CS/*3.0*

The COOL:Gen Component Standard Version 3.0

A Standard for Specifying & Delivering Software Components built with COOL:Gen

30-SEP-1999

STERLING SOFTWARE

The COOL:Gen Component Standard

Version 3.0

September 1999
Part Number 2616394-0005

Changes are periodically made to the information herein. These changes will be incorporated in new editions of this publication.

Trademarks

COOL:Gen is a trademark of Sterling Software, Incorporated.

IDL is a trademark of Object Management Group.

Java is a trademark of Sun Microsystems, Inc.

Microsoft, ActiveX, Object Linking and Embedding, OLE, and OLE/COM are trademarks of Microsoft Corporation.

Other trademarks are the property of their respective owners. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this document, and the editor was aware of a trademark claim, the designations have been printed in initial capital letters and/or acknowledged on this page.

Acknowledgments

This Standard includes concepts originated by Castek Software Factory, Inc. We acknowledge and thank them for their contributions in developing this standard.

This standard includes concepts originated by ICON Computing, Inc. We acknowledge and thank them for their contributions in developing this standard.

Contents

1 Introduction

Overview

A component is a software building block used to build applications or other larger components.

Sterling Software recommends that any COOL:Gen-built components traded between companies follow the Standard defined in this publication. Sterling Software also advises organizations building components with COOL:Gen for their own use to adopt this Standard.

The Standard encompasses good component-building practice. Adhering to this Standard will provide future opportunities to sell or exchange internally developed components. COOL:Gen-built components acquired from external sources are likely to follow this Standard.

By conforming to the rules and terms in this Standard, organizations will find it easier to exchange know-how with other like-minded companies. It also enables Sterling to offer tools that convert components from an old to new version of the Standard.

Organizations have been building software components with COOL:Gen since 1995, and a considerable body of expertise has been established within the COOL:Gen community. Components are available for lease or purchase, or as a part of consulting projects, from a number of Sterling Software partner companies.

Components that conform to this Standard are known as a **CS/*3.0*** components. Components built to earlier versions of this standard continue to be called **CBD*96*** components.

NOTE: Early drafts of **CS/*3.0*** were code-named CBD3, and this term may occasionally occur within the older software documentation.

The Version 3.0 Standard requires COOL:Gen Release 5.1 or later, and takes advantage of new features within this release of COOL:Gen.

The scope of this document is:

- Characteristics of **CS/*3.0*** components.
- Component Delivery Standards.

- Component Specification Standards.
- Recommended disciplines for upgrading components, naming component elements and the component implementation.
- A glossary of **CS/*3.0*** terminology.

Component-based development does not require all components to conform to **CS/*3.0***. You can build applications using a mixture of **CS/*3.0*** components and components conforming to other standards, for example, Microsoft's ActiveX™ controls. In this publication, the term *component* always refers to components conforming to **CS/*3.0***.

What's New

The main differences between **CS/*3.0*** and **CBD*96*** version 2.1 are:

- Three new COOL:Gen 5.1 object types – component specification type, interface type and specification type – are used, so model semantics no longer rely on naming conventions.
- Enhanced component modeling diagrammers are used, so type models are no longer drawn with COOL:Gen's Data Modeling tools. In particular, an Interface Type Model diagram is supported, so subject areas are no longer used to scope the vocabulary of each interface.
- Naming conventions for component modeling objects are no longer mandatory, although the previous conventions continue to be recommended.
- Component upgrading procedures (involving version and revision concepts) are recommended rather than mandatory.
- There is no requirement to deliver a component specification model with a white-box component, since the new Component Manager tool readily enables component specification extraction.
- **CS/*3.0*** does not require the operations of an interface to be "factored" to specification types within the interface type model.
- The content of the return/reason code list is more flexible.
- Component documentation standards are more flexible.
- Dynamically linked sub-transactional operations are permitted; dummy procedure steps are not always needed.
- **CS/*3.0*** covers the nature and use of *component objects*.
- **CS/*3.0*** is better aligned with COOL:Spex.

- Components built to a standard prior to **CS/*3.0*** may be kept in that standard; however, a conversion wizard is provided with Gen 5.1, which enables organizations to convert component specifications to the new standard.

Aim of This Publication

This publication describes a Standard. It is not a component-based development primer. It does not attempt to teach component concepts, explain the benefits of components, or describe the development process used to build components. Other documents and courseware exist for these purposes.

This publication focuses on rules for component specification and delivery. It avoids making rules that constrain component implementation options.

To understand this Standard, readers need to be familiar with CBD concepts and COOL:Gen.

About This Publication

Organization

This publication is organized into the chapters and appendices shown in the table below. A Glossary of Terms, and an Index, are provided at the end of the publication.

Document Organization

Chapter/ Appendix	Title	Description
1	Introduction	Provides an overview of the intent and organization of this Standard
2	Component Characteristics	Defines the basic qualities that all CS/3.0 components should exhibit
3	Component Delivery	Details the delivery requirements for CS/3.0 standard components
4	Component Specification Model	Defines the standards that apply to the component specification model
5	Interface	Describes the standards for the interfaces of components
6	Component Implementation Model	Defines the standards that apply to the component implementation model
7	Component Executable	Defines the standards that apply to the component executable
8	Component Documentation	Defines the documentation that must be delivered with every component.

Chapter/ Appendix	Title	Description
9	Associated Model	A further model that can help the customer make better use of a delivered component.
10	Identifiers and Relationships	The four standardized identifiers used in CS/3.0, one of which is used to maintain cross-component relationships
A	Component Upgrades	How to handle new component releases
B	Recommended Naming Conventions	Provides a set of naming conventions for objects in component models
C	Standard Public Operation Parameters	Describes the standard and recommended public operation parameters
D	Return Codes	Lists standard return codes and their descriptions

References to the Advanced Practices

⊕ This symbol and typeface are reserved for comments about alternative or more difficult practices, which may be important to some organizations, but which are not considered to be the regular practice. The use of an advanced practice is not a violation of the Standard.

2 Component Characteristics

This chapter describes the basic qualities of **CS/*3.0*** components.

In the chapters that follow, we define various rules that help realize these qualities. Adhering to the rules does not guarantee the qualities are achieved: component developers still need to understand the qualities, and design their components accordingly.

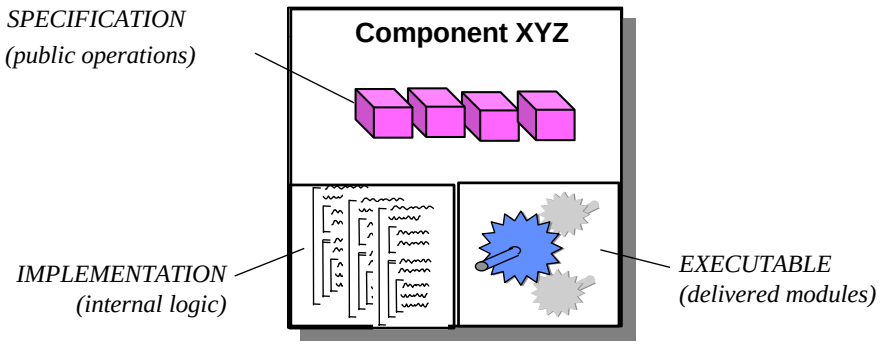
Characteristic 1

A component is a software building block, which can be used to construct applications or larger-grained components, that is made available as an independently delivered software package.

Characteristic 2

A component has three facets:

- Component specification – the definition of component behavior.
- Component implementation – the internal design and code that realizes the specification.
- Component executable – a set of modules that can be executed to provide the specified functionality.



The Three Facets of a Component

Characteristic 3

A component is encapsulated. That is, its specification and implementation are distinct. The implementation can be changed without impacting the software that consumes the component, since the consuming software may only reference the specification.

NOTE: Encapsulation is sometimes broken in order to achieve acceptable performance, or to maintain data integrity, or to wrap legacy software as multiple components. A formal procedure for proposing, examining and approving violations should be established in organizations developing CS/*3.0* components.

Characteristic 4

A component is delivered as a white-box component (specification and implementation facets) or a black-box component (specification and executable facets).

Characteristic 5

The functionality of a component is made available through one or more programmable interfaces.

- An interface is a group of related operations.
- Each operation is a separately callable unit of functionality.
- Interfaces are independent units of definition, unless otherwise specified.
- A component specification is a list of interfaces, plus any additional rules defined on the component specification type itself.
- The same interface may be offered by several components.

NOTE: A COOL:Gen application or component is unable to consume two components that offer the same interface. Interface reuse is mainly confined to new releases of the same component.

Characteristic 6

Each operation of a component has the following features. It:

- Has an operation specification, which defines its behavior and how that behavior must be invoked.
- Belongs to an interface.
- Is a success unit (does not leave component constraints violated).
- May be a transaction or a sub-transaction (a commit unit or not).
- May include end-user interaction, or not.

NOTE: In practice, most operations are designed to run on server processors, and do not embed end-user interactions. User interface designs are normally tailored to the user's circumstances and hardware, and are liable to change. Nevertheless, CS/*3.0* embodies the notion of the user interface-bearing operation, built with a COOL:Gen display step, which incorporates end-user interactions.

Characteristic 7

A component is replaceable by another component that supports at least the same interfaces.

Characteristic 8

A component may have a dependency upon other components. This usually means it invokes operations of those components. The dependency may be:

- A specification dependency, which then applies to every implementation of the component.
- An implementation dependency, which reflects a internal design choice that may not exist in alternative implementations.

Characteristic 9

A component implementation may use a data store to makes its data persistent.

Characteristic 10

A component's data store may not be directly manipulated by any other component, since that would break encapsulation. However, a new release of a component could directly manipulate the data store of a previous release.

3 Component Delivery

Overview

This chapter states what the component provisioner must supply to a customer. White and black-box components are explained, and the various parts of component delivery are introduced.

Delivery Styles

There are two component delivery styles:

- White-box.
- Black-box.

A white-box component provides the customer with the source code (action diagram statements), enabling the customer to examine the internal design, generate it to run on a variety of platforms, and modify the model where necessary.

A black-box component provides the customer with executable modules, but no source code. The customer is unable to change the internal design.

White-box component

A white-box component delivery must include the following:

- COOL:Gen Component Implementation Model.
- Component Documentation.
- Optionally, an Associated Model.

Black-box component

A black-box component delivery must include the following:

- COOL:Gen Component Specification Model.
- Modules of the Component Executable.
- Component Documentation.
- Optionally, an Associated Model.

These delivery parts are illustrated in Figure 3.1, Component Delivery.

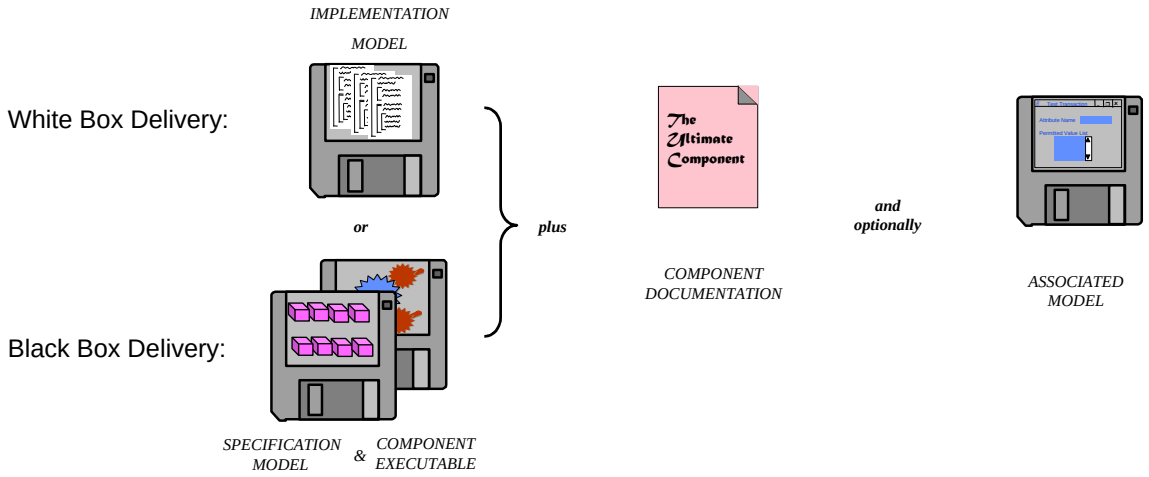


Figure 3.1 Component Delivery

NOTE: A component provisioner can elect to deliver white-box components with the executable for requested platforms, as a further convenience to the customer.

NOTE: Specification-only components can be useful. For example, provisioners can use them to allow potential customers to inspect a component before agreeing to take delivery. They are needed when outsourcing implementation development. However, this publication focuses on black-box and white-box components.

NOTE: In the CBD*96* standard, a specification model had to be delivered with each white-box component, so that the customer was not put to the inconvenience of extracting the component specification. CS/*3.0* does not require a separate specification model to be delivered, since the new Component Manager tool can locate the specification within a component implementation model, and copy it to a consuming model.

Model Formats

A specification, implementation and associated model may be delivered in any convenient format. For example:

- Four .DAT files.
- CHECKOUT.TRN file, which is more compact than the four .DAT files.
- UPDATE transaction file, especially in cases where the model is too large to be downloaded to a work station.

⊕ Delivering several components within one COOL:Gen model is less straightforward. This is regarded as an "advanced practice."

Component Specification

A component specification is a definition of the component's behavior, which do not define the manner in which the component is implemented. Primarily, a component specification is a statement of the component's function, but it may also include constraints on the component implementation and/or executable.

The component specification acts as:

- An instruction to the component implementor.
- A contract of guaranteed behavior for the component consumer.

The component specification consists of:

- The component specification type, which indicates the component name and the interfaces it offers.
- The interface specifications, each of which defines the interface name and any constraints (known as invariants), plus the specification of each operation.
- In addition, the component specification type may define:
  - Further invariants which the component must apply, not included in the interface specifications
  - Constraints that apply to every implementation of this component
  - Constraints that apply to every executable for this component.

A component provisioner must always provide a component specification when delivering a component to a customer. The specification is delivered within the component specification model (for black-box components) or within the component implementation model (for white-box components). Any aspects of the specification not included in the component specification model, must be included in the external component documentation.

Chapter 4, "Component Specification Model," and Chapter 5, "Interface," provide the detailed standards for component specifications.

Component Implementation

This is the component's internal design and logic, which achieves the effect stated in the component's specification.

A COOL:Gen component implementation model contains the component's internal design in the form of action diagrams and, possibly, a database design. The customer can generate the component executable from this model.

The component implementation model also includes the component specification, and is only delivered for a white-box component.

If external action blocks have been used in the implementation, then their code also forms part of the component implementation. This code must be delivered in separate files along with the implementation model. If this external code references persistent data stores, then their design also forms part of the component implementation.

Refer to Chapter 6, “Component Implementation Model,” for more details.

Component Executable

The component executable is a collection of modules that, when executed on the stated platform, collectively produce the behavior defined in the component specification. The modules may be load modules (executable files) and/or object modules and/or operations libraries (modules containing operations linked in at run time, such as DLLs.)

Many components require their own data store (data base or files) to make their data persistent. For these components, the component executable must also include modules which enable the appropriate data store to be installed. Typically, this is a module containing the data definition language statements that define a relational database; plus the bind modules which are needed by database products that support static database binding.

Refer to Chapter 7, “Component Executable,” for more details.

Component Documentation

A component must be delivered with a certain amount of external documentation, which summarizes some basic facts about the component, and also covers anything not explained in the delivered COOL:Gen model.

The external documentation may duplicate information already contained in the COOL:Gen model, if the provisioner considers this helpful to the customer. See Chapter 8, “Component Documentation.”

A component may also be delivered with test cases and test results that the customer can reuse. This is also explained in Chapter 8, “Component Documentation.”

Associated Model

Additional action diagrams may be included in the component delivery to help the customer test and use the component. These additional action diagrams are supplied in a third model, which we call the associated model. Further details appear in Chapter 9, “Associated Model.”

4 Component Specification Model

Overview

A black-box component delivery must include a Component *Specification* Model. This chapter defines the rules that apply to the component specification model.

A white-box component delivery must include a Component *Implementation* Model. The implementation model contains the component specification, and the rules in this chapter also apply to the specification part of the component implementation model.

The specification model contains:

- Specification subject areas.
- The component specification type.
- The interfaces offered by the component.
- The execution parameters for the component (transaction codes and object module names for operations).

NOTE: The Component Manager tool makes it easy to copy the component specification, or an interface, or an operation specification, into an implementation model that consumes this component (or a selected interface or operation). Alternatively, COOL:Gen's migration facility can be used.

Specification Subject Areas

A component specification model must contain at least one subject area, with the role property set to "specification." This is directly contained in the root subject area.

A component specification model may contain further specification subject areas.

All specification elements, that is, the component specification type, the interface types and specification types, must be contained in one of the following:

- A specification subject area directly contained in the root subject area.
- A specification subject area contained within another specification subject area, which is not directly or indirectly within an implementation or general subject area (except for the general root subject area).

If these properties are not met, Component Manager will not transfer the entire component specification to a consuming model.

The recommended practice is one specification subject area, placed directly under the root subject area.

Component Specification Type

A component specification model must contain one component specification type, which represents the component specification.

NOTE: The component specification type is a new modeling element, first introduced in COOL:Gen 5.1. It can only be added to the model using the Specification Model diagrammer.

NOTE: If a model contains multiple component specifications, then it must contain multiple component specification types. CS/*3.0* advocates a separate COOL:Gen model for each component specification.

- The component specification type must be contained in a specification subject area.
- The name of the component specification type is the name of the component specification. The name should include the version and revision number, if **CS/*3.0***'s recommended upgrading approach is adopted (see Appendix A, "Component Upgrades").
- The component specification type must be associated with one or more interface types, representing the interfaces that it offers.
- Any invariants that apply to the component, over and above those defined on the interfaces, are detailed in the description panel.
- Any constraints, which apply to any implementation of this component specification, are detailed in the description panel.
- Any constraints, which apply to any executable of this component specification, are detailed in the description panel.
- If there is insufficient space in the description panel, these constraints must be described in the external documentation for the component.

Interfaces

A component specification model must contain at least one interface type. The standards for documenting interfaces are given in Chapter 5, "Interface."

Business Systems

COOL:Gen requires that all operations belong to a business system.

There are no mandatory rules concerning the use of business systems.

We recommend that all operations, both transactional operations represented by procedure steps and sub-transactional operations represented by BSD action blocks, are placed in the "default" business system which is created in every new model.

Appendix B, "Recommended Naming Conventions," provides the recommended naming convention.

Execution Parameters

A component specification model contains the execution parameters for the component.

NOTE: Execution parameters need to be in the specification model because they have to be transferred into any consumer model. That is, into any component implementation or application model that wants to use an operation of this component. Execution parameters are not normally regarded as a part of a component specification, although they could be defined as execution constraints if they were required to be the same for every implementation and executable of this component.

The execution parameters are:

- Source name for each sub-transactional operation.
  This is used by COOL:Gen to generate the object module name.
- Transaction code for each transactional operation.
- Load module name for each transactional operation.
  The component implementor usually decides the execution parameters.

5 Interface

Overview

This chapter defines the standard for interfaces. An interface is a collection of semantically related public operations.

- The operations are *related* in the sense that:
  - They share concepts.
    All the types referenced by the operations of one interface have a consistent meaning.
  - They are normally inter-dependent and used in conjunction with one another.
    For example, Operation Q of the interface depends upon another, Operation R, of the interface having been used beforehand.
- The operations are described as *public*, to distinguish them from operations used within a component implementation, which are not available to the component consumer.

The functionality of a component is only available through the operations of its interfaces. A component must support one or several interfaces. A component specification is primarily a list of its interfaces.

Figure 5.1 depicts a component offering two interfaces and introduces the "lollipop" icon commonly used to denote an interface.

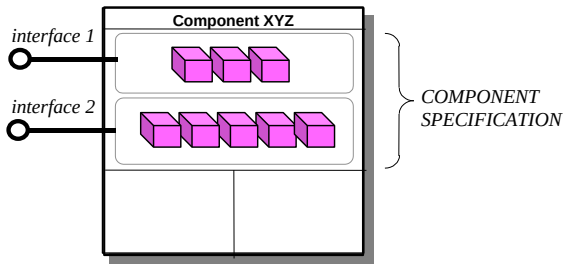


Figure 5.1 Component With Two Interfaces

An interface is a reusable unit of specification. The same interface may be offered by several components.

However, COOL:Gen does not enable a consuming component or application to use two components that offer the same interface. So we advise against reusing interfaces, except in these circumstances:

- For different releases of the same component

- Where two interfaces, although identical, are given different names
- For an industry-agreed component specification that is supported by competitive offerings of the same specification
- Where it is clear that no consumer will need to use both of the components that have a common interface.

Interface definitions must appear in both component specification and implementation models.

NOTE: The term *interface* is used here in the same sense as in Microsoft's COM™ Component Object Model, the Object Management Group's IDL™ and the Java ™ programming language.

Interface Definition

An interface definition has two main facets:

- Specifications for each of the Public Operation offered by the interface.
- An Interface Type Model, consisting of specification types, attributes, relationships, and invariants, which defines the information that the interface can retrieve.

Figure 5.2 depicts a component with two interfaces, and shows that each interface has its own type model and own collection of operations.

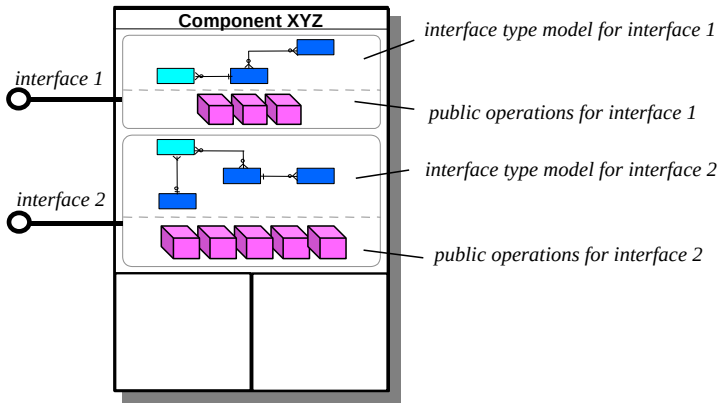


Figure 5.2 Two Facets of an Interface

An interface is documented using COOL:Gen's Interface Type Model tool. It consists of:

- One interface type.
- One or more public operations.

- One interface type model diagram.
- Specification types referenced by the interface. That is:
  - The types used in operation imports and exports.
  - The types appearing within the interface type model.

Specification types can be referenced by more than one interface.

Figure 5.3 shows an example of an Interface Type Model. This model defines the data that the interface is able to recall, by storing it, by obtaining it from other interfaces, or by derivation. This is the information that the interface "remembers."

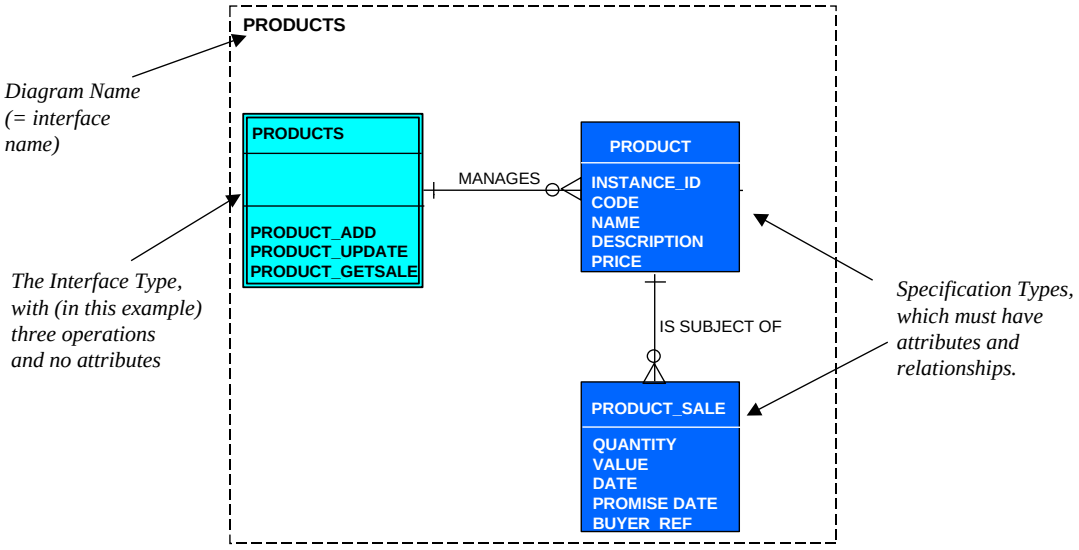


Figure 5.3 Interface Type Model Example

All the parts of the interface definition must be recorded within specification subject areas. That is, within subject areas where the subject area role property set to “specification.”

These specification subject areas must not be directly or indirectly contained in implementation subject areas, since this indicates a consumed interface rather than an interface being defined.

We recommend that component models contain just one specification subject area, which contains the component specification type and all its interfaces. However, multiple specification subject areas may be used if preferred.

NOTE: In a component implementation model in which the implementation consumes further components, then at least one more specification subject area is required. This is because the specifications of consumed components must be placed in specification subject areas which are contained in an implementation subject area.

Summary

This summary defines how each part of an interface definition is recorded in COOL:Gen. The standards for each part are given in subsequent sections of this chapter.

- Interface Type.
  Use the Specification Model diagramming tool to add an interface type to a component model. This automatically creates an Interface Type Model for the interface, containing just the interface type.
- Operations.
  Add one or more operations to the interface types, to define the public operations.

  Operations may be sub-transactional or transactional.
  - Sub-transactional operations must be defined as BSD Action Blocks while adding the operation. You do not need to create a free standing BSD action block prior to registering it as an operation; the BSD Action block is created as you create the operation.
  - Transactional operations must be represented by Procedure Steps. A procedure step must be added to a business system, before it can be registered as an operation in the Specification Model diagram.

NOTE: We recommend that component models contain a single business system, which is automatically added when a new model is created. However, further business systems are permitted by this Standard.

- Interface Type Model.
  Construct the interface type model using the Interface Type Model diagramming tool.

  There is one, and only one, Interface Type Model diagram for each interface, which gets created when an interface is added to a COOL:Gen model.

  The Interface Type Model is constructed from the interface type itself, and from specification types.
- Specification Type.
  These can be added to the COOL:Gen model using either the Interface Type Model diagramming tool or the Specification Model diagramming tool. Whichever tool you use, they are owned by (or "are contained in") a specification subject area and are referenced by (or "are included in") one or more Interface Type Models.

Interface Type

Use the Specification Model diagramming tool to add an interface type to a component model. This automatically creates an Interface Type Model for the interface, which initially contains just the interface type.

The interface must be placed in a subject area that has role set to "specification."

The interface type must have an "offered by" relationship with a component specification type. (At least, this must be the case in a COOL:Gen component implementation model or COOL:Gen component specification model).

The interface must own at least one operation. See section Public Operation Specifications (on page 23) for the details.

The description panel for the interface must contain:

- Revision number of the interface.
  Revision numbering begins at zero.
- Revision date.
- Purpose of the interface.
- Invariants applied by the interface, except where these are already defined by properties within the interface type model.

  For example, the interface type model may express these special kinds of invariant:

  - Cardinality and optionality of relationships.

- Optionality of attributes.
- Permitted values for attributes.
- Exclusive relationship memberships.
- Classifying values for subtypes.

Other invariants have to be expressed in free text, using natural or formal languages. For example:

- Delivery Date is later than Order Date.
- Delivery Date is null unless Order Status is "delivered."

These textual invariants are written in the interface's description panel, even if they already appear on specification types. This makes them easy to locate implementers and consumers, and also makes it quite clear that they must be applied by this interface.

The interface type may own attributes, but does not need to.

(Multi-valued attributes of the interface are represented by the attributes of specification types within the interface type model. See the Interface Type Model section on page 37.)

The interface type normally has its type properties set as follows:

- Single occurrence.
- Not a business object type.
- Transient.
- No identifiers.

⊕ ADVANCED PRACTICE. Interfaces can be defined so they support multiple interface instances, and hence components that have multiple run-time instances, which we call *component objects* (see Chapter 10, “Identifiers and Relationships”). Where this is required:
- The interface is not "single occurrence", and will need an identifier. (In CS/*3.0* we call this identifier the component object identifier, or COID, and is one of the Standard Parameters listed in Appendix C, “Standard Parameters”).
- All operations of the interface need to import the component object identifier. (Operations must only run against one instance of the interface and component).

⊕ ADVANCED PRACTICE. Where an interface instance manages only one business object (instance), then the interface can be registered as a business object type. In COOL:Gen-built business components, it is more usual for specification types to correspond to business object types.

⊕ ADVANCED PRACTICE. Interface types which have an identifier, and perhaps other attributes, can be defined as persistent in component implementation models, enabling create/update/delete/read actions to be performed directly on the interface instance within the implementation. The fact that an interface is defined as persistent is not relevant to the consumer, and should not (ideally) be apparent in consumed specifications.

Public Operation Specifications

Use the Interface Type Model or Specification Model diagramming tool to add one or more operations to the interface type.

Operations must be *sub-transactional* or *transactional* or *user interface-bearing*.

- Sub-transactional operations are formed from BSD Action Blocks. Create the operation using the Add Operation command within the Specification Model diagramming tool. The operation must be defined as a BSD Action Block (not a BAA action block or Elementary Process).

NOTES:
Sub-transactional operations should generally be defined as using high-performance view matching. It is crucial that the operation in the component implementation model and the corresponding stub in the consuming model both have the same high-performance view matching value.

If the COOL:Gen model contains multiple business systems, ensure that the business system into which this BSD action block is to be placed is currently open, otherwise it will be placed in some other business system, and it is not possible to move it.

- Transactional operations are formed from Procedure Steps. You must first add a single-step procedure to a business system, using the Windows Navigation diagram or Dialog Design tools. A transactional operation is a no-display step, also known as a Server/Procedure Step in the Windows Navigation diagram.

  Then, using the Specification Model diagramming tool, use the Add Operation command to register the procedure step as an operation of the interface.

- User Interface-Bearing operations are also permitted by **CS/*3.0***. These are formed from display Procedure Steps, and enable the operation to interact with an end-user.

  You must first add a single-step procedure to a business system, using the Windows Navigation diagram or Dialog Design tools. The user interface-bearing operation is a display step, also known as a Window in the Windows Navigation diagram.

  Then, using the Specification Model diagramming tool, use the Add Operation command to register the procedure step as an operation of the interface.

NOTE: User Interface-Bearing operations are more difficult to work with than non-display operations.

In order to achieve a successful link flow from a display step to a UI-bearing operation, the return exit state set, within the UI-bearing operation's implementation, must be migrated into the consuming model.

To address this, and to hide unnecessary import/export views from the consumer, some organizations define a no-display step which has a link flow to a display step. The no-display step represents a UI-bearing operation, since its implementation always invokes a display step. The no-display step stub is transferred to the consuming model, not the display step.

NOTE: A user interface-bearing operation should not display the attributes of the component's internal entity types. It should only display the attributes of the interface type model, or the operation's parameters, or system attributes (such as current date and user identification).

Technically, a user interface-bearing operation is also transactional.

NOTE: To consume (that is, use, call, or invoke) an operation from another COOL:Gen model, the action diagram that represents the operation must be transferred to that model, using the Component Manager tool, or the migration services of COOL:Gen.

The consuming model can then define USE statements to invoke the operation, or can define dialog flows to invoke a transactional operation.

In the consuming model, the action diagram should not contain any logic statements, only import/export views and documentation NOTES. An action diagram without any logic is often called a *stub*.

Each public operation must have specification that fully describes the operation's behavior. The user of an operation must not need to look at the implementation to understand its behavior. If a black-box component is delivered, then there is no implementation to inspect.

A public operation specification must include the following parts:

- Operation name.
- Description.
- Operation parameters (imports and exports).
- Pre- and post-conditions.
- Return codes.
- Release History (discretionary).

The rules for each part are described in the sub-sections that follow.

NOTE:
A *parameter* is an attribute appearing in the input or export views of an operation.

The set of parameters input and output by an operation is sometimes termed the *signature* of the operation.

Public Operation Name

Aim to make the name unique within your computing environment, otherwise name clashes can occur when consuming a component.

We recommend that you use a name that indicates which interface it belongs to, and what action it performs on a type.

Appendix B, "Recommended Naming Conventions," offers a recommended naming convention.

NOTE: COOL:Gen defaults the source code file name for an action diagram to the first eight characters of the action diagram name. This name is then also used for the object module name. Some organizations find it useful to make the first eight characters unique, and as meaningful as possible, so that they can easily determine which operation is represented by any particular source module or object module.

Public Operation Description

Use the description panel of the public operation's action diagram to record the following:

- Revision number of the operation.
  Revision numbers begin at zero.
- Release date of the revision.
- Cross-reference to another specification.
  Where a transactional operation is functionally identical to a sub-transactional operation of the same interface (except the transactional operation will commit database updates), the description panel may state *SAME AS: operation_name*. In which case, there is no need to describe any purpose, pre-conditions, post-conditions, or return codes.
- Statement of the purpose of the operation.
  This is expected to be one or two sentences. If necessary, the purpose text can be continued as NOTES within the action diagram itself. The first textual line of the Note should be *PURPOSE (CONTINUED):*.

Public Operation Parameters

The parameters are defined using import and export views. These may be views of specification types or work sets.

NOTE: We recommend that all parameters are views of specification types. Work set views are, however, permitted by the Standard.

Take special care when using IEF_SUPPLIED work sets in the parameters of public operations. This is because the IEF_SUPPLIED work set exists in all models. When transferring specifications from one model to another, the various tools do not perform the IEF_SUPPLIED transfer in a uniform and controllable manner.

Recommendations:
(a) IEF_SUPPLIED attributes are best avoided as parameters, especially if you are a company supplying components commercially
(b) Where IEF_SUPPLIED attributes are used as parameters, the organization makes it a rule that the IEF_SUPPLIED work set is left unmodified in all models.

Name type and group views to indicate whether they are used for input, output or both input-and-output.

Give repeating and non-repeating group views meaningful names.

Appendix B, "Recommended Naming Conventions," offers a recommended naming convention.

Only record that an import view is mandatory if it actually *is* mandatory (group or type or attribute view) within the imports. The setting of the mandatory/optional property within an import view is a part of the operation specification.

Procedure steps cannot support combined import/export views, so only use combined import/export views on sub-transactional operations.

NOTE: Avoid using combined import/export views on any sub-transactional operations that are also made available as transactional operations. Otherwise, the two corresponding operations cannot have identical specifications.

In a COOL:Gen component specification model, the public operation's action diagram is a *stub*, and must not include local or entity action views.

Each public operation must export the *standard parameters*. These parameters communicate *exception conditions* to the operation consumer.

The following attributes must be defined in every component model and must be exported by every public operation:

- SEVERITY_CODE
  A single-character text field indicating the severity of the exception. Permitted values are I, W, and E. These characters indicate Information, Warning and Error, respectively.
- ROLLBACK_INDICATOR
  A single-character text field that a public operation uses to request the consumer to roll back any updates that the public operation has made to the persistent storage.

  … OR …

  DATA_STORE_STATUS_CODE
  A single-character text field that indicates the status of persistent storage after an operation execution, where:
  - "1" = data unchanged
  - "2" = changes rolled back
  - "3" = data changed
  - "4" = data integrity compromised
- ORIGIN_SERVID
  A 15-digit field that is used to communicate which installed copy of the component raised the return, and the reason codes being exported in this export view.

NOTE: It is suggested that every installed copy of a component executable should bear a unique identifier known as its server identifier, or *SERVID*.

Where several copies of the same component exist on a network, each of these should have a different server identifier. This assists in debugging and is explained further in the Server Identifiers section of Chapter 10, "Identifiers and Relationships."

Organizations do not need to populate this parameter, if they do not find it of value.

- RETURN_CODE
  A five-digit numeric field used to return a standardized code indicating the type of failure or success encountered during the operation execution. Valid values are defined in Appendix D, "Return Codes."
- REASON_CODE
  A five-digit numeric field used to provide more explanation about why the failure or success notified by the return code has occurred. Reason codes must be greater than, or equal to, zero.

  Reason code values may be *specified* or *unspecified*.
  - Specified reason codes are listed in the component specification, and the consumer can "code to" such values.
  - Unspecified reason codes are not explicitly listed in the component specification. The component specification just states that "any" or "other" reason codes may be exported. Reason code values can vary by implementation, so component provisioners are able to provide exceptions that are specific to their implementation.

NOTE: We recommend that the standard parameters appear in the last export view of each public operation, and are all attributes of a single specification type.

Appendix C, "Standard Parameters," provides more information about standard parameters, including additional parameters and the recommended order of placements in views.

Public Operation NOTEs

Further specification details are included within the action diagram of the public operation, as *NOTEs*.

The Notes should be included in this order:

1. PURPOSE (CONTINUED)
   This Note is only required if the purpose text cannot fit into the action diagram description panel.
2. PRE-CONDITION
   There may be any number of Notes labeled pre-condition. The same Note must contain the corresponding post-condition, and may include further pre/post condition pairs.
3. RETURN/REASON CODES
   There must be a Note that summarizes all the exception codes that can be returned by the public operation.
4. RELEASE HISTORY
   This is an optional Note.

All of these Notes (except PURPOSE) are described in the sub-sections that follow. Component provisioners must begin each Note with the keyword specified. This will enable other software tools to extract information from operation specifications.

Pre- and Post-Conditions

Pre- and post-conditions are defined in pairs. A public operation specification may include one or more pre/post pairs.

- A *pre-condition* is a statement that must be true prior to operation execution in order for its corresponding post-condition statement to be true after execution.
- A *post-condition* is a statement that will be true after the operation has executed, as long as the corresponding pre-condition was true prior to execution.

A pre-condition may be a list of conditions connected by Boolean operators (ANDs and ORs).

The order in which the pre/post pairs are documented has no semantic significance.

If several pre-conditions are found true, then several post-conditions will be true after operation execution. Ensure that this cannot lead to ambiguous outcomes.

Pre-conditions and post-conditions are defined within NOTES of the public operation's action diagram.

The keyword *PRE-CONDITION:* must be used to introduce each pre-condition. The pre-condition expression itself must appear on the lines that immediately follow *PRE-CONDITION:*.

Pre-conditions are not defined for import attribute views that are marked as *mandatory* within the view, although the exception codes used to report their absence must be included in the RETURN/REASON CODES list.

The keyword *POST-CONDITION:* must be used to introduce each post-condition. This keyword must appear within the same *NOTE* as its corresponding pre-condition, and must appear on the line that immediately follows the last line of the corresponding pre-condition.

A post-condition is a list of actions. There is no need to connect them with AND operators. For example:

```
NOTE PRE-CONDITION:
     An Employee with the imported Employee Instance_Id already
     exists
     AND does not have status "deleted".
     POST-CONDITION:
     The Employee Name is changed to the imported value.
     Return Code = 1, Reason Code = 0.


  PRE-CONDITION:
     An Employee with the imported Employee Instance_Id already
     exists AND has status "deleted".
     POST-CONDITION:
     Return Code = -41, Reason Code = 1.
```

Figure 5.4 Example of Two Pre/Post Condition Pairs within one Action Diagram Note

A post-condition action may be conditional, that is, preceded by an IF clause.

Any number of pre/post pairs may be placed in one *NOTE*. Any number of NOTES may be used to fully specify the operation's behavior.

NOTE: Where an organization envisages transferring component specifications from COOL:Gen to COOL:Spex model files, then it is preferable to place each pre/post pair in a separate NOTE statement.

A pre/post pair may be given a name, in which case the name is appended to each keyword as follows:

- PRE_CONDITION OF name:
- POST_CONDITION OF name:

NOTES: The full collection of pre-conditions provided for an operation may be independent or overlapping. They need not be exhaustive. That is, operation executions that do not meet any pre-condition are permissible. However, the consuming model cannot interpret export data of such executions.

- It is clearer for the reader if each pre-/post-condition pair is in a separate Note. Where Notes are built using cut-and-paste from word processors, then multiple pre/post pairs within a single Note will be more practical for the provisioner.
- We recommend that you write references to specification types and attributes in upper/lower case.
- We recommend capitalizing Boolean operators and any other conventional keywords, and using different symbols for nested parentheses; we recommend the following sequence: **([{ }])**
- Starting each action within the post-condition on a separate line is clearer.

Return/Reason Code List

The return and reason codes that can be exported by the operation (in the standard parameters in an export view) may be documented in a Note in the operation's action diagram.

The first line of this Note should be *RETURN / REASON CODES:*

Each subsequent line of the note must contain:

- A return code value.
  Return codes are standardized and are listed in Appendix D, "Return Codes."

- A reason code value, or the keyword OTHER or ANY.
  ANY indicates that no specific reason codes have been documented, so various *unspecified* reason codes may be issued by the operation.

  OTHER is used when one or more specific reason codes have been documented, to indicate that additional, *unspecified* reason codes could also be issued by the operation.
- The meaning of the return/reason code combination.
  The meaning may be omitted if the phrase ANY or OTHER has been used in place of the reason code.

The developer who implements the component may introduce further return and reason codes for the operation, and these must be added to the operation specification.

The consuming software must not "code to" *unspecified* reason codes. Unspecified reason codes should simply be reported to the user or systems administrator.

If a return/reason code is included in the list, but it does not also appear in any post-condition, then this implicitly means that an operation execution which detects this condition (the condition implicit in the meaning documented for the return/reason code value) rolls back any updates asserted within the explicit pre/post pairs.

- Some organizations decide to include every possible return/reason code in the list for an operation, so a full summary of all outcomes is readily accessible.
- Some organizations prefer to omit from the list those return/reason code values which are already explained in pre/post pairs, thus reducing duplicated effort and potential inconsistencies.
- Some organizations prefer to omit the return/reason code list altogether, but they must then ensure that every possible return/reason code value is explained within a pre/post pair.

NOTES: A component may offer a standard operation that enables the operation consumer to obtain further information, or a formatted message, for any given return code/ reason code combination.

Operation implementations can use exit states *within* the implementation, to retain the current state of execution. Standard action blocks can be used to convert exit states into standard return and reason codes.

Whenever you USE an operation (which has been implemented in another COOL:Gen model), be aware that it may update the exit state variable within its implementation, and hence alter the current value of the exit state within the logic you are developing.

Exit states have not been used as the mechanism for communicating exceptions in CS/*3.0* because they are not readily accessible to non-COOL:Gen consumers.

Public Operation Specification Examples

Figure 5.5 is an example of a public operation specification:

```
Operation Name: IHRC1991_EMPLOYEE_CHANGENAME_S of IHRC1_INTERFACE

Action Block Description:
REVISION NUM: 3
RELEASE DATE: 22-Aug-1999
PURPOSE:    To enable the family-name of an Employee to be
corrected, or to change the family name of an employee (after
marriage, for instance).

IHRC1991_EMPLOYEE_CHANGENAME_S of IHRC1_INTERFACE
 IMPORTS:
    Type View IN IHRC1_EMPLOYEE (mandatory, transient, import only
       INSTANCE_ID (mandatory)
       NAME (mandatory)

 EXPORTS:
   Type View OUT IHRC1_STANDARD_PARAMETERS (transient, export only
       SEVERITY_CODE
       ROLLBACK_INDICATOR
       ORIGIN_SERVID
       RETURN_CODE
       REASON_CODE

 NOTE PRE-CONDITION:
    An Employee with the imported Employee Instance_Id already
    exists AND does not have status "deleted".
    POST-CONDITION:
    The Employee Name is changed to the imported value.
    Return Code = 1, Reason Code = 1

 NOTE PRE-CONDITION:
    An Employee with the imported Employee Instance_Id already
    exists AND has status "deleted".
    POST-CONDITION:
    Return Code = -41, Reason Code = 1

 NOTE PRE-CONDITION:
    No Employee with the imported Employee Instance_Id exists.
    POST-CONDITION:
    Return Code = -10, Reason Code = 1

 NOTE: RETURN / REASON CODES
    + 1/1          Update successful
    - 10/1         Employee ID not found
    - 20/1         Employee ID missing in imports
    - 20/2         Employee Name missing in imports
    - 41/1         Employee not updated since flagged as deleted
    - 41/OTHER     Employee update action failed
    - 60/ANY       Persistent storage failure
```

Figure 5.5 Public Operation Specification Example 1

NOTE: In the example above, pre-conditions have not been defined to check that Instance_Id and Name have been input, since these are marked as mandatory within the import view. However, the return and reason codes for missing mandatory attributes have been included in the Return/Reason Codes Note.

The next example (Figure 5.6) shows how the same specification might look if it had been prepared in COOL:Spex, and transferred to COOL:Gen using the Component Manager tool. Observe that a more formal pre/post syntax has been used, that the pre/post pairs are named, and the different style of view name. The COOL:Gen developer has added the return/reason code list.

```
Operation Name:IHRC1991_EMPLOYEE_CHANGENAME_S of IHRC1_EMPLOYEE_MGR

 Action Block Description:
 REVISION NUM: 3
 RELEASE DATE: 22-Aug-1999
 PURPOSE:    To enable the family-name of an Employee to be
 corrected, or to change the family name of an employee (after
 marriage, for instance).

IHRC1991_EMPLOYEE_CHANGENAME_S of IHRC1_EMPLOYEE_MGR
   IMPORTS:
     Type View IN_E IHRC1_EMPLOYEE (mandatory, transient, import only)
         INSTANCE_ID (mandatory)
         NAME (mandatory)

   EXPORTS:
      Type View OUT_S IHRC1_STANDARD_PARAMETERS(transient,export only)
         SEVERITY_CODE
         ROLLBACK_INDICATOR
         ORIGIN_SERVID
         RETURN_CODE
         REASON_CODE

   NOTE PRE-CONDITION OF success:
      Employee e EXISTS IN self.managedEmployee
      WITH e.instance_id = in_e.instance_id AND e.status <>"deleted"
      POST-CONDITION OF success:
      e.Employee.name = in_e.name
      out_s.return_code = 1, out_s.reason_code = 1

   NOTE PRE-CONDITION OF employee_status_error:
      Employee e EXISTS IN self.managedEmployee
      WITH e.instance_id = in_e.instance_id AND e.status = "deleted"
      POST-CONDITION OF employee_status_error:
      out_s.return_code = -41, out_s.reason_code = 1

   NOTE PRE-CONDITION OF employee_not_found_error:
      NO Employee e EXISTS IN self.managedEmployee
      WITH e.instance_id = in_e.instance_id
      POST-CONDITION OF employee_not_found_error:
      out_s.return_code = -10, out_s.reason_code = 1
   NOTE: RETURN / REASON CODES
      + 1/1          Update successful
      - 10/1         Employee ID not found
      - 20/1         Employee ID missing in imports
      - 20/2         Employee Name missing in imports
      - 41/1         Employee not updated since flagged as deleted
      - 41/OTHER     Employee update action failed
      - 60/ANY       Persistent storage failure
```

Figure 5.6 Public Operation Specification Example 2

NOTE: Component Manager converts the parameter names used in Spex, to the import and export view names required by Gen. It prefixes the view name with in_ or out_ unless this prefix has already been given to the parameter name. Where the parameter's type is a view type in COOL:Spex, then the base type name will be shown in the Gen import/export view, but the contained attribute views will correspond to the attributes from the Spex-defined view type.

Release History

We recommend providing a list of previous releases of the operation in a *NOTE*. The releases are listed in newest to oldest order, so the current release is listed first. The component provisioner may choose how many earlier releases to include in the list. The suggested format is shown in Figure 5.7.

```
NOTE RELEASE HISTORY:

      02_00 01-Apr-99 Multiple managers for Employees supported.

      01_01 16-Apr-98 Changed to recognize and support "soft
                                          deletes" of Employees.

      01_00 22_Feb-98 Initial release of Version 1.
```

Figure 5.7 Example of a Public Op. Release History within an Action Diagram Note

Public Operations Offered as Both Transactions and Sub-Transactions

Apart from UI-bearing operations, operations offered as transactions, are likely to be offered as sub-transactions, but not vice-versa. The implementation of such a transactional operation need only involve a single USE statement that invokes its corresponding sub-transactional operation.

In this case:

- The sub-transaction should not be given combined import/export views (that is, <exported> import views, or <imported> export views) because procedure steps do note support these.
- The Notes of the sub-transaction need not be repeated in the corresponding transaction.

Transactional operations may often include special, but standard, processing that is needed at the transaction level, for example:

- Security checking
- Standard client/server data exchange, for example, USER_ID, SERVER_TIME.

In this case the description of the transaction should state the extra processing involved and then state: *OTHERWISE SAME AS: operation_name*.

Interface Type Model

Use the Interface Type Model diagramming tool to construct the interface type model.

The interface type model defines the information that the interface can retrieve. This could be by directly storing the information, or by obtaining it from other components, or by calculating it. If the operations of the interface cannot obtain this information in some way, it should not appear in the interface type model. The operations may also create, modify and delete this information. Effectively, the interface type model is a model of the information that is recallable by the interface. It does not, in anyway, express how the information is stored or derived.

The pre-and post-conditions of an operation can refer to both the interface type model and the operation's parameters.

There is one interface type model per interface.

In COOL:Gen, the interface type model diagram shows the entire interface type model. That is, the interface type model cannot include types, relationships and attributes which are not shown in the diagram.

The interface type model diagram is created when an interface is added to a COOL:Gen model. It automatically includes the interface type itself, and this must not be removed from the interface type model.

It may also include *specification types*, which define further groups of attributes that the interface retains, and defines the cardinality of this information relative to the interface itself. Each specification type appearing in the interface type model must be directly or transitively related to the interface type.

NOTE: The interface type model is simply the attributes of the interface, organized into an entity relationship model. This enables the multiplicities and other invariants of the attributes to be expressed visually. Single-valued attributes of the interface, can be included as attributes of the interface type itself. However, single-valued attributes are quite unusual in the interfaces of business components, so most attributes are assigned to specification types. This enables their multiplicity relative to interface and other attributes to be expressed.

If COOL:Gen enabled the data type of an attribute to be non-scalar, that is, an array or structure, then the attributes of the interface could all be directly attached to the interface type itself. However, this would be difficult to understand by a user; it is much easier to comprehend when the attributes are organized into a type model.

The following section defines the rules that apply to specification types.

Specification Types

A specification type defines a collection of attributes referenced by a component specification. Unlike an interface, it does not define behavior. That is, it cannot own operations.

NOTE: In the **CBD*96*** Standard, operations were "factored" to specification types. This practice has been discontinued in CS/*3.0*.

Specification types are used in:

- The interface type model.
- The parameters of public operations.
- Pre- and post-conditions.

A specification type must own attributes.

A specification type may have relationships with itself, with other specification types and with interface types.

A specification type may be a subtype or supertype of other specification types.

A specification type that has been included in (that is, referenced by) an interface type model must be directly or indirectly related to the interface type for that interface.

A specification type is normally defined as transient. That is, is has no corresponding table in a database design.

⊕ ADVANCED PRACTICE. The specification types included in interface type models may be defined as persistent in the component implementation model, enabling create/update/delete/read actions to be directly performed on the specification type occurrences within the implementation. The fact that a specification type is defined as persistent is not relevant to the consumer, so this should not (ideally) be apparent in consumed specifications.

A specification type usually has at least one identifier, unless it is given the "one occurrence" property.

A specification type may be designated as a business object type. This is for documentation only, and has no downstream significance. It is not necessary for business object types to own transactional operations, although you need to be aware that COOL:Gen issues a warning message if they does not.

The same specification type may appear in the parameters and/or interface type model of several interfaces.

COOL:Gen supports views of specification types (that is, subsets of attributes) within parameters, but not within interface type models.

See Appendix B, “Recommended Naming Conventions,” for the recommended naming conventions.

For a given interface, a specification type may be used in a parameter, without being included in the interface type model. It may also be included in the interface type model, without being used as a parameter.

You should write a definition for the specification type in its description panel.

EXAMPLES.
1. A component that calculates the number of days between two dates has two inputs parameters, both of type Date, and an output parameter which is an Integer. The interface type model is empty, since the interface does not need to recall any information.

2. To simplify the expression of certain post-conditions, the type model of the Human Resources component refers to the Length_of_Service attribute. But this attribute never appears in the inputs or outputs of any operations, although Service_Date (= date of joining) does.

Work Sets

Public operations may import and export work set attributes.

We recommend that specification types are used in preference to work sets, since work sets cannot be included in subject areas, interface type models, or specification model diagrams.

Work sets are unsuitable for use in interface type models.

6 Component Implementation Model

Overview

A component implementation model is a COOL:Gen model containing the internal design of the component. This is, typically, lots of action diagram logic and a database design, plus the component specification.

A white-box component delivery must include the component implementation model. A black-box component delivery does not.

The standards in this chapter apply to delivered component implementation models. The rules for the implementation model are minimal, allowing considerable implementation flexibility.

These standards will also be found useful when developing a component for internal use.

Implementation Model Contents

The component implementation model must contain everything that is in the component specification model (see Chapter 4, "Component Specification Model," plus:

- One or more implementation subject areas.
- Internal types (entity types and work sets).
- Public operation implementations.
- Action diagram statements (not just Notes).
- Further internal operations or free standing action diagrams.
- Source code and/or modules for external action blocks.
- Specifications of consumed components (if any).
- Data storage design (if component directly provides persistency).
- Dummy procedure steps (if required).

Appendix B, "Recommended Naming Conventions," provides recommended naming conventions for:

- The implementation model.
- The implementation subject area.

- Internal entity types.
- Internal action blocks.
- Operations libraries.
- Cascade libraries..

The following sections contain further rules and guidance.

Internal Types

An internal type is an entity type or work, set referenced by action diagrams within the implementation model, which is not referenced by the component specification.

Internal entity types may be persistent or transient. Work sets are always transient.

All internal types must be contained within implementation subject areas. That is, within subject areas, which have their role set to “implementation.”

Component Implementation Type

A component implementation type for the implemented component may be included in one of the implementation subject areas, though this is not required by ***CS/3.0***.

Subject Areas

A component implementation model may contain any number of implementation subject areas.

These may be nested or at the same level.

Implementation subject areas may not be placed within specification subject areas. Specification subject areas may, however, be placed in implementation subject areas. The significance of this is explained below.

We recommend that an implementation model contains just one implementation subject area.

All the component specification types, interfaces and specification types, belonging to *consumed* components, must be placed inside specification subject areas, which are in turn contained within implementation subject areas. Otherwise it would not be possible to distinguish the specification of the component being implemented from the specifications of the components being consumed. Component Manager would not be able to "extract" just the component specification from a component implementation model.

Figure 6.1 provides an example of a component implementation model's contents. The figure on the left is a hierarchical view of the contents; the figure on the right is how the model tree actually appears in COOL:Gen. In this example, the Ordering component's implementation model contains a subject area for its specification and a subject area for its implementation. The specification subject area (SSW_ORD_SPECIFICATION) shows that the Ordering component offers two interfaces (IORM1_INTERFACE and ICHH3_INTERFACE). For brevity, only two operations have been depicted.

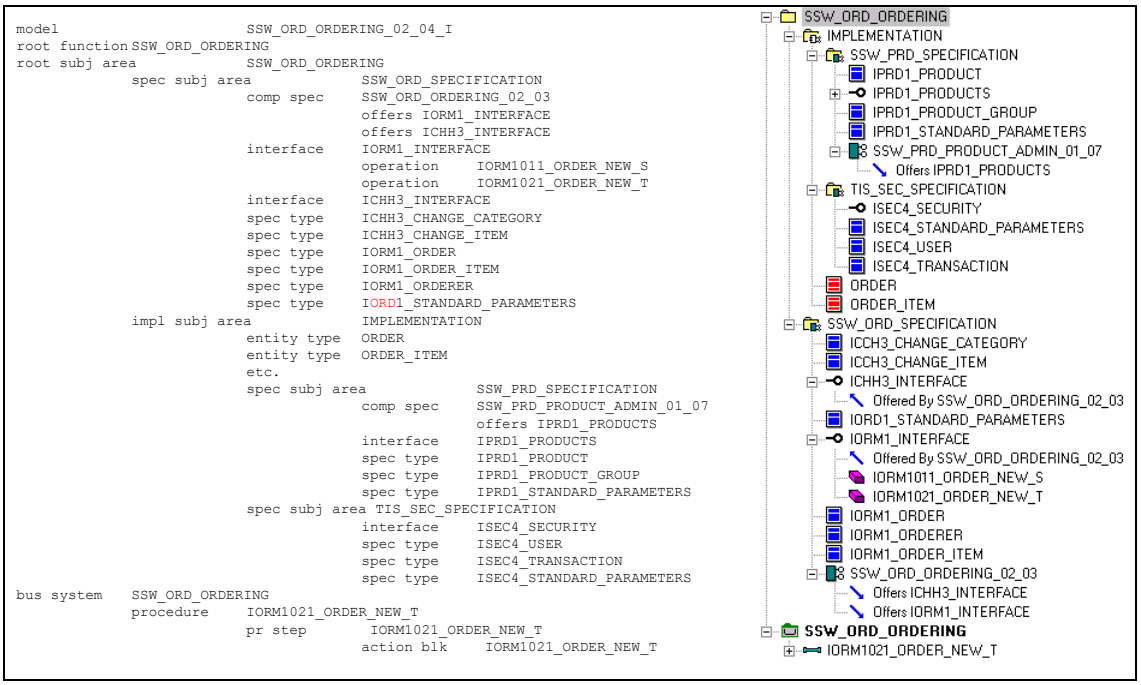


Figure 6.1 Implementation Model Contents

The implementation subject area contains the internal entity types, plus two specification subject areas: one for each consumed component. Note that one of the consumed component specifications includes its component specification type, while the other does not. Component Manager offers you this choice when copying the component specification into a consuming model.

This example follows the recommended naming conventions and subject area arrangement. You can use more subject areas if you wish. Component Manager's conversion wizard and COOL:Spex to COOL:Gen transformer will generally create additional subject areas.

Business Systems

There are no mandatory rules concerning the use of business systems.

In COOL:Gen, all the action diagrams must belong to a business system before they can be generated. This includes all public operations, all consumed public operation stubs, and all internal action diagrams, that is, internal operations or free-standing action blocks.

We recommend that the consumed operation stubs of each component are placed in separate business systems in the consuming model. The business system name should be the same as that in the source component implementation model. Appendix B, "Recommended Naming Conventions," contains the recommended naming convention.

NOTE: When version control (migration) is used to transfer the operation stubs from a source model to consuming model, the business system from the source model is migrated to the consuming model, as well as all the operations.

When Component Manager is used, the original business system name should be typed-in, when Component Manager prompts for the destination business system.

By keeping the consumed operations of each component in separate business systems, the ability to adopt a new version of the consumed operations still exists.

Public Operation Implementation

The operations appearing on the interfaces offered by the component are termed *public* operations. This distinguishes them from internal operations used within the implementation, especially where an object-oriented implementation style has been adopted. Free standing action blocks, and external action blocks, can also be used within a public operation's implementation.

A *public operation implementation* is all the logic that achieves the effect defined in the public operation's specification. This logic consists of:

- The action diagram statements in the public operation's own action diagram.
- Any action diagrams directly or indirectly USE'd by the public operation:
  - If external action blocks are USE'd, then the delivered implementation must include the source code or object modules corresponding to the external action block.

- If operations of *other* components are USE'd, then the delivered implementation model must contain the specifications of the consumed operations, but not the implementation (logic) of the consumed operations, since this is delivered as a separate component.
- Public operations may share action blocks within their implementation. There is no requirement for an internal action block to be exclusively used by one public operation.
- Any execution parameters needed to invoke the operations of consumed components must exist within the consuming implementation model. For example, the transaction codes, source/object module names and operations library names for the consumed operations need to be known by the consuming model
- Any *internal* action blocks may be defined as EXTERNAL, to enable, for example, component persistency via a non-relational database or the logic to be written in an alternative language.

NOTE: To simplify the extraction of a component specification from an implementation model when the model migration capabilities are used, we recommend that action diagrams of public operations exclude action views, local views, and any action statements, other than a single USE statement that invokes another action diagram.

This advice does not apply to user interface-bearing public operations.

CS/*3.0* does not include any further rules about how the operation implementation should be constructed, so there is considerable flexibility in how a component implementation is realized.

Data Storage Design

Components may support persistency. This means that the component appears to store all the data shown in the interface type models, *even if the component stops executing and restarts later*.

Persistency can be implemented using a COOL:Gen-generated relational database, in which case the implementation model must contain the Data Storage and Structure Design (also known as the Technical Design), which defines a table for each internal type that is a persistent entity type.

External action blocks may reference other data stores, and the design of these data stores is also a part of the implementation.

Persistency can also be achieved by invoking the operations of other components, which directly or indirectly have access to data stores.

No other component or software may access the component's persistent storage, otherwise the component is not encapsulated. The only exception is where two releases of the same component are *storage aware* or *storage compatible*.

⊕ ADVANCED PRACTICE. Within an *implementation model*, it is permissible for the specification types of the interfaces being implemented to be defined as *persistent* rather than *transient*.

This technique makes it simpler to develop implementations and may result in better run-time performance.

However, it becomes more difficult to alter the implementation in the future. An interface should not be changed, except by extension, once it has been published (see Appendix A, "Component Upgrades"). All specification types appearing in *specification models* must be transient; internal attributes that are not referenced by any operation should not appear in the specification model. This may cause additional work when extracting a specification from an implementation model that includes persistent specification types.

Component Dependencies

The components upon which the delivered component depends must be listed in the component documentation, as explained in the Chapter 8, "Component Documentation."

- There may be *specification dependencies* upon other components or interfaces, which should then re-occur in all future implementations of this component, although the component or interface release numbers may change.
- There may be *implementation-only dependencies* upon other components, which may be removed or altered in subsequent releases (versions or revisions) of this component.

The delivered component implementation model must contain the stubs of all the consumed operations. These are actions blocks that contain import and export views, and operation specification Notes, but no logic statements.

Dummy Procedure Step

In certain situations, the component implementation model needs to include a "dummy" procedure step (within a "dummy", single-step procedure).

This is because COOL:Gen will not generate object modules for an action block unless they are called (directly or indirectly) by some procedure step. Since sub-transactional operations are modeled using action blocks, you have to create an artificial procedure step that USEs each sub-transaction, to obtain the corresponding object modules.

The rules for a dummy procedure step are that:

- It is a non-display step containing a succession of USE statements.
- There is one *USE action-block* statement for each sub-transaction of the component.
- You should name the dummy procedure and its step *SUB_TRANSACTIONS,* or a similar name.

This procedure step is never executed, hence the term "dummy."

NOTE: We recommend that component implementation models contain just one business system (plus a business system for each component being consumed, which contains the stubs of consumed operations). Where the operations being implemented reside in multiple business systems , we recommend that all the public operations belong to the same business system. If, however, you do assign the sub-transactions to several business systems, then you will need a dummy procedure step in each of those business systems.

The dummy procedure step is not needed in all circumstances. Do not create a dummy procedure step when:

- The component offers only transactional operations.
- Dynamic linking of sub-transactions is preferred *and* the target platform for the component executable is Unix or Windows NT.

NOTE: The COOL:Gen Component Packaging facility can generate *operations libraries* for Unix or Windows NT, without the sub-transactions appearing in any procedure steps beforehand. The generated operations libraries contain the "binary code" for one or more sub-transactions.

* An operations library for Windows NT is known as a dynamic link library or DLL.
* An operations library for Unix is known as a shared library.

Operations libraries are called at run-time, enabling dynamic linking. This is in contrast to static linking, which happens at development-time.

COOL:Gen also supports dynamic linking for MVS environments. This is an option within Cooperative Packaging, and still requires a dummy step to be created.

It is permissible to for a component implementation model to contain several dummy procedure steps.

NOTE: You may find reasons for actually executing the dummy procedure step. For example, you use it to run tests on the component. In this case it is suggested that:

* The procedure step is given import and export views that cover all the imports and exports of the sub-transactions therein.

* The first eight characters of the operation name are used as the operation's associated command. (This makes the commands unique and helps the developer know which command to use.)

---

7 Component Executable

Overview

This chapter defines the standards that apply to a component executable.

As explained in Chapter 3, “Component Delivery,” a black-box component is delivered without the implementation, that is, without the source code. Instead, the customer gets the component specification model and the software (the component executable).

The component executable is a collection of modules, which may be load modules, operations libraries, object modules and/or database definition statements. The *component executable* is this complete collection of modules.

Contents of the Component Executable

The content of the component executable depends on the use of the components, as follows:

- For components offering transactional operations, the component executable must include a set of *load modules* (executable files), which together contain all the transactional operations.

  A transactional operation is typically delivered in its own load module. However, multi-operation load modules will be favored for some platforms, for performance or administration reasons.

- For components offering sub-transactional operations, the component executable must include a set of *operations libraries* which, together, contain all the sub-transactional operations of the component, plus any internal operations and sub-routines called by those operations.

  Operations libraries are the preferred means of delivering sub-transactional functionality. Operations libraries enable run-time linking of sub-transactions, making it much easier to replace the sub-transactions with new versions or revisions in the future. COOL:Gen directly supports the generation of operations libraries for MVS, Unix and Windows NT platforms.

  If operations libraries are not delivered, then object modules must be delivered for each sub-transaction. These enable the customer to statically link the sub-transactions into their consuming software. Even if operations libraries are delivered, the provisioner may choose to supply both operations libraries and the object modules for the sub-transactional operations.

- For components offering sub-transactional operations, for which no operations libraries have been delivered, the component executable must include a set of *object modules*, which are the sub-transactional operations of the component, plus any internal object modules called by those object modules.

  Object modules for any external action blocks must also be delivered with the component.

- For components supporting persistency, the component executable must include the generated database definition statements (DDL) for the relational database and/or the equivalent definition for non-COOL:Gen-generated data stores.

  For storage compatible component upgrades the “delta DDL” must be also be delivered, to facilitate extension of existing database tables.

  For component upgrades that are not storage aware, a data conversion program will need to be delivered. See Appendix A, “Component Upgrades.”

- For database products supporting static binding the component executable must include a directory or library of bind modules. These bind modules contain “plans” of how the database will be accessed and updated.
  - DB2 for NNS requires DBRMs (database request modules).
  - DB2 for Windows requires .BND files.

Execution Parameters

The transaction codes, operation library names and object module names used within the component executable must have exactly the same values as in the corresponding specification model. Otherwise, it will not be possible to invoke the public operations from another component or application.

Black-Box Component Dependencies

Where the delivered component depends on the operations of other components, this must be clearly stated in the component documentation, as explained in Chapter 8, “Component Documentation.”

The provisioner must supply the specification of the consumed component, or its interfaces. The customer must then supply these other components.

The provisioner may also deliver a component executable which contains further consumed components. In this case, the dependency need not be documented, although the provisioner may choose to inform the customer, explaining that the consumed components are already “embedded” within the delivered software.

Where the component offers transactional operations, and the executable is generated to run in environments other than MVS, Unix or Windows NT, then any consumed sub-transactional operations should be statically linked into the delivered component executable by the provisioner. This is because COOL:Gen does not support calls to operations libraries (that is, dynamic linking) in other than MVS, Unix or Windows NT environments. Only static linking is available.

The delivered component executable should *not* include any modules for the stubs of consumed operations. These stubs appear in component implementation models, but are not required within the component executable.

8 Component Documentation

Overview

This chapter defines the information that must be supplied with a delivered component. It is divided into three sections.

- Component Specification
- Delivery Documentation
- Test Data and Results.

Component Specification

The component specification is supplied as a COOL:Gen model.

- For a black-box delivery, the specification is delivered in a component specification model. See Chapter 4, “Component Specification Model,” for the details.
- For a white-box delivery, the specification is delivered within a component implementation model. The standards for the implementation model are given in Chapter 5, “Interface,” the rules concerning the specification aspect are in Chapter 4, “Component Specification Model.”

Delivery Documentation

While the component specification in the COOL:Gen model provides a comprehensive description of the behavior of each operation of the component, there is a certain amount of additional information that the component customer needs to know. This must be supplied as a part of any component delivery.

CS/3.0 does not prescribe the delivery medium or format of this information.

The minimal documentation requirements are set out below. Provisioners can extend and embellish this list, as necessary. For example, they may duplicate or summarize information already contained in the specification model.

Minimum documentation requirements are:

- Commercial or informal name of the component.
- Textual description of the component functionality and purpose.
- Differences from the previous release, if any.
- Name of the provisioner and the provisioner contact details.

- Legal terms and conditions: price, warranties, usage constraints, and so on.
- The formal name of the component, as used within the component specification model.

  If the recommended upgrade and naming conventions have been followed, this name will include a version number and release number of the component.
- Version of the COOL:Gen Component Standards to which the component conforms.
- COOL:Gen release that was used to create the component model.
- Names of any other component specifications, or interfaces, upon which the delivered component depends. For example:
  - The documentation should state whether the dependency is a specification dependency (so it should be detailed within the component specification), or an implementation design choice, in which case the dependency would not be evident in the component specification.
  - It is preferable to record dependencies against interfaces rather than components, since this provides the customer with more flexibility over which components to consume.
  - The specifications of these consumed components or interfaces must be supplied. This should be in the component implementation model for white-box components, or in associated models or documents for black-box components.
  - Where an executable component is delivered, the executable may include the executable of all consumed components, in which case the customer need not be aware that the executable was constructed using other components.
- Whether the component supports persistency.
- Whether the component was engineered to support multiple component objects. See chapter10, “Identifiers and Relationships.”
- When the component is an upgrade to a previously available component, it is necessary to document whether the component is storage aware or storage compatible with the previous release and, if not, what data conversion must be done when switching to the new release. See Appendix A, “Component Upgrades.”
- For the component implementation model, if a white-box delivery:
  - Any special design objectives which the implementation designers had in mind. For example, adaptability, high performance, high data integrity, extensibility, minimal work space, and so on.

  - Target run-time environment currently set within the model, if any. For example, the DBMS, TP Monitor, programming language. The component design should have been optimized for and tested under this platform. The customer can, however, change these settings to generate a component executable that runs on some other platform.
  - The implemented dependencies on other interfaces or component specifications, as explained on the previous page.
- For the component executable, if a black-box delivery:
  - Run-time environment (operating system, database management system, TP monitor) required by the executable, including the release numbers of such products.
  - The execution parameters for the component can be summarized in the documentation, although these should in any case appear in the component specification model. These cannot be altered for a black-box component.
  - The mechanism that enables the customer to set a different server identifier value in each installed component.

Test Data and Results

A component may be delivered with test cases and test results, to show the customer the tests that the provisioner has performed. The customer can reuse these tests to:

- Make sure the component is working correctly after installation in the customer's environment. This is especially important for components that depend upon other components.
- Perform regression testing, checking to see that the component is working correctly after some major software change has been instituted, for example, a new database management system, or the replacement of a component upon which this one depends.
- Test a modified version of an implementation component.

The tests are documented as a collection of test cases and results, along with instructions for running the tests. The tests may also be supplied in the form of inputs to a software testing tool. The test environment (hardware, software and release numbers) should be included in the documentation. Where the tests require a pre-loaded test database, the test database, or a DBMS "dump" (for example, Oracle .DMP file, which loads up both the database structure and some test data) should be supplied.

Summary

A component delivery must always include the specification of the component, and the information listed under Delivery Documentation on page 53.

Test data and results are a discretionary part of a component delivery.

The format and media for the delivery documentation is up to the provisioner.

9 Associated Model

A component provisioner may choose to deliver additional COOL:Gen-based functionality to help the customer make better use of the component. This additional functionality is not a part of the component itself.

- For a white-box component, this additional functionality must be delivered in a separate COOL:Gen model, known as the associated model. The additional functionality must **not** be delivered within a component implementation model.
- For a black-box component, the additional functionality may be delivered in an associated model. It may also be delivered in the component specification model, rather than as a separate model.

The following sections provide examples of some of the additional functionality that may be supplied with a component.

The recommended naming convention for the Associated Model is given in Appendix B, “Recommended Naming Conventions.”

Examples of Contents

Test Transactions

A collection of user interface-bearing procedure steps may be supplied, which enable the customer to test and explore all the operations of a component. These save the customer from having to develop his or her own test transactions for the component.

Ideally, the customer could use these test transactions to perform the test cases supplied within the component documentation.

- Some provisioners may choose to supply a single transaction that can be run to check that the component is installed properly, without testing every aspect of the component.
- Some provisioners may prefer to supply a non-COOL:Gen built test harness that allows the customer to check that the component is correctly installed, or that enables the customer to test individual operations.

Maintenance Transactions

Some components require "constants" to be stored in a database before the public operations can be used within line-of-business applications. For example, a table of the Country Codes may be needed. In such cases, the component provisioner could supply a maintenance application (one or more user-interface bearing procedure steps), that allows the system administrator to set up and maintain these "constants."

The user-interface bearing transactions should USE public operations of the component in order to access and update the data store of the component.

NOTE: Components may offer user interface-bearing *operations*, which are an integral part of a component interface and will be documented to the same standard as any other public operation. These are not be delivered in the Associated Model, since they are the component's public operations. The maintenance transactions referred to in the section above, are not *bona fide* operations of the component, and are supplied as a convenience to customers.

Default User Interface Designs

Components may be supplied with default user-interface designs. These may be used by application developers as a starting point for developing a user interface for an application that uses the component. They are delivered as procedure steps with windows or screen layouts. They are not considered to be operations of the component and do not need formal operation specifications.

The test transactions described earlier can act as both test transactions and default user interface designs.

Sample Application

A sample application demonstrates a possible use of the component. It may have been obtained from an early customer, or it may be a usage example created by the provisioner. The sample application can be used by the customer as an installation test, or may be adapted by the customer to meet specific business needs.

Application Translation Blocks

An application translation block is an "adaptor" for a public operation. It translates the application's (or other consumer's) data model attributes into the names that the component employs. The application never calls a public operation directly; it always calls the translation block, which converts the application's terminology into component terminology (and back again), and calls the actual public operation. It is anticipated that this will be a common practice within application models.

A component provisioner may choose to supply these translation action blocks with the component, to save the customer from having to build them. Of course, the provisioner can only provide a "template" which the customer will need to adjust to the specific application's attribute, type and operation names.

Organizing the Associated Model

The additional functionality should be grouped into business systems that are named to reflect their purpose. For example, TEST_TRANSACTIONS, DEFAULT_WINDOW_DESIGNS, SAMPLE_APPLICATION. If additional entity types are involved, these should be placed in a subject area that takes the same name as the business system.

Action blocks and procedure steps should be named so they are readily distinguishable from public operations. For example, all test transactions can be given names beginning with TEST_.

This is particularly important if the additional functionality is delivered within the specification model, instead of in an associated model.

10 Identifiers and Relationships

Overview

This chapter standardizes the identifiers and relationships used by **CS/*3.0*** components. Four kinds types of identifier are described:

- Instance identifiers, for interface type model types.
- Business identifiers, for interface type model types.
- Server identifiers, for installed copies of components.
- Component object identifiers, for run-time instances of components.

Relationships between instances of specification types that are maintained by different components are recorded using instance identifiers. These are termed *cross-component relationships* and are further explained in the Cross-Component Relationships section on page 65.

Instance Identifiers

An instance identifier is required on any specification type that is likely to be referenced from some other component. That is, some other component "remembers" these identifiers in order to form a relationship with an occurrence of the specification type.

However, a specification type registered as a "single-occurrence" does not have identifiers and will not require an instance identifier of its own.

NOTE: It is suggested that each business (object type) appearing within an interface type model is given a instance identifier, since these are very likely to be referenced by other components.

A standard instance identifier must be an attribute that is:

- Named INSTANCE_ID (instance identifier).
- Defined as an identifier of the specification type.
- A fifteen-digit numeric field.
  A standard size allows for generic treatment of cross-component relationships.

The value of an instance identifier:

- Is immutable.
  An occurrence of a specification type receives the identifier when it is first instantiated, and it remains unchanged until it is deleted.
- Has no business meaning.
  It is not intended that the identifier values are made known to business users.
- Is unique within a component object.

References to instances of specification types maintained within a component that has been built to support multiple component objects, require two attributes: the *component object id* and the *instance id*. Where a component is built to support only a single component object (which is often the case), then references to instances of the specification types need only include one attribute—the *instance id*.

Business Identifiers

Specification types usually define identifiers known to business users. These identifiers are called *business identifiers*. **All** instances of specification types must be uniquely identifiable—by an instance identifier, a business identifier, or both.

A business identifier need not be restricted to a single attribute. It may be a composite of several attributes and/or relationships. A specification type may have several business identifiers.

For example, Suppliers are identified by Supplier Number or by a combination of Name and Address. Products are identified by their Product Code. Claimants are identified by their Social Security Number. Order Lines could be identified by the instance identifier of the Order to which they belong plus the line number.

While most business identifiers are expected to be immutable, this is not always the case. For example, they may need to be extended as the number of customers grows. Business Identifiers vary in format from one specification type to another and cannot be used as the basis for a general treatment of cross-component relationships. Hence the need for internal, immutable instance identifiers.

Server Identifiers

Each installed copy of the component executable is allocated a number that is unique across all the installed components for an organization.

Every operation execution must export this unique number, to communicate to its consumer which of the installed copies of the component actually executed the

operation. This can be useful to know when the same component is installed multiple times.

This unique number is the *component server identifier*. The standard parameter name for it is *ORIGIN_SERVID*.

Note that we originally referred to installed components as component servers. The term *installed component* is now preferred.

CS/*3.0* requires each operation execution to export a component server identifier value in an attribute view named *ORIGIN_SERVID*. This is also explained in the Standard Parameters section of Chapter 5, "Interface."

NOTE: SERVID is a fifteen digit numeric attribute. Some organizations may prefer to keep the value of server identifiers less than 2^{32} so the value can be transferred to or from widely-used 32-bit numeric fields.

Organizations do not need to populate this parameter if they do not find it of value.

An operation normally exports the SERVID of the component just invoked. However, when an error occurs during a nested invocation, the ORIGIN_SERVID should export the server identifier of the component which first detected an error. This is valuable when a failure occurs, since it helps the operations support team to identify exactly which of the installed component copies experienced the failure.

For components supplied as white-box components, the provisioner must provide a means for the customer to set a different server identifier value in each installed component.

NOTE: We recommend that the value of the component server identifier is generated by a special action block, that is always named Cccc999v_SETSERVID. Every public operation execution can invoke this action block to obtain the value of the component server identifier. This action block only exports one attribute: ORIGIN_SERVID. The component customer can modify the action block logic, so that each installed component returns a different ORIGIN_SERVID value. Customers do not have to modify the action block, if they do not see the benefit of unique component server identifiers within their organization.

For components supplied as black-box components, the provisioner must provide a documented mechanism that enables the customer to set a different server identifier value in each installed component.

Component Object Identifiers

Components can be engineered to support several instances of the same component at run-time. The component software is installed once, and it then manages several distinct run-time instances, each instance being called a *component object*.

- Each run-time instance has its own persistent data.
- Each operation can only operate on one component object at a time.

A component object is identified by its immutable *component object identifier*, which should have a unique value across all the component objects (run time instances) for a given component specification.

Components can be built to support multiple component objects by:

- Offering interfaces in which every operation imports the component object identifier value.
- Labeling the persistent data with the component object identifier, so the data belonging to each component object can be differentiated. For example, the key of each table of a relational database could include the component object identifier.

Component object identifier values can be “hard coded” into the component consumers, or allocated programmatically at creation-time. (The latter is usually done by some other component which is often called the “factory” for our component’s objects).

Multiple component objects may be useful whenever the specification type instances (in the interface type models of the interfaces offered by the component) can or must be partitioned into distinct collections.

For example, a company manages retail and personal orders through entirely separate departments. The orders use the same Sales system, but the orders must be kept entirely separate from one another. The company buys an Order Management component, which supports multiple component objects. All the retail orders are managed by component object “R”, and all the personal orders are managed by component object "P“. These two component object identifiers are "hard coded" into the consuming application.

In another example, a company with many stores needs to keep track of the stock in each store. It has a stock control application that runs on a mainframe. The stock component which is developed supports multiple component objects, and each component object manages the stock from one store. When a new store is opened, the application allocates a new component object identifier, effectively “instantiating” a new component object for the store. The stock component is unable to report the total value of all stock in all stores. A different component or

the application software has to do this, since an operation can only act on one component object at a time.

So, component objects are simple a different name for a development technique that has existed for years.

⊕ ADVANCED PRACTICE. To support multiple component objects, the component object identifier (COID) should be defined as an attribute of the interface type. This attribute is registered as the primary identifier of the interface.
All operations of this interface are recorded as *instance operations* of the interface; then each operation you define will automatically include an import view for the COID.

There is no requirement for **CS/3.0** components to support multiple component objects. Components that support a single component object are perfectly valid.

Cross-Component Relationships

A relationship between a specification type managed by one component, and a specification type managed by another, must be recorded using the *instance identifier* of one of the specification types.

Figure 10.1 shows an Example of Cross-Component Relationship.

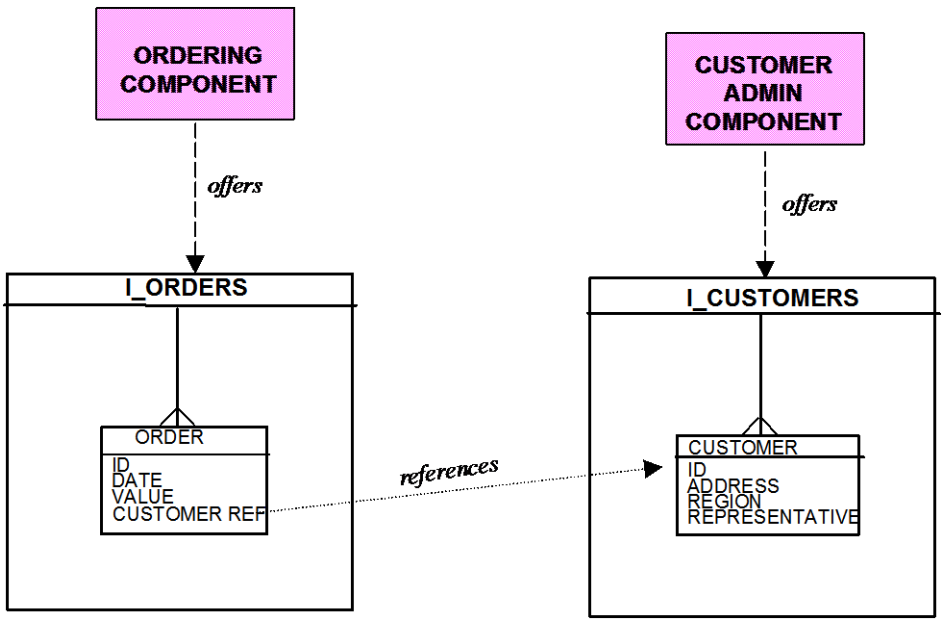


Figure 10.1 Example of a Cross-Component Relationship

In this example, the Ordering_Component offers the I_Orders interface, and the Customer_Administration_Component offers the I_Customers interface. Each Order managed by I_Orders needs to be related to a Customer managed by the I_Customers interface.

The relationship is formed by I_Orders "remembering" the instance identifier of the Customer that placed the Order. This is shown in the interface type model for I_Orders by the attribute Customer_Ref. Customer Ref is what relational database designers would call a "foreign key." The Customer_Ref attribute must contain an instance identifier, not a business identifier, since business identifiers are not immutable, and do not have a standard format.

This simple example explains the rule that *instance identifiers must be used to form cross-component relationships.* A standard format instance identifier enables the referencing attribute to define associations with the instances of various specification types.

CS/3.0 does not require that cross-component instance references are maintained with full referential integrity. There are various mechanisms for maintaining full or partial referential integrity, which are outside the scope of this standard. The integrity may be maintained by the referencing component, or by some mutual consumer. References may be uni-directional, as in our example, or bi-directional. The referencing attribute may be placed in its own specification type if preferred.

Appendix A: Component Upgrades

Overview

This appendix presents recommendations for numbering new releases of **CS/*3.0*** components. It explains how to number new releases of components, interfaces and operations. A summary chart appears at the end of the chapter.

Release Numbering

A component upgrade—designed to correct errors, improve performance, provide additional functionality, and so on—is made available in a new *release* of a component.

A new release of a component is given the same name as the previous release, except its *release number* is incremented using the rules described below. Component naming conventions are summarized in Appendix B, "Recommended Naming Conventions."

Component release numbers are made up of two parts:

- Version number.
- Revision number.

The initial release of a component is assigned release number 1_0. That is, version 1, revision 0.

For subsequent releases:

- If the release represents a new *version*, the version number is incremented and the revision number is reset to 0.
- If the release represents a *revision*, the revision number is incremented.

A revision is a new release that is *specification-compatible* with the previous release. This means that a consumer of the component is not impacted. This allows the new release to be used in place of the old, without impacting any consuming application or component.

A version is a new release that is **not** specification-compatible with the previous one. The introduction of a new version of a component will have an impact on existing consumers of that component and must be managed accordingly.

Similar concepts and rules apply to the *interface* offered by the component, and to each of the public *operations* within that interface. For example, a revision to an interface must be specification-compatible with the previous version of the interface.

Revisions

A revision of a component, interface, or operation must be specification-compatible with its previous release. This means that the specification of the revision must be the *same* as the previous release or an *extension* of the previous release. That is, the new specification must add to the existing one, without contradicting it or removing anything from it.

Component Revisions

A component revision is required when any of the following specification-compatible changes occur:

- Its specification is extended.
- Its implementation or executable is changed, without affecting the specification.
- Its documentation is improved.

The specification is considered extended when any of these occur:

- An interface is revised.
- A new version of an (existing) interface is added.
- A whole new interface is added.

Implementation changes that do not affect the specification will be those that improve the quality of the component (for example, fix bugs or improve performance). Where the component specification has been extended, the implementation must also change to provide the additional behavior.

Changes to a delivered component executable also imply a new component revision, even if the implementation has not changed, since it may affect the installation process. For example, generating for a different database version or generating to a different target language will require a component revision.

The component revision number appears in the name of the component specification type, in model names and in the Delivery Documentation.

NOTE: For a revision that does not change the component specification in any way, it is reasonable to leave the component specification type's revision number unchanged, while the revision number on the component implementation model is incremented. Hence the revision number on the model name may be higher than that on the component specification type, though the version number must not differ.

Interface Revisions

An interface revision is required when the interface specification is extended by any of the following:

- An existing operation is revised.
- A new operation is added.
- The interface type model is extended.

Operation revision is discussed in the next section (on page 70).

New operations (including new operation versions) can be added to the interface. These may be specified in terms of the existing interface type model, or they may require extensions to it. Any interface type model extension required by the new operation must be compatible with the existing model, so that existing operation specifications are unaffected. This means that new specification types can be added and these may be subject to new invariants, but existing types must not be affected, and existing invariants must not be changed.

The following interface type model changes are considered extensions:

- Additional attributes and relationships for existing specification types.
- Additional specification types.
- Additional subtypes for existing specification types.
  This may include additional partitionings.
- New constraints defined against these additional types, subtypes, attributes and relationships.

If the required interface type model changes are more significant and are not simple extensions, a new version of the interface, or even a completely new interface, is required.

Interface revision numbers are recorded in the interface type description, not within the interface name.

Operation Revisions

An operation revision is required when a public operation specification is changed, but is unlikely to have a significant impact on existing consumers of the public operation.

A signature change is always deemed to have significant impact, and requires a new version of a public operation.

Where the changes are limited to the following, then the change will qualify as a revision:

- Further unspecified reason codes for existing return codes.
- Documentation improvements.

Component provisioners are left to decide whether an operation specification change will have a significant impact on a consumer's logic, in cases where the signature of the operation remains unaltered. Where they judge it will have a significant impact, a new version of the public operation should be delivered, rather than simply incrementing the revision number of the existing public operation.

In short, any change to a public operation specification that is not deemed by the provisioner to require a new operation version, is treated as a revision.

Operation revision numbers are recorded in the operation's action diagram description, not within the operation name.

Versions

A new version of a component, interface, or operation need not be specification-compatible with its previous release, and so, in general, it will involve more effort than replacing the old with the new. Typically, the logic in the consuming implementations will need to be altered to take advantage of the newly delivered version.

NOTE: In some cases, a new version may be created for other reasons, such as market requirements or numbering consistency across a portfolio of components, even though the new version is technically specification-compatible with the previous one. These cases are not covered here.

Component Versions

A new component version is required when a change is made that is **not** specification-compatible. This occurs when an interface is removed from a component.

When a new version of an interface is designed, the component should offer both the original and new versions of the interface. This counts as a revision to the component. However, if the original version is discontinued, this requires a new version of the component.

The component version number appears in the name of the component specification type, the delivered model name (see Appendix B, "Recommended Naming Conventions") and the Delivery Documentation.

Interface Versions

A new interface version is required when:

- The interface type model is changed in an incompatible way. That is, the new type model is not an extension of the previous one.
- An operation is removed from the interface.

Incompatible interface type model changes mean that existing operation specifications are affected and a new interface or new interface version must be introduced.

The following interface type model changes are considered incompatible:

- Removal of specification types, subtypes, attributes and relationships.
- Any change to existing invariants on specification types, attributes and relationships. This includes addition of invariants as well as removal of invariants.
- Any change to the data types of attributes.

NOTE: Name changes to specification types and attributes used in operation views are, strictly, incompatible changes since they affect the consumer. However, where the altered operation specification is not migrated into the consumer model, the consumer could be re-linked to the new object module, or dynamically linked to the new operations library, without problem.

For example, changing the cardinality of a relationship membership from one to many is a change to an invariant and is, therefore, an incompatible change. Similarly, changing the length of a text attribute is considered an incompatible change.

Interface version numbers are recorded in the interface name as part of the interface prefix.

When a new interface version is created, any operations that are retained (carried forward onto the new interface and changed accordingly) will require a name change to show that they are different from a prior version. However, the opportunity exists to carry forward the version number and revision number of the corresponding operation on the previous interface version. This may be done, if required, and may be useful for distinguishing between multiple versions of the same operation on the same interface, if both were carried forward, or for identifying the consumer changes needed to incorporate the new component version.

Operation Versions

A new operation version is required when the operation specification is altered in a way that will have significant impact on the current consumers.

A signature change has a significant affect, and always requires a new version.

Even when the signature remains the same, the operation specification might be changed in a way that has significant impact on the consumer. For example:

- Return code changes, that is, additions, deletions or changes in meaning.
- Reason code changes, that is, additions, deletions, or changes in meaning to any reason codes that have been detailed as part of the operation specification.
- Changes to pre-conditions and post-conditions.

The component provisioner is left to judge whether a non-signature change is liable to require significant logic changes for current consumers. Where they judge the change will have significant impact, a new version of the public operation is supplied, and the old version should continue to be supplied.

When a new interface is required, new versions of all the operations it retains are required, including those operations that were not affected by the change.

Documenting Release History

Instead of recording just the current revision number and date in the component specification and interface type description panels, a release history may be provided.

The releases are listed newest first, oldest last. For example:

```
NOTE RELEASE HISTORY:

      02_00 31-Feb-99 Interface ISS1 dropped

      01_04 01-Apr-98 Interface ISS2 added

      01_03 01-Jan-98 Identifier generation algorithm
                  changed - used by many operations

      01_02 01-Dec-97 Indexes added to database design to
                        improve performance

      01_01 01-Feb-97 Operation XYZ added to interface ISS1

      01_00 01-Jan-97 Initial Release
```

The component provisioner decides how many releases to include in the list. The list may include revisions which did not change the specification.

Storage Compatibility

A new component release must be able to access the data stored by the previous release.

There are three main ways of doing this:

- Data conversion.
  A separate one-off migration program is delivered with a new component release. This program converts the data from the old storage format into the new, and has to be run before operations of the new release can access the previous release's stored data.
- Storage aware release.
  The new component understands the storage format of the previous release and can read all the data stored by the previous release, although it might or might not subsequently store data in that format. In this case we say the new release is *storage aware* of the previous version.
- Storage compatible release.
  The storage format of the new release is the same or is a compatible extension to the previous release. In this case we say the new release is *storage compatible* with the previous version.

  Storage compatible releases are a special case of storage aware releases. This means that, when installing the new release, only additions to the current database definition statements need to be compiled. Of course, if this is an initial purchase for a customer and not an upgrade, then all database definition statements must be compiled.

Upgrade Summary

The table "Revisions and Versions Required by Various Kinds of Upgrade" summarizes whether a revision or new version of the operation, interface or component is required for the main classes of change to a component. Refer to previous paragraphs to understand the meaning of "extension" and "incompatible change."

TYPE OF CHANGE:	operation numbering:	interface numbering:	component numbering:
OPERATION CHANGES:			
Only the implementation (of one or more operations, or the data store design) has changed	no change	no change	revision
No significant impact on existing consumers	revision	revision	revision
Signature change, or significant impact on existing consumers	version (see note 1)	revision	revision
INTERFACE CHANGES			
New operation		revision	revision
Interface type model extension		revision	revision
Operation removal		version	revision
Interface type model incompatible change		version (see note 2)	revision
COMPONENT CHANGES:			
New Interface			revision
Interface removal (e.g. an interface replaced by new version)			version

Revisions and Versions Required by Various Kinds of Upgrade

Note 1: The original version of the operation must continue to be supported by the new release of the interface; otherwise this amounts to operation removal.

Note 2: The original version of the interface must continue to be supported by the new release of the component; otherwise this amounts to interface removal.

Appendix B: Recommended Naming Conventions

Introduction

These conventions are not a mandatory part of **CS*/3.0***.

However, it is suggested, that component developers adopt these conventions, unless there is a good reason not to do so. The rationale for each naming convention is given, so where you have other objectives, or other standards to comply with, or simply disagree with the stated rationale, then it is reasonable to establish a different convention – or no convention at all – in your organization.

The following notation is used in the naming formats:

- *lower case italics* indicates an element of the name which is variable
- *UPPER CASE ITALICS* indicates an element of the name which is a fixed literal.

Component Specification (Type) Name

Form the component specification (type) name as follows:
ppp_ccc_componentname_vv_rr

where...	is...
ppp	Unique identifier of the component provisioner
ccc	Unique code for the component
componentname	Succinct textual name for the component (16 characters maximum)
vv_rr	Release number comprised of: • *vv* – Version number, which may be one or two digits; the first version is normally numbered 01. • *rr* – Revision number, which may be one or two digits; this number is normally 00 on the initial release for a new version.

Examples of component specification (type) names:

- CSF_HRC_HUMAN_RESOURCES_1_1
- SSW_PRM_PRODUCT_MANAGMNT_02_00

Rationale:

- Component specifications names are unique, and convey further useful information.
- Different releases of a component specification can be readily distinguished, since the version and revision number are incorporated into the name.
- Components acquired from different provisioners have unique names, even if the same component code has been used.
- The component code (which is reused in other naming conventions) is readily evident from the model name.
- Unique model names can be formed by extending this component specification name.

Model Name

Form model names as follows: *ppp_ccc_componentname_vv_rr_m*

where...	is...
ppp_ccc_componentname_vv_rr	The component specification name. **But,** the release number of an implementation model and associated model can bear a *higher revision number* than the specification, owing to revisions that do not change the specification. The version number of implementation and associated models must be the same as for the corresponding specification.
m	Type of model: • S – Component Specification Model • I – Component Implementation Model • A – Associated Model for a component delivery • other - Used to distinguish the development status of components still under development

Examples of model names for the component specification SSW_PRM_PRODUCT_MANAGMNT_02_00.

- Component specification model name (delivered with a black-box component): SSW_PRM_PRODUCT_MANAGMNT_02_00_S
- Component implementation model name (delivered with a white-box component): SSW_PRM_PRODUCT_MANAGMNT_02_00_I
- Associated model name (additional delivered functionality, not a part of the component): SSW_PRM_PRODUCT_MANAGMNT_02_00_A
- Component implementation model still undergoing unit testing: for example SSW_PRM_PRODUCT_MANAGMNT_02_01_U. Note that this is the first revision to version 2 of the component. The component specification must be version 2, but may bear a lower revision number (that is, 00) if the implementation was changed but the specification was not.

Notes:

1. This convention is unsuitable for models containing multiple component implementations (not recommended) or specifications.
2. The short model name can take the form: *cccvvrrm.IEF*

Rationale:

- Model names are unique and convey useful information.

- The component specification version that this model supports is readily evident.
- Component models acquired from different provisioners have unique names.
- Multiple releases of a component model can coexist in the same encyclopedia.
- The content of the model (implementation, specification-only, or associated functionality) is readily evident from the model name, and all can coexist in one encyclopedia.
- The component code (needed in other naming conventions) is readily evident from the model name.

  If two models acquired from different provisioners incorporate the same component code, and both these components will be deployed in the same computing environment, then it is advisable to change one of the component codes, since other naming conventions depend upon the component code being unique across the computing environment.

Interface (Type)

Form interface names as follows: *Ixxxv_ interfacename*

where...	Is...
I	Leading literal
xxx	Three alphabetic characters forming the interface code
v	One-digit interface version number
interfacename	Succinct textual name for the interface, maximum 26 characters

Examples of interface names are:

- IEMP1_EMPLOYEE_MANAGER
- IPRB3_PRODUCTS
- IACC1_INTERFACE
- IAC1_ACCOUNTS (see Note 4)

Notes:

1. Where an interface manages multiple occurrences of the concept mentioned in the interface name, then, by convention, the keyword MANAGER or MGR is added to the interface name. Otherwise, the interface name is likely to clash with a specification type name. For example, interface IEMP1_EMPLOYEE_MANAGER manages specification type IEMP1_EMPLOYEE.
2. An alternative convention is to pluralize the interface name.
3. The practice of making the *interfacename* the literal value INTERFACE is acceptable, although it conveys less meaning to the reader.
4. Previous versions of the COOL:Gen Component Standard allowed two character interface codes. These are still acceptable, to allow for backward compatibility—and since these naming conventions are guidelines, not mandatory standards.

Rationale:

- To provide each interface with a unique name, so multiple interfaces can coexist in models.
- To readily distinguish interfaces from the non-interface types in COOL:Gen models.

- To enable different versions of an interface to exist in a model. For example, a component specification may offer the original version and a new version of an interface; these are different interfaces, so they must bear different names.
- To define a 5 character code *Ixxxv*, which is reused in the names of various interface features.

Specification Subject Area

Form the specification subject area name as follows: *ppp_ccc_SPECIFICATION*

where...	Is...
ppp	Unique identifier of the component provisioner
ccc	The component code, as used in the component specification type name

Examples of specification subject area names:

- CSF_HRC_SPECIFICATION
- SSW_PRM_SPECIFICATION

Notes:

1. This guideline refers to a single specification subject area that contains all specification elements. No guideline is offered where multiple specification subject areas are defined.

Rationale:

- To provide a unique name for the specification subject area in consuming models. The name needs to be unique because:
  - The specification subject area is imported into models that consume this component specification, even when a single interface or operation is consumed.
  - To avoid subject areas being automatically renamed (when using migration or Component Manager to consume a component specification within an encyclopedia-based model).
  - To avoid the specification subject being of the consuming model being incorrectly re-positioned, when a name clash occurs while using Component Manager on local models.
- So that the component, to which a consumed interface or operation belongs, is visible in a consuming model.

  The component specification type need only be imported into the consuming model when the specification properties – the specification release number (version + revision) and invariants – need to be visible in the consuming model.

Specification Type (or Work Set)

Form specification type names as follows: *Ixxxv_spectypename*

where...	Is...
*Ixxxv_*	Interface prefix, in cases where the specification type is referenced by one interface. In cases where the specification type is referenced by several interfaces of the component, you may prefer to set xxx to the component code. So the prefix is Icccv.
spectypename	Succinct, descriptive name for the specification type

Examples of specification type names:

- IEMP1_EMPLOYEE
- IEMP1_JOB
- IPRB3_PRODUCT_SALE
- IAC1_ACCOUNT
- IHRC1_RESULTS (shared specification type)
- IHRC3_USER_INFO (shared specification type)

Notes:

1. In the above examples, HRC is a component code, not an interface code. The version number given after HRC is not the component version, but the specification type version.
2. Specification types which are referenced by just one interface (in its interface type model, and/or its operation parameters) are given the same prefix as the interface type.
3. Specification types intended to be referenced by any number of interfaces (in its interface type model, and/or its operation parameters) are prefixed *Icccv*, where *v* allows for several versions of this type to coexist in one model.
4. There is no requirement to rename a specification type which was originally given an interface prefix, and which is subsequently referenced by other interfaces.
5. You may wish to prefix every specification type with *Icccv* from the start, since they are all potentially shareable.

6. **CS/*3.0*** permits the use of work sets in component specifications. They may not be used in interface type models, but operation parameters may be views of work sets. However, it is suggested that all these work sets are "promoted" to specification types. If work sets are referenced by parameters, they should follow the same naming convention as specification types.

Rationale:

- Provides a unique name for the type within a model.
  While several interfaces may reference the same underlying type in there type models, the attributes may vary.
- Allows the internal entity types to be named without suffixes or prefixes. (COOL:Gen does not allow specification types and entity types and interface to bear the same name.)
- The interface which this type describes is readily identifiable, without the need to inspect the interface type model.
- Shared specification types adopt a similar naming structure.
- Continuity with **CBD*96*** conventions.

Public Operation Name

Form public operation names as follows: *Ixxxvoow_typename_action_g*

where...	is...
Ixxxvoow	Unique eight-character prefix for the operation, which becomes the operation's object module name. This code is comprised of: • *Ixxxv* – The Interface Prefix of the interface to which this operation belongs • *oo* – The Operation Code, which must have a unique value within its interface. Usually numeric • *w* – The Operation Version Number, enabling an interface to support several versions of the same operation
typename	Name of the type upon which the operation acts. This is usually a specification type, but could be the interface type itself. Avoid using underscores and spaces in the *typename.* This may be a consistently used abbreviation of the full *typename.*
action	Verb summarizing the action that the operation performs. Sometimes several words will be needed to clarify or distinguish the action; avoid using spaces or underscores within the *action* name.
g	Operation category: • S – Sub-transactional Operation • T – Transactional Operation with no user interface support • U – transactional operation with User Interface Support.

Examples of public operation names:

- IEMP1011_EMPLOYEE_ADD_T
- IEMP1021_EMPLOYEE_ADD_S
- IEMP1031_JOB_CHANGE_S
- IEMP1032_JOB_CHANGE_S (a new version of the operation above)
- IAC10011_ACCOUNT_OPEN_S
- IAC10021_ACCOUNT_CLOSE_S
- IPRD3011_PRODGRP_MODDISCOUNT_U
- IPRD3021_INTERFACE_UPDATE_S

Notes:

1. In COOL:Gen models:
   - Sub-transactional operations are represented by BSD action blocks.
   - Transactional operations are represented by non-display procedure steps.
   - User interface-bearing operations are represented by display procedure steps.
2. Where a two-character *xx* interface code is used, the operation code is the three characters *ooo*, and the operation prefix is *Ixxvooow*.
3. An interface type may directly own attributes, which are not "factored" to any specification type. In this case, an operation that updates these attributes is focused on the interface type itself, rather than on a specification type. The last example above is intended to illustrate this possibility. The interface was named INTERFACE to emphasize this point.

⊕ ADVANCED PRACTICE. Where new component objects can be created and deleted programmatically at run time, then these will be performed by the create and delete operations on the interface itself. For example, IPRD1031_INTERFACE_CREATE_S and IPRD1041_INTERFACE_DELETE_S.

Rationale:

- To ensure operation names are unique across an organization, preventing name clashes when they are consumed.
- The owning interface is readily evident from the name.
- The first eight characters are unique, enabling the default source module name set by COOL:Gen to be to be acceptable and meaningful.
  - COOL:Gen creates the source and object module names for an operation from the first eight characters of the action diagram name. By providing a unique first eight characters, there is no need to adjust the names created by COOL:Gen.

    Furthermore, these names allow two operations versions to coexist, and same operations versions on different interface versions to coexist.

⊕ ADVANCED PRACTICE. Some organizations may prefer to use "versionless interfaces." In this case, incompatible interface type model changes and dropped operations are treated as a *revision* to the interface rather than as a new *version*. This violates the recommendation given in Appendix A, “Component Upgrades,” because interface stability has not been preserved, and existing consumers are likely to be impacted by the change. For versionless interfaces, the operation name format is Ixxxooow.

⊕ ADVANCED PRACTICE. Some organizations have found it useful to provide operation prefixes that are longer than eight characters. The first underscore in the

operation name is considered to terminate the operation prefix. Remember that the first eight characters should be unique, so that the module names generated by COOL:Gen (from these characters) are also unique.

Parameters of Public Operations

In COOL:Gen, parameters are represented using import and export views.

Form type and group view names as follows:

- Import view names begin *in_[qualifier_]*
- Export view names begin *out_[qualifier_]*
- Combined import/export views (that is, <exported> import and <imported> export views) are named *inout_[qualifier_]*

The qualifier may be used to distinguish several views of the same type or to improve the semantics of the view.

Notes:

1. Alternative view names (for example, *import*, *export*, *impexp*) can be used, as long as that usage is consistent for all operations of the interface.

Rationale:

- Input and output data can be readily distinguished in both the specification and the implementation.
- COOL:Gen requires the views of a given type and direction to have a unique name

Business System, Root Subject Area, Root Activity

Form the business system name, root subject area name and root activity (function) name as follows: *ppp_ccc_componentname*

where...	is...
ppp	Provisioner code, as in the component specification name
ccc	Component code, as in the component specification name
componentname	Textual name for the component, as in the component specification name

Examples of business system and root subject area names are:

- CSF_HRC_HUMAN_RESOURCES
- SSW_PRM_PRODUCT_MANAGMNT

Notes:

1. The root subject area, the root activity and a business system are automatically created when a new model is opened. The root subject area, root activity and business system are given the same name as the model. So, name the new model in one of the following ways:
   - Name the new model *ppp_ccc_componentname*, so the root subject area, root activity and business system get the correct name, then alter the model name so it include the version number, revision number and model type.
   - Name the new model *ppp_ccc_componentname_vv_rr_I*, and then remove the version number, revision number and model type code from the root subject area, root activity and business system.
2. It is recommended that all the public and internal operations of a component implementation are placed in the default business system. This has the same name as the root subject area and root activity. The public operations of each consumed component should be placed in separate business systems; these business systems should carry the same name as in their respective source models.
3. The root activity is always a function, so is also known as the root function.

Rationale:

- To provide a tidy name for the root subject area, root activity and business system.

- To exclude the version number, revision number and model type code from these names so they do not have to be altered, or become misleading, when models are copied to form new versions or model types.
- To provide unique business system names, in case the business system of a consumed component is migrated into the consuming model.

Implementation Subject Area

Form the implementation subject area name as follows: *IMPLEMENTATION*

Examples of implementation subject area names:

- IMPLEMENTATION

Notes:

1. This guideline refers to a single implementation subject area that contains all implementation elements, including the specifications of consumed components. No guidelines are offered for the case where multiple implementation subject areas are defined.

Rationale:

- To make it obvious that this subject area contains the implementation-only elements of a component implementation design.

Internal Entity Type Name

Internal entity type names are pure text; they do not include any codes or prefixes or suffixes.

Where the internal entity type has a corresponding specification type, then it should have the same name as the specification type, without the *Ixxxv_* interface prefix or *Icccv* shared type prefix.

For example, the internal entity type, which corresponds to specification type IEMP1_EMPLOYEE, is named :

- EMPLOYEE

Notes:

1. Internal Entity types may be persistent or transient.

Rationale:

- Internal entity types can be distinguished from specification types, since they lack the interface prefix. It is easy to find their corresponding specification type.

Internal Action Block

Form the internal action block names as follows: *axxxvoow_opname*

where ...	is ...
a	Any character other than “I” or "C"
xxxvoow	The last seven characters of the operation prefix Ixxxvoow, from the name of the public operation which this internal operation implements Different leading characters may be used to indicate which part of the implementation. For example: M for mapper D for data store manipulation operation T for translator
opname	Succinct textual name for the operation. This may be the *typename_action* text of the operation being implemented.

Or where the internal action block is "common" to the implementations of several, many or all operation implementations, form the name as follows: *Ccccooov_name*

where ...	is ...
ccc	Component code (the code that appears within the component specification type name) In many cases, this may be the same as the code of the principal or only interface.
ooo	Three-character code for the action block, unique within the component implementation
v	Version number for the action block
name	A phrase summarizing the action performed by the action block, ideally in the form *typename_action*

Examples of internal action block names that implement the public operation IEMP1061_EMPLOYEE_PROMOTE_S are:

- MEMP1061_EMPLOYEE_PROMOTE (mapper operation)
- DEMP1061_EMPLOYEE_PROMOTE (persistent data manipulation operation)

Example of common action block names are:

CHRC9991_SETSERVID

CEMP0011_EMPLOYEE_VALIDATE_NAME

Notes:

1. Internal action blocks may be:
   - Internal operations, owned by internal entity types.
   - Free-standing action blocks, not owned by any type.
2. While it is valid practice for specification types to own internal operations, most organizations prefer not to do so, since it means that these operations are displayed in the interface type model and specification diagrams, where they are inappropriate.
3. It is common practice to build a "mapper" block, which is directly called by the public operation's action diagram. The mapper block performs the main control logic of the operation implementation, and "maps" specification types to internal entity types (and vice versa). It is suggested that the first character of the mapper name is an “M.” It is a free-standing operation, since internal operations are not recommended for interface type model types (the interface or its specification types).
4. Some organizations may prefer to use the *Cccc000v_name* convention for all internal action blocks, even for those that are not common.

Rationale:

- Action blocks within an implementation model need a unique name.
- It is helpful for the action block name to indicate which operation it implements.
- It is useful to have a systematic way of naming all the internal parts of a component implementation, so you immediately know the role of each action block.
- **CS*/3.0*** does not seek to constrain the way a component is implemented. This is simply a suggestion.
- As for public operations, the default source and object module name is automatically created from the first eight characters of the action diagram name. It has been found helpful if this name is unique and conveys some meaning to the developer.

Operations Library

Form operations library names as follows: *Lcccvlll*

where...	is...
ccc	The component code, as used in the component specification type name
v	The library version number, if concurrent versions of the same library are needed
lll	A code which indicates the contents of the library. For example: ALL indicates all operations of the component are included in the library xxx indicates the library contains all operations of the interface that has code xxx oow indicates the library contains a single operations, which has the code oow.

Examples of operations library names are:

- LHRC1ALL
- LHRC2ALL
- LHRC1EMP
- LHRC1011

Notes:

1. *lll* is a user-defined code; the suggestions given above will not be suitable in all situations.

Rationale:

- To provide a unique name for the operations library.
- To enable staff to determine which component this library belongs to.
- And, if possible, to give some indication of its contents, and whether the component code appears in multiple libraries.
- To allow for several concurrent versions of the library. The library version number may or may not coincide with the component version number.
- For OS/390 (MVS), the operations library is a "NCAL" module and contains a single operation.

Cascade Library

Form the cascades library names as follows: *Ccccvvrr*

where...	is...
ccc	The component code, as used in the component specification type name
vv	Version number of the component implementation
rr	Revision Number of the component implementation

Examples of specification type names:

- CHRC0100
- CPRM0213

Notes:

1. There is one cascade library per model, and hence one cascade library per component implementation.

Rationale:

- The component implementation release, to which the library applies, is readily identified.

Appendix C: Standard Parameters

This appendix describes a set of attributes, intended for use as standardized parameters of public operations. That is, they would appear in the import and export views of public operations.

Some of these parameters are a ***CS/3.0*** requirement: they must appear in every public operation's exports. These were introduced in Chapter 5, "Interface," but are repeated below.

The other parameters are recommendations. Where a customer acquires components from several sources, it is simpler if all those components have used common parameter names and formats for this generic information.

All the attributes shown in Figure C.1, the Chart of Standard Parameters, may be attributes of any work set or specification type.

NOTE: We recommend that a development organization standardize on the location of the standard parameters.

Recommended Practice

The standard parameters are defined in their own specification type, or as attributes of the interface type itself. Furthermore, the export view of this type should:

- Be the last export view of the public operation
- Include at least the mandatory standard parameters, in this order:

  SEVERITY_CODE
  ROLLBACK_INDICATOR
  ORIGIN_SERVID
  CONTEXT_STRING
  RETURN_CODE
  REASON_CODE

  You may use DATA_STORE_STATUS as an alternative to the required standard parameter ROLLBACK_INDICATOR

  The meaning of the four data_store_status codes is clearer than the meaning of rollback_indicator. Component builders are encouraged to replace rollback_indicator with data_store_status_code. It is permissible for operations to export both rollback_indicator and data_store_status_code.

If CHECKSUM is exported by an operation, it must be the last attribute of the last export view of the operation.

Chart of Standard Parameters

We recommend that all views of the type defining the standard parameters present their attributes in the same order as shown in the chart below, although a view need not contain every attribute.

Figure C.1 Chart of Standard Parameters

Name	Format	Purpose	Role
ACTION	Text 32	The name of an operation sub-action. A public operation might be built to perform a number of alternative sub-actions; in such a case, this attribute appears in the operation's imports, to define which sub-action is required.	(Oi)
COID	variable	The identifier of a component object. Used as an import attribute to indicate which component object the operation is to run against. Only applies to interfaces that have been designed to support multiple component objects. Component object identifiers are explained in Chapter 10, "Identifiers and Relationships."	(Oi)
RELEASE_INFORMATION	Text 20	Describes the release of a public operation. Can be used as an import attribute, to request that a particular release is executed, or as an export to indicate which release of a public operation has actually executed. Larger than **CS/*3.0***'s version/revision number, to allow component provisioners to utilize more extensive "release information."	(Oi) (OX)
DIALECT_CODE	Text 2	A code indicating a human language or dialect. Can be used as an import attribute, to request that exports are returned in a particular dialect, or as an export attribute, to indicate which dialect has been chosen for the exports.	(Oi) (OX)
SEVERITY_CODE	Text 1	A code indicating the severity of exception described by the return/reason code combination. Permitted values are: "I" = informational "W" = warning "E" = error.	(MX)
ROLLBACK_INDICATOR	Text 1	A code returned by a public operation, which requests the consumer to roll back any updates that the public operation has made.	(MX)

Name	Format	Purpose	Role
DATA_STORE_STATUS	Text 1	A code indicating the status of persistent storage after an operation execution. "1" = data unchanged "2" = changes rolled back "3" = data changed "4" = data integrity compromised. See (DSS) below for further explanation.	(OX)
ORIGIN_SERVID	Numeric 15	The server identifier of the component that originated the return/reason code combination. Each installed copy of a component (also known as *a component server*) is expected to identify itself using a unique value—its "served." See Chapter 10 for more information.	(MX)
CONTEXT_STRING	Varying Length Text 512	Intended for additional data about a failure encountered in an operation execution, which can then be incorporated into end-user messages.	(RX)
RETURN_CODE	Numeric 5	A standardized code (from Appendix D, "Return Codes"), indicating the type of failure or success encountered during operation execution.	(MX)
REASON_CODE	Numeric 5	A code which amplifies the return_code.	(MX)
CHECKSUM	Text 15	To enable view compatibility checking.	(RX)

Role Abbreviations

(MX) = must be exported by every public operation.

(RX) = recommend that this is exported by every public operation, even where the component provisioner chooses not to populate the attribute view.

(OX) = this attribute may appear in the export views for those operations that require it.

(Oi) = this attribute may appear in the import views for those operations that require it.

(DSS) The values of the DATA_STORE_STATUS have the following meanings in the exports of a public operation. Unless mentioned otherwise, the codes apply to both transactional and sub-transactional operations.

"1" = Data Unchanged. The execution of this operation, or any operation it uses, has left all stored data unchanged.

"2" = Changes Rolled Back. The execution of this operation has left all stored data unchanged, since all changes have been rolled back (only applies to transactional operations).

"3" = Data Changed. The execution of this operation, or any operation it uses, has changed stored data and data integrity is guaranteed.

"4" = Data Integrity Compromised. The execution of this operation, or any operation it uses, has left data changed but was not able to maintain data integrity. The consumer of this operation should initiate a rollback or employ some other means to restore the integrity of the data store (only applies to sub-transactional operations).

Appendix D: Return Codes

These codes were developed by Castek Software Factory, Inc.

Asterisks (*) indicate variations from the original Castek's specification.

Code	Exception	Description
22	Retrieve Action Returned Obsolete Data	Data has been returned for the instance that matches the imported identifier (usually an instance identifier), but the instance has been logically deleted, or has an expiration date that makes it "invalid" for the context of the consumer.
20	Delete Action Resulted in Logical Delete	The operation was expected to delete the instance that matches the imported identifier. Due to business usage or referential integrity rules, the instance has been logically deleted rather than physically deleted. (Archiving or physical delete is possible at some later date.) The component may or may not respond with this instance when queried later using the same identifier. See exception #0022 for further details. Compare this with exceptions # -0042 and # -0043.
10	Optional Field Missing; Default Used	One or more optional import field values were not populated. Default values were provided by the component.
1	Successful Completion	The operation completed successfully.
0000	**Not a valid return code*	*This code was formerly used to mean both "no pre-conditions met" and "successful completion".*
-1	Operation not Available	The implemented operation has not been linked in. The operation stub has executed instead.
-2	*No pre-conditions met	The operation did not meet any pre-condition, so no post-conditions are guaranteed.
-10	Identifier not Found	The operation has attempted to find the instance that matches the imported identifier. The instance was not found.
-11	Identifier Missing	An identifier was missing from the import view.
-12	Identifier Validation Failed	An imported identifier was found to have an invalid format. For example, an alphabetic character appeared in a numeric instance identifier field.
-20	Mandatory Import Missing	A mandatory import field was not populated.
-21	Mandatory Import Validation Failed	A mandatory imported attribute was found to have an invalid format. For example, an alphabetic character appeared in a numeric instance identifier field.
Code	**Exception**	**Description**
-30	Optional Import Validation Failed	An optional imported attribute was found to have an invalid format. For example, an alphabetic character appeared in a numeric instance identifier field.

Code	Exception	Description
-40	Create Action Failed	A failure occurred during a CREATE action. Although the import data validated successfully, some subsequent problem occurred that prevented the create from occurring. The reason code provides additional details.
-41	Update Action Failed	A failure occurred during an UPDATE action. Although the import data validated successfully, and a valid instance was located, a subsequent problem occurred that prevented the update from occurring. The reason code provides additional details.
-42	Delete Action Failed	A failure occurred during a DELETE action. Although the import data validated successfully and a valid instance was located, a subsequent problem occurred that prevented the delete from occurring. The reason code provides additional details. This exception differs from # -0043 in that the processing did not fail because of referential integrity issues.
-43	Deletion Inhibited	A failure occurred during a DELETE action due to a delete restrict or referential integrity issue, or due to a usage or business rule (in the invoked component or a subsequently invoked component). The import data validated successfully, and a valid instance was located. Unlike exception # -0042, the delete process did not fail due to problems implementing the delete action itself. The reason code provides additional details. This exception is distinct from # 0020 and # -0042.
-44	Associate Action Failed	A failure occurred during an ASSOCIATE action. Although the import data validated successfully, and valid instances were located, a subsequent problem occurred that prevented the associate from occurring. The reason code provides additional details.
-45	*Disassociate Action Failed	A failure occurred during a DISASSOCIATE action. Although the import data validated successfully and valid instances were located, a subsequent problem occurred that prevented the disassociate from occurring. The reason code provides additional details.
-46	*Transfer Action Failed	A failure occurred during a TRANSFER action. Although the import data validated successfully and valid instances were located, a subsequent problem occurred that prevented the transfer from occurring. The reason code provides additional details.
Code	**Exception**	**Description**
-50	Date Format Error	The imports, or the data store, have supplied a date in an invalid format. The operation has stopped, pending the user correcting the field in error. This exception is often exported by validation operations running on a client workstation.

Code	Exception	Description
-51	Numeric Operation Failed	A numeric exception has occurred. The reason code provides additional details, which will help to identify the data items in error.
-55	*View Matching Failure	A failure occurred when importing data into or exporting data from the operation.
-60	Persistent Storage Failure	An error condition has been returned by the data store used to provide persistency of user data. This error cannot be mapped to one of the above exceptions, so is returned in this format. The reason code will indicate the reason for the failure. The reason code probably cannot contain enough information for problem correction, so we recommend exporting a Context_String containing additional detail.
-61	Operating Environment Failure	A serious error has arisen in the operating environment, causing the operation to stop. This error cannot be mapped to one of the above exceptions, so is returned in this format. The reason code will indicate the reason for the failure. The reason code probably cannot contain enough information for problem correction, so we recommend returning a Context_String containing additional detail.
-999	Unexpected Exception	An unexpected or uncontrollable exception has occurred. This error cannot be mapped to one of the above exceptions, nor can the component be safely expected to perform further operations. We recommend that the component attempt to provide additional error information in a Context_String to assist development and support staff in tracing the problem. However, component users cannot expect consistent results from this component and should halt application execution.

The range -1999 to -1000 is reserved for user-defined exceptions.

The range 1000 to 1999 is reserved for user-defined successful outcomes.

These user-defined return code ranges are intended for customers wishing to use additional codes in their internally developed and consumed components. Component provisioners building components for commercial sale are requested to use only the standard return codes listed above.

Glossary of Terms

Underlined words have their own glossary entry.

application

The software which a user runs, that (a) provides a useful set of services to that user or their organization and (b) is perceived by the user to be a single system.

black-box component

A component delivered in the form of a component specification plus the component executable. The customer cannot see the component implementation, hence the term "black-box". Compare with white-box component.

business object

An item that needs tracking by a business. It is usually assigned a unique identifier, and data is kept about the item. The item may be tangible, for example an employee or truck, or something less tangible, such as an accident or a campaign or an account.

The term business object is also used to mean a software object that corresponds to a real-world business object.

business (object) type

A type whose instances are business objects.

CBD

The common abbreviation for component-based development.

CS/*3.0* component

A component that conforms to the Standard defined in this publication.

component

An independently deployable software collection, which has the following characteristics:

- It is a software building block, used to build applications or larger components.
- It is encapsulated.

- It offers its functionality through stable, well-defined interfaces.
- It is replaceable by other components that offer (at least) the same set of interfaces.
- Although independently deployable, it may have usage dependencies upon other components.
- It is delivered in the white-box or black-box style.

component-based development (CBD)

The process of building applications by combining and integrating pre-engineered, pre-tested components.

component documentation

The component specification, and any other essential information that a provisioner must supply with a delivered component for it to be understood and correctly deployed.

component executable

A set of component modules which, together, perform all the operations of a component according to its component specification. These may be load modules (executable files) and/or object modules, and modules containing the database definition statements that define the persistent storage for the component.

component implementation

A particular internal design of a component that achieves the component specification. This consists of the source code (action diagram statements) and possibly a database design.

component implementation model

A COOL:Gen model containing the implementation (and implicitly, the specification) of a component.

component module

A physical file which constitutes a part of (or possibly all) the software of a delivered component.

component object

A run-time instance of a component.

component release

The publication of a new component that is intended as an upgrade to a previously published component. The release number is comprised of version and revision numbers.

component server

A synonym for installed component.

component specification

A definition of the behavior of a component, without unnecessarily pre-empting how it is realized. It may include dependencies on other components or interfaces. Any component implementation or component executable must conform to a component specification.

component specification model

A COOL:Gen model containing a component specification and, possibly, the execution parameters for a particular component executable.

component specification storage model

A COOL:Gen model containing many component specifications. Component specifications are migrated from this model into consumers' (application or component) implementation models. Developers may browse this model, to study component specifications.

COOL:Gen's Component Manager tool makes such a storage model unnecessary.

constraint

A condition that must always be true. A component specification contains many constraints. An invariant is a particular kind of constraint.

consumer

The invoker of an operation of a component. The invoker may be software, for example, an operation of another component, or a human user (for UI-bearing operations).

This is not the same thing as a customer. The term consumer is often called a client in other methods and standards.

customer

The organization that takes delivery of a component and uses it to build applications or other components.

dependency

See usage dependency.

encapsulation

The notion that a component's implementation details are inaccessible to its consumers. Consumers can only access the component's functionality and data through its (programmable) interfaces. The consumer only codes to the interface specification, and must not write code that uses a knowledge of the component implementation.

Encapsulation allows the implementation of a component to be modified without affecting the consumer of the component or the implementations of other components.

Where a component implementation uses a data store to provide persistence, it must not share this data store with other components, since this breaks encapsulation.

executable

See component executable. In **CS/*3.0***, this term does **not** mean a single .EXE file.

execution parameters

The values that need to be known to link-in and invoke an operation of a component, that may vary for each component executable.

extension

Changing a specification or a design by adding to it, and not removing or contradicting anything that existed before. In **CS/*3.0***, we are mainly concerned with specification extension.

factoring

The operations of an interface (type) may be classified by stating which interface type model type they primarily act upon. Factoring was included in **CBD*96***, but has been dropped from **CS/*3.0***.

implementation

The realization of a specified component using the syntactical constructs of some programming language or other tool. This is also termed *implementation design* or *technical design*.

implementation inheritance

An object-oriented programming technique for reusing an existing class. A sub-class inherits all features of an existing super-class.

Implementation Model diagramming tool

A COOL:Gen tool that enables the implementation-only elements of a component implementation to be created, modified and visualized.

inheritance

A technique for extending a specification or implementation. This term may refer to specification inheritance or implementation inheritance, so should be qualified.

installed component

A specific installed copy of a component executable. Each of its run-time component modules must be installed on a node of a computing network, and be registered with the run-time environment. Also known as a component server.

instance

An object that conforms to a type.

instance identifier

In **CS/*3.0*,** a standard format, immutable identifier for an instance of a specification type.

interface (type)

A type that defines a collection of semantically related operations.

A component may support one or more interfaces. The same interface could be offered by several components.

interface type model

The attributes of an interface, organized into a type-relationship model. This defines the values that an interface must be able to recall, in order to realize its operations.

The model always includes the interface type itself, plus one or more specification types, and one or more invariants.

Interface Type Model diagramming tool

A COOL:Gen tool that enables an interface type model to be developed and visualized.

interface type model type

A type that has been included in an interface type model. This may be a specification type or interface type.

internal operation

An operation, that is used within the implementation of a component, that is not exposed by the interfaces of that component.

internal type

An entity type (or class) used inside a component implementation, which is not included in the interface type model, nor exposed as a parameter. Formerly called an implementation type or an implementation-only type.

invariant

A constraint that applies to type models.

An invariant is a condition that must be true before and after any public operation execution, although it may be temporarily untrue during execution. The condition is expressed in terms of interface type model types, their attributes and their relationships.

Invariants are an essential part of the interface type model.

In COOL:Gen, most invariants are recorded as properties of attributes and relationships (for example, relationship cardinality, attribute optionality, and attribute uniqueness). Other constraints have to be recorded textually in the interface type's description panel.

object

Something that is identifiable, exhibits behavior, and has state, where:

- "Identifiable," means it can be distinguished from other objects. For example, it has a unique identifier.
- "Exhibits behavior," means it offers operations, and may invoke the operations of other objects.
- "Has state," means it contains/remembers information about itself.

object type

A definition of object behavior. Many objects may conform to the same object type.

A component specification is a kind of object type.

A run-time component is a type of object, so is called a component object.

operation

A discrete unit of functionality provided by an interface, class or entity type. In **CS/*3.0***, we use the term public operation for an operation of an interface offered by a component.

operation implementation

A mechanism that achieves the effect defined in the operation specification. In COOL:Gen, the mechanism is usually written in action diagram syntax, although external action blocks may also be used.

operation specification

The consumer's view of an operation. The specification consists of the operation name, signature, purpose, pre- and post-condition pairs, and all possible return codes.

operations library

A file containing one or more operation implementations, any of which can be called into memory at run-time, as needed by some larger program.

This run-time calling action is usually termed *dynamic linking*. Contrast this with *static linking*, in which the operations are linked into a load module or executable file at "compile-time."

parameter

An attribute or entity or group view, appearing in the imports or exports of an operation.

persistent/persistence

A quality of an object which means that its state (data values) is remembered outside the scope of an executing machine process.

The interface type model expresses the possible states of the interface.

post-condition

Part of an operation specification. A set of assertions that will be true after the operation has executed, providing that its corresponding pre-condition was true prior to execution.

pre-condition

Part of an operation specification. A condition that must be true prior to operation execution in order for its corresponding post-condition to hold. A false pre-condition does **not** imply the operation does not execute; it means that the corresponding post-condition is not guaranteed to be true.

provisioner

An organization that builds components and delivers them to customers.

public operation

A discrete unit of functionality defined on an interface. A public operation should be a success unit, but need not be a database commit unit.

reason code

A value which amplifies why a particular return code has been output by an operation.

refactor

The action of restructuring a design, so that it produces the same results, but is now improved in some way. For example, it may be more maintainable, or perform better.

release

A new offering of a component made available to a customer by a provisioner. It also applies to new offerings of interfaces or operations that may occur within a new release of a component.

In **CS/*3.0***, we differentiate version releases and revision releases.

replaceable

An important characteristic of components. One component can replace another, so long as it supports at least the same set of interfaces, and respects the same dependencies.

return code

A value indicating the final state of an public operation execution. That is, a success state or a reason for failure. **CS/*3.0*** provides a standardized set of return codes.

revision

A new release of a component, interface or operation that does not impact existing consumers. A specification compatible release.

signature

The operation name, plus the complete set of import and export views, for a given public operation.

specification compatible

The distinguishing quality of a new component release, which requires that its component specification is a specification extension of the previous release, so current consumers are unaffected by the change.

specification extension

A technique for defining a new component specification by adding to an existing component specification without removing anything from it.

Specification inheritance is an acceptable extension technique.

Other specification extension techniques include adding new operations and optional interface type model elements (types, attributes, relationships) to an existing component specification, and weakening its invariants.

specification inheritance

A specification technique that extends an existing type (interface or specification type) by adding sub-types. The new sub-type inherits all the features (operations, attributes, relationships, invariants) of its super-type.

Specification Model diagramming tool

A COOL:Gen tool that enables the elements of a component specification to be created, modified and visualized.

specification type

Specification types are used to define parameters and interface type model types. A specification type has attributes and relationships, but no operations.

storage aware

A quality of a particular release of a component indicating that the release understands the storage format of the previous release.

A storage-aware release can at least read all the information stored by the previous release. It does not imply that the new release stores data in the same format, or even a compatible extension of that format, although this will often be the case.

storage compatible

A quality of a particular release of a component indicating that the release uses the same storage format, or an extension to the storage format, used by the previous release.

A storage compatible release is always a storage aware release, but not the other way around.

sub-transaction, or sub-transactional operation

An operation that is not a database commit unit and will be rolled-back if its enveloping transaction fails.

success unit

A process which, when complete, does not leave any invariant or database integrity rule violated. All public operations should be success units.

test harness

A collection of programs that allow a customer to validate the services of a component. The programs could be built as GUI, web or block-mode transactions, or batch jobs.

transaction, or transactional operation

An operation that forms a database commit unit. This means that once the operation has completed, the database updates cannot be undone. Prior to completion, it is possible to "rollback" the database updates if a failure or deadlock occurs.

In **CS/*3.0***, a transactional operation may offer user-interaction support; this is known as a *user interface-bearing operation*.

Regular transactional operations are supported by COOL:Gen's non-display procedure steps, while user interface-bearing operations are supported by display procedure steps.

transient type

A type that does not (directly) provide persistence for its occurrences. In COOL:Gen terms, this is an entity type or specification type which is not "transformed" into the table of a database.

Normally, all the types contained in a specification subject area are transient. In an implementation subject area, there will usually be some types which are persistent – they have a corresponding database table.

type

A construct that defines behavior and/or data structure.

usage dependency

A relationship between two components or a component and an interface, in which one component requires the presence of the other for its correct functioning or implementation.

Usually, this means that the dependent component invokes some of the operations offered by the other component. In COOL:Gen, invocation is by USE statement or dialog flow.

A component is said to be *independent* if it does not depend on any other component.

A usage dependency may be stated in the component specification, so all component implementations have this dependency, or may be a design choice within a particular implementation.

version

A new release of a component, interface, or operation that may not specification-compatible with the previous release. That is, some consuming software may need to be changed if it is to continue to work with the new release.

white-box component

A component delivered as a component implementation model. The customer can see how the component works, hence it is a "white-box" as compared to a black-box component.

Index