Getting Started with Unified Modeling Language

by Carolyn Duby
Pathfinder Solutions, Inc.
www.pathfindersol.com
508-384-1392

ESC Spring 2000
Class 307 and 317

WHAT IS THE UNIFIED MODELING LANGUAGE?

The Unified Modeling Language (UML) is the standard graphical language for specifying the analysis and design of object-oriented software. The UML defines a set of diagrams, also called models, with well-defined graphical symbols so developers can easily understand each other's diagrams. The graphical symbols are augmented by textual descriptions that also aid in model comprehension. The Object Management Group (OMG) maintains the UML standard.

The UML is a graphical notation only. You must choose a software development process that defines which diagrams to create and the order in which to create them. The UML is very expressive and effectively supports a variety of software development processes and applications from real-time to information systems. Most processes use a subset of UML constructs that best model the subject matter of the system. For example, reactive real-time systems will make extensive use of Statechart Diagrams while database systems will focus on the data modeling provided by Class Diagrams.

FUNDAMENTAL CONCEPTS

Classes and objects are two fundamental concepts in UML. A class is an abstraction of a set of real-world things that all have the same data and behavior. A class may have many instances called objects. Each object conforms to the rules defined by its class. For example, a vehicle class in an automated toll collection application represents the concept of a car or truck that will be charged when passing through a tollbooth. Each vehicle has a VIN number, make, and model. There will be a vehicle object for each car and truck registered with the toll collection system. Each vehicle will have a unique VIN number and a make and model for the particular car or truck.

Attributes define the data encapsulated in the class. Services, functions that act on attributes, describe synchronous class behavior. Asynchronous behavior is described in state machines, life cyles that determine how an object responds to an event. When modeling a class, only include the attributes, services, and states required by the application. For example, the vehicle class in the toll collection system does not capture the color of the vehicle because it is not important for calculating the toll amount.

Classes cooperate with one another to satisfy the requirements of the system. Associations link classes together. Classes communicate with one another by passing events and invoking services.

UML ARTIFACTS

UML defines nine diagram types that document the system software from different viewpoints and at different points in the software process. Some diagrams are very abstract showing a high-level view of the system without much detail. Others show a lot of detail about of a small part of the system. UML diagrams show a higher-level view of the system that is hard to grasp when looking at just the code.

The nine diagram types are as follows:

- Class Diagram – Shows classes, packages, and their associations. Describes the static data used by the system. Shows the separation of the system into packages.

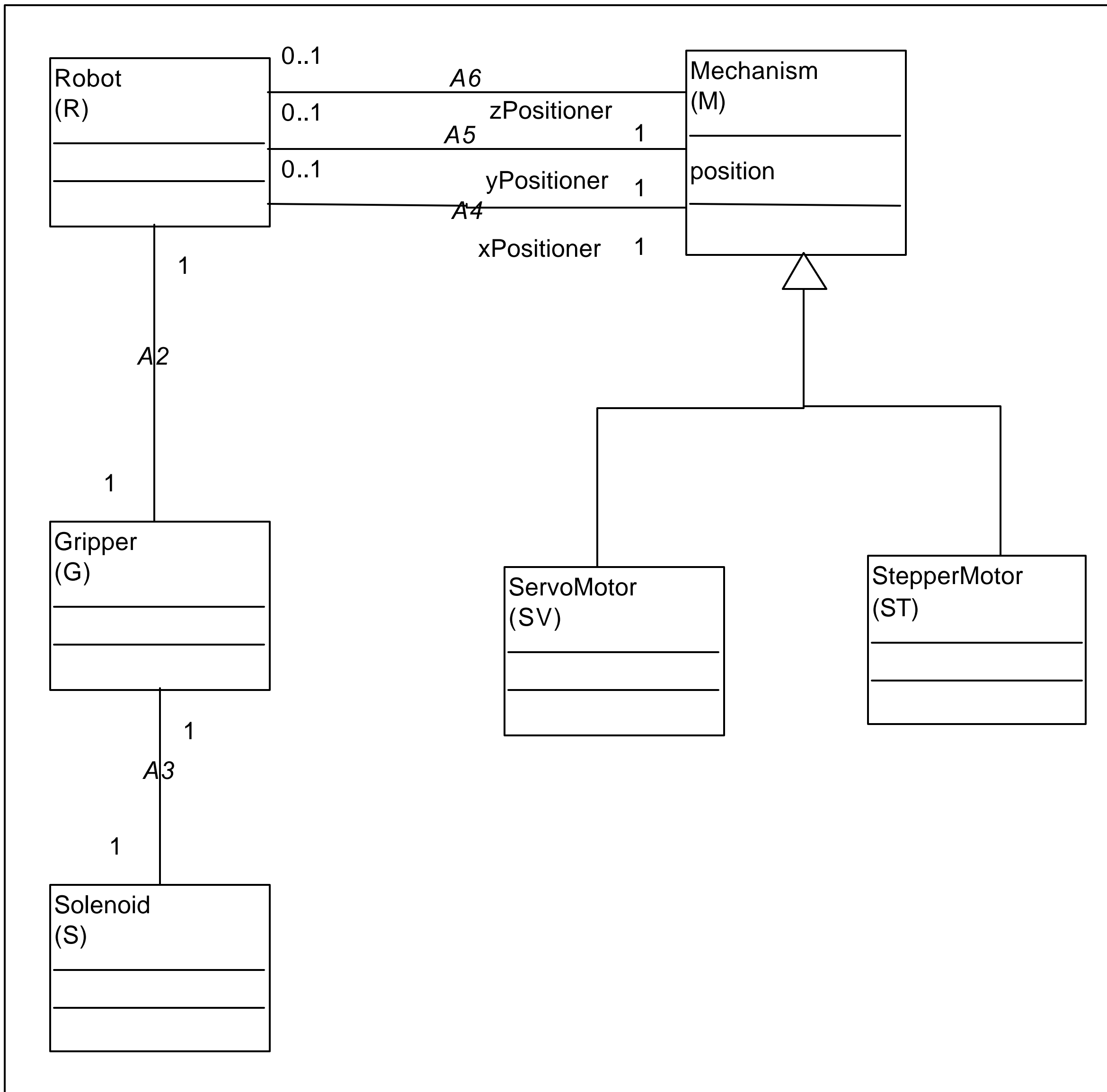

Diagram 1 – Class Diagram Example

- <u>Object Diagram</u> – Shows snapshot of objects and their relationships at a particular time during the execution of a system. Useful for documenting complex data structures and object topology.

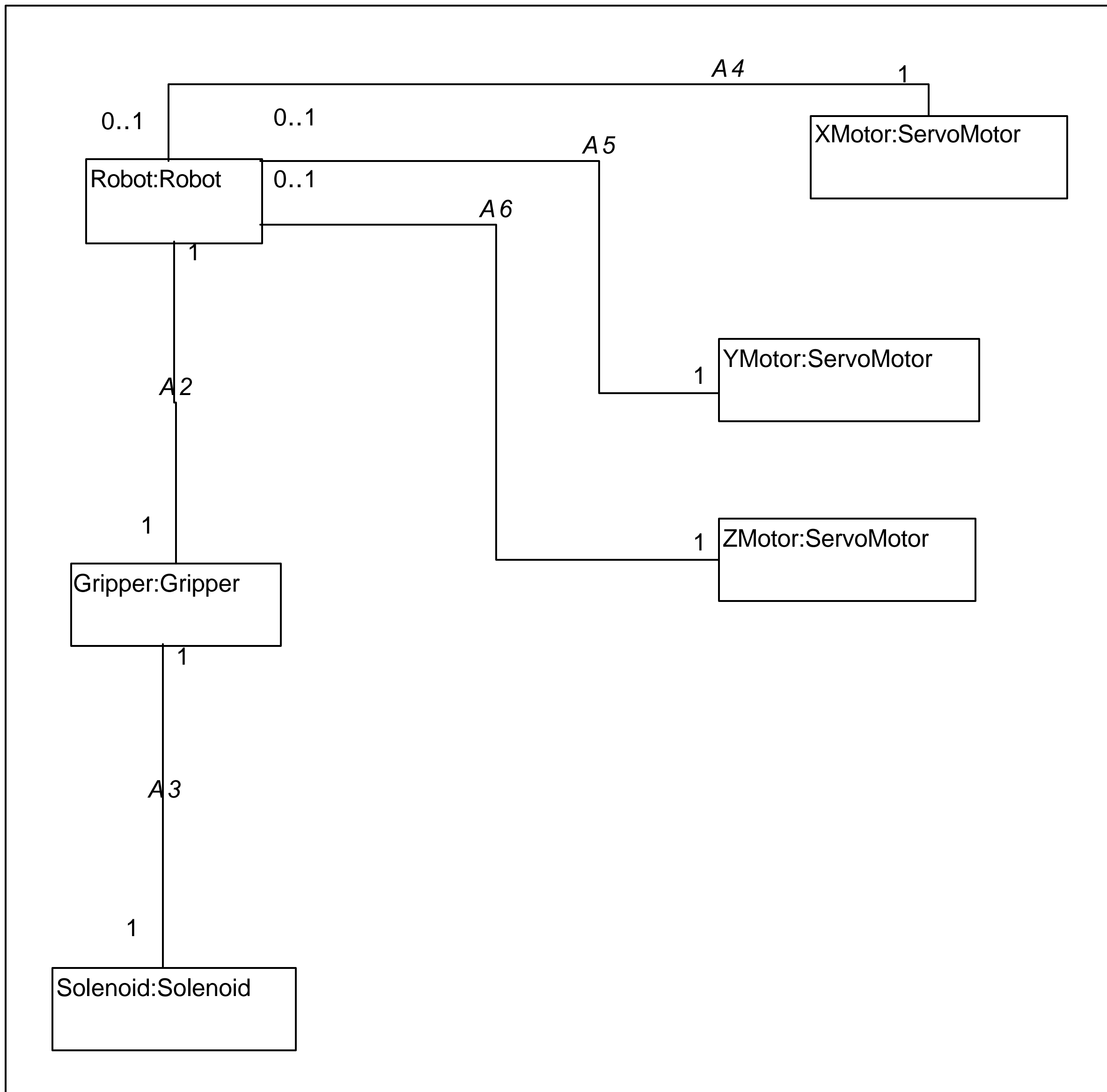

Diagram 2 – Object Diagram Example

- Use Case Diagram – Shows relationships between use cases and actors. Describes context of the system. May be used to formalize and develop behavioral requirements.

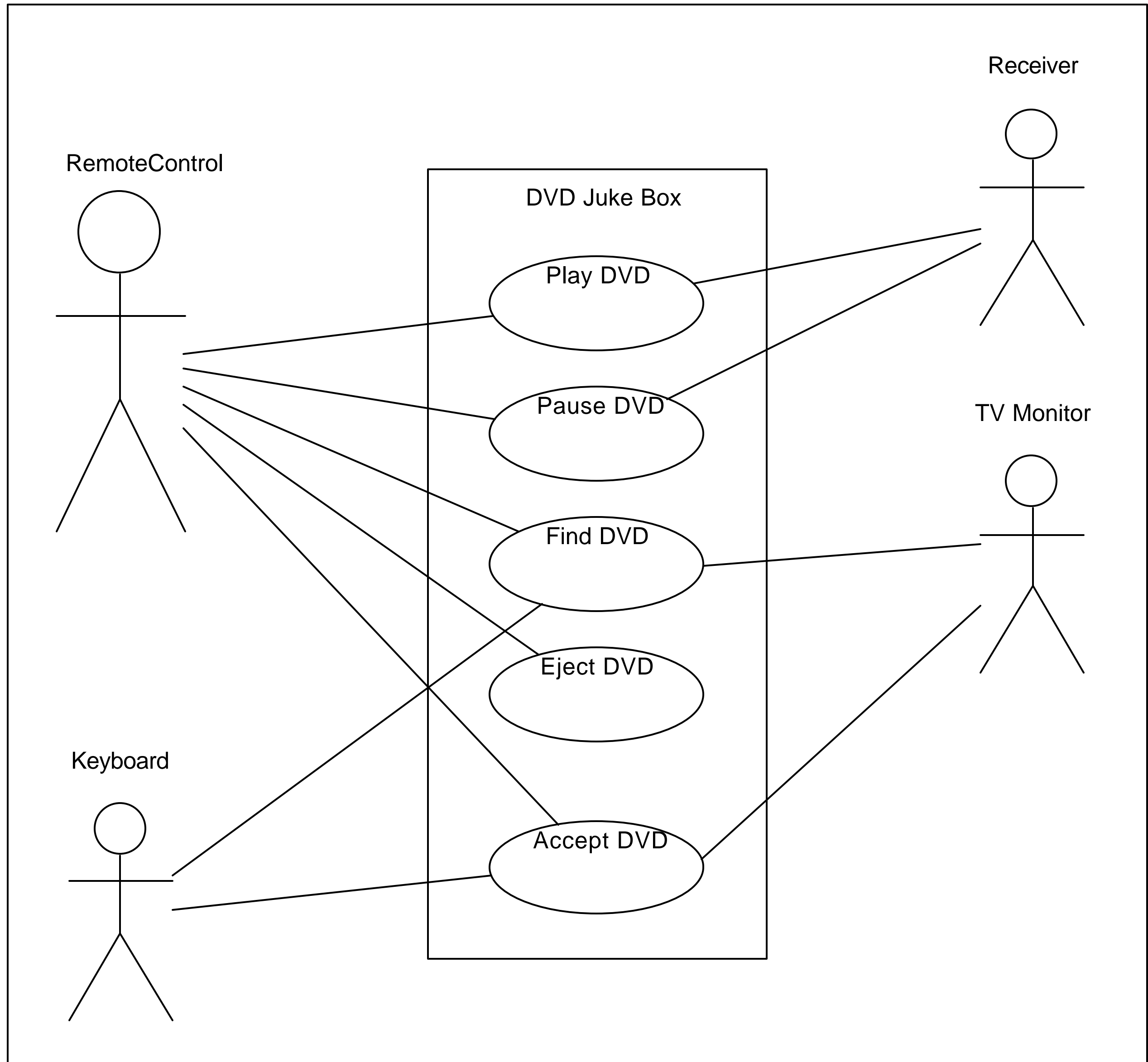

Diagram 3 – Use Case Diagram Example

- Sequence Diagram – Shows order of events exchanged by objects during the execution of a scenario. Shows how objects collaborate to implement the behavior of the system. Shows the same information as the Collaboration Diagram. Format highlights timing.

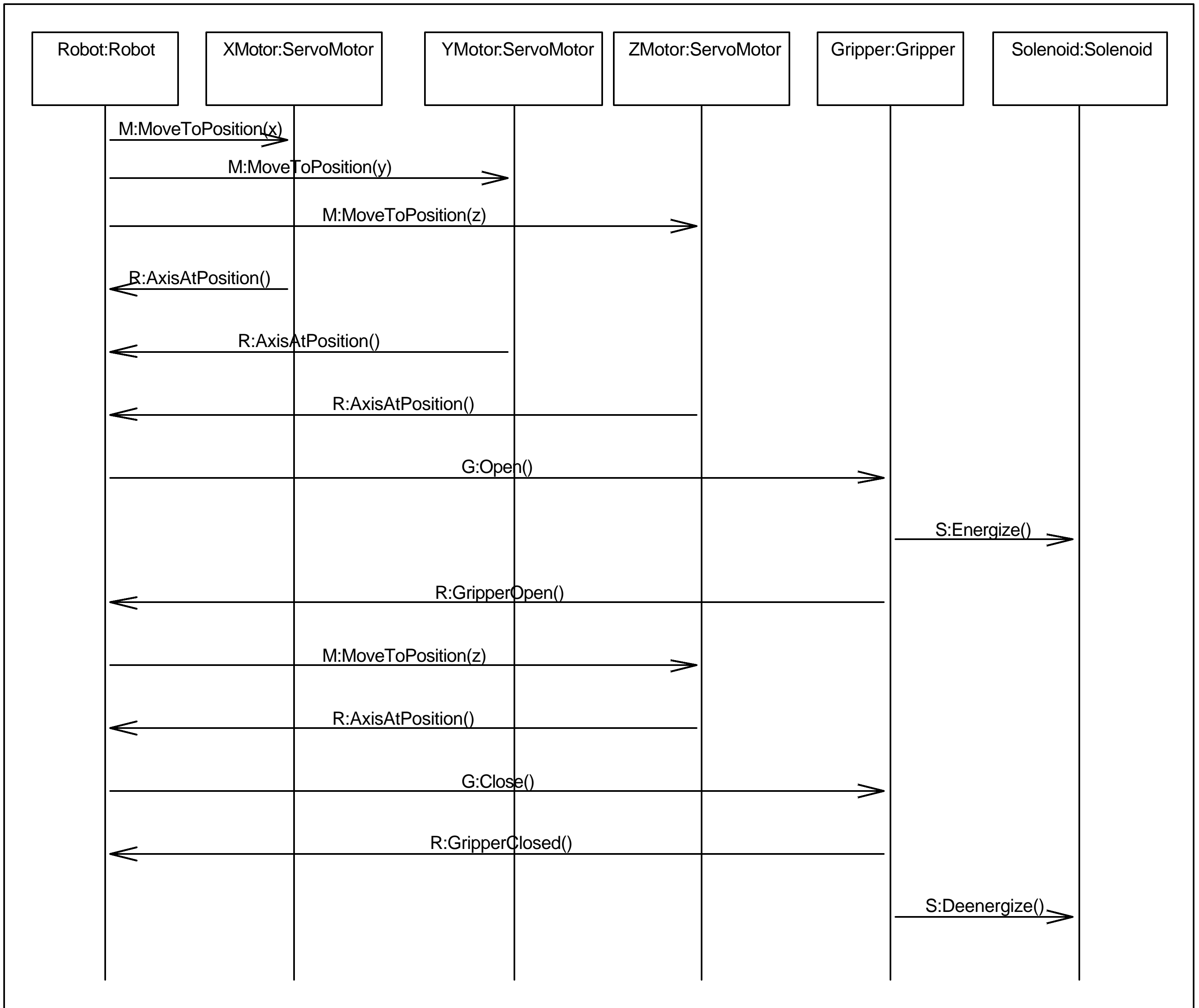

Diagram 4 – Sequence Diagram Example

- Collaboration Diagram – Shows the same information as the Sequence Diagram in a different format. Lines labeled with events connect object nodes. Format highlights the relationships between objects.

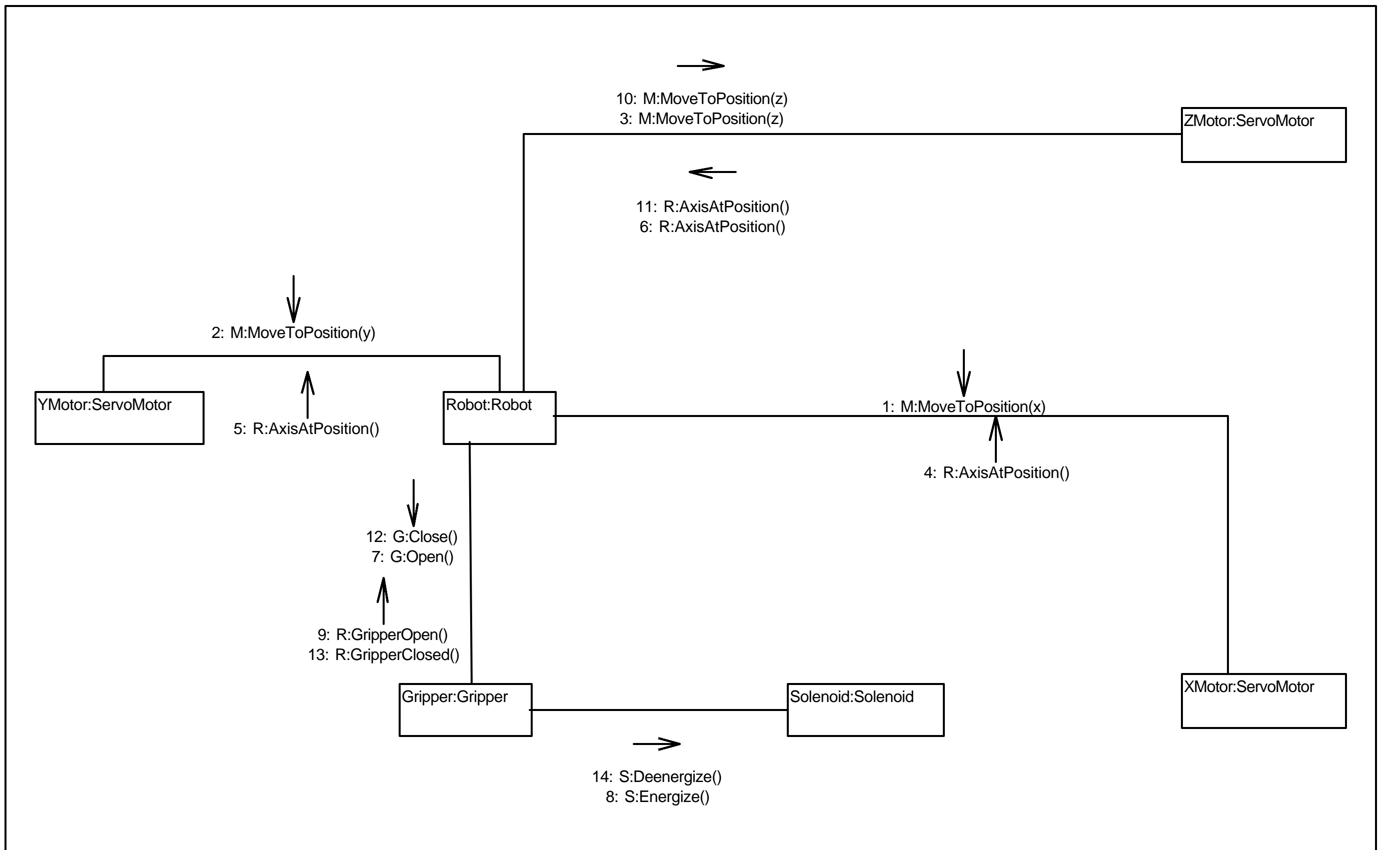

Diagram 5 – Sequence Diagram Example

- <u>Statechart Diagram</u> – Shows the lifecycle of a class as a state machine. Shows details of class behavior.

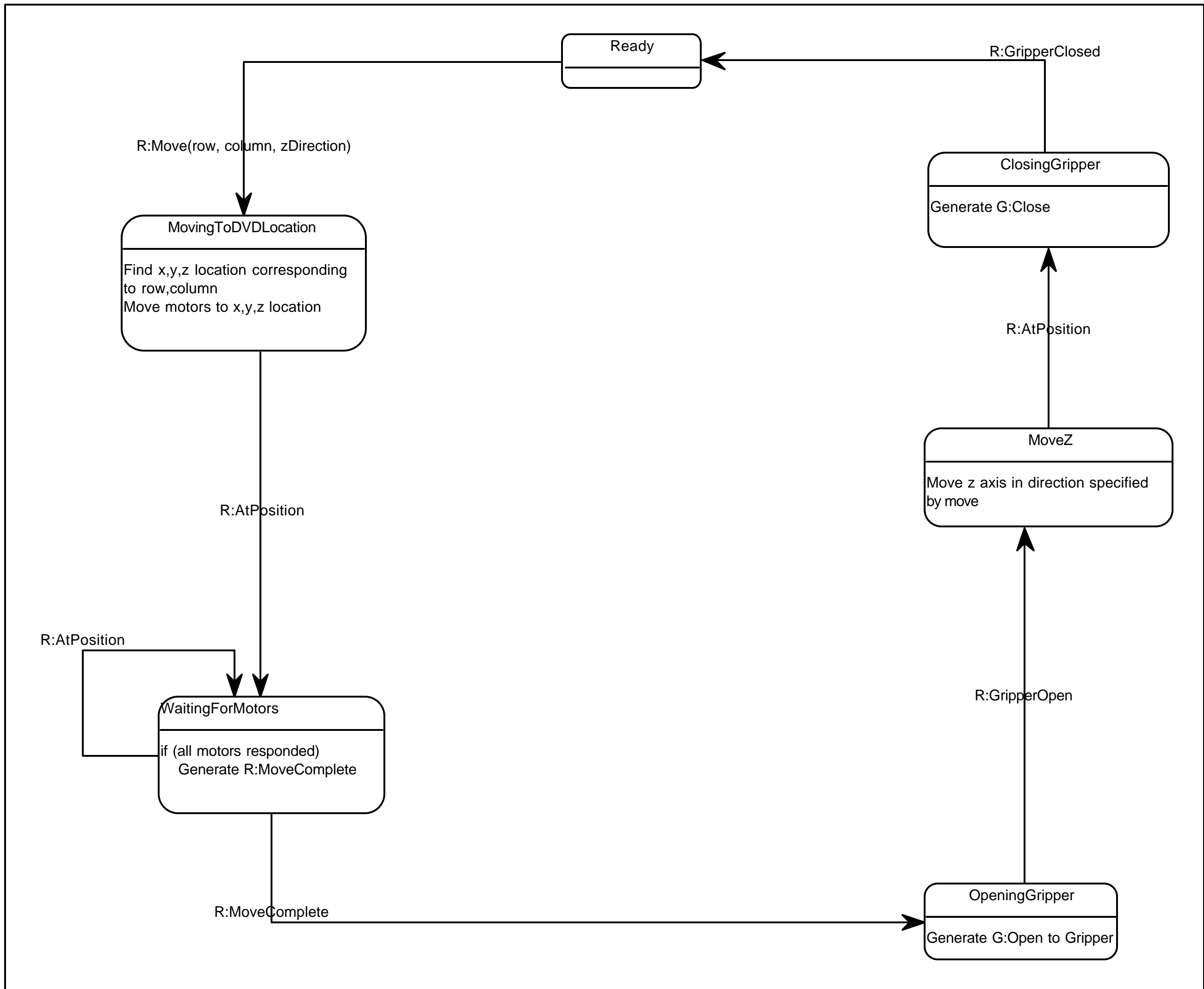

Diagram 6 – Statechart Diagram Example

- Activity Diagram – A special case of a Statechart Diagram used to work flow and model business processes.

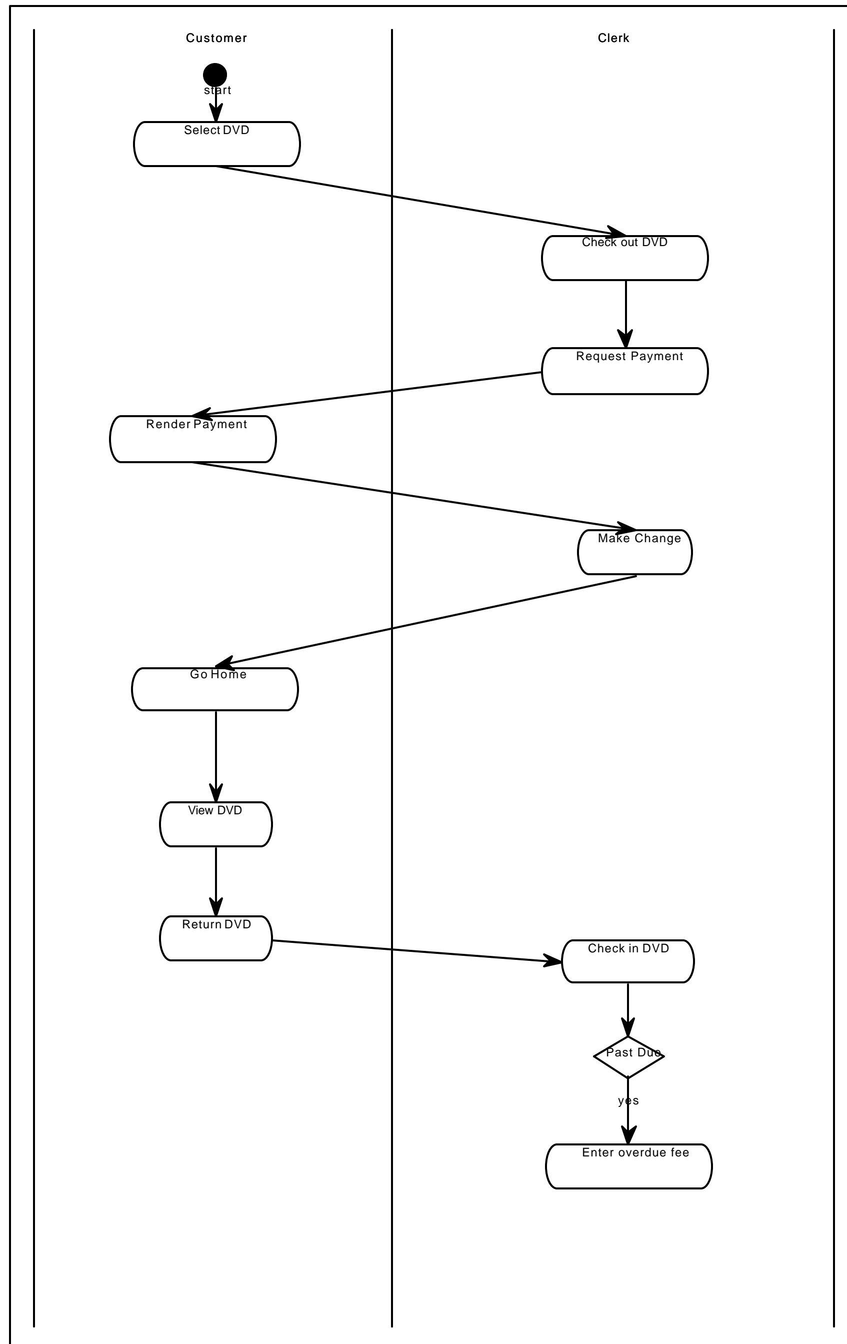

Diagram 7 – Statechart Diagram Example

- <u>Component Diagram</u> – Shows the mapping of the system to physical components such as files, executables, and libraries.

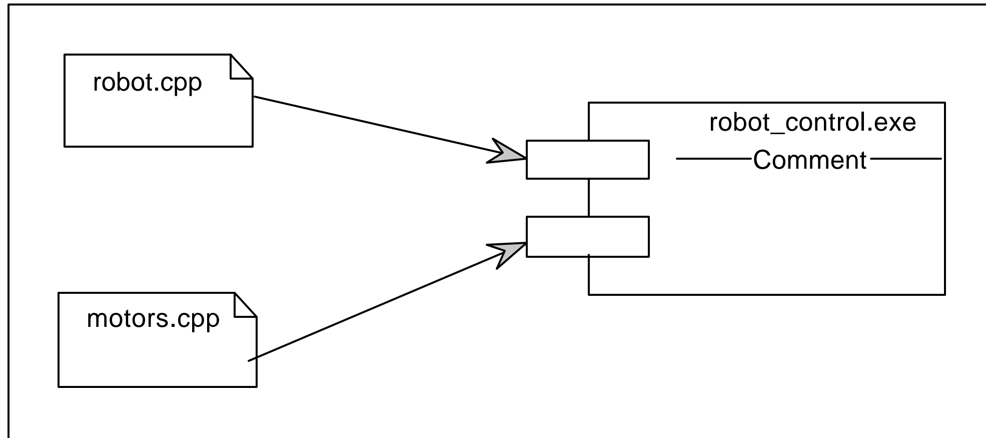

Diagram 8 – Component Diagram Example

- Deployment Diagram – Shows topology of the hardware executing the system.

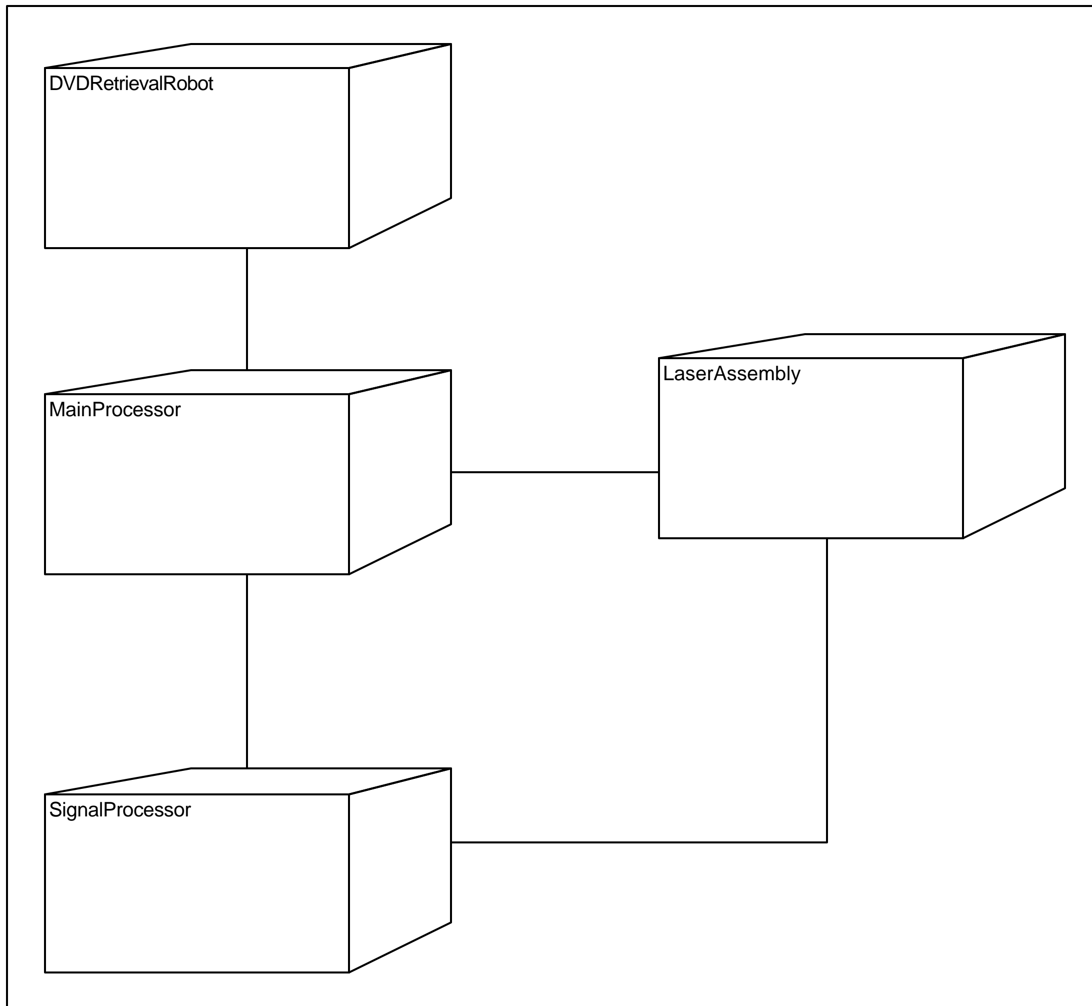

Diagram 9 – Deployment Diagram Example

The rest of this presentation will focus on the diagrams used in the analysis of the software for a system: Class Diagram, Sequence Diagram, and Statechart Diagram. We will assume that the requirements and Use Case Diagrams are complete and that design will follow using either an elaborative or translational process.

Many UML analysis constructs can be automatically mapped to an implementation language such as C, C++, or Java. Automatically mapping models to an implementation language can increase developer productivity, improve the quality of the system software, and enable a team to react more quickly to changes in requirements. Automatic mapping has the added advantage that the models and code are always in sync because the code is generated from the models. The effort to create models in the analysis phase will continue to benefit existing and new team members throughout the entire lifecycle.

HISTORY OF UML

Object-oriented methods evolved as software engineers attempted to develop larger and more complex systems using the object-oriented paradigm. By the early 1990's we were in the heat of the method wars. There were as many as 50 methods with the majority of the market share divided between six major players. Developers had the daunting task of evaluating the methods and their supporting tools to determine the best one for their project.

Each method had its strengths and weaknesses but they all shared a fundamental set of concepts. As the evolution continued, methodologists made improvements to their own methods by incorporating and refining ideas from other methods. In 1994, Grady Booch and Jim Rumbaugh, two leading methodologists joined forces, realizing that their methods were going in the same direction and that they would make faster progress by working together. Shortly after, Ivar Jacobson joined Booch and Rumbaugh. A UML consortium formed in 1996 to develop a draft of the standard notation. After input and improvements by OMG members, UML was adopted as a standard by the OMG in 1997. Today industry support is strong and a wide variety of tools and processes designed for use with UML are commercially available.

WHY IS STARTING A NEW SYSTEM SO DIFFICULT?

So what’s the big deal about starting a new system anyway? On the first day you show up for work bright-eyed and bushy-tailed. You read and digest those unambiguous, complete, and measurable requirements given to you, sit down in front of your computer with all the best tools, and build models like a demon. No big deal. Right?

Then you wake up and realize you’ve been having a pleasant daydream at your desk. In the real world, requirements are usually vague. Your project manager wants to know, by tomorrow, that you can meet the delivery date driven by market timing. Additionally, a flock of developers will soon join the project and you are tasked with bringing them up to speed on a project that is not yet defined. Maybe you don’t know exactly how the hardware will work or if certain parts of the system are even possible to

implement. And what about that legacy code whose programmer retired on her company stock options years ago? Should you reuse it or reimplement it?

Given the state of the world, it is not surprising that many projects have trouble getting off the ground. The sheer number of issues can be paralyzing. This paper discusses three Object-Oriented(OO) model development techniques that break the paralysis and build a strong analysis foundation for your system software. Once the foundation is established, the rest of the analysis falls into place.

MANAGING COMPLEXITY WITH DOMAIN MODELING

The first step when starting a project is to reduce the complexity of the system by dividing the system into manageable pieces called domains.

What is a Domain?

A domain is a set of closely related classes that work together to provide services to other domains. A server domain is any domain that provides services required by another domain, the client domain. Domains implement services using a variety of techniques including:

- OO analysis
- hand written code
- code generated from a Rapid Application Development environment
- legacy code
- off-the-shelf libraries

Client domains access the provided services of a server domain through bridges. The bridge is the only interface permitted from one domain to another. The classes in the client domain requesting the service do not have any knowledge of the details or implementation strategy of the server domain providing the service. The client only knows the data inputs and outputs of the bridge service and any expected asynchronous returns. Restricting accesses between domains to bridges reduces the coupling between parts of the system and enhances maintainability and reusability. Some domains are reusable and may be included in more than one system.

Identifying Domains and Bridges

Step 1 – Identify the Application Domain: The application domain provides the services specific to this particular system. Almost all systems have a single application domain named for the purpose of the system from the end-user's point of view. For example, the application domain for a medical instrument that measures the level of substances in blood might be called Blood Analysis. The application domain for a microwave oven could be called Microwave Cooking.

Step 2 – Identify Key System Functions: Examine the requirements and list the key system functions, which are high-level functions the system must perform to satisfy the end-user requirements.

Consider a typical microwave oven with two cooking modes:

- cook for a duration at a specified power
- execute a hard-coded defrost program given the weight of the food to be defrosted

The microwave oven has a text display for prompts to input time, weight, and power parameters as well as the time left in the cooking program. From a keypad the user selects the cooking program and the program parameters such as time and weight. The user presses the start key to start the program and the stop key to cancel a program. A hardware device called a magnetron produces the microwaves that are dispersed by a fan. A light illuminates the inside of the oven during cooking and when the door is open. If the door is opened during cooking, the current program is paused until the user closes the door and presses the start button. The key system functions of a microwave oven controller are:

- Initiate cooking program in response to key press by the end-user
- Execute cooking program by switching the magnetron, lights, fan, and turntable on and off at the appropriate intervals
- Disable magnetron and fan when door is opened
- Produce microwaves of specified power
- Control turntable and fan hardware
- Switch light on and off
- Detect presses of keypad hardware
- Detect when door is open or closed
- Update text display with message

<u>Step 3 – Assign Key System Functions to Domains</u>: Group related key system functions together into domains and select a meaningful domain name. Allocate key system functions specific to this particular application to the application domain. Ask yourself three questions about each domain name:

1) Does the name adequately summarize the capabilities of the domain?
2) Would the name make sense to someone familiar with your system but not on the project team?
3) Does the name make sense when you read it aloud?

For example, the microwave oven domains follow:

Domain	***Key System Functions***
Microwave Cooking	• Execute cooking program by switching the magnetron, lights, fan, and turntable on and off at the appropriate intervals • Disable magnetron when door is opened
User Interface	• Initiate cooking program in response to button pressed by end-user
Wave Regulation	• Produce microwaves of specified power
Hardware Control	• Control turntable and fan hardware • Switch lights on and off

Display Management	• Update text display with message
Keypad	• Detect presses of keypad hardware
Sensor Monitoring	• Detect when door is open or closed

Step 4 – Write Domain Descriptions: Developers come from a wide variety of backgrounds, so even the best domain names may be unfamiliar to some members or may imply one thing to one developer and something completely different to another. Write a description of the domain that communicates its capabilities as well as the services it requires. The domain description is the definitive definition of the domain. Well-written descriptions improve model comprehension and reduce confusion.

The domain descriptions for the microwave oven are:

DisplayManagement

Capabilities: Displays a text message via the display hardware.

Required Domains:

None

Keypad

Capabilities: Notifies client when a key is pressed.

Required Domains:

None

Microwave Cooking

Capabilities: Executes cooking programs by switching the microwave, lights, fan, and turntable on and off at the appropriate intervals.

Required Domains:

User Interface – Notifies Microwave Cooking when it is time to start or stop a cooking program. Prompts user to input duration, power, and food size parameters to cooking programs.

Sensor Monitoring – Notifies Microwave Cooking when door is open.

Wave Regulation – Produces microwaves of appropriate power needed by cooking programs.

Hardware Control – Turns fan, turntable, and light devices on and off.

Hardware Control

Capabilities: Starts and stops hardware devices such as lights and fan and turntable motors registered at a memory address.

Required Domains:

None

Sensor Monitoring

Capabilities: Notifies registered clients when the status of a sensor changes.

Required Domains:

None

User Interface

Capabilities: Responds to user key presses by updating the display, getting data input, initiating an action, or stopping an action.

Required Domains:

KeyPad – Detects buttons pressed by user.
Display Management – Display prompts for input.

Wave Regulation

Capabilities: Produces microwaves with a magnetron of a specified power.

Required Domains:

None

Step 5 – Build a Domain Chart: Build a Domain Chart diagram to show the domains identified in previous steps and their dependencies. Domains are represented by UML packages, tabbed rectangles, and bridges by dependencies, dotted arrows from client domain to server domain. Shown in the packages are the name and system-unique prefix of the domain. The prefix is a short name for the domain used for identifying services.

First place the application domain at the top of the Domain Chart. Draw any service domains that the application depends upon under the application domain. Connect the application domain to its dependent service domains with directed lines. Add the rest of the domains in a similar fashion. Refer to Diagram 1 for a domain chart of the microwave oven controller.

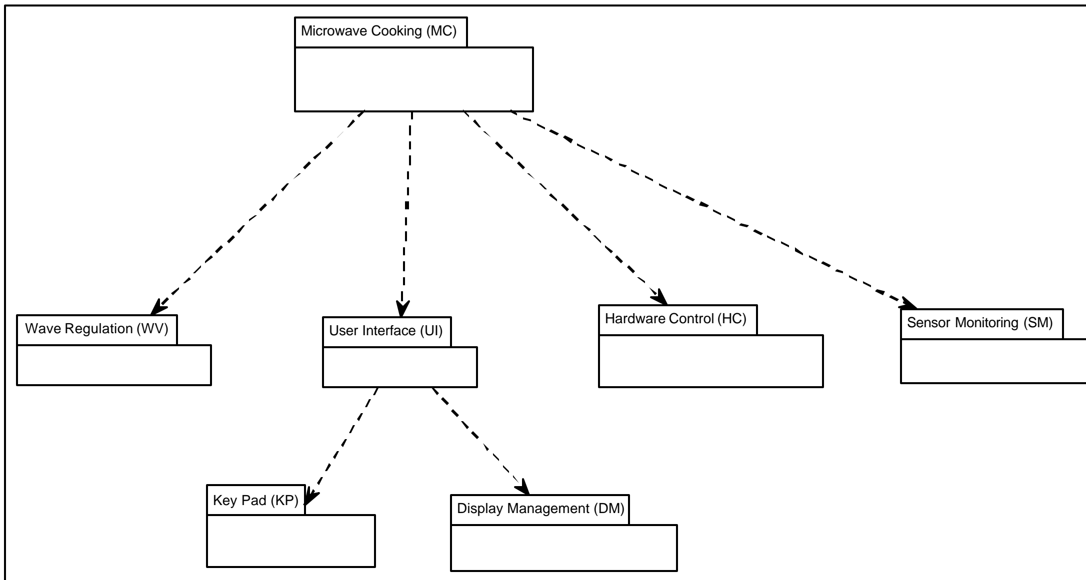

Diagram 10 – Domain Chart for Microwave Controller System

Step 6 – Classify domains as analyzed or realized**:** Some domains will be analyzed and developed with OO modeling techniques and others realized and not developed with OO techniques. Examples of realized domains include legacy code, hand-coded algorithms, and code generated by GUI environments and parser generators. Reasons to realize a domain include:

1) Reuse of legacy code

2) Integration with other parts of the system developed teams not using OO techniques

3) OO modeling is not the appropriate tool to solve the problem. Encapsulate the code in one or more realized domains and define a set of services to access the code.

Realized domains may be noted on the Domain Chart. For example, in the microwave oven, the Hardware Control domain may consist of a set of existing assembly routines that interface with motors and other hardware devices. Services to enable and disable a particular device would call the existing assembly routines. The schedule must include a task to understand any legacy code and incorporate it into the system.

Step 7 – Review Domain Chart: Conduct a review of the domain chart with application experts and team members. Reviewers should check the following:

- Are all key system functions assigned to a domain?
- Are all names and descriptions clearly written?

- Are there any missing bridges?

Benefits of Domain Modeling

Now that you've partitioned your system into domains, you've taken a significant step toward managing complexity. Developers can now think about parts of the system in isolation. It is significantly easier to define a way to execute cooking programs without having to worry about how the microwaves are physically produced.

Domain partitioning illuminates work that can happen in parallel and work that must happen in sequence. When the services provided and required by a domain are specified, development on the domain can begin.

Domain partitioning also helps with project management. Progress can be reported and schedules tracked by domain.

IDENTIFYING CLASSES WITH CLASS BLITZING

Once the domains are identified, identify the core classes in the domain and estimate the total number of classes in the domain by holding a class blitz, a group brainstorming session where team members quickly list the candidate classes for a domain. The core classes are a basis for the Information Model (IM), a diagram that shows the classes, attributes, and relationships owned by a domain. The estimated total number of classes helps determine how to schedule the developer's time for the domain and how many developers to employ on the domain.

Finding the Classes

A class blitz starts when the leader reads the domain description and pertinent sections of the requirements to remind the attendees of the domain's capabilities. Appoint a recorder to write down the class names on a white board. Then open the floor to developers who call out class names, which are immediately written on the board.

During brainstorming, try to avoid lengthy discussions of whether a particular class is valid or not. A productive blitz depends on creativity and free association.

When no one can suggest any more classes for the domain, review each class. Is the name clear? Is the class a valid class for the blitzed domain or is it an attribute or class belonging to another domain. Discard any attributes, invalid classes, or classes supporting requirements in a future release. Combine any similar concepts with different names. If a participant raises an issue, note the issue and move on. Class blitzes should take about 1-2 hours per domain. The author of the IM can sort through the remaining issues and details of the domain later.

Some domains will be easy to blitz. Everyone will be calling out class names and the recorder will be struggling to write everything down. The easiest domains to blitz tend to be the ones where many of the classes map to physical, real-world things. For example, the turntable, magnetron, and light in the Microwave Cooking domain are all physical classes. The more abstract classes are tougher to find. In Microwave Cooking you might not immediately think of the CookingProgram class.

So how do we find less concrete classes? You can start by examining the nouns from pertinent sections of the requirements. Many nouns map to classes. Alternatively,

read models solving similar problems. Another technique is to review the different categories of classes and see if any new classes come to mind. In his book *How to Build Shlaer-Mellor Object Models* [Starr96], Leon Starr defines seven categories of nonphysical classes:

- Discovered: Commonly known concepts from the subject matter of the application. For example, an employee benefits system might have Employee, RetirementAccount, and MutualFund classes.
- Invented: Concepts invented by the developer to enforce the policies of a domain. For example, in the User Interface domain, we need something to display a scrolling text message on the display so we invent the class ScrollingTextDisplay.
- Simulated: Model of a physical class in a simulation. For example, a flight simulator might have an Airplane class.
- Specification: Rules or standards of a set of a type of class. For example, a flight simulator might have an Aircraft Specification class storing the properties and capabilities of the particular model of Aircraft.
- Incident: An event that occurs in the system that has data important to the application. For example, a BloodTest or Scan in a medical instrument.
- Interaction: Stores information about an association between classes. For example, in a scheduling program a Meeting class is created when a Group plans to gather at a Location. The Meeting class might have an attribute recording the time of the meeting.
- Role: Related concepts that act differently depending on the context. For example, in a medical instrument, a Vial may be represented as a SampledVial once it has been tested. The SampledVial may have additional information associated with it such as a relationship to the test results or the quality of the sample.

What if you tried the suggestions above, but you still can't suggest any valid classes for the domain? No class suggestions for a domain often means that either the purpose of the domain is not clear or the domain partitioning is flawed. Go back to the Domain Chart and ask yourself the following questions:

- Should the domain in question be combined with another domain?
- What are the services provided by this domain?
- How can these services be modeled?

Here is a list of some of the classes identified during a class blitz of the Microwave Cooking and User Interface domains:

Microwave Cooking	**User Interface**
Cooking Program	DynamicTextDisplay
Program Segment	ScrollingTextDisplay
Magnetron	ClockDisplay
Lights	ButtonPress
Fan	InputState

Door InputAction
Oven

Estimating Using Class Counts

Once the blitz is complete, the recorder publishes for each domain the list of class names and a class count. Using the class count, the project manager can get a rough idea of the amount of time to complete the analysis of a domain. Estimate the number of hours of development to complete the analysis for a domain by multiplying the number of classes in the domain by the number of hours needed to complete the analysis for a single class. The blitz is a quick estimation of the classes in a domain so we probably missed a few classes that will be discovered during information modeling. To compensate for classes overlooked, we multiply the class count by a FutureDiscoveryFactor. The formula for domain estimation is:

domainLaborHours = FutureDiscoveryFactor * BlitzCount * HoursPerClass
where FutureDiscoveryFactor = 2

The tricky part of this formula is determining the number of hours per class. The number of hours per class depends on the experience of the team and the difficulty of the application subject matter. Apply any metrics collected from a previous project of similar complexity to the estimation formula. If this is your first project, a reasonable starting value for HoursPerClass is 45 hours per class. As the project progresses, track the amount of time spent on each domain so you can adjust this factor to achieve better accuracy.

This brings me to the point of accuracy. The first schedule for a project using class blitz results is going to be a crude estimate; so you shouldn't use these calculations to plan the date for your release celebration quite yet. However, the estimate should at least tell you if the desired release date is in the right ballpark. As the models are completed, more information is available about the domains and the requirements are refined allowing the project manager to construct a more reliable schedule.

Completing the IM

After identifying the key classes in the domains with a class blitz, assign the developers to each domain to complete a preliminary IM. The first IM is preliminary because state model development will reveal new attributes to store the state of a class and relationships navigated in state actions. Modelers sometimes get stuck on the IM because they feel the relationships or classes are not exactly right. If you feel stuck on the IM, look at some scenarios as described in the next section. Scenarios will often help you determine if you have the necessary classes and relationships.

Information Modeling is a whole paper in itself so I'll defer to [Starr96] and [ShlaerMellor92]. Both books provide excellent instruction in information modeling.

Diagram 2 shows the preliminary IM for the Microwave Cooking domain. Please refer to Appendix A for a brief description of the notation.

ESTABLISHING DOMAIN COMMUNICATION PATTERNS WITH SCENARIOS

In a real-time system, the classes are only half the story. The class behavior is of equal importance. Class instances communicate asynchronously by sending events to one another. An event may start a request to perform some function or it may indicate some system state has been achieved or a requested function is complete. A class may have a State Model - a finite state machine- describing its behavior. The State Model shows the states of the class and the processing that occurs when an event causes a transition into the state.

A communication pattern is the sequence of events exchanged by class instances during the execution of a system function. Establishing a domain's communication patterns allows developers to draw skeletons of the State Models. Once the IM is stable and reasonable communication patterns are established, the state and process models fall into place.

Scenario development may be done individually, but an effective scenario development team benefits from a number of different viewpoints and experience levels. A team usually produces more consistent patterns than individuals working alone.

Choosing the Scenarios

Step 1- List the key domain scenarios: Before establishing a communication pattern, you must determine what are the interesting scenarios, covering the core behavioral capabilities of the domain. What requirements must the domain satisfy? For example, the Microwave Cooking domain must be able to execute the time and defrost cooking programs. This is probably the most important scenario in the domain. However, there are lots of other related scenarios such as stopping a running cooking program, pausing a cooking program, restarting a cooking program, and stopping a cooking program in response to a door opening. In addition, the Microwave Cooking domain must be able to start itself up and shut itself down. Sometimes startup and shutdown are trivial, but most of the time they merit their own scenario.

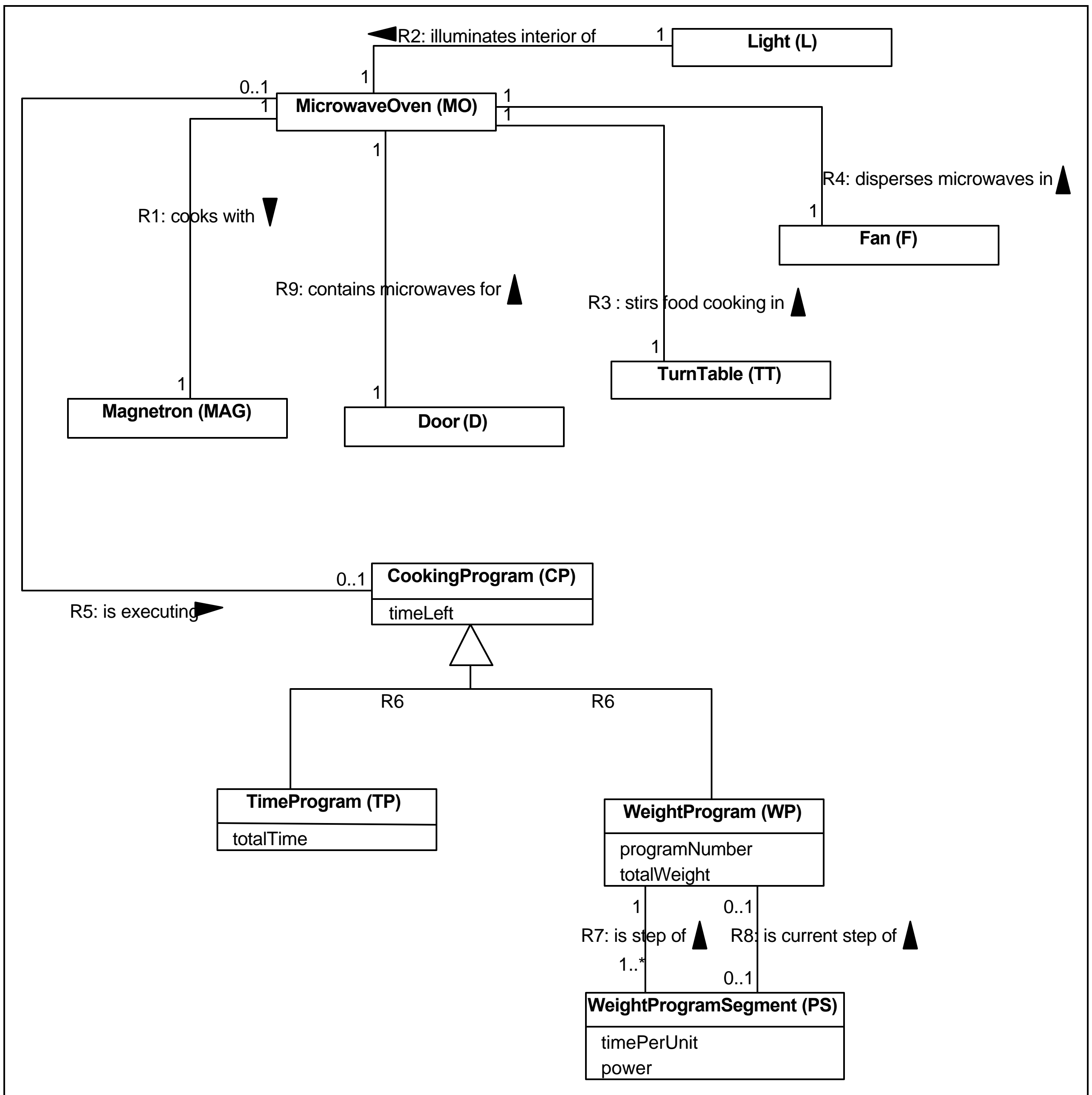

Diagram 11 – Preliminary IM for Microwave Cooking domain

For the first session, concentrate on the normal behavior of the system. Once you understand the normal class interactions, look at the error scenarios.

Step 2 – Identify participating instances: Identify the class instances participating in each scenario. Specify which instances exist before the scenario starts and which ones are created as a result of the scenario. If there is a single instance of each class, name the instance after the class. If there are two instances of the same class, use a name that

reflects the purpose of the instance. Suppose a scenario has two instances of a Motor class. One motor turns the turntable and another runs the fan. Name the instances TurnTableMotor and FanMotor.

Write the names of the participating instances across the top of a whiteboard. Draw a vertical line to the right of the instance name to the bottom of the whiteboard. See Diagram 2 for the preliminary IM for the Microwave Cooking domain. Diagram 3 shows a scenario where MicrowaveOven executes the time cook program. Please refer to Appendix A for a brief description of the IM notation.

Step 3 – Identify participating domains: Class instances in a scenario generally need to invoke the services of other domains. Sometimes the processing for a service generates an event to a class instance in the domain. Add the name of the domain owning the scenario and any required domains to the list of names across the top of the whiteboard.

Step 4 – Identify scenario spark: Something has to happen to initiate the scenario. The scenario spark is the event or domain service invocation that starts the communication pattern.

For an event, draw a horizontal solid arrow from the generator class' vertical line to the receiver class' vertical line. Label the arrow with the name of the event. The event name consists of:

<receiving class' key letters>:<event meaning>

Each class has a short name called the key letters that are shown in parenthesis after the class name on the IM. The event meaning describes the purpose of the event.

For a bridge service invocation, draw a horizontal dashed arrow from the invoking domain's vertical line to this domain's vertical line. Label the line with the name of the service. The service name is similar to an event name:

<prefix of domain providing service>:<service meaning>

There is no need to identify the class from the other domain invoking the service. It should not matter which class invokes a service. If the service may be invoked from many different domains, choose one or draw the dashed arrow coming from offpage. For example, the Time Cook scenario in Diagram 3 starts when the User Interface domain invokes the MC:TimeCook service of the Microwave Cooking domain.

Step 5 – Identify scenario termination condition: To avoid including the whole system in one scenario, specify the event or bridge service invocation completing the scenario. For example, the scenario to execute a time cook is complete when the User Interface domain is notified that the cooking time is complete via a service invocation.

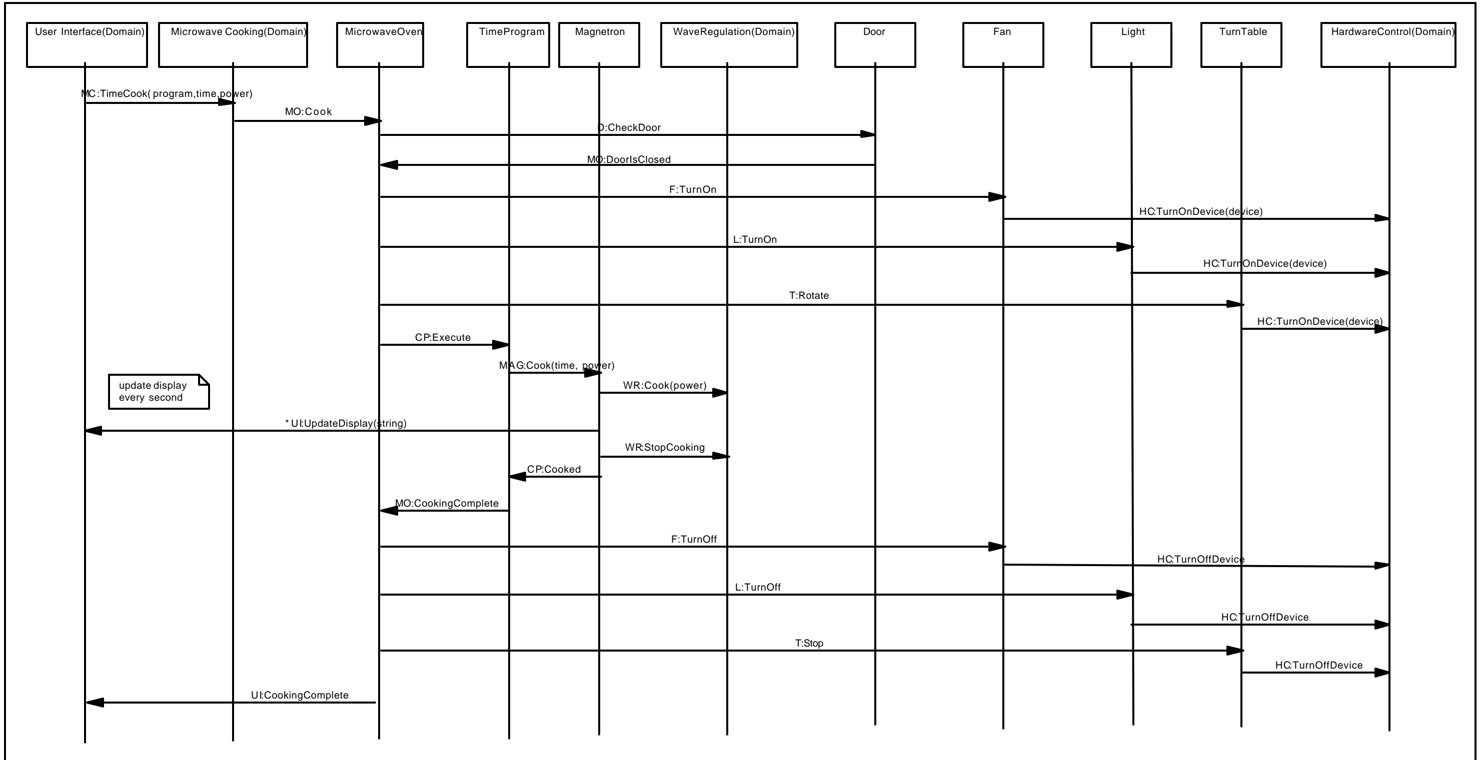

Diagram 12 – Time Cook Scenario

Step 6 – Complete the scenario: Determine what events are generated after the destination receives the scenario spark. What must happen next? Add the generated events and bridge service invocations in time sequence starting from the top of the board and continuing to the bottom of the board. The first generated events and bridge service invocations will be at the top and the last at the bottom.

The TimeCook scenario continues as the Microwave Cooking domain service sets the time and power to the specified TimeCook program. The service relates the MicrowaveOven to the specified TimeCook program using relationship R5 and generates the MO:Cook event to the MicrowaveOven. The MicrowaveOven checks the Door to see if it is closed. For this scenario, the door replies that it is closed. Dealing with an open door is another scenario to be noted for investigation later.

The MicrowaveOven switches on the Fan, Light and TurnTable. The MicrowaveOven executes the CookingProgram related by R5, which happens to be a TimeProgram. The TimeProgram generates an event to the Magnetron to cook for the specified time at the specified power. The Magnetron invokes a bridge to the WaveRegulation domain that starts producing microwaves. The Magnetron sets a timer and after 1 second of cooking, invokes a bridge service of the User Interface domain to display the updated number of seconds left. Note that the timer expiration event does not appear in the scenario. To conserve space, events from timers and events generated and received by the same class instance do not appear in the scenario. When the cooking is complete, the Magnetron stops generating microwaves and the TimeProgram completes. The MicrowaveOven turns off the Fan, Light, and TurnTable. The MicrowaveOven notifies the User Interface domain that the time cook is complete.

While completing the scenarios, you may think of more than one pattern of communication for a scenario. If upon discussion, the benefit of one pattern does not clearly outweigh the benefits of the other, document both. As new scenarios and as error conditions are investigated, one pattern may work better than the other pattern. If there is still no clear winner, pick one and log the other as a discarded pattern for future reference.

Step 7 – Complete state models for class: The set of scenarios for a domain specifies the order of events that each class receives and sends. Since the overall pattern of communication is documented and the events received from other classes are known, State Model development can be divided and assigned to different developers.

Use the scenarios to build the skeleton of the State Model. Place each event on a transition starting with the first received event on top and the last received event on the bottom of the State Model. Add a state between every transition. Diagram 4 shows the skeleton for the MicrowaveOven State Model derived from the scenario in Diagram 3.

Once the skeleton is complete, add descriptions of the processing performed when the state model enters each state.

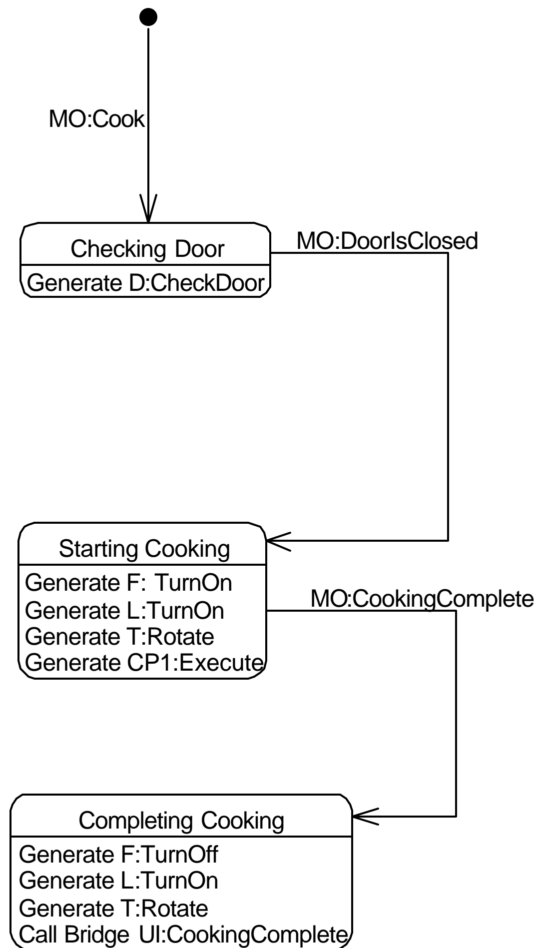

Diagram 13 – State Model Skeleton for Microwave Oven based on Scenario

SUMMARY

Starting a new project is difficult. There are many demands and even more unknowns. Using the three techniques outlined in this paper – domain modeling, class blitzing, and scenario development – you can make your project a success by building a sound foundation. Once the foundation is established, the system will fall nicely into place.

REFERENCES

[Booch99] Booch, Grady, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

[Douglass98] Douglass, Bruce Powel. *Real-Time UML: Developing Efficient Objects for Embedded Systems.* Reading, MA: Addison-Wesley, 1998.

[Fowler97] Fowler, Martin. *UML Distilled: Applying the Standard Object Modeling Language*. Reading, MA: Addison-Wesley, 1997.

[Rumbaugh99] Rumbaugh, James, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

[Selic94] Selic, Brian, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. New York: John Wiley and Sons, Inc., 1994.

[ShlaerMellor92] Shlaer, Sally and Steve Mellor. *Object Lifecycles: Modeling the World in States*. Yourdon Press, Englewood Cliffs, NJ, 1992.

[Starr96] Starr, Leon. *How to Build Shlaer-Mellor Object Models.* Yourdon Press, Upper Saddle River, NJ, 1996.

APPENDIX A: INFORMATION MODEL NOTATION SUMMARY

Each rectangle represents a class. The first line of the rectangle states the class name and the key letters in parenthesis. The key letters are a short name for the class used when naming events destined for that class.

The text lines below the class name are the names of the attributes of the class. Attributes store information about or properties of the class instance.

The lines connecting classes represent relationships. The style of the end of the line indicates the type of relationship. The sample information model shows two types of relationships – bi-directional and generalization relationships.

A bi-directional relationship does not have any shapes at either end of the line. Each bi-directional relationship is labeled with a relationship number and a text description. The arrow on the relationship indicates the direction to read the relationship description. The text at both ends of the relationship represents the number of instances participating in the relationship:

- 1 – one instance
- 0..1 – zero or one instance
- * - zero or more instances
- 1..* - one or more instances

Generalization relationships have a triangle at the parent class end and no shapes at the child class. Generalization relationships are labeled with a relationship number.

ACKNOWLEDGEMENTS

A number of colleagues took the time to review this paper and offer suggestions. Thanks to Peter Fontana and Ted Barbour of Pathfinder Solutions and my husband, David Swift, for the valuable input on method issues. Paul Anderson and Carol Landers, members of the Society for Technical Communication, improved the readability of this paper.