Overview Lecture

Course - CS371S

Instructor - James C. Browne

Course Overview

Model Driven Development Overview

Executable UML Overview

Tool Overview

Model Driven Paradigm for Software Development

The steps in the development cycle are:

a) The system is defined as an executable specification which is an object-oriented analysis model.

b) The system is validated at the analysis model level.

c) A software and execution architecture is defined as a set of class templates in an object-oriented programming system.

d) The executable system is realized by compilation of the validated analysis model to the software execution architecture.

Why

Using models to design complex systems is de rigeur in traditional engineering disciplines. No one would imagine constructing an edifice as complex as a bridge or an automobile without first constructing a variety of specialized system models. from using models and modeling techniques. Models help us understand a complex problem and its potential solutions through abstraction. Therefore, it seems obvious that software systems, which are often among the most complex engineering systems, can benefit greatly

Why

MDD’s defining characteristic is that software development’s primary focus and products are models rather than computer programs.The major advantage of this is that we express models using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain relative to most popular programming languages. This makes the models easier to specify, understand, verify and maintain.

Requirement - Code Generation

If models are merely documentation, they are of limited value, because documentation all too easily diverges from reality. A key premise behind MDD that programs are automatically generated from their corresponding models.

Therefore

Models must be executable.

Further

The resulting executable must be reasonably competitive wrt resource consumption.

Overview Lecture

This course is about designing, constructing, validating and verifying executable models which can be compiled to procedural code such as C++ or Java.

Modeling Language – xUML

Design Principles and Paradigms – textbook and example papers

Validation and verification

Tools – BridgePoint, iUMLite, Rational Rose, etc.

Evaluation of MDD process

Course Work Requirements

This is essentially a laboratory class. The lectures will cover the xUML and the executable specification based development method in detail and other methods as alternatives. The main goal of the course will be to carry through a complete development of a small software system using object-oriented development methods. There will also be periodic laboratory exercises to develop skills before projects are started.

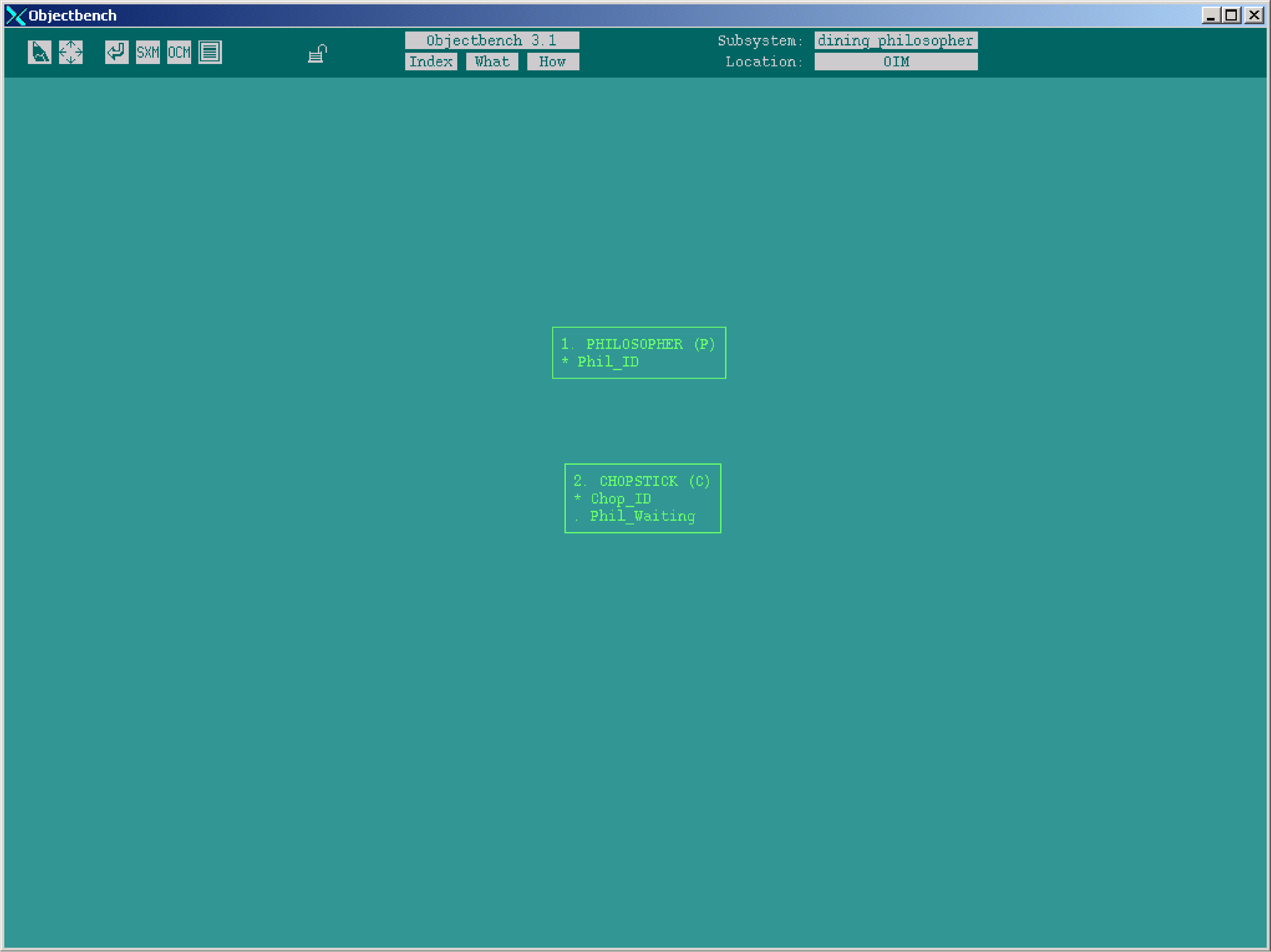
There will be two class examinations but no final examination.

Project Specifications

The project will be development of a small software system through the executable specification development methodology. The projects will be executed by small teams of co-workers. I have a set of possible projects. Each team will do a different project. A team can suggest a project of their own definition by preparing a requirements specification and getting it approved.

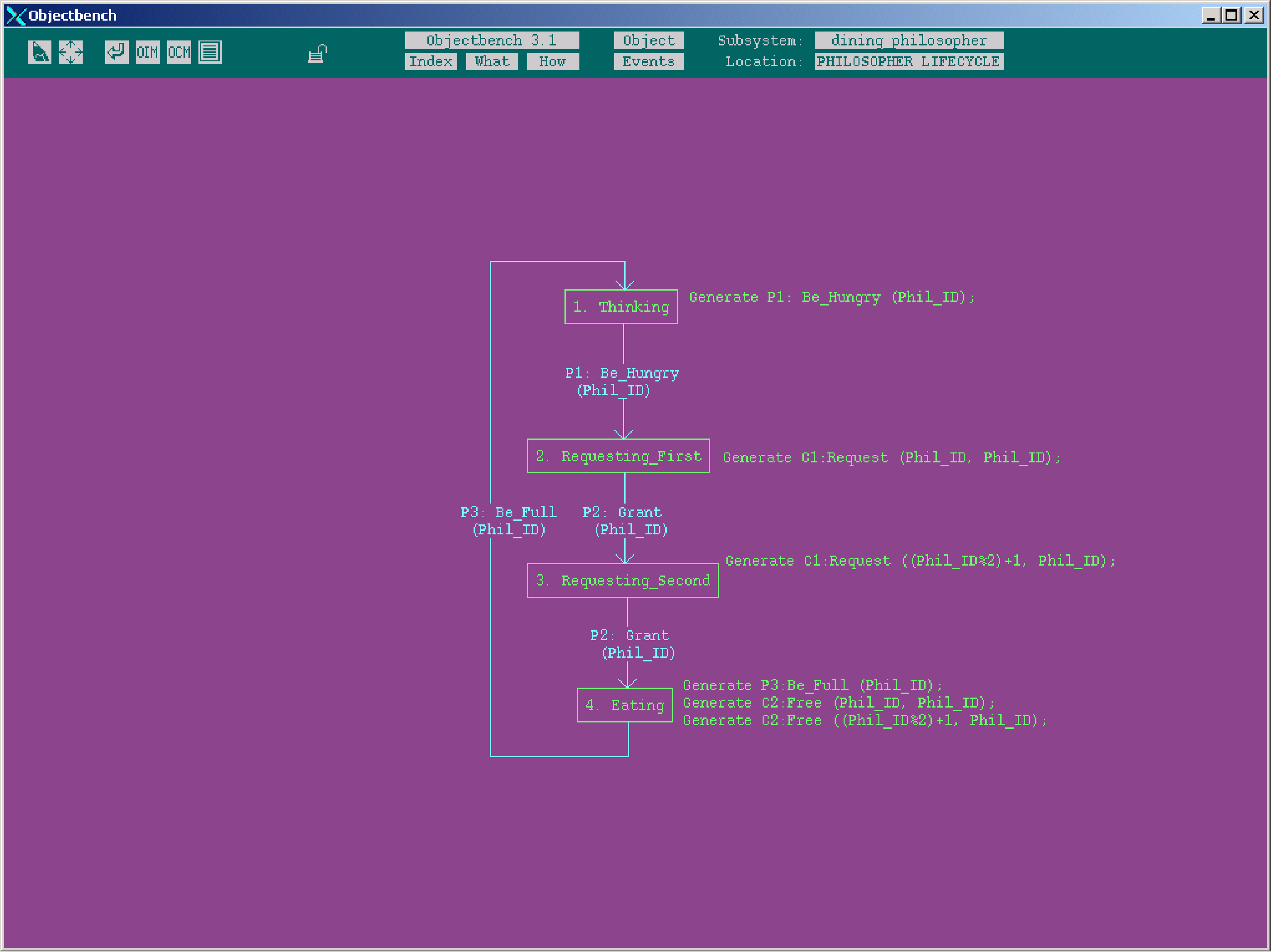
Glimpse at an IDE – Objectbench

1. Graphical capture
2. Execution by discrete event simulator interpretation of program.
3. Early dialect of xUML
4. Dining Philosophers Problem



```
1. PHILOSOPHER (P)
* Phil_ID
```

```
2. CHOPSTICK (C)
* Chop_ID
. Phil_Waiting
```

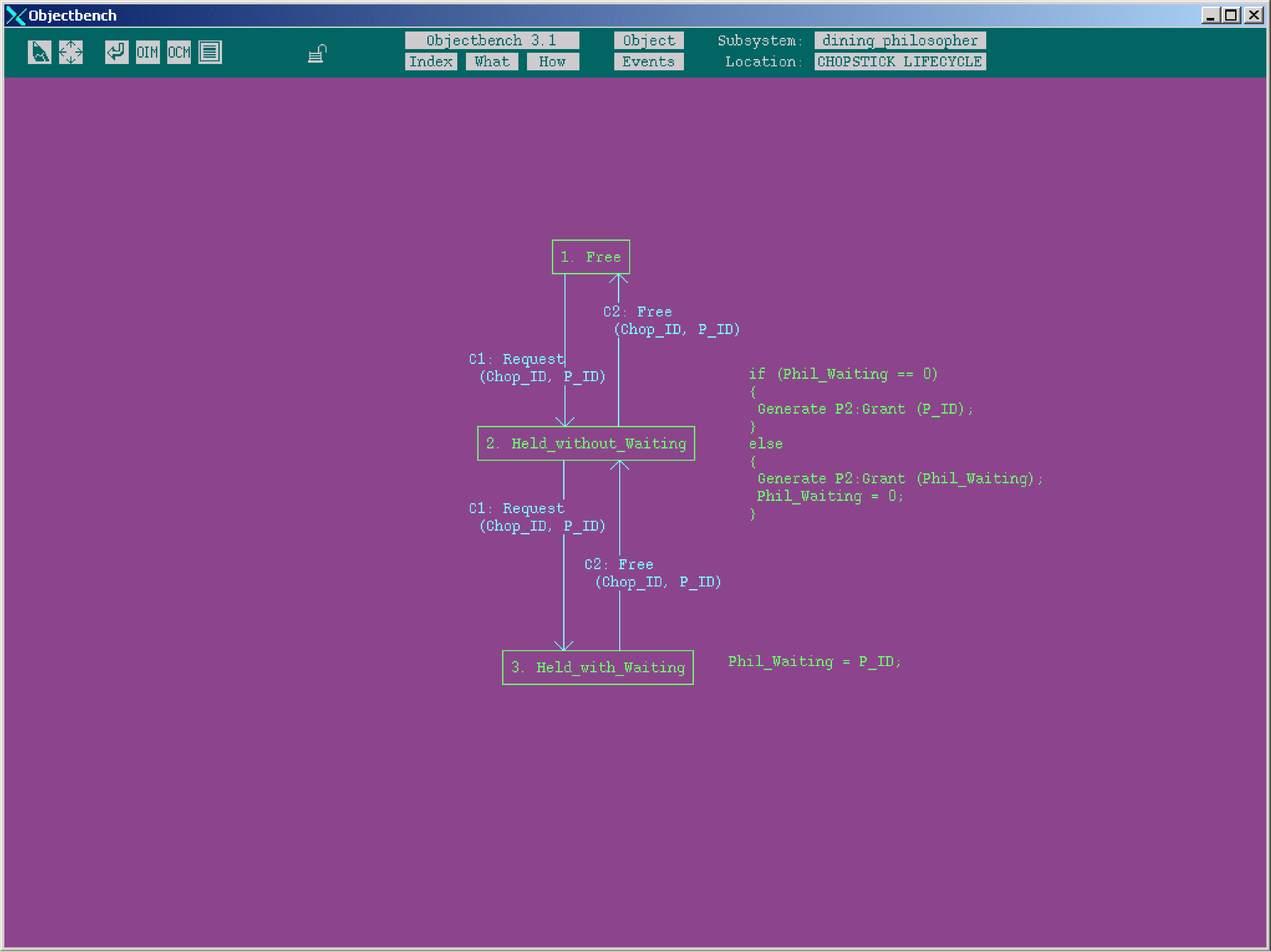


```
Generate P1: Be_Hungry (Phil_ID);
```

```
Generate C1:Request (Phil_ID, Phil_ID);
```

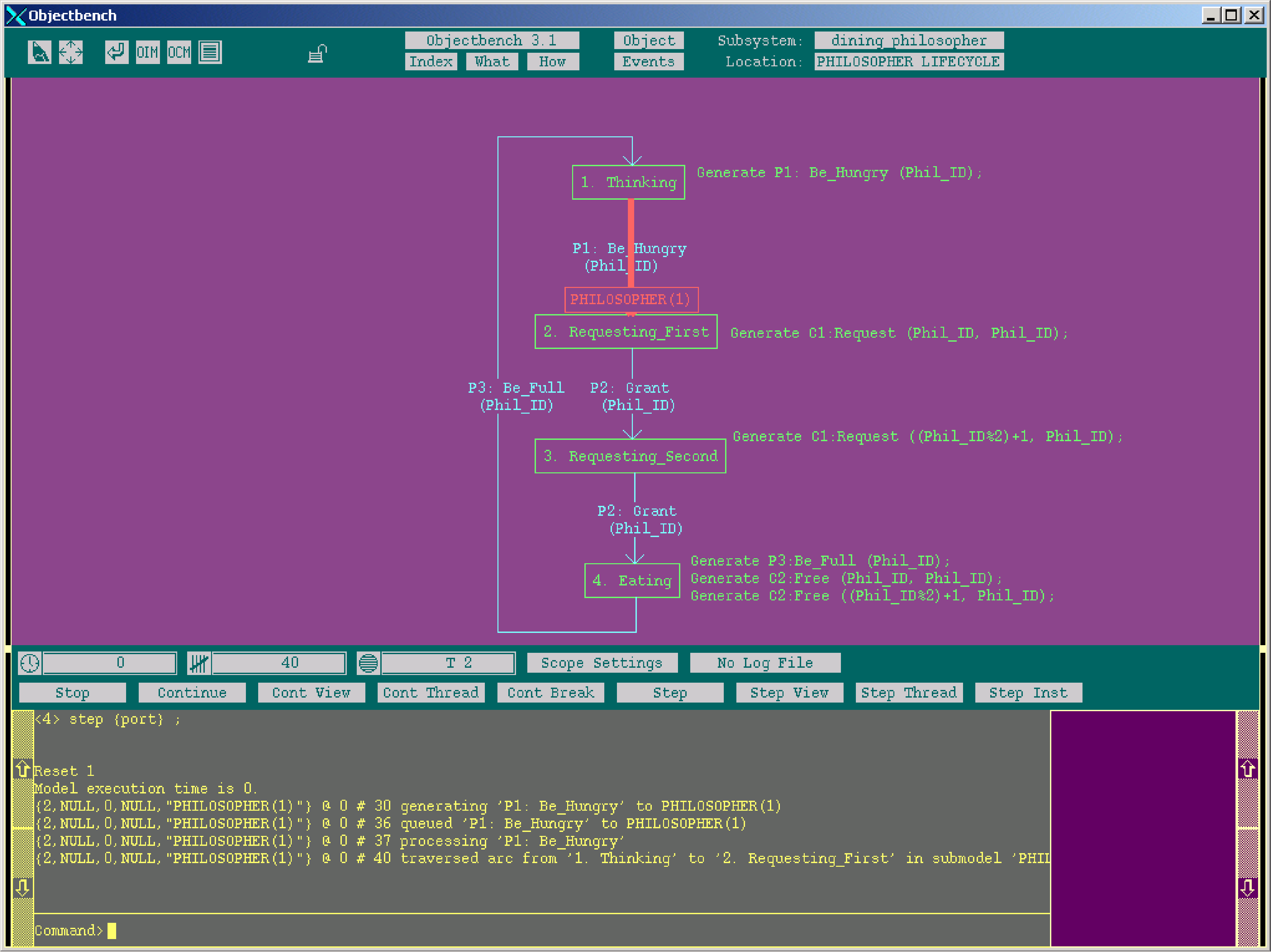
```
Generate C1:Request ((Phil_ID%2)+1, Phil_ID);
```

```
Generate P3:Be_Full (Phil_ID);
Generate C2:Free (Phil_ID, Phil_ID);
Generate C2:Free ((Phil_ID%2)+1, Phil_ID);
```



```
if (Phil_Waiting == 0)
{
 Generate P2:Grant (P_ID);
}
else
{
 Generate P2:Grant (Phil_Waiting);
 Phil_Waiting = 0;
}
```

```
Phil_Waiting = P_ID;
```



```
<4> step {port} ;


Reset 1
Model execution time is 0.
{2,NULL,0,NULL,"PHILOSOPHER(1)"} @ 0 # 30 generating 'P1: Be_Hungry' to PHILOSOPHER(1)
{2,NULL,0,NULL,"PHILOSOPHER(1)"} @ 0 # 36 queued 'P1: Be_Hungry' to PHILOSOPHER(1)
{2,NULL,0,NULL,"PHILOSOPHER(1)"} @ 0 # 37 processing 'P1: Be_Hungry'
{2,NULL,0,NULL,"PHILOSOPHER(1)"} @ 0 # 40 traversed arc from '1. Thinking' to '2. Requesting_First' in submodel 'PHIL

Command>
```

Executable UML

- What is Executable UML
- Executable UML Model
- Domain Modeling
- Classes
- Actions
- Constraints
- Lifecycles
- Signals and Events
- Synchronization
- Model Compilers
- Summary
- References

What is Executable UML?

- Executable UML is a program in a language more abstract than typical procedural languages.
- Subset of UML + Action Language Specification
- Describes data and the behavior
- Doesn’t make coding decisions
- Can be deployed in various software environments without change
- Makes use of Model compiler to generate code

Executable UML Model

- An Executable UML model comprises:
  - UML class diagrams
  - UML state charts
  - Set of procedures, where each procedure is a set of actions.
- Other UML diagrams can be used to support the construction of Executable UML models.

Domain Modeling

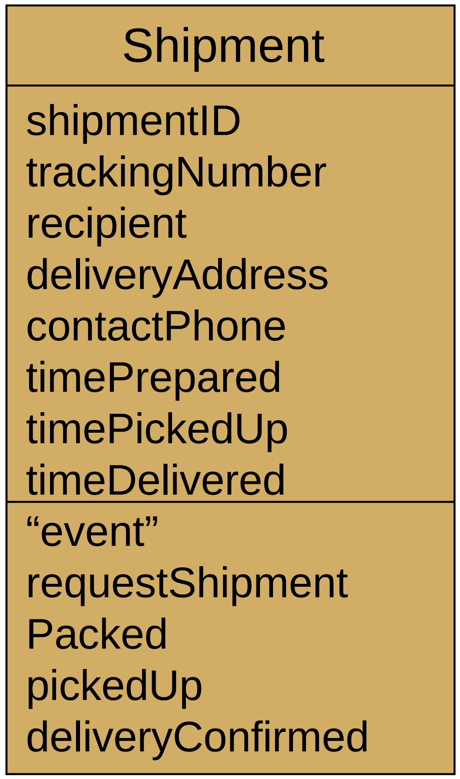
- An executable UML model is to be built for each subject matter, or domain in the system.
- The functional requirements of the system can be expressed in terms of use cases
- For a domain, a class diagram is defined comprising classes, attributes and associations and description for each element.
- A state machine formalizes a lifecycle of an object in terms of states, events, and transitions.
- Each state on the state chart diagram has an associated procedure.
- Each procedure comprises a set of actions.

Classes

- Class diagram in an xUML model comprises:
  - Classes: class’s name, attributes, and events
  - Generalization and Specialization relationships
    - All superclasses are tagged {abstract}
    - All generalizations are tagged {disjoint, complete}
    - Multiple generalization is permitted, but no diamond generalizations
  - Associations: association’s name, multiplicity, and roles.
    - There can be two or more associations between the same two classes.

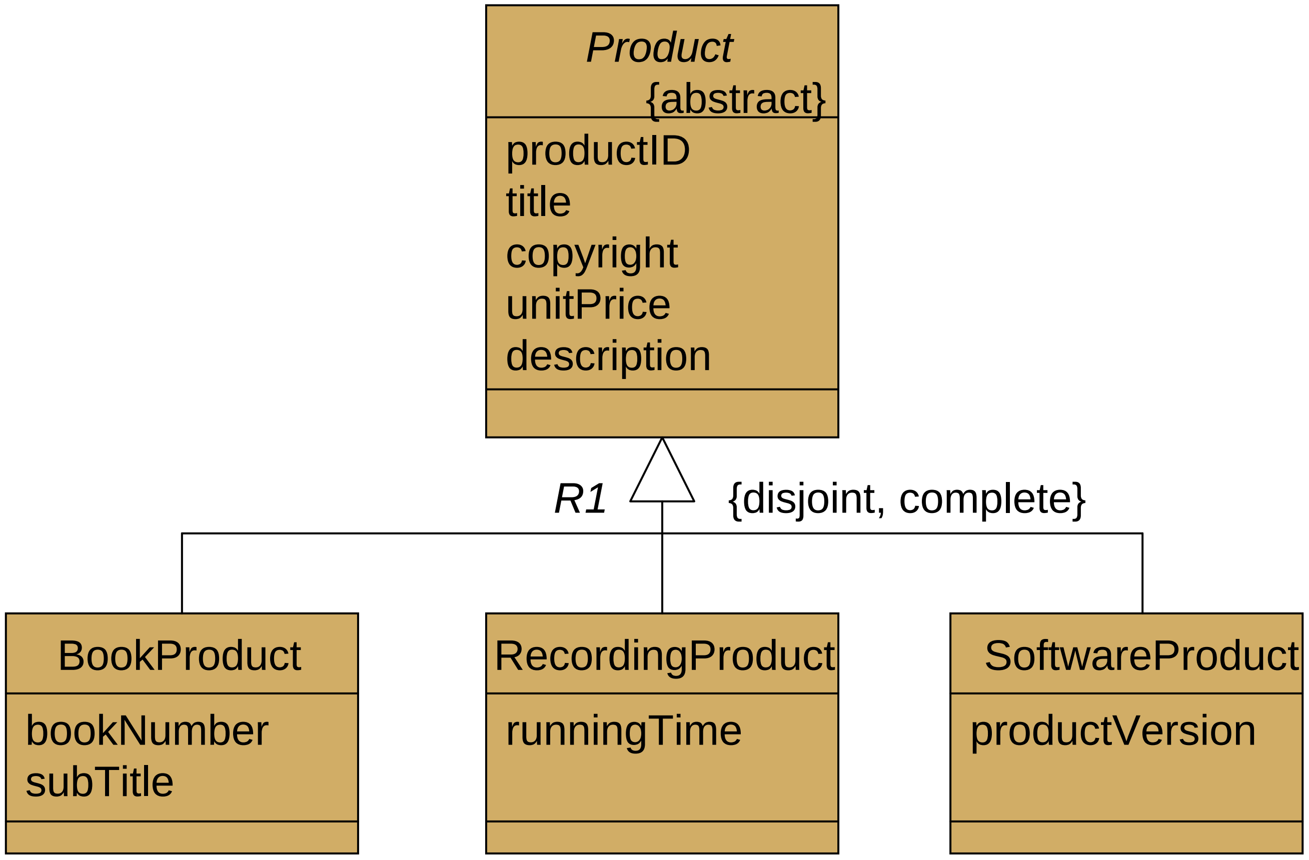
Examples

Class



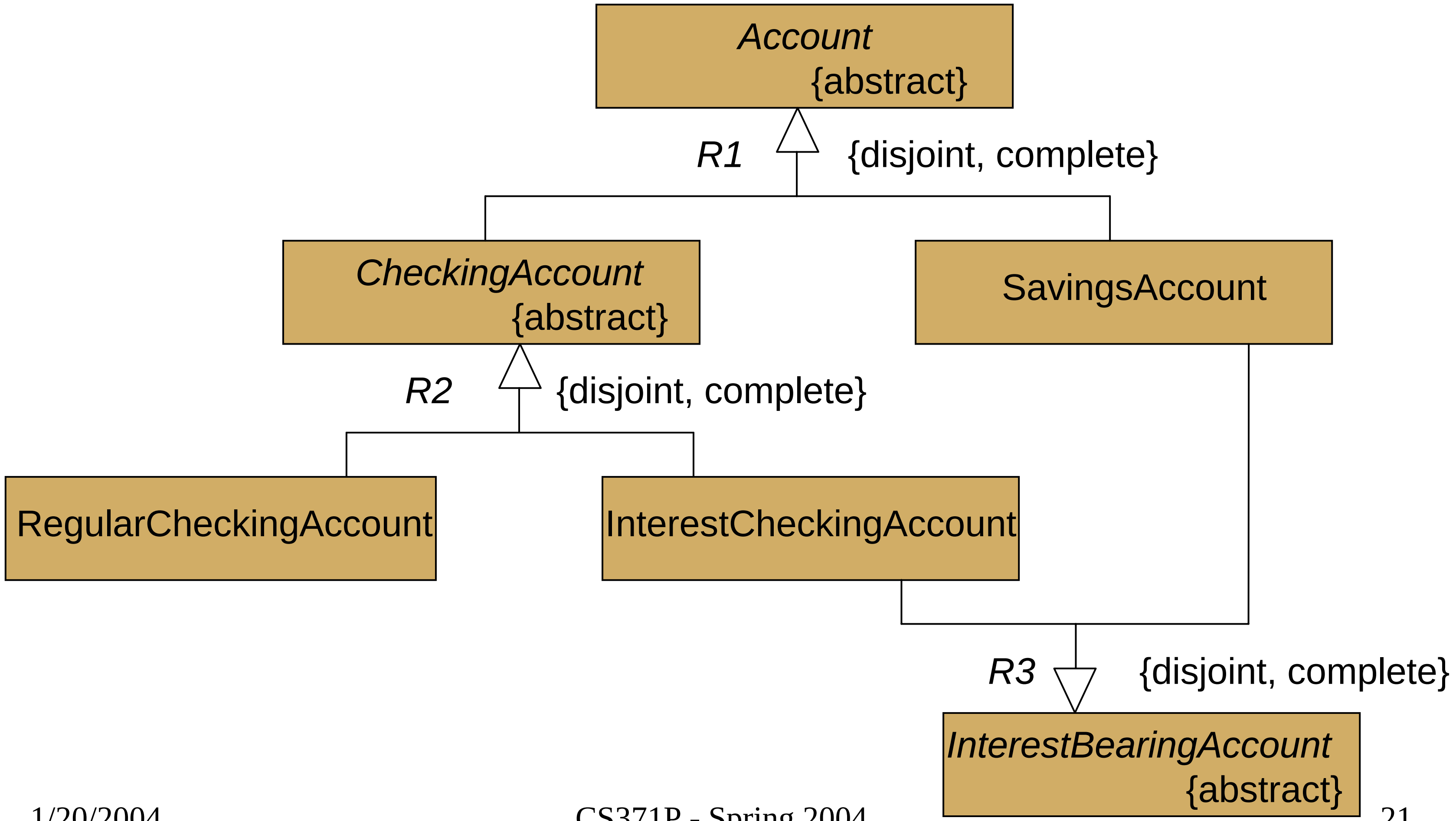
Examples

Generalization



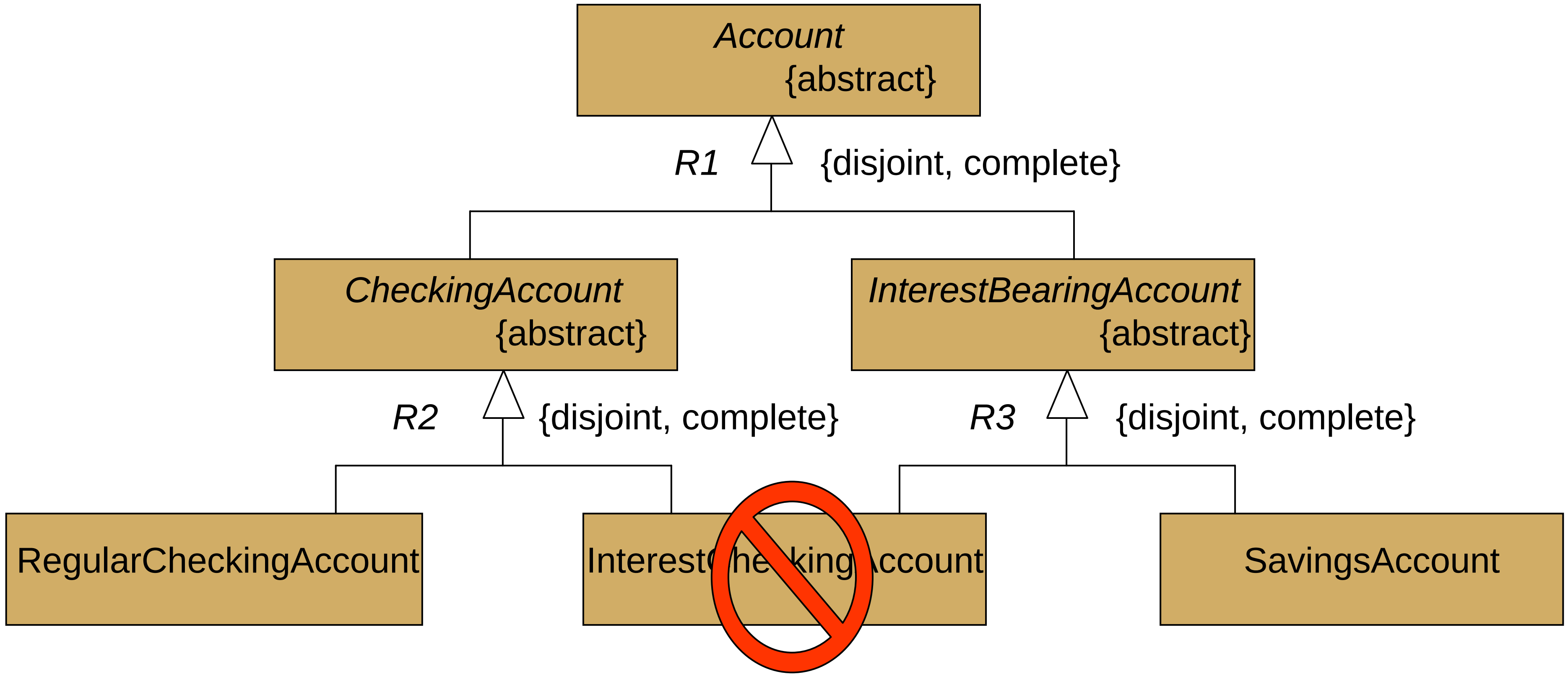
Examples

Multiple Generalization



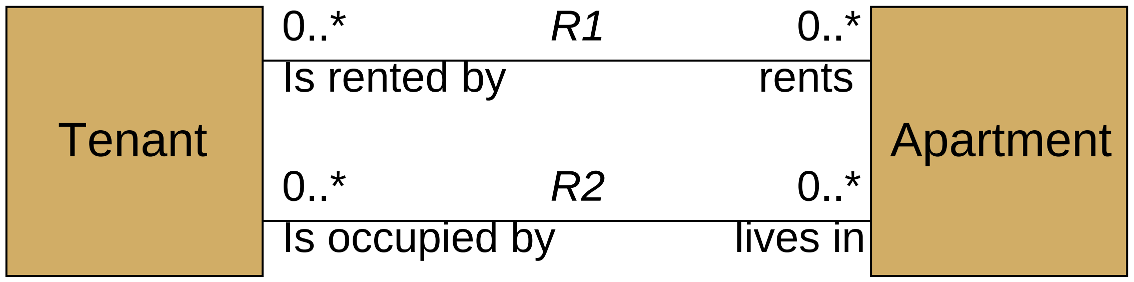
Examples

Improper Multiple Generalization



Examples

Associations



Actions

- An action is an individual operation that performs a single task on an element
- Executable UML relies on the Precise Action Semantics for UML adopted by OMG
- xUML uses BridgePoint Object Action Language
- Classes as well as associations have actions for creating and deleting instances. A reclassification action allows a subclass to move from one leaf subclass in the specialization hierarchy to another
- Other Action Languages: SMALL and TALL

Actions Syntax

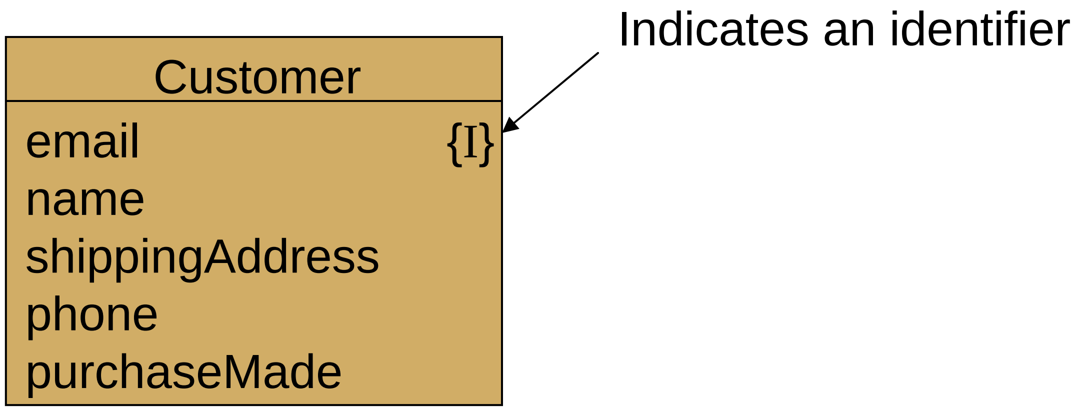
Action	Syntax
Create object	**create object instance** <object reference> **of** <class>;
Write attribute	<object reference>.<attribute name> = <expression>;
Delete object	**delete object instance** <object reference>;
Class extent	**select many** <object reference set> **from instances of** <class>;
Qualification (1)	**select any** <object reference> **from instances of** <class> **where** <clause>;
Qualification (+)	**select many** <object reference set> **from instances of** <class>**where**<clause>;
Loop	**for each** <object reference> **in** <object reference set> <statements> **end for**;
Create link	**relate** <object reference> **to** <object reference> **across** <association>;
Traverse link	**select [one|many]** <object reference> **related by** <object reference>-><class>[<association>];
Delete link	**unrelate** <object reference> **from** <object reference> **across** <associattion>;
Reclassify object	**reclassify object instance** <object reference> **from** <class> **to** <class>;

Constraints

- A constraint is a rule that restricts the value of attributes and/or associations in a model
- Constraints can be expressed in OCL, action language or graphical notation
- Unique Instance Constraints: An identifier is a set of one or more attributes that uniquely distinguishes each instance of a class:
  - Single Attribute Identifier
  - Multiple Attribute Identifier
- Referential Constraints: A referential attribute identifies the instance of associated class

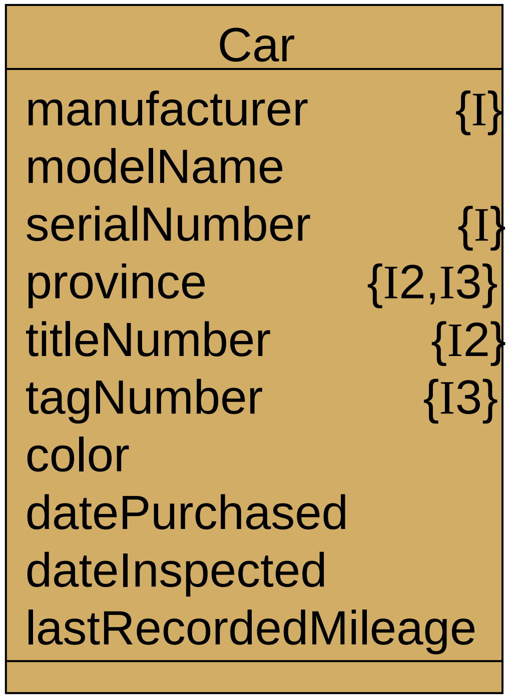
Examples

Single Attribute Identifier



Examples

Multiple Attribute Identifiers



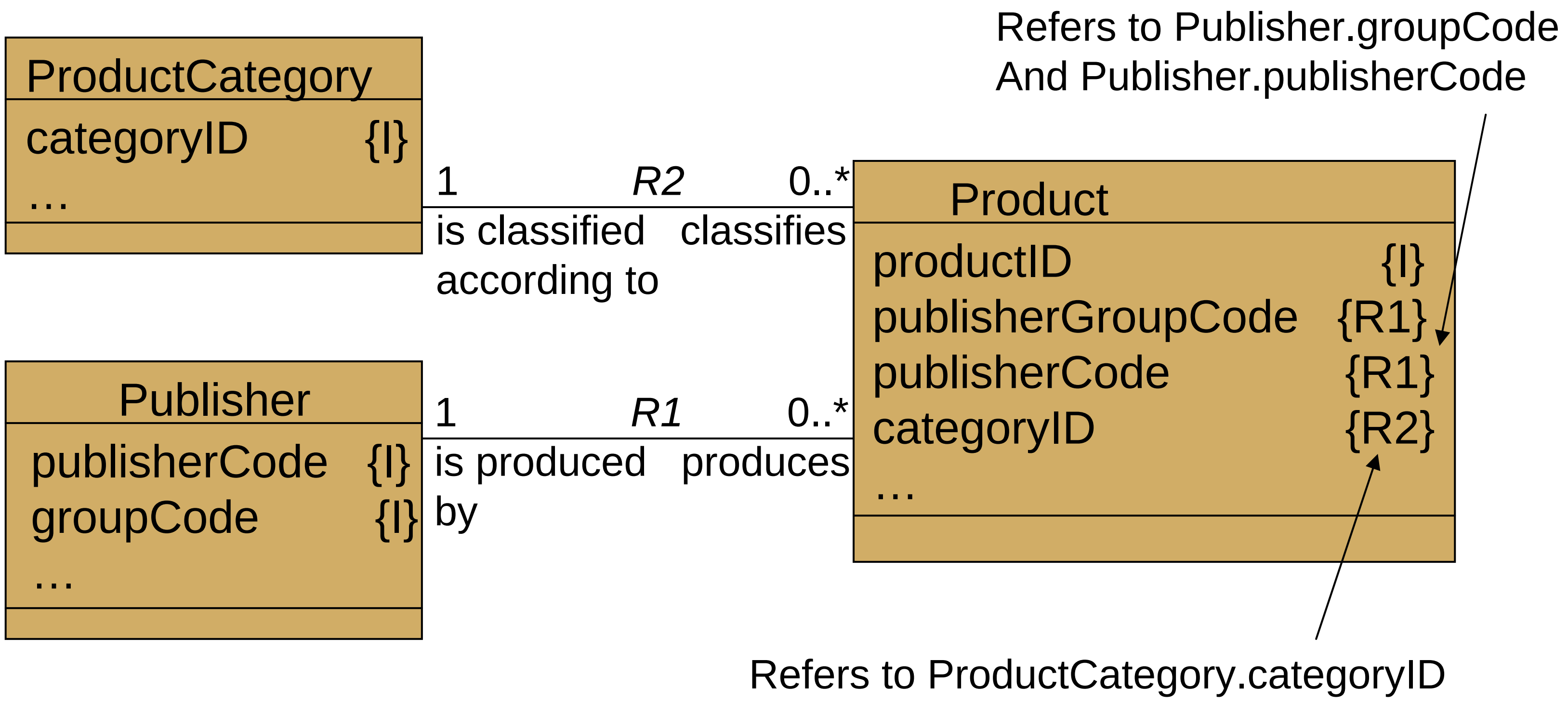
Identifier 1:
manufacturer + serial number

Identifier 2:
province + titleNumber

Identifier 3:
province + tagNumber

Examples

Referential Constraints

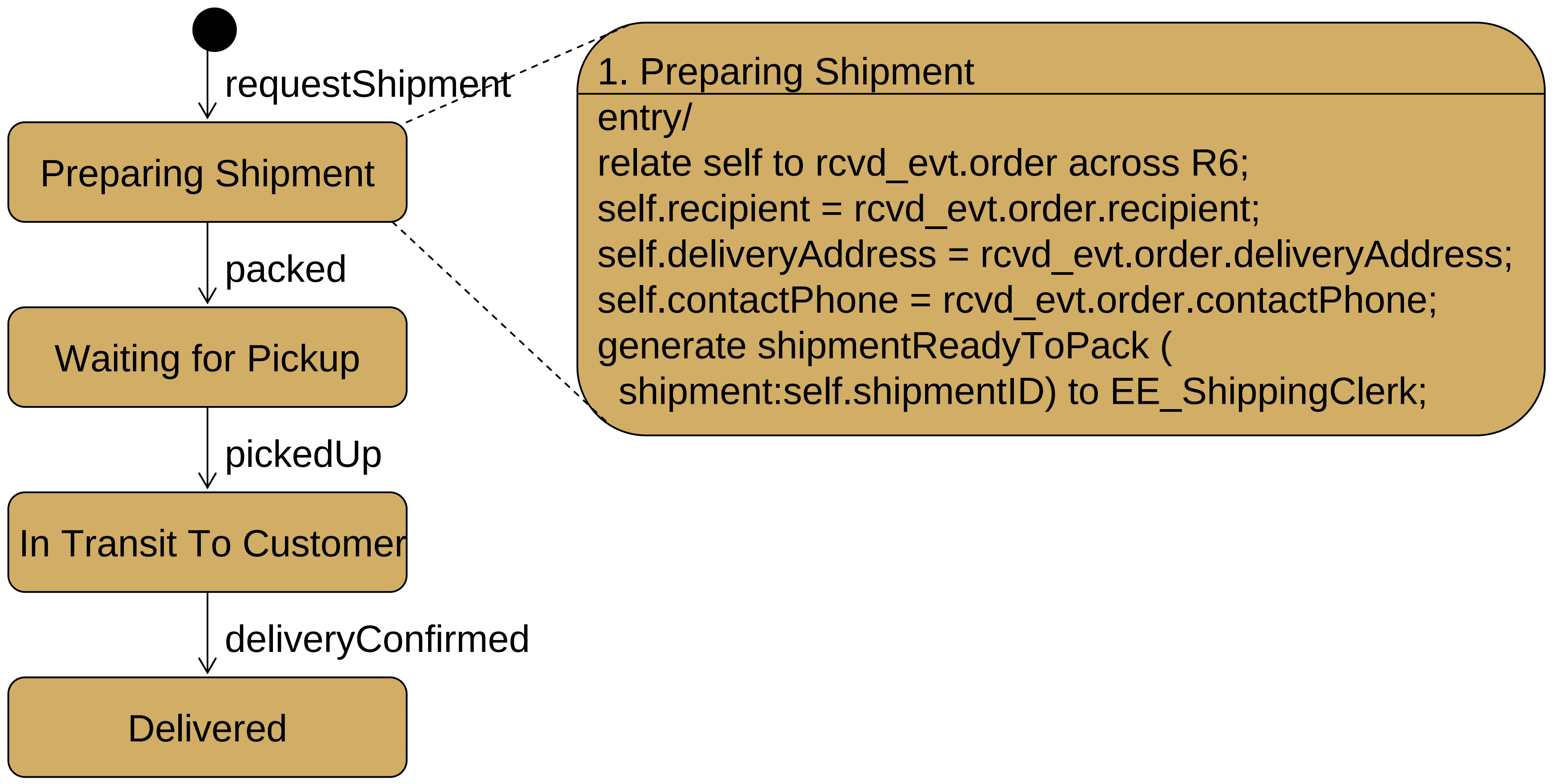


Lifecycles

- Each instance of a class generally has a lifetime. Its behavior is a progression through various stages over time. This is known as the lifecycle of the object
- A lifecycle is formally expressed as a state machine comprising states, events, transitions and procedures
- A state machine can be expressed in terms of a State Transition Table where each row represents a state and each column an event.
- Filling the State Transition Table can result in discovering new states, events and/or transitions

Example

Shipment Class Lifecycle (state chart)



```
1. Preparing Shipment
entry/
relate self to rcvd_evt.order across R6;
self.recipient = rcvd_evt.order.recipient;
self.deliveryAddress = rcvd_evt.order.deliveryAddress;
self.contactPhone = rcvd_evt.order.contactPhone;
generate shipmentReadyToPack (
  shipment:self.shipmentID) to EE_ShippingClerk;
```

Signals and Events

- The behavior of the system is sequenced by signals between state machine instances
- Objects communicate as a result of state machine instances sending signals
- A signal is a message that carry data
- Signaling is asynchronous
- Once an event is detected by the receiver, a transition is made and the receiver executes a procedure
- Event parameters can be viewed as an object rcvd_evt created dynamically by the sender

Example

Signal with Parameter:

```
Generate addSelection(productID:rcvd_evt.productID,
            quantity:rcvd_evt.quantity) to order;
```

Signal to external entity:

```
Generate requestChargeApproval(…) to EE_creditCardCompany;
```

Creating object by signaling:

```
generate requestShipment (order:self) to Shipment creator
```

Synchronization

- In executable UML time is local to each object
- There is no global time and no global synchronization mechanism
- An object synchronizes its behavior with another by sending signals
- Executable UML provides rules for signals and procedures that are designed to describe required synchronization between objects
- It's the job of the model compiler to ensure these rules by whatever mechanism
- It's the developer responsibility to avoid data access conflict by imposing some rules to the modeler or to the model compiler

Model Compilers

- An executable UML model compiler turns an executable UML model into an implementation using a set of decisions about the target hardware and software environment
- Model compilers can be extremely sophisticated
- There are many possible executable UML model compilers for different system architectures
- The choice of the model compiler can be based on the performance requirements and the environment of the application
- Example of an existing model compiler is the transaction safe system with rollback available from Kabira technologies
- A model compiler comprises a set of mechanisms that manages the runtime system, and a set of rules for how to weave the Execution UML models together

Summary

- Executable UML is a profile of UML
- The Executable UML presented here is not the only possible Executable UML
- Executable UML is not a standard
- It relies on model compilers to generate executable code
- Executable UML models are separate from any implementation, yet can readily be executed to test for completeness and correctness
- The choice of a model compiler affects the performance of the generated code