Department of Computer Science
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

COS121 Lecture Notes: L19

UML Activity Diagrams

12 September 2014

Contents

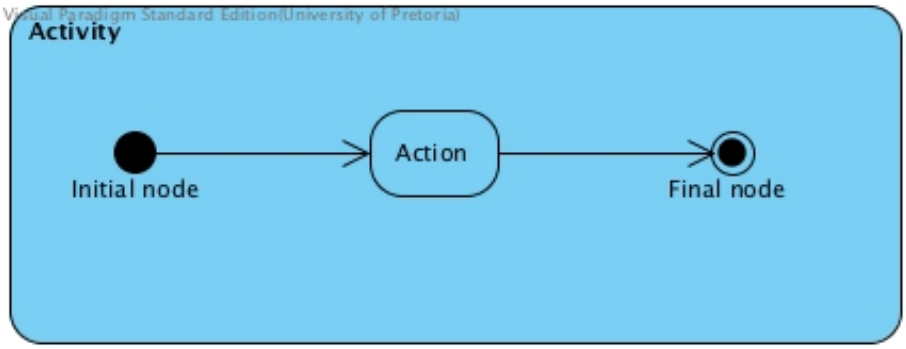

Figure 1: Start and end nodes

L19.1 Introduction

The first structured method for documenting process flow, the "flow process chart", was introduced by [3]. Activity diagrams combine techniques from flow charts, event diagrams [5] and Petri nets [6]. Activity diagrams describe the workflow of a system. The diagrams describe activities by showing the sequence of activities performed. Activity diagrams are useful for analysing a use case by describing what actions need to take place and when they should occur. Usually activity diagrams are used to describe a complicated algorithm or to model the flow in applications with parallel processes. The distinction between state and activity diagrams is only in what they model. Where state diagrams are used to model state-dependent behaviour and conditions for transitions between states, activity diagrams are used to model the flow of actions and the order in which the actions take place.

L19.2 Notational Elements

The notational elements used in state diagrams and activity diagrams are the same except for a few subtle differences. The main difference between UML state diagrams and UML activity diagrams is in their intent.

The basic elements of UML activity diagrams can be classified as activity nodes and edges. Activity nodes can either be action, object or control nodes. An activity is shown as a round-cornered rectangle which encloses all the action and control nodes which make up the activity. Action nodes represent a single step within an activity and are also denoted by a round-cornered rectangle. Control nodes model different flows that are controlled by conditions called guards.

L19.2.1 Initial and final nodes

The starting point of the flow that is shown in an activity diagram is indicated with a filled circle. An activity diagram must have exactly one initial node. An exit point in an activity diagram is called an final node. There are two kinds of final nodes, referred to as as activity final node and flow final node. The flow final node denotes the end of all flow controls within the activity. The activity final node denotes the end of a single

A state van have activities An activity does not have states

Figure 2: The notational difference between states and activities

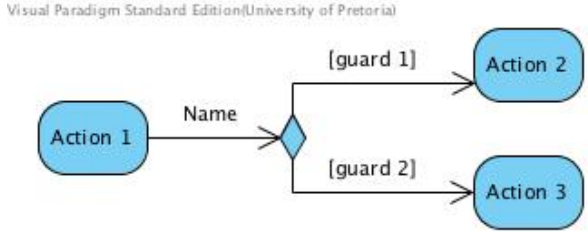

Figure 3: Activity edge

flow control. An activity final node is shown with a filled circle with an outline. Refer to Figure 1. An activity diagram may have multiple final nodes. It is also allowable that it has no final node. If an activity diagram has no final node, it models an infinite activity such as an infinite loop.

L19.2.2 Action nodes

An action node in an activity diagram is a rounded square containing a descriptive name for the action. It looks exactly like a state node in a state diagram, as can be seen by comparing `ReceivingState` in Figure 2 with `Action 1` in Figure 3. The difference between a state node and an activity node and more specifically the action nodes, is that a state node in a state diagram may contain actions that are performed whereas activity nodes **are** actions and do not contain state.

L19.2.3 Activity edge

An activity edge is an arrow indicating the flow between two activities, as shown in Figure 3. The arrow points in the direction of the activity that has to execute next. Optionally activity edges may be named using text. Usually there is no need to name activity edges. When decision nodes (Section **??**) are used, each activity edge exiting from the decision node should have a guard condition shown with text in square brackets. A guard condition indicates that the flow to the next activity may only be executed if the specified condition is true. Activity edges are similar to transition edges found in state diagrams. An important difference is that a transition edge in a state diagram my be decorated with an activity while it is not permissible to indicate an activity on an activity edge in an activity diagram.

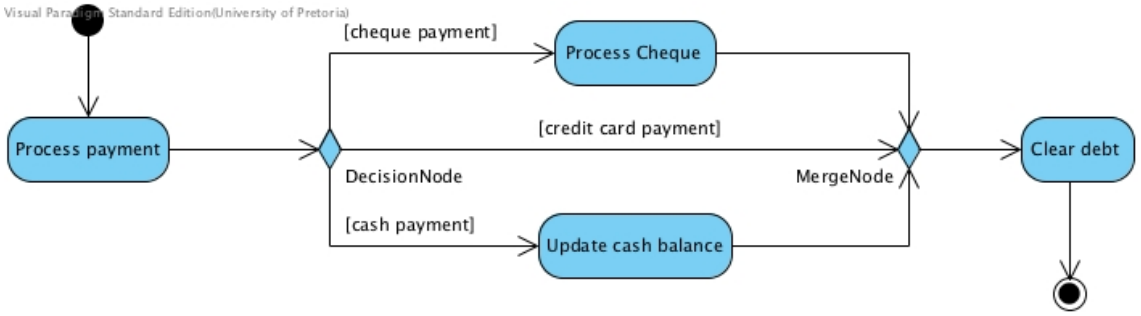

Figure 4: Model of a payment showing the use of decision and merge nodes

L19.2.4 Alternate flows

Alternate flows are shown by using decision and merge nodes. Both these are shown as a diamond shape. If different activity edges leave such shape it is a decision node. If different activity edges meet at the shape, it is a merge node. Figure 4 shows a diagram with alternate flows. The guards on the action edges leaving the decision node on the left side of the figure specifies under what conditions each of these flows should execute. In this example there are three alternate flows depending on the kind of payment. Both cash and cheque payments requires some extra activity while the flow immediately proceed to the clear debt activity if a credit card payment is made. In the case of cheque payments the cheque is processed while cash payments require an update of the cash balance before proceeding to the clear debt action. It is important to note that only one of the alternate flows may execute at any time. Therefore, the guards should be mutually exclusive. In programming alternate flows represent conditional actions like in `if` and `while` statements.

L19.2.5 Parallel flows

Parallel flows are shown by using fork and join nodes. Both these are shown as a heavy vertical or horizontal line. If different activity edges leave such line it is a fork node. If different activity edges meet at the line, it is a join node. Figure 5 shows a diagram with parallel flows. When a fork is reached in the flow, all the activities to which the activity edges that leave the fork point, are executed at the same time using independent execution threads. This example models a shopping experience by a married couple. They both enter the shop and then proceed each on a separate mission. While the one person (probably the wife) picks up fruit, then meat and then milk, the other person move directly to the news stand to pick up a news paper. After both completed their respective actions, they join and proceed to the till. It is important to note that all the flows leaving a fork are executed in parallel. In programming new threads are spawned for each parallel flow.

L19.2.6 Composite activities

A composite activity is an activity that can itself be described in terms of an activity diagram. Sometimes composite activities are shown by drawing the diagram of its activi-

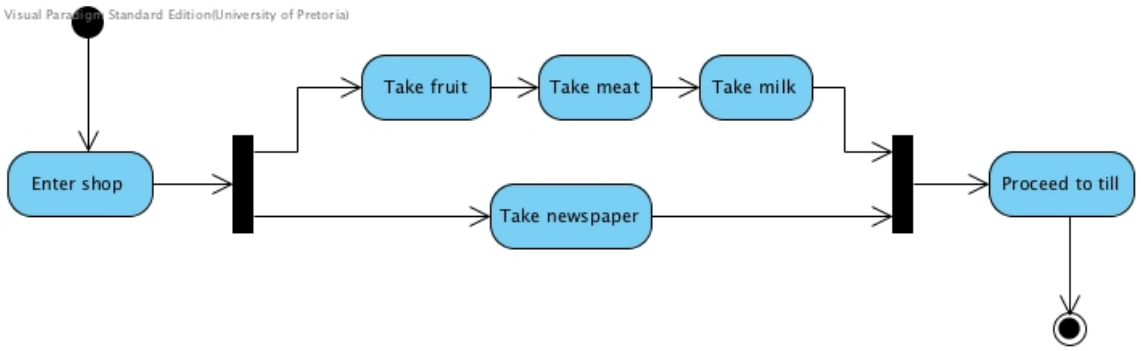

Figure 5: Fork and join nodes

ties inside the action node as shown in the right hand diagram in Figure 6. A composite of which the sub-activities are not shown can be drawn using a rake symbol inside it as shown in the left hand diagram in Figure 6. Such sub-activity is called a *called action*. A called action is like a subroutine call in a program. Called actions are often used to eliminate the need to draw the same sub-activity diagram in more than one place.

L19.2.7 Swimlanes

Swimlanes are used to convey which class is responsible for a given activity. A swimlane is a vertical or horizontal zone in which the activities which are the responsibility of a certain class are grouped. Swimlanes are indicated using solid vertical or horizontal lines to indicate a border between two swimlanes and labeling the zones. The diagram in Figure 7 was adapted from [1]. It illustrates the use of swimlanes. In this diagram the consultant is responsible for filling in the expense form and correcting it if needed. The accountant is responsible for validating such form and taking the appropriate actions while the payroll service is responsible for paying the employer of the consultant.

L19.3 Examples

L19.3.1 Finding and drinking a beverage

The activity diagram in Figure 8 comes from the UML 1.0 documentation and was replicated from [2]. This diagram illustrates the interpretation of an activity diagram on a conceptual level.

There parallel activities `Put Coffee in Filter` ,`Add Water to Reservoir` and `Get Cups` may be interpreted to mean that these activities may be executed in any order or if possible at the same time by interleaving them. The synchronisation point indicated by the join node before `Turn On Machine` should be interpreted that the machine may only be turned on after both the putting the filter with coffee and putting water in the reservoir was completed.

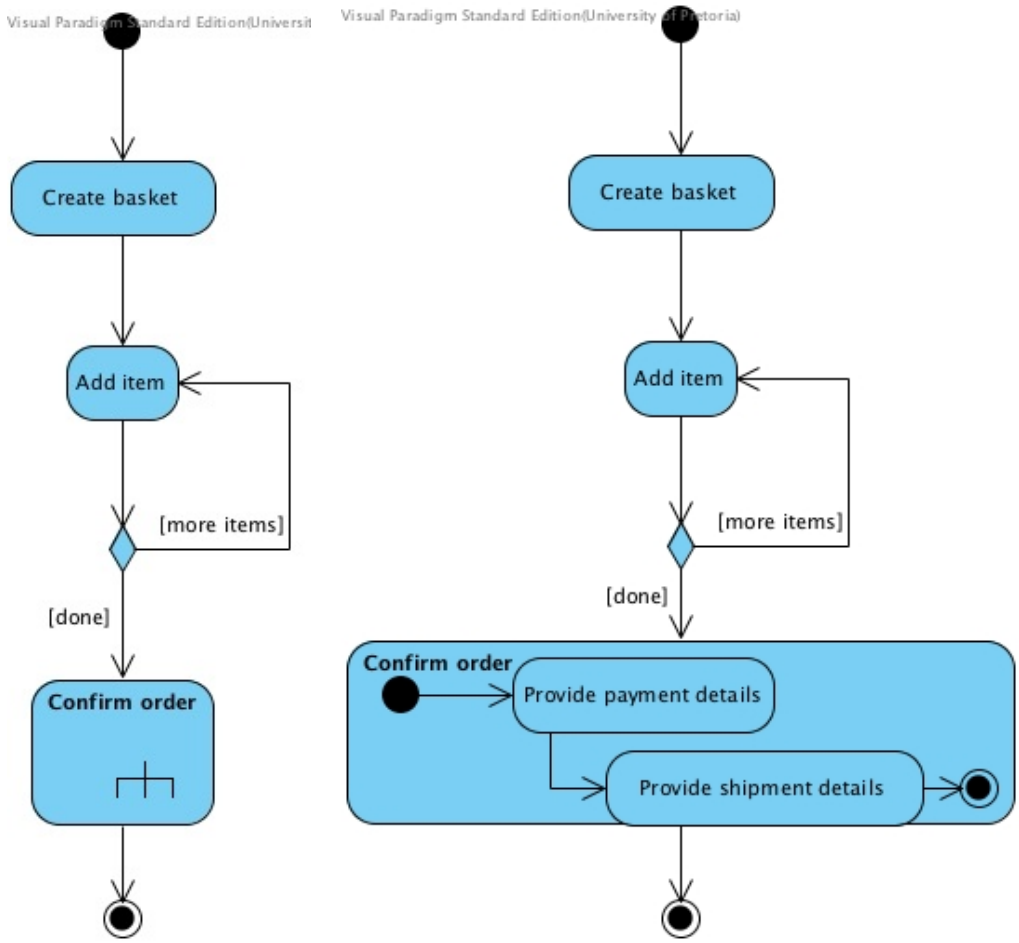

Figure 6: Activities with sub-activities

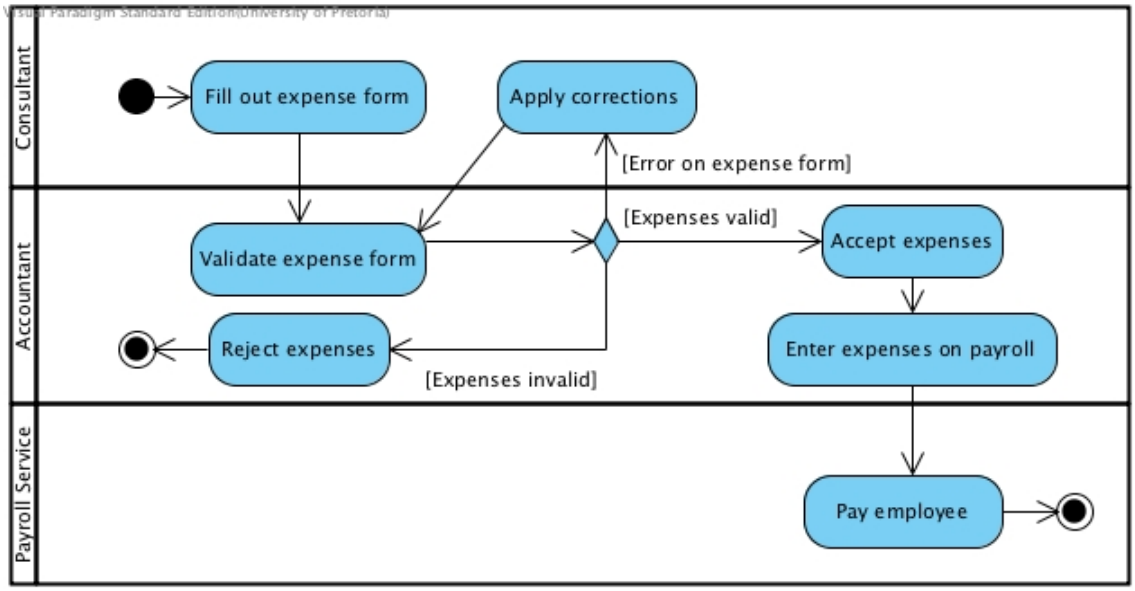

Figure 7: Processing an expense

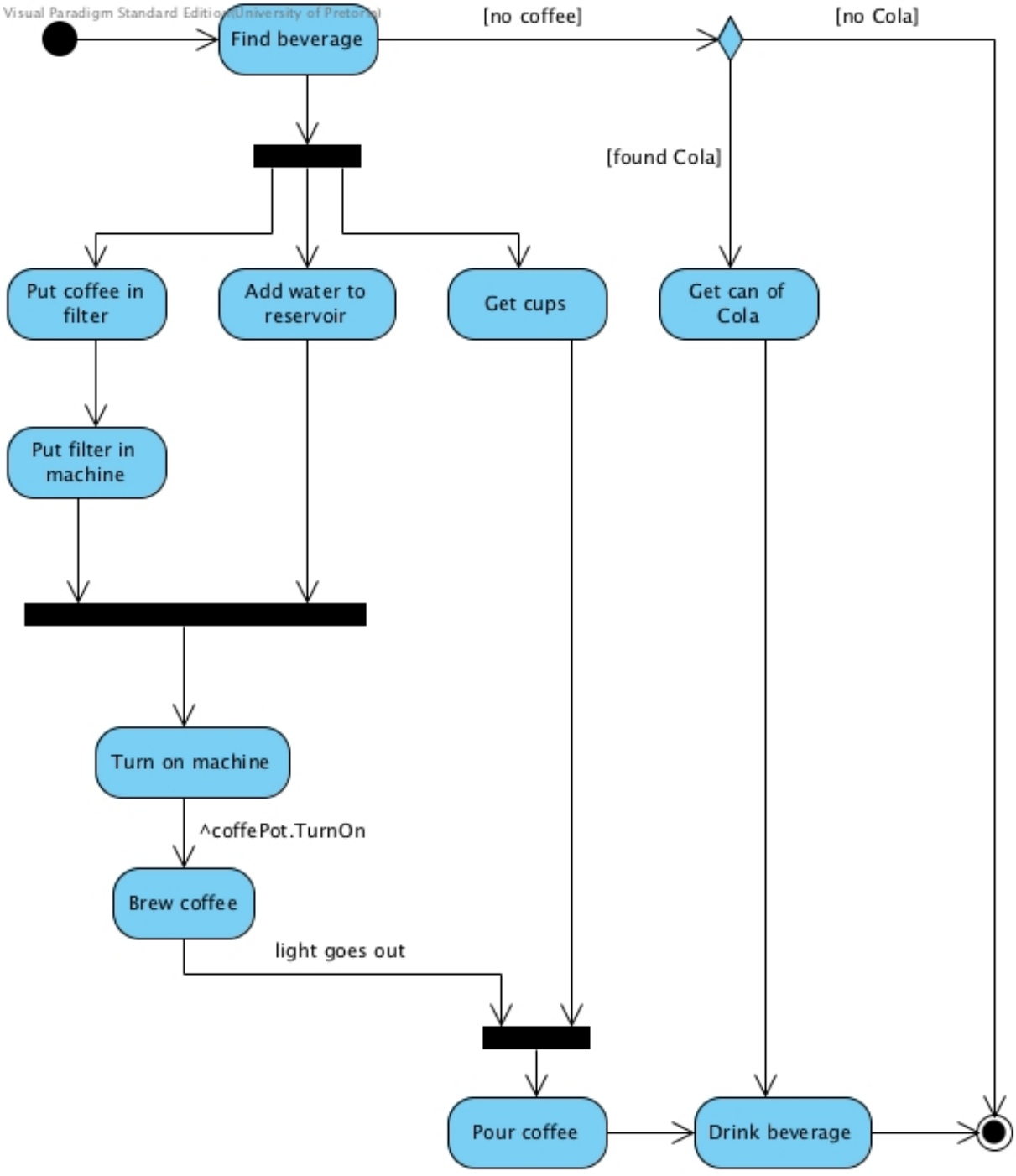

Figure 8: Finding and drinking a beverage

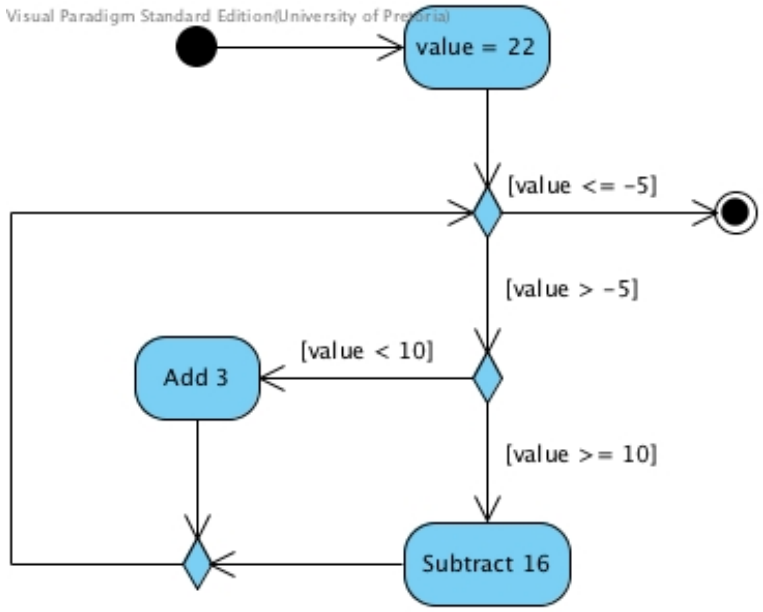

Figure 9: An activity diagram containing a loop

L19.3.2 Implementing code for an activity diagram

Activity diagrams can be translated into code. Each activity node is either a program statement or a function call. If it is a function call it can be indicated using the rake symbol. Decision nodes represent conditional statements like `if` or `switch`. If there is some transition from a later activity node back to a decision node, it represents a loop structure which can be implemented using a `while`-loop, a `for`-loop, or recursion.

The activity diagram in Figure 9 contains a loop because the topmost decision node may be reached a number of times. The following code fragment is an implementation of the diagram in Figure 9 using a `while`-loop

```
int value = 22;
while (value > -5)
{
  if (value < 10)
  {
      value += 3;
  }
  else
  {
      value -= 16;
  }
}
```

L19.3.3 Activities based on age of the participant

From an implementation-perspective the activities shown in an activity diagram represent methods in classes. To illustrate this we have created code to 'implement' the activities

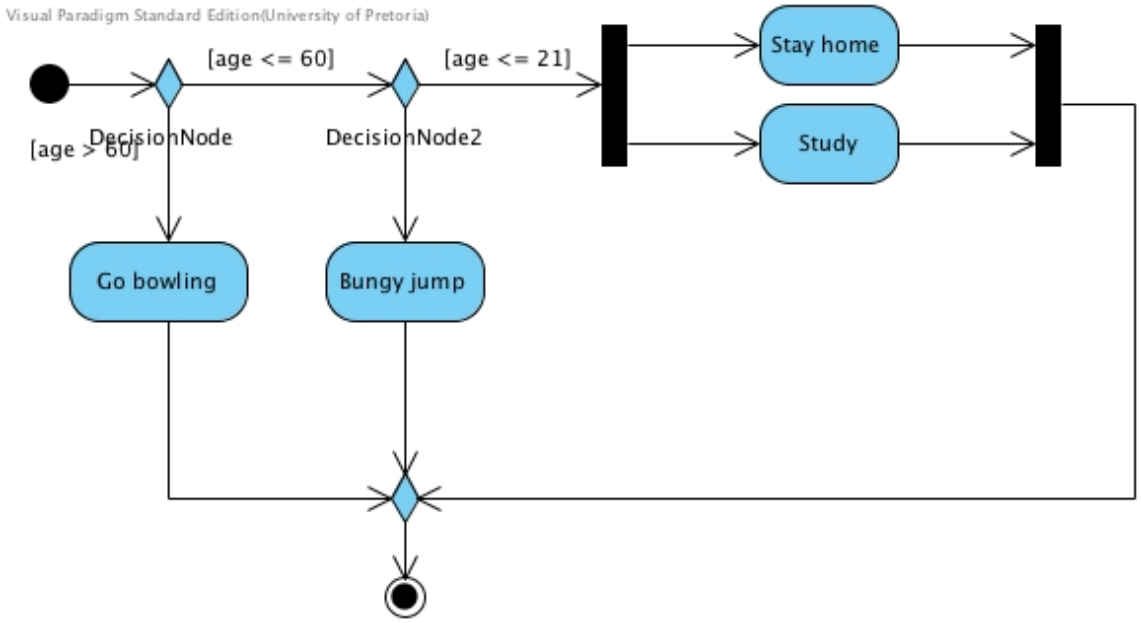

Figure 10: Activities for people of different age groups

in the activity diagram in Figure 10 and came up with the following code:

```
#include <iostream>
#include <boost/thread.hpp>

using namespace std;
using namespace boost ;

void goBowling() ;
void bunjyJump();
void stayHome();
void study() ;

int main() {
        int age ;
        thread stayHomeThread = thread(stayHome);
        thread studyThread = thread(study) ;
        cout << "Please enter your age: ";
        cin >> age;
        if (age > 60)
        {
                goBowling();
        }
        else if (age > 21)
        {
                bunjyJump();
        }
        else
```

```
        {
                stayHomeThread.join ();
                studyThread.join();
                cout << "Actions while threads are running" << endl ;
                stayHomeThread.detach() ;
                studyThread.detach() ;
        }
}

void goBowling()
{
        cout << "Enjoy the day playing bowls" << endl ;
}

void bunjyJump()
{
        cout << "Drive to the precipice , " ;
        cout << "attach the bunjy cord and jump !!!!" << endl;
}

void stayHome()
{
        cout <<  "Stay Home" << endl;
}

void study()
{
        cout << "Study" << endl ;
}
```

An article [4] that was written by the author of the `Boost.Threads` library will get you started on installing and using this library and writing multi-threaded applications using the C++ programming language[1].

References

[1] SW Ambler. Uml 2 activity diagramming guidelines. `http://www.agilemodeling.com/style/activityDiagram.htm`, 2009. [Online; accessed 2011-09-09].

[2] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, Mass, 1997.

[1] In this module you must be able to understand code using threads. However, it will not be expected that you have to be able to write such code yourself

[3] F. B. Gilbreth and L. M. Gilbreth. Process charts: first steps in finding the one best way to do work. Presented at the Annual Meeting of The American Society of Mechanical Engineers, New York, USA, 1921.

[4] William E. Kempf. The boost.threads library. `http://drdobbs.com/cpp/184401518`, 2002. [Online: Accessed 1 Sept 2011].

[5] James Martin and James J. Odell. *Object-Oriented Analysis and Design.* Prentice Hall Inc, Englewood Cliffs, NJ 07632, 1992.

[6] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall Inc, Englewood Cliffs, NJ 07632, 1981.