Module 4:

Programming Tools

BCSE305L

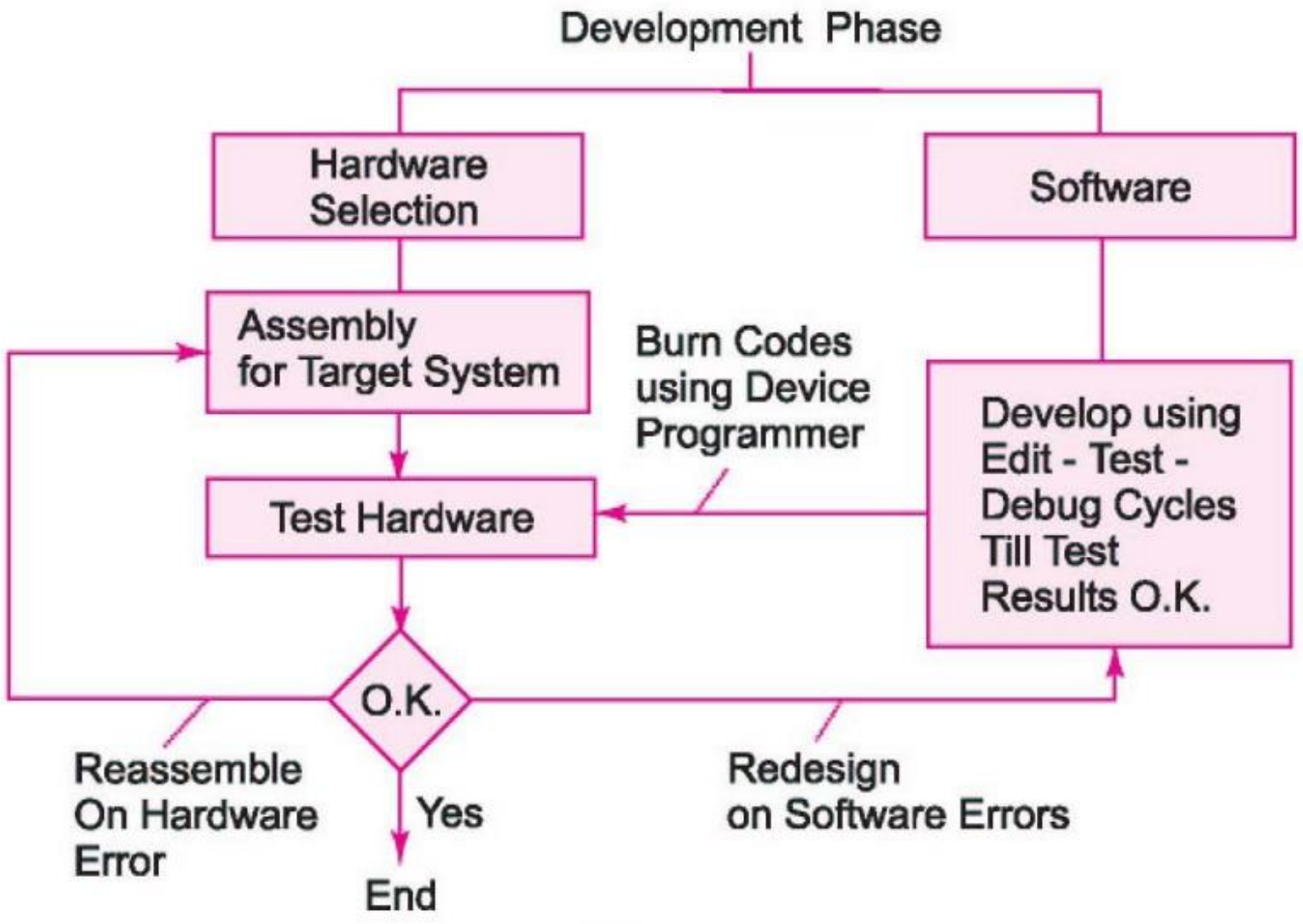
Dr. P. Vijayakumar

Outline

- Evolution of embedded programming tools,
- Modelling programs,
- Code optimization,
- Logic analyzers,
- Programming environment.

Building Process for **Embedded Systems**

Building Process for Embedded Systems



Building Process for Embedded Systems

Approaches During
Edit – Test – Debug Cycle

- Using a Target System
- Using an Emulator for the Target System
- Using Target Processor & ICE
- Using a Simulator for Hardware
- Using IDE / Prototyping Tool

Building Process for Embedded Systems

- Embedded systems programming is not substantially different from the programming you've done before.
- The only thing that has really changed is that you need to have an **understanding of the target hardware platform**.
- Furthermore, **each target hardware platform** is **unique**.
- Unfortunately, this **uniqueness among hardware platforms** leads to a lot **of additional software complexity**, and it's also the **reason** you'll **need** to be **more aware** of the **software build process** than ever before.

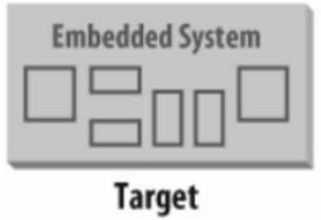
Building Process for Embedded Systems

- When **build tools run** on the **same system** as the **program** they **produce**, they can make a **lot of assumptions** about the system.
- This is **typically not the case in embedded software development**, where the *build tools run on a host computer* that *differs* from the *target hardware platform*.
- Embedded software development tools, **can rarely make assumptions** about the target platform.
- Instead, the **user must provide some knowledge** of the **system** to the **tools** by giving them **more explicit instructions**.

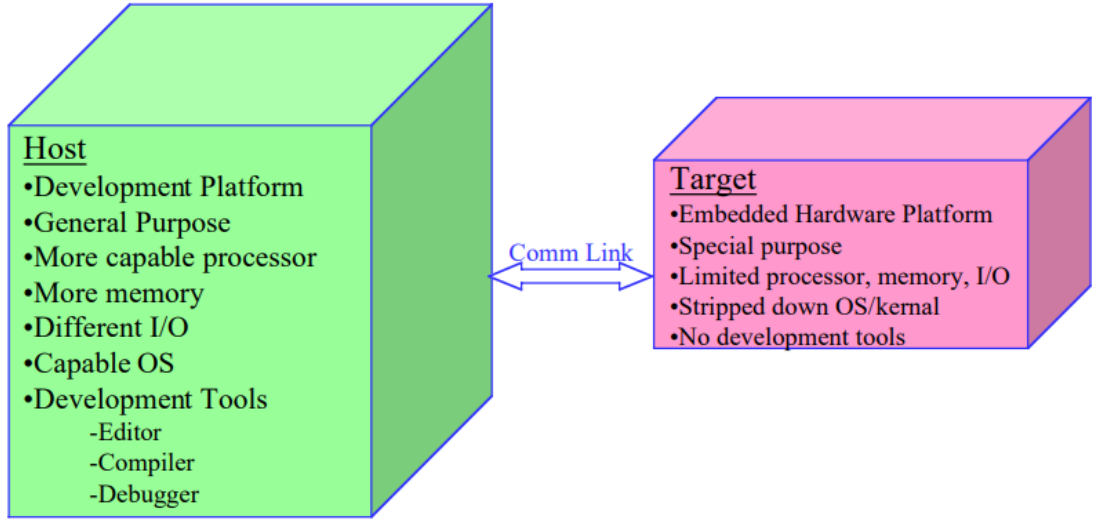
Building Process for Embedded Systems

- ***Software Development*** is ***performed*** on a ***Host computer***
  - *Compiler, Assembler, Linker, Locator, Debugger*
  - Produces **executable binary image** that will run on *Target Embedded System*

The development tools that build the embedded software run on a general-purpose computer



The embedded software that is built by those tools runs on the embedded system



Building Process for Embedded Systems

- **Software Tools**
  - *Software Development Kit (SDK)*
  - *Source-code Engineering Software*
  - *RTOS*
  - *Integrated Development Environment (IDE)*
  - *Emulator*
  - *Editor*
  - *Interpreter*
  - *Compiler*
  - *Assembler*
  - *Cross Assembler*
  - *Locator*
  - *Testing and debugging tools*

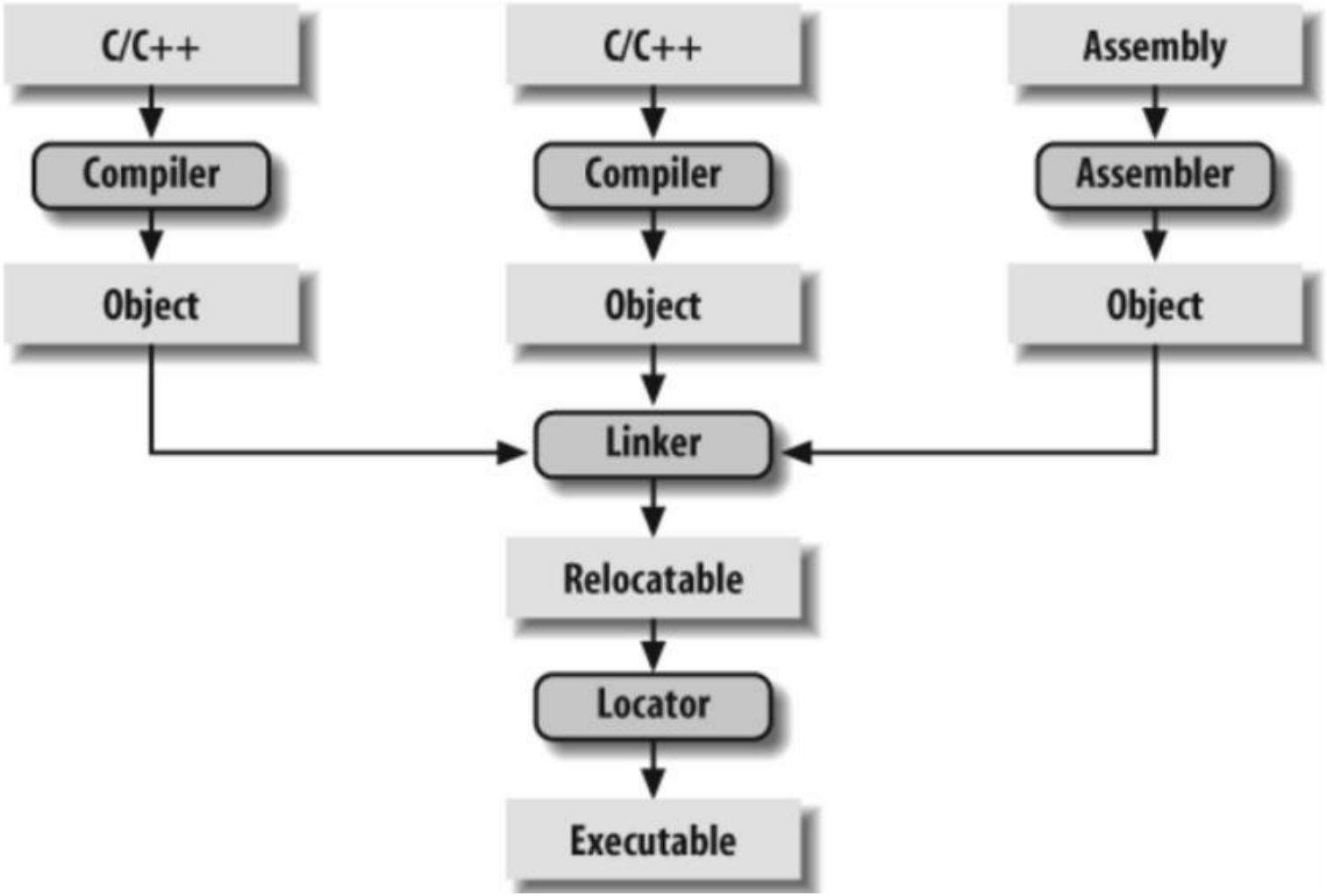
Building Process for Embedded Systems

- **To develop software for a General Purpose Computer**
  - Create source file
  - Type in C code
  - *Build*: compile and link
  - *Execute*: load and run

- **To develop software for an Embedded System**
  - Create source file (on Host)
  - Type in C code (on Host)
  - *Compile*/*Assemble*: translate into machine code (on Host)
  - *Link*: combine all object files and libraries, resolve all symbols (on Host)
  - *Locate*: assign memory addresses to code and data (on Host)
  - *Download*: copy executable image into Target processor memory
  - *Execute*: reset Target processor

Building Process for Embedded Systems

The Embedded Software Development Process



Building Process for Embedded Systems

- **Source Code into Executable Binary Image**

Each of the source files must be compiled or assembled into an object file

All of the object files that result from the first step ***must be linked together to produce a single object file***, called the re-locatable program.

Physical memory addresses must be assigned to the relative offsets within the re-locatable program in a process called relocation.

Preprocessing

- Pre-processing is *not a part of the compiler*, but is a separate step in the compilation process
- Pre-processing is a *process* of *running* a *set of instructions* provided by the programmer *explicitly as code before* the *program compiles*.
- Pre-processor *provides* the *ability* for the **inclusion** of **header** files, **macro** expansions, **conditional** compilation, and **line** control.
- Pre-processor instructions are called **pre-processor directives**, and they all start with a **hash symbol (#).**
- Few examples: "#*include*", "#*define*", "#*line*", and many more.

Compiling

- Compiler *translate programs* written in some human-readable language into an **equivalent *set of op-codes*** (or machine language) for a particular processor.
- **Compiler performs conversion** of
  - ***Source Code → Object file***
- ***Object file***
  - Binary file that contains *set of machine-language instructions* (opcodes) and data resulting from language translation Process
- ***Each processor*** has its ***own unique machine language***,
  - So you need to choose a compiler that produces programs for your specific target processor.

Cross Compiling

- In the embedded systems case, ***compiler*** almost always ***runs*** on the ***host computer***.
- **Native-compiler** runs on a computer platform and produces code for that same computer platform
- **Cross-compiler** runs on one computer platform and produces code for *another* computer/target platform
- The use of a cross-compiler is one of the defining features of embedded software development.

Compiler : *Example*

- *GNU C compiler (gcc)* and *assembler (as)* can be configured as either *native compilers* or *cross-compilers*.
- These tools *support* an *impressive* set of *host-target combinations*.
- ***gcc compiler*** will run on *all common PC* and *Mac* operating systems.
- ***Target processor support*** is **extensive**,
  - including AVR, Intel x86, MIPS, PowerPC, ARM, and SPARC.

Compiler : *Object File Format*

- *Regardless of the input language* (C, C++, assembly, or any other), the **output** of the **cross-compiler** will be an **object file**.
  - Although *parts* of this file *contain executable code*,
  - The *object file cannot* be *executed directly*.
- ***Contents of an object file*** can be thought of as a very large, flexible data structure.
- **Structure of the file:**
  - Often defined by a standard format such as the *Common Object File Format (COFF)* or *Executable and Linkable Format (ELF)*.

Compiler : *Object File*

- Most **object files begin** with a **header** that describes the sections that follow.
- Each of **these sections contains** one or more blocks of code or data that originated within the source file you created.
- However, the *compiler* has *regrouped these blocks* into *related sections*.
- ***For example, in gcc***
  - text - all of the code blocks are collected into this section,
  - data - initialized global variables (and their initial values) into this section
  - bss - uninitialized global variables into this section.

Compiler : *Symbol Table*

- There is also usually a **symbol table somewhere in the object file** that **contains** the *names and locations* of all *variables* and *functions referenced* within the source file.
- *Parts of this table may be incomplete*, however, because not *all of the variables and functions are always defined in the same file*.
  - These are the symbols that *refer to variables* and *functions defined* in *other source files*.
- And it is up to the **linker** to **resolve** such **unresolved references**.

Linker

- **All** of the **object files** resulting from the **compilation** in step one **must be combined**.
- **Object files** themselves are **individually incomplete**, some of the ***internal variable*** and ***function references not*** yet been ***resolved***.
- The **job** of the **linker** is to *combine these object files* and, *in* the *process*, to *resolve all* of the *unresolved symbols*
- Done by ***merging*** the ***text***, ***data***, and ***bss sections*** of the input object files, the ***linker creates*** a ***new object file*** that ***contains all*** of the ***code*** and ***data*** from the ***input object*** files.

Linker

- When the **linker** is **finished executing**, all of the ***machine language code*** from ***all input object files*** will be in ***text section*** of the ***new file***.
- And all of the ***initialized*** and ***uninitialized variables*** will ***reside*** in the ***new data*** and ***bss sections***, respectively.
- After ***merging*** all of the ***code*** and ***data sections*** and ***resolving*** all of the ***symbol references***, the **linker** *produces* an *object file* that is a *special* "*relocatable*" *copy* of the program.
- In other words, the **program is complete except for one thing**: **no memory addresses** have yet been assigned to the code and data sections within.

Locator

- The tool that *performs* the *conversion* from *relocatable program* to *executable binary image* is called a **locator**.
- You have to do most of the work in this step yourself, by *providing information* about the *memory* on the *target board* as *input* to the *locator*.
- The locator uses this information to ***assign physical memory addresses*** to each of the **code** and **data sections** within the **relocatable program**.
- It then *produces* an *output file* that *contains* a *binary memory image* that can be *loaded into* the *target*.

Locator

- The *output of this final step* of the build process is a **binary image** containing **physical addresses** for the *specific embedded system*.
- This *executable binary image* can be *downloaded* to the *embedded system* or *programmed* into a *memory chip*.
- There is a *separate development tool*, called a **locator**, to assign addresses.
- However, in the of ***GNU tools***, this feature is built into the *linker* (ld).
- The *memory information* required by the *GNU linker* can be passed to it in the *form* of a *linker script*.

Locator : *Example*

```
ENTRY (main)

MEMORY
{
    ram : ORIGIN = 0x00400000, LENGTH = 64M
    rom : ORIGIN = 0x60000000, LENGTH = 16M
}

SECTIONS
{
    data :                              /* Initialized data. */
    {
        _DataStart = . ;
        *(.data)
        _DataEnd   = . ;

    } >ram

    bss :                               /* Uninitialized data. */
    {
        _BssStart = . ;
        *(.bss)
        _BssEnd   = . ;
    } >ram

    text :                              /* The actual instructions. */
    {
        *(.text)
    } >ram
}
```

```
AREA     ARMex, CODE, READONLY
                                ; Name this block of code ARMex
         ENTRY                  ; Mark first instruction to execut
start
         MOV      r0, #10       ; Set up parameters
         MOV      r1, #3
         ADD      r0, r0, r1    ; r0 = r0 + r1
stop
         MOV      r0, #0x18     ; angel_SWIreason_ReportException
         LDR      r1, =0x20026  ; ADP_Stopped_ApplicationExit
         SVC      #0x123456     ; ARM semihosting (formerly SWI)
         END                    ; Mark end of file
```

Locator : *Example*

- The first executable instruction is **ENTRY command**, which appears on the **first line** and the **entry point** is the **function main**.
- This **script informs** the **GNU linker's built-in locator** about the **memory** on the **target board**, which contains 64 MB of RAM and 16 MB of flash ROM.
- The **linker script** file instructs the **GNU linker to locate** the data, **bss**, and **text** sections in RAM starting at address 0x00400000.
- Names in the linker command file that begin with an underscore (e.g., **_DataStart**) can be referenced similarly to ordinary variables from source code.

Locator : *Example*

- Linker will *use these symbols* to *resolve references* in the *input object files*.

- So, **for example**, there might be a *part of the embedded software* (usually within the startup code) that *copies the initial values* of the *initialized variables from ROM* to the *data section in RAM*.

- The *start* and *stop addresses* for this operation can be *established symbolically* by referring to the addresses as *_DataStart* and *_DataEnd*.

Build Procedure for **Arcom Viper-lite Development Board**

Build Procedure
Arcom Viper-lite Development Board

- In this section, we show an example *build procedure* for the *Arcom VIPER-Lite development board*.
  - *If another hardware platform is used, a similar process should be followed using the tools and conventions that accompany that hardware.*
- Once the tools are installed, the commands covered in the following sections are entered into a command shell.
- For Windows users, the command shell is a Cygwin bash shell (Cygwin is a Unix environment for Windows); for Linux users, it is a regular command shell.

Build Procedure
Arcom Viper-lite Development Board

COMPILE

- First we look at the *individual commands* in *order* to *manually perform* the *three separate tasks: compiling*, *linking*, and *locating*
- Later, we will learn how to ***automate*** the build procedure with ***makefiles***.
- **Blinking LED example**
  - Consists of **two source modules**: *led.c* and *blink.c*.
- The first step in the **build process** is to **compile** these **two files**.

Build Procedure
Arcom Viper-lite Development Board

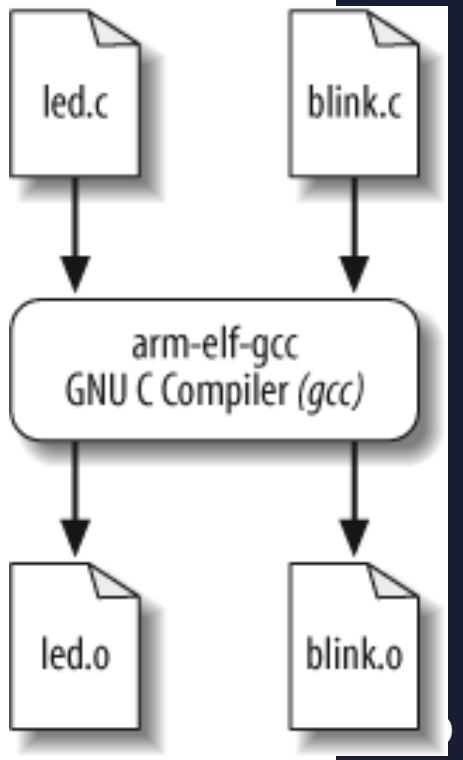
COMPILE

- The basic structure for the gcc compiler command is:

```
arm-elf-gcc [
                options
            ]
              file
            ...
```

- The command-line options we'll need are:
  - **-g** To generate **debugging** info in default format
  - **-c** To **compile** and **assemble** but **not link**
  - **-Wall** To **enable** most **warning messages**
  - **-I../include** To **look** in the **directory** include for **header files**
- Here are the actual commands for compiling:

```
# arm-elf-gcc -g -c -Wall -I../include led.c
# arm-elf-gcc -g -c -Wall -I../include blink.c
```



Build Procedure
Arcom Viper-lite Development Board

COMPILE

- We broke up the **compilation step** into **two separate commands**, but you can compile the two files with one command.
- ***To use a single command***, just put **both** of the **source files** after the **options**.
- If you wanted **different options** for one of the source files, you would need to **compile it separately** as just shown.
- The **result** of each of these commands is the **creation** of an **object file** that has the same **prefix** as the **.c file**, and **the extension .o**.
- ***So if all goes well***, there will now be **two additional files** — led.o and blink.o—in the working directory.

Build Procedure
Arcom Viper-lite Development Board

LINK AND LOCATE

- We now have the **two object files** — ***led.o*** and ***blink.o***
  - *that we need in order to perform the second step in the build process.*
- As we discussed earlier, the **GNU linker performs** the **linking** and **locating** of the **object files**.
- For the third step, **locating**, there is a **linker script file** named ***viperlite.ld*** that we input to ***ld*** in order to establish the ***location*** of ***each section*** in the Arcom board's memory.

Build Procedure
Arcom Viper-lite Development Board

```
arm-elf-ld [
                options
           ]
              file
           ...
```

LINK AND LOCATE

- The structure for the ***linker*** and ***locater ld command*** is:
- The command-line options we'll need for this step are:
  - -Map blink.map — To generate a **map file** and use the given filename
  - -T viperlite.ld — To **read** the **linker script**
  - -N — To set the **text** and **data** sections to be **readable** and **writable**
  - -o blink.exe — To set the output filename (if this option is not included, ld will use the default output filename a.out)
- The actual command for linking and locating is:

```
# arm-elf-ld -Map blink.map -T viperlite.ld -N -o blink.exe led.o blink.o
```

Build Procedure
Arcom Viper-lite Development Board

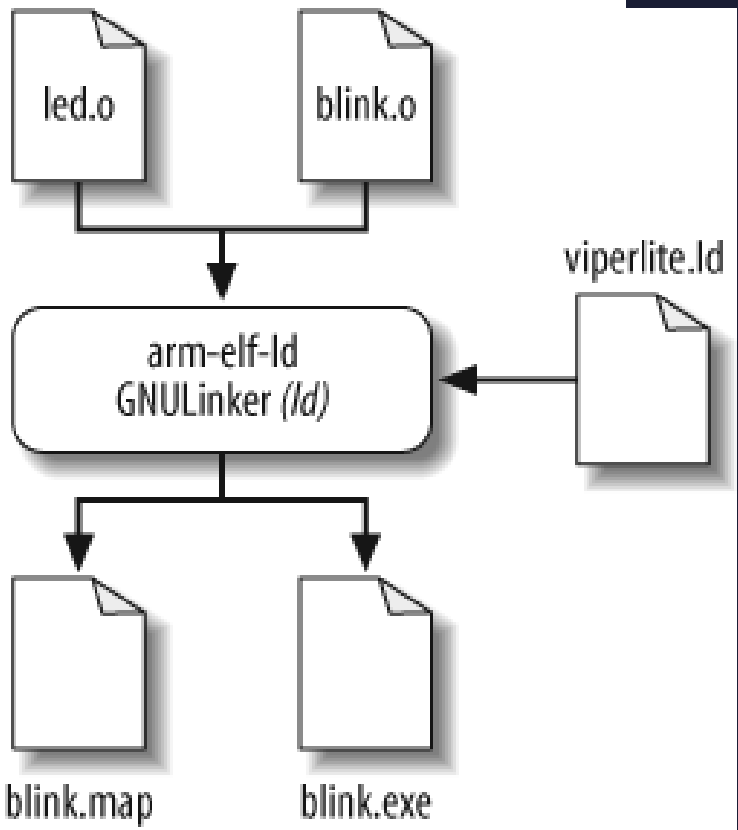
LINK AND LOCATE

- The ***order*** of the ***object files*** determines their **placement** in **memory**.
  - *Because we are not linking in any startup code, the order of the object files is irrelevant.*
- If ***startup code*** were ***included***, you would want that **object** file to be **located** at the **proper address**.
- ***Linker script file*** can be used to **specify** where you want the **startup routine** (and other code) to **reside** in **memory**.
- Also linker script file to specify **exact addresses** for **code** or **data**, should you find it necessary to do so.

Build Procedure
Arcom Viper-lite Development Board

LINK AND LOCATE

- **The two object files** — *led.o* and *blink.o* — are the last arguments on the command line for linking.
- **Linker script file**, *viperlite.ld*, is also passed in for **locating** the **data** and **code** in the Arcom board's memory.
- **Result** of this command is the creation of two files— *blink.map* and *blink.exe* —in the working directory.



Build Procedure
Arcom Viper-lite Development Board

LINK AND LOCATE

- The **.map file** gives a *complete listing* of all *code* and *data addresses* for the final software image.
- It provides information *similar* to the *contents* of the *linker script* described earlier.
- However, these are **results** rather than instructions and therefore *include the actual lengths* of the *sections* and the *names* and *locations* of the public symbols found in the relocatable program.

Linker Map Files

- **Linker map** file provides valuable information that can help you ***understand*** and ***optimize memory***.
- Map file is a ***symbol table*** for the ***whole program***.
- The most straightforward **pieces of information** in the **map file** are the *actual memory regions*, with *location*, *size* and *access rights* granted to those regions

```
Memory Configuration

Name             Origin             Length             Attributes
FLASH            0x0000000000001000 0x000000000000ff000 xr
RAM              0x0000000020000008 0x000000000003fff8 xrw
*default*        0x0000000000000000 0xffffffffffffffff
```

Linker Map Files

- ***Linker script*** and ***memory map*** section contains a **breakdown** of the **memory contribution** of **each** and **every file** that was linked into the final image.

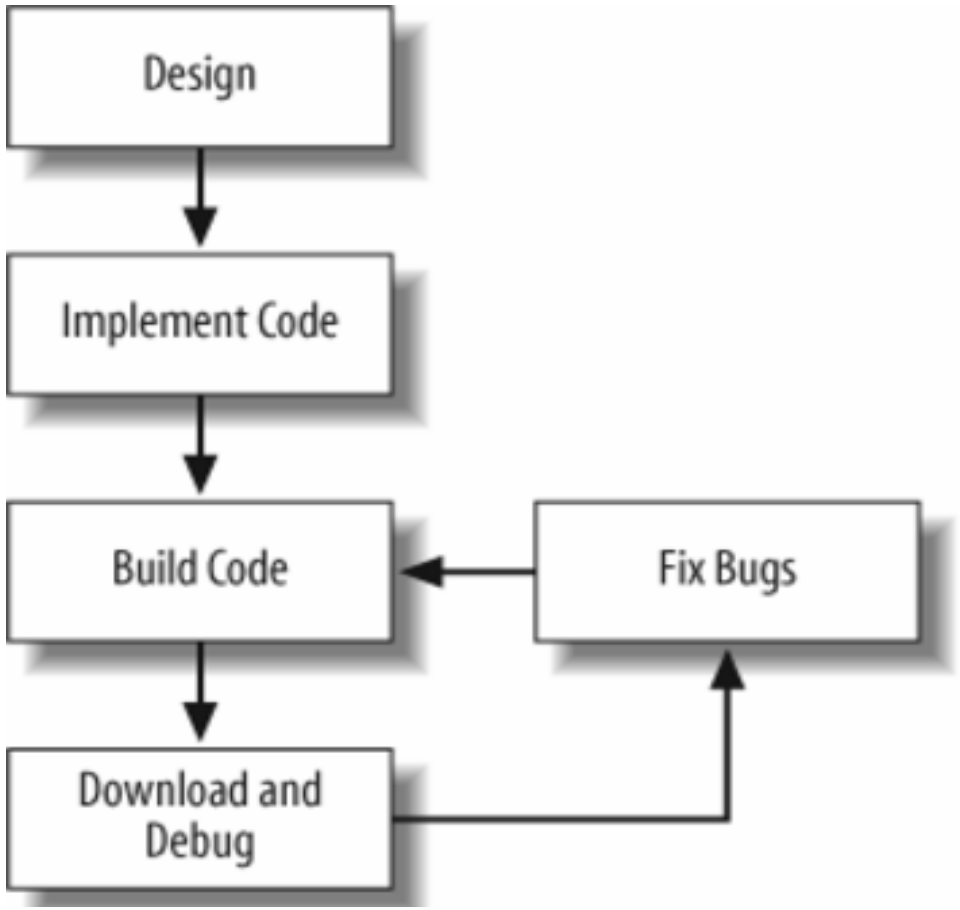
```
Linker script and memory map

.text           0x0000000000001000      0x8c8
 ***(.isr_vector)**
 .isr_vector    0x0000000000001000      0x200 _build/nrf52840_xxaa/**gcc_startup_nrf52840.
                0x0000000000001000                __isr_vector
 *(.text*)
 .text          0x0000000000001200       0x40 /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q
 .text          0x0000000000001240       0x74 /usr/local/Cellar/arm-none-eabi-gcc/8-2018-q
                0x0000000000001240                _mainCRTStartup
                0x0000000000001240                _start
 .text          0x00000000000012b4       0x3c _build/nrf52840_xxaa/gcc_startup_nrf52840.S.
                0x00000000000012b4                Reset_Handler
                0x00000000000012dc                NMI_Handler

[...]

 .text.**bsp_board_led_invert**
                0x00000000000012f0       **0x34** _build/nrf52840_xxaa/**boards.c.o**
                0x00000000000012f0                bsp_board_led_invert
```

Loading on the Target

- Previously, we saw how the *code* or *software* to be *executed* on the *embedded system* (target board) is **written** on a **computer**.
- **Resulting code** created after subjecting it to be build process is called the ***binary executable image*** or simply ***hex code***.
- This topic explains how the ***hex code*** *is* ***loaded*** on the ***target board*** which is referred as **downloading**
- And what are the various possible ways of debugging a code meant to run on a embedded system.

Loading on the Target

- *Two ways of downloading the binary image on the embedded system:*
  - Using a Device Programmer
  - In-system programming (ISP)



Loading on the Target

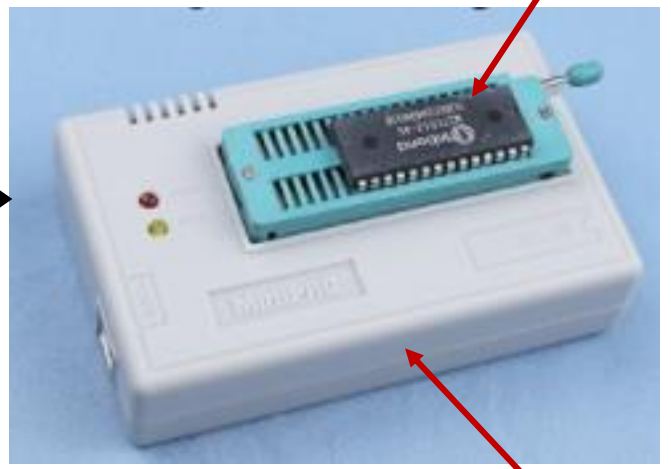
Using a Device Programmer:

- ***Step 1:*** Once the ***binary image*** is ***ready*** on the computer, the ***device programmer*** is ***connected*** to the ***computer***.
- ***Step 2:*** The ***uP/uC*** or ***memory chip***, usually the ***ROM*** which is supposed to contain the **binary image** is placed on the **proper socket** on the **device programmer**.
- ***Step 3:*** **Device programmer** contains a **software interface** through which the **user selects** the **target uP/uC** for which the **binary image** has to be **downloaded**.
- ***Step 4:*** Device programmer then ***transfers*** the ***binary image bit by bit*** to the chip.

Loading on the Target

- **Using a Device Programmer:**

Host Computer



Loading on the Target

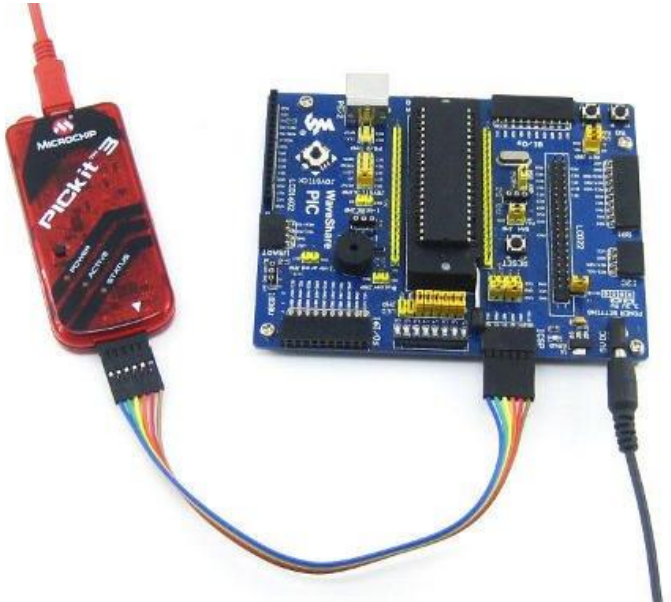
Using In System Programmer(ISP):

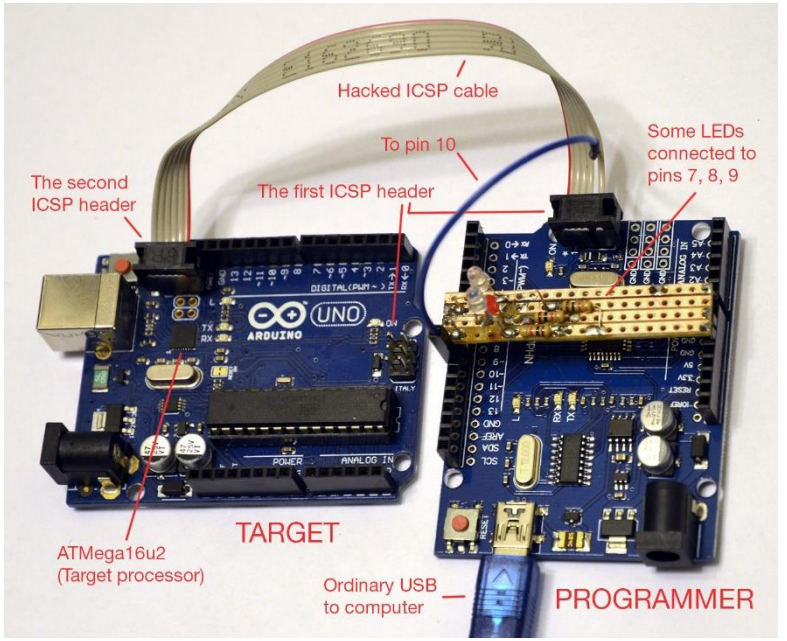
- Certain Target embedded platforms uses a piece of **hardware** called **ISP**.
  - A *hardware interface* to *both* the *computer* as well the *target board's chip* where the code is to be downloaded.
- **ISP** is also called **in-circuit serial programming** (ICSP),
  - Is the *ability of programming* the *embedded processor/controller* while it is *installed* in a *complete system*, rather than requiring the chip to be programmed prior to installing it into the system.
  - Allows ***firmware updates*** to be *delivered* to the *on-chip memory* of *microcontrollers* and related processors without *requiring specialist programming circuitry* on the circuit board, and simplifies design work

Loading on the Target

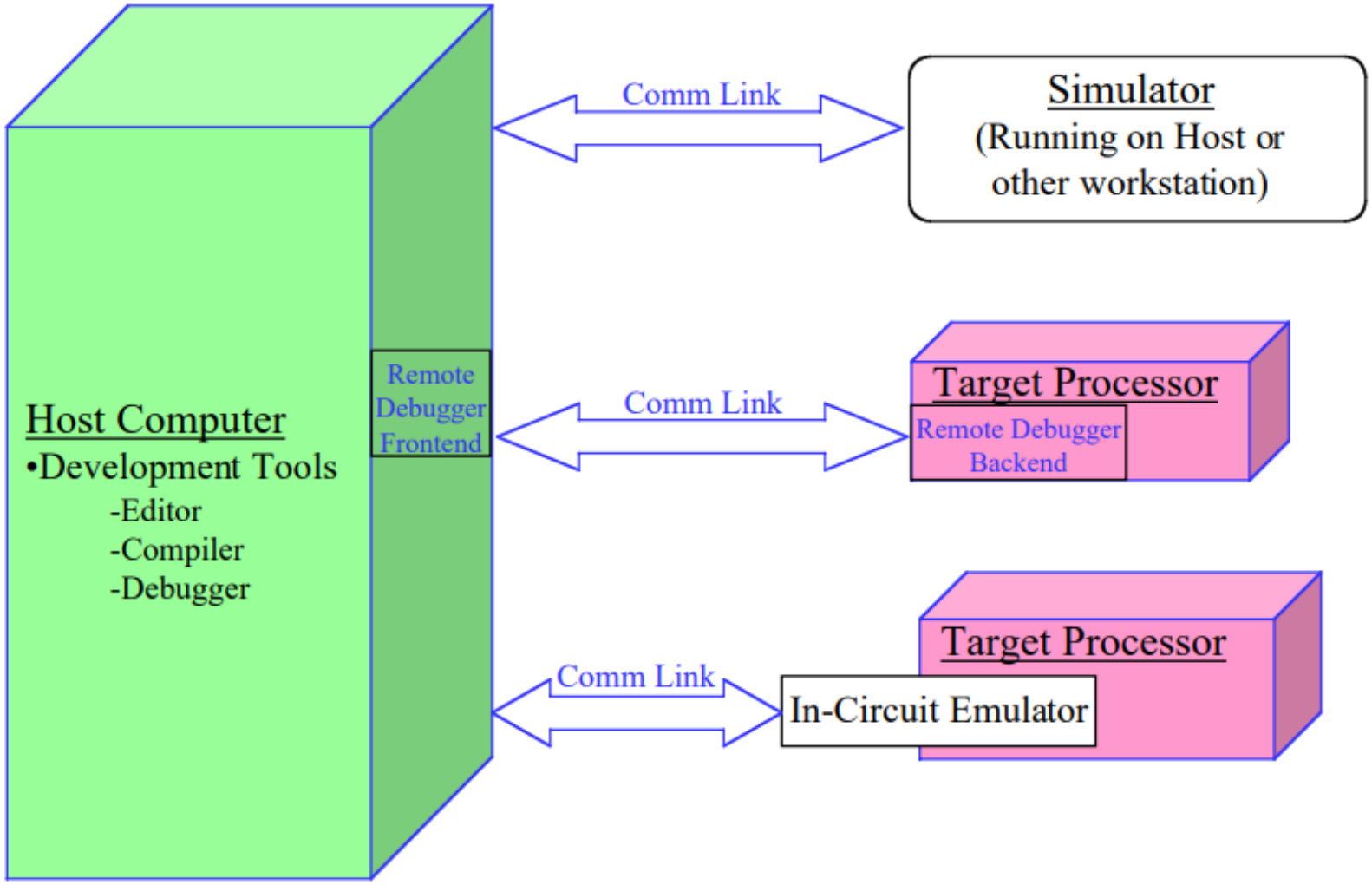
Using In System Programmer(ISP):

- User through the ISP's software interface sends the binary image to the target board.
- This ***avoids*** the ***requirement*** of ***frequently removing*** the ***microprocessor / microcontroller*** or ***ROM*** for downloading the code if a device programmer had to be used.





Debugging Tools



Debugging Tools

SIMULATOR

- Simulator is a ***host-based program*** that **simulates** ***functionality*** and ***instruction*** set of **target processor**.
- **Front-end** has **text** or **GUI-based** windows for source code, register contents, etc.
- **Simulators** are valuable during ***early stages of development***.
- **Disadvantage** of this method is that, it only ***simulates the processor***, and ***not the peripherals***.

Debugging Tools

REMOTE DEBUGGER

- Remote debuggers are one of the commonly used **downloading** and **testing tools** during development of embedded software
- It is used to ***monitor***/***control embedded SW***.
- It is used to **download**, **execute** and **debug** embedded software over a **comm. link**.
- ***Program running*** on the ***host*** of a ***remote debugger*** has a ***user interface*** (GUI/Command-line) that looks just like other debugger
- The **front-end** of the **GUI debuggers** contain several windows to show the active part of the *source code*, *current register contents*, and other relevant information about the executing program.

Debugging Tools

REMOTE DEBUGGER

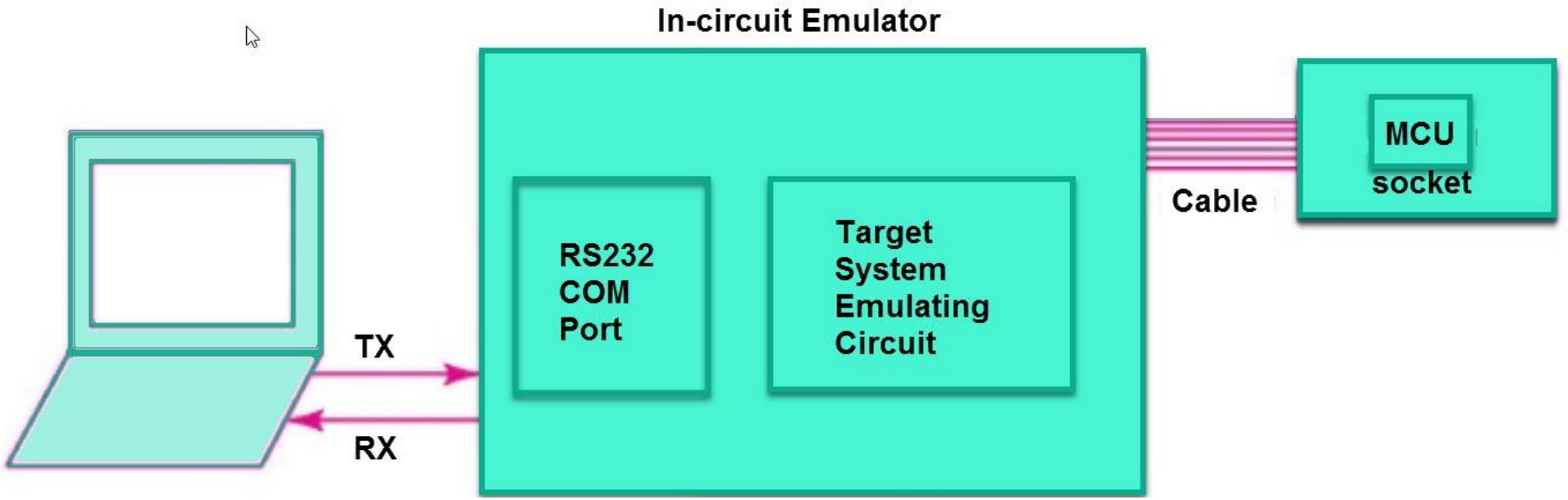
- **Backend** provides *low-level control* of target processor, runs on *target processor* and *communicates* to the *front-end* over a *communication link*.
- ***Debugger*** and ***software being debugged*** are executing on two different computer systems.
  - - Start/restart/kill, and stepping through program.
  - - Software breakpoints.
  - - Reading/writing registers or data at specified address.
- **Disadvantage**: inability to *debug startup code*, code must execute from RAM, requires a target processor to run the final software package

Debugging Tools

IN-CIRCUIT EMULATOR (ICE)

- Emulation refers to the ***ability*** of a ***computer program*** or ***electronic device*** to ***imitate another program*** or ***device***.
- **An emulator** is a piece of *hardware/software* that **enables** one ***computer system*** to ***run programs*** that are ***written*** for ***another computer system***.
- An ***in-circuit emulator (ICE)*** provides a lot more functionality that a remote debugger.
- In addition to providing the features available with a remote debugger, an ICE allows you to *debug startup code* and *programs running from ROM*, etc.,

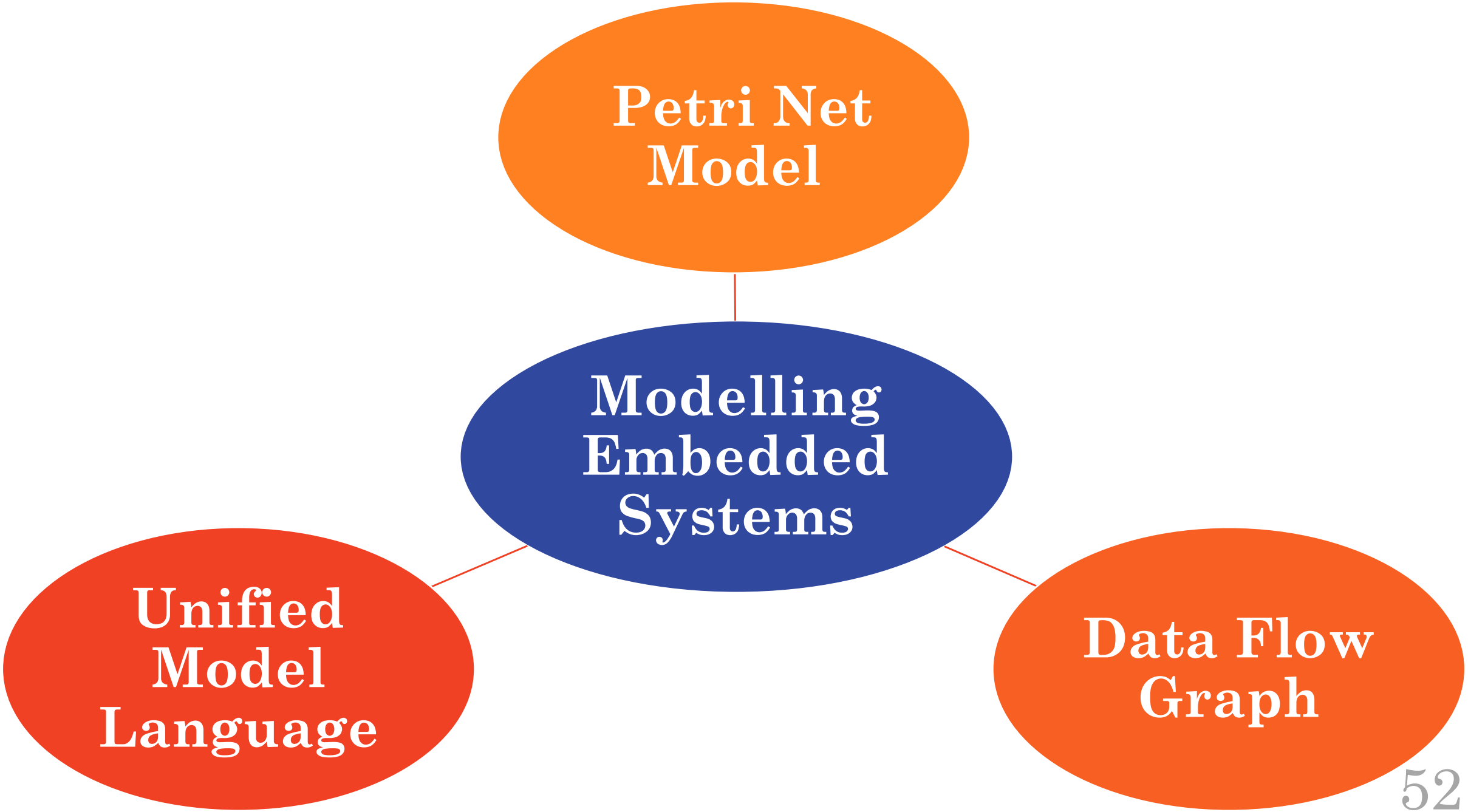
Debugging Tools



Debugging Tools

IN-CIRCUIT EMULATOR (ICE)

- In-Circuit Emulator (ICE) takes the ***place of the target processor***.
- It contains a **copy** of **target processor**, plus *RAM*, *ROM*, and its *own embedded software*.
- It allows you to ***examine*** the ***state of the processor*** while the program is ***running***.
- It uses the remote debugger for human interface.
- ICE **provides greater flexibility**, ease for developing various applications on a single system in place of testing that multiple targeted systems.
- **Disadvantage** of this method is that, it is ***expensive***.


Petri Net Model
Modelling Embedded Systems
Unified Model Language
Data Flow Graph

Modelling Programs

- Modelling the programs in Embedded systems.
- Program has ***basic blocks***, ***conditional statements*** **etc.**
- **Can be represented by**
  - **Data Flow Graph (DFG)**
  - **Control Data Flow Graph (CDFG)**

Modelling Programs

A. Model a program that has no conditionals

- **Basic blocks - having one entry and exit point.**
  - *Before go for DFG represent the program in single assignment form*
- **Single Assignment Form**
  - **Having statement in a program, where the *assignment variables (in left) appears only one*.**
  - **It verifies no cyclic form in the statements or blocks.**
  - **It identifies the unique location of variables in the code**
  - **Any use of repeated assignment to single variable need to be rewrite and assigns the latest assigned variable if it is used further.**

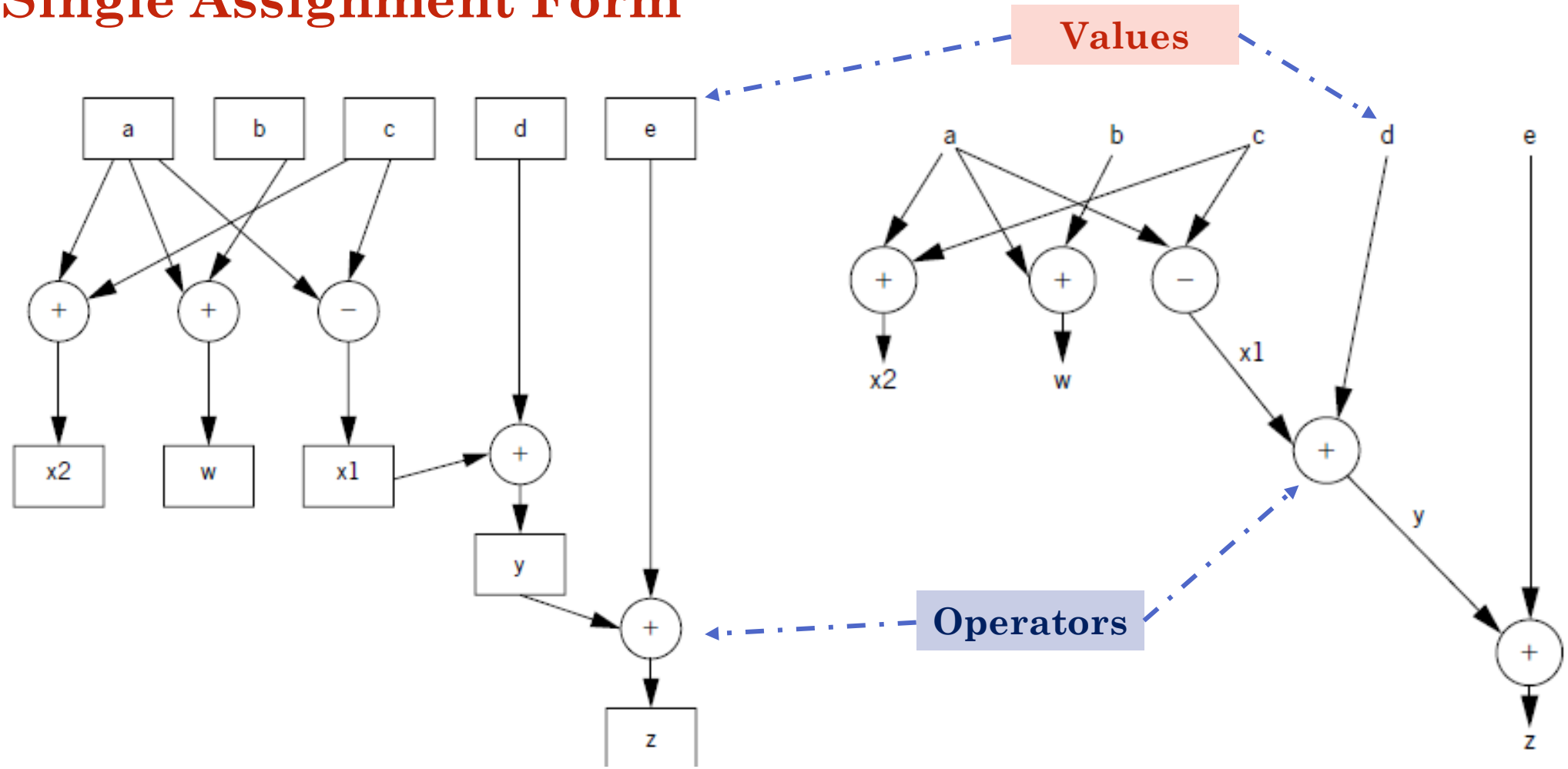
Modelling Programs

Two types of nodes

- Round nodes
  - Represents or denotes operators.
- Square nodes
  - Represents values.

Modelling Programs

Single Assignment Form



Modelling Programs

Advantages of DFG:

- It orders the way operations can be performed in a program.
- Determines the feasible reordering's of the operations, this will reduce the pipeline or cache conflicts

Modelling Programs

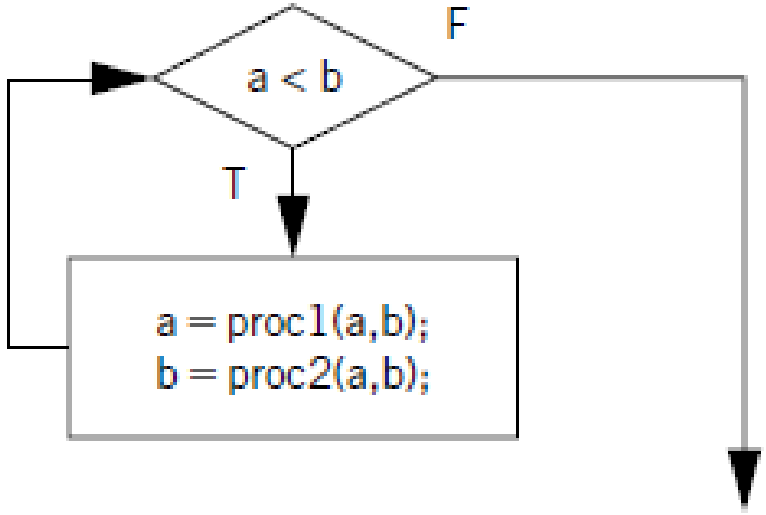
Control DFG:

- Constructs a model for **decision flow** and **data flow** of a program.
- **Data flow nodes** represented by **basic block** (set of statements).
- **Control flow** of a sequential program is represented by **decision nodes**
- **Basic Block Nodes**
  - Decision nodes
  - Edges labelled with possible outcomes

Modelling Programs

Building Loop in Control DFG:

```
while (a < b)
{
a = proc1(a,b);
b= proc2(a,b);
}
```

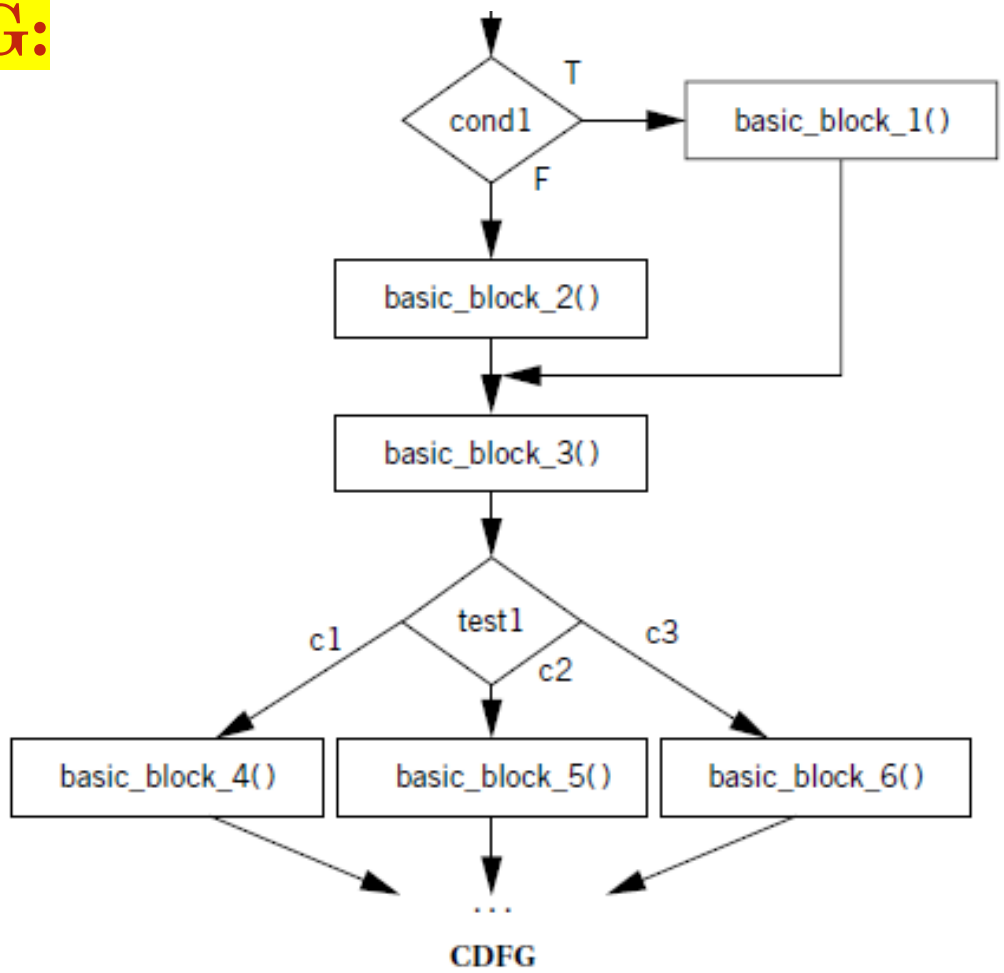


CDFG

Modelling Programs

Conditional Blocks in Control DFG:

```
if (cond1)
    basic_block_1( );
else
    basic_block_2();
basic_block_3( );
switch (test1) {
    case c1: basic_block_4( ); break;
    case c2: basic_block_5( ); break;
    case c3: basic_block_6( ): break;
}
```



CDFG

Modelling Programs

Advantages of Control DFG:

- **CDFG is a hierarchical representation.**
- **Each block in CDFG is expanded using Data flow graph**

Petri Net
Model

Petri Net Model

- Petri net can also called as **Place/Transition Net**
- It is used for **describing** and **analyzing concurrent process**
- Petri net is also a **graphical tool**
- Petri Net(PN) is **very similar** to **State Transition Diagrams** which model the system behavior
- It is **an abstract model** to show the **interaction** **between asynchronous processes**
  - *In asynchronous process, start and sequence of processes may vary during the execution*

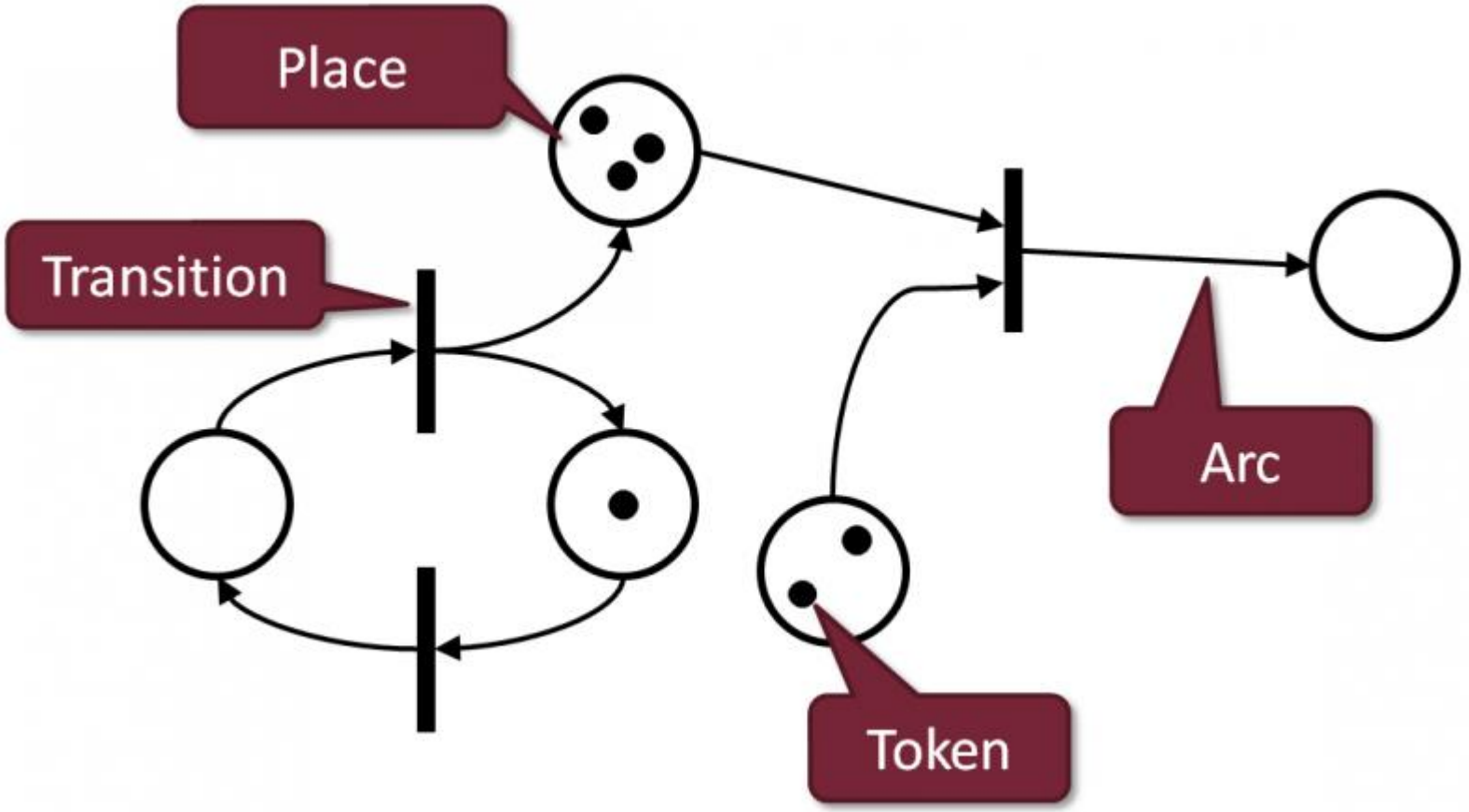
Petri Net Model

COMPONENTS OF PETRINET MODEL

- Petri net Model consists of four types of components –
  - **Places (circles)** : represent **possible states** of the system;
  - **Transitions (rectangles)** : are **events** which **cause** the **change** of **state**
  - **Arcs (arrows)** : Every arc simply ***connects*** a ***place*** with a ***transition*** or a ***transition*** with a ***place***.
  - **Tokens(black dot)**: ***Change*** of ***state*** is denoted by a ***movement*** of token(s) from place(s) to place(s)

Petri Net Model

COMPONENTS OF PETRINET MODEL



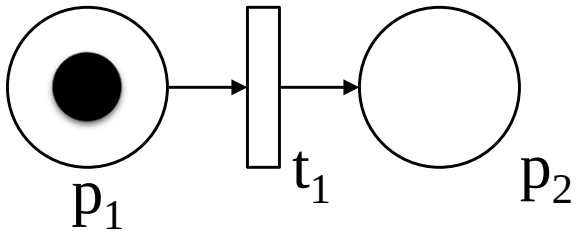
Petri Net Model

- ***Change of state*** is caused by the ***firing*** of a ***transition***
- **Firing** refers to the ***occurrence*** of an ***event***
- **Firing** is based on the ***input conditions***, denoted by ***token availability***
- **Transition** is enabled when ***sufficient tokens*** are ***available*** in input places
- **Once firing is initiated**, ***tokens*** will be ***transferred*** from ***input*** to ***output places***

Petri Net Model

COMPONENTS OF PETRINET

- Below is an example Petri net with **two places** and **one transaction**.

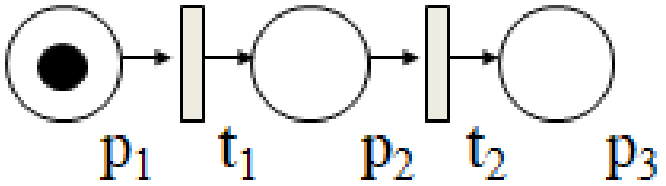


- p_1: input place
- p_2: output place

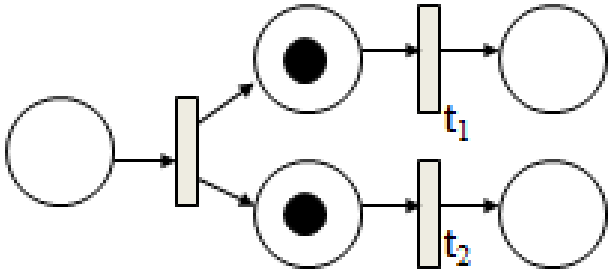
Petri Net Model

PROPERTIES

- ***Sequential Execution*:**
  - Transition t2 can take place only after t1.
  - Here a constraint is imposed “t2 after t1”



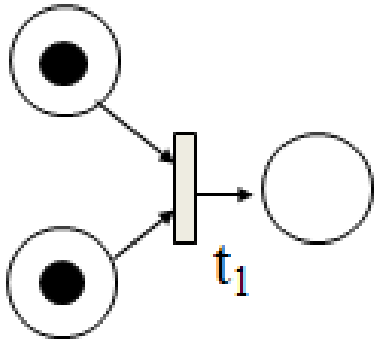
- ***Concurrency*:**
  - This property is used in modelling distributed control system.
  - t1 and t2 are concurrent process



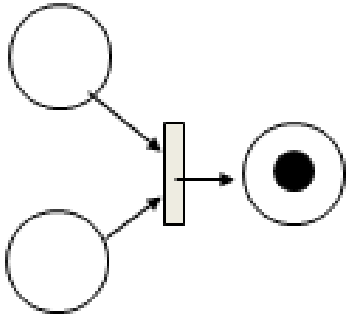
Petri Net Model

PROPERTIES

- **Merging:**
  - When ***several tokens*** are ***arrived*** at same transition merging will take place



- **Synchronization:**
  - **Transition t1** will be **enabled only** when at **least one token** is **available** at each of its input places



Petri Net Model : *Example*

EXAMPLE – 1: POINT OF SALE (POS) MACHINE

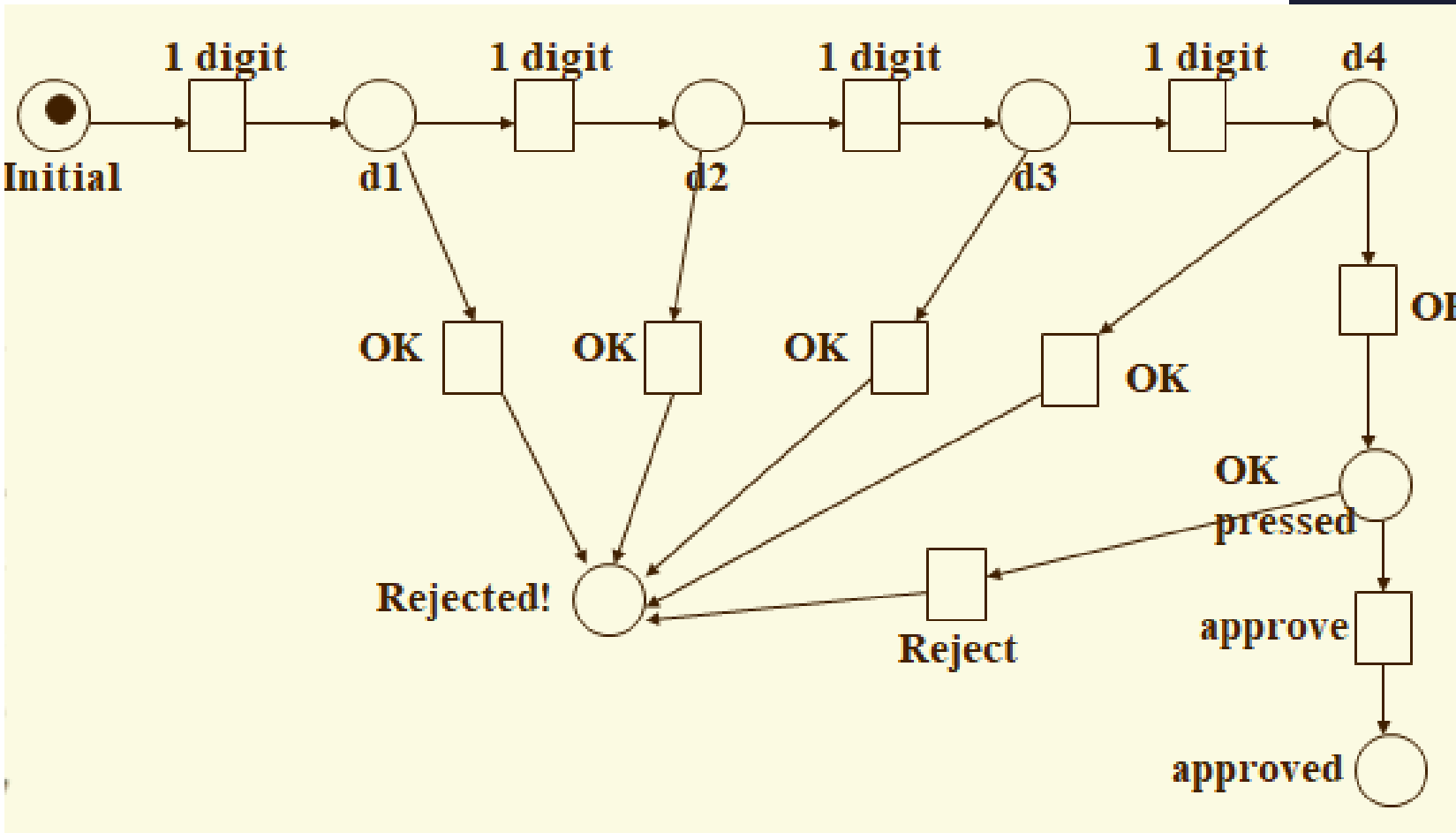
- *A point of sale terminal (POS terminal) is an electronic device used to process card payments at retail locations.*

- **POS terminal generally does the following:**
  - ***Reads*** the ***information*** off a ***customer's*** credit or debit ***card***
  - ***Validate*** the user using 4-digit ***PIN number***
  - ***Checks*** whether the ***funds*** in a customer's bank account are ***sufficient***
  - ***Transfers*** the ***funds*** from the ***customer's*** account to the ***seller's account***
  - ***Records*** the ***transaction*** and ***prints*** a ***receipt***

Petri Net Model : *Example*

EXAMPLE-1 POINT OF SALE (POS) MACHINE

- **Scenario 1: Normal**
  - Enters all 4 digits and press OK.
- **Scenario 2: Exceptional**
  - Enters only 3 digits and press



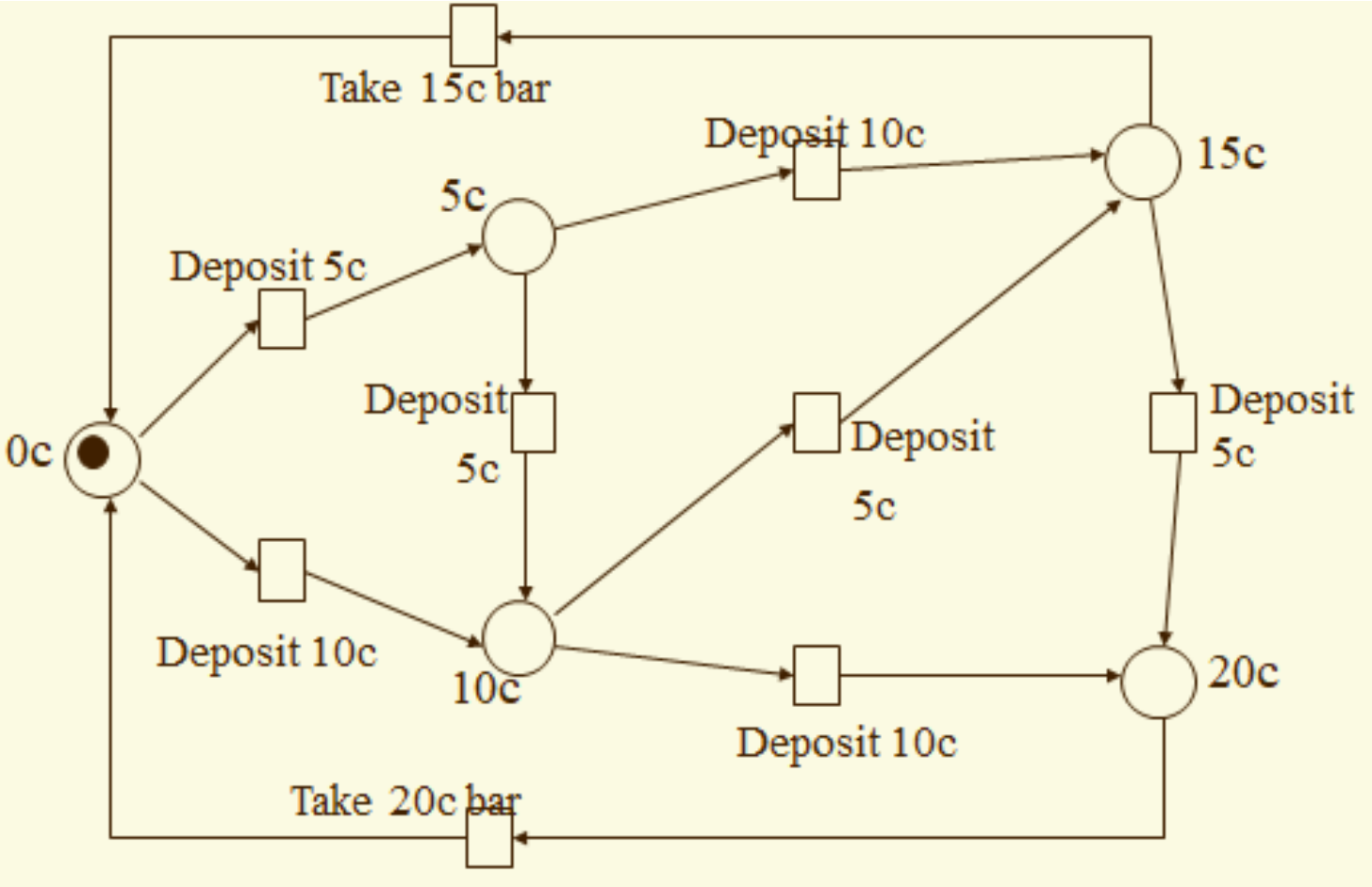
Petri Net Model : *Example*

EXAMPLE – 2 : VENDING MACHINE

- Machine dispenses ***two kinds*** of ***snack bars 20c*** and ***15c***.
- **Constraint**: **10c** and **5c coins** can **only** be **used**
- **Scenario 1:**
  - Deposit four 5c, take 20c snack bar.
- **Scenario 2:**
  - Deposit 10 + 5c, take 15c snack bar.
- **Scenario 3:**
  - Deposit 5 + 10 + 5c, take 20c snack bar.

Petri Net Model : *Example*

EXAMPLE – 2 : VENDING MACHINE



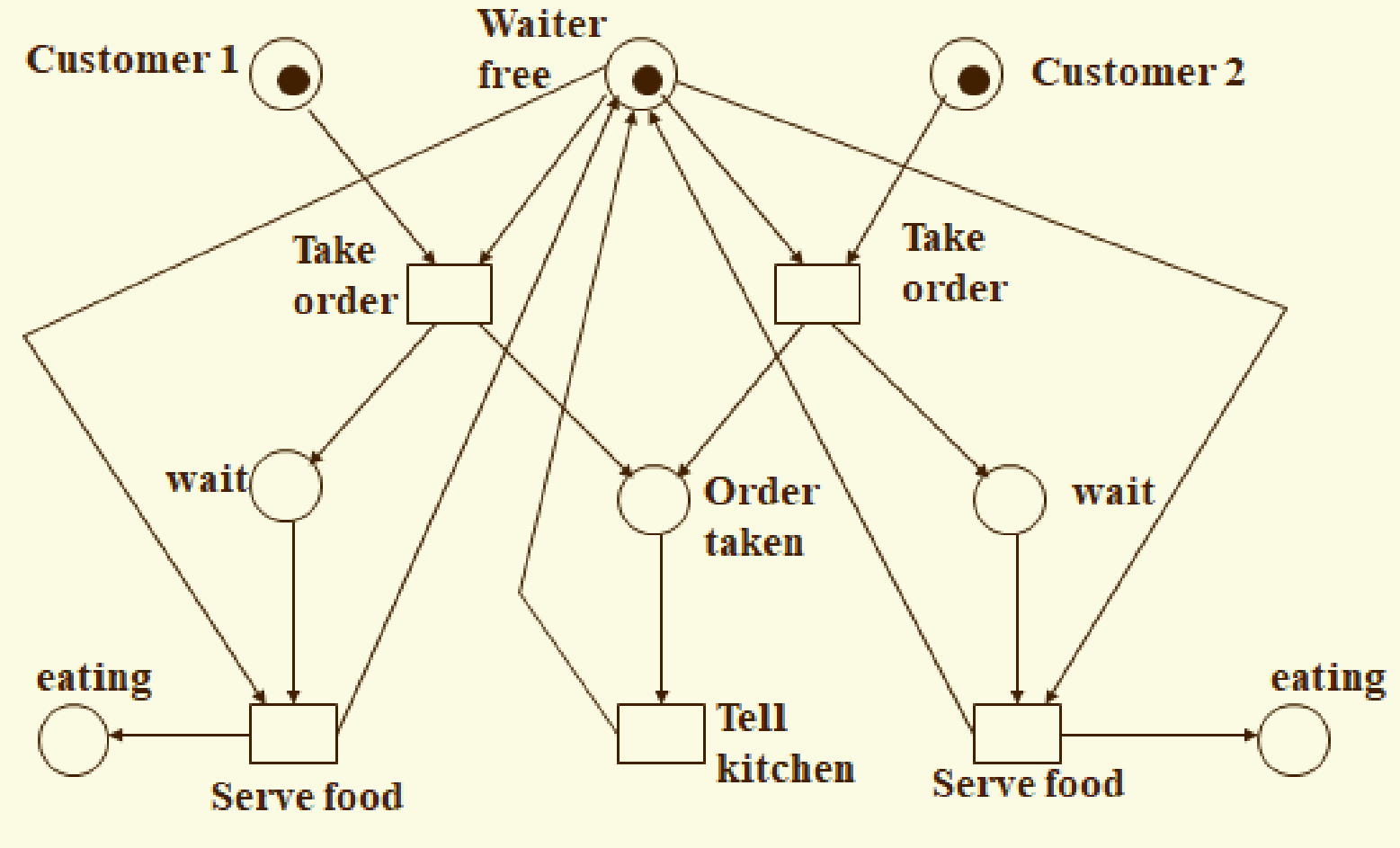
Petri Net Model : *Example*

EXAMPLE – 3 : ORDER MANAGEMENT IN A RESTAURANT

- *In the real world, many events are concurrent in nature. Many applications have global state formed from local state.*
- ***Scenario 1:***
  - Waiter takes order from customer 1
  - Serves customer 1
  - Takes order from customer 2
  - Serves customer 2.
- ***Scenario 2:***
  - Waiter takes order from customer 1
  - Takes order from customer 2
  - Serves customer 2
  - Serves customer 1

Petri Net Model : *Example*

EXAMPLE – 3 : ORDER MANAGEMENT IN A RESTAURANT



Unified Modelling **Language**

Unified Modelling Language

- **What is Modelling Language?**
  - Modelling language is a **graphical/textual computer language** explains ***design*** and ***construction*** of any model or structure ***following a set of guidelines***.

- ***Importance of Modelling*** : Visualization, reduced complexity, documentation

- To effectively model a system, you need a language with which the model can be described and here's where UML comes in.

Unified Modelling Language

INTRODUCTION

- **UML - general purpose visual modelling language**
- It is used to *visualize*, *specify*, *construct*, and *document*
- Using tools, *codes* can be *generated* with *UML diagrams* in various languages
- It is a pictorial language
- It can be used for ***both modelling*** the ***software*** system and ***non-software*** systems
- For example, ***modelling a process flow in automated systems***, etc.

Unified Modelling Language

INTRODUCTION

- UML is used in ***many applications*** by many people like business users, common people to make the system simple, clear and understandable
- It is a ***simple modelling*** mechanism to model most of the practical systems
- It is a ***object-oriented analysis*** based method

Unified Modelling Language

INTRODUCTION

- **UML Uses:**
  - Forecast systems
  - To estimate the reusability.
  - Lower costs
  - Plan and analyze system behavior
  - Easier maintenance/modification

Unified Modelling Language

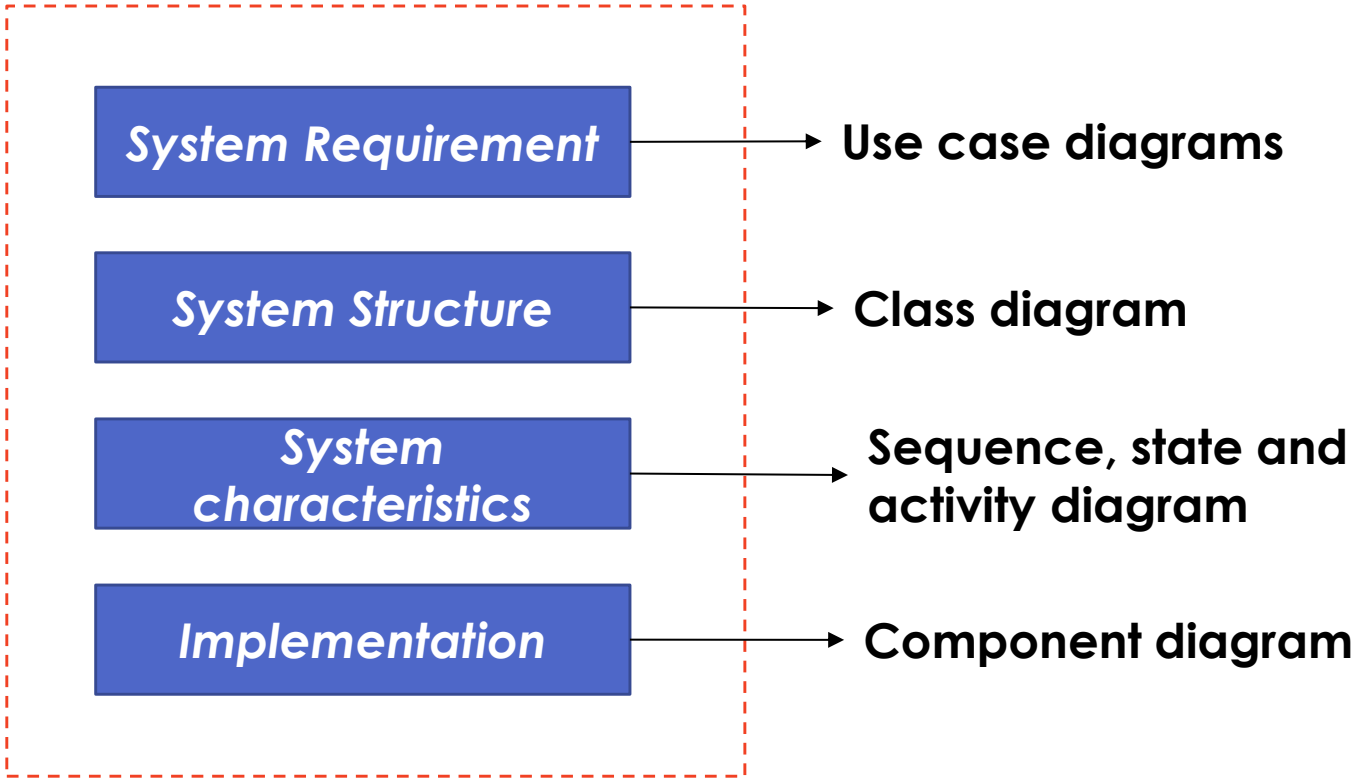
INTRODUCTION

There are two types of diagram in UML

- **Structure diagram:** explains *static structure* of a system
  - Class diagram
  - Object diagram
  - Deployment diagram
  - Package diagram
- **Behavior diagram** : used to *model dynamic changes*
  - Use case diagram
  - Interaction diagram
  - Activity diagram
  - State diagram

Unified Modelling Language

Principles of UML Modelling



Use Case **Diagram**

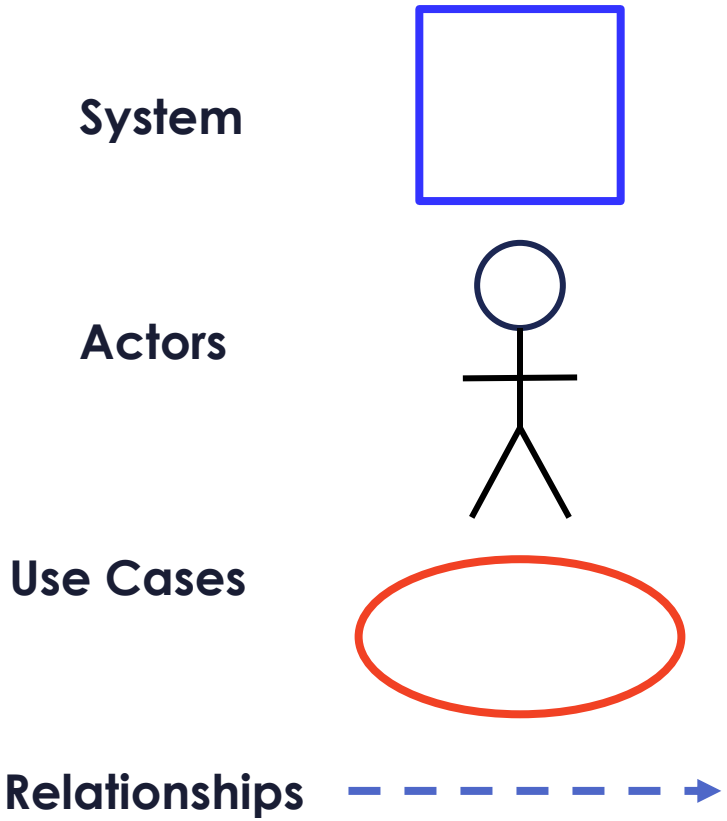
Unified Modelling Language

USE CASE DIAGRAM

- The use case diagram shows **different ways** a **user** can **interact with the system**
  - *It can be used in scenarios like when system interact with people or any other external systems*
- It contains "***use-cases***" and "***'actors***"
- UCD **depicts** how the ***actor interacts*** with ***use cases***
  - Describes the ***relationship*** between ***functionalities*** and their ***controllers***

Unified Modelling Language

USE CASE DIAGRAM



Unified Modelling Language

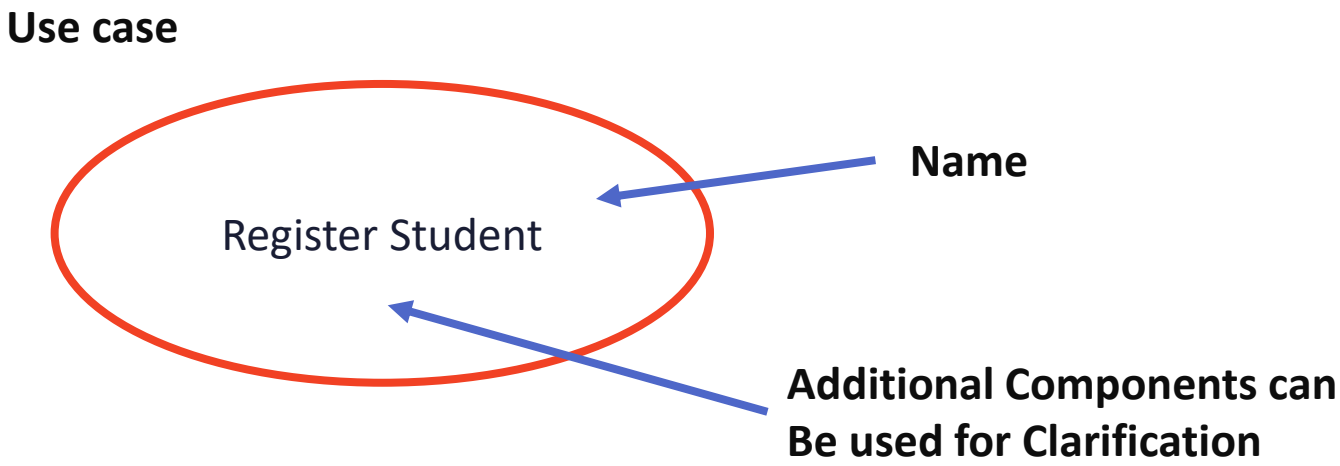
USE CASE DIAGRAM

- **Notations** are very important in any modelling language
- Efficient use of notations play a major role in making any model meaningful
- **Use case diagram**: **Notations of things** and **relationships**
- Extensibility makes UML more powerful and flexible.

Unified Modelling Language

USE CASE NOTATIONS

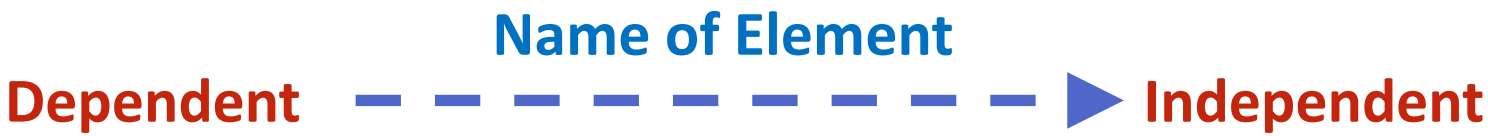
- **Use case** - **high level functionalities** of a system.
- Use case is represented as an eclipse



Unified Modelling Language

USE CASE NOTATIONS

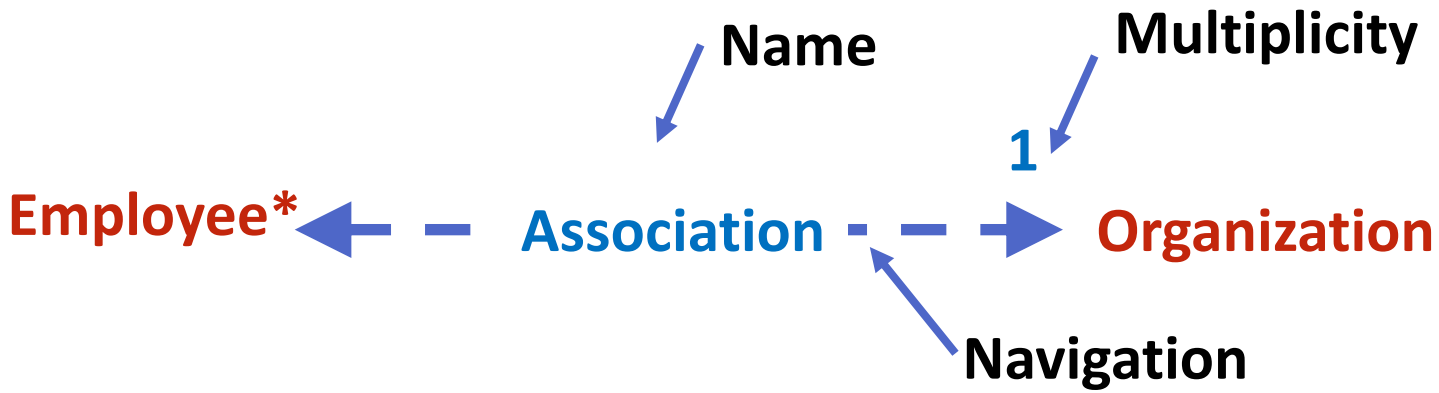
- **Dependency**
  - **Relationship** between **two elements** of a system
  - **Dependency** - **dotted arrow**
  - **Arrow head represents** - independent element
  - Other end - dependent element



Unified Modelling Language

USE CASE NOTATIONS

- **Association**
  - **Relationship between two elements**
  - Association describes how the ***elements are associated***.
  - Association is represented by a **dotted line with (without) arrows on both sides**
  - The **multiplicity** (1, *, etc.) to show how many objects are associated



Unified Modelling Language

USE CASE NOTATIONS

- **Generalization**
  - Parent-child relationship
  - **Generalization** - **inheritance relationship** of the object-oriented concept
  - Generalization - ***arrow with a hollow arrow head***

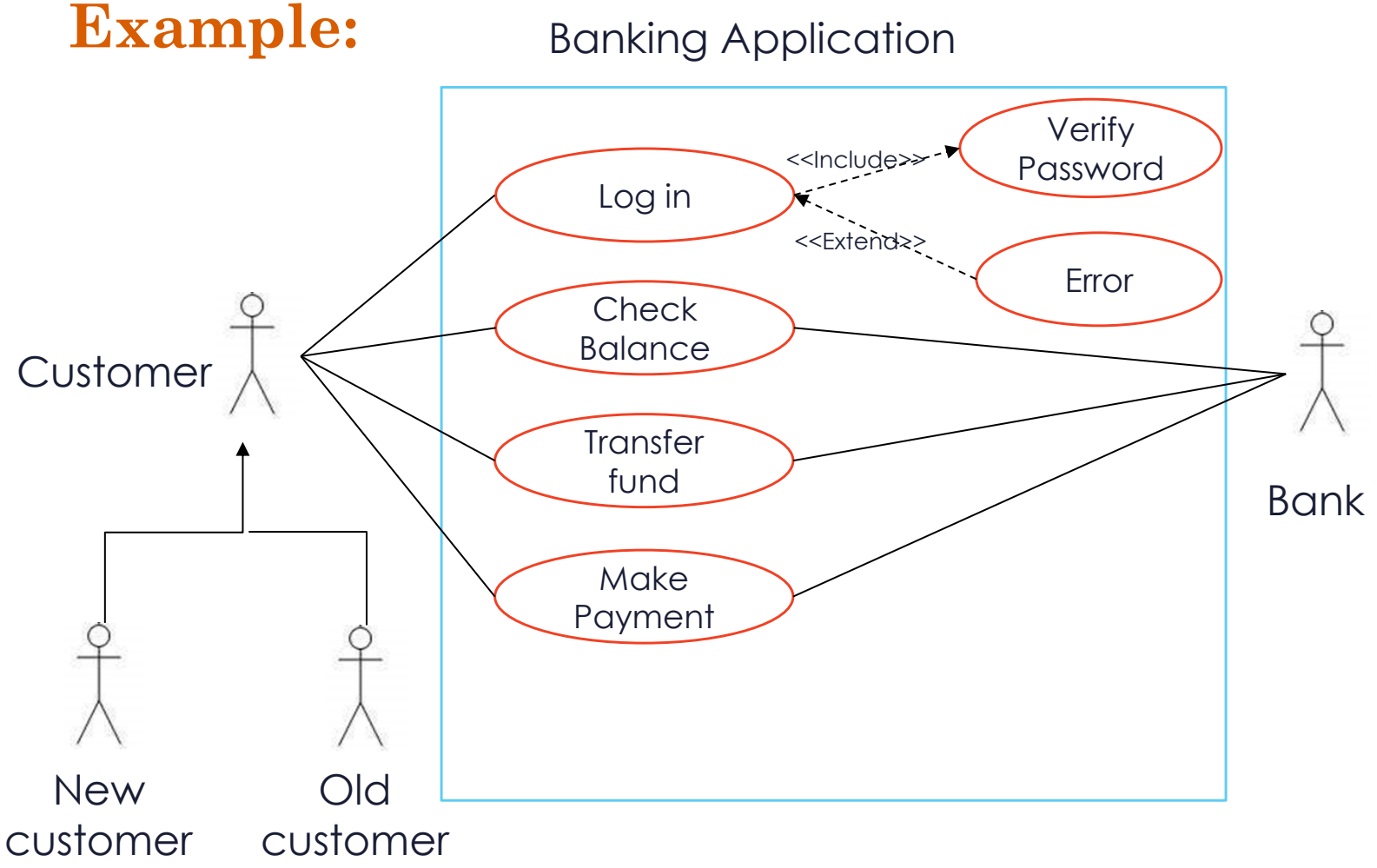
Unified Modelling Language

USE CASE NOTATIONS

- **<<include>> and <<extend>>**
  - '**Extend**' and '**Include**' – between **use cases**
  - **Include**: **invocation** of **one use case** by the **other**
  - **Extend**: **extending use case** will **work exactly** like the **base use case**

Unified Modelling Language

Example:



Primary actor – Initiates the process

Secondary actor – Responds

Use case – Action

Association

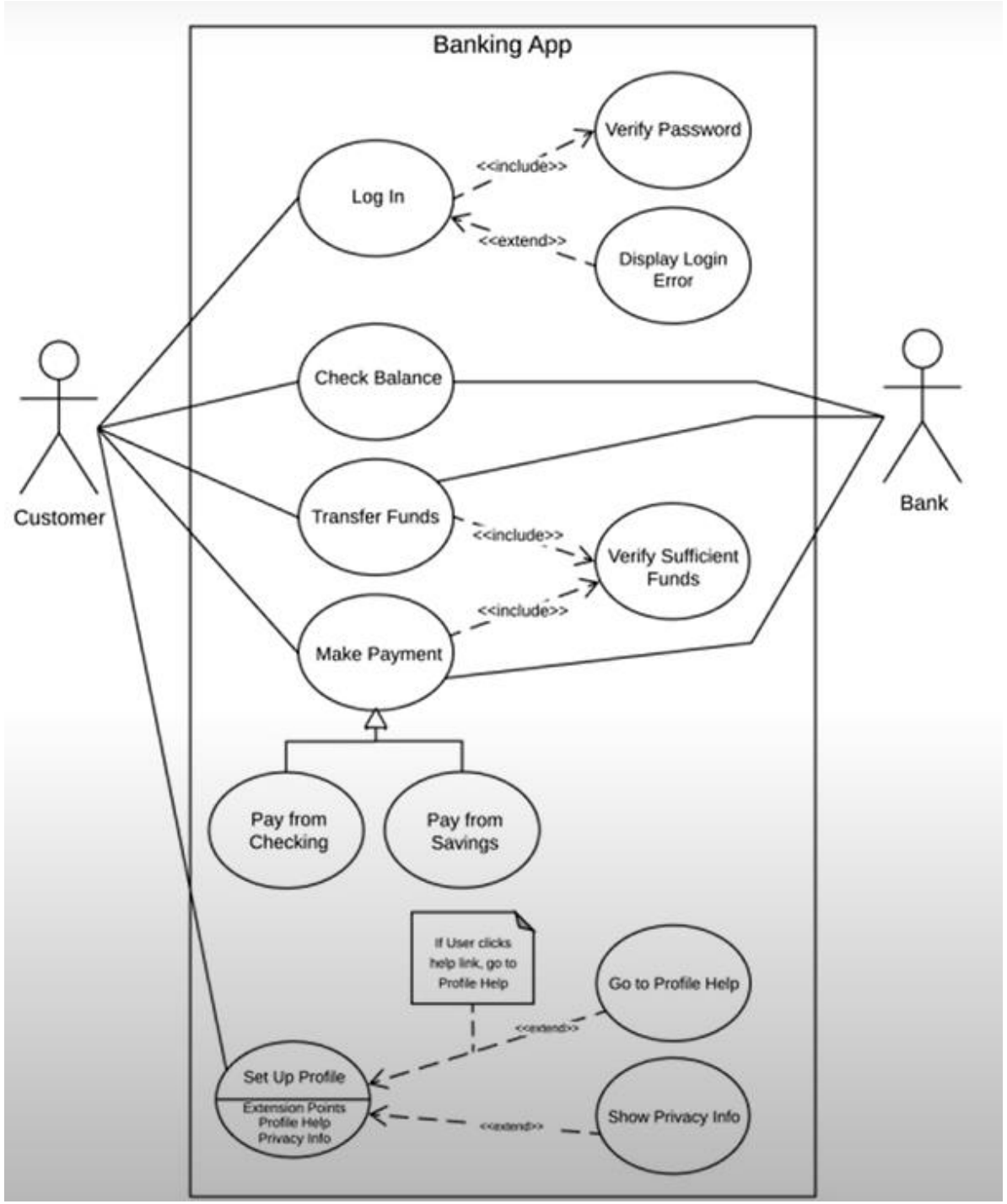
Exclude
Include

Generalization

UML

Example:

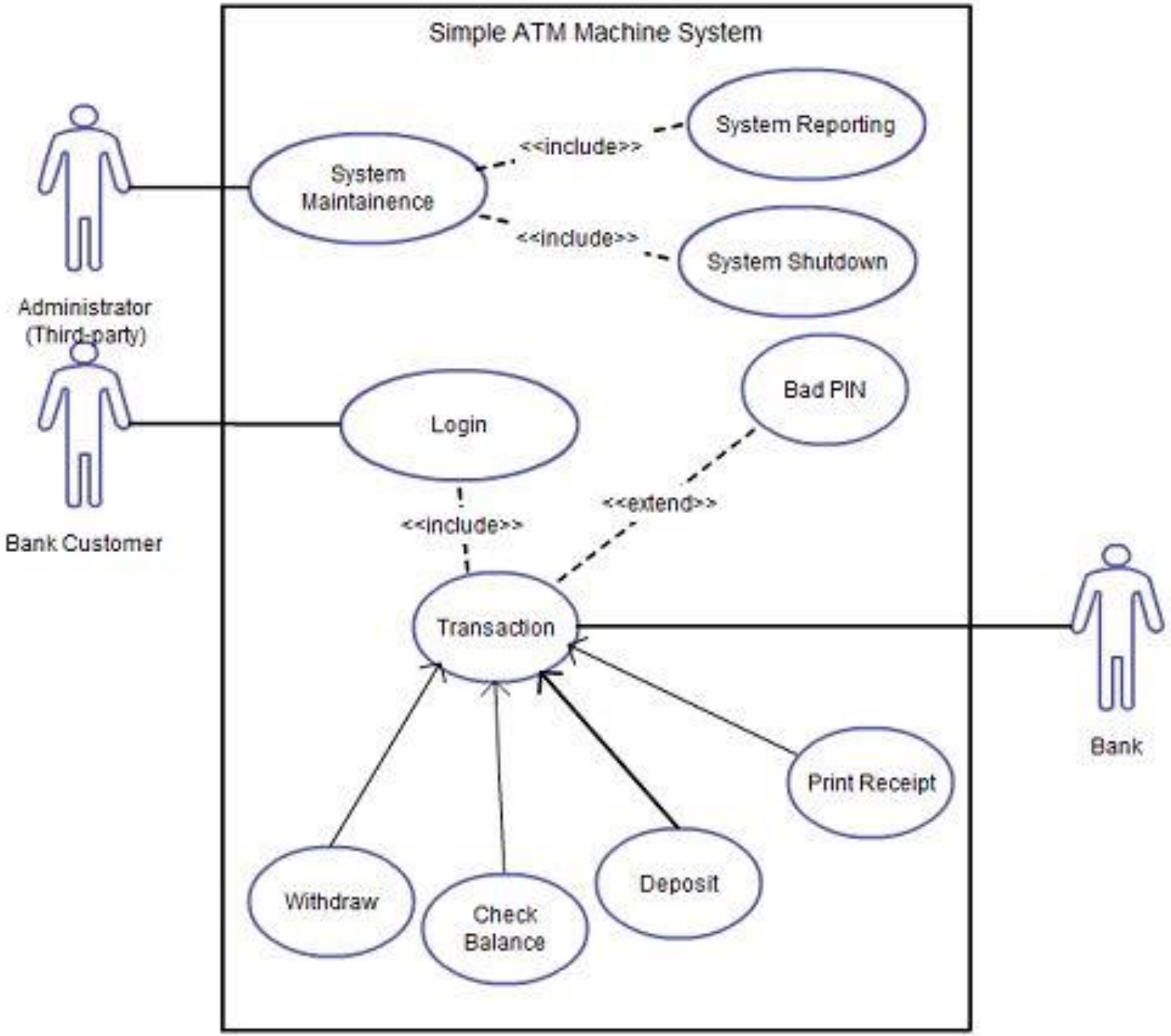
Banking Application



UML

Example:

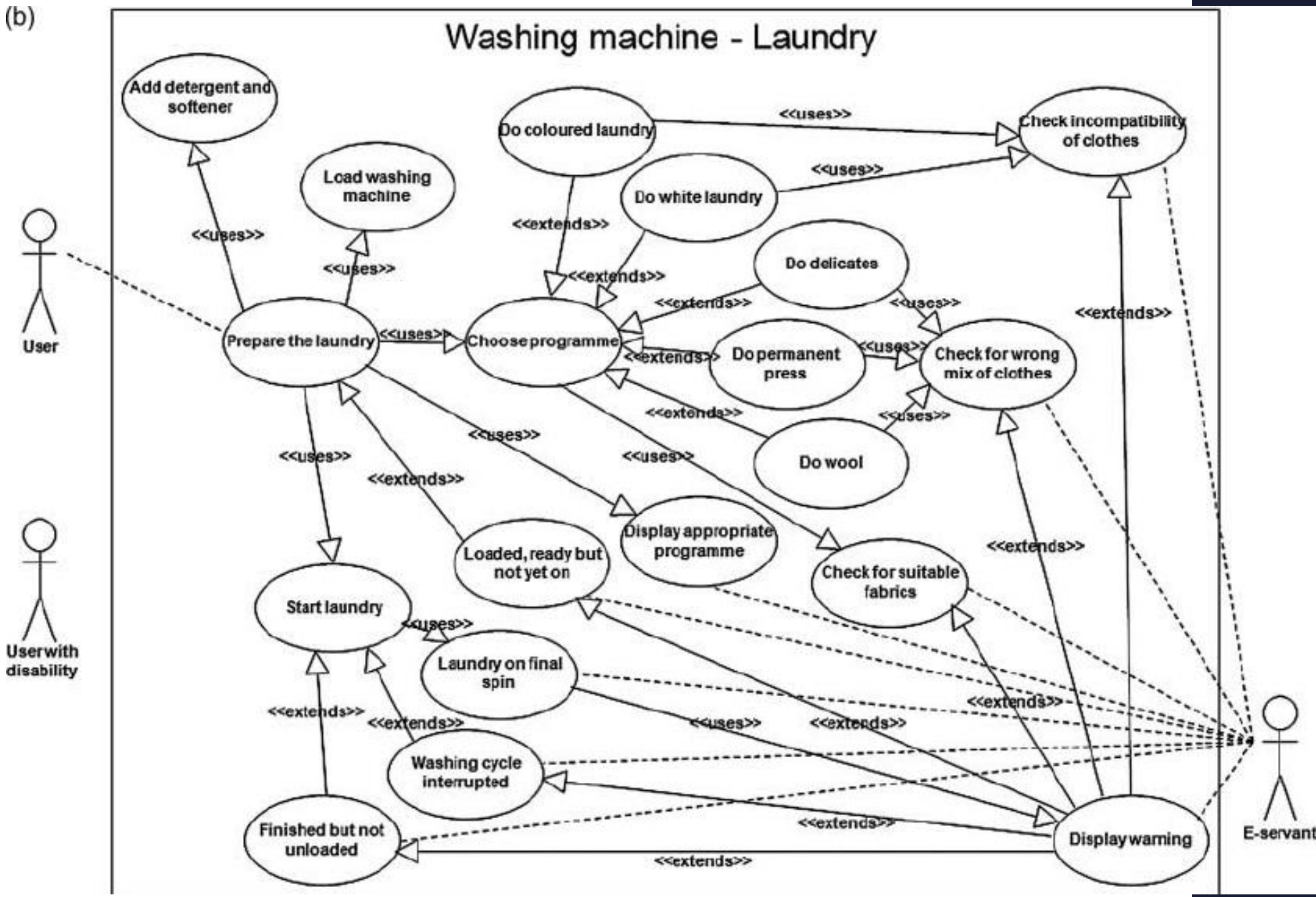
ATM System



UML

Example:

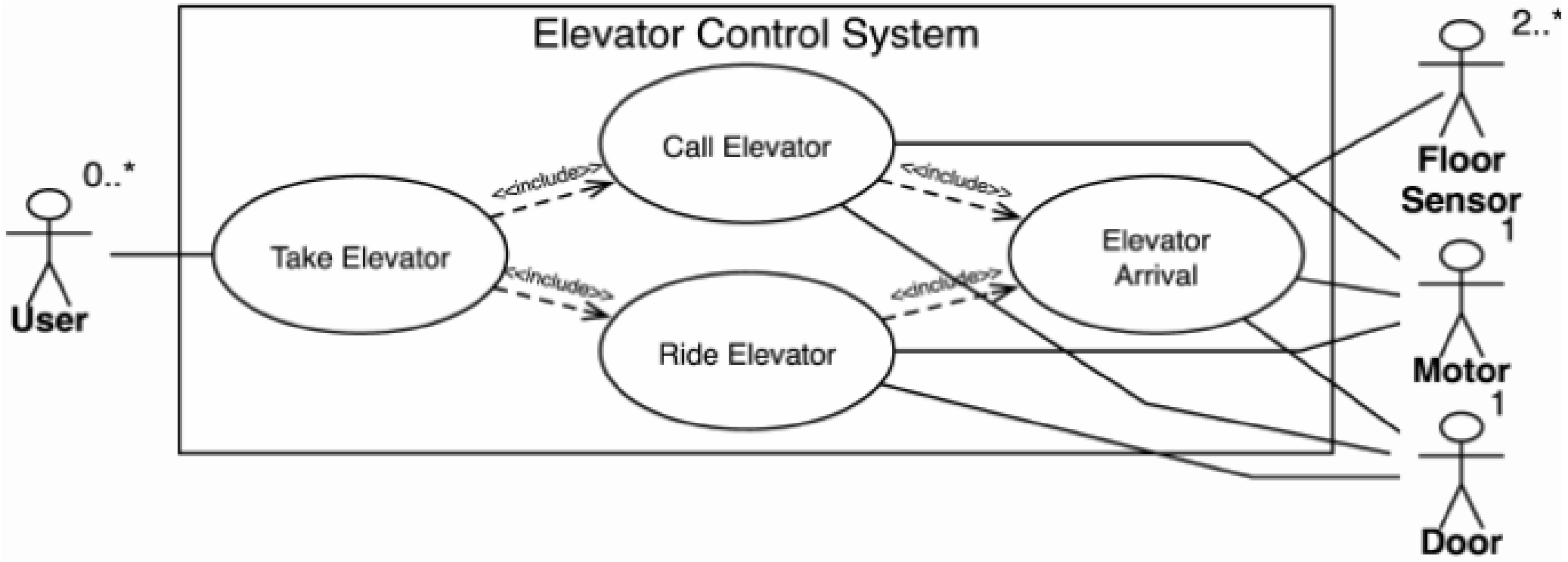
Washing Machine



UML

Example:

ELEVATOR SYSTEM



Class **Diagram**

Class Diagram

- Used to **document** ***software architecture***
- It is used to **refine** the **use case diagram**
- **Set of interrelated classes**
- It contains ***methods*** and ***attributes*** for ***classes***
- Classes can be “**is-a**” or "**has-a**" relationship
- ***Class diagram***
  - ***Static*** diagram, ***Structural*** diagram
  - ***Attributes*** and ***operations*** of a class
  - Explains the ***constraints*** on the system
  - **Collection** of *classes, interfaces, associations, collaborations and constraints*

Class Diagram

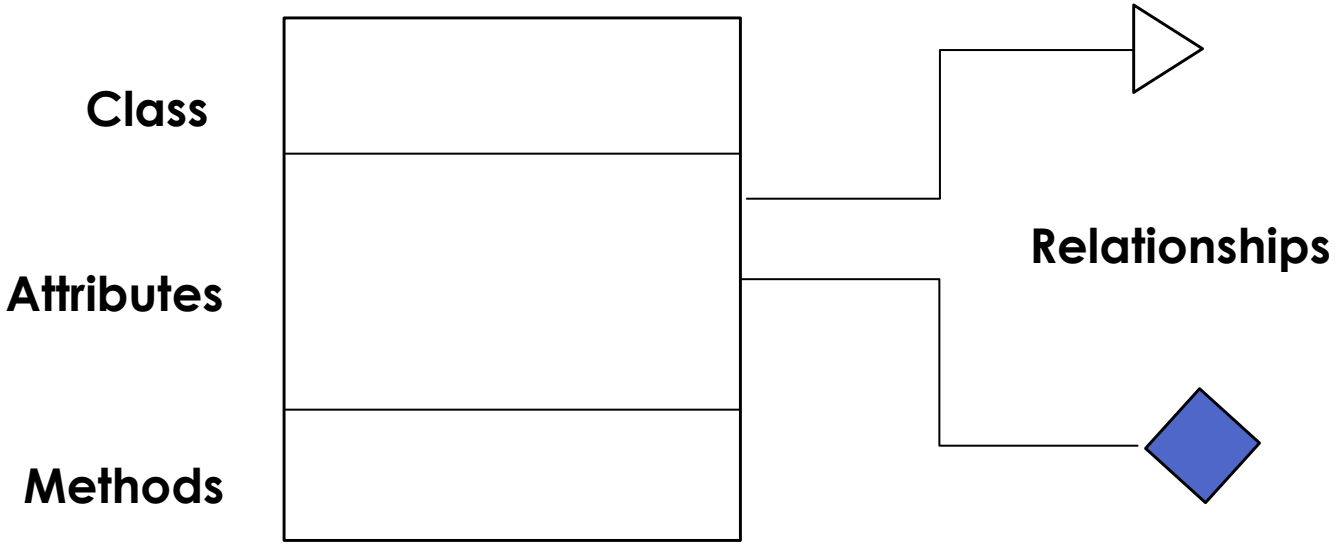
- **Purpose**
  - ***Analysis*** and ***design*** of system
  - Describe ***responsibilities*** of a system
  - ***Conceptual modelling***
  - ***Forward*** and ***reverse engineering***

Class Diagram

- **Classes** are used to **represent** **objects**

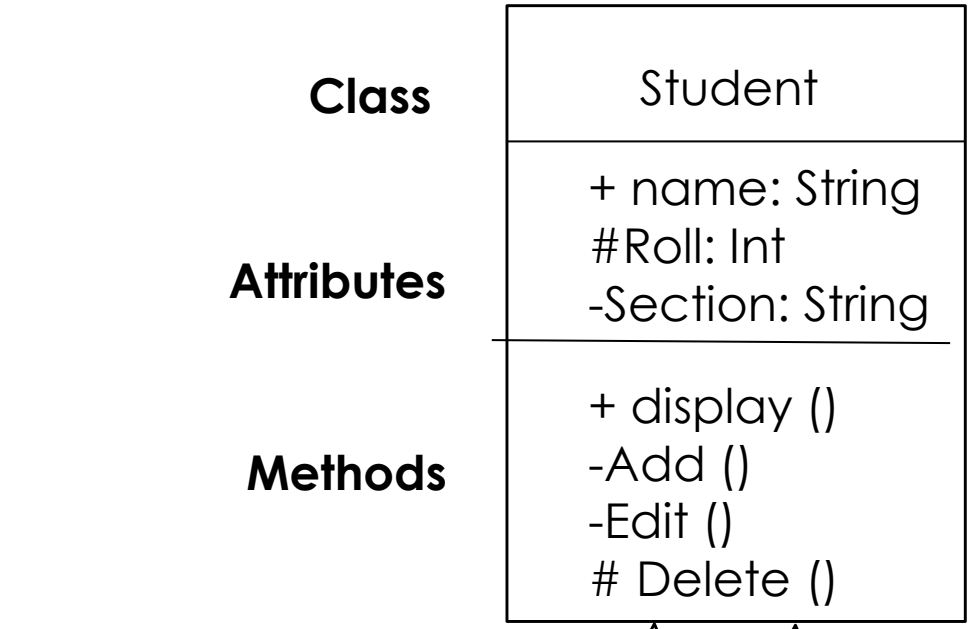
- *UML class diagram is divided into four parts.*
  - ***Top section*** - Name the class.
  - ***Second*** - Attributes of the class.
  - ***Third*** - Operations or methods performed by class.
  - ***Fourth*** - Any additional components.

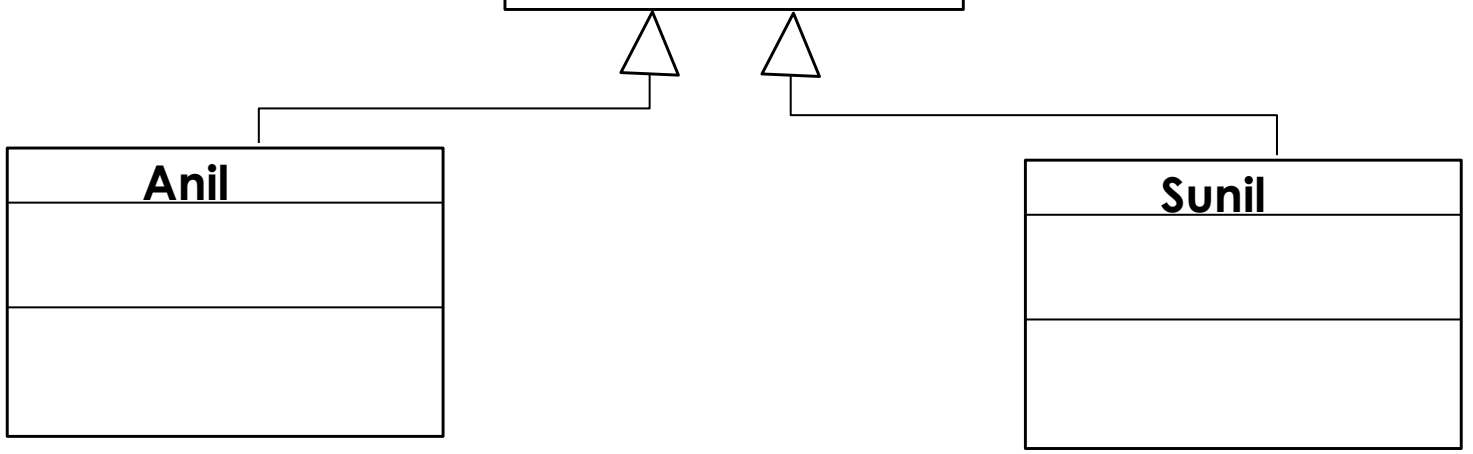
Class Diagram



Class Diagram

Student Information-Class Diagram





Visibility

+ Public
- Private
protected
~ Package/default

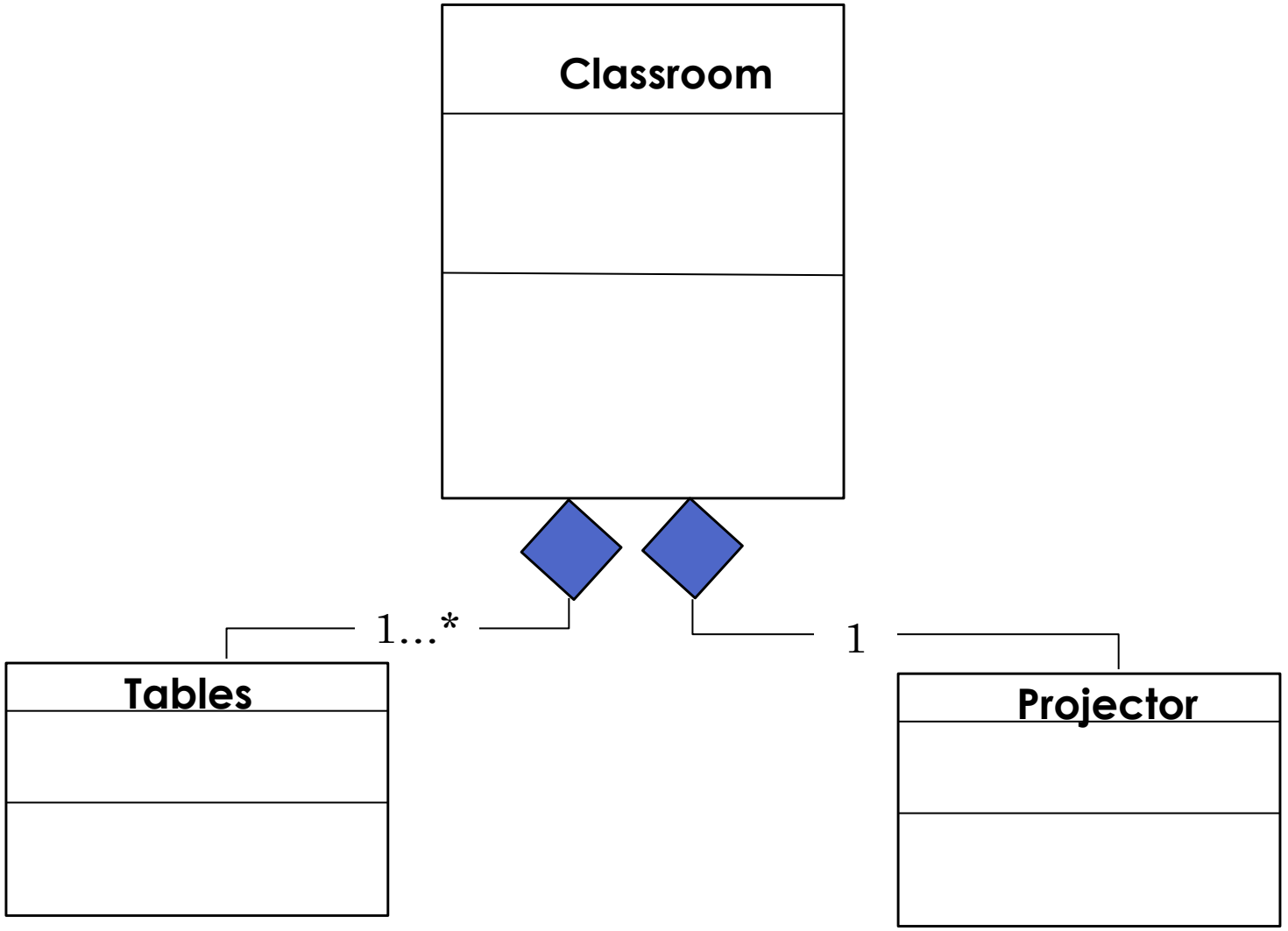
Inheritance

Association

Aggregation

Class Diagram

Student Information-Class Diagram

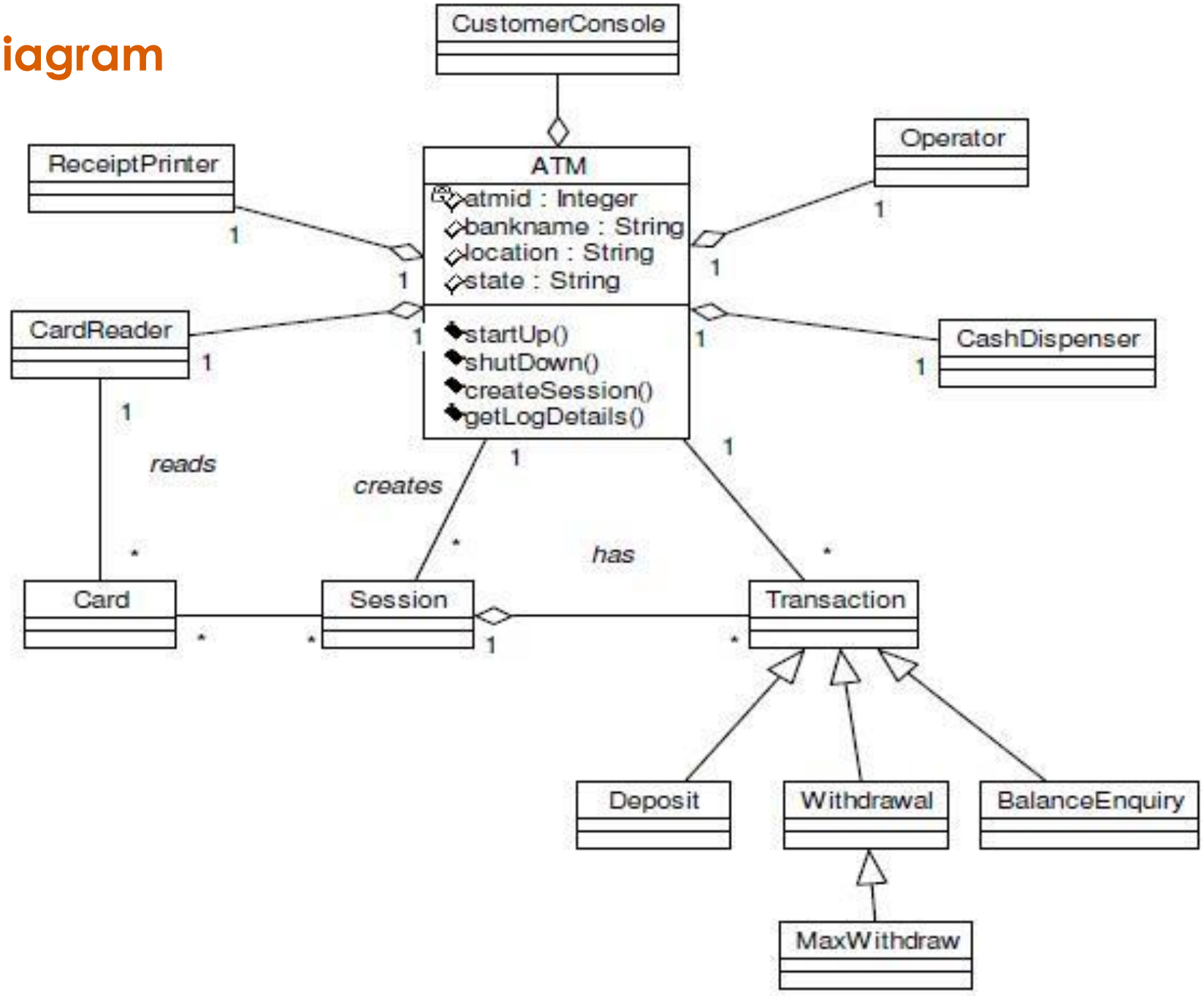


Multiplicity

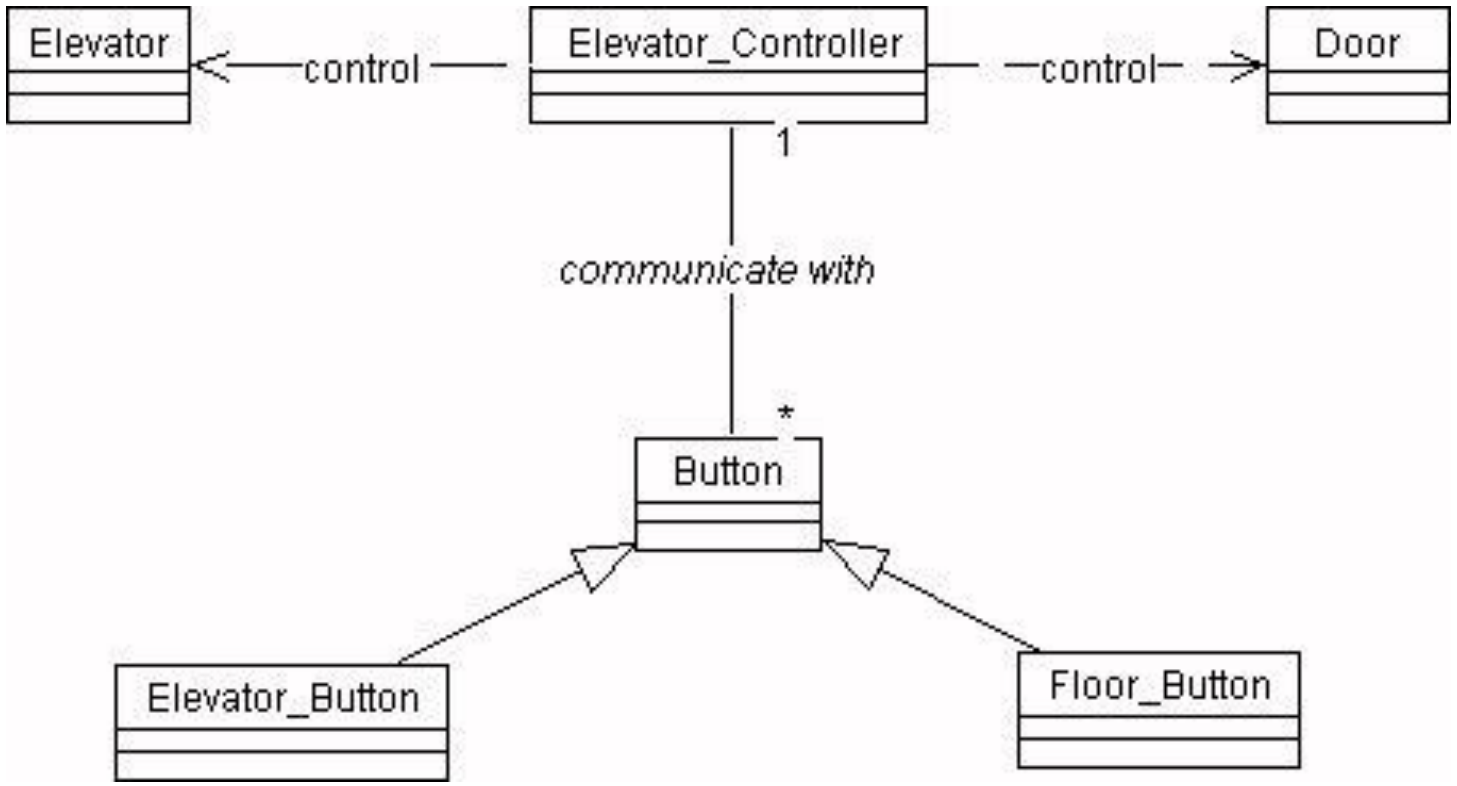
N
0…*
0…1
1…*
m…n

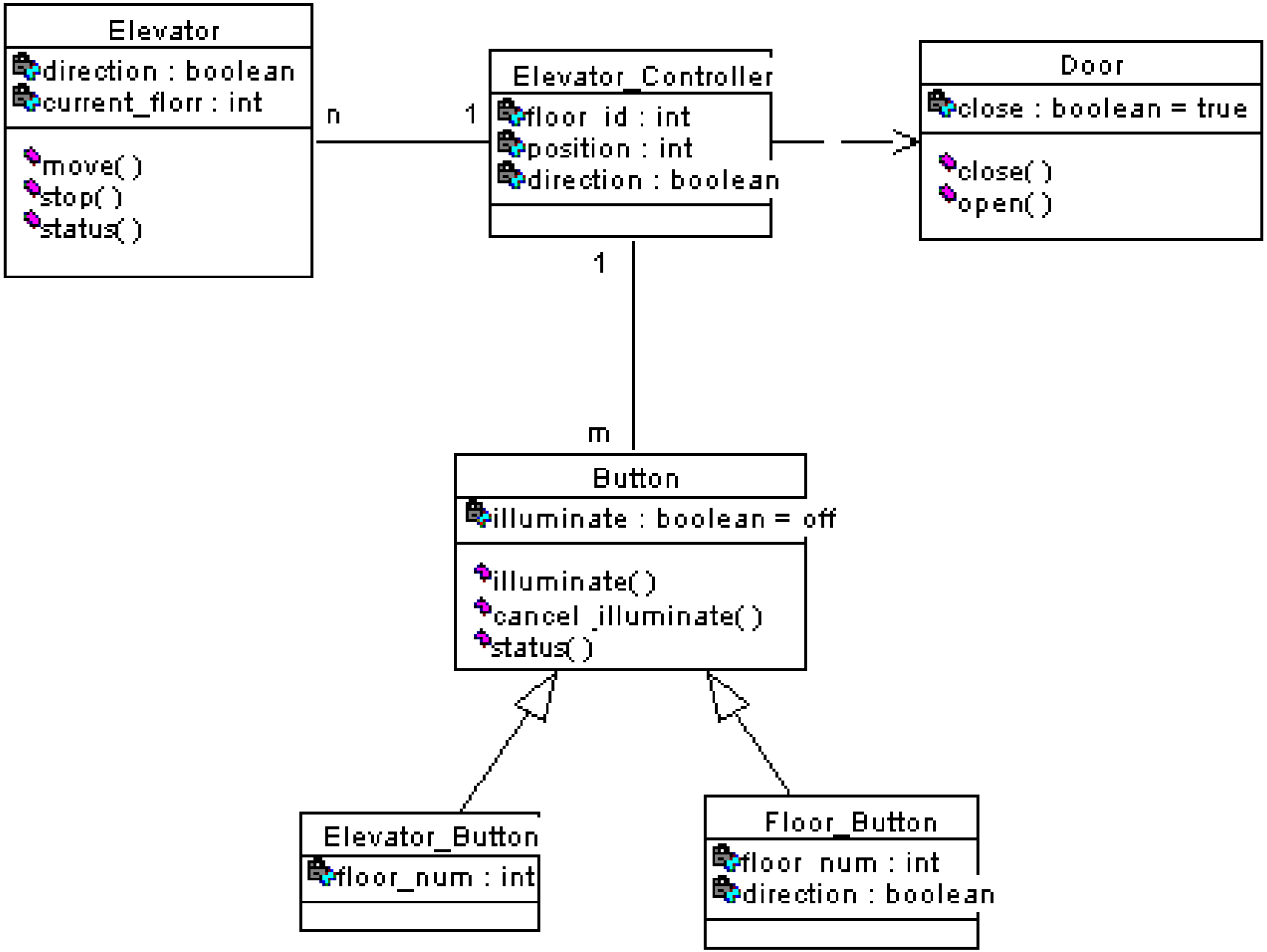
Composition

ATM - Diagram

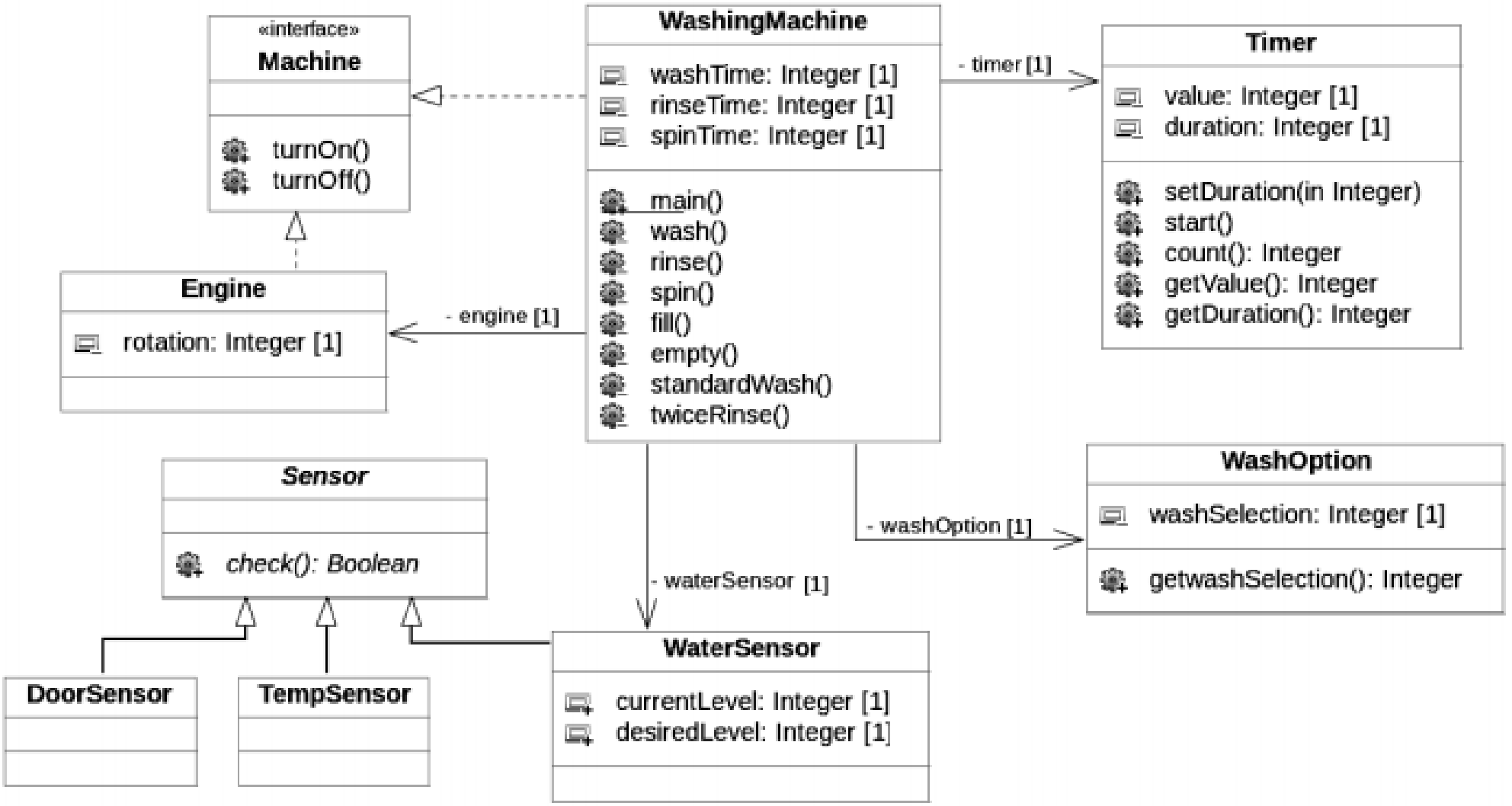

CustomerConsole
ReceiptPrinter
ATM
atmid : Integer
bankname : String
location : String
state : String
startUp()
shutDown()
createSession()
getLogDetails()
Operator
CardReader
CashDispenser
reads
creates
has
Card
Session
Transaction
Deposit
Withdrawal
BalanceEnquiry
MaxWithdraw

Elevator – Class Diagram




Elevator
direction : boolean
current_florr : int
move( )
stop( )
status( )
n
1
Elevator_Controller
floor_id : int
position : int
direction : boolean
Door
close : boolean = true
close( )
open( )
1
m
Button
illuminate : boolean = off
illuminate( )
cancel_illuminate( )
status( )
Elevator_Button
floor_num : int
Floor_Button
floor_num : int
direction : boolean

Class Diagram Example Washing Machine



Sequence **Diagram**

Sequence Diagram

- ***Interactions among*** the ***different elements*** in the ***model***
- Describes the **dynamic behavior** of the system
- Explain the **interaction between** ***objects*** & ***collaborations***
- It emphasizes on the **time sequence of messages** and **structural organizations** of the **objects**

Sequence Diagram

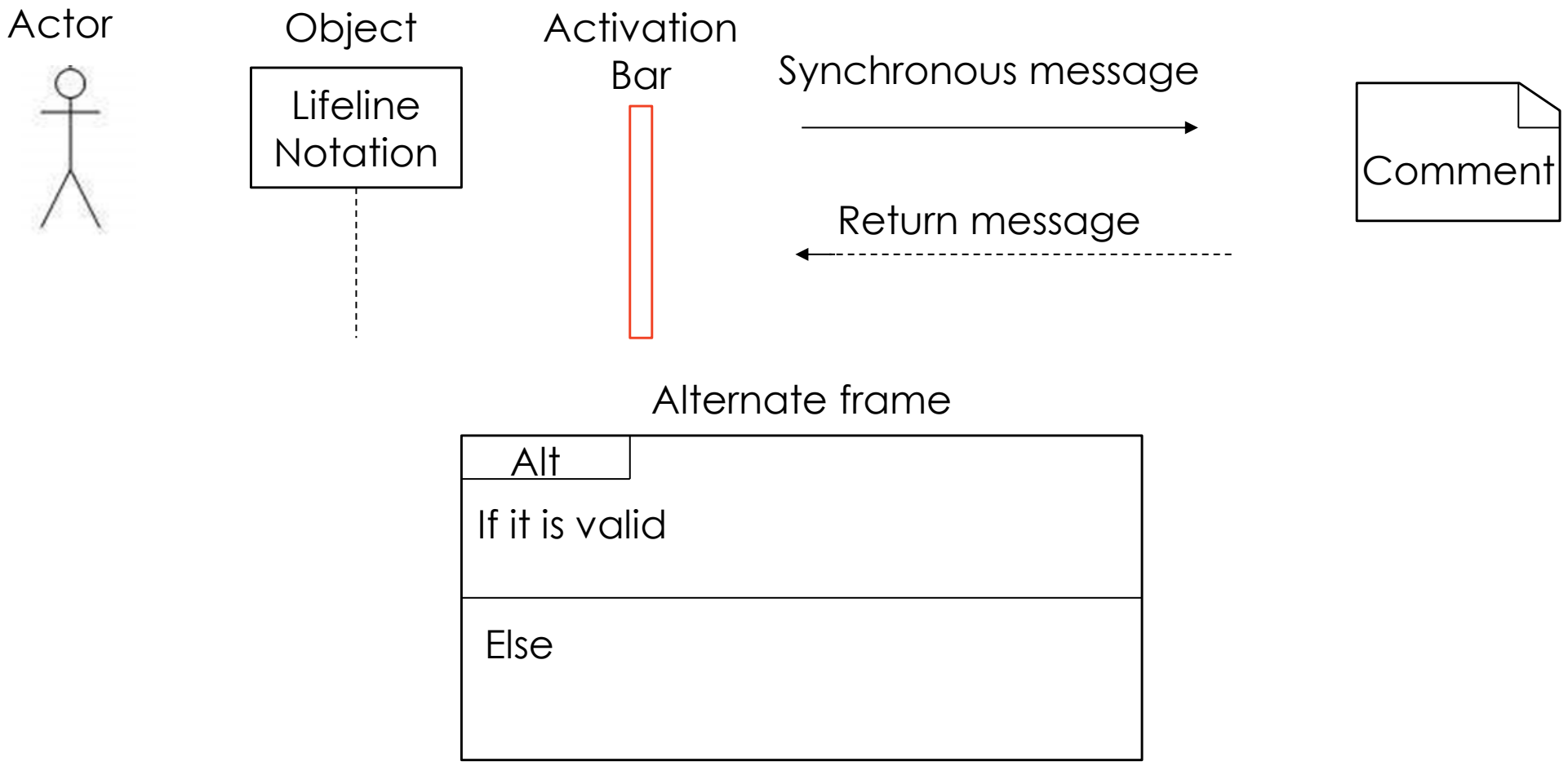
- Shows the details of the use cases
- Shows the complete flow of information, function and operations of the system
- Used in *planning* and *understanding* the *functionality* of existing and future scenario

- Time – vertical direction
- Header elements – Horizontal direction

Sequence Diagram

- *Are suitable for following scenarios:*
  - **Usage scenario** – *to determine how the **system could be used***
  - **Method logic** – *to explore **logic of any function**, **procedure** or **complex process***
  - **Service logic** – *ideal way to **explain high level methods used** by **clients***

Sequence Diagram



Sequence Diagram

- ***EXAMPLE***
  PRODUCT PURCHASE SYSTEM
- **In this sequence diagram we have four objects**
  - **Customer**
  - **Product**
  - **Stock**
  - **Payment**

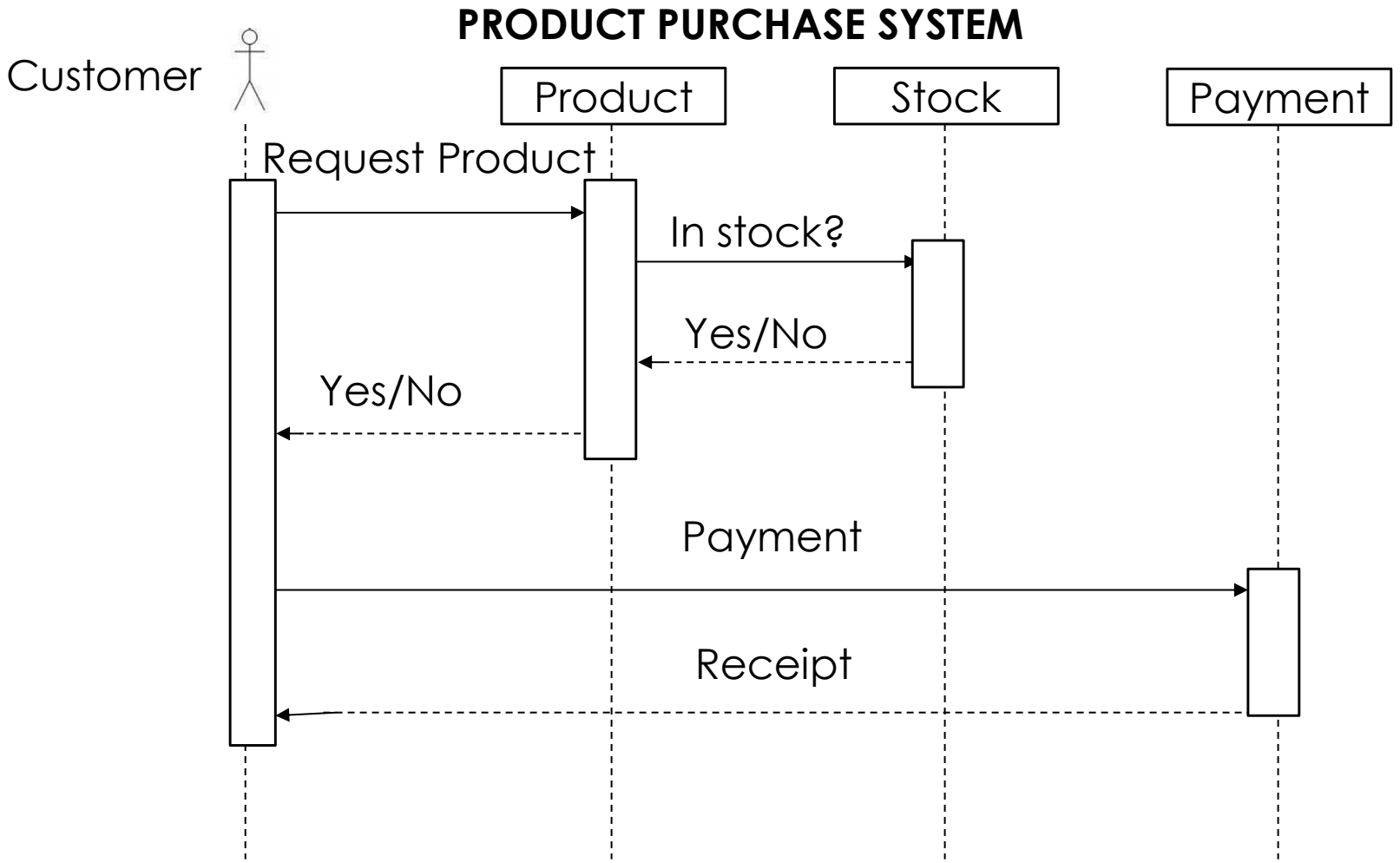
- **Message starts from the top and flows to the bottom (waterfall manner)**
- ***Dashed lines*** **- duration for object**
- ***Horizontal rectangles*** **- activation of the object**
- ***Messages sent*** **- dark arrow and dark arrow head**
- ***Return message*** **- dotted arrow**

Sequence Diagram

- **EXAMPLE**
  PRODUCT PURCHASE SYSTEM

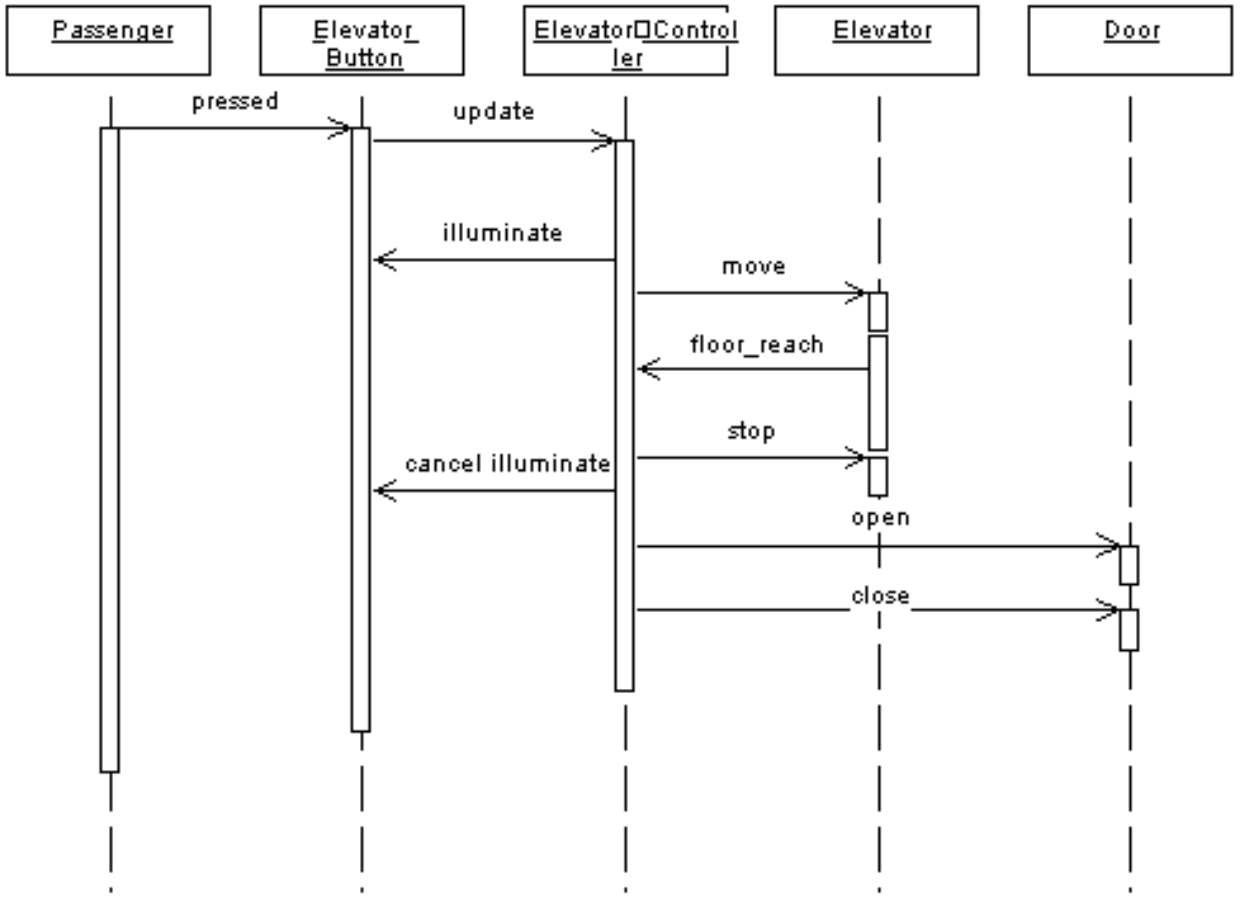
Customer object	**- to the product object**
Product object	**- to the stock object**
Stock object	**- saying yes or No.**
Product object sends	**- customer object.**
Customer object	**- payment object to pay money.**
Payment object	**- receipt to the customer object.**

Sequence Diagram



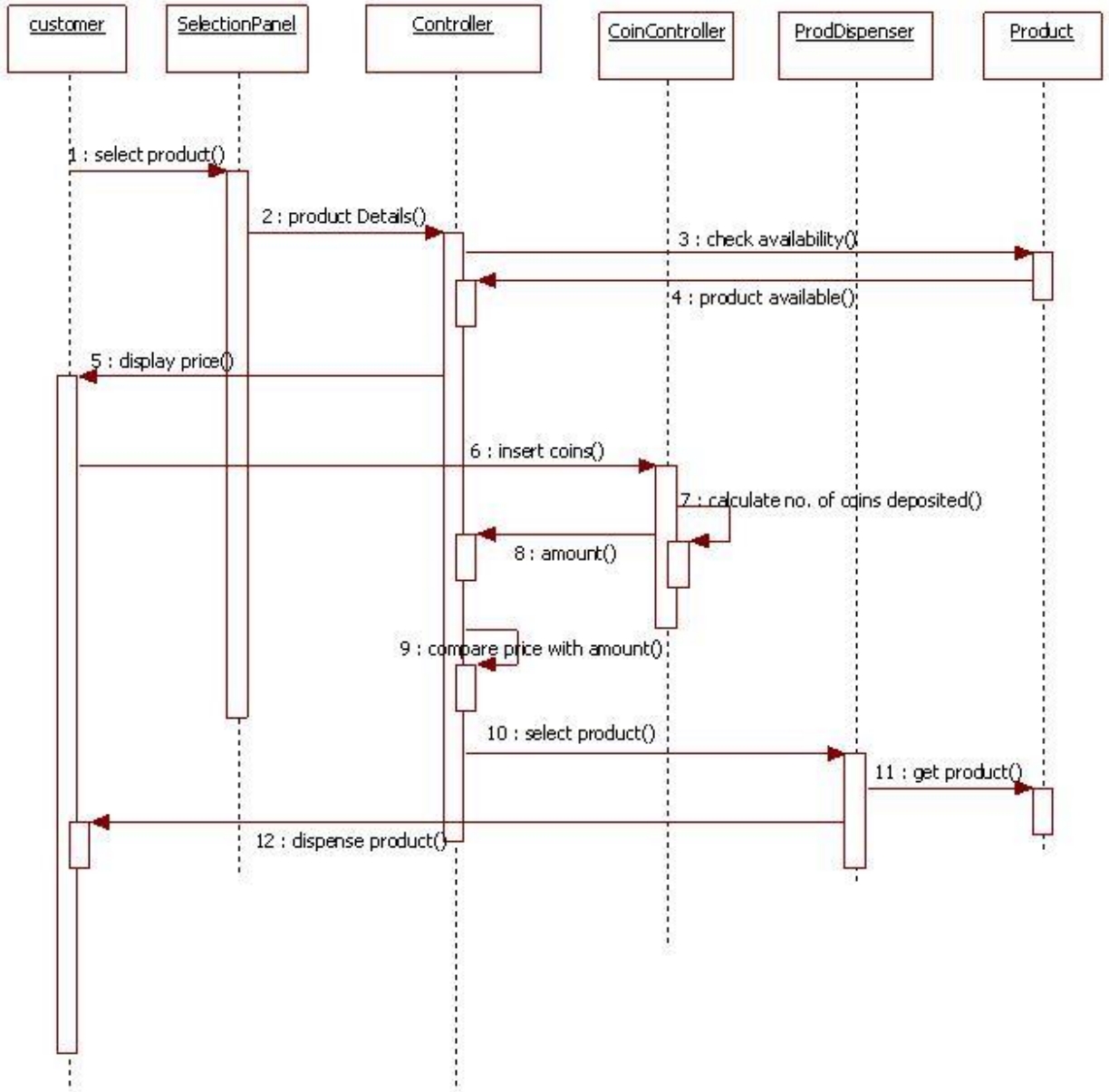
Sequence Diagram

SEQUENCE DIAGRAM EXAMPLE ELEVATOR SYSTEM



Sequence Diagram

Vending Machine



State **Diagram**

State Diagram

- State diagram **stores** the **status of object**
- **Shows the condition** of **system** over a **finite instance of time**
- It's a **behavioral diagram**
  - Referred to as ***State machines*** and ***State-chart Diagrams***.
- It explains the ***dynamic behavior*** of a ***class***
- **State diagrams** an be used to *understand* the *reaction* of *objects* on *receiving external stimuli*

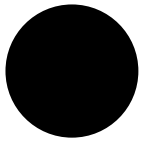
State Diagram

- **Uses of state chart diagram:**
  - We use it to depict event driven objects
  - We use it for showing use cases in business context
  - Shows the overall behavior of state machine
  - We use it to model the dynamic behavior of the system .

- **Steps to draw a state diagram:**
  - Identify the initial state and the final states.
  - Identify the possible states in which the object can exist
  - Label the events which trigger these transitions.

State Diagram

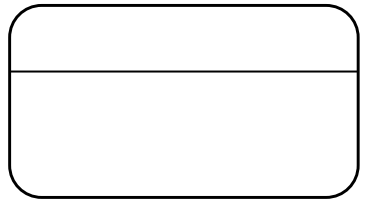
STATE DIAGRAM SYMBOL NOTATIONS



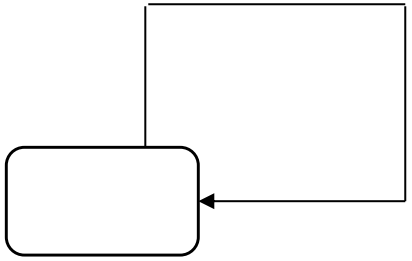
Initial state

Transition Symbol

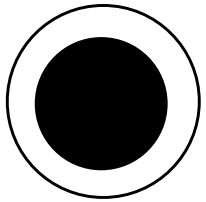
State



Composite State



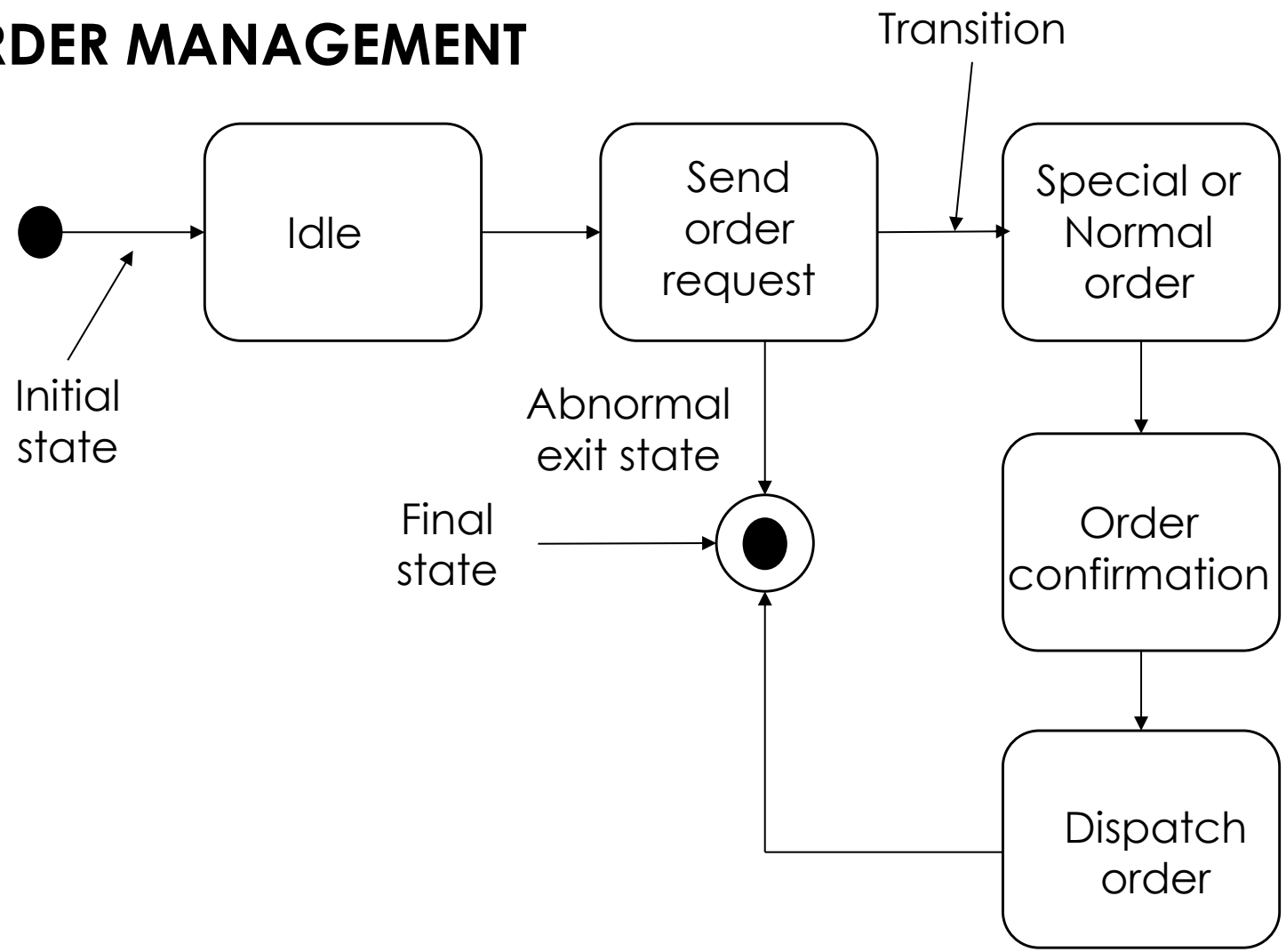
Self Transition



Final state

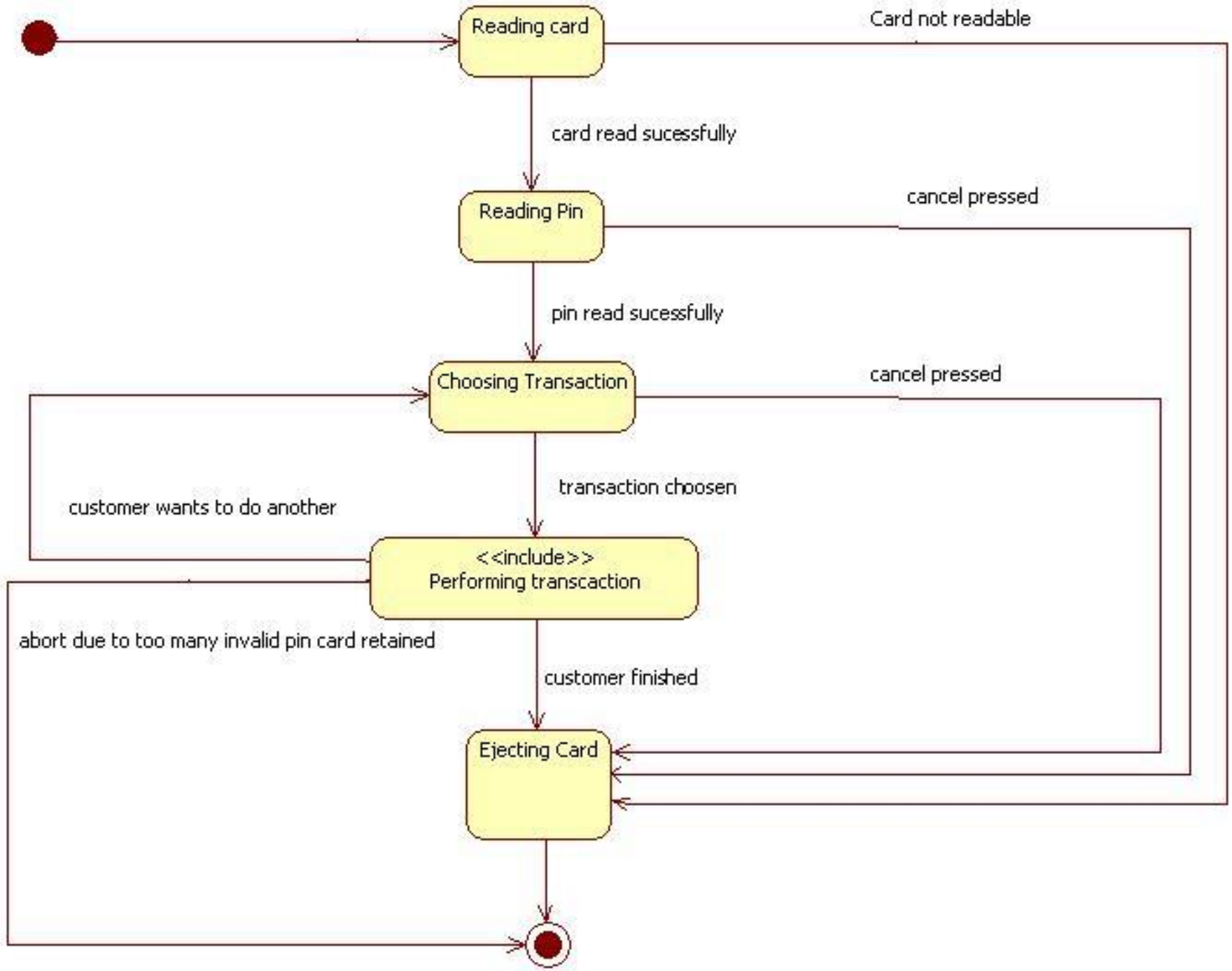
State Diagram

ONLINE ORDER MANAGEMENT



State Diagram

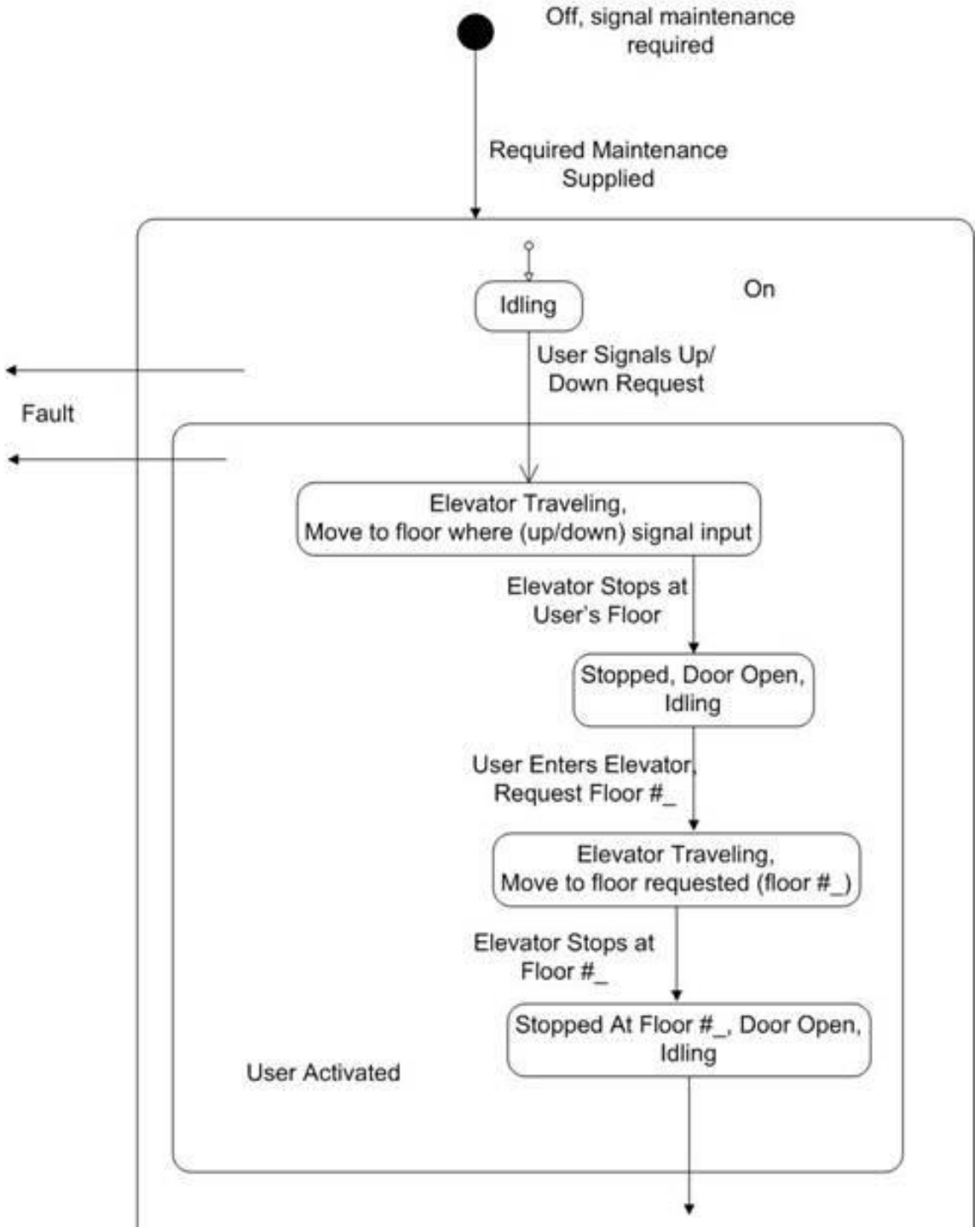
- **ATM machine**



State Diagram

- **Elevator System**

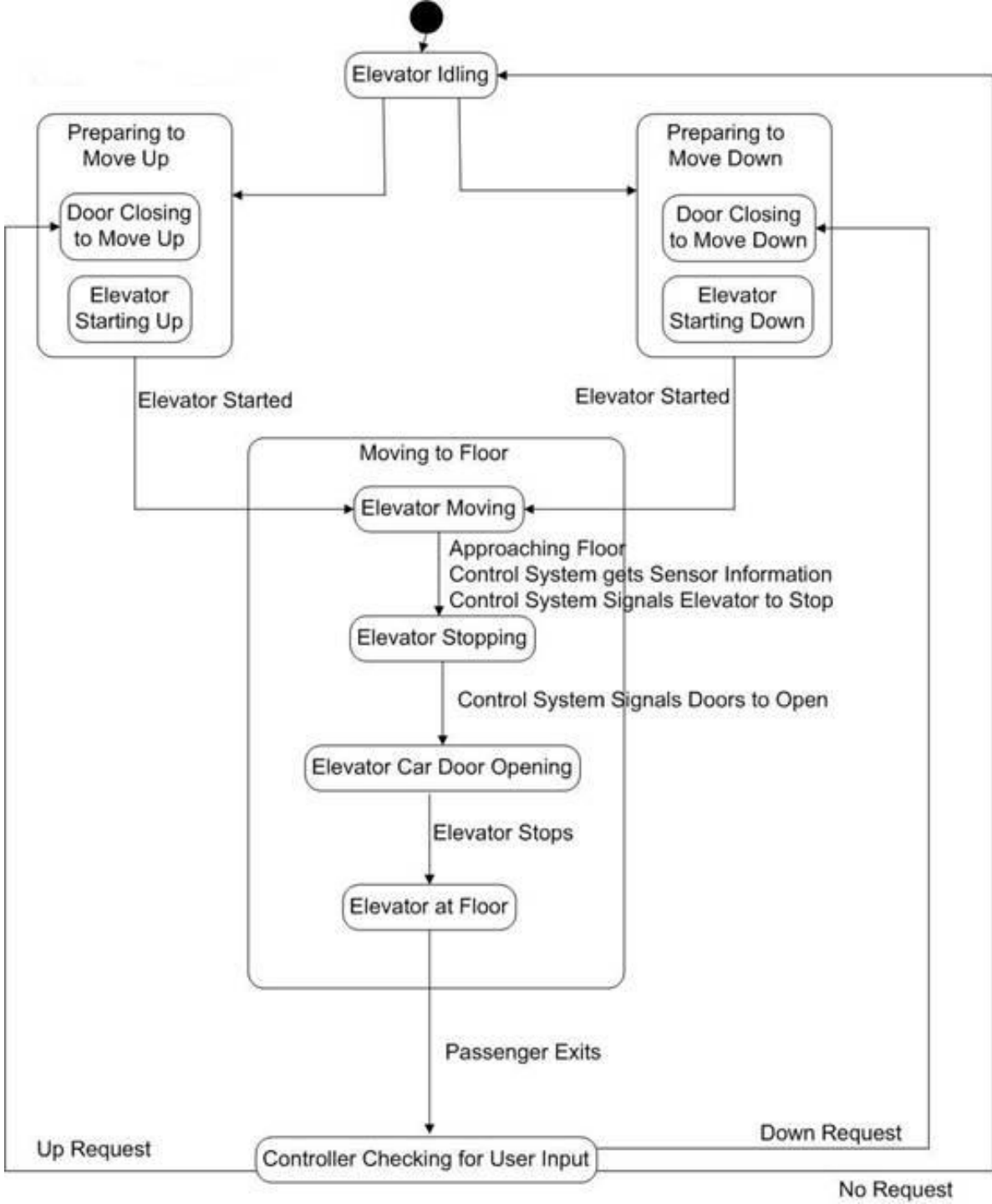
Statechart for "User" Behavior



State Diagram

- **Elevator System**

Statechart for “Elevator Control System"



Activity **Diagram**

Activity Diagram

- **Behavioral diagram**

- Shows the **control flow** from **start to finish point** ***including other decisions paths*** that is being executed

- It can **explain** both ***sequential*** and ***concurrent*** ***processing***

Activity Diagram

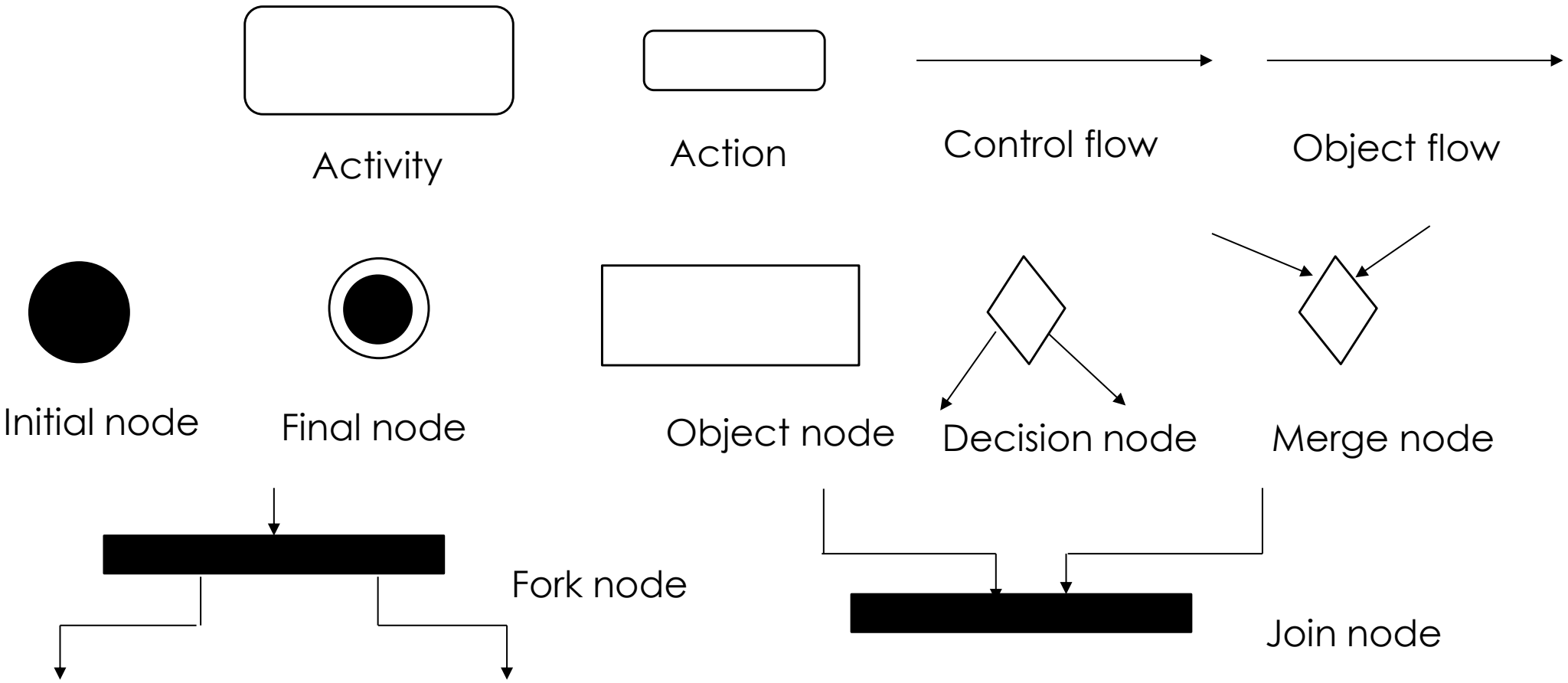
- Uses of Activity chart diagram:
  - ***Modelling work flow*** by ***using activities***
  - Modelling the ***logic*** of ***algorithm***
  - Shows the ***workflow process*** between ***users*** and ***system***
  - Understanding ***system's functionalities***.
  - Investigating business requirements at a later stage.

Activity Diagram

- Steps to draw a activity diagram:
  - **Identify** the ***initial*** state and the ***final states***.
  - **Identify** the ***intermediate activities*** needed to ***reach*** the ***final state*** from the initial state.
  - **Identify** the ***conditions*** or ***constraints*** which cause the ***system*** to ***change control flow***.
  - Draw the diagram with appropriate notations.

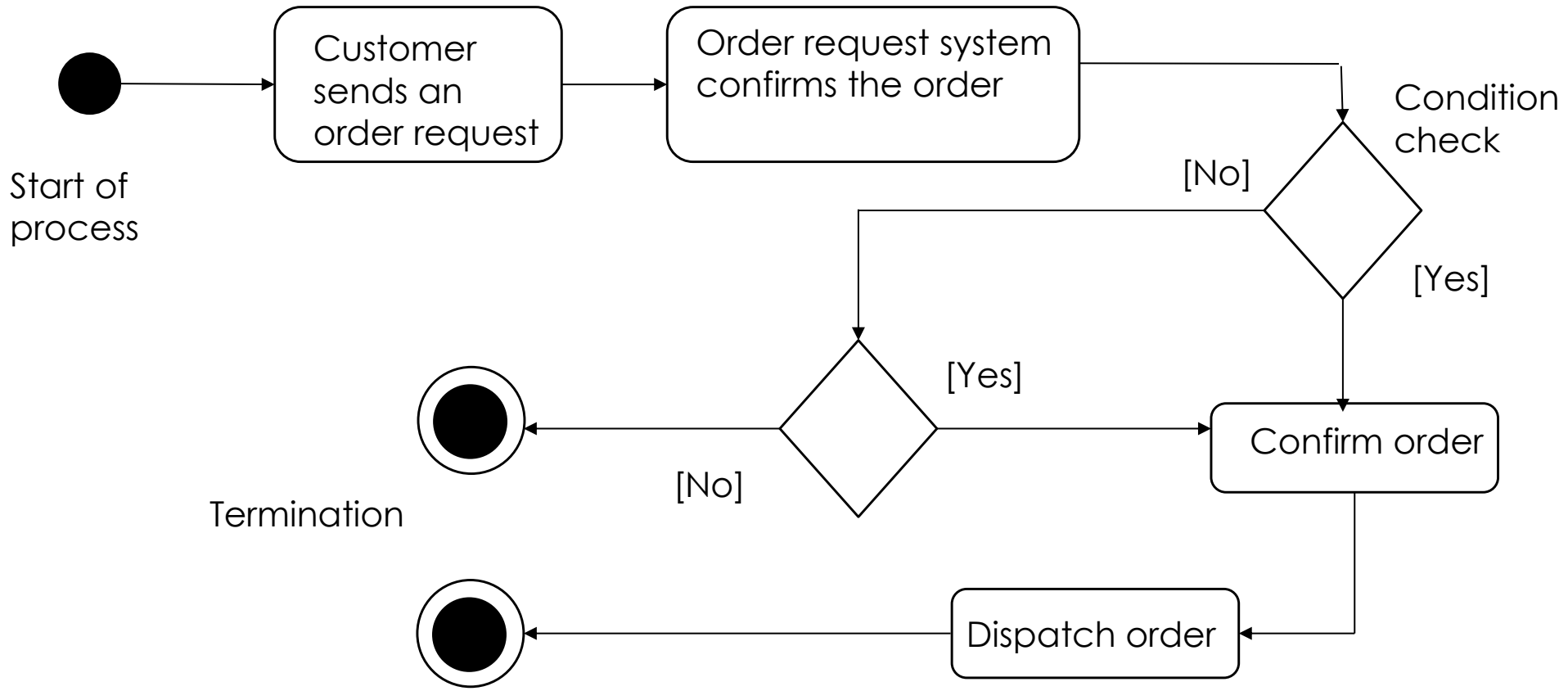
Activity Diagram

ACTIVITY DIAGRAM SYMBOL NOTATIONS



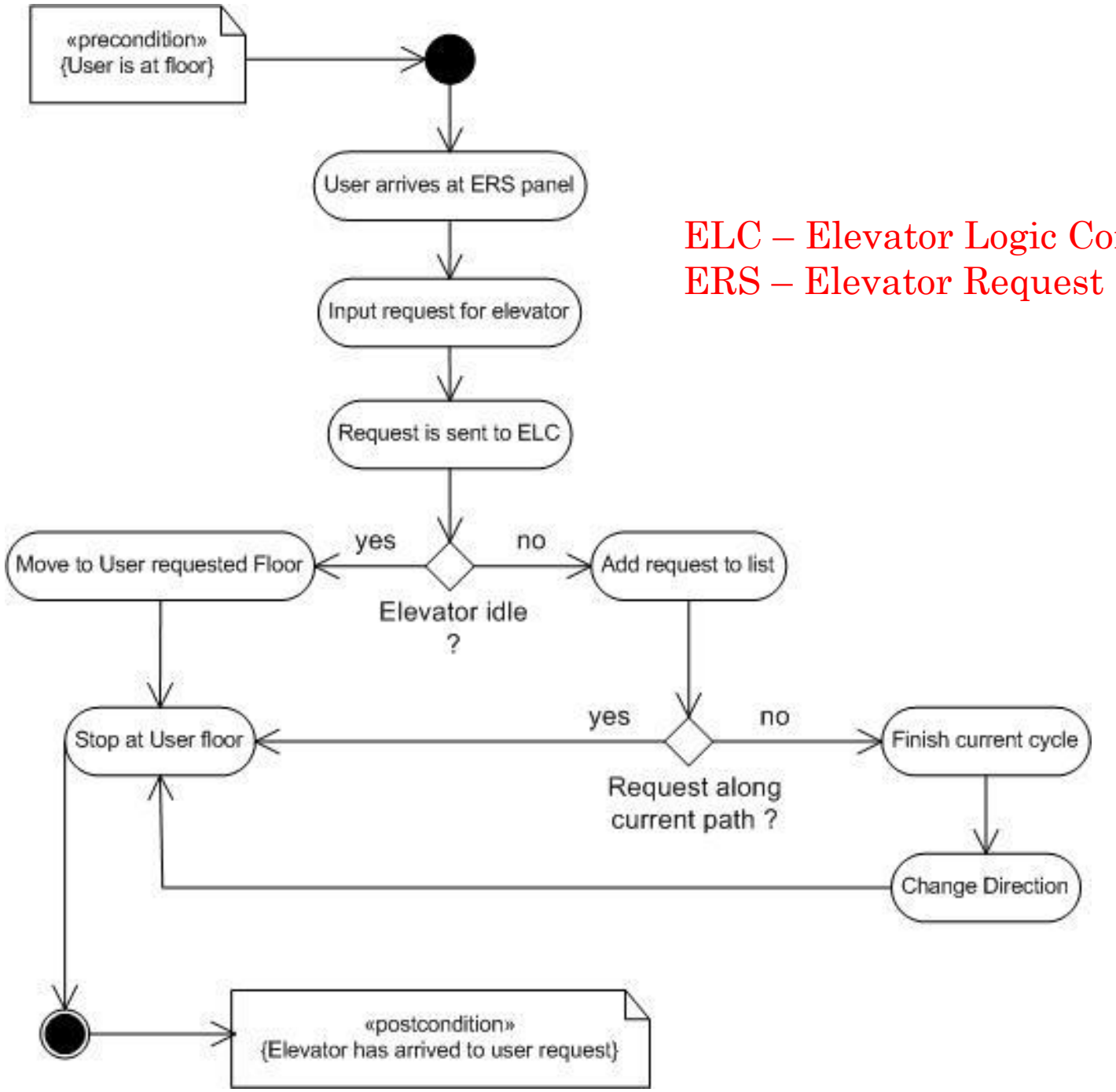
Activity Diagram

Activity Diagram of an Order Management System



Activity Diagram

Elevator



ELC – Elevator Logic Control
ERS – Elevator Request Service

Activity Diagram

Vending Machine