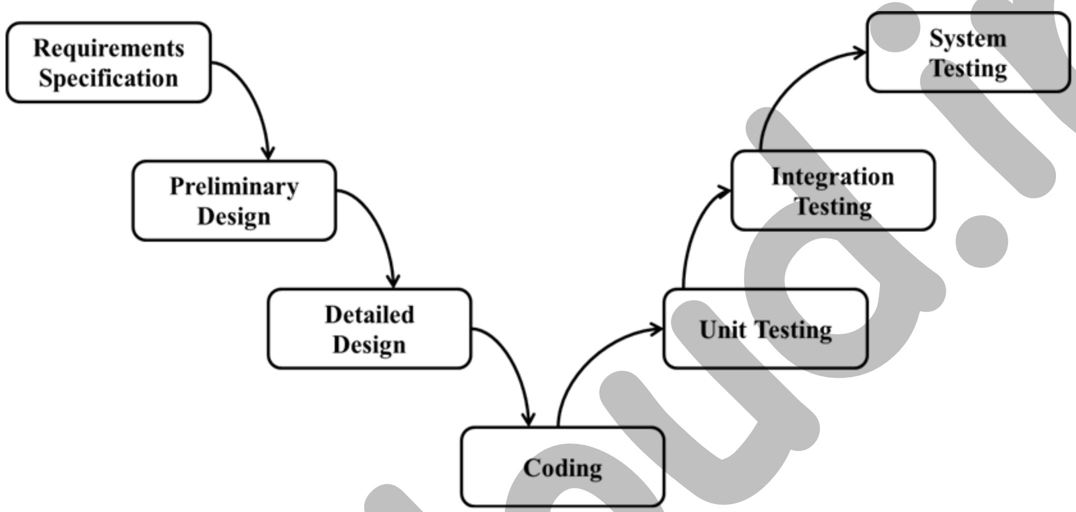
Chapter 12

Levels of Testing

➢ Traditional View of Testing Levels

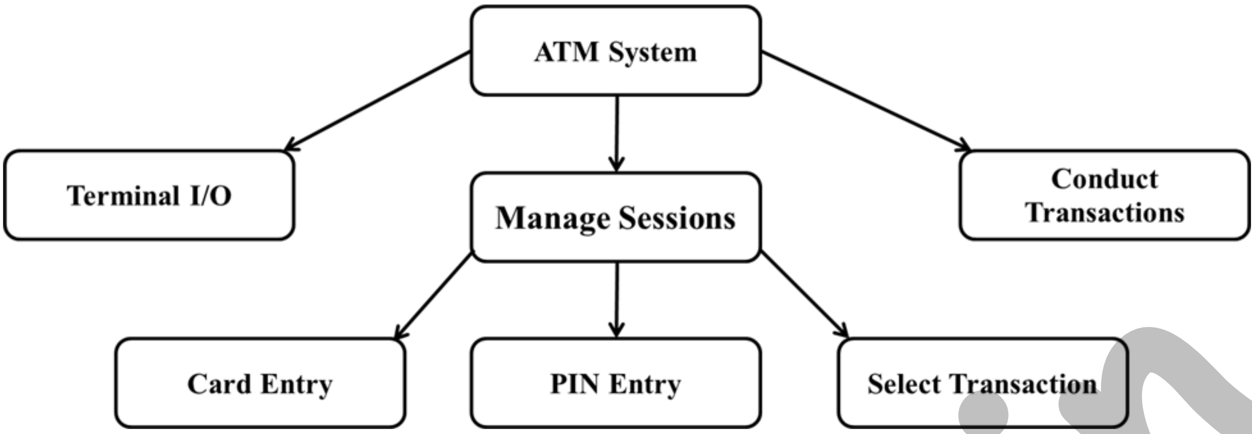
- The **traditional model** of software development is the **Waterfall model**, which is drawn as a **V** in **below Figure** to emphasize the basic levels of testing. In this view, **information produced** in one of the **development phases** constitutes the **basis** for **test case identification** at that level.

  Figure: The Waterfall Life Cycle



- The waterfall model is **closely associated** with **top-down development** and **design** by **functional decomposition**. The **end result** of preliminary design is a **functional decomposition** of the entire system into a **tree like structure** of functional components. The **below Figure** contains a **partial functional decomposition** of **ATM system**. With this decomposition, **top-down integration** would begin with the **main program**, **checking the calls** to the **three next level procedures (Terminal I/O, ManageSessions** and **ConductTransactions)**.
- Following the tree, the ManageSessions procedure would be tested, and then the CardEntry, PIN Entry, and SelectTransaction procedures. In each case, the actual code for lower level units is replaced by a **stub**, which is a **throwaway piece of code** that takes the place of the **actual code**.
- **Bottom-up integration** would be the **opposite sequence**, starting with the **CardEntry, PIN Entry and SelectTransaction** procedures and **working up toward** the **main program**.
- In **bottom-up integration**, **units** at **higher levels** are replaced by **drivers** (another form of throw-away code) that **emulate** the **procedure calls**.
- The **big bang approach** simply puts all the units **together** at once, with **no stubs** or **drivers**.
- Whichever approach is taken, the **goal** of **traditional integration testing** is to **integrate previously tested units** with respect to the **functional decomposition tree.**

Figure: Partial functional decomposition of our ATM system.



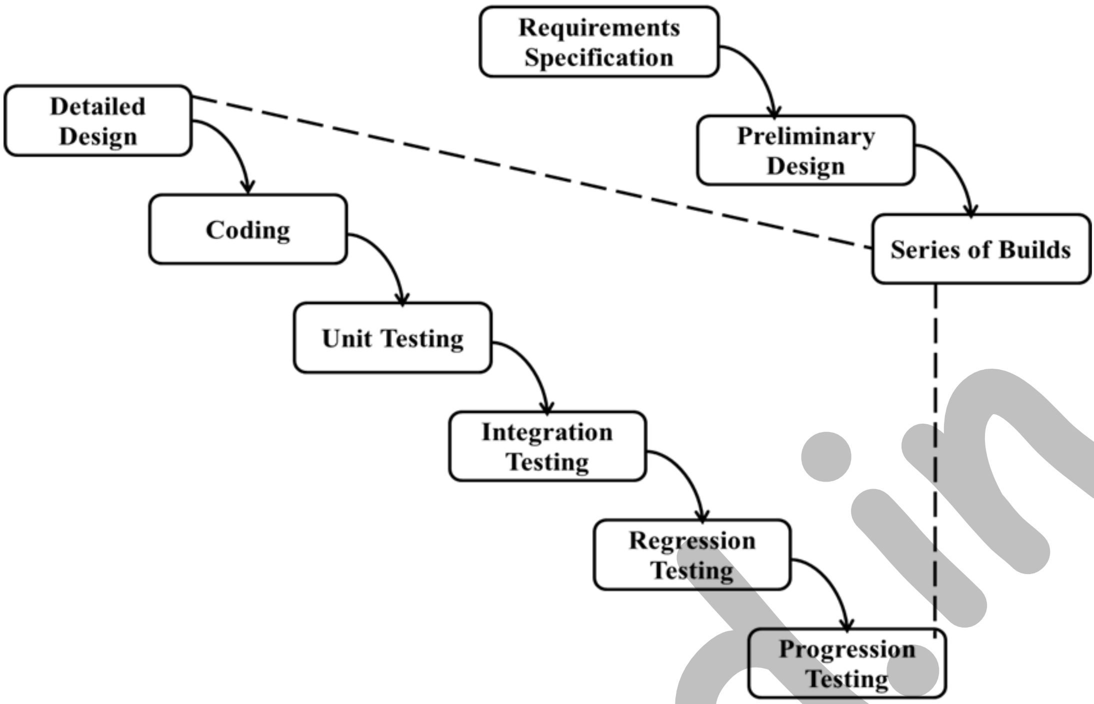
Alternative Life Cycle Models

- One of the major **weaknesses** of **waterfall development** is the over-reliance on this **whole paradigm. Functional decomposition** can only be well done when the system is completely understood.
- The **result** is a **very long separation** between **requirements specification** and a **completed system** and during this interval, there is **no opportunity** for **feedback** from the **customer**.
- A **composition starts** with something **known** and **understood**, then adds to it gradually and may remove undesired portions.

Waterfall Spin-offs

- There are **three mainline derivatives** of the **waterfall model: incremental development, evolutionary development** and the **Spiral model**. Each of these involves a **series** of **increments** or **builds,** as shown in **below Figure**.
- Within a build, the normal **waterfall phases** from **detailed design** through **testing** occur, with **one important difference**, **system testing** is split into **two steps, regression** and **progression** testing.
- It is important to keep **preliminary design** as an **integral phase.**
- The **goal** of **regression testing** is to **assure** that **things** that **worked correctly** in the **previous build still work** with the **newly added code**. **Progression testing** assumes that **regression testing was successful** and that the **new functionality** can be tested.
- The **differences** among the **three spin-off models** are due to **how** the **builds** are **Identified**.
- In **incremental development**, the **motivation** for **separate builds** is usually to **level off** the **staff profile**. With pure **waterfall development**, there can be a **huge bulge** of **personnel** for the phases from **detailed design** through **unit testing.** In **evolutionary development**, there is still the presumption of a build sequence, but only the first build is defined.

Figure: Life Cycle with a Build Sequence

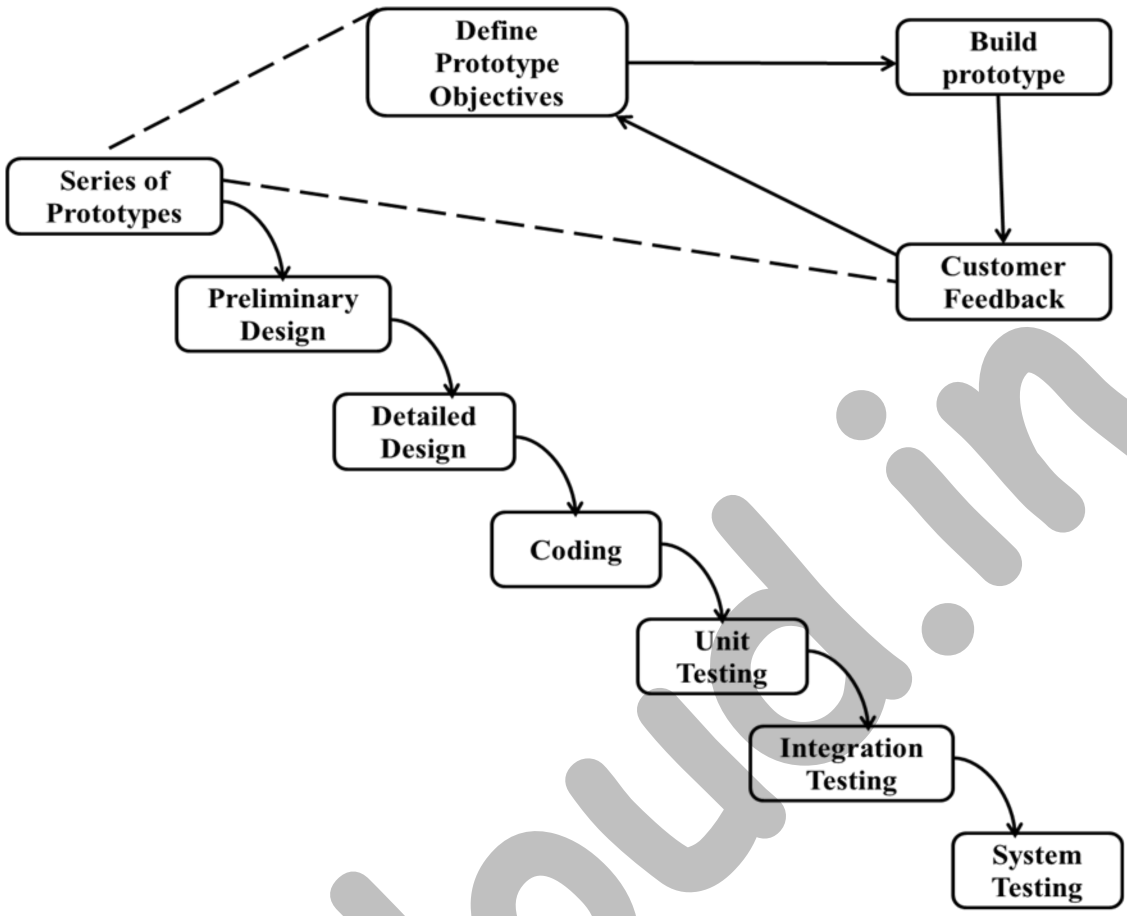


- Based on it, **later builds** are identified, usually in **response** to **priorities** set by the **Customer/user**, so the **system evolves** to meet the **changing needs** of the **user**.
- The **spiral model** is a combination of **rapid prototyping** and **evolutionary development,** in which a **build** is **defined first** in terms of **rapid prototyping**, and then is **subjected** to a **go/no-go decision** based on **technology-related risk factors**. From this we see that **keeping preliminary design** as **an integral step** is **difficult** for the **evolutionary** and **spiral models**. To the **extent** that this **cannot** be **maintained** as **an integral activity**, **integration testing** is **negatively affected.**
- Because a **build** is a **set** of **deliverable end-user functionality**, **one advantage** of these spin-off models is that **all three yield earlier synthesis.** This also **results** in **earlier customer feedback**, so **two** of the **deficiencies** of **waterfall development** are **mitigated**.

✓ Specification-Based Life Cycle Models

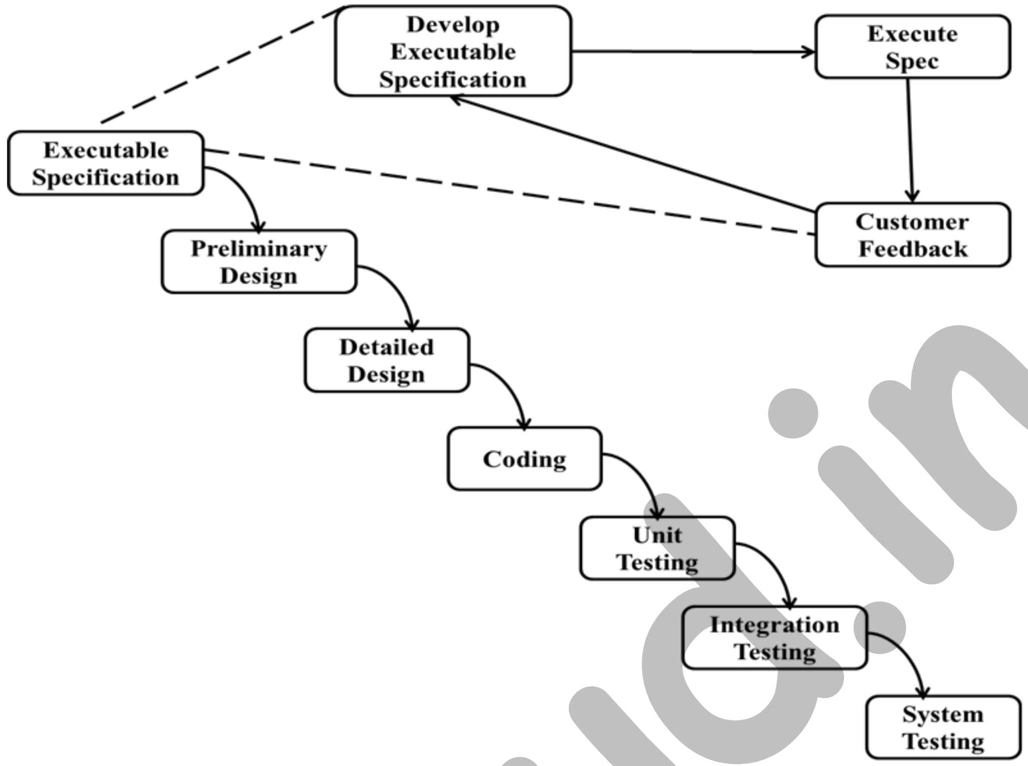
- **Two other variations** are **responses** to the **"complete understanding"** problem.
- When **systems** are **not fully understood** (by either the customer or the developer), **functional decomposition** is **perilous** at best. The **rapid prototyping life cycle** as shown in **below Figure** deals with this by **drastically reducing** the **specification-to-customer feedback loop** to **produce** very **early synthesis**.
- Rather than **build** a **final system**, a **"quick** and **dirty" prototype** is **built** and then **used** to **elicit customer feedback. Depending** on the **feedback**, **more prototyping cycles** may occur. Once the **developer** and the **customer agree** that a **prototype represents** the **desired system**, the **developer** goes ahead and **builds** to a **correct specification.**
- **Rapid prototyping** has **interesting implications** for **system testing**. **Where** are the **requirements?** Is the **last prototype** the **specification? How** are **system test cases traced back** to the **prototype**?

Figure: Rapid Prototyping Life Cycle



- **One good answer** to questions such as these is to **use** the **prototyping cycles** as **information gathering activities,** and then **produce** a **requirements specification** in a more **traditional manner**. **Another possibility** is to **capture what** the **customer does** with the **prototypes**, **define** these as **scenarios** that are **important** to the **customer**, and then use these as **system test cases**.
- The **main contribution** of **rapid prototyping** is that it **brings** the **operational (or behavioral)** viewpoint to the requirements specification phase. **Requirements specification techniques emphasize** the **structure** of a **system, not its behavior**.
- **Executable specifications** as shown in **below Figure** are an **extension** of the **rapid prototyping** concept. With **this approach**, the **requirements** are **specified** in **an executable format** (such as **finite state machines** or **Petri nets**). The customer then **executes** the **specification** to **observe** the **intended system behavior** and **provides feedback** as in the **rapid prototyping model**.
- **One big difference** is that the **requirements specification document** is **explicit**, as opposed to a prototype. **More important**, it is often a **mechanical process** to **derive system test cases** from an **executable specification**. **Another important distinction** is, when **system testing** is **based** on an **executable specification**, we have a **form** of **structural testing** at the **system level**.

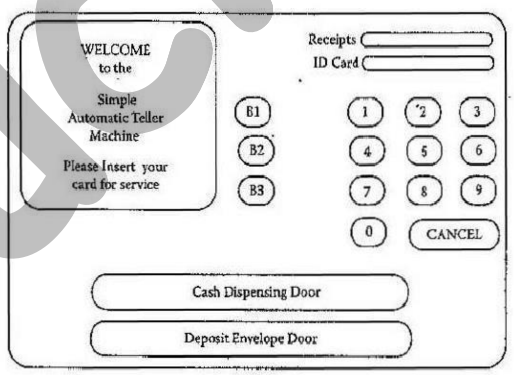
Figure: Executable Specification



The SATM System

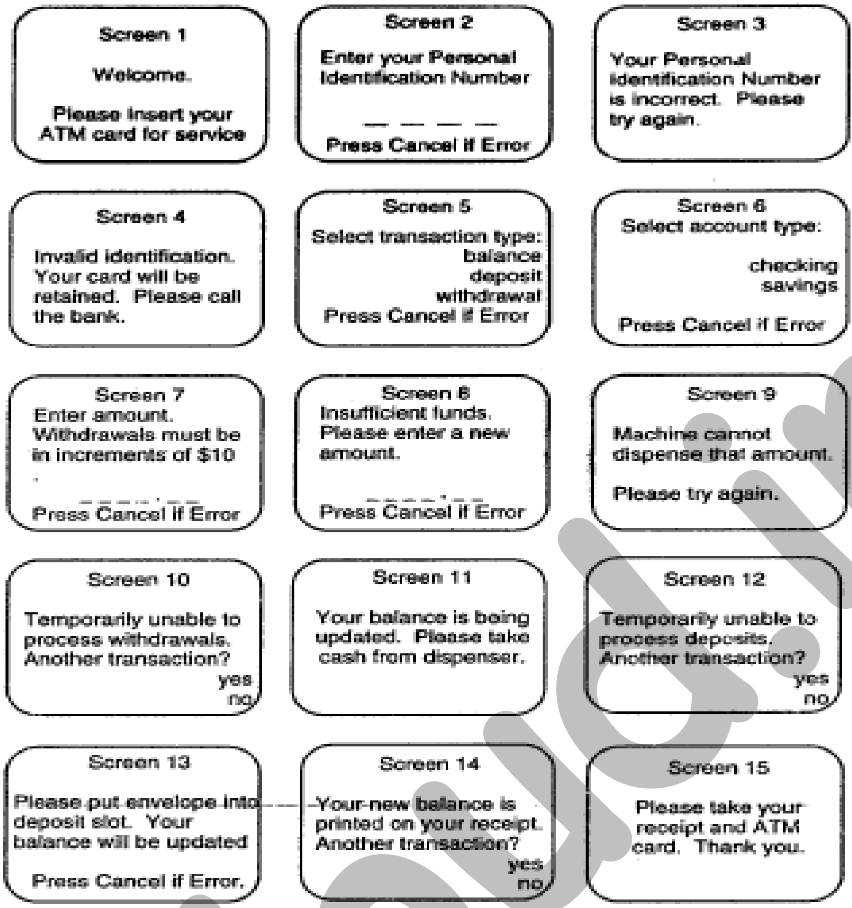
- Consider an example, **Simple Automatic Teller Machine (SATM)**. The **SATM terminal** is given in **below Figure.** In addition to the **display screen**, there are **function buttons B1, B2** and **B3**, a **digit keypad** with a **cancel key**, **slots** for **printer receipts** and **ATM cards** and **doors** for **deposits** and **cash withdrawals**.

Figure: The SATM Terminal



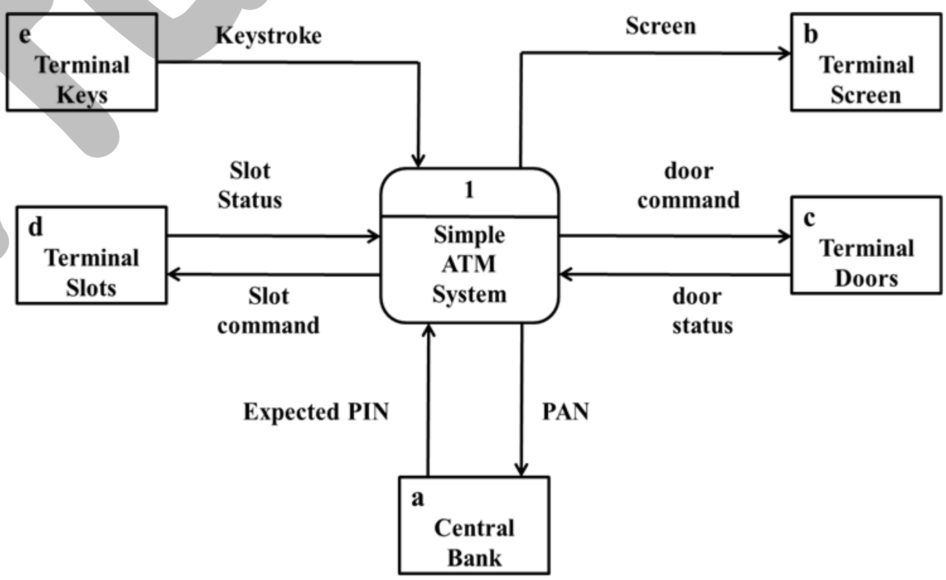
- The **SATM** system which is built around the **fifteen screens** shown in **below Figure.**
- The SATM system is described in **two ways: with a traditional approach (Figure (a))** and **structured analysis approach (Figure (b)).** These **descriptions** are **not complete**, but they contain **sufficient detail** to illustrate the **testing techniques.**
- **The structured analysis approach** to **requirements specification** is the most widely used method in the world. It enjoys extensive **CASE tool** support as well as **commercial training**. The **technique** is based on **three complementary models**: **function, data** and **control.**

Figure: Screens for the SATM System.



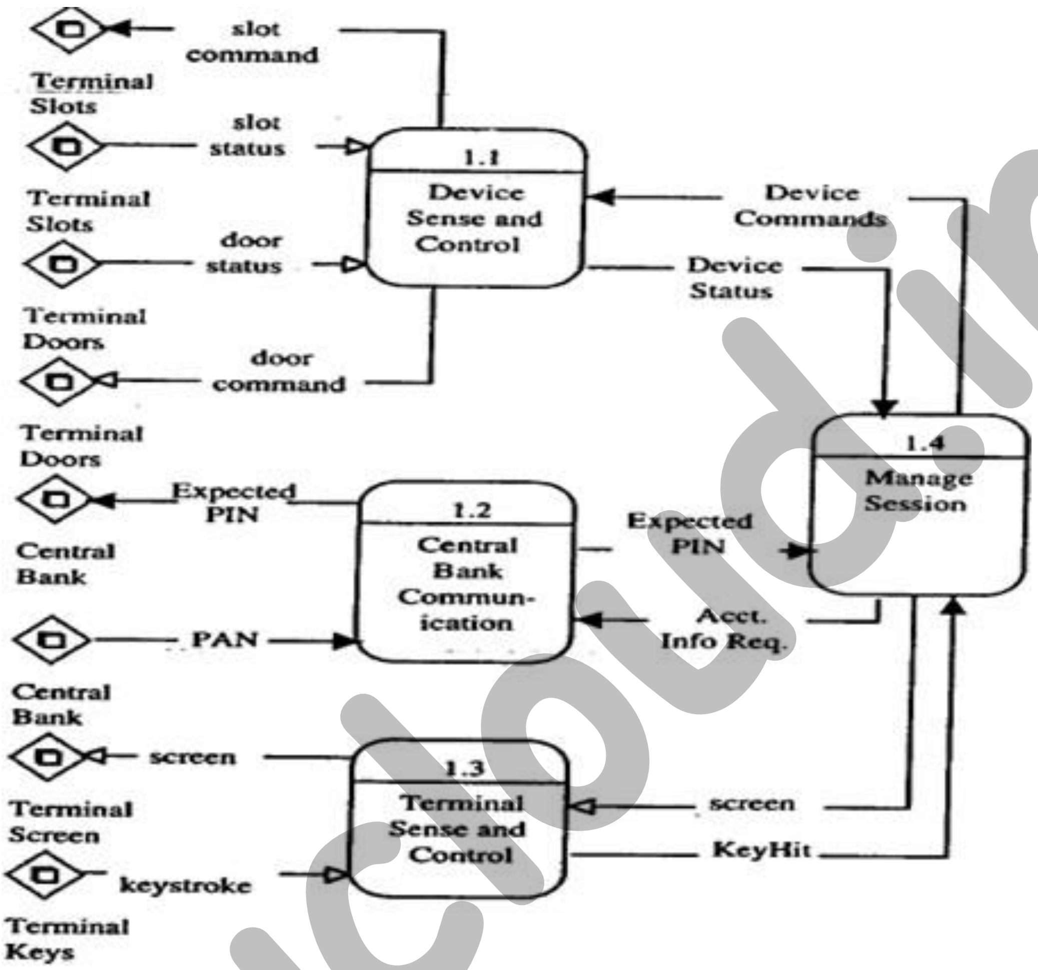
- We use **data flow diagrams** for the **functional models**, **entity/relationship models** for **data** and **finite state machine models** for the **control aspect** of the SATM system. The **functional** and **data models** were drawn with the **Deft CASE tool** from **Sybase Inc**. That tool identifies **external devices** (such as the terminal doors) with **lower case letters** and **elements** of the **functional decomposition** with **numbers**.

Figure (a): Context Diagram of the SATM System



- The **open** and **filled arrowheads** on **flow arrows** signify whether the **flow item** is **simple** or **compound**. The portions of the SATM system shown here pertain generally to the **Personal Identification Number (PIN)** verification portion of the system.

Figure (b): Level 1 Dataflow Diagram of the SATM System



- The **Deft CASE tool** distinguishes between **simple** and **compound flows**, where **compound flows** may be **decomposed** into **other flows**, which may themselves be compound. The **graphic appearance** of this choice is that **simple flows** have **filled arrowheads**, while **compound flows** have **open arrowheads**. As an **example**, the **compound flow "screen"** has the following **decomposition.**

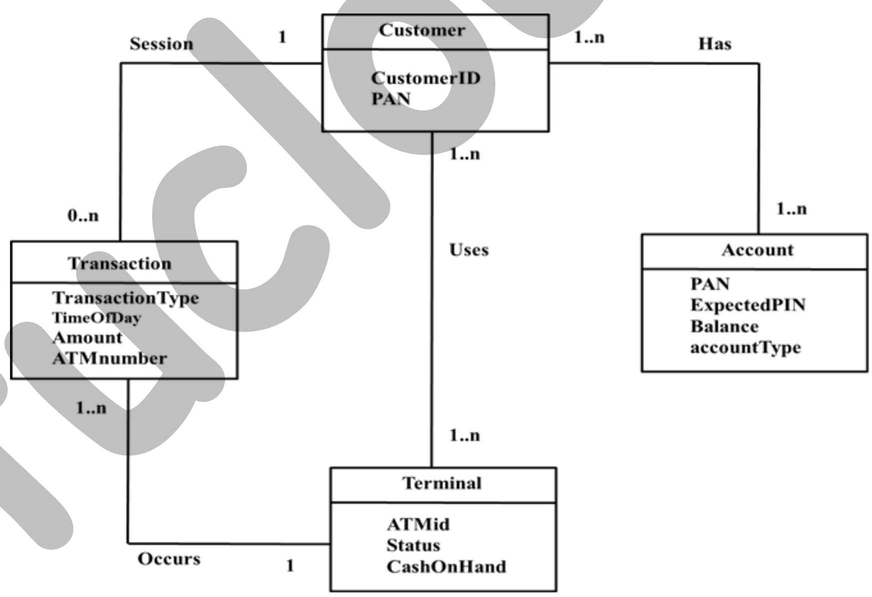
  screen is comprised of:

  screen 1 welcome
  screen 2 enter PIN
  screen 3 wrong PIN
  screen 4 PIN failed, card retained
  screen 5 select trans type
  screen 6 select account type
  screen 7 enter amount
  screen 8 insufficient funds
  screen 9 cannot dispense that amount
  screen 10 cannot process withdrawals
  screen 11 take your cash

screen 12 cannot process deposits
screen 13 put dep envelop in slot
screen 14 another transaction?
screen 15 Thanks, take card and receipt

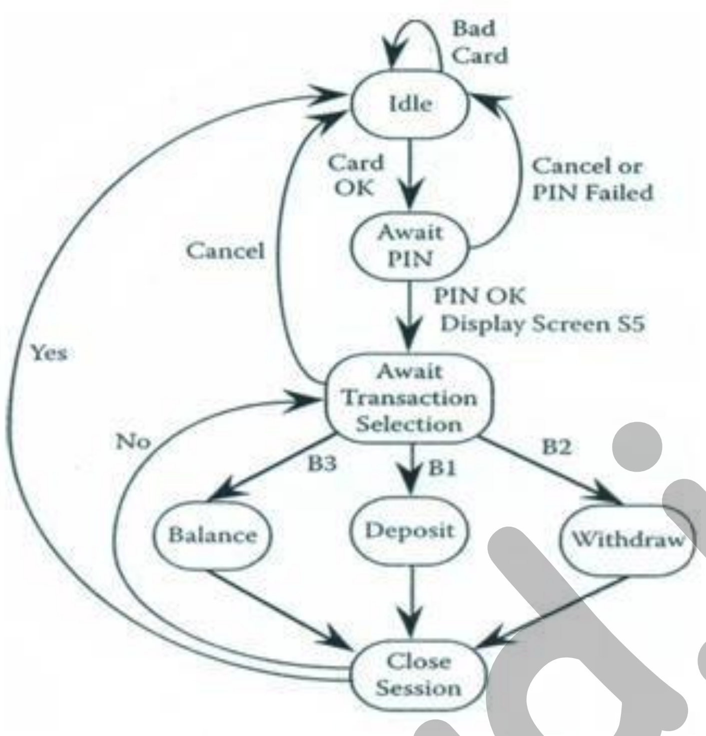
- The **below Figure** is an **(incomplete) Entity/Relationship** diagram of the major **data structures** in the SATM system: **Customers, Accounts, Terminals** and **Transactions**.
- The **data** the **system** would need for **each customer** are the **customer's identification** and P**ersonal Account Number (PAN).** These are **encoded** into the **magnetic strip** on the **customer's ATM card**. We would also want to **know information** about a **customer's accounts**, including the **account numbers**, the **balances**, and the **type of account** (savings or checking) and the **Personal Identification Number (PIN)** of the account.
- **Part** of the **E/R model** describes **relationships** among the **entities**: a customer **HAS** accounts, a customer **conduct** transactions in a **SESSION** and **independent** of **customer information**, transactions **OCCUR** at an ATM terminal.
- The **single** and **double** arrowheads signify the **singularity** or **plurality** of these **relationships: one customer** may have **several accounts** and may **conduct none** or **several** transactions. **Many transactions** may occur at a **terminal,** but **one transaction** never occurs at a **multiplicity** of terminals.

Figure: Entity/Relationship Model of the SATM System



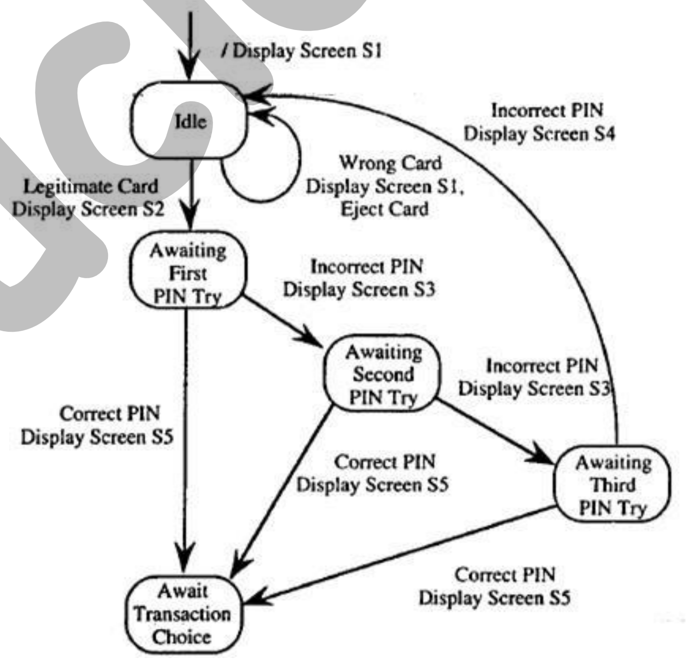
- The **upper level finite state machine** in **below Figure divides** the **system** into **states** that **correspond** to **stages** of customer usage. Finite state machines can be **hierarchically decomposed** in much the same way as **dataflow diagrams**.

Figure: Upper Level SATM Finite State Machine



- The **decomposition** of the **Await PIN** state is shown in **below Figure.** In both of these figures, **state transitions** are **caused** either by **events** at the **ATM terminal** (such as a keystroke) or by **data conditions** (such as the recognition that a PIN is correct). When a transition occurs, a **corresponding action** may also occur.

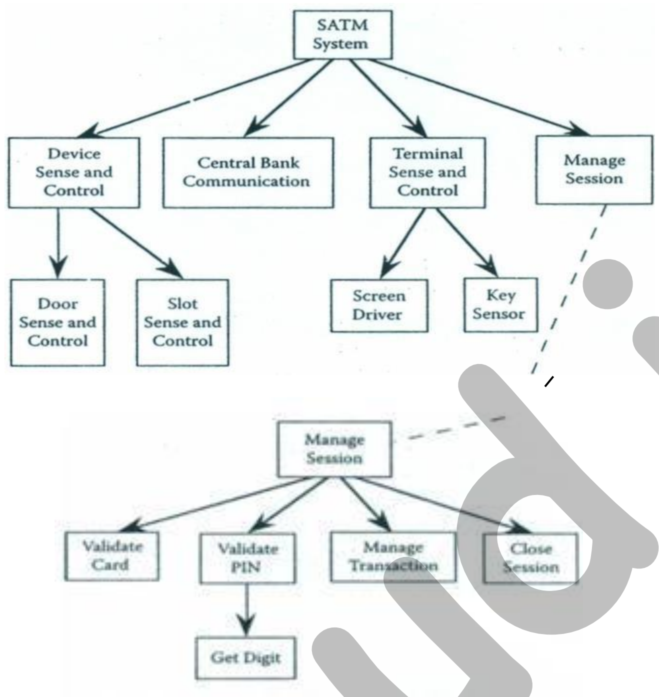
Figure: PIN Entry Finite State Machine



- The **function, data** and **control** models are the **basis** for **design activities** in the waterfall model (and its spin-offs). During **design,** some of the **original decisions** may be **revised** based on **additional insights** and more **detailed requirements**.

- The **end result** is a functional **decomposition** such as **the partial one** shown in the **structure chart** in **below Figure.**

Figure: A Decomposition Tree for the SATM System



- If we use a **structure chart** to **guide integration testing**, we **miss** the fact that some **lower level functions** are used in **more than one place**. Here, **for example**, the **ScreenDriver** function is used by **several** other modules, but it **only appears once** in the functional decomposition.
- A **"call graph"** is a much **better basis** for **integration test case identification**. To **support this**, we need a **numbered decomposition**, and a **more detailed view** of **two** of the **components.**
- Here is the **functional decomposition** carried further in outline form: the **numbering scheme** preserves the **levels** of the **components** in the **above Figure**.

1 SATM System
1.1 Device Sense & Control
1.1.1 Door Sense & Control
1.1.1.1 Get Door Status
1.1.1.2 Control Door
1.1.1.3 Dispense Cash
1.1.2 Slot Sense & Control
1.1.2.1 WatchCardSlot
1.1.2.2 Get Deposit Slot Status
1.1.2.3 Control Card Roller
1.1.2.4 Control Envelope Roller
1.1.2.5 Read Card Strip
1.2 Central Bank Comm.
1.2.1 Get PIN for PAN

1.2.2 Get Account Status
1.2.3 Post Daily Transactions

1.3 Terminal Sense & Control

1.3.1 Screen Driver
1.3.2 Key Sensor

1.4 Manage Session

1.4.1 Validate Card
1.4.2 Validate PIN

1.4.2.1 GetPIN

1.4.3 Close Session

1.4.3.1 New Transaction Request
1.4.3.2 Print Receipt
1.4.3.3 Post Transaction Local

1.4.4 Manage Transaction

1.4.4.1 Get Transaction Type
1.4.4.2 Get Account Type
1.4.4.3 Report Balance
1.4.4.4 Process Deposit
1.4.4.5 Process Withdrawal

- As part of the **specification** and **design** process, each **functional component** is normally **expanded** to show its **inputs, outputs** and **mechanism**. We **do this** here with **pseudo-code** or **Program Design Language (PDL)** for three modules.
- The **main program** description follows the **finite state machine** description given in the **above Figure: Upper Level SATM Finite State Machine.**
- **States** in that diagram are **implemented** with a **CASE statement**.
- The **ValidatePIN** procedure is based on another finite state machine shown in **above Figure: PIN Entry Finite State Machine**, in which **states** refer to the **number** of **PIN entry attempts.**
- The **GetPIN** procedure is based on another **finite state machine** in which **states refer** to the **number of digits received** and in any state, either **another digit key** or the **cancel key** can
be **touched.**

Main Program :

```
State = AwaitCard
Do                      'Main loop
Case State
    Case 1:             AwaitCard
                        ScreenDriver (1, null)
                        WatchCardSlot (CardsSlotStatus)
                        Do While CardSlotStatus is Idle
                            WatchCardSlot ( CardSlotStatus)
                        End While
                        ControlCardRoller (accept)
                        ValidateCard (CardOK, PAN)
                        If  CardOK
```

```
                    Then   State = Await PIN
                    Else   ControlCardRoller (eject)
               End If
               State = AwaitCard
     Case 2:   Await PIN
               Validate PIN (PINok, PAN)
               If  PINok
                    Then   ScreenDriver (2, null)
                           State = AwaitTrans
                    Else   ScreenDriver (4, null)
                           State = AwaitCard
               EndIf

     Case 3:   AwaitTrans
                    ManageTransaction
                    State = CloseSession

     Case 4:   CloseSession
                    If   NewTransactionRequest
                         Then   State = AwaitTrans
                         Else   PrintReceipt
                    End If
                    PostTransactionLocal
                    CloseSession
                    ControlCardRoller (eject)
                    State =AwaitCard
End Case  (State)
Until     'Forever
End.           (Main program SATM)


Procedure ValidatePIN (PINok, PAN)
Get PINforPAN (PAN, ExpectedPIN)
Try = First
Case Try of
   Case 1: First
        ScreenDriver ( 2, null)
        Get PIN (EnteredPIN, CancelHit)
        If  EnteredPIN =  ExpectedPIN
             Then   PINok = True
             Else   ScreenDriver(3, null)
             Try = Second
        Endif

   Case 2: Second
        ScreenDriver(2,null)
        Get PIN (Entered PIN)
        If  EnteredPIN = ExpectedPIN
```

```
                Then   PINok =True
                Else    ScreenDriver (3, null)
                Try = Third
            Endif

    Case 3: Third
            ScreenDriver (2, null)
            GetPIN (EnteredPIN, CancelHit)
            If    EnteredPIN= ExpectedPIN
                Then   PINok = True
                Else    ScreenDriver( 4, null)
                        PINok = False
            End if
EndCase  (Try)
End.           (Procedure ValidatePIN)
Procedure GetPIN (EnteredPIN, CancelHit)
Local Data:  Digital Keys = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
CancelHit =False
EnteredPIN = null string
digitsRcvd=0
Do While NOT (DigitsRcvd = 4 OR CancelHit)
KeySensor (KeyHit)
If    KeyHit  IN  DigitKeys
    Then
            EnteredPIN =EnteredPIN + KeyHit
            digitsRcvd = digitsRcvd  + 1
            If   digitsRcvd  = 1
                Then    ScreenDriver (2, 'X- - - ')
            Endif
            If   digitsRcvd  = 2
                Then    ScreenDriver (2, 'XX- -')
            EndIf
            If   digitsRcvd = 3
                Then    ScreenDriver (2, 'XXX-')
            EndIf
            If   digitRcvd = 4
                Then    ScreenDriver (2, 'XXXX')
            EndIf
    Else
            CancelHit = True
EndIf
End While
End.    (Procedure Get PIN)
```

- If we follow the **pseudocode** in these **three modules**, we can identify the **uses relationship** among the modules in the functional decomposition.

Module	**Uses Modules**
SATM Main	WatchCardSlot
	ControlCardRoller
	ScreenDriver
	ValidateCard
	ValidatePIN
	ManageTransaction
ValidatePIN	GetPINforPAN
	GetPIN
	ScreenDriver
GetPIN	KeySensor
	ScreenDriver

➢ Separating Integration and System Testing

- The clear **distinction** between **integration** and **system testing** is needed to **avoid gaps** and **redundancies** across levels of testing, to **clarify appropriate goals** for these levels, and to **understand** how to **identify test cases** at **different levels**. The discussion is facilitated by the **threads.** A **thread** is a **construct** that refers to **execution time behavior.** When we test a system, **test cases** are used to **select threads**.
- **System threads** describe **system level behavior**, **integration threads** correspond to **integration level behavior** and **unit threads** correspond to **unit level behavior**.
- We shall find that both **structural** and **behavioral** views help us to **separate integration** and **system** testing. The **structural view reflects** both the **process** by which a **system** is **built** and the **techniques used** to **build it**.

✓ Structural Insights

- **Integration testing** is at a more **detailed level** than **system testing**. Integration testing can safely **assume** that the **units** have been **separately tested**, and that, **taken individually**, the **units function correctly**. Integration testing is concerned with the **interfaces among** the **units**.
- In the **waterfall life cycle** model **integration testing** is concerned with **preliminary design information,** while **system testing** is at the level of the **requirements specification**. The **requirements specification** defines **what**, and the **preliminary design** describes **how**.
- In **SATM system**, we could first **postulate** that **system testing** should make sure that all **fifteen display screens** have been **generated**.
- The **entity/relationship model**, the **one-to-one** and **one-to-many** relationships help us **understand how much testing** must be done. The **control model** (in this case, a hierarchy of finite state machines) is the **most helpful**.
- We can **postulate system test cases** in terms of **paths** through the **finite state machine** which yields a **system level analog** of **structural testing.** The **functional models** (dataflow diagrams and structure charts) move in the direction of **levels** because both

express a **functional decomposition.** Even with this, we **cannot** look at a **structure chart** and **identify** where **system testing ends** and **integration testing starts**.

- The **best** we can **do** with **structural information** is **identifying** the **extremes**. **For instance**, the following **threads** are all clearly at the **system level**.
  1. Insertion of an **invalid card**. (this is probably the shortest system thread)
  2. Insertion of a valid card, followed by **three failed PIN entry** attempts.
  3. Insertion of a valid card, a **correct PIN entry attempt**, followed by a **balance inquiry**.
  4. Insertion of a valid card, a **correct PIN entry attempt**, followed by a **deposit**.
  5. Insertion of a valid card, a **correct PIN entry attempt**, followed by a **withdrawal**.
  6. Insertion of a valid card, a **correct PIN entry attempt**, followed by an attempt to **withdraw more cash** than the **account balance**.
- We can also identify some **integration level threads**. Considering the **pseudocode** of **ValidatePIN** and **GetPIN, ValidatePIN calls GetPIN** and **GetPIN waits for KeySensor** to report when a key is touched. If a **digit** is **touched, GetPIN echoes** an **"X"** to the display screen, but if the **cancel key** is **touched, GetPIN terminates** and **ValidatePIN** considers **another PIN entry** attempt. Still lower consider **keystroke sequences** such as **two** or **three digits** followed by **cancel keystroke.**

✓ Behavioral Insights

- Consider a **system** in terms of its **port boundary**, which is the **location** of **system level inputs** and **outputs.** The **port boundary** of the **SATM system** includes the **digit keypad, the function buttons, the screen, the deposit and withdrawal doors, the card and receipt slots** and so on.
- Each of these devices can be thought of as a **"port"** and **events** occur at **system ports**. The **port input** and **output events** are **visible** to the **customer** and the customer very often **understands system behavior** in terms of **sequences** of **port events**. Given this, we **mandate** that **system port events** are the **"primitives"** of a **system test case**, that is, a system test case (or equivalently, a system thread) is expressed as an **interleaved sequence** of **port input** and **port output events**.
- **Threads** support a **highly analytical view** of **testing**. **Unit level threads** are **sequence** of **source statements** that **execute. Integration level threads** can be **sequence** of **unit level threads** where we are **concerned** with **interaction** among them. Finally, **system level threads** can be **interpreted** as **sequences** of **integration level threads**.

Chapter 13

Integration Testing

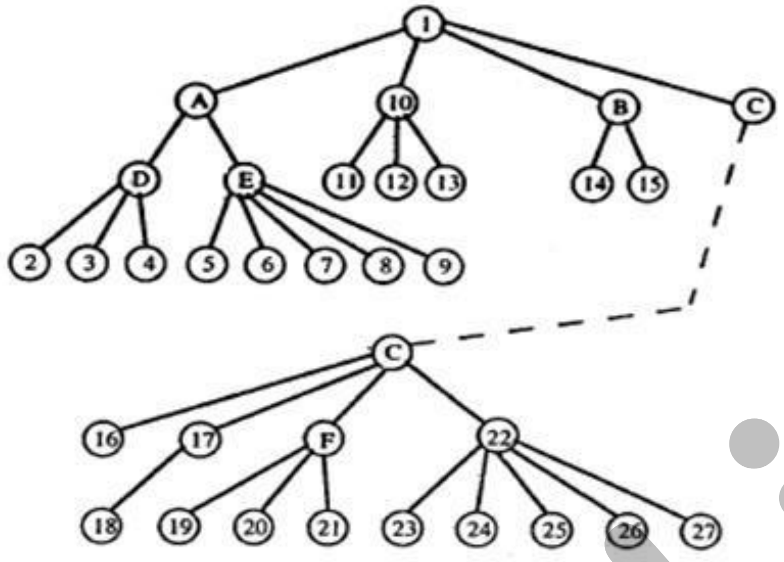
➢ A Closer Look at the SATM System

- The **decomposition** in **below Table** is pictured as a **decomposition tree** in **below Figure**. This decomposition is the **basis** for the integration testing and it is primarily a **packaging partition** of the system.

Table: SATM Units and Abbreviated Names

Unit Number	Level Number	Unit Name
1	**1**	SATM System
A	**1.1**	Device Sense & Control
D	**1.1.1**	Door Sense & Control
2	**1.1.1.1**	Get Door Status
3	**1.1.1.2**	Control Door
4	**1.1.1.3**	Dispense Cash
E	**1.1.2**	Slot Sense & Control
5	**1.1.2.1**	WatchCardSlot
6	**1.1.2.2**	Get Deposit Slot Status
7	**1.1.2.3**	Control Card Roller
8	**1.1.2.4**	Control Envelope Roller
9	**1.1.2.5**	Read Card Strip
10	**1.2**	Central Bank Comm.
11	**1.2.1**	Get PIN for PAN
12	**1.2.2**	Get Account Status
13	**1.2.3**	Post Daily Transactions
B	**1.3**	Terminal Sense & Control
14	**1.3.1**	Screen Driver
15	**1.3.2**	Key Sensor
C	**1.4**	Manage Session
16	**1.4.1**	Validate Card
17	**1.4.2**	Validate PIN
18	**1.4.2.1**	GetPIN
F	**1.4.3**	Close Session
19	**1.4.3.1**	New Transaction Request
20	**1.4.3.2**	Print Receipt
21	**1.4.3.3**	Post Transaction Local
22	**1.4.4**	Manage Transaction
23	**1.4.4.1**	Get Transaction Type
24	**1.4.4.2**	Get Account Type
25	**1.4.4.3**	Report Balance
26	**1.4.4.4**	Process Deposit
27	**1.4.4.5**	Process Withdrawal

Figure: SATM Functional Decomposition Tree



- As software design moves into more detail, the added information lets us refine the functional decomposition tree into a **unit calling graph.**
- The **unit calling graph** is the **directed graph** in which **nodes** are **program units** and **edges** correspond to **program calls,** that is, if **unit A** calls **unit B**, there is a **directed edge** from **node A** to **node B**. The **information** of the call graph for the **SATM system** is captured in the **adjacency matrix** given **below Table.**
- The **SATM call graph** is shown in **below Figure.**

Figure: SATM Call Graph

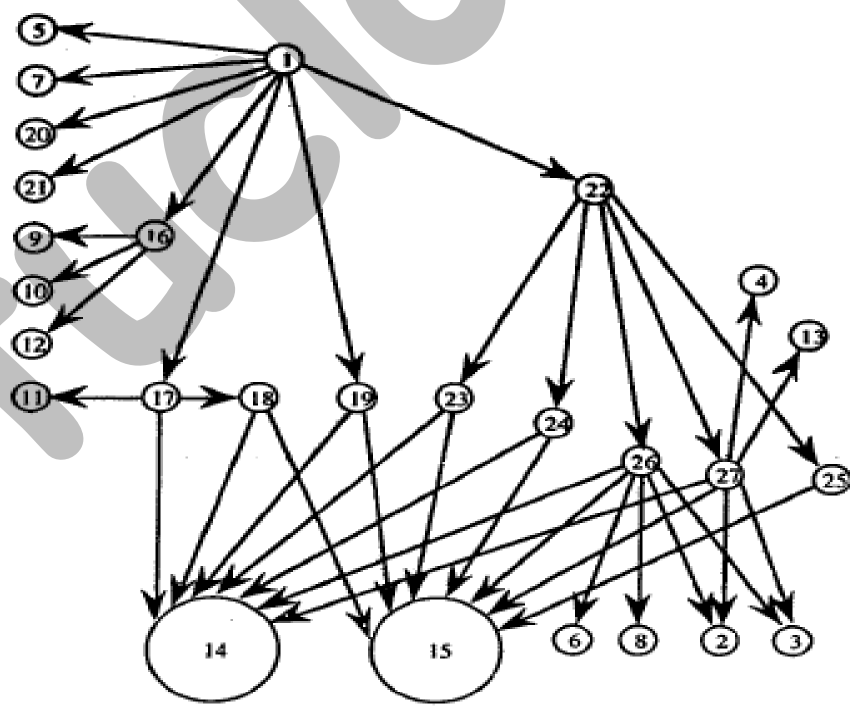


Table: Adjacency Matrix for the SATM call Graph.

Table 13.2 Adjacency Matrix for the SATM Call Graph

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1				x		x							x		x	x		x	x	x	x					
2																										
3																										
4																										
5																										
6																										
7																										
8																										
9																										
10																										
11																										
12																										
13																										
14																										
15																										
16								x	x		x															
17										x			x				x									
18													x	x												
19													x	x												
20																										
21																										
22													x	x								x	x	x	x	x
23													x	x												
24													x	x												
25														x												
26	x	x			x		x					x	x	x												
27	x	x	x		x		x					x	x	x												

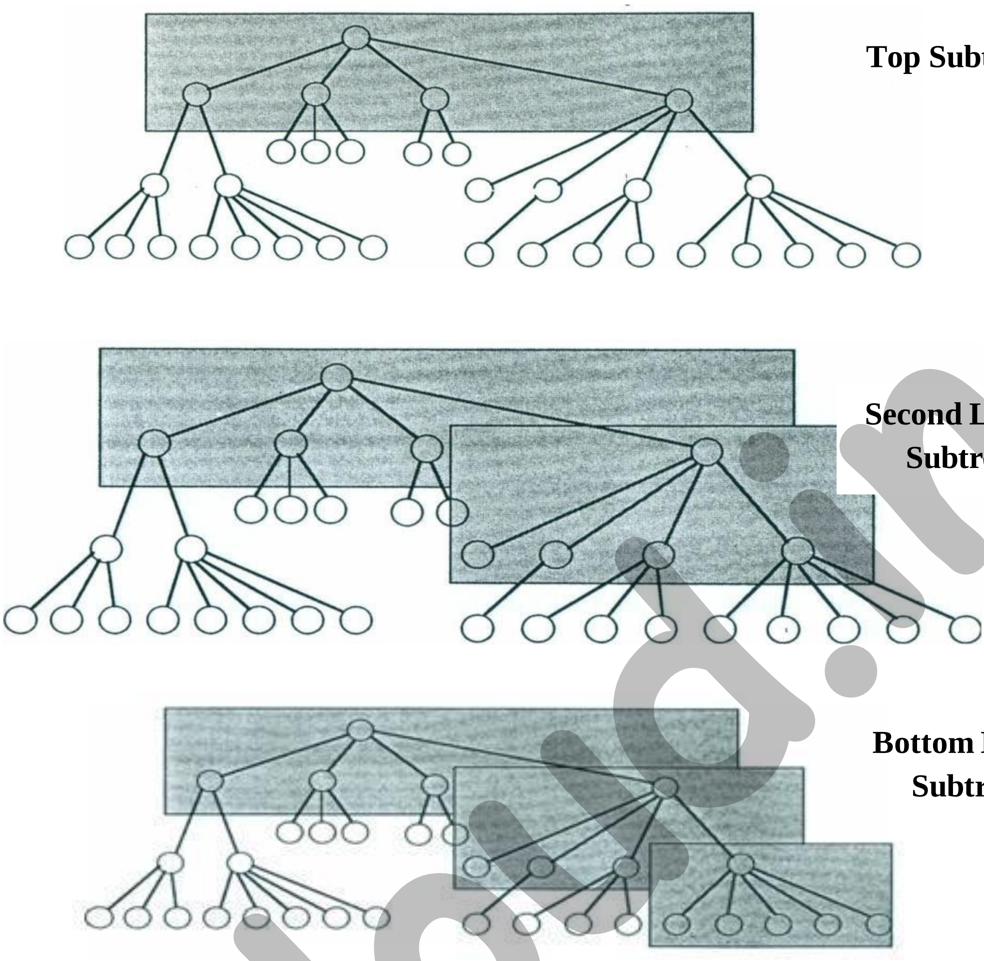
➢ Decomposition-Based Integration

- The **four integration strategies** based on the **functional decomposition tree** of the **procedural software are top–down, bottom–up, sandwich** and the **big bang.**
- Each of these **strategies (except big bang)** describes the **order** in which **units** are to be **integrated**. The **functional decomposition tree** is the **basis** for **integration testing** because it is the **main representation**, usually **derived** from **final source code**, which shows the **structural relationship** of the system with respect to its units.

✓ Top-Down Integration

- **Top-down integration** begins with the **Main program** (the root of the tree). Any **lower level unit** that is **called** by the **Main program** appears as a **"stub"**, where stubs are **pieces of throw-away code** that **emulate** a **called unit**. If we perform **top-down integration** testing for the SATM system, the **first step** would be to **develop stubs** for **all** the **units called** by the **Main program.**
- When we are **convinced** that the **Main program logic** is **correct**, we gradually **replace stubs** with the **actual code**.
- The **below Figure** shows **part of the top-down integration testing sequence** for the **SATM Functional Decomposition Tree** figure shown above. At the upper most level we have stubs for four components in the first level decomposition.
- Even this can be **problematic.** If we **replace one stub** at a time, we **retest** the Main program once for each **replaced stub**. This means that, for the SATM main program example, we would **repeat** its integration test **five times** (once for each replaced stub, and once with all the stubs).

Figure: Top-down Integration



Top Subtree

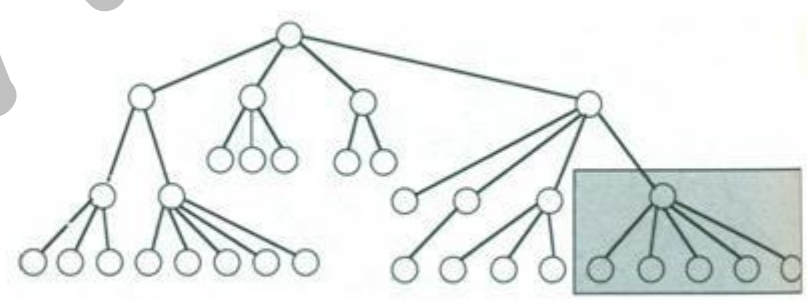
Second Level Subtree

Bottom Level Subtree

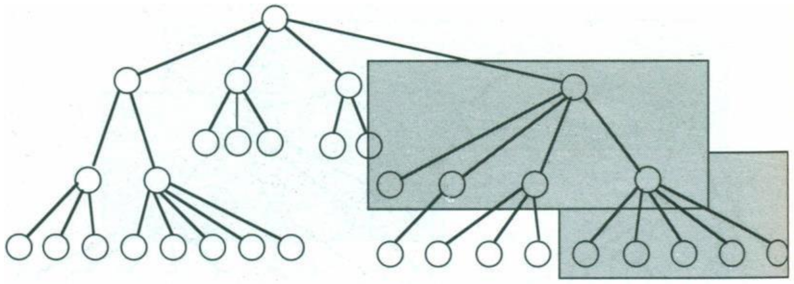
✓ Bottom-up Integration

- **Bottom-up integration** is a **mirror image** to the **top-down order**, with the difference that **stubs** are replaced by **driver modules** that **emulate units** at the **next level up** in the tree as shown in **below** F**igure**. In bottom-up integration, we **start** with the **leaves** of the decomposition tree (units like ControlDoor and DispenseCash) and **test** them with specially **coded drivers.**
- There is probably **less throw-away code** in **drivers** than there is in stubs we will not have as many drivers.

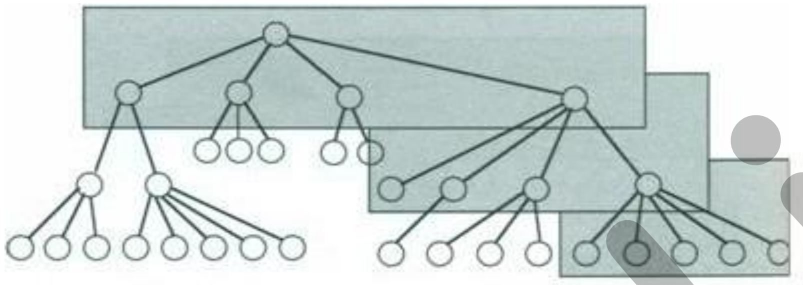
F**igure: Bottom-up Integration**



Bottom Subtree



Second Level Subtree

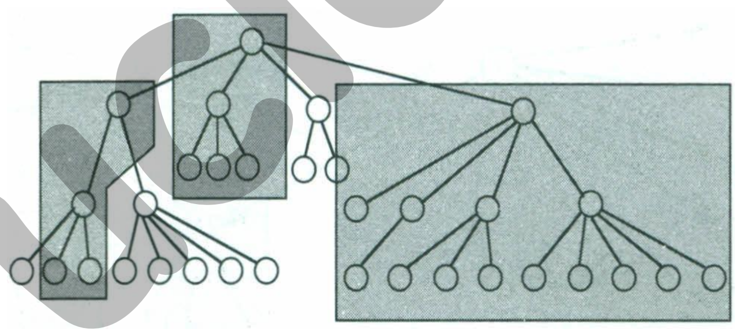


Top Level Subtree

✓ Sandwich Integration

Sandwich integration is a **combination** of **top-down** and **bottom-up** integration. If we think about it in terms of the decomposition tree, we are really **just doing big bang integration** on a **sub-tree** as shown in **below Figure**. There will be **less stub** and **driver** development effort, but this will be **offset** to some extent by the **added difficulty** of **fault isolation** that is a consequence of big bang integration.

Figure: Sandwich Integration



✓ Pros and Cons

- With the **exception** of **big bang integration**, the **decomposition-based approaches** are all **clear**. Whenever a **failure** is observed, the most **recently added unit** is **suspected**. **Integration testing progress** is easily **tracked** against the **decomposition tree**.
- The **top-down** and **bottom-up** terms suggest **breadth-first traversals** of the decomposition tree, but this is **not mandatory**.
- One of the most frequent **objections** to **functional decomposition** and **waterfall development** is that both are **artificial** and both **serve** the **needs** of **project management more than** the **needs** of **software developers**.
- The whole **mechanism** is that **units** are **integrated** with respect to **structure.** The **development effort** for **stubs** or **drivers** is **another drawback** to these approaches, and this is compounded by the **retesting effort**.

- The **formula** that **computes** the **number** of **integration test sessions** for a given decomposition tree is,

  Session = nodes – leaves + edges

- The **SATM system** has **42 integration testing sessions**, which means **42 separate sets** of **integration test cases.** For **top-down integration, (nodes – 1) stubs** are **needed** and for **bottom-up integration (nodes-leaves) drivers** are **needed.** There are **32 stubs** and **10 drivers** in **SATM system.**

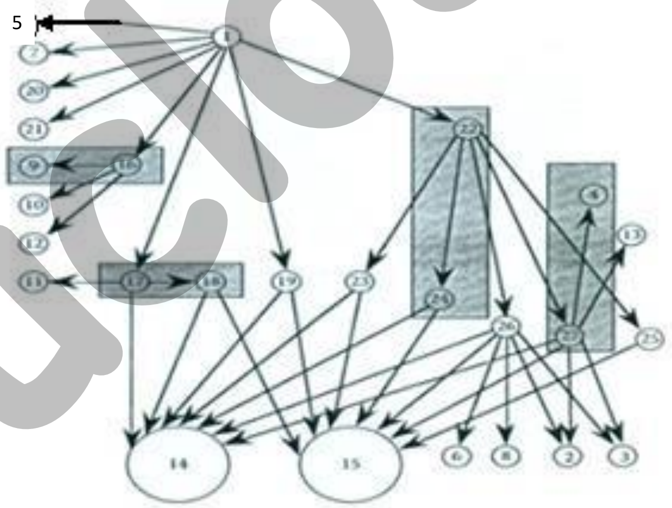
➢ Call Graph Based Integration

- One of the **drawbacks** of **decomposition based integration** is that the **functional decomposition tree**. If we use the **call graph**, we **mitigate** this **deficiency**. The **call graph** is a **directed graph**. The **two new approaches** to **integration testing** are **pair-wise integration** and **neighborhood integration**.

✓ Pairwise Integration

- The **idea** behind **pairwise integration** is to **eliminate** the **stub/driver development effort**. We **restrict** a **session** to just a **pair of units** in the **call graph**. The **end result** is that we have **one integration test session** for **each edge** in the **call graph** (**40 for the above Figure: SATM call graph)**. This is **not much** of a **reduction** in **sessions** from either **top-down** or **bottom-up (42 sessions)**, but it is a **drastic reduction** in **stub/driver development**. **Four pair-wise integration sessions** are shown in **below Figure.**

  Figure: pairwise integration



✓ Neighborhood Integration

- The **neighborhood** of **radius** 1of a node in a graph is the **set of nodes** that are **one edge away** from the given node.
- In a **directed graph**, this includes all the **immediate predecessor nodes** and all the **immediate successor nodes**. The **neighborhoods** for the **nodes 16** and **26** are shown in **below Figure**. The **11 neighborhoods** for the SATM example (based on the above **SATM Call Graph figure**) are given in **below Table.**

Figure: Neighborhood Integration

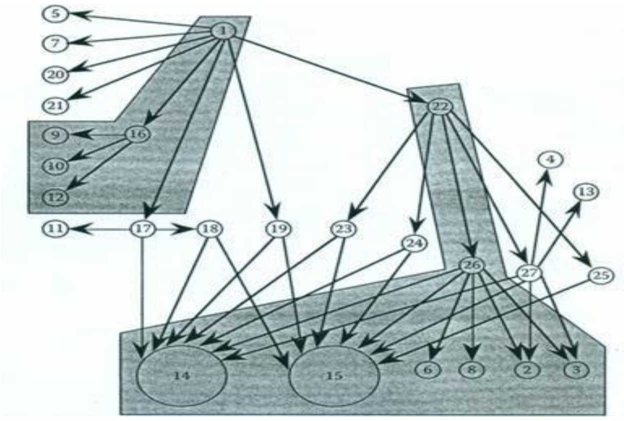


Table: SATM Neighborhoods

Node	Predecessors	Successors
16	1	9,10,12
17	1	11,14,18
18	17	14,15
19	1	14,15
23	22	14,15
24	22	14,15
26	22	14,15,6,8,2,3
27	22	14,15,2,3,4,13
25	22	15
22	1	23,24,26,27,25
1	n/a	5,7,2,21,16,17,19,22

- We can **compute** the **number of neighborhoods** for a given call graph using,
  Neighborhoods = nodes -sink nodes
- **Neighborhood integration** yields a **drastic reduction** in the **number of integration test sessions** (down to 11 from 40), and it **avoids stub** and **driver development**. The **end result** is that neighborhoods are essentially the **sandwiches.** What they share with sandwich integration is **more significant**: neighborhood integration testing has the **fault isolation difficulties** of **"medium bang" integration.**

➢ Path Based Integration

- When a **unit executes**, some **path of source statements** is **traversed.** Suppose that there is a **call** to **another unit** along such a path: at that point, **control** is **passed** from the **calling unit** to the **called unit**, where some other path of source statements is traversed.
- There are **two possibilities, abandon the single-entry, single exit precept** and **treat such calls as an exit followed by an entry**, or **suppress the call statement** because control eventually returns to the calling unit anyway.
- The **suppression choice works well** for **unit testing**.

✓ New and Extended Concepts

Definition

A **source node** in a program is a statement fragment at which **program execution begins** or **resumes.**

The **first executable statement** in a unit is a **source node**. Source nodes also occur immediately after nodes that transfer control to other units.

Definition

A **sink node** in a unit is a statement fragment at which **program execution terminates**.

The **final executable statement** in a program is a **sink node,** these are statements that transfer control to other units.

Definition

A **module execution path** is a **sequence of statements** that **begins** with a **source node** and **ends with a sink node**, with **no intervening sink nodes**.

The effect of the definitions thus far is that program graphs now have multiple source and sink nodes. This would greatly increase the complexity of unit testing, but integration testing presumes unit testing is complete.

Definition

A ***message*** is a programming language mechanism by which one unit transfers control to another unit, and acquires a response from the other unit.

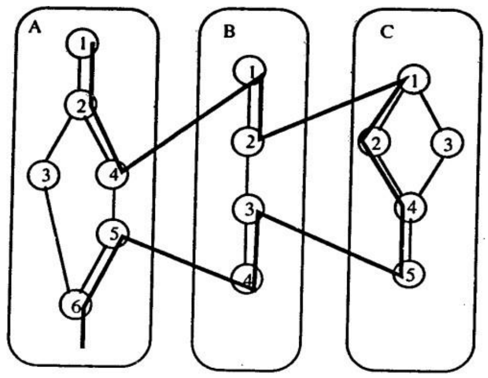
Depending on the programming language, messages can be interpreted as subroutine invocations, procedure calls and function references.

Definition

An **MM-Path** is an interleaved sequence of **module execution paths and messages.**

The basic idea of an MM-Path is that we can now describe sequences of module execution paths that include transfers of control among separate units. Since these transfers are by messages, MMPaths always represent feasible execution paths, and these paths cross unit boundaries. We can find MM-Paths in an extended program graph in which nodes are module execution paths and edges are messages. The hypothetical example in below Figure shows an MM-Path (the dark line) in which module A calls module B, which in turn calls module C.

Figure(a): MM-Path across three units



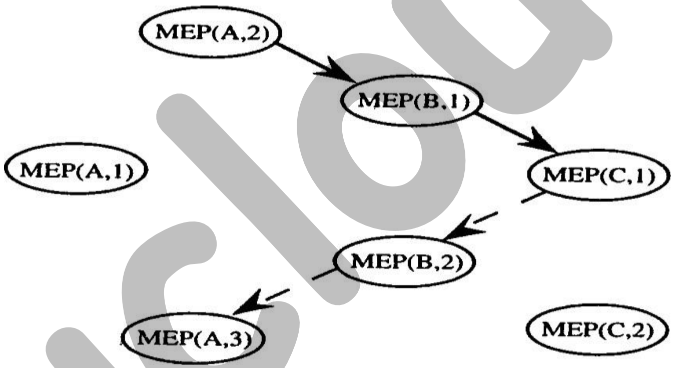
In module A, nodes 1 and 5 are source nodes, and nodes 4 and 6 are sink nodes. Similarly in module B, nodes 1 and 3 are source nodes and nodes 2 and 4 are sink nodes. Module C has a single source node, 1, and a single sink node, 4. There are **seven module execution paths** in **above Figure(a).**

MEP (A, 1) = <1, 2, 3, 6>
MEP (A, 2) = <1, 2, 4>
MEP (A, 3) = <5, 6>
MEP (B, 1) = <1, 2>
MEP (B, 2) = <3, 4>
MEP (C, 1) = <1, 2, 4, 5>
MEP (C, 2) = <1, 3, 4, 5>

We can now define an integration testing analog of the DD-Path graph that serves unit testing so effectively.

Definition
Given a set of units, their **MM-Path graph** is the directed graph in which nodes are module execution paths and edges correspond to messages and returns from one unit to another.

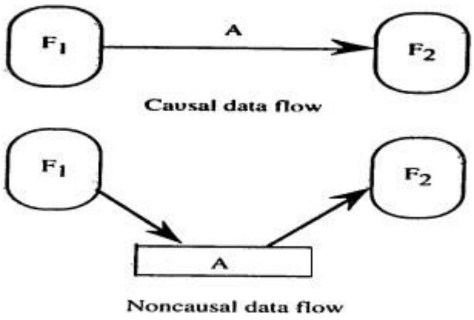
Figure: MM-Path Graph Derived from above Figure(a)



- The **above Figure** shows the **MM-Path graph** for the example in **above Figure(a).** The solid arrows indicate messages. The corresponding returns are indicated by dotted arrows. We should consider the relationships among module execution paths, program path, DD-Paths, and MM-Paths. A program path is a sequence of DD-Paths, and an MM-Path is a sequence of module execution paths.
- Consider the **“intersection”** of an MM-Path with a unit. The module execution paths in such an intersection are an analog of a slice with respect to the (MM-Path) function. Stated another way, the module execution paths in such an intersection are the restriction of the function to the unit in which they occur.
- The MM-Path definition needs some practical guidelines. How long is an MM-Path? Two observable behavioral criteria put endpoints on MMPaths: **message** and **data quiescence. Message quiescence** occurs when a unit that sends no message is reached. (like **module C** in **Figure(a)).**
- Data quiescence occurs when a sequence of processing culminates in the creation of stored data that is not immediately used. In the ValidateCard unit, the account balance is obtained, but it is not used until after a successful PIN entry. The below Figure shows how

data quiescence appears in a traditional dataflow diagram. Points of quiescence are natural endpoints for an MM-Path.

Figure: Data Quiescence



✓ MM-Paths in the SATM System

The statement fragments are numbered as we did to construct program graphs. The messages are numbered as comments. We use these to describe selected MM-Paths. The arguments to ScreenDriver refer to the screens as numbered in SATM system. Procedure is a stub that is designed to respond to a correct event sequence for ExpectedPIN = 1234.

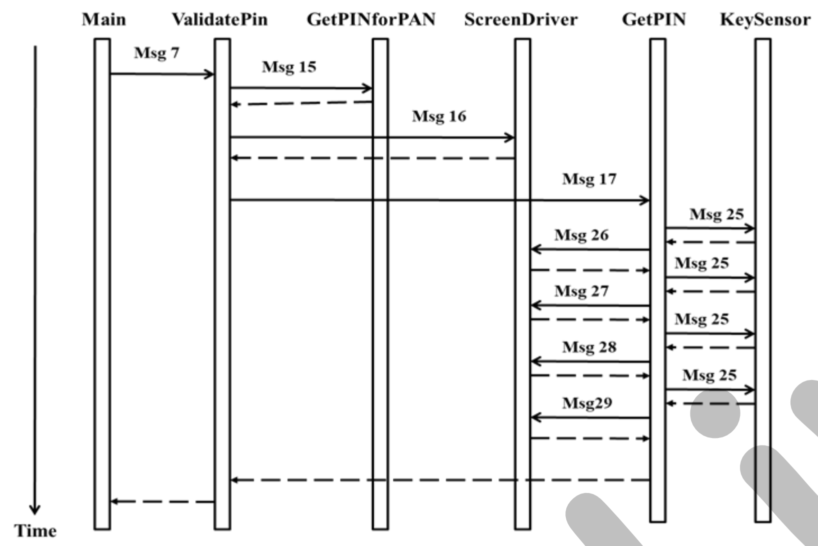
```
1.  Main Program
2.  State  =  AwaitCard
3.  Do              'Main Loop
4.  Case State
5.      Case 1: AwaitCard
6.              ScreenDriver (1, null)                          msg 1
7.              WatchCardSlot ( CardSlotStatus )                msg 2
8.                      Do while CardSlotStatus is Idle
9.                      Watch CardSlot (Card SlotStatus)        msg 3
10.             End While
11.             ControCardRoller (accept)                       msg 4
12.             ValidateCard (CardOK, PAN)                      msg 5
13.             If Card OK
14.                     Then State =AwaitPIN
15.                     Else ControlCardRoller(eject)           msg 6
16.             EndIf
17.             State=AwaitCard
18.     Case:2: AwaitPIN
19.             Validate PIN ( PINOK, PAN)                      msg 7
20.             If PINok
21.                     Then ScreenDriver (2, null)             msg 8
22.                             State =AwaitTrans
23.             ElseScreenDriver (4,null)                       msg 9
24.                             State = AwaitCard
25.             End If
26.     Case 3: AwaitTrans
27.             ManageTransaction                               msg 10
28.             State=CloseSession
```

```
  Case 4 :          Close Session :
                    If NewTransactionRequest
                          Then State = AwaitTrans
                          ElsePrintReceipt                                  msg 11
                    EndIf
                    PostTransactionLocal                                    msg 12
                          CloseSession                                      msg 13
                    Control Card Roller(eject)                              msg 14
                    State = AwaitCard
End Case (State)
Until ‘Forever
End ( Main program SATM)
Procedure Validate PIN ( PIN ok, PAN)
Get PIN for PAN ( PAN , Expected PIN)                                       msg 15
Try =First
Case Try of
  Case 1: First
   ScreenDriver (2, null)                                                   msg 16
   Get PIN ( Entered PIN, Cancel Hit)                                       msg 17
   If  Entered PIN = Expected PIN
         Then PIN ok =True
                    Else ScreenDrivet(3, null)                              msg 18
                    Try =Second
         End If
  Case : 2: Second
   ScreenDriver( 2, null)                                                   msg 19
   GetPIN (Entered PIN, Cancel Hit)                                         msg 20
   If Entered PIN= Expected PIN
         Then PIN ok =True
                    Else Screen Driver ( 3, null)                           msg 21
   End If
   Try=  Third
  Case: 3: Third
   ScreenDriver( 2, null)                                                   msg 22
         Get PIN(Entered PIN, Cancel Hit)                                   msg 23
   If EnteredPIN = ExpectedPIN
         Then PIN ok =True
         Else    ScreenDriver(4, null)                                      msg 24
                          PINok = False
   End If
End case (Try)
End       (Procedure Validate PIN)
```

```
71.  Procedure GetPIN (EnteredPIN , Cancel Hit)
72.  Local Data : Digit Keys = { 0,1,2,3,4,5,6,7,8,9,}
73.  Cancel Hit  = False
74.  Entered PIN = null string
75.  digitsRcvd  = 0
76.  Do While NOT(DigitsRcvd= 4 OR Cancel Hit)
77.    KeySensor(Key Hit)                                          msg25
78.    If Key Hit IN Digit Keys
79.     Then
80.                          EnterdPIN =  Entered PIN + Key Hit
81.                  digitRcvd =  digitsRcvd  + 1
82.                  If digitsRcvd =1
83.                    Then ScreenDriver (2, ‘X- - - ‘)                msg 26
84.                  End If
85.                  If digitsRcvd =2
86.                    Then Screen Driver (2, ‘ XX - - ‘)             :msg 27
87.                          End If
88.                          If digitsRcvd =3
89.                     Then ScreenDriver ( 2, ‘XXX - )                msg 28
90.                  End If
91.                  If digitsRcvd =4
92.                             Then ScreenDriver ( 2, ‘ XXXX’ )        msg 29
93.                  End if
94.          Else
95.                  CancelHit =True
96.     EndIf
97.  End While
98.  End ( Procedure GetPIN)
```

- SATM Main contains 16 source nodes. All except node 1 are where a procedure/function cal returns control: 1, 7, 8, 10, 12, 13, 16, 20, 22, 24, 28, 31, 33, 36, 37 and 38.
- SATM Main contains 16 sink nodes: 6, 7, 9, 11, 12, 15, 18, 19, 21, 23, 27, 32, 34, 35, 36 and 39.
- Most of the module execution paths in SATM Main are very short. This pattern is due to the high density of messages to other units.
- Only one nontrivial module execution path is contained in first 17 lines of SATM Main: <1, 2, 3, 4, 5>. Procedure calls such as <6>, <7>, <9>, <11>, <12> and <15> are trivial. Other very short module execution paths are associated with the control structures. For example, <10, 8>, <10, 11> and <16, 16>.
- Here is the MM-Path for a correct PIN entry on the first try. The module execution paths are described by giving the name of the unit followed by the sequence of the statement fragment numbers. The below Figure illustrates the sequential nature of an MM-Path using Unified Modeling Language (UML)-style sequence diagram.

Figure (b): UML sequence diagram of the sample MM-Path..



UML Sequence diagram of the sample MM-path.

Main (1, 2, 3, 18, 19)
Msg7
ValidatePIN (41, 42)

Msg15
GetPINforPAN (no pseudo-code given)
Validate PIN (43, 44, 45, 46)

Msg16
ScreenDriver (no Pseudo-code given
ValidatePIN (47)
Msg 17
Get PIN (71, 72, 73, 74, 75, 76, 77)

Msg 25
KeySensor (no Psedo-code given) 'first digit
GetPIN (78, 79, 80, 81, 82, 83)

Msg 26
GetPIN (84, 85, 87, 88, 90, 91, 93, 96, 97, 76, 77)

Msg 25
KeySensor (no psedo –code given) ' second digit
GetPIN (78, 79, 80, 81, 82, 84, 85, 86)

Msg 27
ScreenDriver (no psudo-code given)
GetPIN (87, 88, 90, 91, 93, 96, 97, 76, 77)

Msg25
KeySensor (no pseudo-code given) ‘third digit
GetPin (78, 79, 80, 81, 82, 84, 85, 87, 88, 89)

Msg 28
ScreenDriver (no psudo-code given)
GetPIN (90, 91, 93, 96, 97, 76, 77)

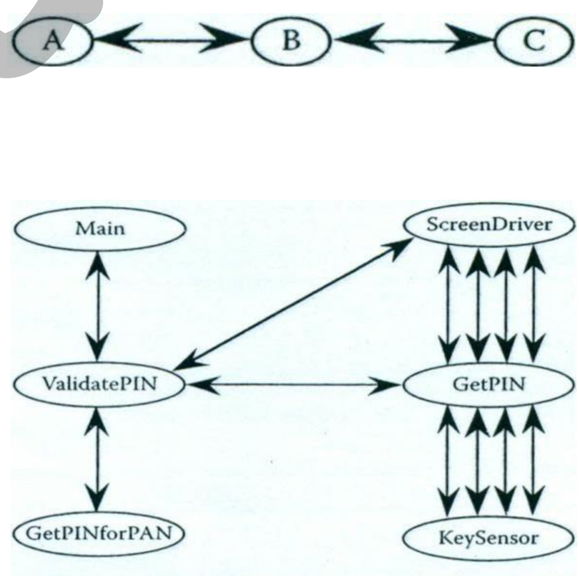
Msg 25
Keysensor (no psudo-code given) ‘forth digit
GetPIN(78,79, 80, 81, 82, 84, 85, 87, 88, 90, 91, 92)

Msg29
ScreenDriver (no pseudo-code given)
GetPIN (93, 96, 97, 76, 98)
ValidatePIN (48, 49, 52, 69, 70)
Main(20)

✓ MM-Path Complexity

- If you compare the MM-paths in Figures (a) and (b), it is clear that the latter is more complex than the former. Their directed graphs are shown together in below Figure. The multiplicity of edges preserves the message connections, and double-headed arrows capture the sending and return of a message.
- Because these are strongly connected directed graphs, we can “blindly” compute their cyclomatic complexities; recall the formula is V (G) = e – n + 2p, where p is the number of strongly connected regions.
- For structured procedural code, we will always have p=1, so the formula reduces to **V (G) = e – n + 2.** The results are **V (G) =3 and V (G) =20** respectively.

Figure: MM-Path directed Graph.



✓ Pros and Cons

- MM-paths are a hybrid of functional and structural testing. They are functional in the sense that they represent actions with inputs and outputs. The structural side comes from how they are identified, particularly MM-Path graph.
- The net result is that the cross-check of the functional and structural approaches is consolidated into the constructs for path-based integration testing. We therefore avoid the pitfall of structural testing, and, at the same time, integration testing gains a fairly seamless junction with system testing.
- Path-based integration testing works equally well for software developed in the traditional waterfall process or with one of the composition-based alternative life cycle models. Finally, the MM-path concept applies directly to object-oriented software.
- The most important advantage of path-based integration testing is that it is closely coupled with actual system behavior, instead of the structural motivations of decomposition and call graph–based integration.
- The advantages of path-based integration come at a price; more effort is needed to identify the MM-paths. This effort is probably offset by the elimination of stub and driver development.

Module IV

1. Explain different integration faults with example.
2. Explain in detail about Integration Testing Strategies.
3. Discuss briefly about Reusable components.
4. What is system testing? Explain.
5. Differentiate between unit, integration and system testing.
6. Explain in detail about acceptance testing.
7. What is usability testing? Which are the steps involved in process of verifying and validating usability.
8. What is regression testing? Explain Regression test selection techniques.
9. Explain in detail about test case prioritization and selective execution.
10. With a neat diagram, explain the traditional view of testing levels of waterfall-life cycle model
11. With a neat diagram, explain Waterfall Spin-offs models.
12. Explain Specification-Based Life Cycle Models with neat diagrams.
13. Explain data flow diagram, Upper level finite state machine and ER diagram with respect to the simple ATM application.
14. Give a decomposition tree for the SATM System along with the numbering scheme that preserves the levels of the components.
15. With SATM Units and Abbreviated Names explain SATM functional decomposition tree and SATM call graph
16. Explain the top-down integration and bottom-up integration with neat diagrams.
17. Explain the decomposition based integration with an example.
18. With a neat diagram explain Pair wise integration and Neighborhood Integration.
19. Explain in detail about MM-path graph.
20. With formula explain MM-Path complexity.

Module IV (VTU questions)

1. With a neat diagram, explain the traditional view of testing levels of waterfall-life cycle and rapid prototyping life cycles.
2. With an example, explain the top-down integration and bottom-up integration.
3. Explain the decomposition based integration with an example.
4. Explain traditional view of testing levels, alternative life-cycle models.
5. Explain in details, path-based and call graph-based integration with an example.
6. Explain the simple ATM application with the help of, (i) level 1 data flow diagram. (ii) Upper level finite state machine.
7. Distinguish between top - down integration and bottom - up integration.
8. Explain call graph-based integration with the help of, (i) pair-wise integration (ii) neighborhood integration
9. Explain in detail about acceptance testing and usability testing.
10. Explain the context diagram of SATM system.