CS446
Software Engineering

Richard Bartels

University of Waterloo
Faculty of Mathematics
Department of Computer Science

October 3, 1996

Diagrammatic Design Techniques

Richard Bartels

University of Waterloo
Faculty of Mathematics
Department of Computer Science

October 3, 1996

Outline

- Design Approaches and System Models
- Data Flow Diagrams
- State Transitions
- Control Flowcharts
- Temporal Sequencing
- Structure Charts
- Entity Relationships
- Object Oriented Analysis
  - Class Diagrams
  - Object Diagrams

Reference:

This is material primarily from Chapters 6, 12, 14, and 15 of Sommerville

System Models

Many models of system structure or behavior exist.

Each fills a certain need and represents a certain aspect.

Those to be mentioned are:

- Data Flows
  - express layout of data and action of process on data
- State Transitions
  - express shifts from process to process
- Control Flowcharts
  - express the control structure of a single process
- Temporal sequencing
  - lays out the timing and interrelationship of events
- Structure charts
  - assist in determining modular decomposition
- Entity relationships
  - depict the associations and services between components
- Object oriented analysis
  - combines data and process into logical components

Analysis and Design

Books often make the distinction between

- Analysis

  The implementation planning phases from preliminary to mature that are characterized by layout, organization, decomposition, and sketching techniques; e.g., data-flow analysis, top-down modular decomposition, entity relationships, class diagrams, etc.

  These constitute the *basic products* of the design process.

- Design

  The implementation planning phases from mature to final that are characterized by formalization and in-depth finalization of details; e.g.

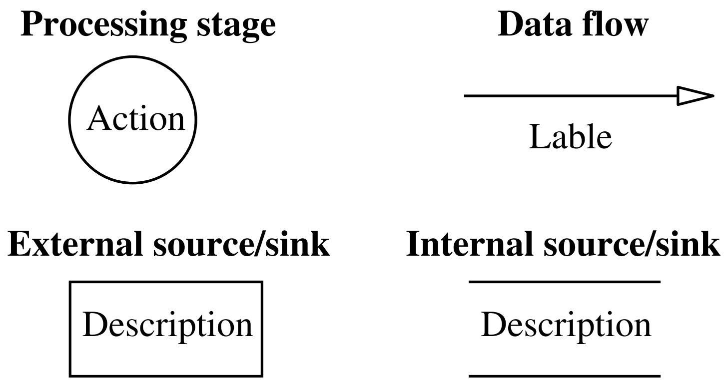
  - process specifications
  - interface specifications
  - data dictionaries
  - decision and input/output tables
  - formal predication notations

  These constitute the *derived products* of the design process.

Data Flow Diagrams

Graphical elements to represent:

- Processing stage
- Data flow
- Data source/sink
  - Internal
  - External



Invented during the data processing era.
Extensions to real-time systems, etc.

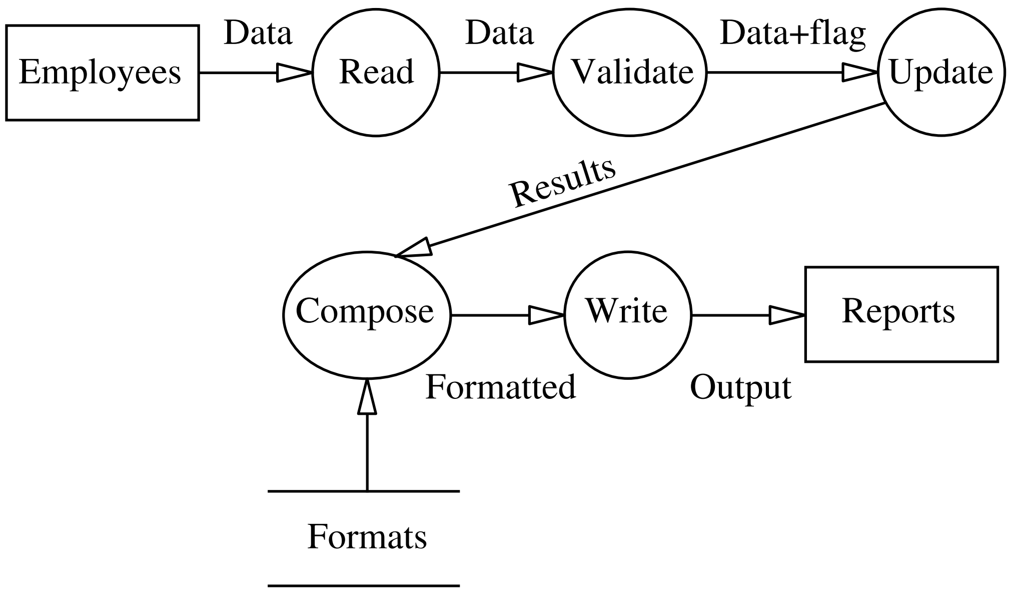
Interpretation:

Information flowing through a fixed graph whose nodes represent stages of processing or locations of storage.

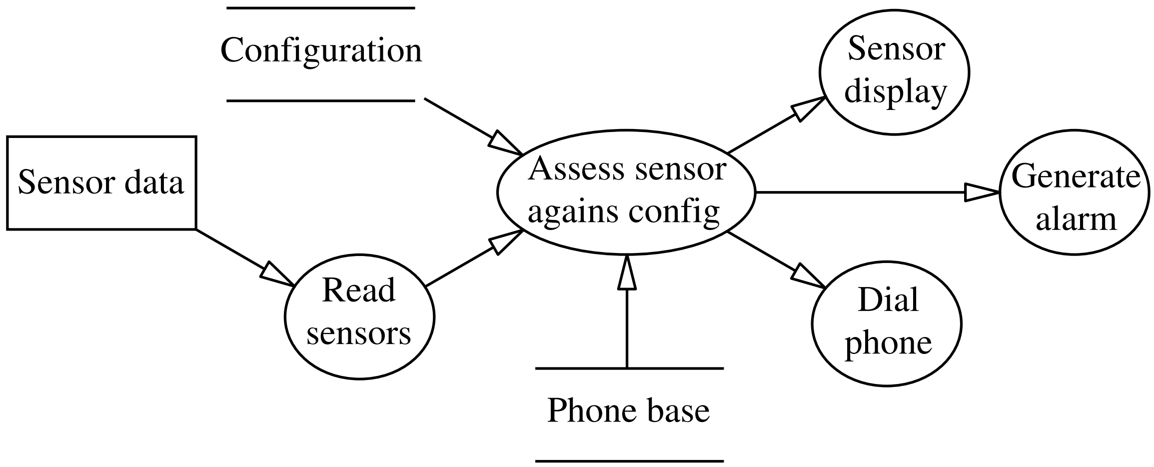
Principal Structures

Two main structures to recognize:

- Transform centered (linear, node in/out degree 1)



- Transaction centered (branched, one node of in/out degree >1)



Remarks

- Any DFD may have one or more localities of either type.
- Subsidiary information mandatory
  - Edge/source/sink labels are data

    Data dictionary
    (form, purpose, using processes)
  - Node labels are process

    Process specification
    (pseudocode and/or state-transition diagrams)
- DFD's should be constructed by stagewise refinement.
  - Organized by level of refinement
  - Levels refer to, but don't repeat, ancestors' information
  - Each level has: graph segments, DD's, PSpec's
  - An overall table of contents or index should be prepared

Data Dictionary

Contains standard, cross-referenced information about a data label

- What is it?
- What does it consist of?
- What is its form, composition, format?
- Where does it come from?
- Who uses it?
- What is its purpose?

Sample layout

Data Name:	`telephone number`
Used by process:	`dial phone`
Comes from:	`phone base` data store
Purpose:	activates `phone tones`, different numbers related to different sensor and alarm conditions (see `phone base` DD and `assess sensor against config` PSpec)
Format:	standard North American form 1-area-XXX-XXXX for long distance XXX-XXXX for local (extensions do not occur)

Process Specification I

Any reasonable formalism

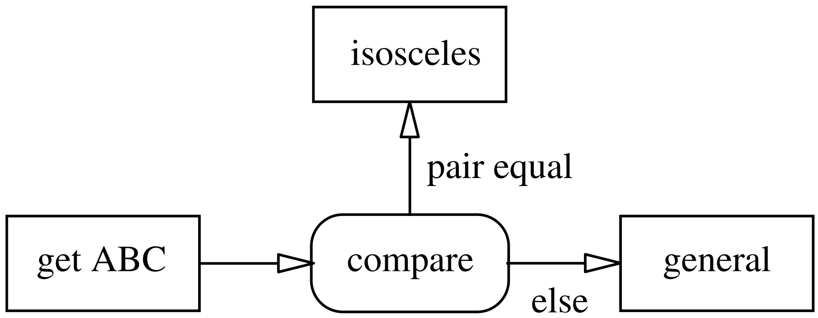
- Processing narrative

  Read A, B, C and check that all are positive. Compare A, B, and C to see if any two are equal. If so, report "isosceles." If not, report "general."

- Structured English

  accept A,B,C
  if A=B
  or if A=C
      or if B=C
          then report "isosceles"
          else report "general"

- Flowchart



Process Specification II

Contains standard, cross-referenced information about a process

- What does it do?
- What purpose does it have?
- What data does it use?
- With whom does it communicate?

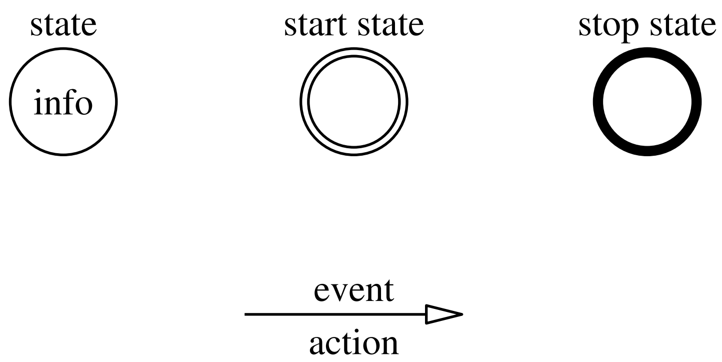
Sample layout

Process Name:	`triangle classification`
Uses data:	`side lengths`
Action:	*see previous slide*
Purpose:	filters data triples so that a specially tuned process can handle isosceles triangles (see *area computation* processes)

State Transition Diagrams

Graphical elements to represent:

- A state of the system
- A change from state to state
- An event that causes state to change
- An action taken when state changes

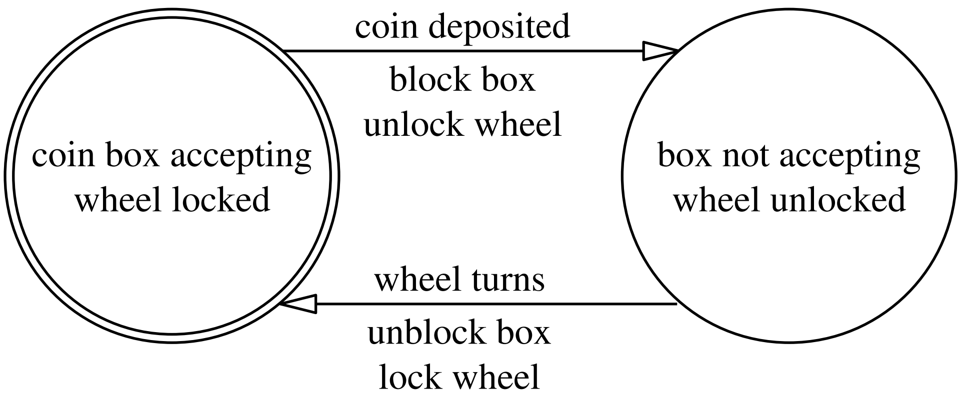


State transition diagrams are particularly useful for laying out user interactions.

Example: Monolithic States

Each state node represents the full details of one of the entire system's states

Each transition gives the full details of all actions undertaken

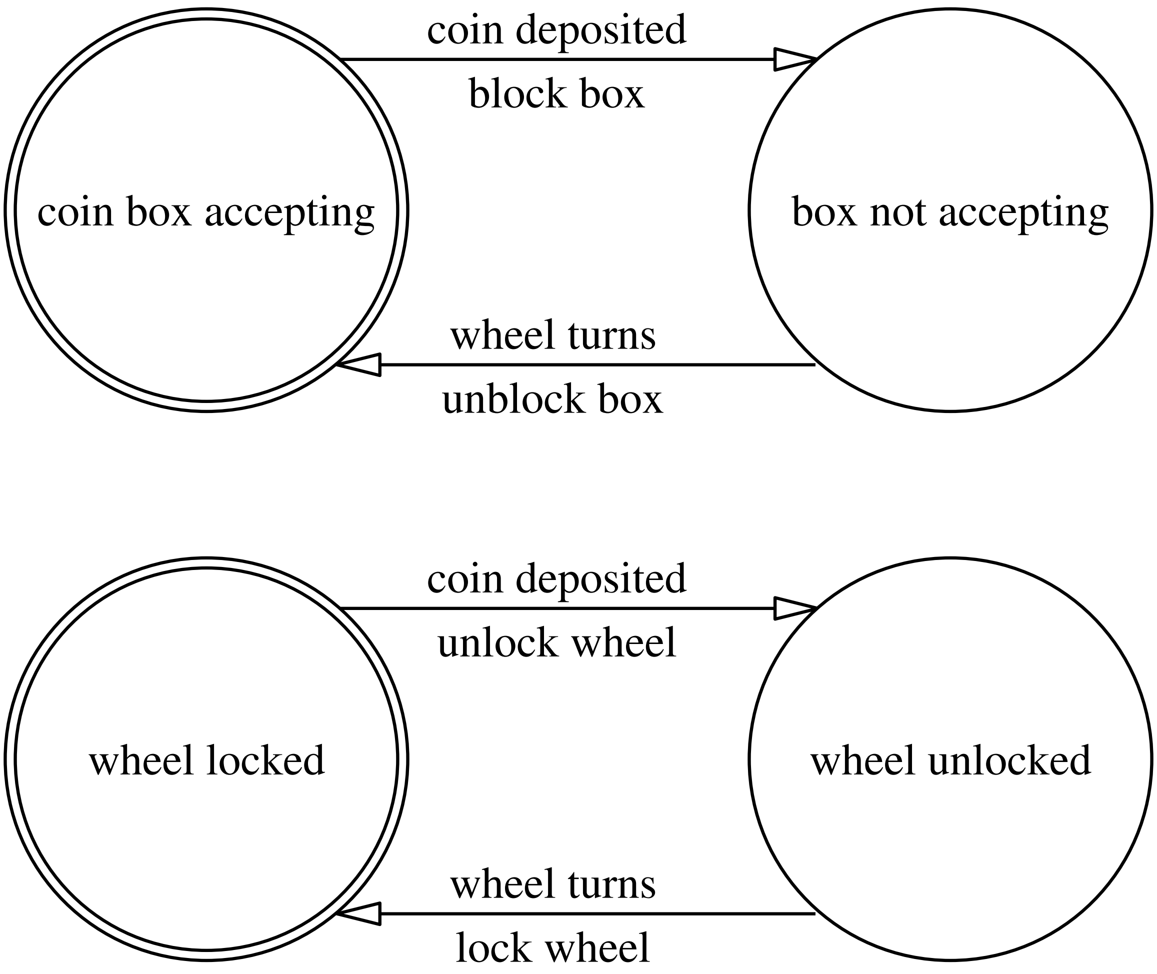


Example: Constituent Element States

Each state node represents the details of some component of the system

Each transition gives the details of a component's action

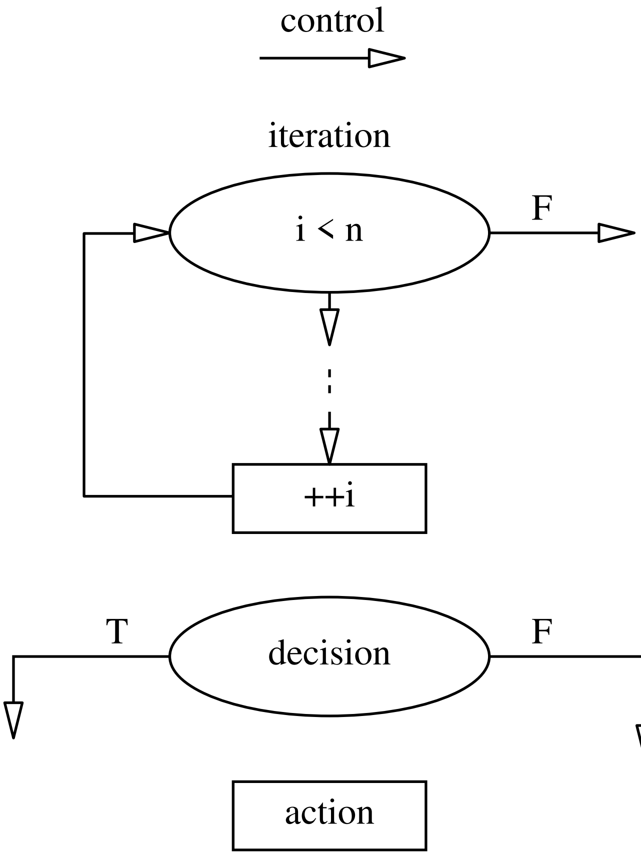
One event may appear on several component's transitions, in which case, the transitions are assumed to take place on components simultaneously



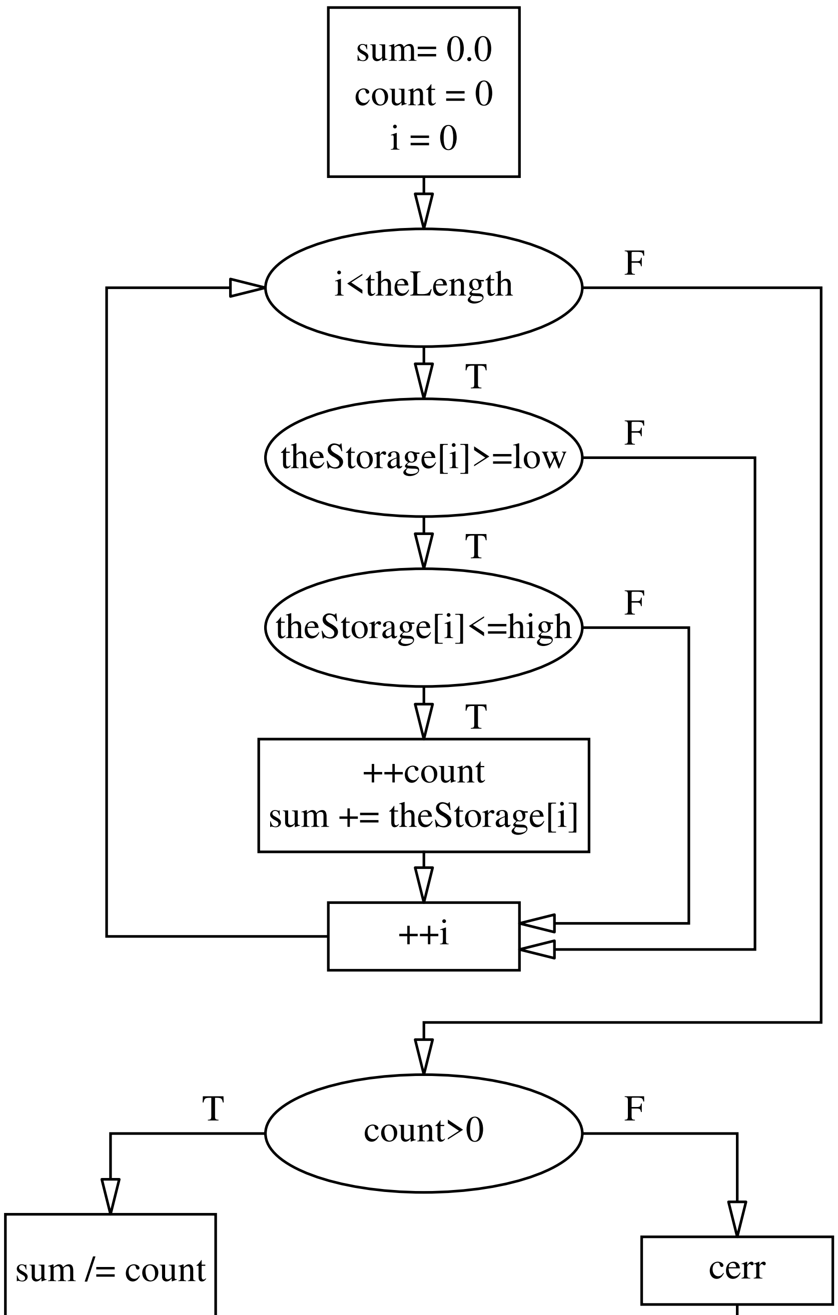
Flow Charts

Graphical elements to represent:

- A passage of control
- A process or action
- A decision branch
- An iteration



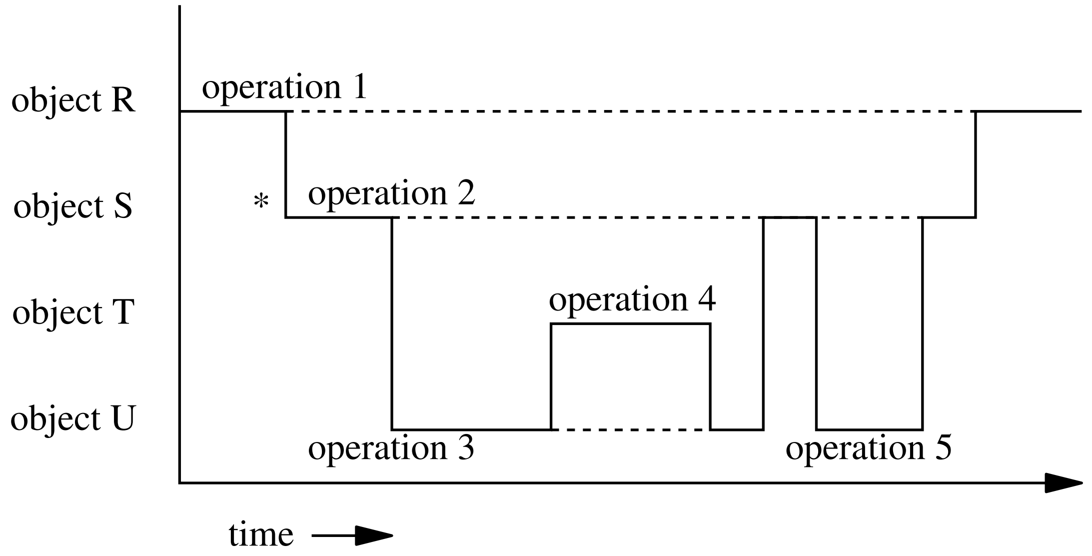
Example


sum= 0.0
count = 0
i = 0
i<theLength
F
T
theStorage[i]>=low
F
T
theStorage[i]<=high
F
T
++count
sum += theStorage[i]
++i
count>0
T
F
sum /= count
cerr
return

Timing Charts

Graph showing interacting objects along the vertical axis, time along the horizontal axis, and operation sequences as a nested timing diagram.

Example: A timing diagram for Operation 1 of Object R, which is implemented using the services of Objects S, T, and U:

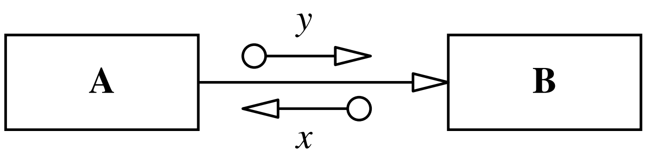


Object R initiates Operation 1
In the execution of Operation 1, Object S is created (*)
Object S initiates Operation 2
Operation 2 invokes Operation 3 of Object U
Operation 3 invokes Operation 4 of Object T
Operation 4 ends; Operation 3 continues
Operation 3 ends; Operation 2 continues
Operation 2 invokes Operation 5 of Object U
Operation 5 ends
Operation 2 ends
Operation 1 ends

Structure Charts I

Elements

- Module
  - A unit, capable of being named, encapsulating code and data, with a defined interface, supported in the language
  - Depicted by a box with a name
- Connection
  - A reference from one module to something defined in another (e.g. a procedure call)
  - Depicted by an arrow from the first module to the second
- Couple
  - A data item transferred from one module to another
  - Depicted by a short arrow with a circular tail

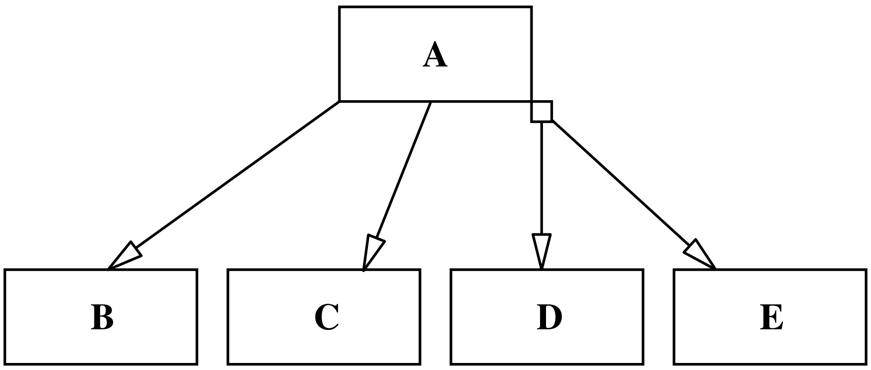


A calls **B** providing data y; **B** responds with result x

Structure Charts II

Elements

- Sequence
  - Left to right and top to bottom convention
- Condition
  - Diamond on tail of connection

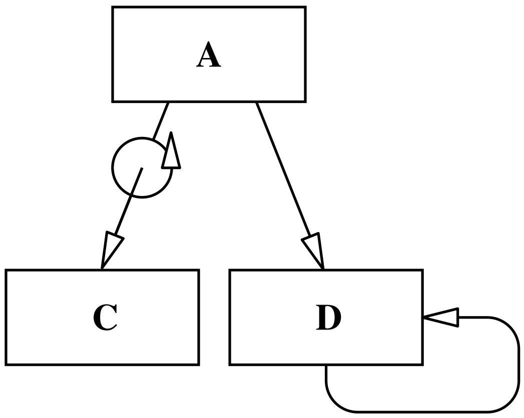


first **A** calls **B**;
next **A** calls **C**;
next **if** something **then A** calls **D else E**

Structure Charts III

Elements

- Repetition
  - Circular arrow
- Recursion
  - Loop arrow



A calls **C** iteratively
until some termination test;
A calls **D**, which calls itself recursively

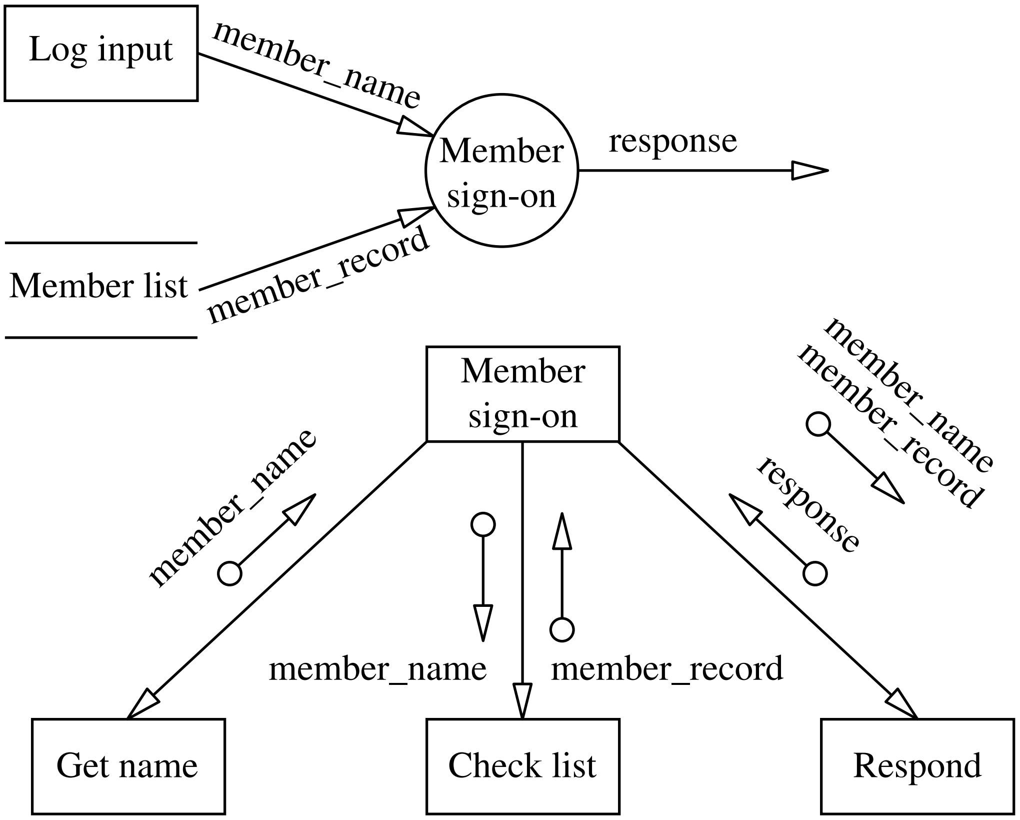
Basic Transform Approach from DFD

Begin with

- a box for each arrow
- plus boxes as necessary for processes
- plus boxes as necessary for control

Collapse boxes later

A loose example (use common sense):



Transaction and Transform

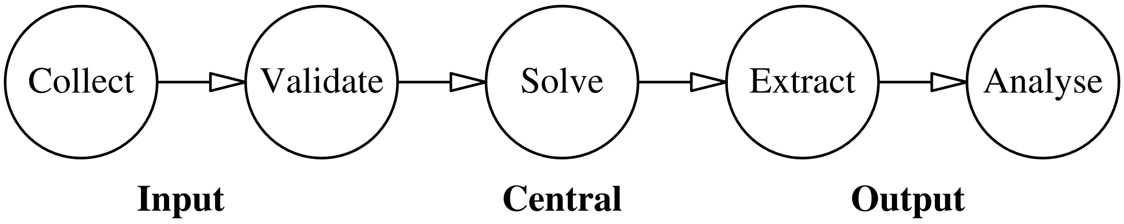
Further approach added by recognizing structure

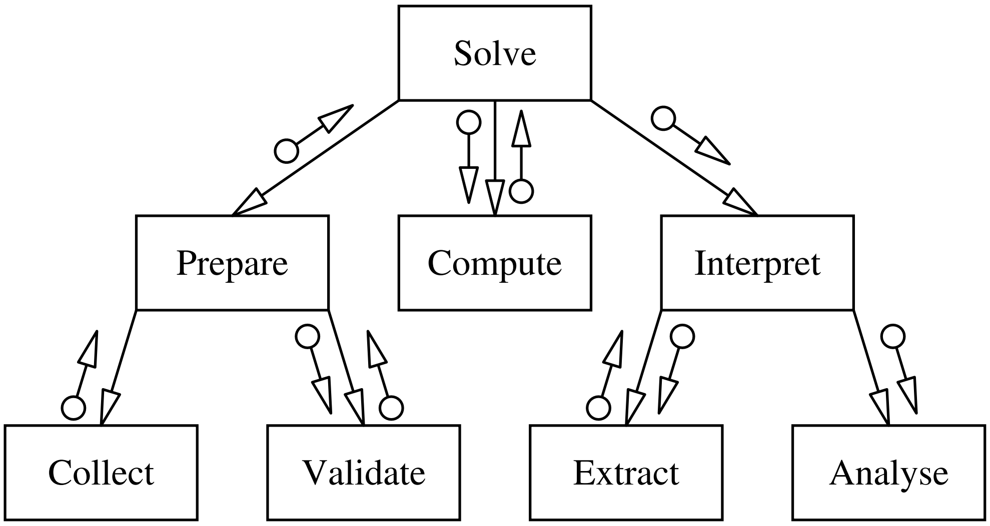
- Generally
  - Separate into input, central, output regions
- Transform structure
  - Nodes of each region according to basic transform approach
- Transaction structure
  - Input and output according to basic transform approach
  - Central region may require control and dispatch

Remember:

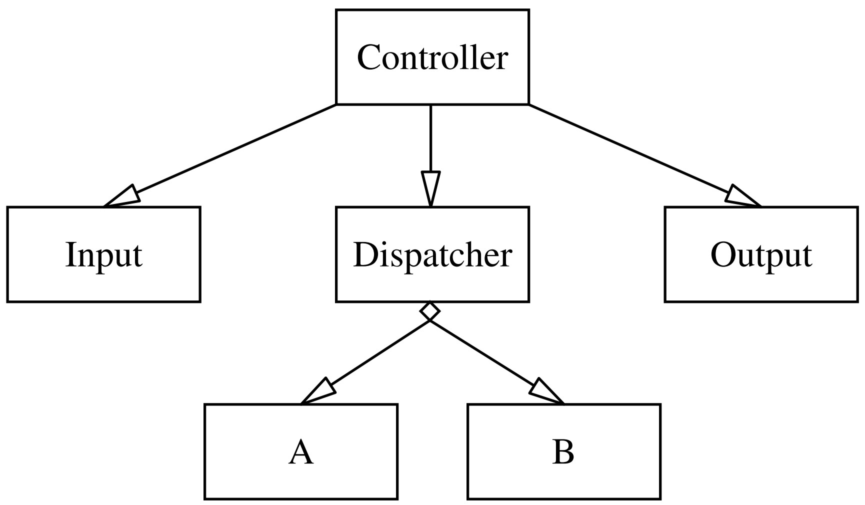
- Avoid rigid, mechanical translation to modular form
- Diagrams should assist design, not constrain it
- Make it your own

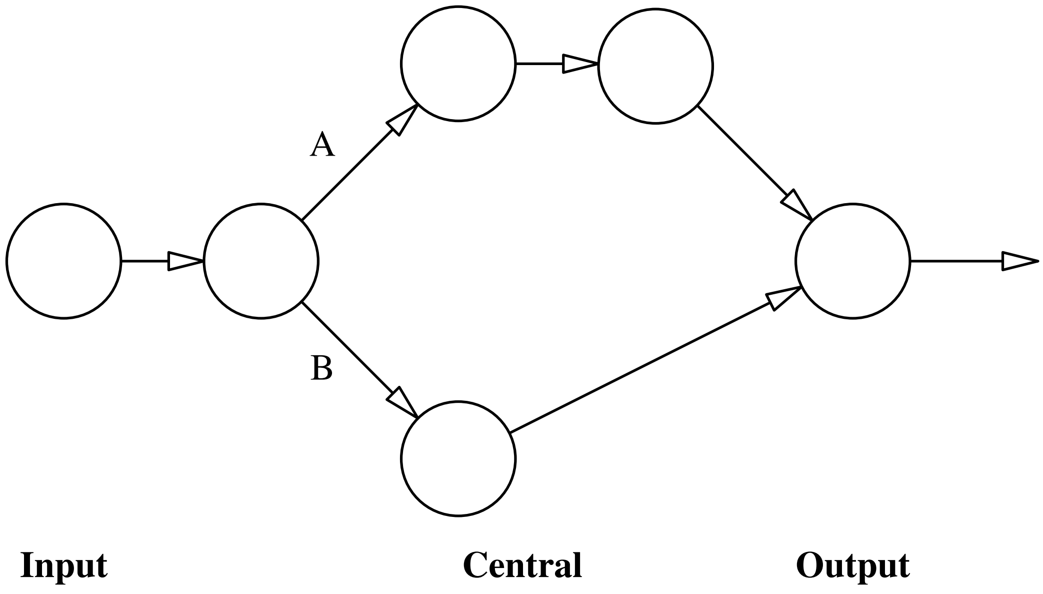
Transform Structure





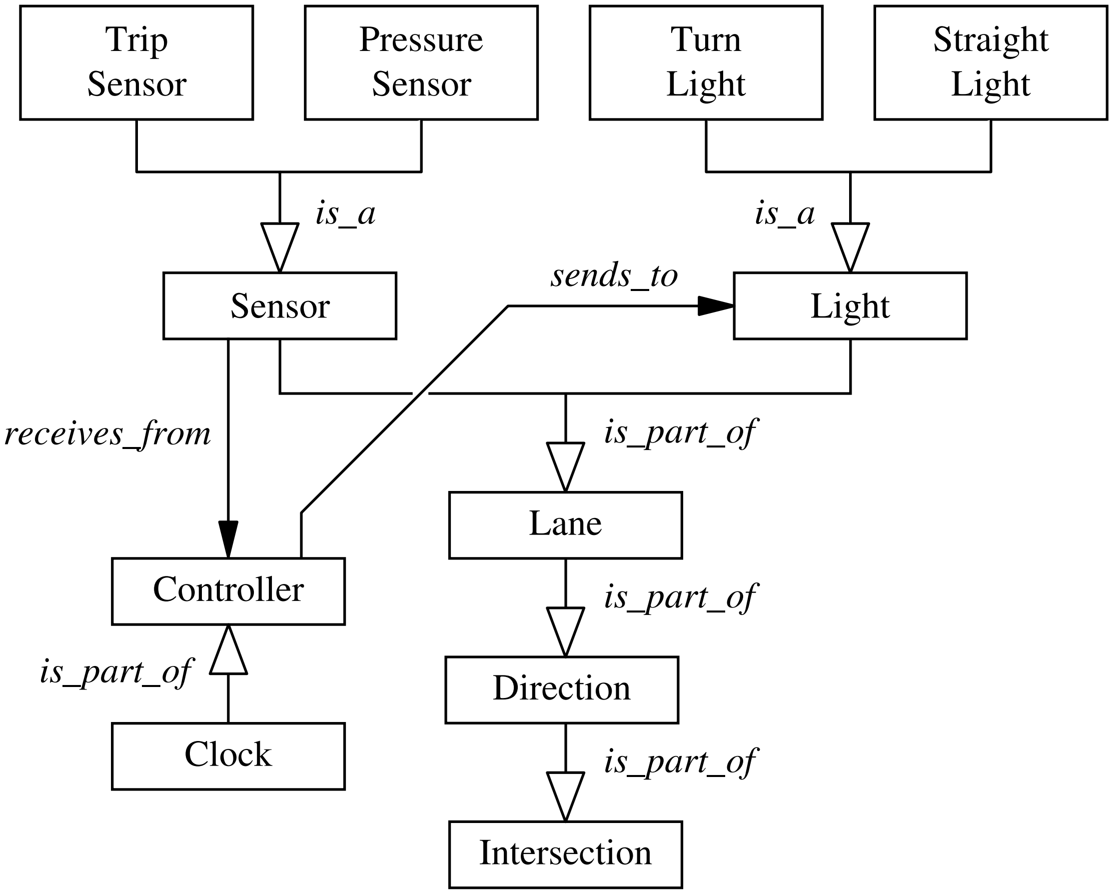
Transaction Structure





Entity Relationships

The traffic intersection problem



C++ Conversions

```
class Sensor
{
  SenseReport report( void );
};

class TripSensor :  public Sensor
{
  bool isTripped;
};

class PressureSensor :  public Sensor
{
  float pressure;
};

class Controller
{
  void takeReading( SenseReport r );
  ControlSignal assertControl( void );
  Clock aClock;
};
```

etc.

Object Oriented Analysis

Basic terms:

- Object
  - Something that has a unique identity (name, pointer, address), has state (contains data), and exhibits behavior (has procedures that operate on its data)
- Class
  - The definition (description) of a set of objects that share a common structure and behavior
- Member
  - A data item or a procedure in a class or object
- Message
  - "Send a message to an object" is another way of saying "call (execute) a member procedure of the object."
- Module
  - A unit, capable of being named, encapsulating code and data, with a defined interface, supported in the language (Examples: subroutine in FORTRAN, procedure in C or C++, **header/code file pair** in C or C++, class in C++, module in Modula, etc.)

Example

```
class Prisoner
{
  public:
    Prisoner( int IDnum );
    ~Prisoner( void );
    void condemn( int years );
    int reportID( void ) const;
  private:
    int yearsOfSentence, IDnumber;
};
int main( void )
{
  Prisoner Zenda(459298);
  Zenda.condemn(99);
  cout  << "Prisoner " << Zenda.reportID()
        << "condemned for life." << endl;
}
```

`Prisoner` is a class, `Zenda` is an object, `void condemn( int )` and `IDnumber` are members of the object `Zenda` and of the class `Prisoner`, the object `Zenda` responds to the `reportID` message by reporting its `IDnumber` member.

Object Interactions

Ranger Rick kills Private Zaphod on orders from General Electric:

```
#include "MilitaryClasses.h"
int main( void )
{
  Ranger Rick;
  Private Zaphod;
  General Electric;

  Rick.obey(Electric.issueKillOrder(),Zaphod);
}
```

The point of this example:

Using member functions it is possible to set up a notational scheme for

- objects requesting services from objects
- objects passing information to objects
- objects creating, destroying, modifying objects

This will serve to mimic the attributes and actions (semantics) and relationships (interactions) of entities in the real world that are to be the subjects of a software system.

Object Relationships

A Car is a kind of Vehicle that has four Wheels:

```
class Vehicle
{
  public:
    void travel( Location start, Location goal );
};
class Car :  public Vehicle
{
  private:
    Wheel leftFront, leftRear,
          rightFront, rightRear, spare;
};
```

The point of this example:

Through member data and inheritance it is possible to express

- the fact that objects may have other objects as components ("has a" relationship)
- the fact that objects may be special kinds of other objects ("is a" relationship)

This will further serve to mimic the relationships and composition of entities in the real world that are to be subjects of a software system.

Analysis

Multiple, Interrelated Views:

- Static semantics
  - Logical view
    - ▷ Class structure
    - ▷ Object structure
  - Physical view
    - ▷ Module architecture
    - ▷ Process architecture
- Dynamic semantics
  - Logical, physical views (repeat above)

Process of analysis and design

- Lay out the modules of a given level of abstraction
- Specify the interfaces between the modules
- Identify the classes and objects comprising each module
- Identify the semantics of these classes and objects
- Identify the relationships among these classes and objects
- Implement these classes and objects (or further refinement)

Good Things

Design issues:

- Decomposability
  - Design can easily be refined
- Composability
  - Design components can be reused in other designs
- Clarity
  - Components are easily understood by themselves
- Continuity
  - Small changes in design impact only few components
- Protection
  - Robust error handling within each component

Achieve through design principles:

- Components exploit linguistic units
- Few interfaces
- Simple interfaces
- Well-designed, effective, meaningful interfaces
- Information/implementation hiding

Approaches

At each level of abstraction, what are the objects/classes?

- The conceptual approach

  Look for:

  things, events, roles, structures, interactions, etc.

- The linguistic approach

  Formulate an English processing narrative and:

  underline nouns, underline verbs

- The design-step approach

  Identify in careful sequence:

  abstractions, attributes, operations, communication

- The design-paradigm (idiom) approach

  Look for known, generic paradigms such as:

  model-view-controller, flyweight, abstract factory, etc.

Graphical Design Notation

- Most complete: Grady Booch
- Covers static and dynamic semantics
- Includes icons for logical and physical details of
  class, object, process, state,
  time, relationship, architecture
- Organized according to:

– Class diagram

Structure of classes and relationships between classes

– Object diagram

Structure of objects and relationships between objects

– Module diagram

Physical packaging of objects and classes

– Process diagram

Allocation of processes to processors

– State diagram

Illustrates the state machine of a class

– Interaction diagram

Illustrates the time sequence of object interaction

OO Design vs. Structured Design

Conventional top-down design

- Forces an early choice of control sequence
- Emphasizes single master and slaves
- Produces tree-structured modularization

Resulting structure may be rigid

- Tree structure resists re-design
- Slaves fixed to design of master resist re-use

Object oriented design

- Encourages early focus on data and agents
- Encourages abstraction of data and agents
- Absence of master control possible
- Emphasis on control sharing among agents
- Produces graph-structured modularization

Data

Object whose focus is on its information (state, data)

Agent

Object whose focus is on its services

OO Promise of Flexibility

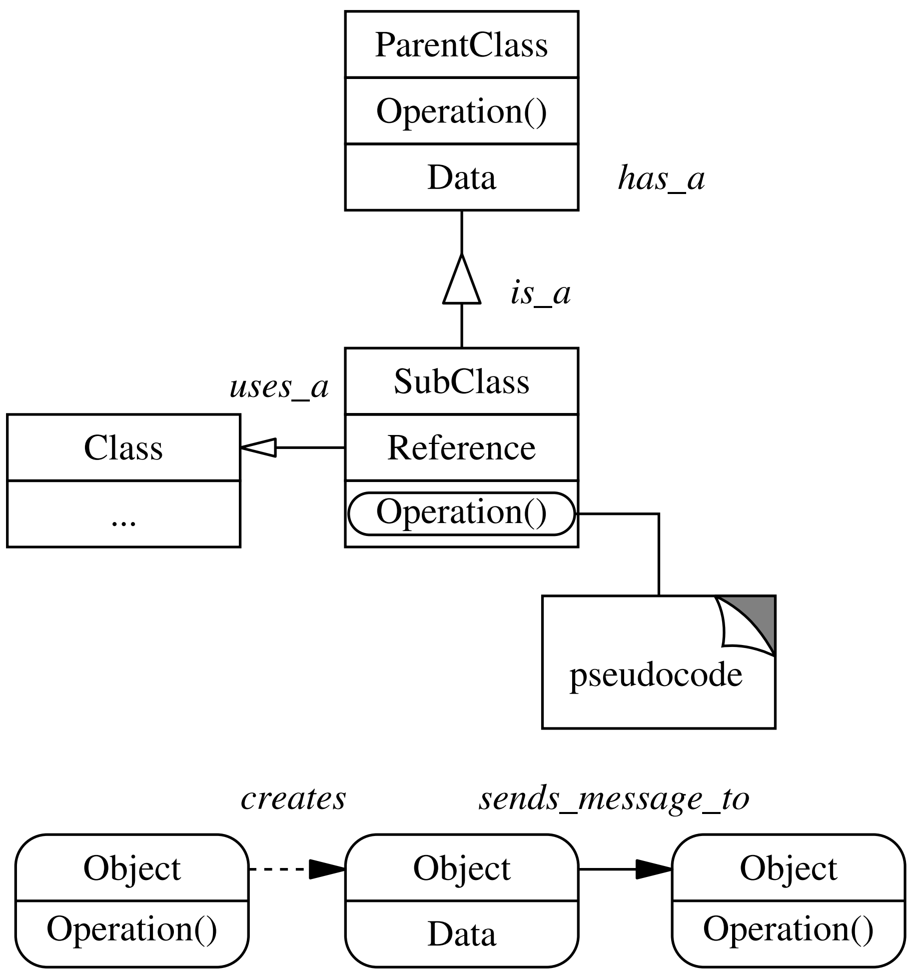
Absence of master control and abstraction of data and agents

- Permits easy re-design
  - Local adjustment of data or agent
  - Re-connection of graph edge (control/data path)
- Permits extensive re-use
  - Properly abstracted data or agents have wide use
  - Inheritance provides easy adjustment

Top-down considerations

- focus on stagewise refinement of data/agent through inheritance
- focus stagewise refinement on control/communication through graph-node expansion

Common Diagram Elements



Is-A

This term refers to the standard concept of

inheritance though subclassing

`ClassB` *is a* kind of `ClassA` if `ClassB` is derived from `ClassA` (thereby providing the same public members as `ClassA`).

This mechanism allows the design of *generic code*.

```
class Plane {
  public:
    virtual double cruiseFuelRate( Conditions cond ) =0;
};
class Cessna172 :  public Plane {
  double cruiseFuelRate( Conditions cond );
};
```

Code can be written in terms of `Plane` objects that provide values of `cruiseFuelRate` under given `Conditions`.

That code can be executed on `Cessna172` type objects (or any objects of a type derived from `Plane`).

Has-A

This term refers to the standard concept of

data aggregation

`ClassX` *has a* copy of `ClassY` if any object of type `ClassX` contains a copy of `ClassY` as part of its member data.

This mechanism allows *encapsulation*, the grouping of a number of objects together that must act as a unit and the *hiding* of their collective details under a single name. It loosens the *coupling* within a system by replacing long argument lists to procedures by single, encapsulated objects.

```
class Conditions {
  double temperature;
  double altimeterSetting;
  double altitude;
};
```

Uses-A

This term arises when one class

calls upon a member function of another class

Sometimes we say that the one class *sends a message* to the other class. A distinction is often made: When a `ClassX` object invokes a member function of a `ClassY` object in the code that defines one of the `ClassX` member functions, we say that

`ClassX` *uses* `ClassY` *for implementation*

```
Interaction::prompt(message) {
  displayDeviceHandle->clear();
  displayDeviceHandle->output(message);
}
```

When a `ClassX` object directly provides a member function of a `ClassY` object as one of its own member functions, we say that

`ClassX` *uses* `ClassY` *for interface*

```
PointingDevice::getPickEvent( void ) {
  myMouse.getPickEvent( void );
}
```

This mechanism contributes to the *re-use*, *localization* and *coherence* of code.

Object Structure