Key Focus of Software Engineering

- How to describe software products?
- Processes to build software products?
  - And assure that the products are “good” at the end
- How to grow and evolve such products
  - At acceptable costs
  - And improving quality

Software Engineering = \textbf{Products} + Processes
Stakeholder Perspective Suggests an Approach

- How to manage the creation and maintenance of a software product that satisfies all needs of all stakeholders?
  - Implies understanding who stakeholders are; what questions they need answered; to what degree of thoroughness
  - Implies supporting reasoning needed to provide adequate answers to stakeholder questions (at acceptable cost)
  - Implies need for representation(s) of product sufficient to support such reasoning; and reasoning techniques

Stakeholder questions and concerns
The Problem of Providing Satisfactory Answers to Stakeholder Questions

- Most important questions are vague
  - Eg. Is this software user-friendly?
  - (ATCS) How to interact with pilots to assure user interfaces are going to be effective?
- Most important questions have open-ended answers
  - Eg. How fast is this system?
  - (ATCS) Different platforms, situations, conditions
- Different stakeholders require different degrees of assurance
  - Affects the degree of thoroughness of (eg.) testing
  - (ATCS) Some functions more critical than others
- Different stakeholders have differing degrees of technical sophistication
  - Affects the choice of formalism in which to couch answers

Using Models to Answer Questions

- What is a model?
- How have they been used elsewhere?
  - Most previous models have been tangible and have typically been models of physical objects
- Software Models
  - Model non-physical objects, are not tangible
  - What should they be like?
- Should be driven by what they are for
No one representation can be expected to suffice for all purposes

- That is the case in manufacturing other kinds of products:
  - Houses: Multiple diagrams and views; appliance user manuals; inspection reports and certificates, etc.
  - Cars: Owners manuals; shop manuals; EPA reports; NTSB reports;
  - Laws: Legislative blueprints; law; opinions; decisions in cases involving the law; impact assessments; agency budgeting reports
  - Movies: Story treatment; script; actual film; reviews; profit and loss accountings

- Representations are quite varied: Notations expressing them are too

- We should expect the same for computer software--products and processes

- Notations are better evolved, more effective, in cases of better established manufacturing disciplines

Typical Approach

- Select a (set of [interconnected?]?) representation(s?) (some of which are?) effective in communication with stakeholder constituenc(ies?)

- Derive information/answers expressed in that representation that are able to satisfy stakeholder(s)

- Assure that the information is consistent with other parts of the product (eg. the code!)

- ATCS: FAA needs proof that all collisions will be detected
  - Statement of proof needed
  - Other artifacts: body of proof, code, code structure representations, etc. Must be derived and shown to be consistent with each other
Key Model Desiderata

• Precise
• Detailed
• Broad
• Clear

Precise

• Based upon rigorously defined semantics
• So that there is a meaning that can be definitively established to the satisfaction of all
  – To resolve disputes and disagreements
• So that definitive reasoning is possible
  – Deriving properties
  – Answering specific stakeholder questions
Detailed

• Stakeholder questions may need answers that require details
• Model must support getting down to the level of detail that is required
  – By the relevant stakeholder(s)

Broad

• Many semantic issues may need to be covered
  – Functionality, timing, robustness, precision, resource utilization,….
• More semantic richness is better
  – It is capable of supporting answering more questions from more stakeholder types
• Broad and Precise and NOT the same
Clear

• Understandable to the relevant stakeholder(s)
• So that they can understand what the model is saying
  – And what the results of reasoning mean
• This is definitely relative to the stakeholder community

The Great Compromise

• No single modeling approach is likely to excel in all four dimensions
• What is each one likely to be good for?
• Different needs under different circumstances
  – During conceptualization, contract negotiation
  – During development
  – During test and evaluation
  – During utilization
  – During evolution
• How to select one(s) that are appropriate for the stakeholder communities?
  – At the relevant time(s)

One of the most important tasks for a software engineer
Some Software Representation Approaches

- Programming Languages
- Natural Language
- Structured Language
- Box-and-Arrow Charts
- Graphs
  - Flowgraphs
  - Callgraphs
  - Dataflow graphs
  - Petri Nets
- Databases
- Charts, Diagrams
- Mathematical formalisms (e.g. predicate logic)

Which of these are best for answering which types of questions for which types of stakeholders?
Roadmap

• Look at some of them now
  – Separate and apart from when, where, and how they might eventually be used
• Understand about “precise, detailed, broad, clear”
• Then examine needs at different points during software development
• We will see many combinations of the basic approaches
  – Understand the basic approaches first

Do We Need Anything More than a Programming Language?

• Code is the ultimate
• Describes what is actually being done
• What more can you ask for?
Programming Languages

- Primary stakeholder constituency is Developers
  - Helps answer: What does the product do?
- Less useful to other stakeholder constituencies:
  - How do you get the product to do what it does? (users)
  - What is the product supposed to do? (users/investors)
  - Will the product ever fail disastrously? (bystanders)
  - Is the product’s development (almost) done? (managers/investors)
  - How extensible/adaptable is the product? (maintainers/managers/investors)
- (Some) Differences in Programming Languages are intended to help them be (more) effective in answering some of these sorts of questions.
- But additional representations are more suitable for most of these questions and stakeholder constituencies

Even Developers Need More

- “What does it do”---for
  - Complicated programs
  - Large programs
  - Badly written programs
  - Programs written in poor programming languages
- “How does it work”
  - NB: Languages that separate specification part from implementation part are attempting to address this question
- “What would happen if I changed.....”
- “How should our team go about changing ....”
Even Developers Need More

• “What does it do”---for
  – Complicated programs
  – Large programs
  – Badly written programs
  – Programs written in poor programming languages

• “How does it work”
  – NB: Languages that separate specification part from implementation part are attempting to address this question

• “What would happen if I changed.....”
• “How should our team go about changing ....”

Most programming languages answer MOST software questions poorly

Example: Elevator Controller

• What is it supposed to do?
  – Stop on every floor it is called to
    » Maybe not so easy: multiple elevators
  – Service users “first come, first served”
    » May conflict with optimal strategies

• These are hard to be precise about, reason about in “plain English”

• Code helps with some of these, but not all

• What should it never do?
  – Allow elevator to move with doors open
  – What else?
• “How does it work”
  – Diagrams: Box-and-Arrow charts, data and control flow
  – Petri-Nets

More Appropriate Representations(?)

• “What is it supposed to do”
  – Natural language
  – Finite State Machines

More Appropriate Representations(?)
• “How does it work”
  – Diagrams: Box-and-Arrow charts, data and control flow
  – Petri-Nets

• “What is it supposed to do”
  – Natural language
  – Finite State Machines

• “What can’t/won’t it do”
  – Natural language

• “How do I get it to work”
  – Structured Natural Language

More Appropriate Representations(?)
• “How does it work”
  – Diagrams: Box-and-Arrow charts, data and control flow
  – Petri-Nets
• “What is it supposed to do”
  – Natural language
  – Finite State Machines
• “What can’t/won’t it do”
  – Natural language
• “How do I get it to work”
  – Structured Natural Language
• “What would happen if I changed.....”
  – Dependency diagrams

More Appropriate Representations(?)

• “How does it work”
  – Diagrams: Box-and-Arrow charts, data and control flow
  – Petri-Nets
• “What is it supposed to do”
  – Natural language
  – Finite State Machines
• “What can’t/won’t it do”
  – Natural language
• “How do I get it to work”
  – Structured Natural Language
• “What would happen if I changed.....”
  – Dependency diagrams
• “How should our team go about changing ....”
  – Process descriptions, using...
  » All of the above
• “How does it work”
  – Diagrams: Box-and-Arrow charts, data and control flow
  – Petri-Nets
• “What is it supposed to do”
  – Natural language
  – Finite State Machines
• “What can’t/won’t it do”
  – Natural language
• “How do I get it to work”
  – Structured Natural Language
• “What would happen if I changed.....”
  – Dependency diagrams
• “How should our team go about changing ....”
  – Process descriptions, using...
  » All of the above
• “What does it do” for large, complex programs
  – Program models and abstractions

More Appropriate Representations(?)

Create them, and assure that they are consistent with other artifact types
Natural Language

- Write in "plain English"
- All stakeholders understand natural language (?)
- Possible to augment with defined terms
- Use of punctuation for clarification
- Text/word processing systems help automate/maintain/alter

Examples of Natural Language artifacts:

User manuals
Most requirements specifications
Most test plans
Development status reports

Advantages of Natural Language

- Easy to train users
- Clarity is possible (but may be difficult)
- Completeness is possible (but by no means assured)
- Easily modified
- It is the “least common denominator”

But--
Disadvantages are far more numerous and serious
Natural Language Disadvantages

• Determining consistency between natural language artifacts and anything else is hard/subjective
  -- Ambiguity in natural language is easy and sometimes intentional
  -- Clear natural language expression is very difficult
  -- The longer the text, the more information, the more the risk of inconsistency, the harder it is to determine

• No way of knowing when a specification is "complete"

Main Problems are:
Cannot reason definitively about natural language
Cannot be sure that natural language artifacts are consistent with other artifacts
Assurances to stakeholders are shaky

Disciplined Use of Natural Language

• Response to natural language problems of:
  -- imprecision
  -- ambiguity
  -- inconsistency (especially when due to size)
  -- inability to reason effectively and definitively

• Familiar approaches:
  -- Restricted use of reserved terms
  -- Structuring (paragraph numbering, outline form, templates, etc.)

• Other, earlier examples of disciplined use of natural language:
  -- Laws and other legal documents
  -- Recipes
  -- Help systems
A Software Engineering Example:
PSL (Problem Statement Language)

DESCRIPTION:
this process performs those actions needed to interpret
time records to produce a pay statement for each hourly employee.;

KEYWORDS: independent;

ATTRIBUTES ARE:
complexity-level high;

GENERATES: pay-statement, error-listing;

RECEIVES: time-card;

SUBPARTS ARE: hourly-paycheck-validation, hourly-emp-update,
h-report-entry-generates, hourly-paycheck-production;

PART OF: payroll-processing;

DERIVES: pay-statement;

USING: time-card, hourly-employee-record;

DERIVES: hourly-employee-record;

USING: time-card, hourly-employee-record;

USING: time-card, hourly-employee-record;

PROCEDURE: read record, add up hours, multiply by pay rate.....

HAPPENS: number-of-payments TIMES-PER pay-period;

TRIGGERED BY: hourly-emp-processing-event;

TERMINATION-CAUSES: new-employee-processing-event;

SECURITY IS: company-only;

---

Discipline Mechanisms in PSL

- Use of keywords (defined elsewhere in specification)
  --fosters precision, clarity
  --helps support consistency determination: some
    keyword fields have defined relations to others
    (eg. Input-to and output-from)

- Use of templates
  --facilitates determination of completeness
  --fosters clarity
  --facilitates consistency checking

- Use of structure:
  HIERARCHY:
  --Standard practice for dealing with size, complexity
  --Exploits innate human capacity for abstraction

  DATA FLOW:
  --Can determine which data objects flow to what activities

  CONTROL FLOW:
  --Can see that some activities precede/follow others
Evaluation of Disciplined Natural Language

- Big step in the right direction: improvement over straight natural language
- Possible to determine some kinds of consistency thru:
  --mechanisms for reducing ambiguity
  --mechanisms for fostering completeness
  --structuring mechanisms for dealing with complexity

BUT--
- Stilted form reduces clarity: less suitable for some key stakeholder groups
- Some residual reliance on natural language means ambiguity remains
- Size is still a problem: PSL specs (for example) can be huge: consistency determination is long/error prone

Pictorial and Diagrammatic Approaches

- Diagrams composed of visual elements having rigorously defined (definable?) semantics
- Diagrams used as modeling devices
- Diagrams depict key structural aspects of system
- Reduction or removal of natural language
- Clarity improved greatly
- Consistency improved greatly
- Completeness of diagrams facilitated
- Possibility of ambiguity considerably reduced

ALTHOUGH:
- Modifiability significantly reduced
- Completeness impeded by restrictions in semantics defined
  --more on these issues later.....
Example: Elevator Controller

- What is it supposed to do?
  - Stop on every floor it is called to
    » Maybe not so easy: multiple elevators
  - Service users “first come, first served”
    » May conflict with optimal strategies
- These are hard to be precise about, reason about in “plain English”
- Code helps with some of these, but not all
- What should it never do?
  - Allow elevator to move with doors open
  - What else?

Data Flow Diagram

- Current Floor
- Select New Floor
- Get copy of Request List
- New Floor
- Move elevator
- Delete Request Just Satisfied
- Turn off Light
- New Request
- Update Request List
- Button Press
- Turn on Light
- Request List
- New Floor
Control Flow Diagram

```
Read Floor, Direction

DB_Read: curr-length, Floors(), Directions()

Direction = Up

incr <-- 1

yes

Direction = Up

incr <-- -1

no

no

NFloor <-- Floor; increment NFloor;

other car is nearer?

yes

do other car is nearer?

output new floor

no

NFloor = Floors(i)
for any 1 <= i <= curr-length

Directions[i] = incr?

no

other car is nearer?

yes

increment NFloor

no

complement Direction

yes
```

Finite State Machine

```
Doors Closed, Elevator Idle

Request to Move to New Floor

No new request timeout

Doors open, Elevator Stopped

Queue empty

Getting new floor request

Request to Move to new floor

Queueing new floor request

Request to Move to new floor

Arrival at New Floor

Request to Move to new floor

Doors closed, Elevator in Motion

Doors open, Elevator in Motion

Request to Move to new floor

Request to Move to new floor

Doors Closed, Elevator Idle

Request to Move to New Floor

No new request timeout
```

CS 520/620 Spring 2013 Univ. of Massachusetts
Copyright L. Osterweil, all rights reserved
Petri Net

Car going up

Up button Pressed

No Press

Down Button Pressed

Stop at Floor

Go to Next Floor Up

Pictures are not enough

• Want to be able to reason about them
  – Verify relations, properties
• Pictures can leave ambiguous impressions
• How to be sure what they say?
An Example: A Data Flow Diagram

What is this diagram specifying?

[Diagram showing data flow between input, height, check arg. values, area, print area, input width, width, compute area, output error msg, error msg, print error msg]

The power of annotation:

Improper use of notation is not saved by annotation here:

[Diagram showing input, arg 1, func 1, out 1, print, input 2, arg 2, func 3, alt out, print alt, func 2, out 1]
What’s wrong with this diagram

There is ambiguity and misuse of notation here:
• one circle is a test, others are functions
What’s wrong with this diagram

There is ambiguity and misuse of notation here:
• one circle is a test, others are functions
• some edge annotations are data, some predicates

What’s wrong with this diagram

There is ambiguity and misuse of notation here:
• one circle is a test, others are functions
• some edge annotations are data, some predicates
• are multiple arrows in and out “and” or “or”? 
Very Simple DFD

Attempt to make this more precise
But iteration complicates matters

Back to Basics

- Review fundamental Finite Mathematics
  - Set theory
  - Graph theory
  - Predicate Calculus
  - Etc.
Relations:
A RELATION, R, over a set, S = \{s_i\} is a set of tuples
\[ R = \{ r_i \}, \text{ where } r_i = (s_{i,1}, s_{i,2}, \ldots, s_{i,n}) \]

An n-ary relation is a relation where all of the tuples are n-tuples
A Binary relation is a relation where all the tuples are 2-tuples
If \((s_i, s_j)\) is an element of R, then we often write \(s_i \mathbin{R} s_j\)

Another view of relations:
The relation, R, over the set S can be defined as:
\[ R = \{ (s_i, \ldots, s_j) \mid \text{PRED}(s_i, \ldots, s_j) = \text{True}, \text{for some predicate, PRED} \} \]

If the tuples are ordered, the relation is called an ordered relation
If the tuples, \(<t_{i,1}, t_{i,2}, \ldots, t_{i,n}>\) are unordered, the relation is an unordered relation

Some Examples
Let I = \{all integers\},
Define Q = \{ \(x, y, z\) \mid x, y, z are integers and \(y = x**2, z = x**3\) \}

Let S = \{all states of the U.S., S_i\},
Define B = \{ \(S_i, S_j\) \mid S_i and S_j are states that share a border \}

Let L = \{all statements L_i in a program, P\},
Define ImmFol = \{ \(L_i, L_j\) \mid the execution of \(L_i\) may immediately follow the execution of \(L_j\) for some execution of P \}
Some Properties of Relations

Some familiar properties of ordered binary relations, \( R \), over the set \( S = \{s_i\} \):
- **Symmetry**: \( s_i R s_j \implies s_j R s_i \) for all pairs, \( s_i \) and \( s_j \) in \( S \)
- **Reflexivity**: \( s R s \), for all \( s \) in \( S \)
- **Transitivity**: \( s_i R s_j \) and \( s_j R s_k \implies s_i R s_k \), for all \( s_i, s_j \) and \( s_k \) in \( S \)

A relation that is symmetric, reflexive and transitive is called an **equivalence relation**.

If \( R = \{(s_i, s_j)\} \) is transitive, then \( C = \{(s_i, s_j) \mid \text{there exists a sequence, } i_1, i_2, \ldots, i_n, \text{ such that } s_{i_1} R s_{i_2}, s_{i_2} R s_{i_3}, \ldots, s_{i_{n-1}} R s_{i_n} = s_j \} \) is called the transitive closure of \( R \).

- **Antisymmetry**: \( s_i R s_j \implies \neg(s_j R s_i) \) for all pairs, \( s_i \) and \( s_j \) in \( S \)
- **Irreflexivity**: \( s \sim R s \) for all \( s \) in \( S \)

Examples

If \( S = \{\text{all subroutines written in Fortran}\} \) \( s_1 R s_2 \) if and only if \( s_1 \) calls \( s_2 \), then \( R \) is an irreflexive relation.

Let \( PS = \{c_s, \text{ all the statements in a program that consists of a set of modules, } M = \{m_i\}\} \),

\begin{align*}
\text{INMOD} &= \{(c_i, c_f) \mid c_i \text{ and } c_f \text{ appear in the same module } m_i\} \\
\text{INMOD} &\text{ is an equivalence relation}
\end{align*}

The relation \( \text{ImmFol} \) (earlier slide) is not transitive.

Change \( \text{ImmFol} \) to \( \text{Fol} \), by defining \( \text{Fol} = \{(L_1, L_2) \mid \text{the execution of } L_2 \text{ may follow the execution of } L_1 \text{ for some execution of } P\} \).
Examples

If \( S = \{ \text{all subroutines written in Fortran} \} \) \( s_1 R s_2 \) if and only if \( s_1 \) calls \( s_2 \), then \( R \) is an irreflexive relation

Let \( PS = \{ c_e, \text{all the statements in a program that consists of a set of modules, } M = \{ m_i \} \} \),
\[ \text{INMOD} = \{ (c_e, c_f) | c_e \text{ and } c_f \text{ appear in the same module } m_i \} \]
INMOD is an equivalence relation

The relation \( \text{ImmFol} \) (earlier slide) is not transitive

Change \( \text{ImmFol} \) to \( \text{Fol} \), by defining \( \text{Fol} = \{ (L_1, L_2) | \text{the execution of } L_2 \text{ may follow the execution of } L_1 \text{ for some execution of } P \} \)

\( \text{Fol} \) is still not transitive

Graphs

A \textit{Graph}, \( G = (N, E) \) is an ordered pair, consisting of a node set, \( N \), and an edge set, \( E = \{ (n_i, n_j) \} \)

If \( OG = (N, E) \) is an ordered graph with \( E = \{ (n_i, n_j) \} \) then its unordered version, \( UG = (N, U) \), where \( U = \{ <n_i, n_j> \} \)
Graphs as Visualization Aids

- Graphs are mathematical structures with obvious visualizations that seem often to help many stakeholder communities to visualize key relations.

- A graph’s edges visually represent the ordered pairs that compose the relation.

- If the pairs in \( E \) are ordered, then \( G \) is a directed graph, and its edges are depicted with arrowheads.
  - If not, the graph is called an undirected graph.

---

Paths

A path, \( P \), through an ordered graph \( G=(N,E) \) is a sequence of edges, \( (n_{i,1}, n_{j,1}), (n_{i,2}, n_{j,2}), ..., (n_{i,t}, n_{j,t}) \) such that \( n_{j,k-1} = n_{i,k} \) for all \( 2 \leq k \leq n \).

A path, \( UP \), thru an unordered graph \( UG=(N,U) \) is a sequence of edges, \( (<n_{i,1}, n_{j,1}>, <n_{i,2}, n_{j,2}>, ..., <n_{i,t}, n_{j,t}> ) \) such that all of the \( <n_{i,k}, n_{j,k}> \) can be ordered to assure that \( n_{j,k-1} = n_{i,k} \) for all \( 2 \leq k \leq n \).

In either case, \( n_{i,1} \) is called the start node and \( n_{i,t} \) is called the end node.

The length of a path is the number of edges in the path.

A graph \( G \) is connected if and only if, for every pair of nodes, \( n_1, n_2 \), there is path from one of them to the other with \( G \) considered to be an unordered graph.

These graph constructs appeal visually to many stakeholders and often effectively support answering their questions.
**Trees**

A *cycle* in a graph G is a path whose start node and end node are the same.

A *simple cycle* in a graph G is a cycle such that all of its nodes are different (except for the start and end nodes).

If a graph G is *connected* and has no path through it that is a cycle, then the graph is called acyclic.

An acyclic, connected, unordered graph is called a *tree*.

A collection of trees is called a *forest*.

If the unordered version of an ordered connected graph is acyclic, the graph is called a directed tree (in this course we drop this distinction and call both graphs trees).

If the unordered version of an ordered graph has cycles, but the ordered graph itself has no cycles, then the graph is called a Directed Acyclic Graph (*DAG*).

---

**Other Types of Graphs**

A *Multigraph* MG is an ordered pair MG = (N, C) where N is a set of nodes \{n_i\} and C is a collection of pairs of nodes (edges) with repetitions allowed (ie. C can be a multiset).

A *Hypergraph* HG is an ordered pair HG = (N, T) where N is a set of nodes \{n_i\} and T is a set of t-tuples of nodes, where t > 2.

A *Hypermultigraph* is a hypergraph where the set of t-tuples can be a multiset.

A *bipartite* graph BG is an ordered pair, BG = \{BN, E\} where BN is a node set that is the union of two disjoint subsets, \(N_1 \cup N_2\), and no edge in E has both nodes in either \(N_1\) or \(N_2\).

A *bipartite graph* is often called a 2-colorable graph.

A *k-colorable* graph is defined analogously, with BN being the disjoint union of k subsets.
Differences in Graphs Result from Different Choices for Nodes & Relations

- **Hierarchy:**
  -- Models "consists of" or "is a part of"
  -- Key to divide-and-conquer approaches to understanding

- **Data Flow:**
  -- Nodes represent set of sites where data is generated/used
  -- Each edge is a (data generated, data used) node pair

- **Control Flow:**
  -- Nodes represent units of functionality
  -- \((n_1, n_2)\) is an edge in this graph if and only if unit \(n_2\) can execute immediately after \(n_1\) executes (ImmFol relation)

- **Finite State Machines**
  -- Nodes represent all possible different "execution states"
  -- \((s_1, s_2)\) is an edge if and only if it is possible for state \(s_2\) to immediately succeed \(s_1\). Called a transition from \(s_1\) to \(s_2\)
  -- Edges annotated with transition condition
  -- Annotations are relations too
  -- Juxtaposition of annotation atop what it is annotating

- **Petri Nets**
  -- Multiple node and edge types in the same diagram

---

Flowgraphs

Let \(S = \{\text{all statements } s_i \text{ in a program, } P\}\)

Let \(\text{ImmFol} = \{ (s_i, s_j) \mid \text{The execution of } s_i \text{ immediately follows the execution of } s_j \text{ for some execution of } P \}\)

Then: If \(FG = (S, \text{ImmFol})\), \(FG\) is called the flowgraph of \(P\)

\(FG\) is an ordered graph

Every execution sequence (ie. the sequence in which the statements of \(P\) are executed for a given execution of \(P\)) corresponds to a path in \(FG\).

However---the converse is not true. A path through \(FG\) may not correspond to an execution sequence for \(P\)

A loop in \(P\) appears as a cycle in \(FG\)
**Callgraphs**

Let $\text{PROC} = \{\text{procedures } S_i \text{ that the program } P \text{ comprises}\}$

Let $\text{CALLS} = \{(S_i, S_j) | S_j \text{ is directly invoked from } S_i \text{ during some execution of } P\}$

Then $\text{CG} = (\text{PROC}, \text{CALLS})$ is called the Call Graph of $P$

CG is a directed graph

If $P$ is written in a language that does not allow recursion, then CG will be acyclic

A cycle in CG indicates that the nodes along the cycle participate in a recursive calling chain

**NOTE:** DEPICTIONS OF THESE GRAPHS MAY BE SUPERIMPOSED OVER EACH OTHER TO CLARIFY (?!) THINGS

---

**DATA FLOW DIAGRAMS**

- Capture system functionality: What does system do? How?
- Basic components of a data flow diagram:
  - Nodes, represented by circles (boxes), are functional units
  - Edges, represented by arrows, are data flows between units
  - Both augmented by separate annotation relations
  - Boxes (sometimes circles), represent I/O data

```
input
height

check arg. values

compute area

area

print area

input width

width

args bad

output error msg.

error msg

print error msg

There is ambiguity and misuse of notation here:
- one circle is a test, others are functions
- are multiple arrows in and out “and” or “or”?
```
Formalizing DFDs

- **DataFlow(i, j)** if node i creates data that node j uses
- **INPUT**—set of nodes {input_i} such that input_i is a provider of input from external source
- **OUTPUT**—set of nodes {output_i} such that output_i is a conveyor of computed artifacts to external source
- **EdgeAnnotation(e, operand)** where operand is the identification of the artifact that flows along edge e
  - Preferably the data artifact is defined rigorously
- **NodeAnnotation(n, text)** if the string text describes the functioning of node n

Questions this helps answer:
- Why create this data? Who uses this data? What results does the end user see? What does the end user have to input?
- Questions this can’t answer: What is the exact sequence of events? How does a node do its job?

Adding Hierarchy

- Supports increased detail
- Increased precision too
- But creates complications
Consistency is a principal concern

- Are the diagrams consistent with each other?
- Top view consistent with elaborations?
  - Arrows consistent
  - Data flows consistent
  - Other semantics?
- Invitation to subtle errors