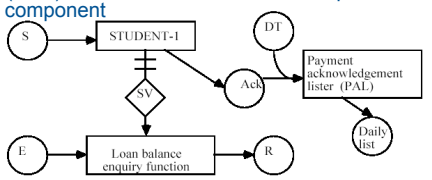
- Data stream carries a message from one process activity to an activity in another process
  - must correlate with output leaf of sending model process
  - must correlate with input leaf of receiving model process
- Data transfer assumed to be asynchronous
  - less restrictive assumption
  - no timing 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

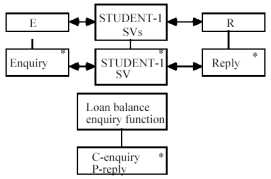
- loan balance inquiry function (LBE) is connected to the Student-1 process by state vector (SV) connection
- The function to produce the student acknowledgments data stream (ACK) is embedded in the student-1 process in the repays component



- 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

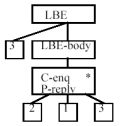
(i) input and output data structures:



(ii) basic program structure

(iii) list of operations:
1 - write 'loan balance for', student-id, 'is', balance
2 - get STUDENT SV (student-id)
3 - read E

(iv) elaborated program structureand text:

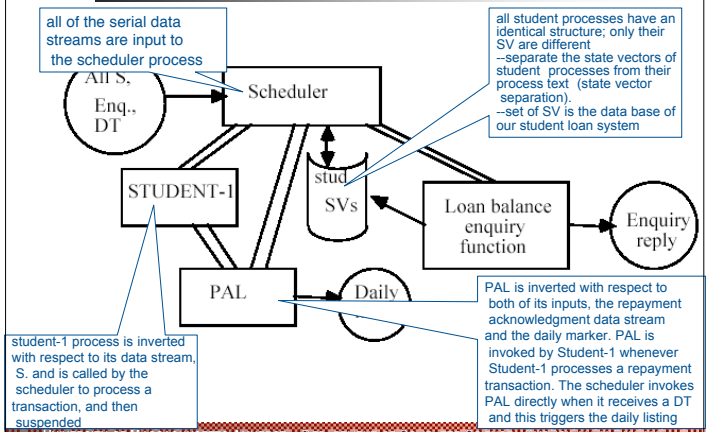



```
LBE   seq
  read E;
  LBE-BODY itr    (forever)
    get STUDENTSV(student-id);
    write 'loan balance for ', student-id,
      ' is ', balance'
    read E;
  LBE-BODY   end
LBE   end
```

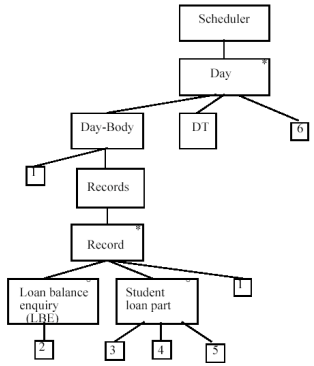

Implementation Phase

- 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

Comments/Evaluation

- Focus on conceptual design
  - But difficult to build a system this way
- Based upon model of real world
- Careful (and experienced) analysis of the model generally points suggested implementation tactics, though
  - Parnas notions of module not perceptible here
  - Not an iterative refinement approach either
- Treatment of data is very much subordinated/secondary
- Does a good job of suggesting possible parallelism
- Contrasts strongly with Objected Oriented notions (eg. Booch, UML)