CSE503: Software Engineering

David Notkin
University of Washington
Computer Science & Engineering
Spring 2006

1

A Michael Jackson presentation

- The following slides are from his keynote at ICSE 1995

2

The World

and

The Machine

Michael Jackson

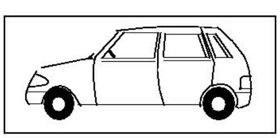
MAJ Consulting Ltd and AT&T Bell Laboratories
ICSE-17 Seattle 28th April 1995

Ways of Looking at Software

- 'Programming should be *literate*'
- ' ... they regarded my programs as *logical poems* ...'
- 'The goal of any system is *organisational change*'
- 'Software development is *engineering*'
  - Because we make *machines* to serve useful purposes in the *world*
    - The *problem* is in the World
    - The Machine is the *solution*

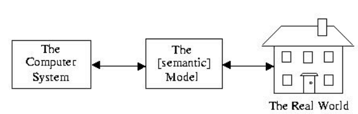
WHAT and HOW

- WHAT does an automobile do?



  - It carries *people* and their *baggage*, travelling over *roads* where its *driver* directs it to go
- WHAT is in the *world*, HOW is in the *machine*

The Machine, the Model, and the World



- *Formal Methods* concern the *left* arrow
- We have no theory for the *right* arrow

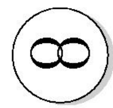
Brian Cantwell Smith; *The Limits of Correctness*

Talking about the World and the Machine

- To develop software we must talk both about the World and about the Machine
- But it's hard to maintain the right balance between these two universes of discourse
  - The relationship between them is varied and often subtle
  - Often we have personal preferences to exploit or resist

Three Topics and a Button

- 4 Facets of the Relationship
- 4 Kinds of Denial of the World
- 4 Principles for Accepting the World
- a Button:



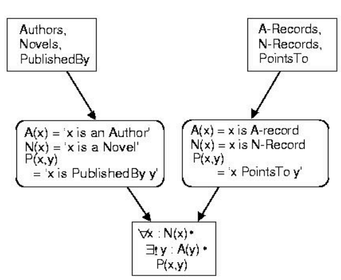
4 Facets of the Relationship

- Modelling:
  the Machine as a *model* of the World
- Interface:
  what the Machine *shares* with the World
- Engineering:
  how the Machine *changes* the World
- Problem:
  the *structure* the Machine must have to fit the problem in the World

Modelling a Reality

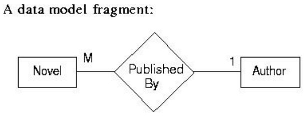
- 'An SADT system *description* is called a "model" …'
- R L Ackoff (*Scientific Method,* 1962):
  - *Iconic* models — pictures, 3-D representations, eg a child's model farm
  - *Analytic* models — manipulable formal descriptions, eg differential equations forming an economic model
  - *Analogic* models — an analogous reality, eg an electrical network modelling the flow of water in pipes
- Software models are analogic: eg, a database, an assemblage of objects, a process network

The Machine As a Model of the World

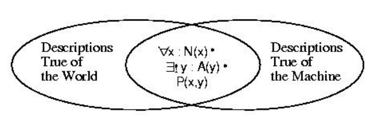


Modelling and ⚭

- A data model fragment:

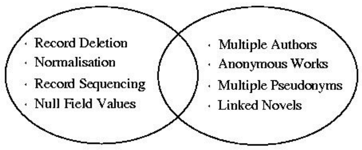


- Three sets of descriptions:



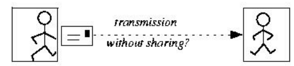
Non-Modelling and OO

- Both the World and the Machine have properties that are private and not shared

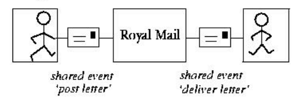


The Machine – World Interface

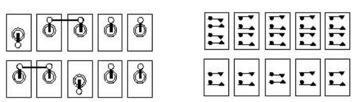
- Shared phenomena: *events*, other *shared individuals*, *facts* visible in both domains
- No communication without sharing:



is 'really' ...



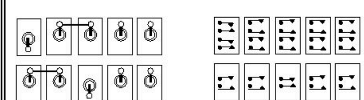
Shared Phenomena



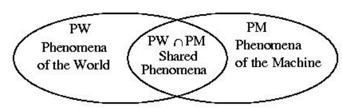
Operator's Panel Domain　　Circuits and Contacts Domain

- Shared phenomena:
  - Levers — Switches
  - FlipUp events — TurnOff events
  - FlipDown events — TurnOn events
- Private phenomena:
  - Links | Contacts
  - LinkedBy (x:Lever, y:Link) | LocatedOn (x:Contact, y:Switch)

Shared Phenomena and OO



- The shared phenomena are in the (small) intersection between two sets of phenomena:



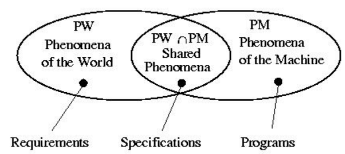
Modelling and Shared Phenomena

- *Sharing phenomena* and *modelling* are different relationships between the Machine and the World
  - Shared phenomena → modelling
    - Any description that is true of the shared phenomena is a shared descriptions
  - But ...
  - ... ¬ (modelling → shared phenomena)
    - The database shares no phenomena with the reality it models

Engineering: Requirements, Specifications, and Programs

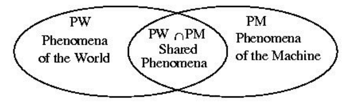
- The purpose of the Machine is to *change* the World: this is the *requirement*
- The required changes are expressible entirely in terms of phenomena of the World ...
- ... but not usually entirely in terms of phenomena shared with the Machine
- The final engineering product:
  - Machine behaving according to the *program* ...
  - ... thus satisfying the *specification* and ...
  - ... thus ensuring achievement of the *requirement*

Requirements, Specifications, Programs



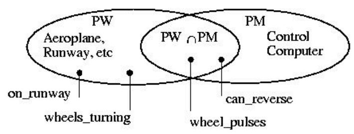
- A specification is also a requirement
- A specification is also a program

Engineering and OO



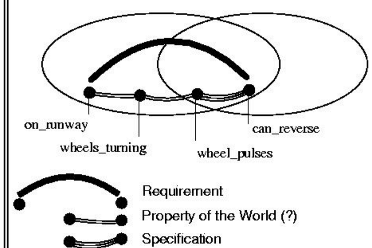
- Programs can satisfy specifications only by virtue of *properties of the machine* (p/l semantics)
- Specifications can satisfy requirements only by virtue of *properties of the world*
- The engineering is in determining, describing and exploiting the properties of the world

A Little Engineering Example



- R: on_runway $\leftrightarrow$ can_reverse
- $D1$: wheel_pulses $\leftrightarrow$ wheels_turning
  $D2$: wheels_turning $\leftrightarrow$ on_runway
- S: can_reverse $\leftrightarrow$ wheel_pulses
- We have: S, D1, D2 $\vdash$ R — is it enough?

Properties of the World



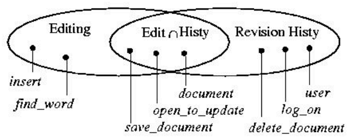
The Problem Facet of the Relationship

- Solution structure should reflect problem structure
  - There's less need for invention
  - It's easier to validate the solution
- Traditional solution structures are often *hierarchical* and *homogeneous* ...
  - Procedure hierarchies, class hierarchies, layered abstract machines, process/dataflow structures
- ... but the World rarely exhibits such structures

A Simple Editing Tool

- Three requirements:
  - *Editing* allows users to create and edit texts
  - *GUI* provides convenient and efficient operation
  - *Revision History* provides progress reporting by users and texts
- The requirements are related by conjunction:
  - *Editing* $\wedge$ *GUI* $\wedge$ *Revision History*
- The requirements *share phenomena*

Two Requirements Sharing Phenomena



Problem Structures

- Problems are usually structured as subproblems that are:
  - heterogeneous
  - related by superimposition
  - pinned together at shared phenomena
- The appropriate metaphor is …
  - … not assemblies and sub-assemblies
  - … but CYMK separations in colour printing

The World and Us (1)

"The world is too much with us …"
— *William Wordsworth*

4 Kinds of Denial

- How we may deny our involvement
  - Denial by Prior Knowledge
  - Denial by Hacking
  - Denial by Abstraction
  - Denial by Vagueness

Denial by Prior Knowledge

"We don't need a requirements capture phase. The problem is already well-defined; our task is merely to solve it."

- Automobile designers don't have a requirements capture phase …
  - The car shall be able to travel over snowdrifts and under water
  - The car shall be able to lift a load of 5 tons
  - The car shall accommodate 10 passengers each of weight up to 500 pounds
- … it would be called 'Rethinking the Motor-car'

Denial by Prior Knowledge

- Legitimate only in applications that are both *specialised* and *standardised*
- Both bridge-design and automobile design are *specialised*
- But only automobile design is *standardised* (human beings, roads and baggage don't vary much)
- Bridge design is not *standardised* (each location has unique characteristics)

Denial by Hacking

- Computers are beautiful and fascinating

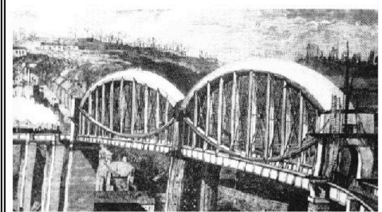
" … Miss Byron, young as she was, understood its working and saw the great beauty of the invention."
Mrs De Morgan, on Ada's visit to Babbage, 1828

- Applications are often much less interesting

"I came into this job to work with computers, not to be an amateur stockbroker."
Member of failed development team, 1993

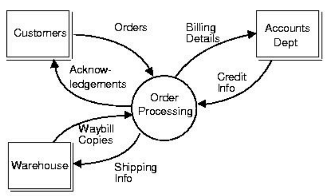
- The Machine is the developers' own creation; the World is not

The Royal Albert Bridge, Saltash



I K Brunel, Engineer, 1849

Looking at the Problem Context



- Which is the World? Which is the Machine?
- Which do you describe at the next level of DFDs?

Denial by Abstraction

"We come now to the decisive step of mathematical abstraction: we forget what the symbols stand for."
Hermann Weyl, quoted by Abelson & Sussman

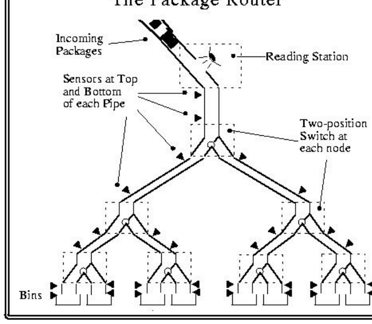
- Abstraction is a valuable intellectual tool …
- … but it must not be a rule of life for software developers
- Too much abstraction blinds you to the nature of many problems

Doing Justice to the Problem

"One tribe always tells the truth and the other always lies. A traveller meets two men, and asks the first: 'Are you a truth teller?'. The reply is 'Goom'. The second says: 'He said Yes, but he is lying'.
Martin Gardener, 2nd Book of Puzzles

- Abstract answer:
  "The reply must always be Yes; so the second man is a truth-teller, and the first is a liar"
- *Lucy Jonelis*' answer:
  "The first man clearly can't speak English: 'Goom' must mean 'What?' or 'Welcome to our land'. So the second man is a liar, and the first is a truth-teller."

The Package Router



Denial by Vagueness

- Central technique:
  - Describe the Machine, but imply that you're describing the World
- Prerequisite:
  - Avoid saying explicitly what is being described
- Facilitators:
  - The modelling relationship (the same description is true of both)
  - The shared phenomena at the interface (two sides of the same penny, isn't it?)

The System and the Real World

" … the Z approach is to construct a specification document which consists of a judicious mix of informal prose with precise mathematical statements. … the informal text can be consulted to find out what aspects of the *real world* are being described…. The formal text in the other hand provides the precise definition of the *system* and hence can be used to resolve any ambiguities present in the informal text."

- Machine = *system*? World = *real world*?
- Which is being described?

Talking About the World: 4 Principles

von Neumann's principle

- *Knowing what you're talking about*

The principle of reductionism

- *Finding the solid ground*

The Shanley principle

- *Recognising versatility*

Montaigne's principle

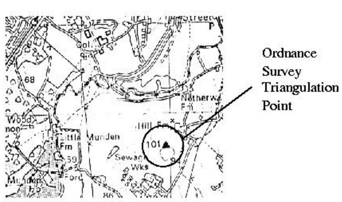
- *Minding your language*

von Neumann's Principle

"There is no point in using exact methods where there is no clarity in the concepts and issues to which they are to be applied."
von Neumann & Morganstern: Theory of Games

- Designations
  - Mother(x,y) ≈ 'x is the genetic mother of y'
  - Formal term ≈ recognition rule
  - Anticipate interventions of the form: "It all depends on what you mean by *mother*"

Aligning a Description

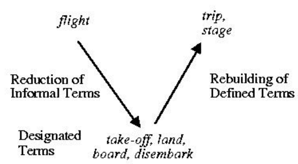


- Designated terms and phenomena are like triangulation points on the map and on the ground

The Principle of Reductionism

- In any informal world many terms — often nouns in English — are obviously important …
  - in telephony: *calls*
  - in a meeting-scheduling system: *meetings*
  - in an airline system: *flights*
- … but difficult or even impossible to designate
- They must be reduced to elementary designated phenomena — often *events*

Reducing Domain Concepts



- The rebuilt defined terms are not the original informal terms
- Definition is not designation

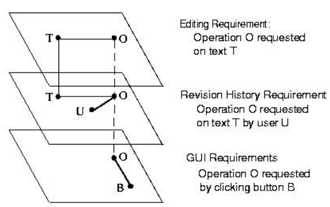
The Shanley Principle

"In civil engineering design it is presently a mandatory concept known as the Shanley Design Criterion to collect several functions into one part."
Pierre Arnoul de Marneffe, cited by D Knuth, 1974

- 1940-1945 rockets had separate components for fuel tank, outer skin, body frame
- Saturn-B had a tubular body that was at once its fuel tank, outer skin, and body frame

- It may (or may not) be good to engineer Machines in this way, but the World is certainly like this!
  - No class hierarchy, no strong typing!

Shanley and Many Descriptions



- One description is not enough

Montaigne's Principle

"The greater part of this world's troubles are due to questions of grammar."

- Demanded for some Government contracts:

"Absolute tense 'shall': a binding, measurable requirement ....
"Future tense 'will': a reference to the future, ... not under control of the system being specified.
"Present tense: for all other verbs ...."

- The distinction is not of *tenses*, but of *moods*
  - Optative: *desired* in the World
  - Indicative: true *regardless* of the Machine

Indicative and Optative

- Natural language distinctions are impractical:
  - "I shall drown, no-one will save me!"
  - "I will drown, no-one shall save me!"
- Mood of a sentence in development changes with its context:
  - In handling the *Revision History* requirement, the *Editing* requirement should be treated as satisfied — not optative but indicative
- So indicative and optative sentences should be kept apart in separate descriptions

Three Topics and a Button

- 4 Facets of the Relationship
  - The Machine as a *model* of the World
  - The interface of *shared phenomena*
  - *Engineering* the World and the Machine
  - Problem and solution *structures*
- 4 Kinds of Denial of the World
- 4 Principles for Accepting the World

The World and Us (2)

"I accept the universe"
— *Margaret Fuller*

"By Gad! she'd better!"
— *Thomas Carlyle*

Abstract data types

- Abstract data types (ADTs) are a common foundation for software development
  - They grew out of Parnas' notion of information hiding, which we'll cover during our design lectures
  - Very roughly, an encapsulated type or a class: a set of procedures (methods) that are the only way to access and manipulate encapsulated data
- ADTs are commonly specified by
  - Natural language comments associated with
  - Signatures of the procedures; for example,
  - `void copyIntBuf(int *pin,int *pout,int len)`

Algebraic specifications

- Algebraic specifications provide a mathematical framework for specifying ADTs
- The intent is to provide clear and well-defined semantics for the operations (procedures), rather than depending on natural language associated with precisely defined syntax
- These define the specification of the abstract operations – defining the equivalence of the implementation with the abstraction is a separate activity

Algebras: roughly

- A set of objects
- A set of rules, called axioms, for determining the equality among those objects
- "K-12" algebra
  - Set of objects is the real numbers
  - x*(y+z) = x*y + x*z
  - x+y=y+x
  - …

Algebraic specification for ADT

1. The name of the *sort* (roughly, the type) being specified
2. The signatures of the primitive operations
3. The axioms

- There are a number of languages that support algebraic specification, including Anna, Clear, Larch, OBJ, …

Sort

- A sort is a set of values
  - roughly a "type" or "class"
  - Ex: integers, stacks of integers, strings, complex numbers, …
- The *sort of interest* is the one that is being defined by a particular specification
- To define this specification may require other sorts (we'll see an example)
- This approach induces a hierarchy of sorts

Signatures

- The name of the operator
- The types of its parameters
- The return type

- Like programming language signatures, but usually represented more abstractly
  - `push: Stack x Elem -> Stack`
  - `+: Integer x Integer -> Integer`
  - `Round: Real -> Integer`
- May look semi-familiar to those who studied ML in 505

55

Axioms

- Rules that must hold true in any legal implementation of the sort

56

Example: queue

- Signature
  - `create: -> Queue`
  - `add: Queue x Element -> Queue`
  - `remove: Queue -> Queue`
  - `front: Queue -> Element`
- Axioms
  - `front(add(create(),x)) = x`
  - `front(add(add(q,x),y)) = front(add(q,x))`
  - `remove(add(create(),x)) = create()`
  - `remove(add(add(q,x),y)) = add(remove(add(q,x)),y)`

57

Conditional axioms

```
front(add(q,i)) =
   if (IsEmpty(q))then i
   else front(q);
```

- In some cases (not necessarily this one) one can increase the clarity with conditional axioms

58

Operations

- Usually separated into
  - Constructors (that create an instance of the sort)
  - Accessors (that take an instance of the sort as a parameter and return an element from a supporting sort)
  - Modifiers (that take an instance of the sort as a parameter and return a modified instance of it)

59

Issues

- Equality: two elements in a sort are equal if and only if all operations applied to them produce equal results
  - Closely related to the rewriting in the lambda-calculus
  - Inequality is defined as the inability to prove equality
- Consistency?
  - Roughly, can we show that the axioms cannot be used to prove “false”?
- Completeness?
  - Roughly, does it represent all the values (e.g., queues) that we intended?

60

Another example: signatures

```
algebra StringSpec;
  sorts String, Char, Nat, Bool;
  operations
    new: () -> String
    append: String, String -> String
    add: String, Char -> String
    length: String -> Nat
    isEmpty: String -> Bool
    equal: String, String -> Bool
```

61

```
StringSpec generated by [new, add]
for all [s1, s2, s3: String; c: Char]

isEmpty (new()) = true;
isEmpty (add(s1,c)) = false;
length (new()) = 0;
length (add(s1,c)) = length (s1) + 1
append (s1, new()) = s1
append (s1, add(s2,c)) = add
         (append(s1,s2), c)
equal (new(), new()) = true
equal (new(), add(s1,c)) = false
equal (add(s1,c), new()) = false
equal (add(s1,c), add(s2,c)) = equal(s1,s2)
```

62

Pros of algebraic specifications

- Language independent
- Implementation independent
- Nicely matched to ADTs
- Strong mathematical foundation
- Suited to automation of the underlying theorem proving
- Can "electrify" the specifications by tracing rewriting

63

Cons of algebraic specifications

- Difficult to deal with procedures that have side effects, reference parameters, multiple returns, etc.
- Not all interesting behaviors are expressed via equality
- The limits of notation can lead to messy and complicated specifications

64

C.A.R. Hoare, 1988

Of course, there is no fool-proof methodology or magic formula that will ensure a good, efficient, or even feasible design. For that, the designer needs experience, insight, flair, judgment, invention. Formal methods can only stimulate, guide, and discipline our human inspiration, clarify design alternatives, assist in exploring their consequences, formalize and communicate design decisions, and help to ensure that they are correctly carried out.

65

Model-oriented specifications

- Model a system by describing its state together with operations over that state
  - An operation is a function that maps a value of the state together with values of parameters to the operation onto a new state value
- A model oriented language typically describes mathematical objects (e.g. data structures or functions) that are structurally similar to the required computer software

66

Z ("zed")

- Probably the most widely known and used model-based specification language
- Good for describing state-based abstract descriptions roughly in the abstract data type style
- Based on typed set theory and predicate logic
- A few commercial successes
  - I'll come back to one reengineering story afterwards

67

The basic idea

- Static schemas
  - States a system can occupy
  - Invariants that must be maintained in every system state
- Dynamic schemas
  - Operations that are permitted
  - Relationship between inputs and outputs of those operations
  - Changes from state to state

68

Illustrative example (Zeil)

- I'll sketch out a standard Z-style example
- Z relies heavily on non-standard characters and formatting, which I will only approximate
  - The reading includes a similar example
  - And uses the Z notation

69

Phone directory: static schema

- A static schema has three parts
  - A name
  - A set of declarations that define the state
  - A set of invariants that constrain all legal states
- `PhoneDB`
  - `members:      ` **`P`** `Person`
    `telephones:   Person <-> Phone`
  - `dom telephones SUBSET-OF members`

70

Type of `members:` **`P`** `Person`

- Atomic elements, like `Person` and `Phone`, represent sets of values
- **`P`** `Person` represents the power set of `Person`, the set of all sets taken from `Person`
- So, `members` is one of those: a set of `Person`

71

Type of `telephones: Person <-> Phone`

- `telephones` is a relation between `Person` and `Phone`
- That is, it is a set of pairs, where the first element is taken from `Person` and the second is taken from `Phone`

72

Invariant

`dom telephones subset-of members`

- This is an invariant that defines a constraint on all legal states of `PhoneDB`
- The domain (the set of first elements in the pairs) of `telephones` must only contain elements that are in `members`
- Without this invariant, there would be no restrictions nor relationship between `members` and `telephones`
- When we define operations that can modify the state of `PhoneDB`, they are obligated to maintain (prove) that this invariant is maintained

Example: a legal `PhoneDB` state

- `Person: { hank, hellmut, bob, paul, jean-loup, ed, david}`
- `Phone: { 5-3798,3-2121,3-5010,3-4755,5-1376, 3-1695,3-2969,3-6175,6-4368}`
- `members: { hank, hellmut, jean-loup, ed, david }`
- `telephones: { (hank |->3-6175), (hellmut |-> 3-6175), (jean-loup |-> 5-1376), (ed |-> 3-4755), (david |-> 5-3798) }`
- `|->` is a "maplet", essentially a pair

A few notes on the example

- The elements of `Person` and `Phone` are atomic: they have no required syntax nor semantics
- `telephones` is a relation, not a function; so adding the tuple `(david |-> 3-1695)` to it is perfectly legal
- And it already contains two tuples with the same range (second element of the pair): `hank` and `hellmut` share `3-6175`
- Z, of course, has and uses functions (both partial and total)
  - But they are notational conveniences, since one can write invariant that constrain relations to be functions

Example: an illegal `PhoneDB` state

- `Person: { hank, hellmut, bob, paul, jean-loup, ed, david, jonathan }`
- `Phone: { 5-3798,3-2121,3-5010,3-4755,5-1376, 3-1695,3-2969,3-6175,6-4368,1-2345}`
- `members: { hank, hellmut, jean-loup, ed, david }`
- `telephones: { (hank |->3-6175), (hellmut |-> 3-1675), (jean-loup |-> 5-1376), (ed |-> 3-4755), (david |-> 5-3798), (jonathan |-> 1-2345) }`
- This would be perfectly legal in the absence of the invariant: but `jonathan`, while being an element of `Person`, is not an element of `members`

Dynamic schema: specifying state transitions

- Static schema specify legal states
- But we also need to specify operations that transform one legal state into another legal state
- Dynamic schema have (just like static schemas)
  - A name
  - A set of declarations
  - A set of invariants that relate the set of declarations to one another
- However, the declarations used are richer

Example declaration

- Declaration: `DELTA PhoneDB`
- A `DELTA` declaration introduces pre- and post-states for each of the declarations in the named schema
  - `members` and `members'`
  - `telephones` and `telephones'`
- The unprimed names represent the pre-states and the primed names the post-state
- Any invariants must hold on the pre-state and then again on the post-state
  - `dom telephones SUBSET-OF members`
  - `dom telephones' SUBSET-OF members'`

Example dynamic schema

- Name: `AddEntry`
- Declarations:
  - `DELTA PhoneDB`
    `name? : Person`
    `newnumber?: Phone`
- Invariants
  - ```
    name? IS-ELEM members
    (name? |-> newnumber?) NOT-ELEM telephones
    telephones' =
             telephones UNION (name? |-> newnumber?)
    members' = members
    ```
- This may or may not be what you expect from `AddEntry`, but it is clear about key issues: for instance, it only adds new phone numbers for existing members

79

What if…

- ```
  telephones' =
           telephones UNION (name? |-> newnumber?)
  ```
  was replaced with
- `(name? |-> newnumber?)IS-ELEM telephones'`

80

Returning information

- `GetNumber`
- ```
  XI PhoneDB
  name?: Person
  number!: P Phone
  ```
- ```
  name? IS-ELEM members
  number! = { n : Phone
     | ((name? |->  n) IS-ELEM telephones)}
  ```
- The `XI` declaration is the equivalent of `DELTA` along with the following invariants that guarantee no change to the `PhoneDB` declarations
  - `members = members'`
  - `telephones = telephones'`

81

Error conditions

- Note that the dynamic schema we've seen so far just specify what happens in the "good cases"
  - Nothing is specified for the error conditions
  - What happens with `AddEntry(mork,0-1010)`?

82

Specify in separate schema

- `NotMember`
- ```
  XI PhoneDB
  name? : Person
  report! : Report
  ```
- ```
  name? NOT-ELEM members
  report! = 'not a member'
  ```

83

But it's still entirely separate

- `Success`
- `report! : Report`
- `report! = 'OK'`
- And then the coolest thing in Z…(at least notationally) is the schema calculus
- ```
  AddEntryWithError ==
      (AddEntry AND Success) OR
       NotMember
  ```
- This is the same as a dynamic schema in which the three schema are commingled according to the stated logic
  - They are "pinned" together by shared names

84
  ```

Z/CICS

- Z was used to help develop the next release of IBM's CICS/ESA_V3.1, a transaction processing system
  - Integrated into IBM's existing and well-established development process
  - Many measurements of the process indicated that they were able to reduce their costs for the development by almost five and a half million dollars
  - Early results from customers also indicated significantly fewer problems, and those that have been detected are less severe than would be expected otherwise

1992 Queen's Award for Technological Achievement

- "Her Majesty the Queen has been graciously pleased to approve the Prime Minister's recommendation that The Queen's Award for Technological Achievement should be conferred this year upon Oxford University Computing Laboratory.
- "Oxford University Computing Laboratory gains the Award jointly with IBM United Kingdom Laboratories Limited for the development of a programming method based on elementary set theory and logic known as the Z notation, and its application in the IBM Customer Information Control System (CICS) product. ...
- "The use of Z reduced development costs significantly and improved reliability and quality. Precision is achieved by basing the notation on mathematics, abstraction through data refinement, re-use through modularity and accuracy through the techniques of proof and derivation.
- "CICS is used worldwide by banks, insurance companies, finance houses and airlines etc. who rely on the integrity of the system for their day-to-day business."

Pros and cons?

- Your turn…