Thinking about Software

An introduction to software development for non-CS majors

Gary Pollice

ABSTRACT

This paper describes an approach to designing a program rather than simply writing code until something works. It describes a basic philosophy of creating software as a product and presents an example component—a state machine—that may be useful to robotics engineers and other embedded systems engineers.

1. Introduction

Creating software is more than writing code until it does what you want it to do. It's a creative, yet disciplined, activity that requires precision and design skills. Too often software is written as a necessary evil, or an afterthought to other parts of a product, especially when that product is a robot or embedded system developed for a course. This view is misleading with respect to the expectations that employers will place on engineers when they enter the workplace. While software engineers may be part of a development team, one cannot assume this to be the case. Developing software design and coding skills as part of robotics engineering (RBE) learning provides the student with skills that enhance their value to prospective employers and will enable them to confidently assume leadership positions on development teams.

This paper is the first in a series of documents designed to help RBE students improve their software development skills. It is divided into the following major sections:

1. Introduction: this section.
2. Software as product: a fundamental approach and way of thinking about software as part of the overall product that one must deliver.
3. Designing software: how to consider the abstractions that are represented in a system and represent them in the software design. This section focuses on abstraction and application programming interfaces (APIs).
4. Design to implementation: developing the program from the design. This section includes unit testing topics.
5. Delivery: making the software robust, documented, and ready for production use.

Throughout the paper we use a single application to illustrate the topics. We develop a configurable finite state machine (FSM) that one might use in a typical embedded or robotics system.

2. Software as product

In order to produce high-quality systems, each component of the system must receive the appropriate amount of design and implementation attention. For RBE, there are three different types of components that must come together correctly in order to deliver a working robot. These are the mechanical, electric, and software components. If any one of the three is not given proper attention, the final system will exhibit poor quality and fail to meet its requirements.

2.1. Systems Engineering

Robotics and embedded systems are good examples of how hardware and software are combined to produce a product. Such products are produced by applying *systems engineering (SE)*.

> Systems Engineering is an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem. (INCOSE)

RBE students already understand multidisciplinary systems. Robot development combines hardware and software that require mechanical, electrical, and software engineering skills. One must develop competency in each area to be a successful robotics engineer. Systems engineering follows a process such as the one shown in the following diagram:

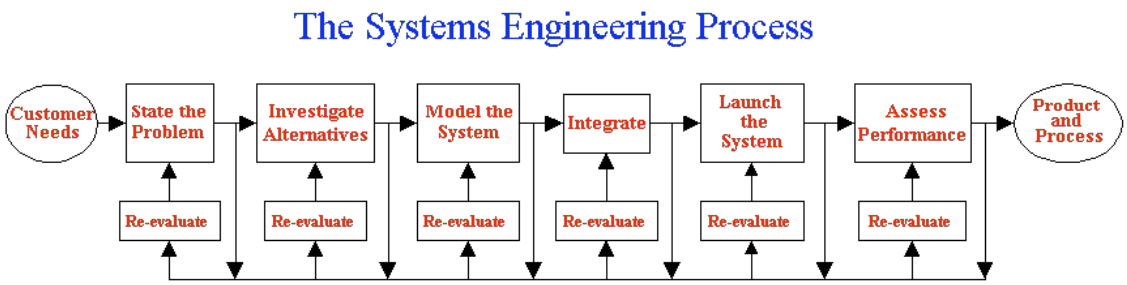

Figure 1. The systems engineering process. (Dean)

One must consider all parts of the system under development in each step of the process. Notice that there is a feedback loop inherent in the process. While the process shown in Figure 1 seems quite logical and implies a sequential progression to the final product, this almost never occurs. In practice the feedback loops are used frequently to move back and forth between steps in a seemingly random sequence.

A key practice in building software systems that can be used in the systems engineering process is called *iterative* development. The fundamental principle of iterative development consists of dividing the project into specific periods, called iterations, during which the development team performs all of the steps in Figure 1 to some degree during the iteration. Depending upon where in the overall project an iteration occurs, more time might be spent on one step than it would be spent in a different part of the project. As the project proceeds, the product is developed incrementally. Some say that the product *evolves* over time. This approach is very effective for software development since software constantly changes throughout the development lifecycle. Software engineering addresses ways of managing such radical change, especially with recent developments in modern iterative techniques like the Unified Process and Agile development techniques.

2.2. Iterations and systems engineering

Iterations are not appropriate for most RBE assignments and labs. Working iteratively on term projects and major qualifying projects is certainly appropriate. For some courses the assignments and labs are similar to iterations for the course

project. This section offers a short description of iterative development. The main topics of the paper focus on the lower-level activities that one must perform to produce working software during any single iteration.

An iteration is a specified time period, often called a *time box.* Project teams use iterations for planning purposes. Before the start of an iteration, the team plans the iteration. This activity identifies the goals for the iteration and breaks down the work into smaller tasks. Individuals or groups of team members work on tasks during the iteration. At the end of the iteration, the completed tasks become part of the iteration's deliverables. The unfinished tasks—even if they are 99% complete—get moved to a subsequent iteration for completion.

Characteristics of iterations include:

- Hard start and stop times. Iterations are not extended in order to allow tasks to be completed.
- Varied durations. Iterations do not have to span the same amount of time. The length of each iteration can vary based upon the amount of work planned for it.
- The amount of work completed in an iteration is called the iteration's *velocity*.[1]
- Each iteration results in a working system. Early iterations contain limited, but working capabilities. These capabilities increase incrementally in each subsequent iteration.

Short iterations have been shown to be most effective for projects of the scale that students work on. Such iterations may last between a couple of days and a couple of weeks. Week long iterations provide a good starting point for student project teams starting with iterative development.

2.2.1. Bit-by-bit: Decomposing work

The iteration plan defines the work planned for a given iteration. Individuals, or small groups of developers, implement the tasks planned for the iteration. Tasks define one "piece" of work that can be implemented and tested. Tasks map to the overall product requirements.

This paper discusses how one might approach implementing a task. This focus provides information about skills one must master before developing expertise at higher levels of project development and management.

After selecting a task the developer will formulate a plan for completing the task. The plan does not have to be written down and often takes form only in the

[1] Velocity is a feature of eXtreme Programming (XP) and is used in the XP Planning Game. The Planning Game is a simple approach to project management and planning for XP projects. Cara Taber and Martin Fowler, Planning and Running an XP Iteration, January 2001, 2 February 2009 <http://martinfowler.com/articles/planningXpIteration.html>.

developer's mind. Good developers formulate such plans naturally and may not even be aware of the fact that they are following a plan. Their experience leads them to follow a plan similar to one that they've used before—knowingly or unknowingly—to solve another programming problem. The plan consists of a series of small steps that lead up to the final solution to the problem. After completing each step the developer has a partial solution that can be tested for correctness before continuing to the next step. The task's implementation progresses in steps, called *increments*, to the final solution for the task's problem.

The decomposition of the work in the manner described may seem like a lot of work, especially when a task is small. As one gains experience, this process becomes natural and imposes little or no overhead on the developer's work.

2.3. The example problem

The rest of this paper describes the implementation of an example problem—a flexible finite state machine that is common in robotics and embedded systems. First we need to understand what a FSM is.

A FSM is a model of a system that consists of various *states* and rules for transitioning from one state to another. As the "machine" runs it moves from state to state according to the transition rules. The FSM lets one reason about the system the FSM models. FSM models take many forms, using many notations. In this paper we use the Unified Modeling Language (UML). (Object Management Group) UML provides a formal notation that make communication about software and systems precise and formal. UML defines several diagrams and associated notation. One diagram type, the State Diagram, lets us accurately describe a FSM.[2]

2.3.1. A simple FSM

A simple example shows some basic techniques using UML to describe an FSM. Consider a light switch. The light switch has two states, ON and OFF. Flipping the switch when the light is off (or in the OFF state) turns the light on and transitions the FSM to the ON state. Conversely, flipping the switch when the light is on turns the light off and transitions to the ON switch. The following UML state diagram describes this system completely.

[2] Robert Martin has written a tutorial on UML State Diagrams and how to use them with FSMs. Robert C. Martin, "UML Tutorial: Finite State Machines," *Object Mentor*, June 1998, http://www.objectmentor.com/resources/articles/umlfsm.pdf (accessed February 5, 2009).

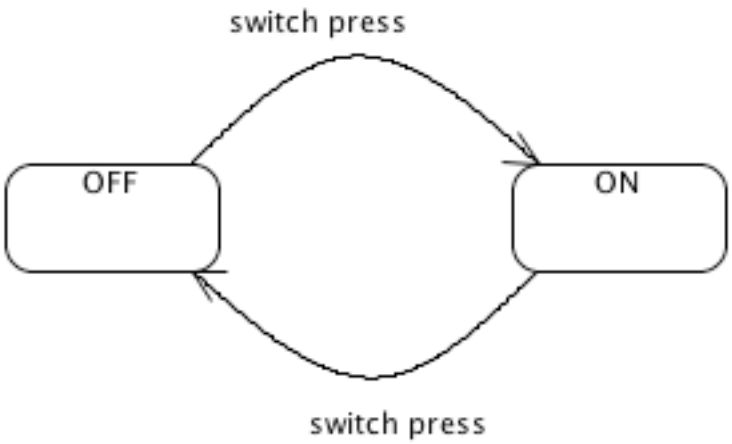

Figure 2. State diagram for a light switch.

Rounded rectangles represent states. A state has a name (e.g., ON and OFF). States can contain other information, but for now we just need the name. The directed connectors between states represent transitions that cause the FSM to move from one state to the next; that is, the *active* or *current* state changes. Events, also called *triggers*, initiate a transition. The transitions in Figure 2 have labels that name the triggers.

UML defines much more that one can add to state diagrams. We will use some of these later in the paper as we need them. Our preference tends to use a minimal number of features, choosing simplicity and readability over complete, but complex diagrams.

2.3.2. A robotics problem

We use a mobile robot example taken from (Jones).[3] Consider a mobile robot similar to the iRobot Roomba®. When the Roomba encounters an object it tries to move away from the area where the collision occurs. This behavior is called an Escape behavior. The Roomba is a cylindrical robot with bumpers that compress when it collides with an obstacle. Bumper compression causes a signal to trigger the Escape behavior.

We might first try describing the Escape behavior in writing. This takes time and is prone to error. We might forget some feature of the behavior and have difficulty uncovering it in a lengthy description. English prose tends to be ambiguous when used to describe complex technical systems. This means that a developer might interpret a specification differently from the author's intended meaning. A UML state diagram can provide the same information in a more concise and precise form. Figure 3 shows how the Escape behavior looks in a state diagram.

[3] pp. 63-69.

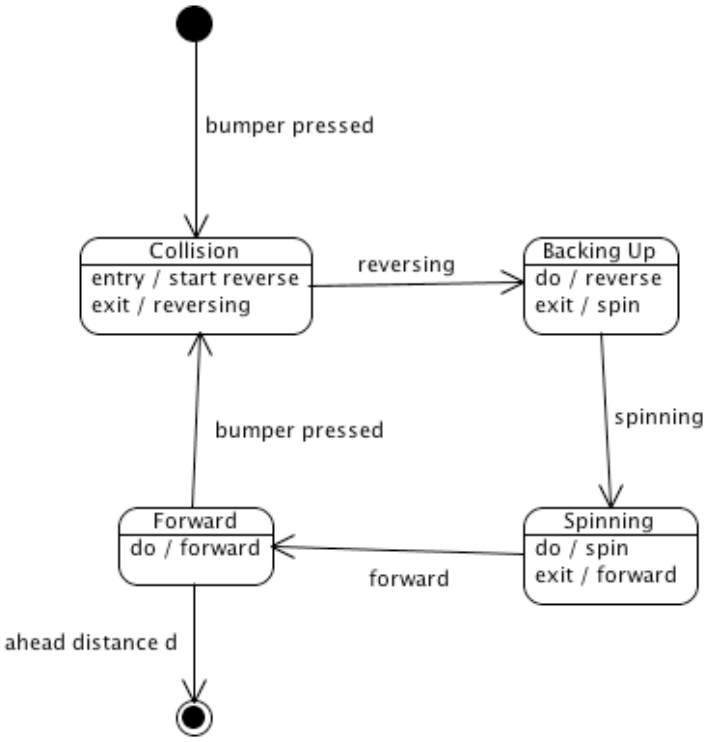

Figure 3. Escape behavior FSM.

The solid circle represents the initial, or start state of the FSM. The diagram describes the following behavior:

1. When a bumper is depressed, due to the robot encountering some obstacle, the Escape behavior begins and the system enters the Collision state.
2. On entry to the Collision state the robot initiates the actions that cause it to move in reverse.
3. Once the robot is in reverse the system transitions to the Backing Up state.
4. While in the Backing Up state the robot continues reversing. The reversing continues until the robot has backed up a specified distance or stops moving (because it encounters an obstacle behind it).
5. When reversing finishes, the system initiates the actions to cause the robot to spin and transitions to the Spinning state.
6. The spinning state keeps the robot spinning until it has turned a specified amount.
7. After the spinning is complete, the robot starts to move forward and the system transitions to the Forward state.
8. In the Forward state the robot moves forward until one of two mutually exclusive conditions occur. If the robot encounters an obstacle, the system transitions to the Collision state. If this happens, the system transitions to the Collision state and the process continues with step 2. If

the robot moves a specified distance without a collision, the Escape behavior finishes. We show this with a transition to the end state (the solid circle in another circle).

The problem before us requires us to develop software that implements the Escape behavior. Good developers dislike writing the same software over and over so they attempt to develop general-purpose reusable solutions. This does not happen all at once. Software evolves from specific to general purpose use. The rest of the paper describes the journey of the evolution from developing software that implements the Escape behavior to a more robust FSM implementation one could use for many different problems.

3. Designing software

At this point we understand the problem—or at least we think we do. Engineering students who have not developed software as a product tend to begin coding the "final" program at this point. One should not avoid writing code, but the code written at this point rarely becomes part of the final deliverable. Experienced software developers multitask at this point, performing the following activities:

- Consider what types of data structures and functions fit the requirements defined by the problem. (Analysis)
- Select candidate structures and functions that best fit the requirements and perceived constraints such as memory constraints and processing time constraints. (Design)
- Try out the selected designs, measuring the performance and correctness. (Prototype implementation)

Developer experience plays a major part in design activities. Good developers have participated in several project, developing software for different systems, and have catalogued—either consciously or otherwise—the techniques and structures that worked. They apply this experience by recalling why certain things succeed and others fail and relate them to the system under development. Computer scientists and software developers call this area of recognizing and applying such knowledge *design patterns*.

3.1. Identify appropriate abstractions

Consider the FSM described in Section 2.3.2. What are some of the words we use to describe the system? Ones that might come to mind are: state, signal (or trigger), sensor, action, motor, and others. Not all of these are things we have to consider for a general purpose FSM, but it's a good idea to thing as broadly as possible at first.

If we can identify physical elements in the system we have a good start to understanding the type of software components and structures we need in order to produce a program that supports those elements. In this example we will certainly need to have a software representation of a state, so we'll begin there.

We will describe our initial thoughts about a state by placing the information on 3x5 index cards. The technique is based upon the Class-Responsibility-Collaboration method of object-oriented design. (Beck and Cunningham) We start with a blank card and place the name of the object (abstraction) we want to describe on the top. We then divide the card into two columns. The column on the left contains the object's responsibilities. The right column contains the name of other objects the current one needs to work with in order to fulfill its responsibilities. At this point, we have a card like the one below.

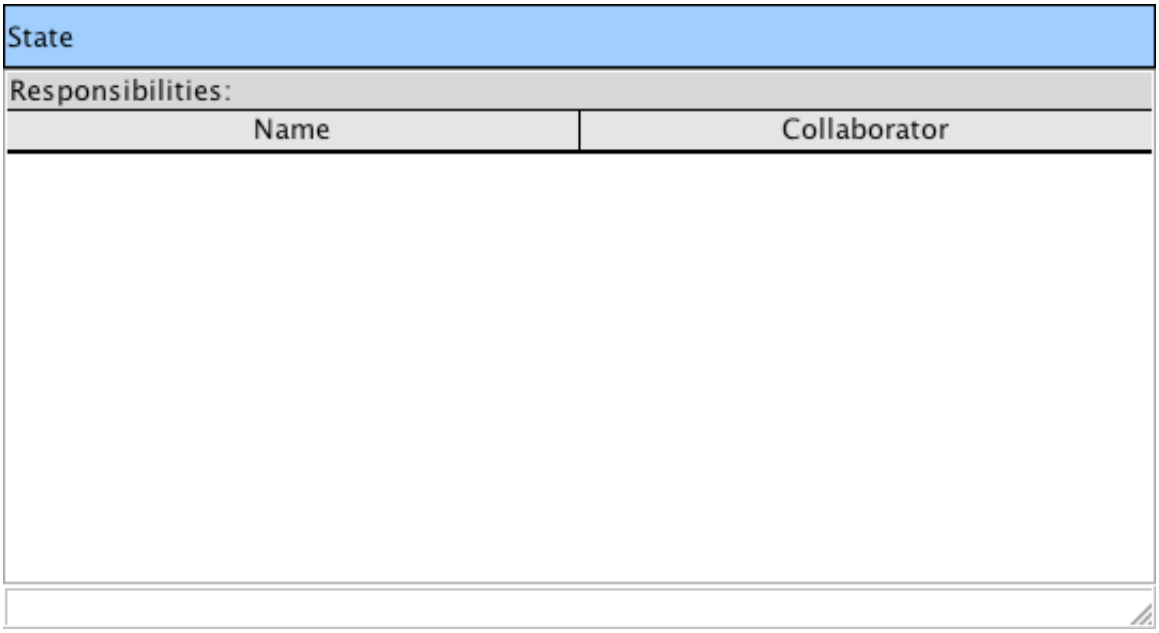

Figure 4. CRC card for the State object.

Next we identify what the State object must do. Inexperienced software developers and designers tend to add to many responsibilities. They think about all of the things that the State might be responsible for in some applications and try to address all of the possibilities in the beginning. *This is absolutely the wrong way to approach the problem!* Thinking too broadly at this point can lead to over-analyzing the problem and a failure to produce any working software. Some call this problem "analysis paralysis." This does not mean that you should build a brittle, simple solution that just gets something to work. It means you should use judgment to decide when you've done enough analysis to be able to start designing a workable, maintainable solution. Two phrases, coined by the Agile development community can help you put things in balance. The first is *"You ain't going to need it"* (YAGNI). The second is *"Do the simplest thing that can possibly work."*

Some programmers think that following this advice gives them the right to do sloppy work or to build incomplete software. In fact, the opposite effect results. When you build just what the requirements demand and do not add features that you think might be useful, you have more time to do a good job on the necessary features. Following the advice helps avoid over-engineering and analysis-paralysis.

Let's continue thinking about the requirements of the State object. We might identify the following responsibilities show in Figure 5.

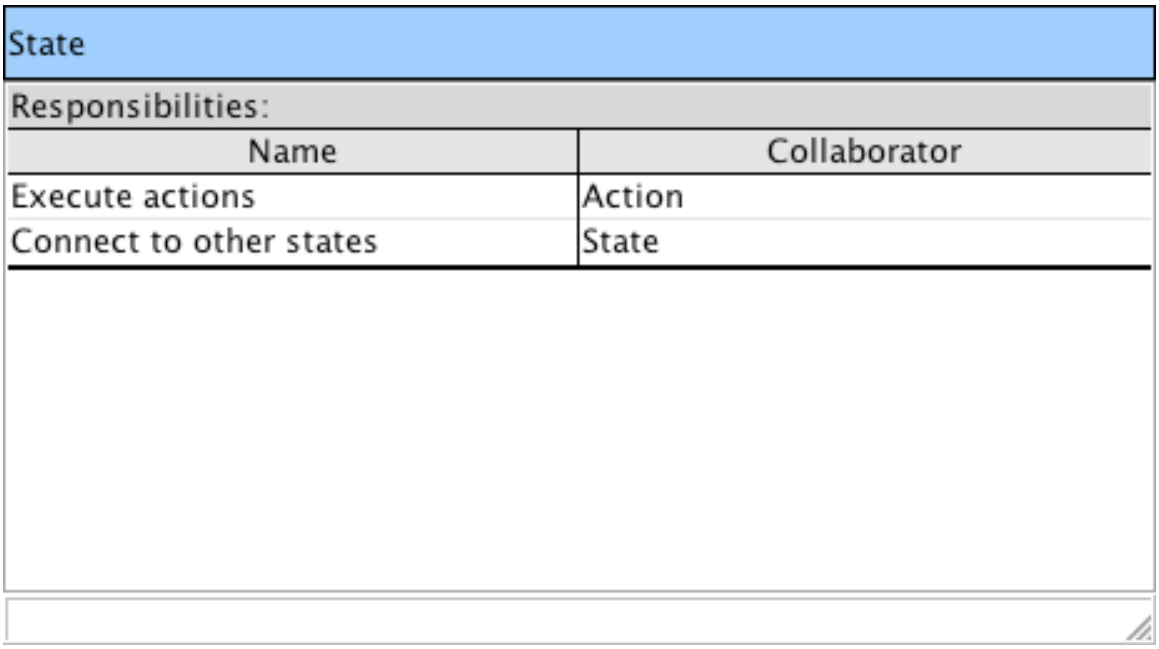

Figure 5. State and its responsibilities.

Notice that the State object collaborates with Actions. Actions are functions that execute when a state is entered, exited, or while state is the *current* state of the FSM. This leads us to create a card for Action objects. We will also want a card for the FSM. The CRC cards might look like Figure 6.

The approach thus far works well for object-oriented development. What if our development language does not have O-O capabilities? The approach still works with some modifications. Objects are simply data structures with associated behavior. We will focus on the behavior and model our system as a set of functions that operate on data structures. The data structures become parameters to the functions. This simulates objects and lets us have multiple instances of the objects.

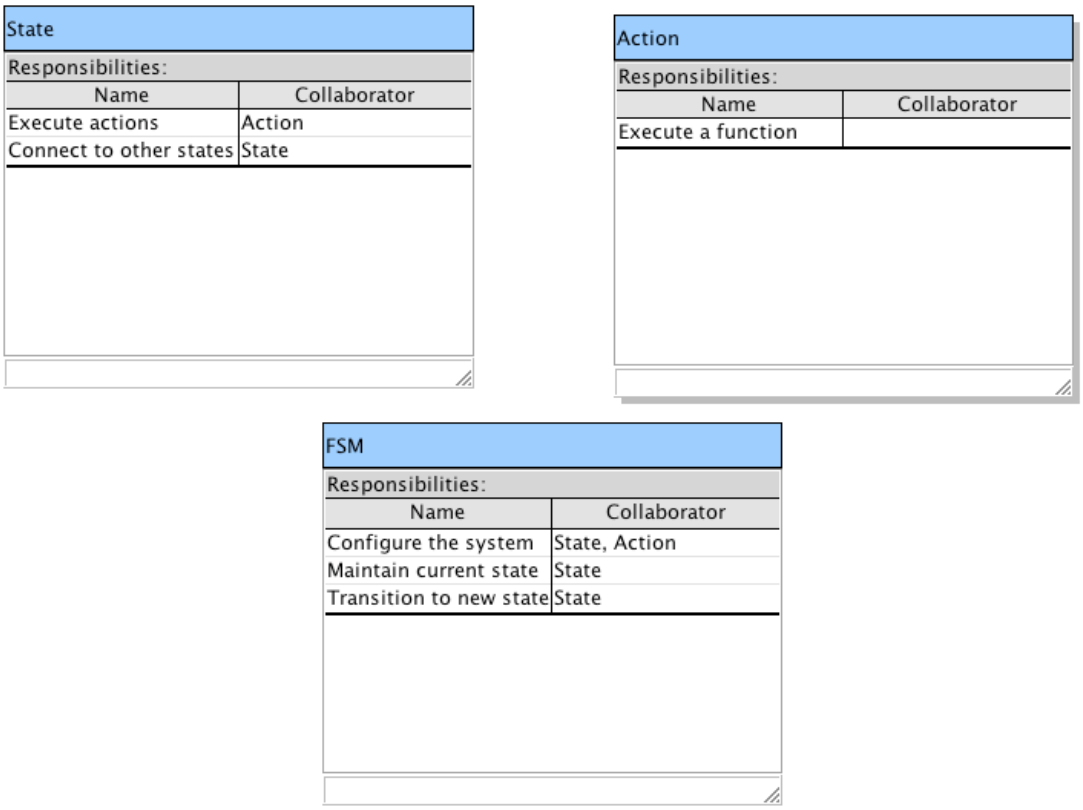

Figure 6. The design evolves.

3.2. Defining the structures and functions

Some software developers prefer to develop programs in a top-down manner while others prefer a bottom-up approach. When working top-down, the developer creates the "main" module first and then those modules at the next level of abstraction, and so on until the system is complete. Bottom-up proceeds in the reverse order—small, independent modules get built first and then the modules that use the first set, and so on. Good developers work in both modes, depending upon the type of problem, the programming language, and other factors. Top-down is a good place to start when we work with procedural languages. In the case of the FSM, some of the conceptual objects shown in Figure 6 may not require implementation.

3.2.1. Designing the FSM module

In the simplest case we will need a single FSM in any application. We begin the design by looking at the responsibilities and identifying possible functions that the FSM module must support. The FSM card in Figure 6 contains three responsibilities. The following table lists the functions that might provide the required capability for an FSM.

Table 1. Functionality of the FSM.

Responsibility	Function	Parameters	Description
Configure the system	initializeFSM	• nStates	Initialize a FSM containing the number of states specified.
	addState	• state	Add the state to the FSM.
	addTransition	• fromState • toState • trigger	Add a transition between states on the trigger.
Maintain current state	<None>		This will be a by-product of normal operation.
Transition to new state	transition	• trigger	Transition to a new state from the current state when the trigger occurs

Given the information in Table 1 we can now create the header file for the FSM module. If you have never approached designing software this way, you may feel a bit nervous. Several things have not been defined, such as the state, trigger, and so on. This is normal. In fact, we do have enough information to begin developing the implementation. The header file—which will not compile yet—can be written as shown in Listing 1.[4]

```
#ifndef FSM_H_
#define FSM_H_

void initializeFSM(int nStates);
void addState(State state);
void addTransition(State fromState, State toState,
      Trigger trigger);

State transition(Trigger trigger);

#endif /* FSM_H_ */
```

Listing 1. Initial FSM.h.

With the appropriate comments explaining each function—starting with the description in Table 1—a reviewer can understand the behavior of the FSM you plan to implement.

[4] For presentation purposes, comments are omitted from the listings.

At this point one can proceed to design the components—State and Trigger—or go into more detail on the FSM component design. We will go one level into the design of the FSM before turning our attention to the other components. The details we want to explore involve the internal representation of the FSM; specifically, how will we represent the actual graph of states and their transitions. We have already encapsulated the implementation with the `initialize`, `addState`, and `addTransition` functions. Notice that choosing to put these functions and responsibilities on the FSM component means that a state does not have to know anything about how to transition to another state or even its connected states. The FSM has assumed this responsibility.

How can we represent the states in the FSM in a way that lets us run the FSM efficiently? We need to find a data structure that allows us to locate any state quickly and determine if a transition is possible from one state to another. A search through data structure or algorithms books uncovers a couple of possibilities. A linked list or other collection of some type can certainly represent a graph where the nodes are the states. The time to find a specific state can be prohibitive, especially in systems where the processing time allowed for a task is a critical resource. We can also represent a graph in a square matrix where the number of rows and columns corresponds to the number of states. As long as we do not have a large number of states and few transitions, the cost extra space required for empty cells in the matrix far outweighs the gains in processor time to access a state. This seems like a good choice. The FSM in Figure 3 would fit into a 5 x 5 matrix. We have the four states that contain behavior and the start state, which can also represent the exit or stop state.

We can number each state in the following way: 0-start, 1-Collision, 2-Backing Up, 3-Forward, and 4-Spinning. Our matrix would look like the following where the cells show the connecting states for transitions.

Table 2. FSM represented as a transition matrix

	0	1	2	3	4
0		x			
1			x		
2					x
3	x	x			
4				x	

Our representation choice now forces us to make other design choices. Clearly, every state must have a number that lets us use it as an index into the matrix. Also, the transition matrix does not indicate which trigger creates the transition. We will need to address these soon (perhaps you would address them now). We will come back to this later in the paper, but now that we have a fairly good idea about how we might structure the FSM, let's look at what we need to create the state objects.

3.2.2. Designing the State

According to our preliminary design in Figure 5 the state has two responsibilities. A little further though shows that the responsibility: *Connect to other states*, has been taken care of by the FSM. This leaves only the *Execute actions* responsibility.

We have also introduced some other requirements on the state in the previous section. We need to have some sort of a state number or identifier that lets us index into the transition matrix. An integer type of some sort will work fine for this.

Each state has three possible types of actions it needs to execute: entry, exit, and in-state actions. We can construct a table like Table 1 or we can jump to creating a header file for the State component. Considering the number of things we need to do, let's just create the header file.

```
#ifndef STATE_H_
#define STATE_H_

typedef short unsigned StateNumber;

typedef void (*ActionPtr) (void);

typedef struct State {
      StateNumber       stateNumber;
      ActionPtr         entryAction;
      ActionPtr         inStateAction;
      ActionPtr         exitAction;
} State, *StatePtr;

StatePtr makeState(StateNumber stateNumber,
                   ActionPtr entryAction,
                   ActionPtr inStateAction,
                   ActionPtr exitAction);

#endif /* STATE H  */
```

Listing 2. State.h.

Notice that we defined types to represent the state number, the actions, the structure for the state, and a pointer to a state. You should be familiar with the use of `typedef`, but you may not be familiar with function pointers like the `ActionPtr`. Any C or C++ reference has sufficient information for these, but you may need to spend some time with the function pointers. The recommended text for RBE3001-2 has such discussions. (Deitel and Deitel) This will be time well spent since many algorithms you might need to implement in C require using such pointers for efficiency.

We now have enough description that we can implement the State module and test it. Listing 2 shows the implementation of the State module. Since we want to have the states persist outside of the `makeState` function, we need to allocate memory for the structure on the heap. The `malloc` function does this. Make sure you know how to use `malloc` and `free` before moving on.

```
#include "State.h";
#include <stdlib.h>

StatePtr makeState(StateNumber stateNumber, ActionPtr entryAction,
            ActionPtr inStateAction, ActionPtr exitAction)
{
      StatePtr s = (StatePtr)(malloc(sizeof (State)));
      s->stateNumber = stateNumber;
      s->entryAction = entryAction;
      s->inStateAction = inStateAction;
      s->exitAction = exitAction;
      return s;
}
```

Listing 2. Implementation of the State object (State.c).

Writing tests for *all* code you write is a good idea. Test frameworks like CUnit and CPPUnit can help you do this painlessly. For this paper we will write a single simple test to make sure that we can create a State object and execute one of its actions. If the test runs, it gives us confidence that we have written code that works in some instance and we can continue with the development. We see the test file in **Error! Reference source not found.**.

We can run this test by itself any time we want to. It does not require us to have the complete FSM implemented. In fact, it only relies upon a very specific set of capabilities that we have already implemented for the State module. This type of test is called a *unit test*. We try to build a good set of unit tests for each of our modules. We want the tests to run quickly so that every time we make a change to any of our code we can easily run all of the tests to see if we have broken anything. If we have broken something, we must fix it before we continue.

```
#include <stdio.h>
#include "State.h"

void helloAction(void)
{
	printf("Hello");
}

void doStateTest()
{
	StatePtr sp = makeState(0, (ActionPtr)0,
				helloAction, (ActionPtr)0);
	(sp->inStateAction)();
}

int main()
{
	doStateTest();
	return 0;
}
```

Listing 3. A test for the State module.

If we run our test and see "Hello" on the console, we're good to go. We need to ask whether we have actually defined the responsibilities of the State module. We have not. Even though we know that we *can* execute one of the actions, we have not provided the behavior in the state module. We need to add three functions to our State.h file and implement them. Add the following code to State.h.

```
void executeEntryAction(StatePtr state);
void executeInStateAction(StatePtr state);
void executeExitAction(StatePtr state);
```

Listing 4. Action functions for the State module.

One might ask at this point why we need to implement the functions since we can execute the state's functions directly by getting the field from the State

structure. First, this encapsulates action execution, making the State responsible for invoking its own action functions. As a by-product of this encapsulation we can change the way the state calls the action functions without worrying about breaking some other part of our system. For example, if we decide that the current state must be place in some specific location after invoking the exit action function, we can make that change in `executeExitAction` and no other code will be affected.

A second similar, but equally important, reason for the encapsulation is that it gives us the ability to not worry about how the functions are actually selected and executed. If you think about the way any FSM works you can imagine the following sequence of steps repeated as long as the FSM runs:

1. Enter a state and execute the entry action.
2. Execute the inStateAction.
3. Wait for a trigger.
4. Execute the exit action.
5. Transition to a new state.

We can implement this code in the FSM module by creating a function called `runFSM`. We will implement the function incrementally. The initial increment contains the code shown in **Error! Reference source not found.**. This listing shows the doxygen comments for the `runFSM` method. We simply transfer the written specification from above to doxygen format. The functions `waitForTrigger` and `transition` have not been implemented, except to create a stub that does nothing. This allows us to compile the code we already have. We also had to revise our original declaration of `waitForTrigger` so that it returns a StatePtr rather than a State. Such changes are expected and should be part of any development process. The important thing to remember is to test your code every time you make a change to ensure that you have not broken anything and that you previously implemented.

```
#include "FSM.h"
#include "State.h"

#define TRUE 1
#define FALSE 0

volatile StatePtr             currentState;
volatile int                  machineIsRunning = FALSE;

/** \fn void runFSM(void)
 * \brief Start the FSM running and continue until a final state is
 *    reached, which turns the machine off.
 *
 * The FSM runs by repeatedly going through the following sequence of
 * steps:
 * <ol>
 * <li> Execute the entry action. </li>
 * <li> Execute the in-state action. </li>
 * <li> Wait for a trigger. </li>
 * <li> Execute the exit action. </li>
 * <li> Make the transition. </li>
 * </ol>
 *
 * The <tt>currentState</tt> variable is used to identify the current
 * state the machine is in. While the machine is not in the final state,
 * the value of <tt>machineIsRunning</tt> will be set to TRUE. Once the
 * machine transitions to the final state, the value changes to FALSE
 * and the function exits.
 */
void runFSM(void)
{
      machineIsRunning = TRUE;
      while (machineIsRunning) {
            executeEntryAction(currentState);
            executeInStateAction(currentState);
            Trigger trigger = waitForTrigger();       // TODO
            executeExitAction(currentState);
            currentState = transition(trigger);
      }
}
```

Listing 5. Initial implementation of the FSM.

The `runFSM` function has short, straight forward code—an indication of well-designed code. You can provide the function signature with the description from the

doxygen comments as a section in a design document. That along with a brief diagram and description of the module structure as shown in section 3.1 gives an interested reviewer the information needed to understand your design. It also gives a grader confidence that you understand what your code does and that you have, in fact, designed it.

Works Cited

Beck, Kent and Ward Cunningham. "A Laboratory for Teaching Object-Oriented Thinking." OOPSLA'89 Conference Proceedings. Association of Computing Machinery, 1989.

Dean, Frank F. What is Systems Engineering. 15 January 2009. 2 February 2009 <http://gd.tuwien.ac.at/systeng/bahill/whatis/whatis.html>.

Deitel, P.J. and H.M. Deitel. C++ How to Program, 6 ed. Upper Saddle River: Pearson Education, Inc., 2008.

INCOSE. What is System Engineering. 2004 June 2004. International Council on Systems Engineering. 2 February 2009 <http://www.incose.org/practice/whatissystemseng.aspx>.

Jones, Joseph L. Robot Programming A Practical Guide to Behavior-Based Robotics. New York: McGraw-Hill, 2004.

Martin, Robert C. "UML Tutorial: Finite State Machines." June 1998. Object Mentor. 5 February 2009 <http://www.objectmentor.com/resources/articles/umlfsm.pdf>.

Object Management Group. Object Management Group - UML. 18 January 2009. 5 February 2009 <http://www.uml.org/>.

Taber, Cara and Martin Fowler. Planning and Running an XP Iteration. January 2001. 2 February 2009 <http://martinfowler.com/articles/planningXpIteration.html>.