Simple Bookstore Model Translation

A Translation of the Case Study

Legal Notices and Information

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.1	November 2, 2015	Initial coding started.	GAM
1.0	November 9, 2015	First release.	GAM
1.1	November 10, 2015	Adapted to use new features of rosea version 1.5.	GAM
1.2	November 19, 2015	Adapted to use new features of rosea version 1.6.	GAM
1.3	November 27, 2015	Minor copy edits.	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

Book Product Class . . . 6

Recording Product Class . . . 7

Software Product Class . . . 7

Special Order Product Class . . . 7

Stocked Product Class . . . 8

Product Category Class . . . 8

Recording Class . . . 8

Artist Class . . . 8

Performance Credit Class . . . 8

Recording Format Class . . . 9

Platform Class . . . 9

Computer Software Class . . . 9

Ordering Subsystem 9

Customer Class . . . 10

Order Class . . . 10

Order Class State Model . . . 10

Product Selection Class . . . 12

Product Selection Class State Model . . . 13

ShoppingCart Class . . . 14

Credit Card Charge Class . . . 16

Credit Card Charge Class State Model . . . 16

Shipping Subsystem **17**
Shipment Class . . . 18
Shipment Class State Model . . . 19
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 . . . 22
Stock Clerk Class . . . 25
Off Duty Clerk Class . . . 25
Shipment Item Class . . . 25

Product Specification Subsystem Population **26**
Product Population . . . 26
Publisher Population . . . 27
Author Population . . . 28
Authorship Population . . . 28
Book Product Population . . . 29
Recording Product Population . . . 29
Software Product Population . . . 29
Special Order Product Population . . . 29
Stocked Product Population . . . 30
Product Category Population . . . 30
Recording Population . . . 30
Artist Population . . . 31
Performance Credit Population . . . 31
Recording Format Population . . . 31
Platform Population . . . 31
Computer Software Population . . . 32

Ordering Subsystem Population **32**

Shipping Subsystem Population **32**
Shipping Company Population . . . 32
Warehouse Population . . . 33
Warehouse Clerk Population . . . 33
Shipping Clerk Population . . . 33
Stock Clerk Population . . . 33
Off Duty Clerk Population . . . 34

External Entities **34**

Execution Scenarios **35**

Scenario 1 37

Scenario 2 39

Scenario 3 43

Scenario 4 45

Code Organization **46**

Source Code 46

Initial Instance Population 47

Literate Programming **47**

Index **48**

List of Figures

1 Scenario 1 Sequence Diagram 39
2 Scenario 2 Sequence Diagram 42
3 Scenario 3 Sequence Diagram 44
4 Scenario 4 Sequence Diagram 45

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 Tcl implementation language using the `rosea` package. 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 `rosea` has a manual component. The model diagrams are transcribed into a domain specific language (DSL) that is provided by `rosea`. The classes, relationships and state models are defined in a declarative manner. The state activities must be translated from action language into Tcl using procedures provided by `rosea` to implement the model level actions. For example, action language statements to select related instances are translated into invocations of the `findRelated` procedure. Generally, action language statements translate one-for-one into lines of Tcl code and the correspondence is direct if you understand the conventions of Tcl programming. Once transcribed into the `rosea` DSL, a code generation phase is invoked. Unlike translation for compiled languages, this code generation does not produce any files. Tcl is an interpreted language and so the commands and data required by the run time component of `rosea` are simply created *in situ*. Once generation is completed, we can exercise the resulting program by signaling events or invoking any operations that are provided.

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 `rosea`. 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. The results of running the scenarios is given in two forms. `Rosea` has many capabilties to deal with execution traces. First, is a textual log of the event dispatch. Second, is a sequence diagram. The sequence diagram is just another view on the event dispatch data, organized in the usual UML sequence diagram form.

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 `rosea` 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 `rosea` package and its underlying Tcl language constructs 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 `rosea` 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 and so no external programming interface to the domain is provided.

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 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 only a first step. When building a system of multiple domains, itegration 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 dependent 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 online bookstore system.

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, to translate using `rosea` requires them to be reinserted into the implementation. The `rosea` translation is very relation oriented and referential attributes are an integral part of the the relational model of data. `Rosea` also requires each class to have an identifier. We will point out where we have added attributes to deal with the particulars of `rosea`. The added benefit of supplying the referential attributes is that `rosea` performs referential integrity checks and insures that the model does not violate any of the constraints expressed by the relationships and identifiers.

`Rosea` does not support the notion of derived attributes. We have a couple of options to deal with them:

1. Delete the derived attributes and add code into the translation to compute them whenever they are read.
2. Keep the derived attribute and add the redundant code required to keep them up to date when any of the dependent attributes change.

We have chosen the later approach in this translation since it makes access to the class diagram data from outside the domain much easier. We should also note that the model as presented in the appendix does not state what the composition of 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 `rosea`. Action language allows one to create a class instance without supplying all the attributes a value at the time the creation operation is invoked. Since `rosea` is relational in its orientation and since a class instance in `rosea` is mapped to a tuple in a relational variable, all the values of the attributes of an instance must be given a valid value for the data type of the attribute at the time of the creation operation. 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 `rosea` 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. Assigning attributes piecemeal in this regime is easily done. However, for software architectures targeted at databases or, like

[1] as opposed to a procedural way

rosea, 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.

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 rosea, 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 exceptions to this are associations with an associative class and generalization relationships. For associative classes, we define the corresponding association with them and we define the generalization along with the superclass. This ordering is used to clarify the translation's correspondence with the book text. Rosea 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 rosea 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

Rosea requires attributes to be given an data type. That data type is, of course, an implementation data type and must be one of the Tcl data types supported by TclRAL. It may seem strange to discuss types in a dynamically typed language such as Tcl where, at the script level, all values have a string representation. Tcl uses a form of *duck typing*. Values are converted to the type required by the operation and should they fail to be converted an error is thrown. In Tcl, string is the universal type. Attributes with types other than string will be converted to that type when they are updated into an instance. This is an integrity check. If you declare an attribute as int, then whatever value you assign to the attribute must be convertable to an int because presumably you intend to perform some arithmetic on the value. Rosea prevent assignment of values to attributes that cannot possibly be converted to the correct type.

That said, data types like int are usually too permissive. It is usually a subset of values of the type that are desired. Rosea does allow an expression to be evaluated on an attribute value as a check that it is reasonable. We will use this feature in several situations.

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

arbitrary_id
: As of version 1.5, rosea supports both system supplied sequential integer default values and command generated default values. We will make use of sequential integer default values to create arbitrary identifying attribute values.

Count
: A simple check on an int type to be greater than zero is used.

Money
: We'll hold money as an int in units of cents.

date
: Here we are using formatted string as dates. Another choice would be to use the number of seconds since the epoch. The formatted string are easier to read. The seconds representation is easier to compute on. The model does not appear to perform any date computations.

[2] Well databases can handle it with some awkward NULL value nonsense, but rosea has no notion of a NULL and none of the three values logic required to deal with NULL values.

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.

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>>
<<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>>
<<Recording Class Definition>>
<<Artist Class Definition>>
<<Performance Credit Class Definition>>
<<Recording Format Class Definition>>
<<Platform Class Definition>>
<<Computer Software Class Definition>>
```

Product Class

```
<<Product Class Definition>>=
class Product {
    attribute productID string -id 1
    attribute productName string
    attribute copyright string -check {$copyright > 0}
    attribute unitPrice int -check {$unitPrice > 0}
    attribute description string
    attribute website string
    attribute currentlyAvailable boolean

    attribute categoryID int ; # ❶
    reference R15 ProductCategory -link categoryID

    attribute groupCode string ; # ❷
    attribute publisherCode string
    reference R1 Publisher -link groupCode -link publisherCode

}
```

❶ Here we introduce a referential attribute to Product Category to realize `R15`.

❷ Again, the need to be precise about how `R1` is realized requires referential attributes. In this case there are two referential attributes corresponding to the two identifying attributes of Publisher. We will generally follow the convention to name referential attributes that we are supplying to be the same as the identifying attribute they reference. It is not always

possible to do that, but we feel it clarifies matters especially as we are reintroducing the referential attributes into the model.

```
<<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 string -id 1 ; # ❶
    attribute publisherCode string -id 1
    attribute name string
    attribute address string
    attribute website string
}
```

❶ Per the model, there are two identifying attributes.

Author Class

```
<<Author Class Definition>>=
class Author {
    attribute name string -id 1 ; # ❶
    attribute website string
    attribute email string
}
```

❶ The implication of using `name` as an identifier is that authors cannot have the same name. Probably not true in the *real world*.

Authorship Class

Authorship is an association class. From the relational view, association classes have referential attributes to refer to both the participating classes. Because `R2` is a many-to-many association, those referential attributes also constitute an identifier. It is also 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 string -default {}
    attribute postCredit string -default {}

    attribute name string -id 1 ; # ❶
    reference R2 Author -link name
    attribute productID string -id 1
    reference R2 BookProduct -link productID

    attribute previousName string ; # ❷
    attribute previousProductID string
    reference R3 Authorship\
```

```
        -link {previousName name}\
        -link {previousProductID productID} ; # ❸
}
```

❶ The association class has referential attributes that refer to both participating classes, in this case `Author` and `BookProduct`.

❷ `R3` is a reflexive association so we must introduce referential attributes to realize the association. Since there are two identifying attributes there will be two referential attributes.

❸ The referential attribute linkage can be a bit confusing at first. You can think of this as a form of *back linkage*. As one Authorship instance follows another, the following instance has referential attributes that refer back to the previous instance in the ordering. Hence the names of the referential attributes. The start of the ordering is that instance which has the empty string value for the `previousName` and `previousProductID` attributes.

```
<<Authorship Class Definition>>=
association R2 BookProduct 1..*--0..* Author -associator Authorship
association R3 Authorship 0..1--0..1 Authorship\
    -path {name previousName productID previousProductID} ; # ❶
```

❶ Again, reflexive relationships require a bit more specification. Because the source and target of the association are the same class, there is an ambiguity as to which direction we are traversing. The `-path` option defines the forward direction of the relationship. In this case, forward is semantically the same as the model annotation, **follows**.

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. Another alternative is to realize this relationship using an attribute that is ordered, such as a sequential integer. Then, the ordered set of authors of a book could be found by selecting on the `productID` and sorting by the ordering attribute. Depending upon the way Authorship is being used might favor one approach over another. Here we just stick with the straight forward referential attributes and contrive to populate the model to achieve the order we want.

Book Product Class

```
<<Book Product Class Definition>>=
class BookProduct {
    attribute bookNumber string
    attribute titleCode string
    attribute title string
    attribute subtitle string

    attribute productID string -id 1 ; # ❶
    reference R11 Product -link productID
}
```

❶ All subclasses of a generalization have a referential attribute back to the superclass of the generalization. Since there is a one-to-one correspondence between a subclass and its superclass, the referential attribute is also an identifying attribute. The fact that an attribute can serve many roles happens often and is another facet of the ability of referential attributes and identity constraints to constraint the instance attribute values properly.

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 string

    attribute productID string -id 1 ; # ❶
    reference R11 Product -link productID

    attribute recordingID int -id 2 ; # ❷
    attribute formatID string -id 2
    reference R16 Recording -link recordingID
    reference R16 RecordingFormat -link formatID
}
```

❶ Here we introduce the referential attribute for the generalization.

❷ Here are the referential attributes for the role of `RecordingProduct` as an associator class.

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

Software Product Class

```
<<Software Product Class Definition>>=
class SoftwareProduct {
    attribute productVersion string

    attribute productID string -id 1
    reference R11 Product -link productID

    attribute softwareID string
    reference R20 ComputerSoftware -link softwareID

    attribute platformID string
    reference R19 Platform -link platformID
}
```

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

Special Order Product Class

```
<<Special Order Product Class Definition>>=
class SpecialOrderProduct {
    attribute specialOrderInstructions string
    attribute daysToDeliver int -check {$daysToDeliver >= 0}

    attribute productID string -id 1
    reference R12 Product -link productID
}
```

Stocked Product Class

```
<<Stocked Product Class Definition>>=
class StockedProduct {
    attribute quantityOnHand int -check {$quantityOnHand >= 0}
    attribute reorderThreshold int -check {$reorderThreshold >= 0}
    attribute reorderIncrement int -check {$reorderIncrement >= 0}
    attribute reorderInstructions string

    attribute productID string -id 1
    reference R12 Product -link productID
}
```

Product Category Class

```
<<Product Category Class Definition>>=
class ProductCategory {
    attribute categoryID int -id 1 -system 100
    attribute categoryName string

    attribute parentCategoryID int
    reference R14 ProductCategory -link {parentCategoryID categoryID}
}
```

```
<<Product Category Class Definition>>=
association R14 ProductCategory 0..*--0..1 ProductCategory
```

Recording Class

```
<<Recording Class Definition>>=
class Recording {
    attribute recordingID int -id 1 -system 1 ; # (1)
    attribute title string
}
```

(1) The model does not define an identifier for this class. A simple title is unlikely to be sufficient to identify all the recording in the store, so we have introduced an arbitrary identifier here.

Artist Class

```
<<Artist Class Definition>>=
class Artist {
    attribute artistID int -id 1 -system 1
    attribute artistName string
}
```

Performance Credit Class

```
<<Performance Credit Class Definition>>=
class PerformanceCredit {
    attribute role string -default {}
```

```
    attribute recordingID int -id 1
    attribute artistID int -id 1
    reference R17 Recording -link recordingID
    reference R17 Artist -link artistID

    attribute prevRecordingID int
    attribute prevArtistID int
    reference R18 PerformanceCredit\
        -link {prevRecordingID recordingID}\
        -link {prevArtistID artistID}
}
```

```
<<Performance Credit Class Definition>>=
association R17 Recording 1..*--1..* Artist -associator PerformanceCredit
association R18 PerformanceCredit 0..1--0..1 PerformanceCredit\
    -path {recordingID prevRecordingID artistID prevArtistID}
```

Recording Format Class

```
<<Recording Format Class Definition>>=
class RecordingFormat {
    attribute formatID string -id 1
    attribute formatName string
}
```

Platform Class

```
<<Platform Class Definition>>=
class Platform {
    attribute platformID string -id 1
    attribute platformName string
}
```

Computer Software Class

```
<<Computer Software Class Definition>>=
class ComputerSoftware {
    attribute softwareID string -id 1
    attribute title string
}
```

Ordering Subsystem

The Ordering Subsystem consist of the following classes.

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

Customer Class

```
<<Customer Class Definition>>=
class Customer {
    attribute email string -id 1
    attribute name string
    attribute shippingAddress string
    attribute phone string
    attribute purchasesMade int -default 0 ; # ❶
}
```

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 int -id 1 -system 100
    attribute dateOrderPlaced string -default {}
    attribute totalValue int
    attribute recipient string
    attribute deliveryAddress string
    attribute contactPhone string

    attribute cartID int -default {} ; # ❶
    reference R10 ShoppingCart -link cartID

    attribute email string -default {} ; # ❷
    reference R5 Customer -link email

    statemodel {
        <<Order State Model>>
    }
}
```

❶ R10 is conditional, so when we create an instance of Order we will start with it being unrelated to any `ShoppingCart`.

❷ Likewise, with R5 being conditional.

```
<<Order Class Definition>>=
association R10 Order 0..1--1 ShoppingCart
association R5 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.

Order Class State Model

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

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

```
<<Order State Model>>=
state EstablishingCustomerandVerifyingPayment {
        cart accountNumber billingAddress cardExpirationDate
        cardholderName customerEmail} {
    R10 link $self $cart

    set customer [Customer findById email $customerEmail]
    assignAttribute $self recipient deliveryAddress contactPhone
    if {[isEmptyRef $customer]} {
        set customer [Customer create\
            email $customerEmail\
            name $recipient\
            shippingAddress $deliveryAddress\
            phone $contactPhone
        ]
    } else {
        updateAttribute $customer\
            name $recipient\
            shippingAddress $deliveryAddress\
            phone $contactPhone
    }

    R5 link $self $customer

    updateAttribute $self dateOrderPlaced [clock format [clock seconds]]

    signal $self submitCharge $accountNumber $billingAddress\
        $cardExpirationDate $cardholderName
}

transition EstablishingCustomerandVerifyingPayment - submitCharge ->\
    SubmittingCharge
```

The translation of the state activity to Tcl is quite direct. Action language `relate` statements correspond to the `link` subcommand for the relationship involved, in this case `R10`. We also see the differences that result from creating having to supply the values of all the attributes of the created instance. Using `rosea`, we must make a clear distinction between creating the instance and updating the attributes of an existing instance.

```
<<Order State Model>>=
state SubmittingCharge {accountNumber billingAddress cardExpirationDate
        cardholderName} {
    CreditCardCharge createasync makeCharge [list $self]\
        accountNumber $accountNumber\
        cardholderName $cardholderName\
        billingAddress $billingAddress\
        cardExpirationDate $cardExpirationDate\
        chargeAmount [readAttribute $self totalValue] ; # ❶
}

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 `[list $self]` argument is the parameter of the creation event itself, *i.e.* the instance reference of an order is a parameter to the `makeCharge` event.

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 Tcl.

```
<<Order State Model>>=
state PaymentNotApproved {} {
    set customer [findRelated $self R5]
    # generate chargeDeclined(customerEmail: customer.email)
    #       to EE_OnlineCustomer
    EE_OnLineCustomer::chargeDeclined [readAttribute $customer email] ; # ❶
}

transition PaymentNotApproved - subCharge -> SubmittingCharge
```

❶ The action language text from the model is shown on the line immediately above. `Rosea` does not have any direct support for generating events to external entities. Instead, we create explict bridge operations, in this case `EE_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 `rosea` 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 {} {
    set customer [findRelated $self R5]
    EE_OnLineCustomer::chargeApproved [readAttribute $customer email]

    assignAttribute $self recipient deliveryAddress contactPhone
    Shipment createasync requestShipment [list $self]\
        recipient $recipient\
        deliveryAddress $deliveryAddress\
        contactPhone $contactPhone
}
transition BeingPackedandShipped - orderDelivered -> DeliveredtoCustomer

state DeliveredtoCustomer {} {
    set customer [findRelated $self R5] ; # ❶
    EE_OnLineCustomer::orderReportedDelivered [readAttribute $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 and any attempt to link an order to multiple `Customers` will fail referential integrity checks. So, might we want make the test anyway just to be sure? Again, the answer is no. If somehow the `customer` instance reference does not refer to exactly one instance then the `readAttribute` command will throw an error (as documented in the manual for that command). The lesson here is that superfluous multiplicity tests are not necessary and only serve to clutter the code. The referential integrity checks performed by `rosea` go a long way to insure correct operation.

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 terminal state. The state, `RemovingSelection`, is terminal and the instance is deleted after executing the state activity for that state.

```
<<Product Selection Class Definition>>=
class ProductSelection {
    attribute quantity int -default 0 -check {$quantity >= 0}
    attribute unitPriceOfSelection int -default 0\
            -check {$unitPriceOfSelection >= 0}
    attribute selectionValue int -default 0 -check {$selectionValue >= 0}

    attribute productID string -id 1 -default {} ; # ❶
    attribute cartID int -id 1 -default 0
    reference R4 Product -link productID
    reference R4 ShoppingCart -link cartID

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

❶ Note the referential attributes needed for the class association and those attributes form an identifier.

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

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 {cartID productID quantity} {
    set product [Product findById productID $productID]
    updateAttribute $self\
        productID $productID\
        cartID $cartID\
        unitPriceOfSelection [readAttribute $product unitPrice] ; # ❶

    signal $self changeQuantity $quantity
}
transition NewSelection - changeQuantity -> ChangingQuantity
```

❶ So this is a bit tricky here. If an instance of `ProductSelection` is created asynchronously, then this state actvity is executed. We need to link up the references to the participating instances of `Product` and `ShoppingCart`. We do that by updating the `productId` and `cartID` referential attributes. However, synchronous creation happens as a side effect of linking instances of `Product` and `ShoppingCart` across R4. In that case, the referential attributes are set by the `link` operation, the instance is placed in the `NewSelection` state and the above state activity is **not** executed. This also implies that the synchonous creation must supply values for the `quantity`, `unitPriceOfSelection` and `selectionValue` attributes. In this case the attributes have been given default values so we don't really have to supply them unless they are to be different from the default. Okay, I said it was a bit tricky.

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 what to do? Rather than signaling an event to the `ShoppingCart`, we will invoke an instance based operation on it. This means we do not have to modify the state diagram and it will give us an opportunity to update the `totalValue` derived attribute. See the ShoppingCart for the definition of the operation.

```
<<Product Selection State Model>>=
state ChangingQuantity {quantityOfSelection} {
    withAttribute $self quantity unitPriceOfSelection selectionValue {
        set quantity $quantityOfSelection
        set selectionValue [expr {$quantity * $unitPriceOfSelection}]
    }
    if {$quantity == 0} {
        signal $self removeSelection
    } else {
        set cart [findRelated $self ~R4]
        instop $cart cartUpdated
    }
}
transition ChangingQuantity - removeSelection -> RemovingSelection

state RemovingSelection {} {
    set cart [findRelated $self ~R4]
    R4 unlink $self
    set remainingProduct [findRelated $cart R4]
    if {[isEmptyRef $remainingProduct]} {
        signal $cart cancel
    } else {
        instop $cart cartUpdated
    }
}
terminal RemovingSelection
```

ShoppingCart Class

```
<<Shopping Cart Class Definition>>=
class ShoppingCart {
    attribute cartID int -id 1 -system 1
    attribute totalValue int -default 0 -check {$totalValue >= 0}

    statemodel {
        <<Shopping Cart State Model>>
    }

    instop cartUpdated {} {
        updateAttribute $self totalValue [tcl::mathop::+ {*}[pipe {
                findRelated $self {R4 ProductSelection} |
                deRef ~ selectionValue |
                relation list ~ selectionValue
            }]]
    } ; # ❶
}
```

❶ The intent here is much simpler than the code looks. We want to update the `totalValue` attribute to be the sum of the values of the `selectionValue` attribute for all the instances of `ProductSelection` related to this cart. How we get there is a bit terse. The `tcl::mathop::+` command will compute the sum all of its arguments. So we need each `selectionValue` attribute value to be an argument to the + command. We gather all the `selectionValue` attribute values by:

1. finding all the related `ProductSelection` instances.
2. dereference the `selectionValue` attribute from those instances. This yields a relation value where each tuple has a `selectionValue` attribute.
3. creating a Tcl list from the set of `selectionValue` attributes.

   The `pipe` command is a control structure command that sequences the three commands taking the result of one command and substituting it as an argument to the next command where the `~` placeholder is located. The result of the `pipe` command is the Tcl list generated by the final `relation list` command and the `{*}` operator takes that list and makes it into individual arguments to the `tcl::mathop::+` command. This sequence could have been written in other ways, of course. The usage here shows some of the expressive power of the underlying relational algebra but comes with the cognitive burden of understanding exactly how those commands work.

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

state NewOrder {productID quantity} {
    signal $self addSelection $productID $quantity
}
transition NewOrder - addSelection -> AddingSelectiontoOrder

state AddingSelectiontoOrder {productID quantity} {
    set product [Product findById productID $productID]
    set unitprice [readAttribute $product unitPrice]
    set selectionvalue [expr {$unitprice * $quantity}]
    set newSelection [R4 link $self $product\
        quantity $quantity\
        unitPriceOfSelection $unitprice\
        selectionValue $selectionvalue\
    ] ; # ❶
    withAttribute $self totalValue {
        incr totalValue $selectionvalue
    }
}
transition AddingSelectiontoOrder - addSelection -> AddingSelectiontoOrder
transition AddingSelectiontoOrder - cancel -> CancelingEntireOrder
transition AddingSelectiontoOrder - checkOut ->\
        EstablishingCustomerandVerifyingPayment
```

❶ Linking `R4` results in the synchronous creation of an instance of `ProductSelection`. As discussed above, we must supply the values of all the non-referential attributes of the associative class. The setting of the value of the referential attributes is handled by the `R4 link` command.

```
<<Shopping Cart State Model>>=
state CancelingEntireOrder {} {
    set selections [findRelated $self {R4 ProductSelection}]
    R4 unlink $selections
}
terminal CancelingEntireOrder ; # ❶
```

❶ The `CancelingEntireOrder` is a terminal state and the instance will be automatically deleted when the state activity finishes.

```
<<Shopping Cart State Model>>=
state EstablishingCustomerandVerifyingPayment {
        accountNumber billingAddress cardExpirationDate
        cardholderName customerEmail customerName customerPhone
        shippingAddress} {
```

```
    Order createasync checkOut [list $self $accountNumber\
        $billingAddress $cardExpirationDate $cardholderName\
        $customerEmail]\
        totalValue [readAttribute $self totalValue]\
        recipient $customerName\
        deliveryAddress $shippingAddress\
        contactPhone $customerPhone
}
```

Credit Card Charge Class

```
<<Credit Card Charge Class Definition>>=
class CreditCardCharge {
    attribute chargeID int -id 1 -system 100
    attribute accountNumber string
    attribute cardholderName string
    attribute billingAddress string
    attribute cardExpirationDate string
    attribute dateChargeMade string -default {}
    attribute chargeAmount int
    attribute approvalCode string -default pending -check {$approvalCode in
            {approved overLimit noAccount connectionFailed
            accountDataMismatch expired pending}
        }

    attribute attemptOrderID int -default {}
    reference R7 Order -link {attemptOrderID orderID}

    attribute paidOrderID int -default {}
    reference R8 Order -link {paidOrderID orderID}

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

```
<<Credit Card Charge Class Definition>>=
association R7 CreditCardCharge 0..*--1 Order
association R8 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.

Credit Card Charge Class State Model

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

state RequestingChargeApproval {order} {
    R7 link $self $order ; # ❶
    updateAttribute $self dateChargeMade [clock format [clock seconds]]

    withAttribute $self accountNumber billingAddress cardholderName\
            cardExpirationDate chargeAmount {
        EE_CreditCardCompany::requestChargeApproval\
```

```
            $accountNumber\
            $billingAddress\
            $cardholderName\
            $cardExpirationDate\
            [readAttribute $self chargeID]\
            $chargeAmount
    }

    delaysignal 60000 $self chargeProcessingNotCompleted
}
transition RequestingChargeApproval - chargeProcessed -> ProcessingCompleted
transition RequestingChargeApproval - chargeProcessingNotCompleted ->\
    DeclineForTimeout
```

❶ The action language in this state and all the other states for this class keep refering to relationship, `R8`. Clearly, `R7` is the one that is intended.

```
<<Credit Card Charge State Model>>=
state ProcessingCompleted {resultCode} {
    set order [findRelated $self R7] ; # ❶
    canceldelayed $self $order chargeProcessingNotCompleted ; # ❷

    updateAttribute $self approvalCode $resultCode
    if {$resultCode eq "approved"} {
        R8 link $self $order ; # ❸
        signal $order paymentApproved
    } else {
        signal $order 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 link `R8` to establish the charge that actually paid for the order.

```
<<Credit Card Charge State Model>>=
state DeclineForTimeout {} {
    updateAttribute $self approvalCode connectionFailed
    set order [findRelated $self R7] ; # ❶
    signal $order paymentDeclined
}
```

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

Shipping Subsystem

The Shipping subsystem consists of the following classes.

```
<<Shipping Subsystem>>=
<<Shipment 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 int -id 1 -system 200
    attribute shippingCompany string -default {} ; # ❶
    reference R21 ShippingCompany -link {shippingCompany companyName}
    attribute trackingNumber string -default {}

    attribute recipient string
    attribute deliveryAddress string
    attribute contactPhone string
    attribute timePrepared string -default {}
    attribute timePickedUp string -default {}
    attribute timeDelivered string -default {}
    attribute waitingToBePacked boolean -default true

    attribute warehouseName string -default {}
    reference R24 Warehouse -link warehouseName

    attribute clerkID string -default {}
    reference R22 WarehouseClerk -link clerkID

    attribute orderID int -default 0
    reference R6 Order -link orderID

    statemodel {
        <<Shipment State Model>>
    }

    <<Shipment Operations>>
}
```

❶ 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. Because of that change, we don't really want to make the `shippingCompany` attribute part of an identifier as it will change depending upon when the `R21` relationship is established. A better approach would be to factor `R21` as a class based association where `shippingCompany` and `trackingNumber` could be proper identifying attributes. However, we are loath to add classes to the model during translation and so take the easier approach of removing the attributes as an identifer.

```
<<Shipment Class Definition>>=
association R21 Shipment 0..*--0..1 ShippingCompany ; #❶
association R24 Shipment 0..*--1 Warehouse
association R22 Shipment 0..*--0..1 WarehouseClerk
association R6 Shipment 0..1--1 Order
```

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 get a number before 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.

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 Tcl code to the action language source.

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

state PreparingShipment {order} {
    R6 link $self $order

    set warehouse [Warehouse chooseWarehouse [readAttribute $order orderID]] ; # ❶
    R24 link $self $warehouse

    set items [findRelated $order R10 {R4 ProductSelection}] ; #❷
    R9 link $items $self

    signal $warehouse shipmentReadyToPack ; #❸
}
transition PreparingShipment - packed -> PackedandAwaitingTrackingNumber
transition PreparingShipment - trackingNumberAssigned ->\
    NumberAssignedandWaitingtobePacked
```

❶ This state 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).

❷ 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.

❸ 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 an d 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. Not sure why this was important, but it's there.

```
<<Shipment State Model>>=
state PackedandAwaitingTrackingNumber {clerkID} {
    instop $self updatePackingInfo $clerkID
}
transition PackedandAwaitingTrackingNumber - trackingNumberAssigned ->\
    NumberAssigned

state NumberAssignedandWaitingtobePacked {
        shippingCompany trackingNumber} {
    instop $self updateTrackingInfo $shippingCompany $trackingNumber
}
transition NumberAssignedandWaitingtobePacked - packed -> Packed

state NumberAssigned {shippingCompany trackingNumber} {
    instop $self updateTrackingInfo $shippingCompany $trackingNumber
}
transition NumberAssigned - pickedUp -> InTransittoCustomer

state Packed {clerkID} {
```

```
    instop $self updatePackingInfo $clerkID
}
transition Packed - pickedUp -> InTransittoCustomer
```

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

```
<<Shipment Operations>>=
instop updateTrackingInfo {shippingCompany trackingNumber} {
    updateAttribute $self trackingNumber $trackingNumber
    set company [ShippingCompany findById companyName $shippingCompany]
    R21 link $self $company
}

instop updatePackingInfo {clerkID} {
    updateAttribute $self timePrepared [clock format [clock seconds]]

    set clerk [WarehouseClerk findById clerkID $clerkID]
    R22 link $self $clerk

    EE_ShippingCompany::shipmentReadyForPickup\
        [readAttribute $self shipmentID]
}
```

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

```
<<Shipment State Model>>=
state InTransittoCustomer {} {
    updateAttribute $self timePickedUp [clock format [clock seconds]]
}
transition InTransittoCustomer - deliveryConfirmed -> Delivered

state Delivered {timeDelivered} {
    updateAttribute $self timeDelivered $timeDelivered
    set order [findRelated $self R6]
    signal $order orderDelivered
}
```

Shipping Company Class

```
<<Shipping Company Class Definition>>=
class ShippingCompany {
    attribute companyName string -id 1
    attribute trackingWebsite string
    attribute customerServicePhone string
    attribute localDispatchPhone string
    attribute localOffice string
    attribute localContact string
}
```

Warehouse Class

```
<<Warehouse Class Definition>>=
class Warehouse {
    attribute warehouseName string -id 1
    attribute warehouseLocation string

    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. Here, we do the simplest thing we can and simply choose an arbitrary `Warehouse`.

```
<<Warehouse Operations>>=
classop chooseWarehouse {orderID} {
    return [limitRef [Warehouse findAll]]
}
```

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 {} {
    set readyShipment [findRelatedWhere $self ~R24 {$waitingToBePacked}]
    if {[isNotEmptyRef $readyShipment]} {
        signal $self shipmentReadyToPack
    }
}
transition WaitingforaShipment - shipmentReadyToPack -> WaitingforaFreeClerk

state WaitingforaFreeClerk {} {
    set freeClerk [findRelatedWhere $self {~R25 {~R27 ShippingClerk}}\
        {$awaitingAssignment eq "true"}]
    puts $freeClerk
    if {[isNotEmptyRef $freeClerk]} {
        signal $self clerkFree
    }
}
transition WaitingforaFreeClerk - clerkFree -> AssigningClerktoShipment

state AssigningClerktoShipment {} {
    set readyShipment [findRelatedWhere $self ~R24 {$waitingToBePacked}]
    set readyShipment [limitRef $readyShipment]

    set freeClerk [findRelatedWhere $self {~R25 {~R27 ShippingClerk}}\
        {$awaitingAssignment}]
    set freeClerk [limitRef $freeClerk]

    R23 link $readyShipment $freeClerk
    updateAttribute $readyShipment waitingToBePacked false
```

```
    updateAttribute $freeClerk awaitingAssignment false

    signal $freeClerk clerkAssigned
    signal $self clerkAssignedToShipment
}
transition AssigningClerktoShipment - clerkAssignedToShipment ->\
        WaitingforaShipment
```

Warehouse Clerk Class

```
<<Warehouse Clerk Class Definition>>=
class WarehouseClerk {
    attribute clerkID string -id 1
    attribute clerkName string
    attribute goOffDutyAtEndOfJob boolean -default false ; # ❶

    attribute warehouseName string
    reference R25 Warehouse -link warehouseName
}
```

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 clerkID string -id 1
    reference R27 WarehouseClerk -link clerkID

    attribute awaitingAssignment boolean -default true

    attribute shipmentID int -default 0
    reference R23 Shipment -link shipmentID

    statemodel {
        <<Shipping Clerk State Model>>
    }
}
```

```
<<Shipping Clerk Class Definition>>=
association R23 ShippingClerk 0..1--0..1 Shipment
```

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 state activity for 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.

```
<<Translation Helpers>>=
proc ::bookstore::ShippingClerk::randomdelaysignal {instref event {max 10}} {
    delaysignal [expr {int(rand() * $max) * 100}] $instref $event
}
```

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 {} {
    updateAttribute $self awaitingAssignment true
    set warehouse [findRelated R27 R25]
    signal $warehouse clerkFree
}
transition WaitingforaJob - clerkAssigned -> SelectingBooks
transition WaitingforaJob - offDuty -> OffDuty

state SelectingBooks {} {
    EE_ShippingClerk::shipmentReadyToPack\
        [readAttribute $self shipmentID]

    randomdelaysignal $self booksSelected
}
transition SelectingBooks - booksSelected -> PackingBox

state PackingBox {} {
    randomdelaysignal $self 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.

```
<<Shipping Clerk State Model>>=
state SealingBox {} {
    set allSelections [deRef [ProductSelection findAll] quantity] ; # ❶

    ShipmentItem update [pipe {
        findRelated $self R23 {R9 ShipmentItem} |
        deRef ~ |
        relation join ~ $allSelections |
        relation project ~ shipmentID productID cartID quantity |
        relation rename ~ quantity quantityShipped
    }] ; # ❷

    randomdelaysignal $self boxSealed
}
transition SealingBox - boxSealed -> AttachingShippingLabel
```

❶ We will need a relation value that contains the `quantity` attributes for the `ProductSelection` instances. It is used below.

❷ There are several way in which we could accomplish updating `quantityShipped` attribute of the `ShippmentItems` instances. One way would be to traverse `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. Then we could

read the `quantity` attribute and use its value to update the `quantityShipped` value. Alternatively, we can avoid all the iterative stuff and do this in a much more relational fashion. Relational algebra eliminates virtually all the explicit iteration in state activities since relation values are sets and the algebra operates on the entire value at one time. In this case we want to use the `update` command to change non-identifying attributes to new values. This command is supplied by `rosea` specifically for these kinds of situations. To use `ShipmentItem update` we need a relation value that has a heading consisting of the identifier of `ShipmentItem` and with any additional attributes we wish to modify, in this case `quantityShipped`. The five commands in the `pipe` compute the needed relation value.

1. `findRelated` traverse `R23` and `R9` ending at `ShipmentItem`, the associative class.
2. `deRef` dereferences the instance reference into a relation value so that we can now operate on it with the next three commands.
3. `relation join` performs a join on the `ProductSelection` projection that contains the `quantity` attributes and yields a new relation value that contains tuples where the `productID` and `cartID` attribute values match those in the `ShipmentItem` value. This will give a relation value with a `quantity` attribute which properly corresponds to the instances of `ShippmentItem`.
4. We can now begin to shape the heading of the joined value to match what we need. We need the `shipmentID`, `productID` and `cartID` attributes to identify an instance of `ShipmentItem` and the `quantity` attribute is what we want to magically turn into the `quantityShipped` attribute.
5. The magic is provided by using `relation rename` to rename the `quantity` attribute to be `quantityShipped`. The heading of the resulting relation value is now just what we need to update into the `ShipmentItem` instances.

The relational approach to performing the update of the `quantityShipped` attributes will certainly seem strained and confusing if you are not familiar with the algebraic operators. If you have had some experience programming in SQL, it might not seem that different, although the approach to computing results in SQL and TclRAL are quite different. In the end, the update is a single statement that clearly shows we are updating the `ShipmentItem` instances with something, even if that something takes several steps to compute. Sadly, most action languages take an imperative and iterative approach to these types of computations. It is relatively easy to deduce how to obtain any required iteration when the computation is formulated relationally. The opposite is not true. For comparison we state the iterative approach below.

```
set items [findRelated $self R23 {R9 ShipmentItem}]
forAllRefs item $items {
    set selection [findRelated $item R9]
    updateAttribute $item quantityShipped [readAttribute $selection quantity]
} ; # ❶
```

❶ `Rosea` does provide a means to iterate across a set of instance references. It is not that iteration is bad or unnecessary. It is simply that it is often overused when *set at a time* operations are readily available.

Although shorter to write, it performs a lot more computation as the relationships must be navigated repeatedly to perform what is equivalent to a single relational join.

```
<<Shipping Clerk State Model>>=
state AttachingShippingLabel {} {
    randomdelaysignal $self shippingLabelAttached
}
transition AttachingShippingLabel - shippingLabelAttached ->\
    DeliveringBoxtoLoadingDock

state DeliveringBoxtoLoadingDock {} {
    randomdelaysignal $self boxAtLoadingDoc
}
transition DeliveringBoxtoLoadingDock - boxAtLoadingDoc -> CompletingJob

state CompletingJob {} {
    set currentShipment [findRelated $self R23]
    signal $currentShipment packed [readAttribute $self clerkID]
```

```
    R23 unlink $currentShipment
    set wc [findRelated $self R27]
    if {[readAttribute $wc goOffDutyAtEndOfJob]} {
        signal $self offDuty
    } else {
        updateAttribute $self awaitingAssignment true
        set myWarehouse [findRelated $wc R25]
        signal $myWarehouse clerkFree
    }
}
transition CompletingJob - clerkAssigned -> SelectingBooks
transition CompletingJob - offDuty -> OffDuty

state OffDuty {} {
    R27 migrate $self OffDutyClerk
} ; # 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 clerkID string -id 1
    reference R27 WarehouseClerk -link clerkID
    attribute idle boolean -default true
}
```

Off Duty Clerk Class

```
<<Off Duty Clerk Class Definition>>=
class OffDutyClerk {
    attribute clerkID string -id 1
    reference R27 WarehouseClerk -link clerkID
}
```

Shipment Item Class

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

    attribute shipmentID int -id 1
    reference R9 Shipment -link shipmentID

    attribute productID string -id 1
    attribute cartID int -id 1
    reference R9 ProductSelection -link productID -link cartID
}
```

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

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 `rosea` it results in a set of procedures and data structures that represent the actions of the domain. `Rosea` provides a run time component to map model level actions, *e.g.* navigating relationships to find related instances, onto Tcl 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 quanity 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 are doing 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>>
<<Book Product Population>>
<<Recording Product Population>>
<<Software Product Population>>
<<Special Order Product Population>>
<<Stocked Product Population>>
<<Product Category Population>>
<<Recording Population>>
<<Artist Population>>
<<Performance Credit Population>>
<<Recording Format Population>>
<<Platform Population>>
<<Computer Software Population>>
```

`Rosea` 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 supply referential attributes, simply setting referential attributes to the values that correspond to the identifying attributes of the related instance is sufficient. So below, although we are obviously execution commands in Tcl, we are simply supplying values to attributes. If we fail to supply correct ones, then the population command will fail with error messages that indicate the problem.

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>>=
set eumlid 780201 ; # ❶
set recid 7789930
set wordid 938763
```

❶ We start with some basic ordinary Tcl variables that will be used to hold counters and attribute values that are used in several places.

```
<<Product Population>>=
class Product {
    productID
    productName
    copyright
    unitPrice
    description
    website
    currentlyAvailable
    categoryID
    groupCode
    publisherCode
}\
    $eumlid\
    {Executable UML: A Foundation for Model-Driven Architecture}\
    2002\
    2995\
    {Book on software development using models.}\
    www.executableumlbook.com\
    true\
    [set bookid [incr categorytag]]\
    0\
    201\
\
    $recid\
    {Carnival Music for Children}\
    1955\
    1000\
    {Music as heard at old time carnivals}\
    {}\
    true\
    [set musicid [incr categorytag]]\
    1\
    523\
\
    $wordid\
    {Word 2020 for MacOSX}\
    2020\
    5995\
    {Infamous word processing program}\
    www.microsoft.com\
    true\
    [set swid [incr categorytag]]\
    2\
    666 ; # ❶
```

❶ This is an example of populating using the form of the commands that is most convenient when variables are involved. The first argument is a list of the attributes which sets the order in which they appear in the subsequent arguments. There is a bit of Tcl syntax here that might be unfamiliar.

Publisher Population

```
<<Publisher Population>>=
class Publisher {
    groupCode
    publisherCode
```

```
    name
    address
    website
} {
    0
    201
    Addison-Wesley
    {New York, NY}
    www.aw.com

    1
    523
    {Old-Time Music}
    {St. Louis, MO}
    www.otm.com

    2
    666
    {Microsoft, Inc.}
    {Seattle, WA}
    www.microsoft.com
} ; # ❶
```

❶ Here we see the form of population that is most convenient for constant values. Fewer backslashes are required and whitespace is less significant. Again, Tcl syntax rules apply.

Author Population

```
<<Author Population>>=
class Author {
    name
    website
    email
} {
    {Stephen J. Mellor}
    stephenmellor.com
    stephen@stephenmellor.com

    {Marc J. Balcer}
    marcbalcer.com
    marc@marcbalcer.com
}
```

Authorship Population

```
<<Authorship Population>>=
class Authorship {
    name
    productID
    previousName
    previousProductID
}\
    {Stephen J. Mellor}\
    $eumlid\
    {}\
    {}\
\
```

```
    {Marc J. Balcer}\
    $eumlid\
    {Stephen J. Mellor}\
    $eumlid
```

Book Product Population

```
<<Book Product Population>>=
class BookProduct {
    productID
    bookNumber
    titleCode
    title
    subtitle
}\
    $eumlid\
    {ISBN 0-201-74804-5}\
    74804-5\
    {Executable UML}\
    {A Foundation for Model-Driven Architecture}
```

Recording Product Population

```
<<Recording Product Population>>=
class RecordingProduct {
    productID
    runningTime
    recordingID
    formatID
}\
    $recid\
    3:23\
    [set cmusid [incr recordingtag]]\
    [set cdid [incr formattag]]
```

Software Product Population

```
<<Software Product Population>>=
class SoftwareProduct {
    productID
    productVersion
    softwareID
    platformID
}\
    $wordid\
    20.17.3\
    [set softid [incr softwaretag]]\
    [set platformid [incr platformtag]]
```

Special Order Product Population

```
<<Special Order Product Population>>=
class SpecialOrderProduct {
    productID
    specialOrderInstructions
    daysToDeliver
}\
    $recid\
    {Their part number: 30557}\
    10
```

Stocked Product Population

```
<<Stocked Product Population>>=
class StockedProduct {
    productID
    quantityOnHand
    reorderThreshold
    reorderIncrement
    reorderInstructions
}\
    $eumlid\
    100\
    200\
    100\
    {Ship as quickly as possible}\
\
    $wordid\
    55\
    25\
    55\
    {Get a discount}
```

Product Category Population

```
<<Product Category Population>>=
class ProductCategory {
    categoryID
    categoryName
    parentCategoryID
}\
    $bookid\
    book\
    {}\
\
    $musicid\
    music\
    {}\
\
    $swid\
    software\
    {}
```

Recording Population

```
<<Recording Population>>=
class Recording {
    recordingID
    title
}\
    $cmusid\
    {Carnival Music for Children}
```

Artist Population

```
<<Artist Population>>=
class Artist {
    artistID
    artistName
}\
    [set artist1id [incr artisttag]]\
    {Mary Jane}\
\
    [set artist2id [incr artisttag]]\
    {Fred Smith}
```

Performance Credit Population

```
<<Performance Credit Population>>=
class PerformanceCredit {
    recordingID
    artistID
    prevRecordingID
    prevArtistID
}\
    $cmusid\
    $artist1id\
    {}\
    {}\
\
    $cmusid\
    $artist2id\
    $cmusid\
    $artist1id
```

Recording Format Population

```
<<Recording Format Population>>=
class RecordingFormat {
    formatID
    formatName
}\
    $cdid\
    CD
```

Platform Population

```
<<Platform Population>>=
class Platform {
    platformID
    platformName
}\
    $platformid\
    MacOSX
```

Computer Software Population

```
<<Computer Software Population>>=
class ComputerSoftware {
    softwareID
    title
}\
    $softid\
    {Word 2020 for MacOSX}
```

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.

Shipping Subsystem Population

```
<<Shipping Population>>=
<<Shipping Company Population>>
<<Warehouse Population>>
<<Warehouse Clerk Population>>
<<Shipping Clerk Population>>
<<Stock Clerk Population>>
<<Off Duty Clerk Population>>
```

Shipping Company Population

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

```
    UPS
    www.ups.com
    224-555-1212
    225-555-1212
    {220 Elm St., Anywhere, CA 94000}
    {John Parcel}
}
```

Warehouse Population

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

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

Warehouse Clerk Population

```
<<Warehouse Clerk Population>>=
class WarehouseClerk {
    clerkID
    clerkName
    warehouseName
}\
    [set clerk1id [incr clerktag]]\
    {Fred Smith}\
    {Acme Bookstore Warehouse}\
\
    [set clerk2id [incr clerktag]]\
    {John Industrious}\
    {Acme Bookstore Warehouse}\
\
    [set clerk3id [incr clerktag]]\
    {George Slacker}\
    {My Fulfillment}
```

Shipping Clerk Population

```
<<Shipping Clerk Population>>=
class ShippingClerk {
    clerkID
}\
    $clerk1id
```

Stock Clerk Population

```
<<Stock Clerk Population>>=
class StockClerk {
    clerkID
}\
    $clerk2id
```

Off Duty Clerk Population

```
<<Off Duty Clerk Population>>=
class OffDutyClerk {
    clerkID
}\
    $clerk3id
```

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.

Our strategy for external entities is to define a set of namespaces in the domain namespace to hold the required procedures. All of these namespaces will start with `EE_` and that should avoid any naming conflicts.

All the procedures start by printing out their invocation sequence, *i.e.* the Tcl idom `info level 0` provides that information. For most of the external entity procedures that is all we strive to do.

```
<<External Entity Stubs>>=
namespace eval ::bookstore::EE_CreditCardCompany {
    proc requestChargeApproval {accountNumber billingAddress cardholderName
                cardExpirationDate chargeID chargeAmount} {
        puts [info level 0]

        set chargeRef [::bookstore::CreditCardCharge findById chargeID $chargeID]
        after 100 [list rosea tunnel $chargeRef signal chargeProcessed approved] ; # ❶
    }
}
```

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 Entity Stubs>>=
namespace eval ::bookstore::EE_OnLineCustomer {
    proc chargeDeclined {email} {
        puts [info level 0]
    }
    proc chargeApproved {email} {
        puts [info level 0]
    }
    proc orderReportedDelivered {email} {
        puts [info level 0]
    }
}
```

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 Entity Stubs>>=
namespace eval ::bookstore::EE_ShippingCompany {
    proc shipmentReadyForPickup {shipmentID} {
        puts [info level 0]

        set shipmentRef [::bookstore::Shipment findById shipmentID $shipmentID]
        rosea tunnel $shipmentRef signal trackingNumberAssigned\
            FedEx 130047332

        after 200 [list rosea tunnel $shipmentRef signal pickedUp]
        after 400 [list ::bookstore::EE_ShippingCompany::reportDelivery\
                $shipmentID]
    }

    # We use a separate procedure, rather than directly tunneling the event, so
    # we can get a better value for the deliveryConfirmed event parameter.
    proc reportDelivery {shipmentID} {
        puts [info level 0]

        set shipmentRef [::bookstore::Shipment findById shipmentID $shipmentID]
        rosea tunnel $shipmentRef signal deliveryConfirmed\
                [clock format [clock seconds]]
    }
}
```

```
<<External Entity Stubs>>=
namespace eval ::bookstore::EE_ShippingClerk {
    proc shipmentReadyToPack {shipmentID} {
        puts [info level 0]
    }
}
```

Execution Scenarios

Finally, we are ready to run the translated domain. We will construct a single test file that contains a set of execution scenarios. We have chosen to do it this way 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.

For each execution scenario, we will record the state machine event dispatch traces. Those traces will then be presented as text and as a UML sequence diagram. It is important to remember that the sequence diagrams are programmatically generated and show the flow of events that happened during actual execution.

Tcl has direct support in its core for event driven programming. It has the concept of an event loop and to have state machine events dispatched we must *enter the event loop*. Normally, a domain would enter the event loop immediately after initialization and would probably run forever or until it explicitly exited based on some condition. What we want to do is enter the event loop and run until some state activity is completed. Then we wish to take event dispatch traces and manage other aspects of the flow of our execution scenario. This is definitely a contrived situation because we want to stop things and look at what has just happened. We will create some convenience procedures to make the effort a bit less tedious.

Our stategy for being able to synchronize to the execution of state activities depends upon two factors.

1. `Rosea` converts state activity code into ordinary Tcl procedures.
2. Tcl provides the means to trace the execution of a procedure.

Using this along with a timer to insure that we don't wait forever if something goes wrong, we can generate a procedure that will allow us to block until either a given state activity is executed or we time out.

```
<<Common Execution Utilities>>=
proc ::syncToStateActivity {class state {timeout 3000}} {
    set activityproc\
        [rosea info statemodel activityproc bookstore $class $state] ; # ❶

    trace add execution $activityproc leave syncToStop ; # ❷

    set totimer [after $timeout set ::done TIMEOUT]
    vwait ::done ; # ❸
    after cancel $totimer
    trace remove execution $activityproc leave syncToStop

    if {$::done eq "TIMEOUT"} {
        error "Sync to $class / $state timed out"
    }
    return $::done
}
```

❶ Here we obtain the name of the Tcl procedure that contains the state activity.

❷ This command tells Tcl to invoke the `syncToStop` command when execution leaves the activity procedure.

❸ Enter the event loop waiting for the `::done` variable to be written. That will happen either when `syncToStop` is executed or when the timeout script is executed.

Tcl arranges the execution trace callback to have a number of interesting arguments about the context of the trace. We don't use any of them and directly write to the global `::done` variable.

```
<<Common Execution Utilities>>=
proc ::syncToStop {cmd code result op} {
    set ::done 1
}
```

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

```
<<testscenario.tcl>>=
<<copyright info>>

package require rosea 1.5

<<Common Execution Utilities>>

rosea configureFromFile simplebookstore.tcl ; # ❶
rosea generate ; # ❷
rosea populateFromFile sbs_population.tcl
```

```
<<External Entity Stubs>>
<<Translation Helpers>>

rosea trace control on

<<scenario1>>
<<scenario2>>
<<scenario3>>
<<scenario4>>

rosea trace control off
```

❶ These three commands build the domain from our description, generate the run time data and install the population. After they run, the domain is ready to receive events.

❷ Because our domain was named `bookstore` and we did not specify a namespace where it was to be located, the domain will be placed in the `::bookstore` namespace.

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. In order create a shopping cart we must supply a product ID and a quantity. To obtain the product ID, we will look up a book by name and then extract the productID from the instance. In this case, we look up the name of the Executable UML book, which our bookstore conveniently stocks.

```
<<scenario1>>=
set exumlref [bookstore::Product findWhere\
        {[string match {Executable*} $productName]}]
set id [rosea tunnel $exumlref readAttribute productID] ; # ❶
```

❶ The `rosea tunnel` command allows us to bridge into the domain to perform implicit operations on the internals. Yes, this breaks encapsulation, but in a very controlled manner and is essential for bridging domains together. Testing is just another form of bridging and one that usually exercises the implicit abilities of the software architecture.

We have to manufacture an ID for the cart. Any old number will do. We'll just keep one around in the `carttag` variable.

```
<<scenario1>>=
set cartID [incr carttag]
bookstore::ShoppingCart createasync\
    startCart [list $id 1]\
    cartID $cartID totalValue 0 ; # ❶
```

❶ Signal a creation event, passing the productID and quantity as event parameters and supplying the other `ShoppingCart` attributes.

Since the instance creation is asynchronous, we must wait until that event is dispatched. Consulting the state diagram, we can see that we need to wait until the `AddingSelectiontoOrder` state is executed.

```
<<scenario1>>=
syncToStateActivity ShoppingCart AddingSelectiontoOrder
```

Now we will effectively cancel the order by changing the quantity associated with the product selection to 0.

```
<<scenario1>>=
set selection [bookstore::ProductSelection findById\
        cartID $cartID productID $id]
rosea::tunnel $selection signal changeQuantity 0
```

After signaling the event, we must await for it to execute.

```
<<scenario1>>=
syncToStateActivity ShoppingCart CancelingEntireOrder
```

Finally, we want to record the event dispatch traces.

```
<<scenario1>>=
logTraces scenario1
```

The procedure to record the event dispatch traces obtains both the text representation as well as the graphical representation.

```
<<Common Execution Utilities>>=
proc ::logTraces {filenamebase} {
    set logchan [::open $filenamebase.log w]
    try {
        puts $logchan [rosea trace format [rosea trace decode all]]
    } finally {
        ::chan close $logchan
    }
    rosea trace diagram all $filenamebase.diag ; # ❶
    rosea trace control clear ; # ❷
}
```

❶ This command outputs a file suitable for use with the `seqdiag` program that does the actual sequence diagram graphical drawing.

❷ Clear out the traces in preparation for the next scenario.

Below are the event dispatch traces in text log form. The traces are timestamped and the time difference from the previous trace is given. For creation events, the details of the instance reference of the new created instance are given. For ordinary transitions, the source of the event, the event and its arguments and the target of the event are given. After the ==> mark, the transition from current state to new state is shown. You can follow along with the model graphic to see the transitions that happened.

State Machine Trace Log For Scenario 1

```
14:07:49.582.529:   0.000.000: Creation: {} - startCart -> ::bookstore:: ↩
    ShoppingCart ==> ::bookstore::ShoppingCart{cartID 1}
14:07:49.582.766:   0.000.237: Transition: {} - startCart(780201, 1) -> :: ↩
    bookstore::ShoppingCart{cartID 1} ==> @ -> NewOrder
14:07:49.583.514:   0.000.748: Transition: ::bookstore::ShoppingCart{cartID 1} -  ↩
    addSelection(780201, 1) -> ::bookstore::ShoppingCart{cartID 1} ==> NewOrder ->  ↩
    AddingSelectiontoOrder
14:07:49.585.879:   0.002.365: Transition: {} - changeQuantity(0) -> ::bookstore:: ↩
    ProductSelection{productID 780201 cartID 1} ==> NewSelection ->  ↩
    ChangingQuantity
14:07:49.586.400:   0.000.521: Transition: ::bookstore::ProductSelection{productID ↩
     780201 cartID 1} - removeSelection -> ::bookstore::ProductSelection{productID  ↩
    780201 cartID 1} ==> ChangingQuantity -> RemovingSelection
14:07:49.587.788:   0.001.388: Transition: ::bookstore::ProductSelection{productID ↩
     780201 cartID 1} - cancel -> ::bookstore::ShoppingCart{cartID 1} ==>  ↩
    AddingSelectiontoOrder -> CancelingEntireOrder
```

Below is the same trace information in a graphical form of a UML sequence diagram. Time flow is from top to bottom. The events that originate from the `EXTERNAL` class are those that come from outside of the domain. Otherwise, the net flow of events as they were dispatched during execution is shown.

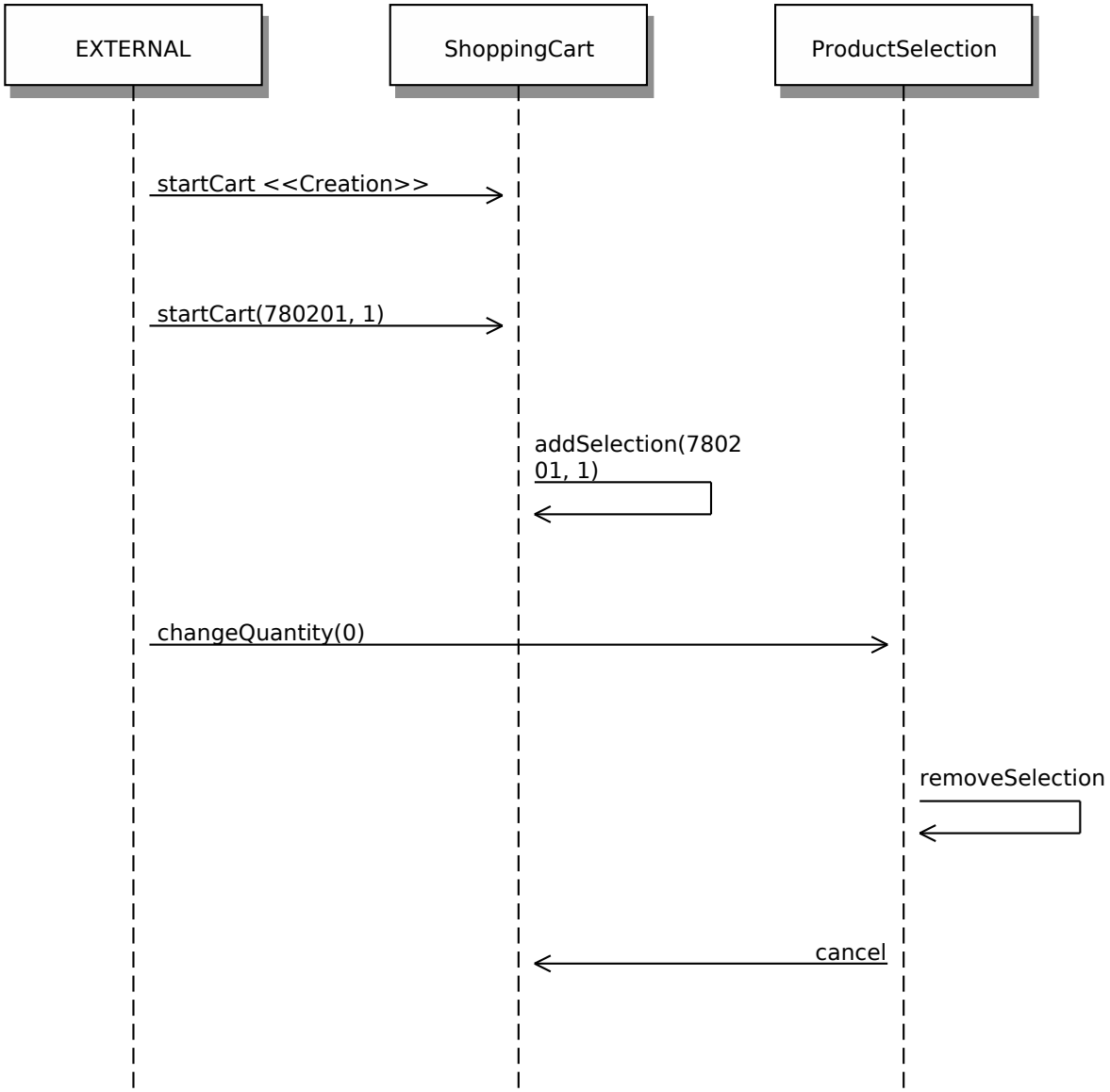

Figure 1: Scenario 1 Sequence Diagram

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 start by generating a new cart ID.

```
<<scenario2>>=
set cartID [incr carttag]
```

We create the new shopping cart, waiting for the creation event to be dispatched.

```
<<scenario2>>=
bookstore::ShoppingCart createasync\
```

```
    startCart [list $id 1]\
    cartID $cartID totalValue 0

syncToStateActivity ShoppingCart AddingSelectiontoOrder
```

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

```
<<scenario2>>=
set selection [bookstore::ProductSelection findById\
        cartID $cartID productID $id]
rosea::tunnel $selection signal changeQuantity 3

syncToStateActivity ProductSelection ChangingQuantity
```

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

```
<<scenario2>>=
set musicRef [bookstore::Product findWhere\
        {[string match {Carnival*} $productName]}]
set id [rosea tunnel $musicRef readAttribute productID]

set cartRef [bookstore::ShoppingCart findById cartID $cartID]
rosea::tunnel $cartRef signal addSelection $id 2 ; # ❶
syncToStateActivity ShoppingCart AddingSelectiontoOrder
```

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>>=
rosea::tunnel $cartRef signal checkOut\
        {5555 6666 7777 8888}\
        {100 E Main St., Anywhere, CO 88888}\
        {10/17}\
        {Fred Smith}\
        fsmith@gmail.com\
        {Fred Smith}\
        {222-555-1212}\
        {100 E Main St., Anywhere, CO 88888}

syncToStateActivity Order BeingPackedandShipped

logTraces scenario2
```

Again we show the text of the event dispatch traces as well as the sequence diagram.

State Machine Traces For Scenario 2

```
14:07:49.652.275:   0.000.000: Creation: {} - startCart -> ::bookstore:: ↩
    ShoppingCart ==> ::bookstore::ShoppingCart{cartID 2}
14:07:49.652.344:   0.000.069: Transition: {} - startCart(780201, 1) -> :: ↩
    bookstore::ShoppingCart{cartID 2} ==> @ -> NewOrder
14:07:49.652.576:   0.000.232: Transition: ::bookstore::ShoppingCart{cartID 2} -  ↩
    addSelection(780201, 1) -> ::bookstore::ShoppingCart{cartID 2} ==> NewOrder ->  ↩
    AddingSelectiontoOrder
14:07:49.653.691:   0.001.115: Transition: {} - changeQuantity(3) -> ::bookstore:: ↩
    ProductSelection{productID 780201 cartID 2} ==> NewSelection ->  ↩
    ChangingQuantity
```

```
14:07:49.655.295:   0.001.604: Transition: {} - addSelection(7789930, 2) -> :: ↩
    bookstore::ShoppingCart{cartID 2} ==> AddingSelectiontoOrder ->  ↩
    AddingSelectiontoOrder
14:07:49.656.363:   0.001.068: Transition: {} - checkOut(5555 6666 7777 8888, 100  ↩
    E Main St., Anywhere, CO 88888, 10/17, Fred Smith, fsmith@gmail.com, Fred Smith ↩
    , 222-555-1212, 100 E Main St., Anywhere, CO 88888) -> ::bookstore:: ↩
    ShoppingCart{cartID 2} ==> AddingSelectiontoOrder ->  ↩
    EstablishingCustomerandVerifyingPayment
14:07:49.656.768:   0.000.405: Creation: ::bookstore::ShoppingCart{cartID 2} -  ↩
    checkOut -> ::bookstore::Order ==> ::bookstore::Order{orderID 100}
14:07:49.656.840:   0.000.072: Transition: ::bookstore::ShoppingCart{cartID 2} -  ↩
    checkOut(::bookstore::ShoppingCart {{cartID int} {{cartID 2}}}, 5555 6666 7777  ↩
    8888, 100 E Main St., Anywhere, CO 88888, 10/17, Fred Smith, fsmith@gmail.com)  ↩
    -> ::bookstore::Order{orderID 100} ==> @ ->  ↩
    EstablishingCustomerandVerifyingPayment
14:07:49.658.662:   0.001.822: Transition: ::bookstore::Order{orderID 100} -  ↩
    submitCharge(5555 6666 7777 8888, 100 E Main St., Anywhere, CO 88888, 10/17,  ↩
    Fred Smith) -> ::bookstore::Order{orderID 100} ==>  ↩
    EstablishingCustomerandVerifyingPayment -> SubmittingCharge
14:07:49.659.093:   0.000.431: Creation: ::bookstore::Order{orderID 100} -  ↩
    makeCharge -> ::bookstore::CreditCardCharge ==> ::bookstore::CreditCardCharge{ ↩
    chargeID 100}
14:07:49.659.170:   0.000.077: Transition: ::bookstore::Order{orderID 100} -  ↩
    makeCharge(::bookstore::Order {{orderID int} {{orderID 100}}}) -> ::bookstore:: ↩
    CreditCardCharge{chargeID 100} ==> @ -> RequestingChargeApproval
14:07:49.760.315:   0.101.145: Transition: {} - chargeProcessed(approved) -> :: ↩
    bookstore::CreditCardCharge{chargeID 100} ==> RequestingChargeApproval ->  ↩
    ProcessingCompleted
14:07:49.761.587:   0.001.272: Transition: ::bookstore::CreditCardCharge{chargeID  ↩
    100} - paymentApproved -> ::bookstore::Order{orderID 100} ==> SubmittingCharge  ↩
    -> BeingPackedandShipped
```

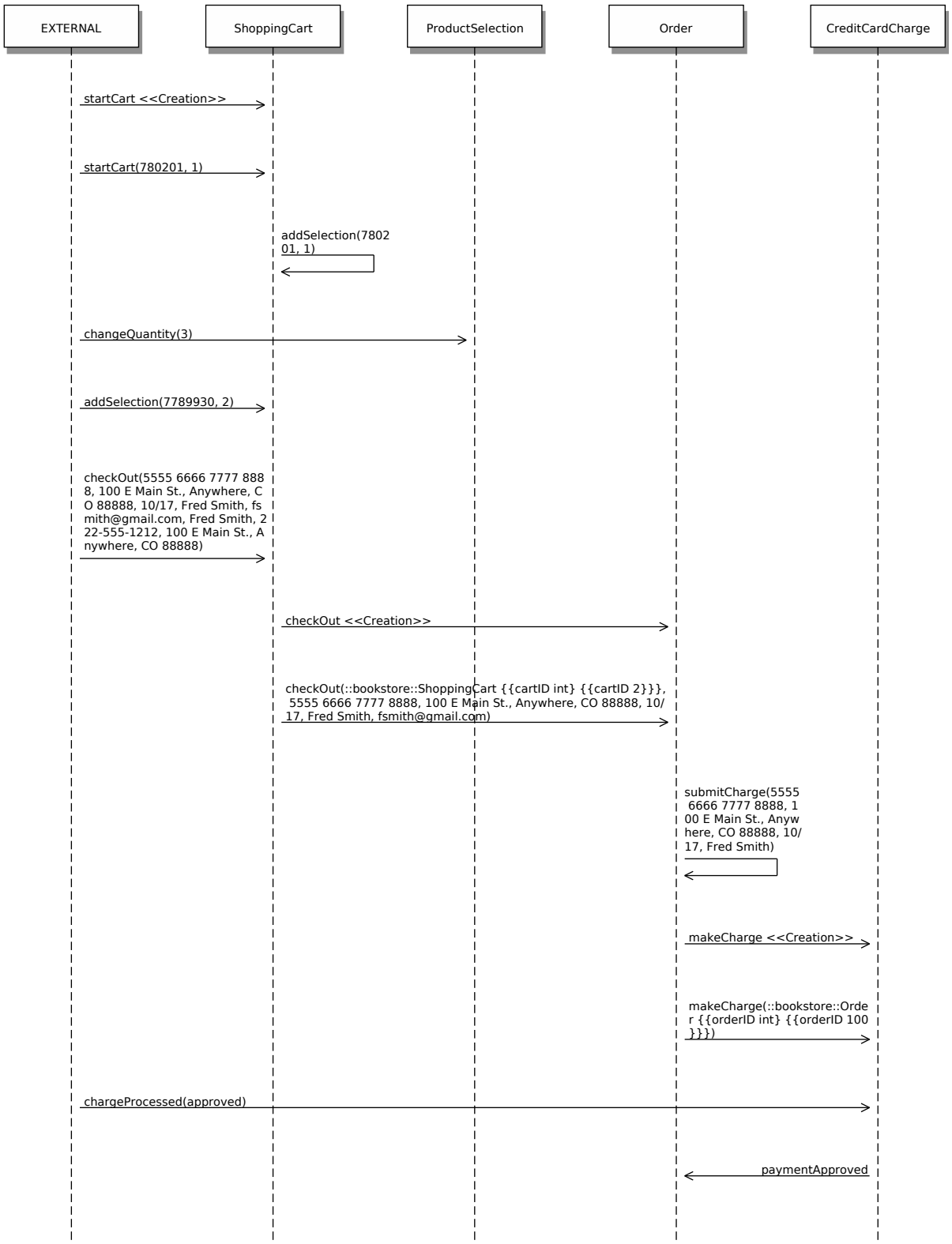

Figure 2: Scenario 2 Sequence Diagram

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>>=
syncToStateActivity ShippingClerk CompletingJob 10000

logTraces scenario3
```

State Machine Traces For Scenario 3

```
14:07:49.768.514:   0.000.000: Creation: ::bookstore::Order{orderID 100} -  ↩
    requestShipment -> ::bookstore::Shipment ==> ::bookstore::Shipment{shipmentID  ↩
    200}
14:07:49.768.583:   0.000.069: Transition: ::bookstore::Order{orderID 100} -  ↩
    requestShipment(::bookstore::Order {{orderID int} {{orderID 100}}}) -> :: ↩
    bookstore::Shipment{shipmentID 200} ==> @ -> PreparingShipment
14:07:49.770.181:   0.001.598: Transition: ::bookstore::Shipment{shipmentID 200} - ↩
     shipmentReadyToPack -> ::bookstore::Warehouse{warehouseName {Acme Bookstore  ↩
    Warehouse}} ==> WaitingforaShipment -> WaitingforaFreeClerk
14:07:49.770.831:   0.000.650: Transition: ::bookstore::Warehouse{warehouseName { ↩
    Acme Bookstore Warehouse}} - clerkFree -> ::bookstore::Warehouse{warehouseName  ↩
    {Acme Bookstore Warehouse}} ==> WaitingforaFreeClerk ->  ↩
    AssigningClerktoShipment
14:07:49.771.960:   0.001.129: Transition: ::bookstore::Warehouse{warehouseName { ↩
    Acme Bookstore Warehouse}} - clerkAssignedToShipment -> ::bookstore::Warehouse{ ↩
    warehouseName {Acme Bookstore Warehouse}} ==> AssigningClerktoShipment ->  ↩
    WaitingforaShipment
14:07:49.772.156:   0.000.196: Transition: ::bookstore::Warehouse{warehouseName { ↩
    Acme Bookstore Warehouse}} - clerkAssigned -> ::bookstore::ShippingClerk{ ↩
    clerkID 1} ==> WaitingforaJob -> SelectingBooks
14:07:50.672.675:   0.900.519: Transition: ::bookstore::ShippingClerk{clerkID 1} - ↩
     booksSelected -> ::bookstore::ShippingClerk{clerkID 1} ==> SelectingBooks ->  ↩
    PackingBox
14:07:50.673.070:   0.000.395: Transition: ::bookstore::ShippingClerk{clerkID 1} - ↩
     boxPacked -> ::bookstore::ShippingClerk{clerkID 1} ==> PackingBox ->  ↩
    SealingBox
14:07:51.074.013:   0.400.943: Transition: ::bookstore::ShippingClerk{clerkID 1} - ↩
     boxSealed -> ::bookstore::ShippingClerk{clerkID 1} ==> SealingBox ->  ↩
    AttachingShippingLabel
14:07:51.274.434:   0.200.421: Transition: ::bookstore::ShippingClerk{clerkID 1} - ↩
     shippingLabelAttached -> ::bookstore::ShippingClerk{clerkID 1} ==>  ↩
    AttachingShippingLabel -> DeliveringBoxtoLoadingDock
14:07:52.074.911:   0.800.477: Transition: ::bookstore::ShippingClerk{clerkID 1} - ↩
     boxAtLoadingDoc -> ::bookstore::ShippingClerk{clerkID 1} ==>  ↩
    DeliveringBoxtoLoadingDock -> CompletingJob
```

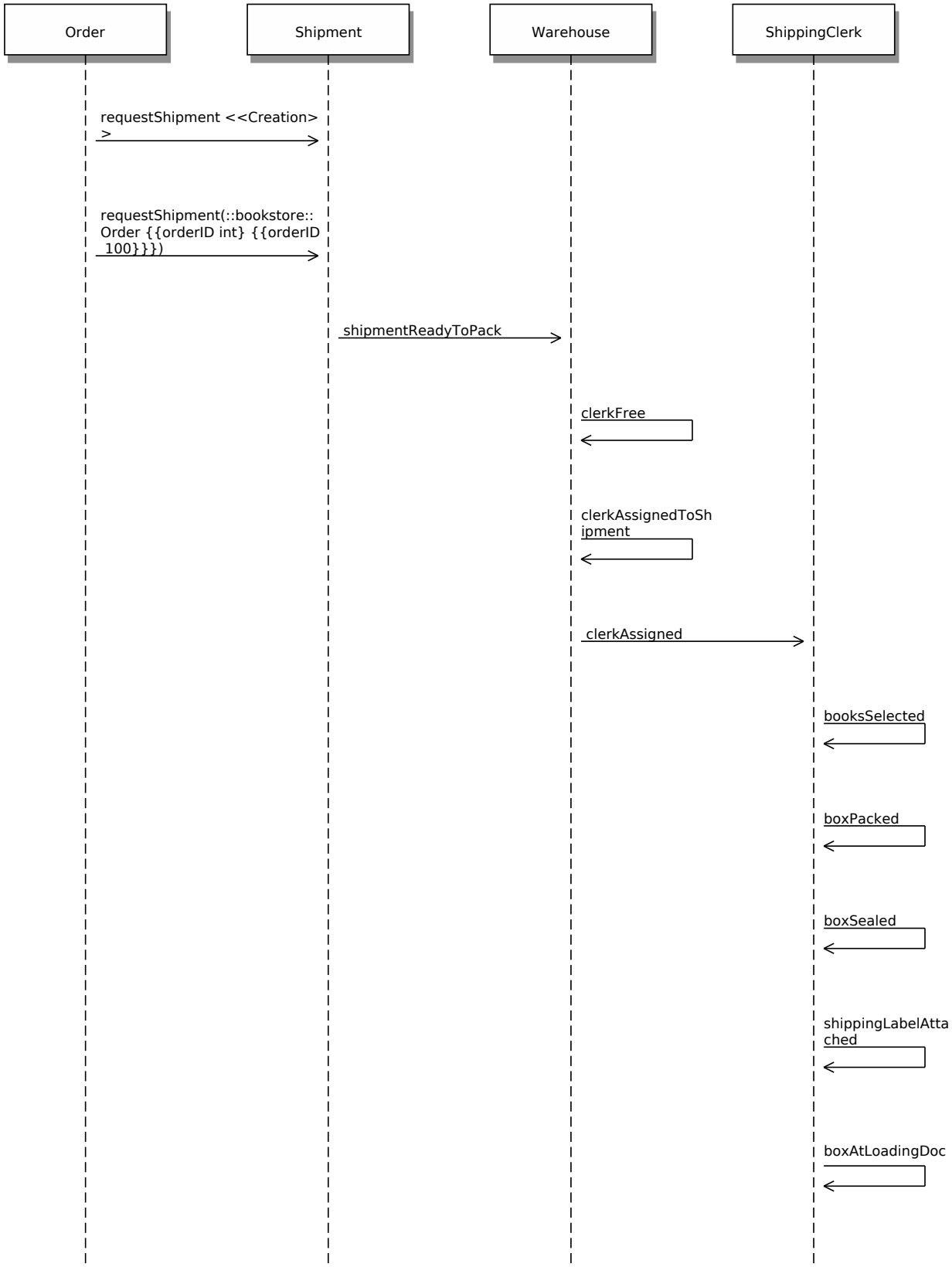

Figure 3: Scenario 3 Sequence Diagram

Scenario 4

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

```
<<scenario4>>=
syncToStateActivity Shipment Delivered 10000

logTraces scenario4
```

State Machine Traces For Scenario 4

```
14:07:52.081.632:   0.000.000: Transition: ::bookstore::ShippingClerk{clerkID 1} - ↩
    packed(1) -> ::bookstore::Shipment{shipmentID 200} ==> PreparingShipment ->  ↩
   PackedandAwaitingTrackingNumber
14:07:52.082.543:   0.000.911: Transition: ::bookstore::Shipment{shipmentID 200} - ↩
    trackingNumberAssigned(FedEx, 130047332) -> ::bookstore::Shipment{shipmentID  ↩
   200} ==> PackedandAwaitingTrackingNumber -> NumberAssigned
14:07:52.083.051:   0.000.508: Transition: ::bookstore::ShippingClerk{clerkID 1} - ↩
    clerkFree -> ::bookstore::Warehouse{warehouseName {Acme Bookstore Warehouse}}  ↩
   ==> WaitingforaShipment -> IG
14:07:52.282.803:   0.199.752: Transition: {} - pickedUp -> ::bookstore::Shipment{ ↩
   shipmentID 200} ==> NumberAssigned -> InTransittoCustomer
14:07:52.482.885:   0.200.082: Transition: {} - deliveryConfirmed(Fri Nov 27  ↩
   14:07:52 PST 2015) -> ::bookstore::Shipment{shipmentID 200} ==>  ↩
   InTransittoCustomer -> Delivered
```

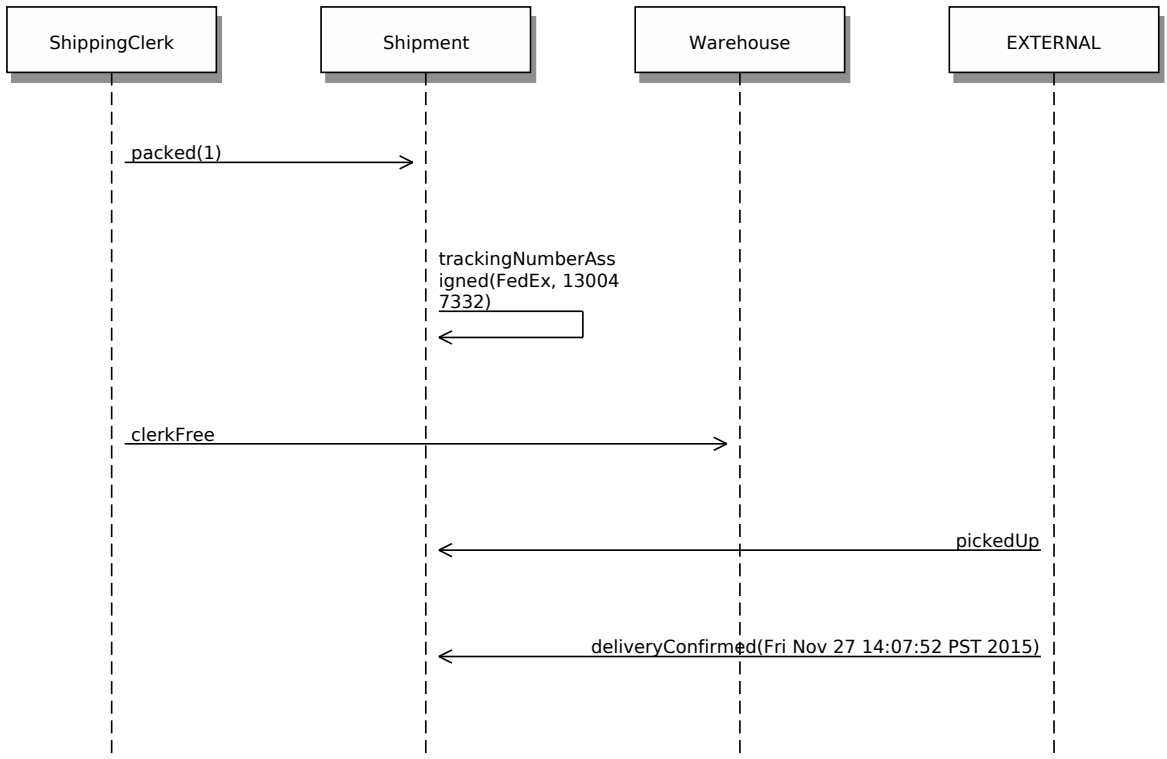

Figure 4: Scenario 4 Sequence Diagram

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 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

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

package require Tcl 8.6
package require rosea 1.5

domain bookstore {
```

```
    <<Product Specification Subsystem>>
    <<Ordering Subsystem>>
    <<Shipping Subsystem>>
}
```

Initial Instance Population

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

domain bookstore {
    <<Product Specification 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 Tcl interpreter. 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 Tcl code in an order suitable for the Tcl interpreter. 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 Tcl interpreter.

Index

A
Artist, 8, 31
Author, 5, 28
Authorship, 5, 28

B
Book Product, 29
BookProduct, 6

C
Classes
- Artist, 8
- Author, 5
- Authorship, 5
- BookProduct, 6
- Computer Software, 9
- Credit Card Charge, 16
- Customer, 10
- Off Duty Clerk, 25
- Order, 10
- Performance Credit, 8
- Platform, 9
- Product, 4
- Product Category, 8
- Product Selection, 12
- Publisher, 5
- Recording, 8
- Recording Format, 9
- Recording Product, 7
- Shipment, 18
- Shipment Item, 25
- Shipping Clerk, 22
- Shipping Company, 20
- ShoppingCart, 14
- Software Product, 7
- Special Order Product, 7
- Stock Clerk, 25
- Stocked Product, 8
- Warehouse, 20
- Warehouse Clerk, 22

Computer Software, 9, 32
Credit Card Charge, 16
Customer, 10

O
Off Duty Clerk, 25, 34
Order, 10
Ordering
- Classes
  - Credit Card Charge, 16
  - Customer, 10
  - Order, 10
  - Product Selection, 12
  - ShoppingCart, 14
- Relationships
  - R10, 10
  - R4, 13
  - R5, 10
  - R7, 16
  - R8, 16
- State Model
  - Credit Card Charge, 16
- State Models
  - Order, 10
  - Product Selection, 13
  - Shopping Cart, 15

P
Performance Credit, 8, 31
Platform, 9, 31
Population
- Artist, 31
- Author, 28
- Authorship, 28
- Book Product, 29
- Computer Software, 32
- Off Duty Clerk, 34
- Performance Credit, 31
- Platform, 31
- Product, 26
- Product Category, 30
- Publisher, 27
- Recording, 30
- Recording Format, 31
- Recording Product, 29
- Shipping Clerk, 33
- Shipping Company, 32
- Software Product, 29
- Special Order Product, 29
- Stock Clerk, 33
- Stocked Product, 30
- Warehouse, 33
- Warehouse Clerk, 33

Product, 4, 26
Product Category, 8, 30
Product Selection, 12, 13
Product Specification
- Classes
  - Artist, 8
  - Author, 5
  - Authorship, 5
  - BookProduct, 6
  - Computer Software, 9
  - Performance Credit, 8
  - Platform, 9
  - Product, 4
  - Product Category, 8

Publisher, 5
Recording, 8
Recording Format, 9
Recording Product, 7
Software Product, 7
Special Order Product, 7
Stocked Product, 8
Population
Artist, 31
Author, 28
Authorship, 28
Book Product, 29
Computer Software, 32
Performance Credit, 31
Platform, 31
Product, 26
Product Category, 30
Publisher, 27
Recording, 30
Recording Format, 31
Recording Product, 29
Software Product, 29
Special Order Product, 29
Stocked Product, 30
Relationships
R1, 5
R11, 5
R12, 5
R14, 8
R15, 5
R16, 7
R17, 9
R18, 9
R19, 7
R2, 6
R20, 7
R3, 6
Publisher, 5, 27

R
R1, 5
R10, 10
R11, 5
R12, 5
R14, 8
R15, 5
R16, 7
R17, 9
R18, 9
R19, 7
R2, 6
R20, 7
R21, 18
R22, 18
R23, 22
R24, 18
R3, 6
R4, 13
R5, 10
R6, 18
R7, 16
R8, 16
R9, 25
Recording, 8, 30
Recording Format, 9, 31
Recording Product, 7, 29
Relationships
R1, 5
R10, 10
R11, 5
R12, 5
R14, 8
R15, 5
R16, 7
R17, 9
R18, 9
R19, 7
R2, 6
R20, 7
R21, 18
R22, 18
R23, 22
R24, 18
R3, 6
R4, 13
R5, 10
R6, 18
R7, 16
R8, 16
R9, 25
RR, 22
RR, 22

S
Shipment, 18, 19
Shipment Item, 25
Shipping
Classes
Off Duty Clerk, 25
Shipment, 18
Shipment Item, 25
Shipping Clerk, 22
Shipping Company, 20
Stock Clerk, 25
Warehouse, 20
Warehouse Clerk, 22
Population
Off Duty Clerk, 34
Shipping Clerk, 33
Shipping Company, 32
Stock Clerk, 33
Warehouse, 33
Warehouse Clerk, 33
Relationships

R21, 18
R22, 18
R23, 22
R24, 18
R6, 18
R9, 25
RR, 22
State Model
Shipment, 19
Shipping Clerk, 23
Warehouse, 21
Shipping Clerk, 22, 23, 33
Shipping Company, 20, 32
Shopping Cart, 15
ShoppingCart, 14
Software Product, 7, 29
Special Order Product, 7, 29
State Model
Credit Card Charge, 16
Shipment, 19
Shipping Clerk, 23
Warehouse, 21
State Models
Order, 10
Product Selection, 13
Shopping Cart, 15
Stock Clerk, 25, 33
Stocked Product, 8, 30

W

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