# Simple Bookstore Model Translation

---

## A Translation of the Case Study

**REVISION HISTORY**

| NUMBER | DATE | DESCRIPTION | NAME |
|---|---|---|---|
| 0.1 | April 13, 2016 | Initial coding started. | GAM |
| 1.0a1 | May 21, 2016 | First release built with pre-released version of micca. | GAM |
| 1.0a2 | May 25, 2016 | Converted some class operations to use the embedded commands "foreachWhere" and "foreachRelatedWhere". Corrected typos found by TBM. Still built with pre-release version of micca. | GAM |
| 1.0a3 | June 14, 2016 | Updated translation to reflect the latest way that micca deals with conditional associations and associative classes. Still built with pre-release version of micca. | GAM |

# Contents

**Introduction** 1
How to Read This Document . . . 1

**Model Details** 1

**Translation Conventions** 2

**Data Types** 3

**Product Specification Subsystem** 4
Product Class . . . 4
Publisher Class . . . 5
Author Class . . . 5
Authorship Class . . . 5
Authorship Sequence Class . . . 5
Book Product Class . . . 6
Recording Product Class . . . 6
Software Product Class . . . 6
Special Order Product Class . . . 6
Stocked Product Class . . . 6
Product Category Class . . . 7
Category Group Class . . . 7
Recording Class . . . 7
Artist Class . . . 7
Performance Credit Class . . . 7
Performance Credit Sequence Class . . . 7
Recording Format Class . . . 8
Platform Class . . . 8
Computer Software Class . . . 8

**Ordering Subsystem** 8
Customer Class . . . 8
Order Class . . . 9
Order Class State Model . . . 9
Order Placement Class . . . 12
Product Selection Class . . . 12
Product Selection Class State Model . . . 12
ShoppingCart Class . . . 13
Credit Card Charge Class . . . 15
Credit Card Charge Class State Model . . . 15
Order Payment Class . . . 16

**Shipping Subsystem** **17**

Shipment Class . . . 17

Shipment Class State Model . . . 17

Delivery Class . . . 19

Past Packing Class . . . 20

Packing Class . . . 20

Shipping Company Class . . . 20

Warehouse Class . . . 20

Warehouse Class State Model . . . 21

Warehouse Clerk Class . . . 22

Shipping Clerk Class . . . 22

Shipping Clerk Class State Model . . . 23

Stock Clerk Class . . . 25

Off Duty Clerk Class . . . 25

Shipment Item Class . . . 25

**Product Specification Subsystem Population** **25**

Product Population . . . 26

Publisher Population . . . 27

Author Population . . . 27

Authorship Population . . . 28

Book Product Population . . . 28

Recording Product Population . . . 29

Software Product Population . . . 29

Special Order Product Population . . . 29

Stocked Product Population . . . 29

Product Category Population . . . 30

Category Group Population . . . 30

Recording Population . . . 30

Artist Population . . . 30

Performance Credit Population . . . 30

Performance Credit Sequence Population . . . 31

Recording Format Population . . . 31

Platform Population . . . 31

Computer Software Population . . . 31

**Ordering Subsystem Population** **31**

**Shipping Subsystem Population** **32**
Shipment Population . . . 32
Delivery Population . . . 32
Past Packing Population . . . 32
Packing Population . . . 32
Shipping Company Population . . . 33
Warehouse Population . . . 33
Warehouse Clerk Population . . . 33
Shipping Clerk Population . . . 34
Stock Clerk Population . . . 34
Off Duty Clerk Population . . . 34
Shipment Item Population . . . 34

**External Entities** **34**

**Execution Scenarios** **37**
Scenario 1 . . . 37
Scenario 2 . . . 39
Scenario 3 . . . 40
Scenario 4 . . . 40
Run Results . . . 41

**Code Organization** **42**
Source Code . . . 43
Initial Instance Population . . . 44

**Literate Programming** **45**

**Index** **46**

# Introduction

This document presents a translation of the case study model from the book, *Executable UML: A Foundation for Model-Driven Architecture* by Stephen J. Mellor and Marc J. Balcer. The translation is targeted at the "C" implementation language using the `micca` translation scheme. The model can be found in Appendix B in the book on page 313.

This document is also a literate program. It contains all the code of the translation and the execution scenarios that exercise the program. The literate program source and all the output is available at the Model Realization web site.

Translation using `micca` involves a manual component. The model diagrams are transcribed into a domain specific language (DSL) that is provided by `micca`. The classes, relationships and state models are defined in a declarative manner. The state activities must be translated from action language into "C" using embedded macros provided by `micca` to implement the model level actions. For example, action language statements to select related instances are translated into invocations of the `%<instance ref foreachRelated self ...>%` macro. Generally, action language statements translate one-for-one into lines of embedded macro code and the correspondence is direct if you understand the conventions of "C" programming and the embedded macro language. Once transcribed into the `micca` DSL, a code generation phase is invoked. This code generation produces two files, a "C" header file and a "C" code file. Once generation is completed, the application can be compiled and linked and we can exercise the resulting program by signaling events or invoking any operations that are provided.

The target of a `micca` translation is small embedded micro-computer based systems. `Micca` can produce code that runs in a POSIX environment and that is what has been used for this example. For an online bookstore, the `micca` target platform is clearly not appropriate. Real world online stores use database management systems and much more sophisticated data management facilities. So the purpose here is not to produce a program that could be an online bookstore application. Rather we intend to show that executable models can be translated to any platform, even if that platform does not represent implementation technology which is appropriate for the computational demands of the application. However, we will show that all the logic of the model is implemented and we produce a running program which, in a limited context, behaves as an online bookstore in exactly the manner shown by the executable model.

In the next sections, we explain some preliminary details that must precede the translation. Some of these details have to do with the on-line bookstore model itself and some deal with the specifics of translating using `micca`. After the preliminaries, we present the translation organized into subsystems and presented in the order found in the book. Next we show a sample population of the model and that is followed by procedures that stub out the explicit external entity functions found in the model. Finally, we present a set of scenarios to exercise the model. `Micca` has many capabilties to deal with execution traces and we show a textual log of the event dispatch.

## How to Read This Document

This document presents the translation of the online bookstore case study in the same order as the model is shown in the appendix of the book. This is intended to facilitate easier reference between the text and the translation and demonstrate how the model directly drives the implementation as it is transcribed into `micca` configuration commands. However, if you are more interested in the result and what happens when the translation is run, then you should skip to the Execution Scenarios section. There you can see the result of running a set of scenarios on the translated model.

# Model Details

My experience is that until they are translated and made into running programs, all models contain small errors and inconsistencies. This is not an reflection on the authors of the model or of the quality of their work. Because translatable models must be so detailed, it is inevitable that inconsistencies and errors arise. After all, no one, no matter how carefully coded, expects a program to compile the first time without errors. On the rare occasions it happens, it is a cause for celebration! The same situation applies here. We will endeavor to point out the errors and state clearly how we deal with inconsistences. Since the publishing of the book, a set of errata have been published on the book's website. We have taken those errata into account in this translation.

We must also remember that this domain is but one in what would be a larger system to handle online ordering. See Figure 17.1 on page 276 of the book to get an indication of what that larger system might look like. Consequently, we will not deal with any parts of the translation that involve joining multiple domains. This is a bit disappointing as this part of assembling a system can be quite complex and is required for all but the tiniest of systems. The `micca` package and its underlying bridging facilities are

well suited to many of the problems one faces during the construction of bridge operations. It would be an interesting undertaking to construct another domain for the Online Bookstore application and show how bridging in the `micca` implementation could be accomplished, but we do not undertake that in this translation.

It should also be noted that no domain level operations are defined for the online bookstore domain by the model. As noted below, we have added a domain operation for changing orders. The processing for this activity is complicated enough that it can't be done via bridge and such a function is not an unreasonable domain operation to expect.

The result of this translation is the code for the domain itself and a set of scenarios that exercise the domain. You can think of this translation as providing the code for the domain and a preliminary set of unit tests in the form of execution scenarios. When I undertake system development using translation, this is usually the first step in the process of building the system. It is quite useful to be able to run a domain in isolation and construct a set of unit tests to exercise it. But it is only a first step. When building a system of multiple domains, integration of the domains is also required. I usually build a set of running programs integrating one additional domain until the system is fully built. The order of integration follows the usual mantra of:

> Analyze from the top down and build from the bottom up.

A better way of stating this is to analyze the most application specific portions first, but build the system infrastructure first. This is because it is usually easier to build code to drive a partial system than it is to build stubs intelligent enough to simulate missing system components.

These intermediary applications are useful for regression testing and for dealing with the inevitable changes that happen to domain models. Often the tests developed for the single domain unit tests are useful in integrating domains. It is important to remember that building is not *one big step* but rather one of building many integration and test applications, the final one of which contains everything and is the one delivered. This is a rather long explanation of the fact that what you will find in this document is only the first part of what it would take to build a real system from executable models.

## Translation Conventions

During the evolution of Executable UML, the authors of *Executable UML* decided that referential attributes and identifying constraints were optional. In my opinion, this decision was a regression of the method. It certainly can reduce the clutter of the class diagram graphic and reduce the need for invented identifiers, but it makes it much easier to skip over the details of how relationships are actually formalized. Also, combining referential attributes makes constraints on the data model and does so in a declarative[1] way.

Regardless of your view of referential attributes, a translation using `micca` discards them anyway. `Micca` adopts the usual translation convention when static languages are targeted, namely to use the address of the class instance as an architectural identifier and as the basis for navigating relationships. So, attributes that only serve the role of identifiers or referential attributes are discarded from the model.

`Micca` does support the notion of derived attributes. We should note that the model as presented in the appendix does not state what the algorithms for the derived attributes are, but we can make reasonable guesses from the context and will point them out as we encounter them.

Creating instances is also slightly different in `micca`. Action language allows one to create a class instance without supplying all the attributes a value at the time the creation operation is invoked. `Micca` insists that all attributes have some legal value for their data type at the time an instance is created. This will involve some shuffling around of actions that perform instance creation. In particular, this model creates a number of class instances asynchronously. Often the model will pass along as event parameters the values of attributes which are then set in the state activity first executed when the creation event is delivered. It is a simple matter to remove those values as event parameters and supply them as attribute values at the signaling of the creation event. Since `micca` supports the notion of a default value, we will also use that feature to reduce some of the tedium of supplying attribute values at instance creation time.

The matter of allowing instances to be created without setting all the attributes to a value is a bit more troubling in general. Since most translation schemes are targeted at statically compiled languages with the implicit assumption that class instances are held in memory, the idea of creating a class instance usually maps to allocating a piece of memory of the appropriate size. In this regime, assigning attributes piecemeal is easily done. However, for software architectures targeted at databases which

[1] as opposed to a procedural way

have strong relational underpinnings, piecemeal setting of attribute values is not allowed[2]. The danger of piecemeal attribute assignment during creation is that the possibility exists that an attribute might not be given an intial value during certain execution paths through a state activity. For in-memory data structures, this means an attribute might have an arbitrary value or the value represented by all bits of the memory being zero. Best practice is to have the model set the values of all attributes at instance creation time.

`Micca` also takes a different approach to relationships. It does not allow simple associations that are conditional on the referenced side, *i.e.* association of the type `0..1--0..1` or `0..*--0..1` must be realized with a class based association. This restriction avoids the use of `NULL` as a meaningful value for a referential pointer, means that instance deletion can properly mirror its counterpart in a relational context and makes the calculation of referential integrity easier and more robust. Consequently, we have had to add a number of associator classes to realize associations that, in the model, were specified as conditional on the referenced side. We will point these out as they come up.

To make the correspondence to the text of the book explicit, we will order the presentation in the same order as the book. The domain is presented as three subsystems. Within each subsystem, the class definitions appear in the same order as the book. Relationship definitions are dealt with differently. In the book text, each *side* of the relationship is presented with the class that is a participant in the relationship and the relationship is stated from point of view of the participating class. For `micca`, we must define the relationships exactly once. So we give relationship definitions near the class that contains the referential attributes that formalize the relationship. The exception is generalization relationships. We define the generalization along with the superclass. This ordering is used to clarify the translation's correspondence with the book text. `Micca` allows an arbitrary order for these definitions.

Readers will probably find it essential to refer to the text of the book as you read the translation. It's impractical to print all the text and graphics of the model in this document if only because it would be a violation of copyright law. However, we will include small portions of the model here when it serves to clarify the translation. Consequently, you will find much of the translation presented with little additional text. We will comment when clarity is required or when we are doing something `micca` specific, but little else is really required. Reading the translation and the text of the book side-by-side is sufficient to show how the translation was accomplished.

## Data Types

`Micca` requires attributes to be given a data type. That data type is, of course, an implementation data type and must be a data type defined in "C".

Comments on the mapping of domain data types to implementation data types are given below.

**arbitrary_id**
: Arbitrary id's do occur in the model and we hold them as `unsigned` quantities and provide a class operation to create them.

**Money**
: We'll hold money as an `unsigned` in units of cents.

**date**
: Since we are intending to run in a POSIX environment, we will use the usual time value for dates, the number of seconds since the epoch. The model does not appear to perform any date computations.

**MailingAddress**
: Real world addresses are painful. We punt here and just hold strings. Another option would be to have a tuple valued attributes that could separate out the various component parts of the address, *e.g.* number, street, city, state, etc. Since state activities don't deal with the address as anything other than a unit, we'll stick with simple strings.

**WebsiteURL , InternetEmailAddress , TelephoneNumber**
: All these types could benefit from check expressions to validate their string format and content. We have not done that here and just use simple strings.

[2] Well databases can handle it with some awkward NULL value nonsense, but `micca` has no notion of a NULL (in the database sense, only in the "C" pointer sense) and none of the three values logic required to deal with NULL attribute values.

## Product Specification Subsystem

The Product Specification Subsytem consists of the following classes.

```
<<Product Specification Subsystem>>=
<<Product Class Definition>>
<<Publisher Class Definition>>
<<Author Class Definition>>
<<Authorship Class Definition>>
<<Authorship Sequence Class Definition>>
<<Book Product Class Definition>>
<<Recording Product Class Definition>>
<<Software Product Class Definition>>
<<Special Order Product Class Definition>>
<<Stocked Product Class Definition>>
<<Product Category Class Definition>>
<<Category Group Class Definition>>
<<Recording Class Definition>>
<<Artist Class Definition>>
<<Performance Credit Class Definition>>
<<Performance Credit Sequence Class Definition>>
<<Recording Format Class Definition>>
<<Platform Class Definition>>
<<Computer Software Class Definition>>
```

### Product Class

```
<<Product Class Definition>>=
class Product {
    attribute productID {char[32]}
    attribute productName {char[64]}
    attribute copyright {char[8]}
    attribute unitPrice unsigned
    attribute description {char[256]}
    attribute website {char[128]}
    attribute currentlyAvailable bool

    classop {struct Product *} findByProductID {productID {char const *}} {
        %<Product findWhere product {strcmp(productID, product->productID) == 0}>%
        return product ;
    } ; # ❶
}
```

❶ We will find it useful for a number of classes to include one or more class based operations to search for instances of the class. Typically the search is based on identifying attributes and so finds at most one instance. Here we are assuming the number of instances of `Product` is small and that a linear search is sufficient. It this is not the case, the this is also the place where you could add code to use a more sophisticated search algorithm.

```
<<Product Class Definition>>=
generalization R12 Product SpecialOrderProduct StockedProduct
generalization R11 Product BookProduct RecordingProduct SoftwareProduct
association R15 Product 0..*--1 ProductCategory
association R1 Product 0..*--1 Publisher
```

### Publisher Class

```
<<Publisher Class Definition>>=
class Publisher {
    attribute groupCode {char[8]}
    attribute publisherCode {char[16]}
    attribute name {char[64]}
    attribute address {char[128]}
    attribute website {char[128]}
}
```

### Author Class

```
<<Author Class Definition>>=
class Author {
    attribute name {char[64]}
    attribute website {char[256]}
    attribute email {char[64]}
}
```

### Authorship Class

Authorship is an association class. It is true that the instances of the association class correspond one-to-one to the instances of the association itself and that the number of instances is less than or equal to the product of the number of instances of the participating classes.

```
<<Authorship Class Definition>>=
class Authorship {
    attribute preCredit {char[32]} -default {""}
    attribute postCredit {char[32]} -default {""}
}
```

```
<<Authorship Class Definition>>=
association R2 -associator Authorship BookProduct 1..*--0..* Author
```

### Authorship Sequence Class

This class must be introduced to deal with relationship, `R3`, which is reflexive and conditional on both sides.

```
<<Authorship Sequence Class Definition>>=
class AuthorshipSequence {}
```

```
<<Authorship Sequence Class Definition>>=
association R3 -associator AuthorshipSequence Authorship 0..1--0..1 Authorship
```

To be absolutely pedantic the `R3`, relationship does not, in and of itself, guarantee a linear ordering. It requires that the initial instance population or the action code set things up correctly to achieve desired ordering. In practice, this is easily done and the number of classes required to guarantee an ordering by virtue of referential integrity constraints is usually not worth it.

### Book Product Class

```
<<Book Product Class Definition>>=
class BookProduct {
    attribute bookNumber {char[32]}
    attribute titleCode {char[32]}
    attribute title {char[128]}
    attribute subtitle {char[128]}
}
```

### Recording Product Class

In addition to being a subclass in a generalization, the `Recording Product` class is also an associative class. We follow the same pattern here as we established in the `Authorship` class translation.

```
<<Recording Product Class Definition>>=
class RecordingProduct {
    attribute runningTime unsigned
}
```

```
<<Recording Product Class Definition>>=
association R16 -associator RecordingProduct Recording 1..*--1..* RecordingFormat
```

### Software Product Class

```
<<Software Product Class Definition>>=
class SoftwareProduct {
    attribute productVersion {char[16]}
}
```

```
<<Software Product Class Definition>>=
association R19 SoftwareProduct 1..*--1 Platform
association R20 ComputerSoftware 1..*--1 SoftwareProduct
```

### Special Order Product Class

```
<<Special Order Product Class Definition>>=
class SpecialOrderProduct {
    attribute specialOrderInstructions {char[256]}
    attribute daysToDeliver unsigned
}
```

### Stocked Product Class

```
<<Stocked Product Class Definition>>=
class StockedProduct {
    attribute quantityOnHand unsigned
    attribute reorderThreshold unsigned
    attribute reorderIncrement unsigned
    attribute reorderInstructions {char[256]}
}
```

### Product Category Class

```
<<Product Category Class Definition>>=
class ProductCategory {
    attribute categoryName {char[32]}
}
```

### Category Group Class

This associative class is introduced to realize `R14`, which is conditional on the `ProductCategory` side.

```
<<Category Group Class Definition>>=
class CategoryGroup {}
```

```
<<Category Group Class Definition>>=
association R14 -associator CategoryGroup\
        ProductCategory 0..*--0..1 ProductCategory
```

### Recording Class

```
<<Recording Class Definition>>=
class Recording {
    attribute title {char[128]}
}
```

### Artist Class

```
<<Artist Class Definition>>=
class Artist {
    attribute artistName {char[128]}
}
```

### Performance Credit Class

```
<<Performance Credit Class Definition>>=
class PerformanceCredit {
    attribute role {char[32]} -default {""}
}
```

```
<<Performance Credit Class Definition>>=
association R17 -associator PerformanceCredit Recording 1..*--1..* Artist
```

### Performance Credit Sequence Class

This class is introduced as an associator for `R18`, which is reflexive and conditional on both sides.

```
<<Performance Credit Sequence Class Definition>>=
class PerformanceCreditSequence {}
```

```
<<Performance Credit Sequence Class Definition>>=
association R18 -associator PerformanceCreditSequence\
        PerformanceCredit 0..1--0..1 PerformanceCredit
```

### Recording Format Class

```
<<Recording Format Class Definition>>=
class RecordingFormat {
    attribute formatName {char[32]}
}
```

### Platform Class

```
<<Platform Class Definition>>=
class Platform {
    attribute platformName {char[32]}
}
```

### Computer Software Class

```
<<Computer Software Class Definition>>=
class ComputerSoftware {
    attribute title {char[128]}
}
```

## Ordering Subsystem

The Ordering Subsystem consist of the following classes.

```
<<Ordering Subsystem>>=
<<Customer Class Definition>>
<<Order Class Definition>>
<<Order Placement Class Definition>>
<<Product Selection Class Definition>>
<<Shopping Cart Class Definition>>
<<Credit Card Charge Class Definition>>
<<Order Payment Class Definition>>
```

### Customer Class

```
<<Customer Class Definition>>=
class Customer {
    attribute email {char[64]}
    attribute name {char[64]}
    attribute shippingAddress {char[64]}
    attribute phone {char[32]}
    attribute purchasesMade unsigned -default 0 ; # ❶

    classop {struct Customer *} findByEmail {email {char const *}} {
        %<Customer findWhere customer {strcmp(email, customer->email) == 0}>%
        return customer ;
    }
}
```

❶ When Customer instances are created, they have certainly not made any purchases so we can specify a default value to ease the burden of specifying attributes at creation time.

## Order Class

The `Order` class is the first we encounter that has a state model associated with it.

```
<<Order Class Definition>>=
class Order {
    attribute orderID unsigned
    attribute dateOrderPlaced DateType_t -default {0}
    attribute totalValue unsigned
    attribute recipient {char[128]}
    attribute deliveryAddress {char[128]}
    attribute contactPhone {char[32]}

    statemodel {
        <<Order State Model>>
    }

    classop unsigned genOrderID {} {
        static unsigned orderID = 100 ;

        return orderID ++ ;
    } ; # ❶
}
```

❶ The `orderID` is an arbitrary identifier and we will use class operations to generate a value for the attribute.

```
<<Order Class Definition>>=
association R10 Order 0..1--1 ShoppingCart
```

### Order Class State Model

```
<<Order State Model>>=
transition @ - checkOut -> EstablishingCustomerandVerifyingPayment ; # ❶
```

❶ Here we encounter our first creation event. In `micca` the `@` symbol is used to represent the pseudo-initial state out of which a creation event transitions.

```
<<Order State Model>>=
state EstablishingCustomerandVerifyingPayment {
        accountNumber {char const *}
        billingAddress {char const *}
        cardExpirationDate DateType_t
        cardholderName {char const *}
        customerEmail {char const *}} {
    %<Customer instref customer>%
    customer = %<Customer operation findByEmail email customerEmail>% ;
    if (customer == NULL) {
        %<Customer create customer\
            email customerEmail\
            name self->recipient\
            shippingAddress self->deliveryAddress\
            phone self->contactPhone>%
    } else {
        %<instance customer update\
            name self->recipient\
            shippingAddress self->deliveryAddress\
            phone self->contactPhone>%
```

```
    }

    %<OrderPlacement create op R5 {Order self Customer customer}>% // ❶
    time(&self->dateOrderPlaced) ;

    %<my signal submitCharge\
        accountNumber accountNumber\
        billingAddress billingAddress\
        cardExpirationDate cardExpirationDate\
        cardholderName cardholderName>%
}

transition EstablishingCustomerandVerifyingPayment - submitCharge ->\
    SubmittingCharge
```

❶ Creating the instance of `OrderPlacement` creates an instance of `R5` and relates the `Order` and `Customer` instances together.

```
<<Order State Model>>=
state SubmittingCharge {
        accountNumber {char const *}
        billingAddress {char const *}
        cardExpirationDate DateType_t
        cardholderName {char const *}} {
    unsigned chargeID = %<CreditCardCharge operation genChargeID>% ;
    %<my assign totalValue>%
    %<CreditCardCharge createasync makeCharge {order self}\
        chargeID chargeID\
        accountNumber accountNumber\
        billingAddress billingAddress\
        cardholderName cardholderName\
        cardExpirationDate cardExpirationDate\
        chargeAmount totalValue\
        R7 self>% // ❶ ❷
}

transition SubmittingCharge - paymentApproved -> BeingPackedandShipped
transition SubmittingCharge - paymentDeclined -> PaymentNotApproved
```

❶ Generating an event to the class *creator*, *i.e.* generating a creation event, is accomplished as the `createasync` subcommand on the target class. Again, all the values of the attributes must be supplied since the instance must be created before the creation event is dispatched. The model is a bit inconsistent here in the event parameters. The receiving creation state treats `order` as an instance reference, but the action language send `orderID` as the parameter. We use the instance reference here to agree with the receiving state activity. Note also that we must supply the instance reference require to instanciate `R7` at creation time.

❷ The action language Credit Card Charge state activities class keep refering to relationship, `R8`. Clearly, `R7` is the one that is intended.

Note that we have factored the `makeCharge` event parameters. In the model graphic on page 342, you can see that there are many parameters passed with the event. All of these parameters, except the `Order` instance reference, are simply used to set the values of the `CreditCardCharge` instance and so we have factored them out to the creation event generation as discussed above. This is an interesting example of how the idoms of action language have to be translated into a different set of idoms in `micca`.

```
<<Order State Model>>=
state PaymentNotApproved {} {
    %<instance customer findOneRelated self R5>%
```

```
    MRT_InstId customerID = %<instance customer instid>% ;
    %<externalop OnlineCustomer_chargeDeclined customerID customerID\
            email customer->email>% ; // ❶
}

transition PaymentNotApproved - subCharge -> SubmittingCharge
```

❶ `Micca` does not have any direct support for generating events to external entities. Instead, we create explict bridge operations, in this case `OnlineCustomer_chargeDeclined`, passing the required parameters. That bridge operation can then map the semantics of a *declined charge* onto the appropriate operation of the domain that represents the Online Customer external entity. In a `micca` translation, all the interactions with external entities are represented by procedures. Those procedures then handle whether the interaction is synchronous or event based. We will see examples of this in the External Entities section below.

```
<<Order State Model>>=
state BeingPackedandShipped {} {
    %<instance customer findOneRelated self R5>%
    %<instance customer attr purchasesMade>% += 1 ; // ❶

    MRT_InstId customerID = %<instance customer instid>% ;
    %<externalop OnlineCustomer_chargeApproved customerID customerID\
            email customer->email>% ;

    unsigned shipmentID = %<Shipment operation genShipmentID>% ;
    %<Warehouse instref warehouse>% // ❷
    %<my assign orderID>%
    warehouse = %<Warehouse operation chooseWarehouse orderID orderID>% ;
    %<Shipment createasync requestShipment {}\
        shipmentID shipmentID\
        recipient self->recipient\
        deliveryAddress self->deliveryAddress\
        contactPhone self->contactPhone\
        R6 self\
        R24 warehouse>%
}
transition BeingPackedandShipped - orderDelivered -> DeliveredtoCustomer
```

❶ The model never updates this attribute, but this seems to be the appropriate place.

❷ The model is rather vague about how `R24` is established. The relationship is unconditional on the `Warehouse` side so we must establish it in this activity. We have punted here and invoke a class based operation of the `Warehouse` passing the `orderID`. Presumably that procedure has some policy in it to select a warehouse based on the contents of the `Order` given by the `orderID`. We will see in the Warehouse how the choice of a `Warehouse` is resolved (hint, for our purposes very, very simply).

```
<<Order State Model>>=
state DeliveredtoCustomer {} {
    %<instance customer findOneRelated self R5>% // ❶
    MRT_InstId customerID = %<instance customer instid>% ;
    %<externalop OnlineCustomer_orderReportedDelivered customerID customerID\
            email customer->email>% ;
}
```

❶ So the question arises as to whether we should test the `customer` instance reference for being empty. After all, R5 is conditional on the `Customer` side. In this case the answer is no, since all paths to this state come through the **Establishing Customer and Verifying Payment** state where a link to `R5` is made and so we have every expectation that

we will find a single instance related by `R5`. We know that we will find at most one since `R5` has a multiplicity of one to use `findOneRelated` on an association with multiplicity greater than one will fail to generate. Depending upon your level of paranoia, you might want to `assert` that `customer !=NULL` since that code can be removed by conditional compilation. The referential integrity checks performed by `micca` go a long way to insure correct operation.

### Order Placement Class

```
<<Order Placement Class Definition>>=
class OrderPlacement {}
```

```
<<Order Placement Class Definition>>=
association R5 -associator OrderPlacement Order 1..*--0..1 Customer ; # ❶
```

❶ There is an interesting policy decision represented here. A Customer is not a Customer unless he/she has placed at least one order. Interesting — not wrong or bad — just interesting.

### Product Selection Class

The `ProductSelection` class is an associative class and has a state model. The state model is interesting because it has both a creation event and a final state. The state, `RemovingSelection`, is final and the instance is deleted after executing the state activity for that state.

```
<<Product Selection Class Definition>>=
class ProductSelection {
    attribute quantity unsigned -default 0
    attribute unitPriceOfSelection unsigned -default 0
    attribute selectionValue unsigned -dependent {
        *selectionValue = %<my attr unitPriceOfSelection>% * %<my attr quantity>% ;
    }

    statemodel {
        <<Product Selection State Model>>
    }
}
```

```
<<Product Selection Class Definition>>=
association R4 -associator ProductSelection ShoppingCart 0..*--1..* Product
```

#### Product Selection Class State Model

```
<<Product Selection State Model>>=
initialstate NewSelection ; # ❶
```

❶ As it turns out, we also do synchronous creation of this class so we want to be explicit about the initial state.

```
<<Product Selection State Model>>=
transition @ - addSelection -> NewSelection

state NewSelection {
        productID {char const *}
        quantity unsigned} {
    %<my update quantity quantity>%

    %<Product instref product>%
```

```
    product = %<Product operation findByProductID productID productID>% ;
    %<instance product assign unitPrice>%
    %<my update unitPriceOfSelection unitPrice>%

    %<my signal changeQuantity quantity quantity>%
}
transition NewSelection - changeQuantity -> ChangingQuantity
```

Here we run into a quandary in the model. The two states below signal the `cartUpdated` event to a `ShoppingCart`. Unfortunately, the Shopping Cart state model (see page 340) does not define a `cartUpdated` event. So, we just ignore it.

```
<<Product Selection State Model>>=
state ChangingQuantity {quantity unsigned} {
    %<my update quantity quantity>%
    if (%<my attr quantity>% == 0) {
        %<my signal removeSelection>%
    }
}
transition ChangingQuantity - removeSelection -> RemovingSelection

state RemovingSelection {} {
    %<instance cart findOneRelated self ~R4>%
    %<instset prodset selectRelated cart R4>%
    if (%<instset prodset cardinality>% <= 1) { // ❶
        %<instance cart signal cancel>%
    }
}
final RemovingSelection
```

❶ This is somewhat tricky, conceptually. The intent is to figure out if we need to signal `cancel` to the `cart`. We want to do that when `self` is the only remaining instance of Product Selection related to the `cart`. Why test against 1 (less than or equal to 1 is just "belt and suspenders")? Because `RemovingSelection` is a final state (see below), our `self` instance will be automatically deleted at the end of the state activity and that deletion will also clean up the association pointers. So we test against 1, since at this time the `self` instance still exists. Note also that referential integrity is violated, temporarily, in this case. It is not until the `cancel` event to `cart` is received does the `cart` instance get deleted, thereby restoring referential integrity. This case does not cause any referential integrity violation because signaling the `cancel` event continues the thread of control, postponing the integrity evaluation until after the event is dispatched. The model uses an `unrelate` construct that `micca` does not provide since `micca` know how to clean up association pointers. This is why the model tests for empty, since the `unrelate` presumably subtracted `self` from the set of `R4` instances and the navigation to `Product` would then come up empty.

### ShoppingCart Class

```
<<Shopping Cart Class Definition>>=
class ShoppingCart {
    attribute cartID unsigned -default 0
    attribute totalValue unsigned -dependent {
        unsigned total = 0 ;
        %<instance psel foreachRelated self {R4 ProductSelection}>%
            %<instance psel assign selectionValue>% ;
            total += selectionValue ;
        %<end>%
        *totalValue = total ;
    }

    statemodel {
        <<Shopping Cart State Model>>
    }
```

```
    classop unsigned genCartID {} {
        static unsigned cartID = 1 ;

        return cartID++ ;
    }

    classop {struct ShoppingCart *} findByCartID {cartID unsigned} {
        %<ShoppingCart findWhere cart {cartID == cart->cartID}>%
        return cart ;
    }
}
```

```
<<Shopping Cart State Model>>=
transition @ - startCart -> NewOrder

state NewOrder {productID {char const *} quantity unsigned} {
    %<my attr cartID>% = %<ShoppingCart operation genCartID>% ;
    %<my signal addSelection productID productID quantity quantity>%
}
transition NewOrder - addSelection -> AddingSelectiontoOrder

state AddingSelectiontoOrder {productID {char const *} quantity unsigned} {
    %<Product instref product>%
    product = %<Product operation findByProductID productID productID>% ;
    %<ProductSelection create newSelection\
            quantity quantity\
            unitPriceOfSelection product->unitPrice\
            R4 {Product product ShoppingCart self}>%
}
transition AddingSelectiontoOrder - addSelection -> AddingSelectiontoOrder
transition AddingSelectiontoOrder - cancel -> CancelingEntireOrder
transition AddingSelectiontoOrder - checkOut ->\
        EstablishingCustomerandVerifyingPayment
```

```
<<Shopping Cart State Model>>=
state CancelingEntireOrder {} {
    %<instance selection foreachRelated self {R4 ProductSelection}>%
        %<instance selection delete>% // ❶
    %<end>%
}
final CancelingEntireOrder
```

❶ This solution in the model seems a bit weak. The only way we get to the `CancelingEntireOrder` state is by receiving the `cancel` event. That event is only signaled by the `ProductSelection` class when it has detected that there are no other selections in the shopping cart. So, it seems that search for related `ProductSelection` instances will always come up empty and this iteration loop never executes. The model also seems to be missing the deletion of the associative class. Since this state is a final state, the `ShoppingCart` instance is deleted at the end of the state activity. An alternative would be to eliminate this state and the `cancel` event and simply delete the instance where `ProductSelection` signaled the event. It could be that the analysts were attempting to "future proof" the model for the case that the `cancel` event could be send under different circumstances than it actually is.

```
<<Shopping Cart State Model>>=
state EstablishingCustomerandVerifyingPayment {
        accountNumber {char const *}
        billingAddress {char const *}
        cardExpirationDate DateType_t
        cardholderName {char const *}
        customerEmail {char const *}
```

```
        customerName {char const *}
        customerPhone {char const *}
        shippingAddress {char const *}} {
    unsigned orderID = %<Order operation genOrderID>% ;
    %<my assign totalValue>%
    %<Order createasync checkOut {
                accountNumber accountNumber
                billingAddress billingAddress
                cardExpirationDate cardExpirationDate
                cardholderName cardholderName
                customerEmail customerEmail}\
            orderID orderID\
            totalValue totalValue\
            recipient customerName\
            deliveryAddress shippingAddress\
            contactPhone customerPhone\
            R10 self>%
}
```

## Credit Card Charge Class

```
<<type aliases>>=
typealias APCode_t {enum {AP_approved, AP_overlimit, AP_noAccount,\
        AP_connectionFailed, AP_accountDataMismatch, AP_expired, AP_pending}}
typealias DateType_t typename(time_t)
```

```
<<Credit Card Charge Class Definition>>=
class CreditCardCharge {
    attribute chargeID unsigned
    attribute accountNumber {char[32]}
    attribute cardholderName {char[128]}
    attribute billingAddress {char[128]}
    attribute cardExpirationDate DateType_t
    attribute dateChargeMade DateType_t -default {0}
    attribute chargeAmount unsigned
    attribute approvalCode APCode_t -default AP_pending

    statemodel {
        <<Credit Card Charge State Model>>
    }

    classop unsigned genChargeID {} {
        static unsigned idnum = 100 ;
        return idnum++ ;
    }
}
```

```
<<Credit Card Charge Class Definition>>=
association R7 CreditCardCharge 0..*--1 Order
```

### Credit Card Charge Class State Model

```
<<Credit Card Charge State Model>>=
transition @ - makeCharge -> RequestingChargeApproval

state RequestingChargeApproval {order {struct Order *}} {
    MRT_InstId cccID = %<my instid>% ;
```

```
    %<externalop CreditCardCompany_requestChargeApproval\
        cccID cccID\
        accountNumber self->accountNumber\
        billingAddress self->billingAddress\
        cardholderName self->cardholderName\
        cardExpirationDate self->cardExpirationDate\
        transactionID self->chargeID\
        chargeAmount self->chargeAmount>% ;

    %<my delaysignal 60000 chargeProcessingNotCompleted>%
}
transition RequestingChargeApproval - chargeProcessed -> ProcessingCompleted
transition RequestingChargeApproval - chargeProcessingNotCompleted ->\
    DeclineForTimeout
```

```
<<Credit Card Charge State Model>>=
state ProcessingCompleted {resultCode APCode_t} {
    %<my update approvalCode resultCode>%
    %<instance order findOneRelated self R7>% // ❶
    %<my canceldelayed chargeProcessingNotCompleted>% // ❷
    if (resultCode == AP_approved) {
        %<OrderPayment create payment R8 {CreditCardCharge self Order order}>% // ❸
        %<instance order signal paymentApproved>%
    } else {
        %<instance order signal paymentDeclined>%
    }
}
```

❶ Here again we need to use `R7`.

❷ To be tidy, we cancel the delayed event that times out the processing of the credit card charge. Otherwise it is necessary to ignore that event in this state.

❸ Upon approval of the charge we can then create an instance of `R8` to record the charge that actually paid for the order.

```
<<Credit Card Charge State Model>>=
state DeclineForTimeout {} {
    %<my update approvalCode AP_connectionFailed>%
    %<instance order findOneRelated self R7>% // ❶
    %<instance order signal paymentDeclined>%
}
```

❶ And the final substitution of `R7` for what was orginally `R8` in the model.

### Order Payment Class

```
<<Order Payment Class Definition>>=
class OrderPayment {}
```

```
<<Order Payment Class Definition>>=
association R8 -associator OrderPayment CreditCardCharge 0..1--0..1 Order ; #❶
```

❶ This looks like an error in the model. The `R7` relationship is tracking attempts to pay for an order by a credit card charge. The `R8` relationship is tracking when an attempt actually results in a purchase. It has to be conditional on the Order side because an order must be able to exist before it is paid for. The relationship is rather weak and we will see other problems in the state activities for the `CreditCardCharge` class.

## Shipping Subsystem

The Shipping subsystem consists of the following classes.

```
<<Shipping Subsystem>>=
<<Shipment Class Definition>>
<<Delivery Class Definition>>
<<Past Packing Class Definition>>
<<Packing Class Definition>>
<<Shipping Company Class Definition>>
<<Warehouse Clerk Class Definition>>
<<Warehouse Class Definition>>
<<Shipping Clerk Class Definition>>
<<Stock Clerk Class Definition>>
<<Off Duty Clerk Class Definition>>
<<Shipment Item Class Definition>>
```

### Shipment Class

```
<<Shipment Class Definition>>=
class Shipment {
    attribute shipmentID unsigned
    attribute trackingNumber {char[32]} -default {""}

    attribute recipient {char[128]}
    attribute deliveryAddress {char[128]}
    attribute contactPhone {char[32]}
    attribute timePrepared DateType_t -default {0}
    attribute timePickedUp DateType_t -default {0}
    attribute timeDelivered DateType_t -default {0}
    attribute waitingToBePacked bool -default true

    statemodel {
        <<Shipment State Model>>
    }

    <<Shipment Operations>>
}
```

The model shows the `shippingCompany` and `trackingNumber` attributes form a secondary identifier. This is problematic because `shippingCompany` is a referential attribute that realizes `R21`. Below we discuss the problems of the `R21` and in the end make it conditional on the `ShippingCompany` side. In our approach, we factor `R21` as a class based association as required by `micca`.

```
<<Shipment Class Definition>>=
association R24 Shipment 0..*--1 Warehouse
association R6 Shipment 0..1--1 Order
```

#### Shipment Class State Model

There are a number of minor issues that have to be resolved in the state activities for the `Shipment` class. Pay particular attention to the callouts when comparing the translated code to the action language source.

```
<<Shipment State Model>>=
transition @ - requestShipment -> PreparingShipment

state PreparingShipment {} {
```

```
    %<instance order findOneRelated self R6>%
    %<instance order assign orderID>%

    %<instance selection foreachRelated order R10 {R4 ProductSelection}>%
        %<ShipmentItem create item\
                R9 {Shipment self ProductSelection selection}>%
    %<end>% // ❶
    %<instance warehouse findOneRelated self R24>%
    %<instance warehouse signal shipmentReadyToPack>% // ❷
}
transition PreparingShipment - packed -> PackedandAwaitingTrackingNumber
transition PreparingShipment - trackingNumberAssigned ->\
    NumberAssignedandWaitingtobePacked
```

❶ The model doesn't seem to establish R9 with the Shipment. We do so here assuming no items are shipped initially. Presumably the Shipping Clerk will have to update the quantity shipped as the items are packed.

❷ This event should be going to the Warehouse. The model graphic ships it off to an external entity, yet the event name appears on the Warehouse state diagram and it is the Warehouse class that assigns Shipping Clerks for packing.

Note that the state activity for the `Packed and Awaiting Tracking Number` and the `Packed` states are the same. We have factored that into an instance based operation, `updatePackingInfo`. The same is true of the `Number Assigned and Waiting to be Packed` and `Number Assigned` states which is factored into the `updateTrackingInfo` instance operation. These states allow shipment packing by the shipping clerk and tracking number assignment by the shipping company to happen in either order. The way we resolved the problems the model has with `R21` means we only take one path through the states. But the states are there and so we implement it as given.

```
<<Shipment State Model>>=
state PackedandAwaitingTrackingNumber {clerkID unsigned} {
    %<my operation updatePackingInfo clerkID clerkID>% ;
}
transition PackedandAwaitingTrackingNumber - trackingNumberAssigned ->\
    NumberAssigned

state NumberAssignedandWaitingtobePacked {
        shippingCompany {char const *}
        trackingNumber {char const *}} {
    %<my operation updateTrackingInfo\
            shippingCompany shippingCompany\
            trackingNumber trackingNumber>% ;
}
transition NumberAssignedandWaitingtobePacked - packed -> Packed

state NumberAssigned {
        shippingCompany {char const *}
        trackingNumber {char const *}} {
    %<my operation updateTrackingInfo\
            shippingCompany shippingCompany\
            trackingNumber trackingNumber>% ;
}
transition NumberAssigned - pickedUp -> InTransittoCustomer

state Packed {clerkID unsigned} {
    %<my operation updatePackingInfo clerkID clerkID>% ;
}
transition Packed - pickedUp -> InTransittoCustomer
```

Here are the factored instance based operations. The translation from action language to "C" follows our usual pattern. The only thing we have done here is to place duplicated code into one place.

```
<<Shipment Operations>>=
instop void updateTrackingInfo {
        shippingCompany {char const *}\
        trackingNumber {char const *}} {
    %<my update trackingNumber trackingNumber>%
    %<ShippingCompany instref company>%
    company = %<ShippingCompany operation findByCompanyName\
            companyName shippingCompany>% ;
    %<Delivery create delivery R21 {Shipment self ShippingCompany company}>%
}

instop void updatePackingInfo {clerkID unsigned} {
    time(&self->timePrepared) ;
    %<WarehouseClerk instref clerk>%
    clerk = %<WarehouseClerk operation findByClerkID clerkID clerkID>% ;
    %<PastPacking create pp R22 {Shipment self WarehouseClerk clerk}>%

    MRT_InstId shipmentID = %<my instid>% ;
    %<externalop ShippingCompany_shipmentReadyForPickup shipmentID shipmentID>% ;
}

classop unsigned genShipmentID {} {
    static unsigned shipmentID = 200 ;

    return shipmentID++ ;
}
```

Once we are packed and tracked, the last two states track the delivery of the shipment.

```
<<Shipment State Model>>=
state InTransittoCustomer {} {
    time(&self->timePickedUp) ;
}
transition InTransittoCustomer - deliveryConfirmed -> Delivered

state Delivered {timeDelivered DateType_t} {
    time(&self->timeDelivered) ;
    %<instance order findOneRelated self R6>%
    %<instance order signal orderDelivered>%
}
```

## Delivery Class

This is another class introduced to be an associator, in this case for `R21`.

```
<<Delivery Class Definition>>=
class Delivery {}
```

```
<<Delivery Class Definition>>=
association R21 -associator Delivery Shipment 0..*--0..1 ShippingCompany ; #❶
```

❶ The model has this relationship as unconditional on the Shipping Company side. This is problematic when a Shipment is created. At that point in time we have no shipping company or tracking number. We could probably choose a company and ask for a number when the Shipment is created, but that work seems to be handled by the Shipment state model. That state model implies that we could pack the shipment or receive a tracking number in either order. Consequently, the easiest thing to do is to weaken the relationship to make it conditional on the ShippingCompany side to await the final company and tracking number assignment.

### Past Packing Class

This class is introduced to serve as the associator for R22.

```
<<Past Packing Class Definition>>=
class PastPacking {}
```

```
<<Past Packing Class Definition>>=
association R22 -associator PastPacking Shipment 0..*--0..1 WarehouseClerk
```

### Packing Class

This class is introduced to serve as the associator for R23.

```
<<Packing Class Definition>>=
class Packing {}
```

```
<<Packing Class Definition>>=
association R23 -associator Packing ShippingClerk 0..1--0..1 Shipment
```

### Shipping Company Class

```
<<Shipping Company Class Definition>>=
class ShippingCompany {
    attribute companyName {char[32]}
    attribute trackingWebsite {char[256]}
    attribute customerServicePhone {char[32]}
    attribute localDispatchPhone {char[32]}
    attribute localOffice {char[128]}
    attribute localContact {char[64]}

    classop {struct ShippingCompany *} findByCompanyName {
            companyName {char const *}} {
        %<ShippingCompany findWhere company\
                {strcmp(companyName, company->companyName) == 0}>%
        return company ;
    }
}
```

### Warehouse Class

```
<<Warehouse Class Definition>>=
class Warehouse {
    attribute warehouseName {char[32]}
    attribute warehouseLocation {char[128]}

    statemodel {
        <<Warehouse State Model>>
    }

    <<Warehouse Operations>>
}
```

As discussed above, we created a class based operation to choose which Warehouse would be used to ship an order. Presumably, there is a policy about such choices. Such policies can be quite complicated, involving inventory, distances and other costs. However, the model is silent on the choice policies. Here, we do the simplest thing we can and simply choose an arbitrary Warehouse.

```
<<Warehouse Operations>>=
classop {struct Warehouse *} chooseWarehouse {orderID unsigned} {
    %<Warehouse idtoref 0 warehouse>% // ❶
    return warehouse ;
}
```

❶ Just pick the first warehouse we have.

```
<<Warehouse Operations>>=
instop {struct Shipment *} findReadyShipment {} {
    %<instance readyShipment foreachRelatedWhere\
            {readyShipment->waitingToBePacked} self ~R24>%
        return readyShipment ;
    %<end>%

    return NULL ;
}

instop {struct ShippingClerk *} findFreeShippingClerk {} {
    %<instance freeClerk foreachRelatedWhere\
            {freeClerk->awaitingAssignment} self ~R25 {~R27 ShippingClerk}>%
        return freeClerk ;
    %<end>%

    return NULL ;
}
```

#### Warehouse Class State Model

The state model for the `Warehouse` is actually an assigner that assigns `ShippingClerks` to pack and ship `Shipments`.

```
<<Warehouse State Model>>=
initialstate WaitingforaShipment ; # ❶
defaulttrans IG ; # ❷
```

❶ Just to be explicit, we state the initial state of the state model.

❷ For assigners, we want requests for shipment and free clerk notifications to be able to arrive at any time. So we will ignore any that arrive when we are dealing with other matters. Notice that each state searches for shipments and clerks in the state that is needed. This is a common assigner idom.

```
<<Warehouse State Model>>=
state WaitingforaShipment {} {
    %<Shipment instref readyShipment>%
    readyShipment = %<my operation findReadyShipment>% ;
    if (readyShipment != NULL) {
        %<my signal shipmentReadyToPack>%
    }
}
transition WaitingforaShipment - shipmentReadyToPack -> WaitingforaFreeClerk

state WaitingforaFreeClerk {} {
    %<ShippingClerk instref freeClerk>%
    freeClerk = %<my operation findFreeShippingClerk>% ;
    if (freeClerk != NULL) {
        %<my signal clerkFree>%
```

```
    }
}
transition WaitingforaFreeClerk - clerkFree -> AssigningClerktoShipment

state AssigningClerktoShipment {} {
    %<Shipment instref readyShipment>%
    readyShipment = %<my operation findReadyShipment>% ;

    %<ShippingClerk instref freeClerk>%
    freeClerk = %<my operation findFreeShippingClerk>% ;

    %<Packing create pack R23 {ShippingClerk freeClerk Shipment readyShipment}>% // ❶
    %<instance readyShipment update waitingToBePacked false>%
    %<instance freeClerk update awaitingAssignment false>%

    %<instance freeClerk signal clerkAssigned>%
    %<my signal clerkAssignedToShipment>%
}
transition AssigningClerktoShipment - clerkAssignedToShipment ->\
        WaitingforaShipment
```

❶ Creating the associative instance establishes the relationship.

### Warehouse Clerk Class

```
<<Warehouse Clerk Class Definition>>=
class WarehouseClerk {
    attribute clerkID unsigned
    attribute clerkName {char[64]}
    attribute goOffDutyAtEndOfJob bool -default false ; # ❶

    classop {struct WarehouseClerk *} findByClerkID {clerkID unsigned} {
        %<WarehouseClerk findWhere clerk {clerkID == clerk->clerkID}>%
        return clerk ;
    }
}
```

❶ This attribute is not in the model graphic, but it is mentioned in an errata and in the state model for `ShippingClerk`.

```
<<Warehouse Clerk Class Definition>>=
generalization R27 WarehouseClerk OffDutyClerk StockClerk ShippingClerk
association R25 WarehouseClerk 1..*--1 Warehouse
```

### Shipping Clerk Class

```
<<Shipping Clerk Class Definition>>=
class ShippingClerk {
    attribute awaitingAssignment bool -default true

    statemodel {
        <<Shipping Clerk State Model>>
    }

    classop MRT_DelayTime randomdelay {} {
        // random number 1 - 10 in tenths of a second
        return (MRT_DelayTime)((rand() % 10 + 1) * 100) ;
```

```
    } ; # ❶
}
```

❶ The `randomdelay` class operation is used to simulate the actions of a shipping clerk by signalling delayed events showing the clerk has performed some action. We use this to resolve some of the issues with the `ShippingClerk` state model. See below.

#### Shipping Clerk Class State Model

The state model for `ShippingClerk` also has a number of minor issues that need to be resolved. The state activities refer to `R7` and `R8`. These are not correct and should be `R23` and `R25`. Also we are not given any action language for the state activities of a number of states. We presume these are states driven by the activity of a real shipping clerk as he/she selects, packs and labels the shipment.

For our purposes it is sufficient to simply drive the state transition along with some self generated events. We will put some time delay in the events to try to simulate the time take to accomplish the tasks in the physical world. It is easy enough to capture this strategy in a procedure.

This procedure will generate a delayed event at a pseudo-random time between 0 and `max` tenths of a second. We use it below to drive the `ShippingClerk` state model through the states that were not completely specified in the model graphic.

```
<<Shipping Clerk State Model>>=
transition @ - startShipping -> WaitingforaJob

state WaitingforaJob {} {
    %<my update awaitingAssignment true>%
    %<instance warehouse findOneRelated self R27 R25>%
    %<instance warehouse signal clerkFree>%
}
transition WaitingforaJob - clerkAssigned -> SelectingBooks
transition WaitingforaJob - offDuty -> OffDuty

state SelectingBooks {} {
    %<instance shipment findOneRelated self R23>%
    MRT_InstId clerkID = %<my instid>% ;
    %<externalop ShippingClerk_shipmentReadyToPack clerkID clerkID\
            shipmentID shipment->shipmentID>% ;

    MRT_DelayTime time = %<ShippingClerk operation randomdelay>% ; // ❶
    %<my delaysignal time booksSelected>%
}
transition SelectingBooks - booksSelected -> PackingBox
```

❶ We are adding this to the state activity to simply drive the model along simulating actions by a real shipping clerk. This technique is also used below to keep the model transitions going.

```
<<Shipping Clerk State Model>>=
state PackingBox {} {
    MRT_DelayTime time = %<ShippingClerk operation randomdelay>% ;
    %<my delaysignal time boxPacked>%
}
transition PackingBox - boxPacked -> SealingBox
```

The `SealingBox` state gives us the opportunity to update the quantity of items that was actually shipped. This seems to be missing from the model, but presumably this is updated by the shipping clerk as the box is packed since at that time he/she would know if the requested quantity of an item was in stock and able to be shipped. Below we take the optimistic attitude that we have infinite stock and always ship the requested quantity of each item. The model does not (but probably should) deal with `StockedProduct` instances, accounting for the quantity on hand and triggering any reordering.

```
<<Shipping Clerk State Model>>=
state SealingBox {} {
    // ❶
    %<instset items selectRelated self R23 {R9 ShipmentItem}>%
    %<instset items foreachSelected item>%
        %<instance selection findOneRelated item R9>%
        %<instance selection assign quantity>%
        %<instance item update quantityShipped quantity>%
    %<end>%
    MRT_DelayTime time = %<ShippingClerk operation randomdelay>% ;
    %<my delaysignal time boxSealed>%
}
transition SealingBox - boxSealed -> AttachingShippingLabel
```

❶ We accomplish updating `quantityShipped` attribute of the `ShippmentItems` instances by traversing `R23` and `R9` to obtain a set of `ShipmentItem` instance references and then iterate across them, continuing along `R9` to obtain the corresponding `ProductSelection` instance. We then read the `quantity` attribute and use its value to update the `quantityShipped` value.

```
<<Shipping Clerk State Model>>=
state AttachingShippingLabel {} {
    MRT_DelayTime time = %<ShippingClerk operation randomdelay>% ;
    %<my delaysignal time shippingLabelAttached>%
}
transition AttachingShippingLabel - shippingLabelAttached ->\
    DeliveringBoxtoLoadingDock

state DeliveringBoxtoLoadingDock {} {
    MRT_DelayTime time = %<ShippingClerk operation randomdelay>% ;
    %<my delaysignal time boxAtLoadingDoc>%
}
transition DeliveringBoxtoLoadingDock - boxAtLoadingDoc -> CompletingJob

state CompletingJob {} {
    %<instance currentShipment findOneRelated self R23>%
    %<instance clerk findOneRelated self R27>%
    %<instance clerk assign clerkID>%
    %<instance currentShipment signal packed clerkID clerkID>%

    %<instance packing findOneRelated self {R23 Packing}>%
    %<instance packing delete>%
    if (%<instance clerk attr goOffDutyAtEndOfJob>%) {
        %<my signal offDuty>%
    } else {
        %<my update awaitingAssignment true>%
        %<instance myWarehouse findOneRelated self R27 R25>%
        %<instance myWarehouse signal clerkFree>%
    }
}
transition CompletingJob - clerkAssigned -> SelectingBooks
transition CompletingJob - offDuty -> OffDuty

state OffDuty {} {
    %<instance clerk findOneRelated self R27>%
    %<R27 reclassify clerk OffDutyClerk offduty>%
} ; # ❶
```

❶ Note that we do **not** define the `OffDuty` state as `terminal` despite the model graphic containing a terminal transition.

This is because the `ShippingClerk` instance will not exist after the migration to an `OffDutyClerk` and so we don't want the run time architecture to attempt to delete what no longer exists anyway.

### Stock Clerk Class

```
<<Stock Clerk Class Definition>>=
class StockClerk {
    attribute idle bool -default true
}
```

### Off Duty Clerk Class

```
<<Off Duty Clerk Class Definition>>=
class OffDutyClerk {
}
```

### Shipment Item Class

```
<<Shipment Item Class Definition>>=
class ShipmentItem {
    attribute quantityShipped unsigned -default 0
}
```

```
<<Shipment Item Class Definition>>=
association R9 -associator ShipmentItem Shipment 0..*--1..* ProductSelection
```

## Product Specification Subsystem Population

At this point we have completed the translation of the model. When the above code is assembled together and submitted to `micca` it results in a set of procedures and data structures that represent the actions of the domain. `Micca` provides a run time component to map model level actions, *e.g.* navigating relationships to find related instances, onto "C" language constructs to implement the intent of the model.

However, we are not done yet. First we must populate the domain with instances. Although many of the classes create their instances at run time, most of the Product Specification subsystem must be populated with instances in order to have anything to buy at our online store. In the next sections, we will supply values for those classes that are not created at run time.

Note that populating a domain before running it is, in general, a necessary task and one that is frequently overlooked. Depending on the quantity of data, population can be a large undertaking. It is easy to forget about the population step as we are involved in working out the domain and obtaining its translation. Also remember in real development, you may need several domain populations to support testing and integration. This is particularly true if the domain contains a lot of data. Workflow and timing may dictate that a separate group of people is creating the population for the delivered system and the development team will need to supply their own population for testing and integrating. Also, testing and integration concerns will probably drive the values selected for some attributes as this may drive the code path in some of the state activities. Finally, testing that randomizes, in a controlled and repeatable way, the values of attributes can often find bugs that human composed values miss. The idea is that randomized attribute values will drive the domain's execution into broader areas of its state space more effectively than narrowly composed values made up by humans.

The population given below is quite minimal and is intended only to demonstrate the translation. Every class is populated with something and, as you will see, many of the attribute values are nonsense.

```
<<Product Specification Population>>=
<<Product Population>>
<<Publisher Population>>
<<Author Population>>
<<Authorship Population>>
<<Authorship Sequence Population>>
<<Book Product Population>>
<<Recording Product Population>>
<<Software Product Population>>
<<Special Order Product Population>>
<<Stocked Product Population>>
<<Product Category Population>>
<<Category Group Population>>
<<Recording Population>>
<<Artist Population>>
<<Performance Credit Population>>
<<Performance Credit Sequence Population>>
<<Recording Format Population>>
<<Platform Population>>
<<Computer Software Population>>
```

`Micca` provides support to populate a domain. One of the advantages of the relational approach is that a domain can be populated by pure data values alone. It is not necessary to execute any action language equivalent code to create instances. This even includes relating instances across relationships. Since we have replaced referential attributes, we use names for the instances as a means of specifying relationship linkages.

The basic command to populate a class provides two approaches to specify values. One is better suited to specifying constant values and the other better suited when attribute values are held in ordinary variables. We will see both used.

### Product Population

```
<<Product Population>>=
class Product {
    table {
        productID
        productName
        copyright
        unitPrice
        description
        website
        currentlyAvailable
        R1
        R15
    } mbxuml {
        {"14387"}
        {"Executable UML: A Foundation for Model-Driven Architecture"}
        {"2002"}
        2995
        {"Book on software development using models."}
        {"www.executableumlbook.com"}
        true
        aw
        non_fiction
    } cmc {
        {"14388"}
        {"Carnival Music for Children"}
        {"1955"}
        1000
        {"Music as heard at old time carnivals."}
        {""}
```

```
        true
        otm
        music
    } macwd {
        {"14389"}
        {"Word 2020 for MacOSX"}
        {"2020"}
        5995
        {"Infamous word processing program"}
        {"www.microsoft.com"}
        true
        ms
        software
    } ; # ❶
}
```

❶ This is an example of populating using the table command. It saves some typing to specify repeatedly the attribute names. Note that "C" string literals in quotes must also be enclosed in braces. Turns out that double quotes are Tcl metacharacters and we need the braces to insure the quote characters themselves are passed through as part of the attribute value.

### Publisher Population

```
<<Publisher Population>>=
class Publisher {
    table {
        groupCode
        publisherCode
        name
        address
        website
    } aw {
        {"0"}
        {"201"}
        {"Addison-Wesley"}
        {"New York, NY"}
        {"www.aw.com"}
    } otm {
        {"1"}
        {"523"}
        {"Old-Time Music"}
        {"St. Louis, MO"}
        {"www.otm.com"}
    } ms {
        {"2"}
        {"666"}
        {"Microsoft, Inc."}
        {"Seattle, WA"}
        {"www.microsoft.com"}
    }
}
```

### Author Population

```
<<Author Population>>=
class Author {
    table {
        name
```

```
        website
        email
    } sjm {
        {"Stephen J. Mellor"}
        {"stephenmellor.com"}
        {"stephen@stephenmellor.com"}
    } mjb {
        {"Marc J. Balcer"}
        {"marcbalcer.com"}
        {"marc@marcbalcer.com"}
    }
}
```

### Authorship Population

```
<<Authorship Population>>=
class Authorship {
    instance sjm R2 {Author sjm BookProduct mbxuml}
    instance mjb R2 {Author mjb BookProduct mbxuml}
}
```

```
<<Authorship Sequence Population>>=
class AuthorshipSequence {
    instance mbxuml R3 {backward mjb forward sjm} ; # ❶
}
```

❶ Note that we are interpreting `R3` in the forward direction as the "precedes" semantic. Thus, instance `sjm` is preceded by nothing and instance `mjb` is preceded by instance `sjm`. You may interpret this type of reflexive association in either way, however, you must remain consistent. The reason for this interpretation is that the first instance of the implied ordered set is that instance where traversing `R3` is empty. Finding the next instance in the set is equivalent to tranversing `~R3`, with the last instance having no related instance for `~R3`. Using the opposite interpretation can make finding the first instance of the set more complicated. An easy way to remember this interpretation is that the references *back link* together the members of the ordered set implied by the association.

### Book Product Population

```
<<Book Product Population>>=
class BookProduct {
    table {
        bookNumber
        titleCode
        title
        subtitle
        R11
    } mbxuml {
        {"ISBN 0-201-74804-5"}
        {"74804-5"}
        {"Executable UML"}
        {"A Foundation for Model-Driven Architecture"}
        mbxuml
    }
}
```

### Recording Product Population

```
<<Recording Product Population>>=
class RecordingProduct {
    table {
        runningTime
        R11
        R16
    } cmc {
        203
        cmc
        {Recording cmc RecordingFormat cd}
    }
}
```

### Software Product Population

```
<<Software Product Population>>=
class SoftwareProduct {
    instance macwd\
        productVersion {"20.17.3"}\
        R11 macwd\
        R19 macosx
}
```

### Special Order Product Population

```
<<Special Order Product Population>>=
class SpecialOrderProduct {
    instance cmc\
        specialOrderInstructions {"Their part number: 30557"}\
        daysToDeliver 10\
        R12 cmc
}
```

### Stocked Product Population

```
<<Stocked Product Population>>=
class StockedProduct {
    table {
        quantityOnHand
        reorderThreshold
        reorderIncrement
        reorderInstructions
        R12
    } mbxuml {
        100
        200
        100
        {"Ship as quickly as possible"}
        mbxuml
    } macwd {
        55
        25
        55
        {"Get a discount"}
```

```
        macwd
    }
}
```

### Product Category Population

```
<<Product Category Population>>=
class ProductCategory {
    instance book categoryName {"book"}
    instance fiction categoryName {"fiction"}
    instance non_fiction categoryName {"non_fiction"}
    instance music categoryName {"music"}
    instance software categoryName {"software"}
}
```

### Category Group Population

```
<<Category Group Population>>=
class CategoryGroup {
    instance fictioncat R14 {forward book backward fiction} ; # ❶
    instance nonfictioncat R14 {forward book backward non_fiction}
}
```

❶ We show an example of how `R14` is used to represent the "hierarchy" of product categories. Here, `fiction` and `non_fiction` are subcategories of `book`.

### Recording Population

```
<<Recording Population>>=
class Recording {
    instance cmc title {"Carnival Music for Children"}
}
```

### Artist Population

```
<<Artist Population>>=
class Artist {
    instance mj artistName {"Mary Jane"}
    instance fs artistName {"Fred Smith"}
}
```

### Performance Credit Population

```
<<Performance Credit Population>>=
class PerformanceCredit {
    instance mj R17 {Recording cmc Artist mj}
    instance fs R17 {Recording cmc Artist fs}
}
```

### Performance Credit Sequence Population

```
<<Performance Credit Sequence Population>>=
class PerformanceCreditSequence {
    instance cmc R18 {backward fs forward mj}
}
```

### Recording Format Population

```
<<Recording Format Population>>=
class RecordingFormat {
    instance cd formatName {"CD"}
}
```

### Platform Population

```
<<Platform Population>>=
class Platform {
    instance macosx platformName {"MacOSX"}
}
```

### Computer Software Population

```
<<Computer Software Population>>=
class ComputerSoftware {
    instance macwd title {"Word 2020 for MacOSX"} R20 macwd
}
```

## Ordering Subsystem Population

There are no initial instances in the Ordering subsystem. All instances are created at run time. Most instances are created asynchronously by events. The requirement that **Customers** must have placed an order means that those instances are created when an **Order** is created. So here we only need to allocate the amount of space we require for the ordering subsystem classes. The numbers given below are quite arbitrary.

```
<<Ordering Population>>=
class Customer {
    allocate 20
}
class Order {
    allocate 20
}
class OrderPlacement {
    allocate 20
}
class ProductSelection {
    allocate 40
}
class ShoppingCart {
    allocate 20
}
class CreditCardCharge {
    allocate 40
```

```
}
class OrderPayment {
    allocate 40
}
```

## Shipping Subsystem Population

```
<<Shipping Population>>=
<<Shipment Population>>
<<Delivery Population>>
<<Past Packing Population>>
<<Packing Population>>
<<Shipping Company Population>>
<<Warehouse Population>>
<<Warehouse Clerk Population>>
<<Shipping Clerk Population>>
<<Stock Clerk Population>>
<<Off Duty Clerk Population>>
<<Shipment Item Population>>
```

### Shipment Population

All Shipments are created dynamically, so we must allocated the number of Shipment instance we will allow.

```
<<Shipment Population>>=
class Shipment {
    allocate 20
}
```

### Delivery Population

```
<<Delivery Population>>=
class Delivery {
    allocate 20
}
```

### Past Packing Population

```
<<Past Packing Population>>=
class PastPacking {
    allocate 20
}
```

### Packing Population

```
<<Packing Population>>=
class Packing {
    allocate 20
}
```

### Shipping Company Population

```
<<Shipping Company Population>>=
class ShippingCompany {
    table {
        companyName
        trackingWebsite
        customerServicePhone
        localDispatchPhone
        localOffice
        localContact
    } fedex {
        {"FedEx"}
        {"www.fedex.com"}
        {"222-555-1212"}
        {"223-555-1212"}
        {"200 Elm St., Anywhere, CA 94000"}
        {"George Shipper"}
    } ups {
        {"UPS"}
        {"www.ups.com"}
        {"224-555-1212"}
        {"225-555-1212"}
        {"220 Elm St., Anywhere, CA 94000"}
        {"John Parcel"}
    }
}
```

### Warehouse Population

```
<<Warehouse Population>>=
class Warehouse {
    table {
        warehouseName
        warehouseLocation
    } acme {
        {"Acme Bookstore Warehouse"}
        {"100 Broad St., Somewhere, CA 94000"}

    } mf {
        {"My Fulfillment"}
        {"100 Fremont St., Somewhere, CA 94000"}
    }
}
```

### Warehouse Clerk Population

```
<<Warehouse Clerk Population>>=
class WarehouseClerk {
    table {
        clerkID
        clerkName
        R25
    } fs {
        41
        {"Fred Smith"}
        acme
    } ji {
```

```
        42
        {"John Industrious"}
        acme
    } gs {
        43
        {"George Slacker"}
        mf
    }
}
```

### Shipping Clerk Population

```
<<Shipping Clerk Population>>=
class ShippingClerk {
    instance fs R27 fs
}
```

### Stock Clerk Population

```
<<Stock Clerk Population>>=
class StockClerk {
    instance ji R27 ji
}
```

### Off Duty Clerk Population

```
<<Off Duty Clerk Population>>=
class OffDutyClerk {
    instance gs R27 gs
}
```

### Shipment Item Population

```
<<Shipment Item Population>>=
class ShipmentItem {
    allocate 40
}
```

## External Entities

At this point we have translated and populated the domain. However, we are not quite yet ready to run it. The last step is to deal with the explicit external entities that crop up in the translation.

We must write stub procedures for the external entities that are mentioned in the model. When stubbing external entities, two situations arise:

1. The external entity action is synchronous and can return its result immediately.
2. The external entity action is asynchronous and we must arrange for some further action, usually signaling an event, to occur in order to simulate the external entity properly.

How much of a simulation of the external entities is required to drive the domain execution is a matter that usually requires some introspection in the the model. Many are quite simple as we have in this domain. Others may be quite complex and difficult to simulate. This is one of the reasons why it is often better to integrate domains *from the bottom up* to avoid needing to create complicated external entity stubs. As it turns out, integrating from the bottom up also has the added advantage of always having something that runs. This usually makes project managers happy.

All the procedures start by printing out their invocation sequence. For most of the external entity procedures that is all we strive to do.

```
<<external operations>>=
externalop void CreditCardCompany_requestChargeApproval {
        cccID MRT_InstId
        accountNumber {char const *}
        billingAddress {char const *}
        cardholderName {char const *}
        cardExpirationDate DateType_t
        transactionID unsigned
        chargeAmount unsigned} {
    MRT_DEBUG("%s:\n", __func__) ;
    MRT_DEBUG("    cccID = %u\n", cccID) ;
    MRT_DEBUG("    accountNumber = %s\n", accountNumber) ;
    MRT_DEBUG("    billingAddress = %s\n", billingAddress) ;
    MRT_DEBUG("    cardholderName = %s\n", cardholderName) ;
    MRT_DEBUG("    cardExpirationDate = %s", ctime(&cardExpirationDate)) ;
    MRT_DEBUG("    transactionID = %u\n", transactionID) ;
    MRT_DEBUG("    chargeAmount = $%.2f\n", (double)chargeAmount / 100.0) ;

    struct bookstore_CreditCardCharge_chargeProcessed__EPARAMS cccparams = {
        .resultCode = AP_approved
    } ;
    int result = mrt_PortalSignalDelayedEvent(&bookstore__PORTAL,
        BOOKSTORE_CREDITCARDCHARGE_CLASSID,
        cccID,
        BOOKSTORE_CREDITCARDCHARGE_CHARGEPROCESSED_EVENT,
        &cccparams,
        sizeof(cccparams),
        100) ; // ❶
    assert(result == 0) ;
}
```

❶ Here we must simulate some behavior. We take the simplest approach and approve all charges 100 ms after they are requested. This approach **will not** drive the execution of `CreditCardCharge` instances to all their possible states.

```
<<external operations>>=
externalop void OnlineCustomer_chargeDeclined {
        customerID MRT_InstId
        email {char const *}} {
    MRT_DEBUG("%s:\n", __func__) ;
    MRT_DEBUG("    customerID = %u\n", customerID) ;
    MRT_DEBUG("    email = %s\n", email) ;
}
externalop void OnlineCustomer_chargeApproved {
        customerID MRT_InstId
        email {char const *}} {
    MRT_DEBUG("%s:\n", __func__) ;
    MRT_DEBUG("    customerID = %u\n", customerID) ;
    MRT_DEBUG("    email = %s\n", email) ;
}
externalop void OnlineCustomer_orderReportedDelivered {
        customerID MRT_InstId
        email {char const *}} {
```

```
    MRT_DEBUG("%s:\n", __func__) ;
    MRT_DEBUG("    customerID = %u\n", customerID) ;
    MRT_DEBUG("    email = %s\n", email) ;
}
```

The interaction with the Shipping Company external entity also requires more simulation to drive the domain to where we want it to go. In this case we want to signal an event to assign a tracking number. Later we need to report that we have picked up the package. Finally, we need to report the package delivery.

```
<<external operations>>=
externalop void ShippingCompany_shipmentReadyForPickup {shipmentID MRT_InstId} {
    MRT_DEBUG("%s:\n", __func__) ;
    MRT_DEBUG("    shipmentID = %u\n", shipmentID) ;

    struct bookstore_Shipment_trackingNumberAssigned__EPARAMS tnaparams = {
        .shippingCompany = "FedEx",
        .trackingNumber = "130047332",
    } ;

    int result = mrt_PortalSignalEvent(&bookstore__PORTAL,
        BOOKSTORE_SHIPMENT_CLASSID,
        shipmentID,
        BOOKSTORE_SHIPMENT_TRACKINGNUMBERASSIGNED_EVENT,
        &tnaparams,
        sizeof(tnaparams)) ;
    assert(result == 0) ;

    result = mrt_PortalSignalDelayedEvent(&bookstore__PORTAL,
        BOOKSTORE_SHIPMENT_CLASSID,
        shipmentID,
        BOOKSTORE_SHIPMENT_PICKEDUP_EVENT,
        NULL, 0,
        200) ; // ❶
    assert(result == 0) ;

    time_t now ;
    time(&now) ;
    now += 1 ;
    struct bookstore_Shipment_deliveryConfirmed__EPARAMS dcparams = {
        .timeDelivered = now,
    } ;

    result = mrt_PortalSignalDelayedEvent(&bookstore__PORTAL,
        BOOKSTORE_SHIPMENT_CLASSID,
        shipmentID,
        BOOKSTORE_SHIPMENT_DELIVERYCONFIRMED_EVENT,
        &dcparams,
        sizeof(dcparams),
        1000) ; // ❷
    assert(result == 0) ;
}
```

❶ We say that the shipment was picked up 200 ms after it was declared ready.

❷ Then we say the shipment was delivered 1 s after it was picked up. We're not interested waiting around for a long time so the example can run.

```
<<external operations>>=
externalop void ShippingClerk_shipmentReadyToPack {
        clerkID MRT_InstId
```

```
        shipmentID unsigned} {
    MRT_DEBUG("%s:\n", __func__) ;
    MRT_DEBUG("    clerkID = %u\n", clerkID) ;
    MRT_DEBUG("    shipmentID = %u\n", shipmentID) ;
}
```

## Execution Scenarios

Finally, we are ready to run the translated domain. We will include the code to drive the test scenarios directly in the `main` function. We have chosen to do it this way for simplicity and so that we can use the state of the domain created by one scenario as the starting point for the next one. This will prevent us from having the start afresh and drive the domain to some state just to get into position to start a run. Real world testing would probably take another approach.

Finally, we can show the code for running the execution scenarios.

```
<<test scenarios>>=
<<scenario1>>
<<scenario2>>
<<scenario3>>
<<scenario4>>
```

### Scenario 1

Our first scenario is very simple. We want to create a shopping cart and then decide to change our mind and zero out our selection. This will cause the whole order to be canceled. In order create a shopping cart we must supply a product ID and a quantity. To obtain the product ID, we read the productID from an initial instance of `Product`. In this case, we use the Executable UML book, which our bookstore conveniently stocks.

```
<<scenario1>>=
puts("**** Scenario 1: begin") ;

char const *xuml_productID ;
int result = mrt_PortalGetAttrRef(&bookstore__PORTAL,
    BOOKSTORE_PRODUCT_CLASSID,
    BOOKSTORE_PRODUCT_MBXUML_INSTID,
    BOOKSTORE_PRODUCT_PRODUCTID_ATTRID,
    (void **)&xuml_productID,
    NULL) ;
assert(result == 0) ;
```

```
<<scenario1>>=
puts("---- creating a shopping cart") ;

union {
    MRT_EventParams params ;
    struct bookstore_ShoppingCart_startCart__EPARAMS startcart ;
} cartparams = {
    .startcart = {
        .productID = xuml_productID,
        .quantity = 1,
    }
} ; // ❶

int cartInstID = mrt_PortalCreateAsync(&bookstore__PORTAL,
    BOOKSTORE_SHOPPINGCART_CLASSID,
    BOOKSTORE_SHOPPINGCART_STARTCART_EVENT,
    cartparams.params) ;
assert(cartInstID >= 0) ;
```

❶ Supplying event parameters via the runtime portal functions is somewhat painful. The ability for `C11` to use designated initializers for automatic variables makes it look a little cleaner. A structure is defined for each event that carries parameters. The union insures that any data copied from the parameters argument to the event does not ramble out into uncharted waters.

Since the instance creation is asynchronous, we must process events to cause the transitions.

```
<<scenario1>>=
dispatchEvents() ;
```

We create a helper function to process events until there are none left.

```
<<helper functions>>=
static void dispatchEvents(void)
{
    int max = 100 ; // ❶
    bool didEvent ;
    do {
        didEvent = mrt_ProcessOneEvent() ;
        max-- ;
        assert(max > 0) ;
    } while (didEvent) ;
}
```

❶ Just in case things get out of hand, we want a maximum number of events to process.

Now we will effectively cancel the order by changing the quantity associated with the product selection to zero. To do this, we will need a domain operation to find the product selection that needs to be signaled.

```
<<domain operations>>=
domainop void changeSelectionQuantity {
        cartID MRT_InstId
        productID MRT_InstId
        quantity unsigned} {
    %<ShoppingCart idtoref cartID cart>%
    %<Product idtoref productID product>%
    %<ProductSelection instref selection>%
    %<instance selection foreachRelated cart {R4 ProductSelection}>%
        %<instance selectedProduct findOneRelated selection R4>%
        if (selectedProduct == product) {
            break ;
        }
    %<end>%
    if (selection != NULL) {
        %<instance selection signal changeQuantity quantity quantity>%
    }
}
```

After invoking the domain operation, we must dispatch the events.

```
<<scenario1>>=
puts("---- changing quantity to 0") ;
bookstore_changeSelectionQuantity(
    cartInstID, BOOKSTORE_PRODUCT_MBXUML_INSTID, 0) ;
dispatchEvents() ;

puts("**** Scenario 1: end") ;
```

## Scenario 2

For the second scenario, we will start a new cart and add some products to it. We will also change the quantity of one of the products. Finally, we will check out and purchase the order.

We create the new shopping cart, with the same product selection as our first cart, waiting for the creation event to be dispatched.

```
<<scenario2>>=
puts("**** Scenario 2: begin") ;
puts("---- creating a new cart") ;
cartInstID = mrt_PortalCreateAsync(&bookstore__PORTAL,
    BOOKSTORE_SHOPPINGCART_CLASSID,
    BOOKSTORE_SHOPPINGCART_STARTCART_EVENT,
    cartparams.params) ;
assert(cartInstID >= 0) ;

dispatchEvents() ;
```

We change our minds and decide we really need three copies of the Executable UML book.

```
<<scenario2>>=
puts("---- change quantity to 2 books") ;
bookstore_changeSelectionQuantity(
    cartInstID, BOOKSTORE_PRODUCT_MBXUML_INSTID, 3) ;
dispatchEvents() ;
```

We also decide to buy some music. We must look up the product ID and then add the selection to the cart.

```
<<scenario2>>=
puts("---- adding music to the shopping cart") ;
char const *cmc_productID ;
result = mrt_PortalGetAttrRef(&bookstore__PORTAL,
    BOOKSTORE_PRODUCT_CLASSID,
    BOOKSTORE_PRODUCT_CMC_INSTID,
    BOOKSTORE_PRODUCT_PRODUCTID_ATTRID,
    (void **)&cmc_productID,
    NULL) ;
assert(result == 0) ;

struct bookstore_ShoppingCart_addSelection__EPARAMS addselect = {
    .productID = cmc_productID,
    .quantity = 2, // ❶
} ;
result = mrt_PortalSignalEvent(&bookstore__PORTAL,
    BOOKSTORE_SHOPPINGCART_CLASSID,
    cartInstID,
    BOOKSTORE_SHOPPINGCART_ADDSELECTION_EVENT,
    &addselect,
    sizeof(addselect)) ;
assert(result == 0) ;

dispatchEvents() ;
```

❶ We decide we need two copies of the music.

Finally, we checkout. Since we stubbed out all credit cards to be approved, we will know the order will progress to the `Being Packed and Shipped` state.

```
<<scenario2>>=
puts("---- checking out") ;
struct tm expiration ;
```

```
memset(&expiration, 0, sizeof(expiration)) ;
strptime("10/17", "%m/%y", &expiration) ;
struct bookstore_ShoppingCart_checkOut__EPARAMS coparams = {
    .accountNumber = "5555 6666 7777 8888",
    .billingAddress = "100 E Main St., Anywhere, CO 88888",
    .cardExpirationDate = mktime(&expiration),
    .cardholderName = "Fred Smith",
    .customerEmail = "fsmith@gmail.com",
    .customerName = "Fred Smith",
    .customerPhone = "222-555-1212",
    .shippingAddress = "100 E Main St., Anywhere, CO 88888",
} ;

result = mrt_PortalSignalEvent(&bookstore__PORTAL,
    BOOKSTORE_SHOPPINGCART_CLASSID,
    cartInstID,
    BOOKSTORE_SHOPPINGCART_CHECKOUT_EVENT,
    &coparams,
    sizeof(coparams)) ;
assert(result == 0) ;

dispatchEvents() ;
```

There is a time delay associated with credit card approval.

```
<<scenario2>>=
puts("---- payment approval") ;

pause() ; // ❶
dispatchEvents() ;

puts("**** Scenario 2: end") ;
```

❶ The `pause()` is a POSIX standard function that waits until a signal happens. In our case the signal will be `SIGALRM` indicating that the timer for the delayed event expired. We will use this technique below to keep the flow of code linear as we wait for delayed events to be signaled.

## Scenario 3

For our third scenario, we will let the events unwind in the Shipping subsystem and see how the shipment is packed and made ready for shipment.

```
<<scenario3>>=
puts("**** Scenario 3: begin") ;
puts("---- packing order") ;

for (int i = 0 ; i < 5 ; i++) { // ❶
    pause() ;
    dispatchEvents() ;
}

puts("**** Scenario 3: end") ;
```

❶ To step through the shipment packing requires 5 delayed events to be delivered.

## Scenario 4

For scenario 4, we continue to let the events unwind until we see that the shipment was delivered.

```
<<scenario4>>=
puts("**** Scenario 4: begin") ;
puts("---- delivering shipment") ;

for (int i = 0 ; i < 2 ; i++) {
    pause() ;
    dispatchEvents() ;
}

puts("**** Scenario 4: end") ;
```

### Run Results

Below is the event dispatch trace for all the test scenarios. Intermixed with the trace is some other external operations output and marker output to show the process of the scenarios. Event traces are timestamped and show the type of event dispatched. This is followed by the event details that include the class and instance, event information and state transition information. Events that originate from `?.?` are those signaled outside of a state activity.

```
**** Scenario 1: begin
---- creating a shopping cart
2016-10-25T17:24:46.734.627: Creation: ?.? - startCart -> ShoppingCart ==> 0
2016-10-25T17:24:46.734.660: Transition: ?.? - startCart -> ShoppingCart.0: @ ==>  ↩
    NewOrder
2016-10-25T17:24:46.734.670: Transition: ShoppingCart.0 - addSelection ->  ↩
    ShoppingCart.0: NewOrder ==> AddingSelectiontoOrder
---- changing quantity to 0
2016-10-25T17:24:46.734.695: Transition: ?.? - changeQuantity -> ProductSelection ↩
    .0: NewSelection ==> ChangingQuantity
2016-10-25T17:24:46.734.702: Transition: ProductSelection.0 - removeSelection ->  ↩
    ProductSelection.0: ChangingQuantity ==> RemovingSelection
2016-10-25T17:24:46.734.709: Transition: ProductSelection.0 - cancel ->  ↩
    ShoppingCart.0: AddingSelectiontoOrder ==> CancelingEntireOrder
**** Scenario 1: end
**** Scenario 2: begin
---- creating a new cart
2016-10-25T17:24:46.734.735: Creation: ?.? - startCart -> ShoppingCart ==> 1
2016-10-25T17:24:46.734.742: Transition: ?.? - startCart -> ShoppingCart.1: @ ==>  ↩
    NewOrder
2016-10-25T17:24:46.734.749: Transition: ShoppingCart.1 - addSelection ->  ↩
    ShoppingCart.1: NewOrder ==> AddingSelectiontoOrder
---- change quantity to 2 books
2016-10-25T17:24:46.734.766: Transition: ?.? - changeQuantity -> ProductSelection ↩
    .1: NewSelection ==> ChangingQuantity
---- adding music to the shopping cart
2016-10-25T17:24:46.734.775: Transition: ?.? - addSelection -> ShoppingCart.1:  ↩
    AddingSelectiontoOrder ==> AddingSelectiontoOrder
---- checking out
2016-10-25T17:24:46.734.801: Transition: ?.? - checkOut -> ShoppingCart.1:  ↩
    AddingSelectiontoOrder ==> EstablishingCustomerandVerifyingPayment
2016-10-25T17:24:46.734.812: Creation: ShoppingCart.1 - checkOut -> Order ==> 0
2016-10-25T17:24:46.734.819: Transition: ShoppingCart.1 - checkOut -> Order.0: @  ↩
    ==> EstablishingCustomerandVerifyingPayment
2016-10-25T17:24:46.734.826: Transition: Order.0 - submitCharge -> Order.0:  ↩
    EstablishingCustomerandVerifyingPayment ==> SubmittingCharge
2016-10-25T17:24:46.734.838: Creation: Order.0 - makeCharge -> CreditCardCharge  ↩
    ==> 0
```

```
2016-10-25T17:24:46.734.845: Transition: Order.0 - makeCharge -> CreditCardCharge ↩
    .0: @ ==> RequestingChargeApproval
---- payment approval
2016-10-25T17:24:46.833.875: Transition: ?.? - chargeProcessed -> CreditCardCharge ↩
    .0: RequestingChargeApproval ==> ProcessingCompleted
2016-10-25T17:24:46.833.914: Transition: CreditCardCharge.0 - paymentApproved ->  ↩
    Order.0: SubmittingCharge ==> BeingPackedandShipped
2016-10-25T17:24:46.833.929: Creation: Order.0 - requestShipment -> Shipment ==> 0
2016-10-25T17:24:46.833.936: Transition: Order.0 - requestShipment -> Shipment.0:  ↩
    @ ==> PreparingShipment
2016-10-25T17:24:46.833.944: Transition: Shipment.0 - shipmentReadyToPack ->  ↩
    Warehouse.acme: WaitingforaShipment ==> WaitingforaFreeClerk
2016-10-25T17:24:46.833.951: Transition: Warehouse.acme - clerkFree -> Warehouse. ↩
    acme: WaitingforaFreeClerk ==> AssigningClerktoShipment
2016-10-25T17:24:46.833.959: Transition: Warehouse.acme - clerkAssigned ->  ↩
    ShippingClerk.fs: WaitingforaJob ==> SelectingBooks
2016-10-25T17:24:46.833.969: Transition: Warehouse.acme - clerkAssignedToShipment  ↩
    -> Warehouse.acme: AssigningClerktoShipment ==> WaitingforaShipment
**** Scenario 2: end
**** Scenario 3: begin
---- packing order
2016-10-25T17:24:47.234.027: Transition: ShippingClerk.fs - booksSelected ->  ↩
    ShippingClerk.fs: SelectingBooks ==> PackingBox
2016-10-25T17:24:47.934.180: Transition: ShippingClerk.fs - boxPacked ->  ↩
    ShippingClerk.fs: PackingBox ==> SealingBox
2016-10-25T17:24:48.734.437: Transition: ShippingClerk.fs - boxSealed ->  ↩
    ShippingClerk.fs: SealingBox ==> AttachingShippingLabel
2016-10-25T17:24:49.334.651: Transition: ShippingClerk.fs - shippingLabelAttached  ↩
    -> ShippingClerk.fs: AttachingShippingLabel ==> DeliveringBoxtoLoadingDock
2016-10-25T17:24:49.734.830: Transition: ShippingClerk.fs - boxAtLoadingDoc ->  ↩
    ShippingClerk.fs: DeliveringBoxtoLoadingDock ==> CompletingJob
2016-10-25T17:24:49.734.969: Transition: ShippingClerk.fs - packed -> Shipment.0:  ↩
    PreparingShipment ==> PackedandAwaitingTrackingNumber
2016-10-25T17:24:49.735.053: Transition: ShippingClerk.fs - clerkFree -> Warehouse ↩
    .acme: WaitingforaShipment ==> IG
2016-10-25T17:24:49.735.186: Transition: ?.? - trackingNumberAssigned -> Shipment ↩
    .0: PackedandAwaitingTrackingNumber ==> NumberAssigned
**** Scenario 3: end
**** Scenario 4: begin
---- delivering shipment
2016-10-25T17:24:49.934.171: Transition: ?.? - pickedUp -> Shipment.0:  ↩
    NumberAssigned ==> InTransittoCustomer
2016-10-25T17:24:50.735.467: Transition: ?.? - deliveryConfirmed -> Shipment.0:  ↩
    InTransittoCustomer ==> Delivered
2016-10-25T17:24:50.735.605: Transition: Shipment.0 - orderDelivered -> Order.0:  ↩
    BeingPackedandShipped ==> DeliveredtoCustomer
**** Scenario 4: end
```

## Code Organization

In this section we show the organization of the files that can be tangled from the literate source.

First, this software is copyrighted. It is licensed in the same manner as Tcl itself.

```
<<copyright info>>=
# DO NOT EDIT THIS FILE!
```

```
# THIS FILE IS AUTOMATICALLY GENERATED FROM A LITERATE PROGRAM SOURCE FILE.
#
# This software is copyrighted 2015 - 2016 by G. Andrew Mangogna.
# The following terms apply to all files associated with the software unless
# explicitly disclaimed in individual files.
#
# The authors hereby grant permission to use, copy, modify, distribute,
# and license this software and its documentation for any purpose, provided
# that existing copyright notices are retained in all copies and that this
# notice is included verbatim in any distributions. No written agreement,
# license, or royalty fee is required for any of the authorized uses.
# Modifications to this software may be copyrighted by their authors and
# need not follow the licensing terms described here, provided that the
# new terms are clearly indicated on the first page of each file where
# they apply.
#
# IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR
# DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING
# OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES
# THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF
# SUCH DAMAGE.
#
# THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES,
# INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT.  THIS SOFTWARE
# IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE
# NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS,
# OR MODIFICATIONS.
#
# GOVERNMENT USE: If you are acquiring this software on behalf of the
# U.S. government, the Government shall have only "Restricted Rights"
# in the software and related documentation as defined in the Federal
# Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2).  If you
# are acquiring the software on behalf of the Department of Defense,
# the software shall be classified as "Commercial Computer Software"
# and the Government shall have only "Restricted Rights" as defined in
# Clause 252.227-7013 (c) (1) of DFARs.  Notwithstanding the foregoing,
# the authors grant the U.S. Government and others acting in its behalf
# permission to use and distribute the software in accordance with the
# terms specified in this license.
```

Each *root chunk* is described in a section below.

## Source Code

The organization of the literate program chuncks that make up the `micca` source is show below. Note that for simplicity, we include the `main` function as part of the generated epilogue.

```
<<simplebookstore.micca>>=
<<copyright info>>

domain bookstore {
    <<Product Specification Subsystem>>
    <<Ordering Subsystem>>
    <<Shipping Subsystem>>
    <<type aliases>>
    <<external operations>>
    <<domain operations>>
    interface {
        #include <time.h>
    }
```

```
    prologue {
        #include <unistd.h>
    }
    epilogue {
        #ifdef MRT_HARNESS
        <<harness main>>
        #else
        <<non-harness main>>
        #endif /* MRT_HARNESS */
    }
}
```

```
<<non-harness main>>=
<<helper functions>>
int
main(
    int argc,
    char *argv[])
{
    mrt_Initialize() ;
    <<test scenarios>>

    return EXIT_SUCCESS ;
}
```

```
<<harness main>>=
extern MTH_DomainHarness const bookstore__HARNESS ;
int
main(
    int argc,
    char *argv[])
{
    mrt_Initialize() ;
    if (!mth_Initialize()) {
        fprintf(stderr, "mth_Initialize() failed\n") ;
        exit(EXIT_FAILURE) ;
    }

    mth_RegisterDomain(&bookstore__HARNESS) ;

    mth_Run() ;

    return EXIT_SUCCESS ;
}
```

### Initial Instance Population

We keep the initial instance population in a separate file. This makes it much easier to have different populations for different testing and demonstration scenarios.

```
<<sbs_population.micca>>=
<<copyright info>>

population bookstore {
    <<Product Specification Population>>
    <<Ordering Population>>
    <<Shipping Population>>
}
```

## Literate Programming

The source for this document conforms to asciidoc syntax. This document is also a literate program. The source code for the implementation is included directly in the document source and the build process extracts the source that is then given to the `micca` program. This process is known as *tangle*ing. The program, `atangle`, is available to extract source code from the document source and the `asciidoc` tool chain can be used to produce a variety of different output formats, although PDF is the intended choice.

The goal of a literate program is to explain the logic of the program in an order and fashion that facilitates human understanding of the program and then *tangle* the document source to obtain the code in an form suitable for processing. Briefly, code is extracted from the literate source by defining a series of *chunks* that contain the source. A chunk is *defined* by including its name as:

```
<<chunk name>>=
```

The trailing = sign denotes a definition. A chunk definition ends at the end of the source block or at the beginning of another chunk definition. A chunk may be *referenced* from within a chunk definition by using its name without the trailing = sign, as in:

```
<<chunk definition>>=
    <<chunk reference>>
```

Chunk names are arbitrary strings. Multiple definitions with the same name are simply concatenated in the order they are encountered. There are one or more *root chunks* which form the conceptual tree for the source files that are contained in the literate source. By convention, root chunks are named the same as the file name to which they will be tangled. Tangling is then the operation of starting at a root chunk and recursively substituting the definition for the chunk references that are encountered.

For readers that are not familiar with the literate style and who are adept at reading source code directly, the chunks definitions and reordering provided by the tangle operation can be a bit disconcerting at first. You can, of course, examine the tangled source output, but if you read the program as a document, you will have to trust that the author managed to arrange the chunk definitions and references in a manner so that the tangled output is acceptable to the remaining processing.

# Index

**A**
Artist, 7, 30
Author, 5, 27
Authorship, 5, 28
AuthorshipSequence, 5, 28

**B**
Book Product, 28
BookProduct, 6

**C**
Category Group, 7, 30
Classes
    Artist, 7
    Author, 5
    Authorship, 5
    AuthorshipSequence, 5
    BookProduct, 6
    Category Group, 7
    Computer Software, 8
    Credit Card Charge, 15
    Customer, 8
    Delivery, 19
    Off Duty Clerk, 25
    Order, 9
    Order Placement, 12
    Packing, 20
    Past Packing, 20
    Performance Credit, 7
    Performance Credit Sequence, 7
    Platform, 8
    Product, 4
    Product Category, 7
    Product Selection, 12
    Publisher, 5
    Recording, 7
    Recording Format, 8
    Recording Product, 6
    Shipment, 17
    Shipment Item, 25
    Shipping Clerk, 22
    Shipping Company, 20
    ShoppingCart, 13
    Software Product, 6
    Special Order Product, 6
    Stock Clerk, 25
    Stocked Product, 6
    Warehouse, 20
    Warehouse Clerk, 22
Computer Software, 8, 31
Credit Card Charge, 15
Customer, 8

**D**
Delivery, 19, 32

**O**
Off Duty Clerk, 25, 34
Order, 9
Order Placement, 12
Ordering
    Classes
        Credit Card Charge, 15
        Customer, 8
        Order, 9
        Order Placement, 12
        Product Selection, 12
        ShoppingCart, 13
    Relationships
        R10, 9
        R4, 12
        R5, 12
        R7, 15
        R8, 16
    State Model
        Credit Card Charge, 15
        Order, 9
        Product Selection, 12
        Shopping Cart, 14

**P**
Packing, 20, 32
Past Packing, 20, 32
Performance Credit, 7, 30
Performance Credit Sequence, 7, 31
Platform, 8, 31
Population
    Artist, 30
    Author, 27
    Authorship, 28
    AuthorshipSequence, 28
    Book Product, 28
    Category Group, 30
    Computer Software, 31
    Delivery, 32
    Off Duty Clerk, 34
    Packing, 32
    Past Packing, 32
    Performance Credit, 30
    Performance Credit Sequence, 31
    Platform, 31
    Product Category, 30
    Publisher, 27
    Recording, 30
    Recording Format, 31
    Recording Product, 29
    Shipment, 32
    Shipment Item, 34

Shipping Clerk, 34
Shipping Company, 33
Software Product, 29
Special Order Product, 29
Stock Clerk, 34
Stocked Product, 29
Warehouse, 33
Warehouse Clerk, 33
Product, 4
Product Category, 7, 30
Product Selection, 12
Product Specification
Classes
Artist, 7
Author, 5
Authorship, 5
AuthorshipSequence, 5
BookProduct, 6
Category Group, 7
Computer Software, 8
Performance Credit, 7
Performance Credit Sequence, 7
Platform, 8
Product, 4
Product Category, 7
Publisher, 5
Recording, 7
Recording Format, 8
Recording Product, 6
Software Product, 6
Special Order Product, 6
Stocked Product, 6
Population
Artist, 30
Author, 27
Authorship, 28
AuthorshipSequence, 28
Book Product, 28
Category Group, 30
Computer Software, 31
Performance Credit, 30
Performance Credit Sequence, 31
Platform, 31
Product Category, 30
Publisher, 27
Recording, 30
Recording Format, 31
Recording Product, 29
Software Product, 29
Special Order Product, 29
Stocked Product, 29
Relationships
R1, 4
R11, 4
R12, 4
R14, 7
R15, 4
R16, 6
R17, 7
R18, 7
R19, 6
R2, 5
R20, 6
R3, 5
Publisher, 5, 27

**R**
R1, 4
R10, 9
R11, 4
R12, 4
R14, 7
R15, 4
R16, 6
R17, 7
R18, 7
R19, 6
R2, 5
R20, 6
R21, 19
R22, 20
R23, 20
R24, 17
R25, 22
R27, 22
R3, 5
R4, 12
R5, 12
R6, 17
R7, 15
R8, 16
R9, 25
Recording, 7, 30
Recording Format, 8, 31
Recording Product, 6, 29
Relationships
R1, 4
R10, 9
R11, 4
R12, 4
R14, 7
R15, 4
R16, 6
R17, 7
R18, 7
R19, 6
R2, 5
R20, 6
R21, 19
R22, 20
R23, 20
R24, 17
R25, 22
R27, 22

R3, 5
R4, 12
R5, 12
R6, 17
R7, 15
R8, 16
R9, 25

**S**

Shipment, 17, 32
Population
Delivery, 32
Packing, 32
Past Packing, 32
Shipment, 32
Shipment Item, 34
Shipment Item, 25, 34
Shipping
Classes
Delivery, 19
Off Duty Clerk, 25
Packing, 20
Past Packing, 20
Shipment, 17
Shipment Item, 25
Shipping Clerk, 22
Shipping Company, 20
Stock Clerk, 25
Warehouse, 20
Warehouse Clerk, 22
Population
Off Duty Clerk, 34
Shipping Clerk, 34
Shipping Company, 33
Stock Clerk, 34
Warehouse, 33
Warehouse Clerk, 33
Relationships
R21, 19
R22, 20
R23, 20
R24, 17
R25, 22
R27, 22
R6, 17
R9, 25
State Model
Shipment, 17
Shipping Clerk, 23
Warehouse, 21
Shipping Clerk, 22, 23, 34
Shipping Company, 20, 33
Shopping Cart, 14
ShoppingCart, 13
Software Product, 6, 29
Special Order Product, 6, 29
State Model
Credit Card Charge, 15
Order, 9
Product Selection, 12
Shipment, 17
Shipping Clerk, 23
Shopping Cart, 14
Warehouse, 21
Stock Clerk, 25, 34
Stocked Product, 6, 29

**W**

Warehouse, 20, 21, 33
Warehouse Clerk, 22, 33