REQUIREMENTS FOR SOFTWARE TOOLS TO SUPPORT REUSE OF SPECIFICATIONS AND DESIGNS

Paul T. Ward
Software Development Concepts
New York, NY

Paul Ward is a principal partner of Software Development Concepts, a New York City based firm providing consulting and training to organizations that develop real-time systems. He has more than 25 years experience as a systems developer, project leader, methods developer, and consultant. He is co-developer of the Ward-Mellor notation which is implemented in many real-time CASE products. Among his books, *Structured Development for Real-Time Systems* has become a standard text for software engineers.
1. Reuse of Code versus Reuse of Specifications and Designs

2. Do Current CASE Tools Support Reuse?

3. A Proposed CASE Product to Support Reuse
   3.1 Introduction
   3.2 Library Organization
   3.3 Program Organization

Credits

The material presented here is based on the pioneering work of Professor David Harel of the Weizmann Institute of Science.

An introduction to this work may be found in Harel's article "On Visual Formalisms" in Communications of the ACM, Vol. 31, No. 5, May 1988.

1. Reuse of Code versus Reuse of Specifications and Designs

- Effective software engineering requires reuse of development products.
- Reuse of specifications and designs is more robust than reuse of code.
- CASE tools for specification and design must assist developers with reuse.
1. Reuse of Code versus Reuse of Specifications and Designs

- Effective software engineering requires reuse of development products.
- Reuse of specifications and designs is more robust than reuse of code.
- CASE tools for specification and design must assist developers with reuse.

Credits

The material presented here is based on the pioneering work of Professor David Harel of the Weizmann Institute of Science.

An introduction to this work may be found in Harel's article "On Visual Formalisms" in Communications of the ACM, Vol. 31, No. 5, May 1988.
Effective Software Engineering Requires Reuse

- Most systems development organizations produce families of related systems.
- Other engineering disciplines which build families of products (i.e., automotive engineering) commonly reuse components.
- Software engineering will not be highly productive until reuse is well established.

Reuse of Code is Difficult

TYPICAL: Before Implementation

Diagram:
- Knowledge of Application
- Knowledge of Resources (e.g., Cost(Time) > Cost(Space))
- "Debugging" (i.e., Re-Specification and Re-Design)
- Specification (I/I)
- Design (I/I)
- Program

I/I = Informal or Internal
Effective Software Engineering Requires Reuse

- Most systems development organizations produce families of related systems.
- Other engineering disciplines which build families of products (i.e., automotive engineering) commonly reuse components.
- Software engineering will not be highly productive until reuse is well established.

Reuse of Code is Difficult

TYPICAL: Before Implementation

\[ \text{SPECIFICATION (I/I)} \rightarrow \text{DESIGN (I/I)} \rightarrow \text{PROGRAM} \]

\[ \text{KNOWLEDGE OF APPLICATION} \]

\[ \text{KNOWLEDGE OF RESOURCES (E.G., COST(TIME) > COST(SPACE))} \]

\[ \text{"DEBUGGING" (I.E., RE-SPECIFICATION AND RE-DESIGN)} \]

I/I = INFORMAL OR INTERNAL
TYPICAL: After Implementation

- Specification is out of date and useless.
- Design is out of date and useless.
- Program code is unreadable by anyone except original programmer

Reuse of Specifications and Designs is Easier
TYPICAL: After Implementation

- Specification is out of date and useless.
- Design is out of date and useless.
- Program code is unreadable by anyone except original programmer

Reuse of Specifications and Designs is Easier
2. Do Current CASE Tools Support Reuse?

Current CASE tools have some limited reuse support, such as cut-and-paste features.

However, most CASE tools make it difficult to reuse pieces of one model in another, or to reuse pieces more than once in the same model.

This is due to:

- Paper-and-pencil modeling paradigm
- Top-down bias
- Global name space

Paper-and-Pencil Modeling Paradigm

- A model consists of a number of separate diagrams.
- Pieces of the model are connected across diagrams.
- Accessing a piece of a model means retrieving its diagram.
2. Do Current CASE Tools Support Reuse?

Current CASE tools have some limited reuse support, such as cut-and-paste features.

However, most CASE tools make it difficult to reuse pieces of one model in another, or to reuse pieces more than once in the same model.

This is due to:

- Paper-and-pencil modeling paradigm
- Top-down bias
- Global name space

Paper-and-Pencil Modeling Paradigm

- A model consists of a number of separate diagrams.
- Pieces of the model are connected across diagrams.
- Accessing a piece of a model means retrieving its diagram.
EXAMPLE: Moving this collection of pieces to a single diagram on another model requires 3 retrievals, 3 copies, and 3 pastes.

Top-Down Bias

- A model is a pure hierarchy of diagrams.
- Diagrams and pieces of diagrams are indexed by a number keyed to the hierarchy.
EXAMPLE: Moving this collection of pieces to a single diagram on another model requires 3 retrievals, 3 copies, and 3 pastes.

Top-Down Bias

- A model is a pure hierarchy of diagrams.
- Diagrams and pieces of diagrams are indexed by a number keyed to the hierarchy.
EXAMPLE: Illegal! Diagram 1 and its lower-level diagrams must be duplicated and renumbered.

Global Name Space

Each piece of the model must have a unique name.
EXAMPLE: Illegal! Diagram 1 and its lower-level diagrams must be duplicated and renumbered.

Global Name Space

Each piece of the model must have a unique name.
3. A Proposed CASE Product to Support Reuse

3.1 Introduction

Basic features (required for any effective CASE tool)

- State-of-the-art user interface (electronic desktop, color)
- Support of visual formalisms
- Execution and property analysis of models
- Code generation from models
3. A Proposed CASE Product to Support Reuse

3.1 Introduction

Basic features (required for any effective CASE tool)

- State-of-the-art user interface (electronic desktop, color)
- Support of visual formalisms
- Execution and property analysis of models
- Code generation from models
Additional features which specifically support reuse:

- Distinction between "libraries" and "programs"
- Definition, browsing, and instantiation capabilities
- Model transformation capabilities

Distinction between "Libraries" and "Programs"

Since models are abstract programs, the CASE tool should borrow features from comprehensive programming environments.

- "Libraries" are used to define, store, and retrieve model components
- "Programs" consist of executable assemblies of components
Additional features which specifically support reuse:

- Distinction between "libraries" and "programs"
- Definition, browsing, and instantiation capabilities
- Model transformation capabilities

Distinction between "Libraries" and "Programs"

Since models are abstract programs, the CASE tool should borrow features from comprehensive programming environments.

- "Libraries" are used to define, store, and retrieve model components
- "Programs" consist of executable assemblies of components
Definition, Browsing, and Instantiation

- Definition: Adding a component to a library, by direct entry or by copying a program component
- Browsing: Finding a desired component in a library
- Instantiation: Copying a library component into a program

Model-Building Paradigm 1: Assemble Program from Libraries

![Diagram showing the process of assembling a program from libraries with menu-based and visual browsing, as well as instantiation steps.]
Definition, Browsing, and Instantiation

- Definition: Adding a component to a library, by direct entry or by copying a program component
- Browsing: Finding a desired component in a library
- Instantiation: Copying a library component into a program

Model-Building Paradigm 1: Assemble Program from Libraries

![Diagram showing libraries and program components](image-url)
Model-Building Paradigm 2: Define Library Components from Program

3.2 Library Organization

QUESTION: What is the best organization of library components?

ANSWER: Classes of objects

QUESTION: What is the best representation of library components?

ANSWER: A higraph-based visual formalism
3.2 Library Organization

QUESTION: What is the best organization of library components?

ANSWER: Classes of objects

QUESTION: What is the best representation of library components?

ANSWER: A higraph-based visual formalism
Nesting of Component Classes and Subclasses

The component object names are qualified by the composite object name – e.g., for a professor named Sam, SAM.NAME and SAMPHONE.

Visualizing Operations

Operations permit communications with objects of a class

Operation names are qualified by the name of the containing object – an integer named x has operations x.SET, x.ADD, ...
Nesting of Component Classes and Subclasses

The component object names are qualified by the composite object name – e.g., for a professor named Sam, SAM.NAME and SAM.PHONE.

Visualizing Operations

Operations permit communications with objects of a class

Operation names are qualified by the name of the containing object – an integer named x has operations x.SET, x.ADD, ...
The Object Memory

The object memory contains one or more values that constitute the "state" of objects of a particular class.

When an operation is invoked, it may use or retrieve values from the object memory.

Messages

Messages are composite classes that represent communication among objects.

The components of a message are (optionally) displayed in text form.
The Object Memory

The object memory contains one or more values that constitute the "state" of objects of a particular class.

When an operation is invoked, it may use or retrieve values from the object memory.

Messages

Messages are composite classes that represent communication among objects.

The components of a message are (optionally) displayed in text form.
Control Operations

A control operation may be defined on an object that:

- is associated with a protected variable of class `#STATE` in the object memory (read-only to other operations)
- is defined by a statechart
- may access the object memory and receive messages from the other object operations
- may start and stop the other object operations
Visualizing Classes versus Objects

A control operation may be defined on an object that:

- is associated with a protected variable of class #STATE in the object memory (read-only to other operations)
- is defined by a statechart
- may access the object memory and receive messages from the other object operations
- may start and stop the other object operations
3.3 Program Organization

QUESTION: What is the best organization of program components?

ANSWER: There is no "best" organization. It is useful to view a program both as a network of objects and as a network of functions.

QUESTION: What is the best representation of program components?

ANSWER: A modified version of the visual formalism used for the library.
EXAMPLE:

3.3 Program Organization

QUESTION: What is the best organization of program components?

ANSWER: There is no "best" organization. It is useful to view a program both as a network of objects and as a network of functions.

QUESTION: What is the best representation of program components?

ANSWER: A modified version of the visual formalism used for the library.
NOTE: The fully qualified names of the memory values are ABCD.ID and ABCD.BALANCE

Message Fusion

An output message from one operation may be linked with an input message from another operation, if the classes within the message are compatible.

The names of the objects within the message need not match.
EXAMPLE:

instantiate: object name = ABCD, ID = ABCD, BALANCE = 10000

NOTE: The fully qualified names of the memory values are ABCD.ID and ABCD.BALANCE

Message Fusion

An output message from one operation may be linked with an input message from another operation, if the classes within the message are compatible.

The names of the objects within the message need not match.
The Object View of a Program

A program, as instantiated from a library, is a network of interacting objects.

The Base Network View of a Program

A program can also be viewed as a network of interacting operations by:

- removing class/object boundaries
- replicating object operations and memories
- distributing constructor and destructor operations over the components of objects
The Object View of a Program

A program, as instantiated from a library, is a network of interacting objects.

The Base Network View of a Program

A program can also be viewed as a network of interacting operations by:

- removing class/object boundaries
- replicating object operations and memories
- distributing constructor and destructor operations over the components of objects
The Response View of a Program

A response (functional) view of a program can be created by superimposing, on the base network, boundaries corresponding to the responses to input messages to the program.
The Response View of a Program

A response (functional) view of a program can be created by superimposing, on the base network, boundaries corresponding to the responses to input messages to the program.
Visualizing Program Execution

A program is executed by running a scenario, that is, by specifying a sequence of input messages to the program.

The program display is animated, showing:

- active operations
- messages being sent
- message and memory values

EXAMPLE
Visualizing Program Execution

A program is executed by running a scenario, that is, by specifying a sequence of input messages to the program.

The program display is animated, showing:

- active operations
- messages being sent
- message and memory values

EXAMPLE
Restricting Concurrency

When a program is instantiated from a library, it has unrestricted concurrency.

The concurrency of a program may be restricted by transforming operations and objects.

Non-concurrent components of programs are represented by rectangles instead of parallelograms.

EXAMPLE:

NOTE: after transformation, ACCEPT operation sequences messages, only one copy of each operation, and only one operation active at one time.
Restricting Concurrency

When a program is instantiated from a library, it has unrestricted concurrency.

The concurrency of a program may be restricted by transforming operations and objects.

Non-concurrent components of programs are represented by rectangles instead of parallelograms.

EXAMPLE:

NOTE: after transformation, ACCEPT operation sequences messages, only one copy of each operation, and only one operation active at one time.
Concurrency Restriction and Design

Concurrency restriction modifies a program to limit its use of resources.

Such programs can be used to generate code for realistic target environments.
Concurrency Restriction and Design

Concurrency restriction modifies a program to limit its use of resources.

Such programs can be used to generate code for realistic target environments.

**EXAMPLE:**

The diagram on the page illustrates a statechart for a controller. The statechart is labeled 'IDLE' and transitions to 'HEATING' and 'APPLYING PRESS' under specific conditions. The control operation diagram shows the actions associated with the statechart.
Visibility Assertions

A visibility assertion places a constraint on a pair of classes by requiring that objects of one class reference objects of the other.

EXAMPLE:

```
<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>&lt; [1:1]</th>
<th>ACCOUNT</th>
</tr>
</thead>
</table>
```

- each account object must be referenced to exactly one customer object
- each customer object must be referenced to zero or more account objects

Consistency of Class Structures with Assertions

If an assertion is placed on a pair of classes, either

- the operations and object memories of the classes must be consistent with the assertion, or
- an additional class must be created to satisfy the assertion

The assertion on the previous page can be satisfied by:

- placing a list of account object names in the memory of each customer object, or
- placing a customer object name in the memory of each account object, or
- creating a new class whose objects contain customer-to-account cross references
Visibility Assertions

A visibility assertion places a constraint on a pair of classes by requiring that objects of one class reference objects of the other,

EXAMPLE:

- Each account object must be referenced to exactly one customer object.
- Each customer object must be referenced to zero or more account objects.

Consistency of Class Structures with Assertions

If an assertion is placed on a pair of classes, either

- the operations and object memories of the classes must be consistent with the assertion, or
- an additional class must be created to satisfy the assertion

The assertion on the previous page can be satisfied by:

- placing a list of account object names in the memory of each customer object, or
- placing a customer object name in the memory of each account object, or
- creating a new class whose objects contain customer-to-account cross references
Naming Visibility Assertions

A visibility assertion can be given names to describe the basis for the assertion.

EXAMPLE:

3.3 Program Organization

QUESTION: What is the best organization of program components?

ANSWER: There is no "best" organization. It is useful to view a program both as a network of objects and as a network of functions.

QUESTION: What is the best representation of program components?

ANSWER: A modified version of the visual formalism used for the library.
Naming Visibility Assertions

A visibility assertion can be given names to describe the basis for the assertion.

EXAMPLE:

3.3 Program Organization

QUESTION: What is the best organization of program components?

ANSWER: There is no "best" organization. It is useful to view a program both as a network of objects and as a network of functions.

QUESTION: What is the best representation of program components?

ANSWER: A modified version of the visual formalism used for the library.
Instantiation of Objects

Instantiating a class definition means creating zero or more objects of that class within the program.

If at least one object of a class is to be created:

- the values in its object memory must be initialized
- its name must be specified if not included in the library definition

EXAMPLE:

instantiation; object name =ABCD, ID = ABCD, BALANCE = 10000

NOTE: The fully qualified names of the memory values are ABCD.ID and ABCD.BALANCE
Instantiation of Objects

Instantiating a class definition means creating zero or more objects of that class within the program.

If at least one object of a class is to be created:

- the values in its object memory must be initialized
- its name must be specified if not included in the library definition

EXAMPLE:

```
instantiation; object name = ABCD,
ID = ABCD, BALANCE = 10000
```

NOTE: The fully qualified names of the memory values are ABCD.ID and ABCD.BALANCE
Viewing Multiple Objects

If multiple objects of a class are instantiated, they may be viewed by using a selector to rotate through object "windows."

Instantiation of Operations and Memory

When a class definition is instantiated, the default is to instantiate all class and object operations.

If desired, only a subset of these operations may be instantiated.

The object memory is always instantiated, but values not referenced by any of the instantiated operations are omitted.
Viewing Multiple Objects

If multiple objects of a class are instantiated, they may be viewed by using a selector to rotate through object "windows."

Instantiation of Operations and Memory

When a class definition is instantiated, the default is to instantiate all class and object operations.

If desired, only a subset of these operations may be instantiated.

The object memory is always instantiated, but values not referenced by any of the instantiated operations are omitted.
Instantiation Rules

- Instantiation of a subclass causes the "view subclass" transformation to be applied.
- Instantiation of a composite object also instantiates the component objects.
- Instantiation of a class that initiates a message exchange with another class instantiates the other class if necessary.

EXAMPLE OF SUBCLASS INSTANTIATION:

NOTE: Instantiating a superclass does not instantiate any details of its subclasses. Superclass/subclass distinctions are not made in programs.
Instantiation Rules

- Instantiation of a subclass causes the "view subclass" transformation to be applied.
- Instantiation of a composite object also instantiates the component objects.
- Instantiation of a class that initiates a message exchange with another class instantiates the other class if necessary.

EXAMPLE OF SUBCLASS INSTANTIATION:

```
:PROFESSOR
  :NAME: STRING
  :PHONE: INTEGER
  :RESIDENT_PROF
    :NAME: STRING
    :PHONE: INTEGER
    :SALARY: REAL
  :VISITING_PROF
    :NAME: STRING
    :PHONE: INTEGER
    :SALARY: REAL

:RESIDENT_PROF
```

NOTE: Instantiating a superclass does not instantiate any details of its subclasses. Superclass/subclass distinctions are not made in programs.
Message Exchanges and Instantiation

When a class that initiates a message exchange is instantiated:

- If the class containing the "used" operation is not in the program, the class is added with the "used" operation only.
- If the class containing the "used" operation is in the program but the "used" operation is not, the operation is added.

**EXAMPLE:**

Instantiating ACCOUNT will instantiate CUSTOMER.VERIFY if needed
Message Exchanges and Instantiation

When a class that initiates a message exchange is instantiated:

- If the class containing the "used" operation is not in the program, the class is added with the "used" operation only.

- If the class containing the "used" operation is in the program but the "used" operation is not, the operation is added.

EXAMPLE:

Instantiating ACCOUNT will instantiate CUSTOMER.VERIFY if needed
Message Fusion

An output message from one operation may be linked with an input message from another operation, if the classes within the message are compatible.

Fusion Rules

- If the messages to be fused contain multiple objects of the same class, and the object names do not match, the ambiguity must be resolved at fusion time.

**EXAMPLE:**

```
X:REAL  ?  A:REAL
----->
Y:REAL  ?  B:REAL
```

- An outgoing message must have a class no more abstract than the incoming message with which it is fused.

**EXAMPLE:**

```
POS_INT  INT  INT  POS_INT
----->  ---->  ---->  ----->
yes     no
```
Message Fusion

An output message from one operation may be linked with an input message from another operation, if the classes within the message are compatible.

The names of the objects within the message need not match.

Fusion Rules

- If the messages to be fused contain multiple objects of the same class, and the object names do not match, the ambiguity must be resolved at fusion time.

**EXAMPLE:**

\[
\begin{align*}
X &: \text{REAL} & A &: \text{REAL} \\
\rightarrow & \quad \rightarrow & \\
Y &: \text{REAL} & B &: \text{REAL}
\end{align*}
\]

- An outgoing message must have a class no more abstract than the incoming message with which it is fused.

**EXAMPLE:**

\[
\begin{align*}
\text{POS\_INT} & \rightarrow \quad \text{INT} \rightarrow \quad \text{INT} \rightarrow \quad \text{POS\_INT} \\
\text{yes} & \quad \text{no}
\end{align*}
\]
The Object View of a Program

A program, as instantiated from a library, is a network of interacting objects.

The Base Network View of a Program

A program can also be viewed as a network of interacting operations by:

- removing class/object boundaries
- replicating object operations and memories
- distributing constructor and destructor operations over the components of objects
The Object View of a Program

A program, as instantiated from a library, is a network of interacting objects.

The Base Network View of a Program

A program can also be viewed as a network of interacting operations by:

- removing class/object boundaries
- replicating object operations and memories
- distributing constructor and destructor operations over the components of objects
The Response View of a Program

A response (functional) view of a program can be created by superimposing, on the base network, boundaries corresponding to the responses to input messages to the program.
The Response View of a Program

A response (functional) view of a program can be created by superimposing, on the base network, boundaries corresponding to the responses to input messages to the program.
Visualizing Program Execution

A program is executed by running a scenario, that is, by specifying a sequence of input messages to the program.

The program display is animated, showing:

- active operations
- messages being sent
- message and memory values
EXAMPLE
Restricting Concurrency

When a program is instantiated from a library, it has unrestricted concurrency.

The concurrency of a program may be restricted by transforming operations and objects.

Non-concurrent components of programs are represented by rectangles instead of parallelograms.
EXAMPLE:

NOTE: after transformation, ACCEPT operation sequences messages, only one copy of each operation, and only one operation active at one time.