Module 5

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 5 Study Guide and Deliverables

Module Topics:
- Topic 1: More UML Tools in Requirement Analysis and Design
- Topic 2: Testing Techniques

Readings:
- Online lecture notes
- Braude, Parts V and VII (or related chapters in other textbooks)

Discussions:
- Weekly Group Meeting

Assignments:
- Project Iteration 2 due **Tuesday, October 12 at 6:00 AM ET**
- Post-Project Iteration 2 Review due **Tuesday, October 12 at 6:00 AM ET**
- Weekly Report due **Tuesday, October 12 at 6:00 AM ET**

Live Classrooms:
- **Tuesday, October 5 from 7:00-9:00 PM ET**
- **Saturday, October 9 from 8:00-9:00 PM ET**

Learning Outcomes

By the end of this module, you will be able to do the following:

- Describe the differences and similarities between use cases and user stories.
- Draw use-case diagrams for your group project.
- Use state diagrams and sequence diagrams to help analyze and develop more complicated logics in your group project.
- Explain different code-coverage metrics and how to design test cases to improve the coverage.
- Develop more test codes and collect the code-coverage metrics from your group project.
- Apply risk-based domain-testing techniques to develop more test cases for your group project.

Introduction

In the previous modules, we discussed how to perform requirement analysis and design. In particular, we introduced user stories as the main method to present the requirements, and used class models to analyze the requirements. Then we discussed basic design goals and principles, as well as several architectural styles and design patterns, with a focus on MVC architecture and related patterns. In this module, we will introduce additional techniques and practices used in requirement analysis and design.

Topic 1: More UML Tools in Requirement Analysis and Design

In a previous module, we discussed the importance and challenges of requirement analysis. We introduced "user stories," which are commonly used in the Agile software-development framework to represent requirements. User stories provide flexibility in requirement management and drive the whole development process. We also introduced class diagrams, which can be used in requirement analysis and design.

In this module, we will discuss some other UML (Unified Modeling Language) tools. The UML tools were developed by the "three amigos" (Ivar Jacobson, Grady Booch, and James Rumbaugh) with the unified-process (UP) framework. They have been widely used as the industry-standard techniques for software built using object-oriented (OO) technology. UML provides a common vocabulary to describe OO components and concepts. UML 2.0 defines 13 diagrams, classified into 3 categories:

- **Structure Diagrams**—Class diagram, object diagram, component diagram, composite-structure diagram, package diagram, deployment diagram
- **Behavior Diagrams**—Use-case diagram (used in some methodologies during requirements gathering), activity diagram, and state-machine diagram
- **Interaction Diagrams**—Sequence diagram, communication diagram, timing diagram, and interaction-overview diagram

In the previous modules, we introduced one of the structure models, the class model, to represent the static structure of the system. This module will introduce some behavior diagrams and interaction diagrams to represent the dynamic behavior of the system—namely, use-case diagrams, state-machine diagrams and sequence diagrams.

Use-Case Model

Use-case models are used in traditional UP framework throughout the whole software-development life cycle. The use-case diagram is one of the behavior diagrams defined in UML and used extensively for requirement analysis in the last decades.

Similar to user stories, use cases describe the external behavior of a software system from an external agent's point of view; such agents are called "actors" in a use-case model. Actors can be either users or other external systems that interact with the target system. However, use cases are more rigid and structured, and contain more details. The use-case model has two important elements:

- Use-case diagram—Describes the relationship between actors and use cases, as well as system boundary.
- Use-case description—Each use case is described in text, using a specific format to describe all details—particularly how the actor interacts with the system.
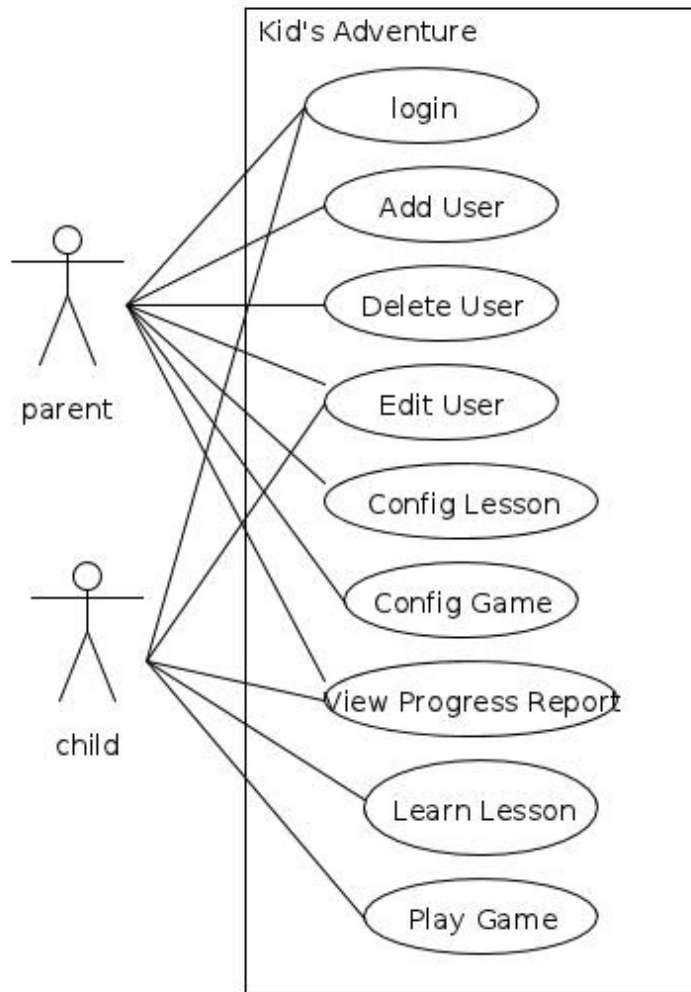
Use-Case Diagram

A use-case diagram usually consists of four components:

1. System boundary, represented as a rectangle box
2. Actors, who are external entities that interact with this system
3. Use cases, represented by ovals
4. The relationships between actors and use cases, represented by lines

The following is a use-case diagram of the Kid's Adventure software system. Kid's Adventure is an application developed by previous students to help small kids learn. The goal is to provide a configurable learning system for young kids. The app helps kids learn letters, basic words, and math. This application differs from other learning applications by allowing parents to configure the learning. Parents can customize the sound and pictures used in the lesson or game to make the learning experience more personal. For example, when teaching kids the word "elephant," parents can record the word with their own voice or even in different languages. They can also use different pictures, such as a photo taken by themselves in the local zoo.

The following use-case diagram shows these high-level functional requirements. It includes two actors, parent and child, and a number of use cases showing what actors can do with the software. From the use-case diagram, we can see that both parent and child can log in, edit their user profiles, and view a progress report. Children can learn lessons and play games, while parents can add or delete child users and configure the game and lesson. The rectangle box presents the system boundary, which shows that anything outside is an external component. The lines between actors and use cases show which actors interact with which use cases. Compared with user stories, a use-case diagram helps provide a big picture of the whole system's requirements, but is less flexible to manage.

Use-Case Text Description

Each use case in the use-case diagram needs to be described in detail. Here, we show the description of the "Add User" case:

- Name: Add User
- Participator Actor: Parent
- Entry Condition: Parent successfully logs in
- Exit Condition: New user added or error message
- Flow of Events:

    1. Parent clicks the "Add User" button
    2. System displays an "Add User" form
    3. Parent fills in the username and age and chooses a picture for the child user
    4. Parent clicks the "Submit" button
    5. System displays either a message to show that the child user has been successfully added or an error message

There are several sections in this use-case description. The title of a use case is usually a verb phrase in active and present tense. The entry-condition section defines the conditions before invoking this use case, and the exit-condition section defines the status after completing this use case. The flow-of-events section details the interactions between the actor and the system. Usually, the actor initiates an action and the system responds, and then they interact back and forth to complete a particular task.

Here is another example. Consider a bank customer who uses an ATM to withdraw money.

- Name: Withdraw money using ATM
- Initiating Actor: Bank customer
- Entry Condition:
    - Bank customer has opened a bank account with the bank and
    - Bank customer has received an ATM card and PIN
- Exit Condition:
    - Bank customer has the requested cash or
    - Bank customer receives an explanation from the ATM about why the cash could not be dispensed
- Flow of Events:

Actor Steps	System Steps
1. The bank customer inputs the card into the ATM. 3. The bank customer types in the PIN. 5. The bank customer selects an account. 7. The bank customer inputs an amount.	2.The ATM requests the input of a four-digit PIN. 4. If several accounts are recorded on the card, the ATM offers a choice of the account numbers for the bank customer to select from. 6. If only one account is recorded on the card or the bank customer has completed the selection, the ATM requests the amount to be withdrawn. 8. The ATM outputs the money and a receipt and stops the interaction.

In both examples above, only the basic flow is considered. In many cases, there may be alternative flows. In particular, the basic flow (or the prime flow) usually describes the successful case, and the failure cases or exceptions are described in the alternative flows. For example…:

Actor Steps	System Steps

Actor Steps	System Steps
1. The bank customer inputs the card into the ATM. [Invalid card]	2. [Invalid card] The ATM outputs the card and stops the interaction.
3. The bank customer types in the PIN. [Invalid PIN]	4. [Invalid PIN] The ATM announces the failure and offers a 2nd try as well as canceling the whole use case. After 3 failures, it announces the possible retention of the card. After the 4th failure it keeps the card and stops the interaction.
5. The bank customer selects an account.	
7. The bank customer inputs an amount. [Amount over limit]	8. [Amount over limit] The ATM announces the failure and the available limit and offers a second try as well as canceling the whole use case.

Alternative flow (or exception flow)

1a. Customer inserts an invalid card. 2a. The ATM outputs the card and stops the interaction.

3a. Customer inputs an invalid pin. 4a. The ATM announces the failure and offers a second try, as well as the option of canceling the whole use case. After three failures, it announces the possible retention of the card. After the fourth failure, it keeps the card and stops the interaction.

5a. Customer inputs the amount that exceeds the limit. 6a. The ATM announces the failure and the available limit and offers a second try, as well as the option of canceling the whole use case.
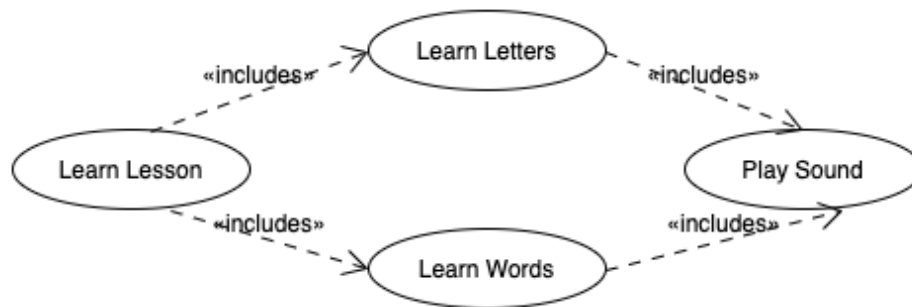
Relationships

Use cases can be related. The use-case diagram shows not only the actors and all of the use cases in the system, but also the relationships between them. There are several types of relationships:

- **Communication relationship**—This is a general relationship between actors and use cases to represent the information exchange between them. It is represented by a solid line.
- **Includes relationship**—This relationship can be used to factor out functional behavior common to more than one use case. It can also be used to break complex use cases into several use cases. In general, if a use-case description is longer than one page, it may be too complicated and should be split into multiple use cases. It is represented by an arrowed, dotted line with the "includes" notation, and the arrow pointing to the use cases that are included. The including use-case text description should clearly mention the included use cases.

- **Extends relationship**—This relationship can be used to represent seldom-invoked use cases or exceptional functionality. It is represented using an arrowed, dotted line with the "extends" notation, and the arrow pointing to the use cases that are extended. The extending use-case text description should clearly mention the extended use cases.
- **Inheritance relationship**—This relationship can be used between actors or between use cases. One use case can specialize another, more general one, or one actor can specialize another general actor.

The following diagram shows how to use the "includes" relationship to break down the use case "Learn Lesson" into two use cases: "Learn Letters" and "Learn Words." In addition, we can factor out a common use case, "Play Sound," from both use cases. In the "Learn Lesson" text description, we need to clearly mention two included use cases. In each of these two use-case text descriptions, we need to mention the "Play Sound" use case.

The following example shows the usage of both "includes" and "extends" relationships. For example, the "Exceed File Size" use case is an exceptional use case that can extend both the "Change Pictures" and "Change Sound" use cases. In its text description, it should clearly specify which use case(s) it extends. In addition, the "Incorrect Image Resolution" can be another exceptional use case to extend the "Change Pictures" use case.
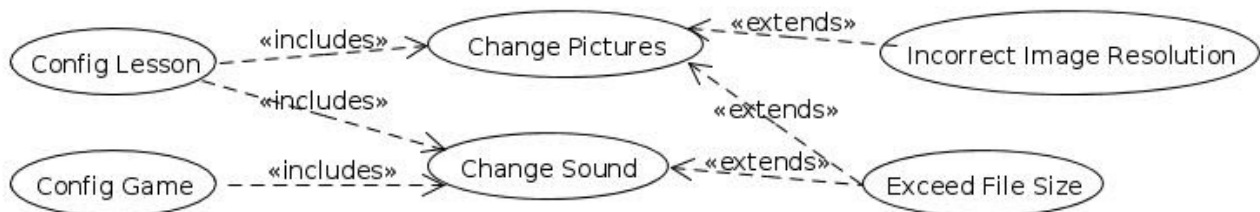
In Module 2, we introduced the ProjectPortal application. The use-case diagram below shows the functional requirements of ProjectPortal. Here, we use the inheritance relationship between actors. "Project Viewer" can log in, log out, edit user profile, view projects, and rate projects. "Project Owner" inherits from "Project Viewer" and thus can perform all of the above use cases that "Project Viewer" can do. In addition, "Project Owner" can create, edit, and delete projects.
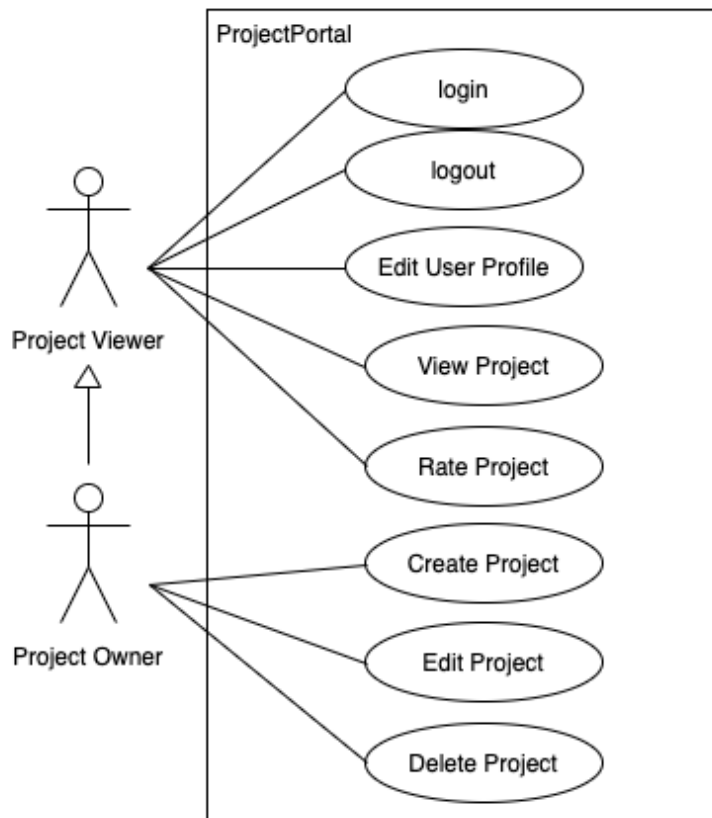
ProjectPortal Use-Case Diagram

> **Test Yourself**
>
> Please write the detailed text description for the "Add Project" use case in the ProjectPortal project.

Use Cases vs. User Stories

Are the use-case model and the user-story model the same or different? Is the user-story model just a simplified version of the use-case model? People have different opinions. Some people, such as Martin Fowler, think the two are essentially the same. Others, such as Alistair Cockburn, totally disagree.

Here is what Kent Beck said about user cases and user stories:

> I coined the term UserStory, as far as I know, so I'll tell you what I had in mind. My purpose is to maintain a balance of political power between business and development. Use cases as I have seen them used are complex and formal enough that business doesn't want to touch them. This leads to development asking all the questions and writing down all the answers and taking responsibility for the resulting corpus. Business is reduced to sitting on the other side of the table and pointing.

I want a very different dynamic. I want business to feel ownership of and take responsibility for the care and maintenance of "the requirements." I want business to feel comfortable making priority decisions about the requirements. I want businesses to feel free to add new requirements, and add new detail to existing requirements, as development progresses (see also ProgrammingIsSocialLearning).

This requires a form of expression that is more approachable than a formalized use case. It also helps if the communication medium is something approachable, like IndexCards. So I say, 'Tell me the stories of what the system will do. Write down the name of the story and a paragraph or two.'

My experience is that business, properly trained, takes to managing stories like the proverbial duck to the equally proverbial water. Business has to be trained not to just throw new stories into the CommitmentSchedule or WorkQueue without a DevelopmentEstimate and the necessary reshuffling. Development has to be trained to begin examining stories enough ahead of IterationPlanning so learning the next level of detail does not become a bottleneck or a risk.

So, to answer your first question, yes and no. The idea of specifying the behavior of the system from an outside perspective, and using those specifications throughout the life of the system is the same. The execution is quite different. Comments, Alistair, oh guru of use cases?

You can find other people's opinions at [User Story and Use Case Comparison](#).

The use-case model was formulated by Ivar Jackson in 1986 to be used in UP (unified process). It is a formal, complex, and detailed approach to requirement analysis. "User-story model" was coined by Kent Beck and used in Agile frameworks, such as XP and Scrum. It is a more informal, simpler, and more flexible approach than the user-case model. Both are used to describe the behavior of the system, from an outside perspective, throughout the life of the system. Both are functional models for high-level requirements.

Class Diagrams

The developer tea will then analyze the high-level requirements described by either use cases or user stories, usually using an object-oriented analysis. In the previous modules, we discussed how to use the entity-boundary-control pattern to analyze the requirements and generate an object model.

- **Entity**—Represents the persistent information tracked by the application
- **Boundary**—Represents the interfaces between the user and the application
- **Control**—Represents the control tasks (actions) performed by the application

Below, we show the "Add User" use-case description in the Kid's Adventure project from the previous section:

- Name: Add User

- Participator Actor: Parent
- Entry Condition: Parent successfully logs in
- Exit Condition: New user added or error message
- Flow of Events:
    1. Parent clicks the "Add User" button
    2. System displays an "Add User" form
    3. Parent fills in the username and age and chooses a picture for the child user
    4. Parent clicks the "Submit" button
    5. System displays either a message to show that the child user has been successfully added or an error message

Based on this description, we can identify the following entity, boundary and control objects:

1. **Entity**—System, user, parent
2. **Boundary**—"Add User" button, "Add User" form, "Submit" button
3. **Control**—addUser

We can have the following class diagram:
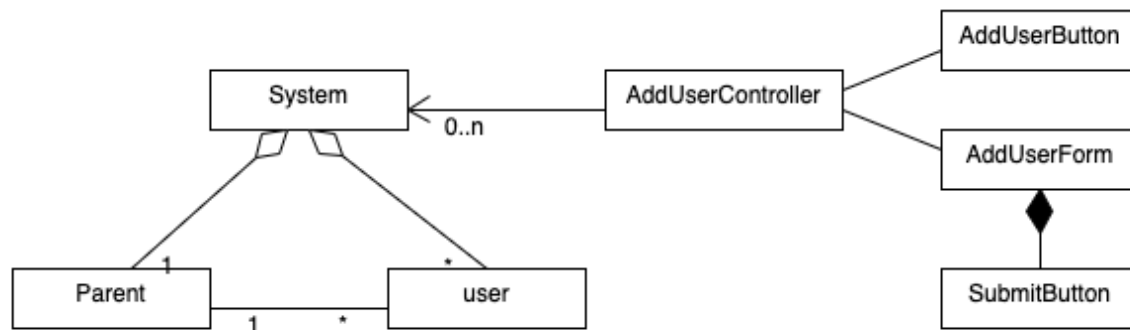
The class/object diagrams are commonly used to provide a structure model of the detailed analysis of the application-domain objects. However, it is a static model. It cannot describe the dynamic behavior of these objects. In this section, we will introduce two dynamic models: state-machine diagrams and sequence diagrams.

- State-machine diagrams are used to model the dynamic behavior of a single object.
- Sequence diagrams are used to model the dynamic behavior between objects.

State-Machine Diagrams

In the previous modules, we briefly introduced state-machine (also called state-transition) diagrams and shown how to use simple state-transition diagrams to model the UI screen transition of a software system.

Actually, state diagrams are more often used to show the dynamic behavior, the state transition, of a single object. A state-machine diagram describes all of the states that an object can have, the events under which the object changes state (transitions), the conditions that must be fulfilled before the transition will occur (guards), and the activities undertaken during the life of the object (actions). Consider the example of a commonly used vending machine. The user inserts the money and makes a selection, and then the vending machine dispenses the item. The following state-transition diagram can be used to model the dynamic behavior of this vending machine:
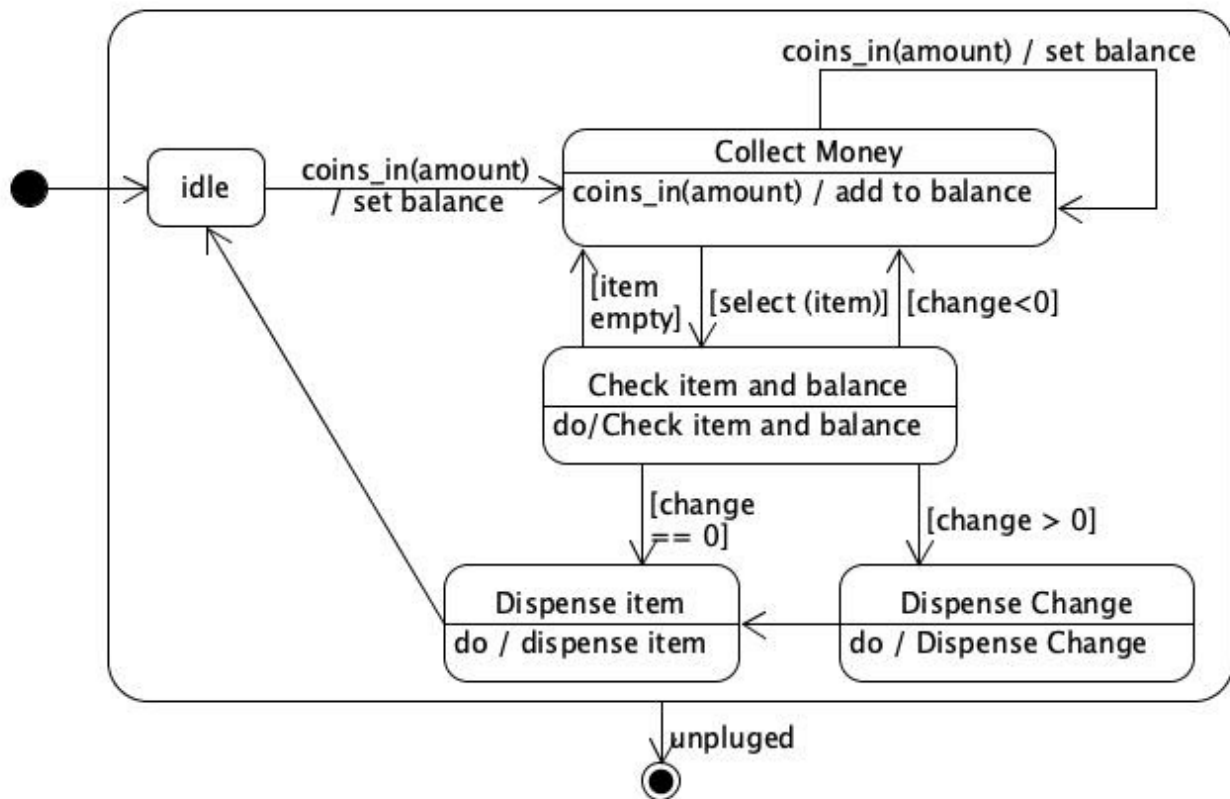

In the state-transition diagram, each node is a state, drawn as a rounded box. The directed arcs between nodes are transitions labeled with event names. The state transition is triggered by certain events, or conditions.

Each state-transition diagram can only have one initial state, which is the state of a new object, immediately following its creation. In this example, the initial state is the idle state, in which the vending machine does nothing. When someone inserts money, it goes into a different state, in which it keeps collecting money and calculating the balance. The transition can take place not only between states, but also in the same state. When the actions (or events) do not trigger the state change, the self-transition happens. When the vending machine is in the "Collect Money" state, it will stay in the same state when more money is inserted. When the user makes an item selection, the vending machine changes to another state, in which it checks whether the selected item is available and whether the inserted money is enough to purchase it. If the money is not enough, it goes back to the "Collect Money" state to collect more money. If the item is not available, it goes back to the "Collect Money" state, too. If the inserted money is just right to purchase the selected item, the vending machine transitions to another state, "Dispense the Item," to dispense the selected item. Afterward, it goes back to idle. If the inserted money is more than is needed, the change is dispensed first, and then the item is dispensed. At any state, if the

vending machine is unplugged, then it goes to the final state, in which the vending machine is not working anymore.

From this example, we can see how the state-transition diagram can be used to model the dynamic behavior of a single object. The above state-transition diagram doesn't consider using a credit card and doesn't allow the user to cancel the transition in the middle. Can you try to modify the diagram to support using a credit card and canceling the transaction?

Let us look at another example. Suppose we have an alarm clock, as shown below. The alarm clock has a time-display screen and several buttons. To set the time, press the "Set" button and then press "M" or "H" to increase the minute or hour in the current display time. After you are done, press the "Set" button again. To enable the alarm, press the "Alarm" button. To disable the alarm, press the "Alarm" button again. The red light on the display will blink when the alarm is up, and there will be an alarming sound. It will last for one minute, or you can press the "Alarm" button to disable it. To set the alarm time, when the alarm is enabled, press the "Set" button then press "M" or "H" to increase the minute or hour. After you are done, press the "Set" button.
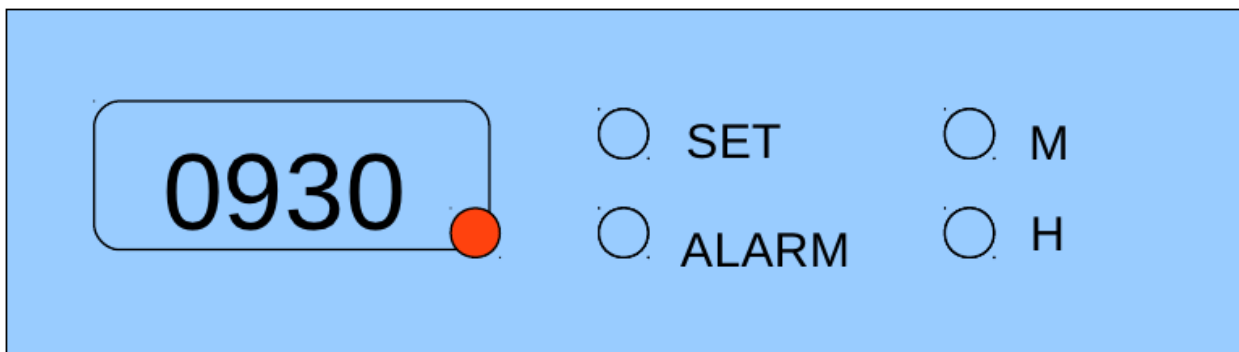
Based on the above description, we can see that the alarm clock engages in different behaviors when you press the same button, depending on its current state. Therefore, we can use a state-machine diagram to describe this dynamic behavior in a visual way, helping us better understand the problem.

To draw a state-machine diagram, we need to answer following two questions:

1. How many states does this alarm clock have? What are they?
2. How does the alarm clock transition from one state to another? What conditions or actions can trigger the transition?

Test Yourself

Can you identify all states of this alarm clock?

After carefully analyzing the problem, we can identify five states:

Test Yourself

Can you draw the state-machine diagram for this alarm clock?

In this diagram, we also show a possible end state for the alarm clock when the clock is out of battery:

State Pattern

The state pattern is often used when an object's behavior changes according to its states. In this alarm-clock example, when the "Set" button is pressed, the alarm clock behaves differently in various states. When it is in the "Time Display" state, it transitions to the "Time Setting" state after the "Set" button is pressed. If it is in the "Alarm Enabled" state, it then transitions to the "Alarm Setting" state. Similarly, when we press the hour or minute button, the alarm clock also behaves differently. If it is in the "Time Display" state, it then does nothing. But if it is in the "Time Setting" state, the display time is changed. If it is in the "Alarm Setting" state, the alarm time is changed.

Instead of using the "if-else" or "switch" selection statement in each of these cases, the state pattern is commonly used to encapsulate varying behavior for the same object, based on its internal state. By decoupling the state and the corresponding behavior from the object itself, we improve the maintainability and extensibility. It is then much easier to add another new state and change the current state behavior.

The following class diagram shows the idea of a state pattern. A separate, abstract class (or interface), State, is used to decouple the internal states from the context using a composition relationship. When the *Request()* method of the context is called, it delegates to the *Handle()* method in the State class, and then to the *Handle()* method of a *ConcreteState*, depending on the current state. This is again implemented using specification inheritance and polymorphism.
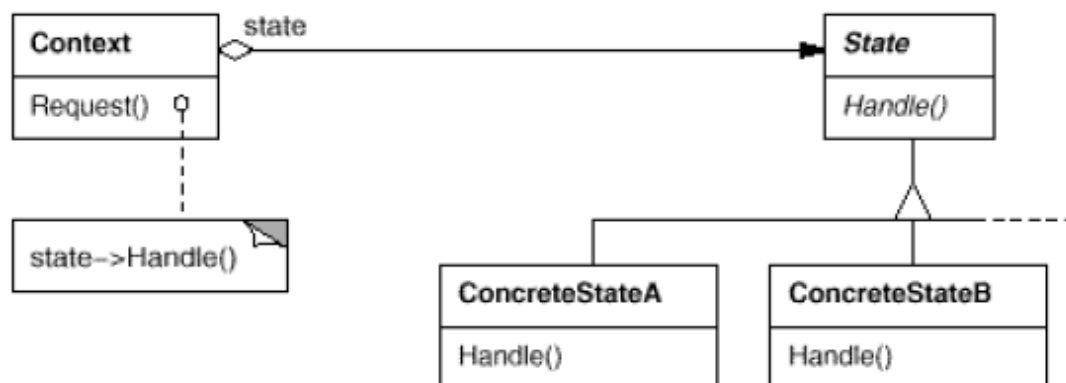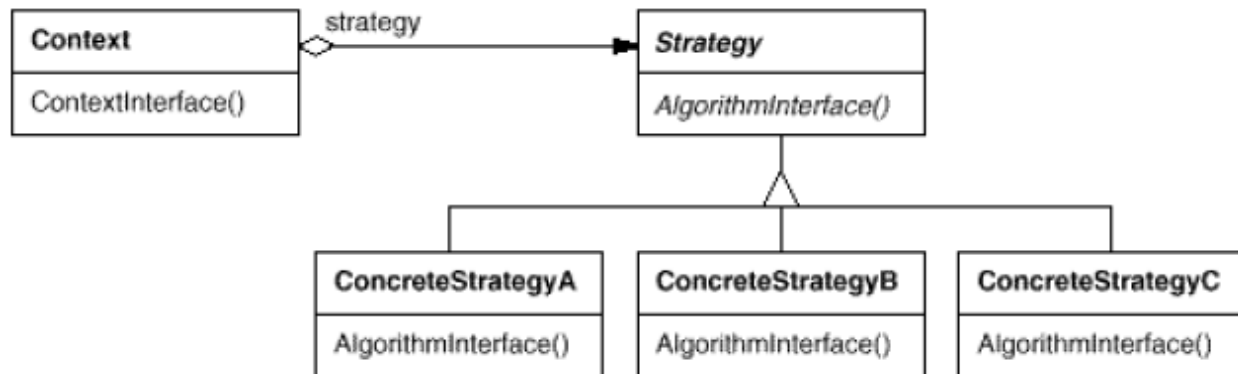

State-Pattern Class Diagram

Difference between the state pattern and the strategy pattern

You may find that this class diagram is quite similar to the strategy pattern we discussed before. While both use composition to decouple and specification inheritance and polymorphism to support dynamic behavior changes,

the main difference is how the behavior changes dynamically. In the state pattern, the behavior changes based on the object's internal state, and the action, in turn, triggers the changes of internal state. For example, the alarm clock behaves differently when the "Set" button is pressed, depending on its current state, and the action of pressing the "Set" button also triggers the state changing to another. Contrarily, in the strategy pattern, the strategy (or algorithm) is dynamically changed based on the **external policy or context, not the internal state**.

State-Pattern Class Diagram

Using the state pattern in the alarm-clock example, we can have the following class diagram:
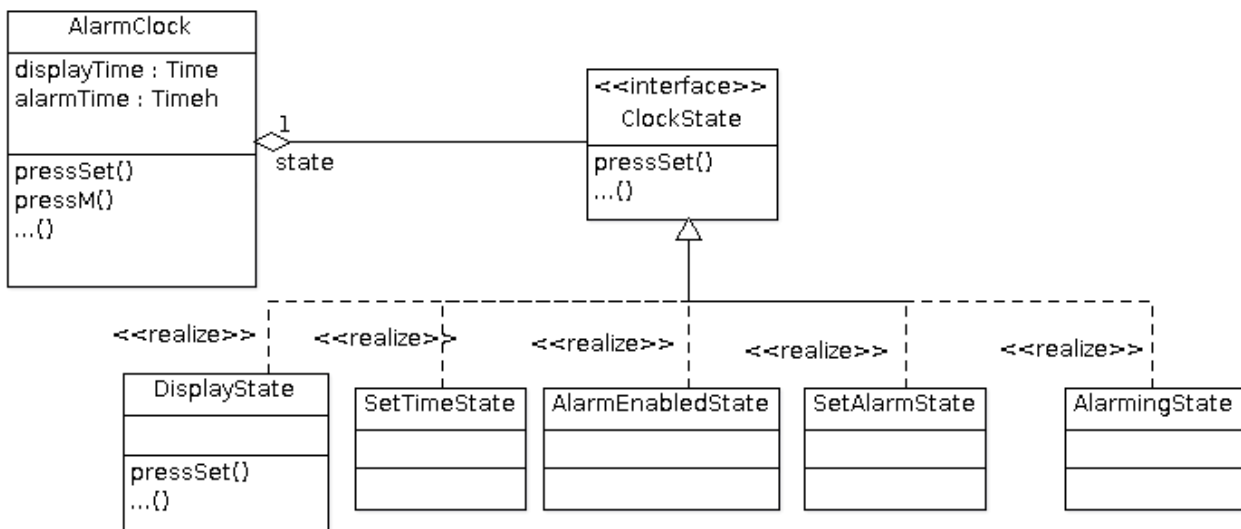
Alarm-Clock Class Diagram Using State Pattern

A possible implementation using switch statements without using the state pattern:

```
public class AlarmClock
{
    Time displayTime, alarmTime;
    int state;
    //0: displayState, 1: setTimeSate
    //2: alarmEnalbedState, 3:setAlarmState
    //4: alarmingState
    public void pressSet()
    {
```

```java
            switch(state)
            {
             case 0: state = 1;break;
             case 1: state = 0;break;
             case 2: state = 3;break;
             case 3: state = 2;break;
             default: break;
            }
        }

        public void pressH()
        {
            if (state == 1) {
                    displayTime.increaseHour();
            } else if (state == 3)
                    alarmTime.increaseHour();
            }
        }
    }
```

Implementation using the state pattern:

```java
        public interface ClockState {
                public abstract void pressSet();
                public abstract void pressM();
                public abstract void pressH();
                public abstract void pressAlarm();
        }

        public class DisplayState implements ClockState{
                AlarmClock alarmClock;

                public DisplayState(AlarmClock clock)
                {
                        alarmClock = clock;
                }

                @Override
                public void pressSet() {
                        alarmClock.changeState(alarmClock.getSetTimeState());
                }

                @Override
                public void pressM() {
                        //doing nothing
                }
                // Todo: implement other methods

        }

        public class SetTimeState implements ClockState{
                private AlarmClock alarmClock;

                public setTimeState(AlarmClock clock)
                {
                        alarmClock = clock;
                }
```

```java
                @Override
                public void pressSet() {
                        alarmClock.changeState(alarmClock.getDisplayState());
                }

                @Override
                public void pressM() {
                        alarmClock.incMDisplayTime();
                }
                // Todo: implement other methods

        }
        //Todo: implement states

        public class AlarmClock {
                private Time displayTime, alarmTime;


                private ClockState displayState = new  DisplayState(this);
                private ClockState setTimeState = new SetTimeState();
                private ClockState setAlarmState = new SetAlaramState();
                private ClockState alarmEnabledState = new AlarmEnabledState(this),
                private ClockState alarmingState = new AlarmingState(this);
                private ClockState curState = displayState;



                public Time getDisplayTime() {
                        return displayTime;
                }

                public Time getAlarmTime() {
                        return alarmTime;
                }

                public ClockState getDisplayState() {
                        return displayState;
                }

                public ClockState getSetTimeState() {
                        return setTimeState;
                }

                public ClockState getSetAlarmState() {
                        return setAlarmState;
                }

                public ClockState getAlarmEnabledState() {
                        return alarmEnabledState;
                }

                public ClockState getAlarmingState() {
                        return alarmingState;
                }

                public ClockState getCurState() {
                        return curState;
                }

                public void pressSet()
```

```
        {
                curState.pressSet();
        }

        public void pressM()
        {
                curState.pressM();
        }

    //Todo: implement other methods


    }
```

Sequence Diagrams

Sequence diagrams illustrate dynamic interactions among objects. They complement the class diagrams, which represent only static structures. A sequence diagram can be used in both the requirement-analysis and design phases. It helps the developers understand requirements better and identify missing objects. It usually starts from the event flow described in the use-case text description and provides additional details of how things work from the developers' point of view.

Here are some basic elements used in the sequence diagrams defined in UML 2.5:

Lifeline—Lifetime of a participating object
ExecutionSpecification—A thin rectangle on the lifeline that represents the period of an action performed
Synchronous message (synchronous call) between objects
Asynchronous message between objects
Return message
Foundmessage (sender not specified)

At the top of the diagram, the participating objects are listed from left to right, horizontally. The involving objects can be identified using the textual analysis from the requirement description and the entity-boundary-control pattern, usually in the order of boundary objects, control objects, and entity objects. The vertical axis shows the timeline in order to show the time order of various events. The interaction between objects is through message passing. In the object-oriented languages, a message is usually implemented by triggering a method call of the receiving object. The types of messages that can be sent are limited by the methods defined in the receiving

object. There can be two types of messages: synchronous and asynchronous. Synchronous message passing requires the sender and receiver to wait for each other while transferring the message. In asynchronous communication, the sender and receiver do not wait for each other and can carry on their own computations while messages are being transferred.

Usually, the actor initiates some action by sending messages to a boundary object, and then the boundary object triggers the control object's performance of certain actions. In return, it sends messages back to the boundary object to update the display, or to the entity objects to update the application date.

Let us take a look at the sequence diagram for the above *addUser* use case. It starts with clicking the "Add User" button (by the parent actor, which is not specified in this diagram) and then displays the "Add User" form. After the form is filled, the "Add User" controller will perform add user action by creating a user object and adding it to the database. After it is done, the controller returns a message to the "Add User" form to indicate whether it was successful.
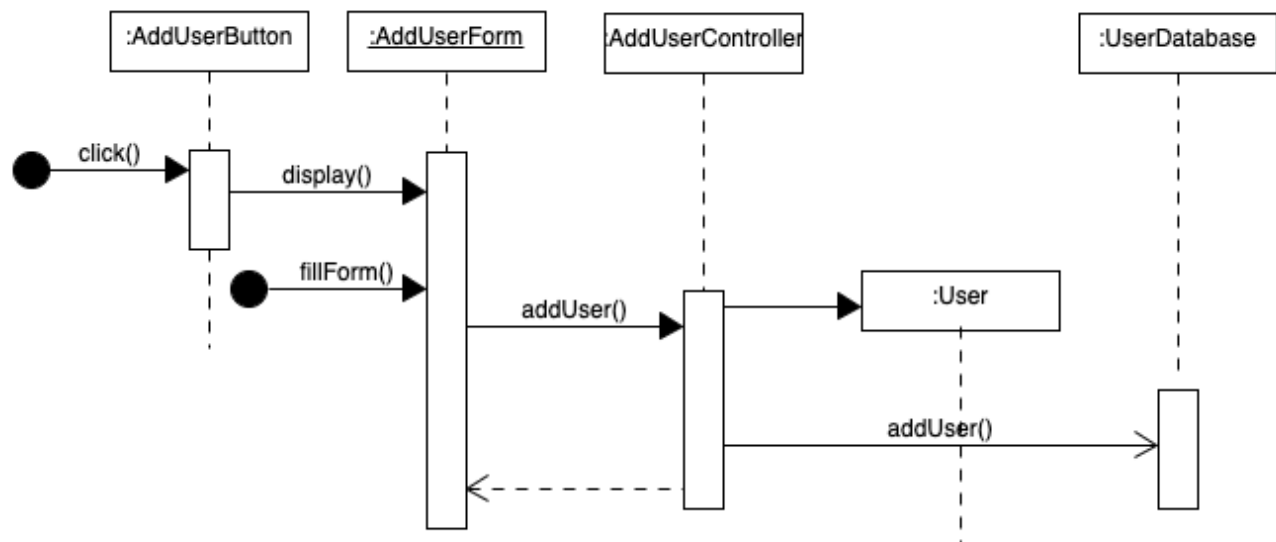
Sequence diagrams can also model the loop and the selection using combined fragments.

Loop	Selection
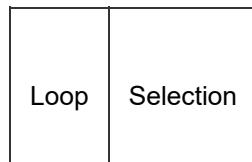
For example, we can use an alternative combined fragment to show different return values, depending on whether the user can be created successfully.
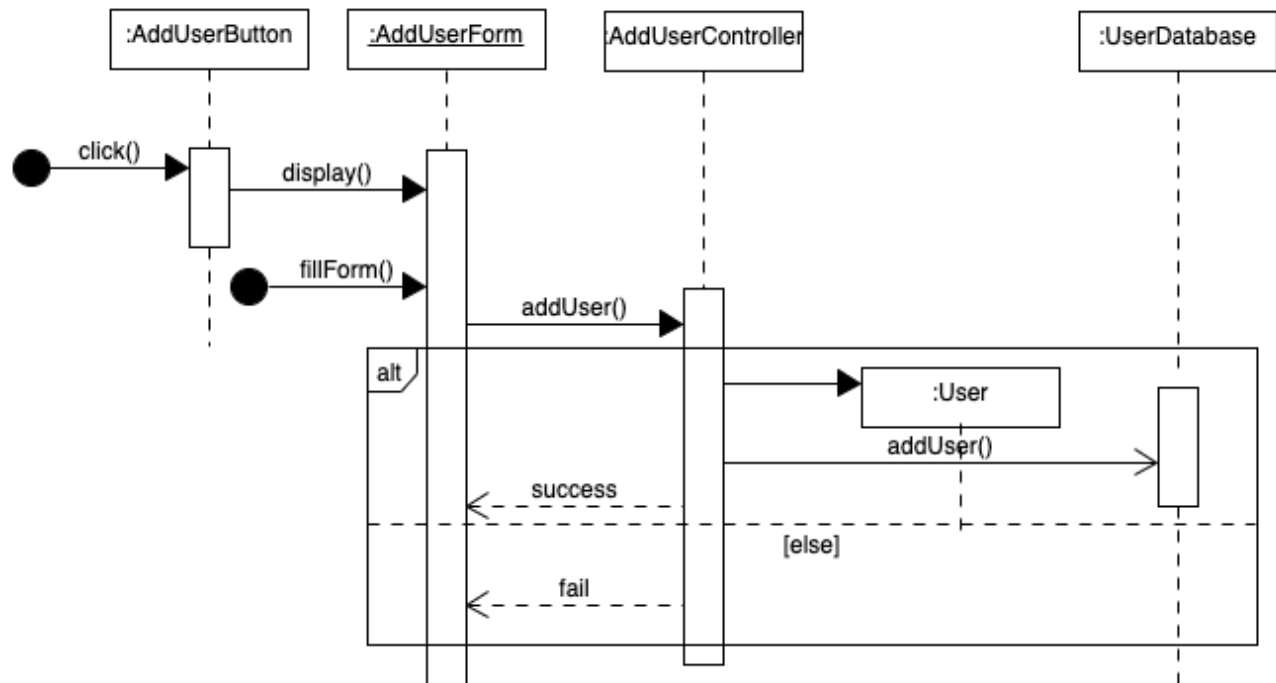
A sequence diagram is also used in the design to visualize the complex logic. There are also tools that can generate sequence diagrams directly from code. As Agile values working software more than documentation, the value of using these tools is to make the requirement analysis and design easier to understand through visualization and, in return, reduce possible errors.

Test Yourself

Test Yourself

Use case model is a _____ model.

✓

Functional

Structural

Test Yourself

For the following question, choose the answer that has the words in the correct order to correctly fill in the blanks.

In use-case diagrams, the _____ relationship is used to split a complex use case into multiple sub–use cases. The _____ relationship is used for exception handling. The _____ relationship is

used for inheritance.

generalization, includes, extends

includes, extends, generalization

extends, generalization, includes

includes, generalization, extends

Test Yourself

In a use-case text description, the *flow-of-events* section should clearly describe

How the actors interact with the system.

Test Yourself

The _____ pattern is often used with the state-transition diagram when an object's behavior changes according to its states.

State

Test Yourself

What is the difference between the state pattern and the strategy pattern?

Test Yourself

What is the purpose of sequence diagrams?

To illustrate the dynamic interactions among objects.

Test Yourself

In a sequence diagram, what is the difference between synchronous and asynchronous messages? A line with a solid triangle head represents a(n) _____ message, and a line with a simple arrowhead represents a(n) _____message.

Synchronous message passing requires the sender and receiver to wait for each other while transferring the message. In asynchronous communication, the sender and receiver do not wait for

each other and can carry on their own computations while messages are being transferred. Synchronous, asynchronous.

Test Yourself

Consider the following scenario of posting a thread in the discussion forum in the Blackboard system:

John is a student in CS673. He has a question about sequence diagrams when he is studying this topic, so he posts the question in the "Ask the Facilitators" forum. Write the detailed text description for the "Post a Thread in the Discussion Forum" use case. Identify all objects in your description and draw a sequence diagram to show the object interactions.

Topic 2: Testing Techniques

In the last module, we discussed the importance of testing and the TDD and BDD practices. In this module, we will focus on detailed testing techniques, particularly how to design test cases to maximize value while reducing cost.

White-Box Testing

White-box testing is a technique to design test cases based on the system's internal design and code. It is also known as clean-box testing. The term "white box" was used because of the see-through box concept. The "clear box" or "white box" name symbolizes the ability to see through the software's outer shell (or "box") to its inner workings. Unit testing is an example of white-box testing, performed by the developers to test single units. White-box testing techniques can also be used at the integration and system levels. The main goal of white-box testing is to test the internal behavior and structure of the software system, not just the external behavior.
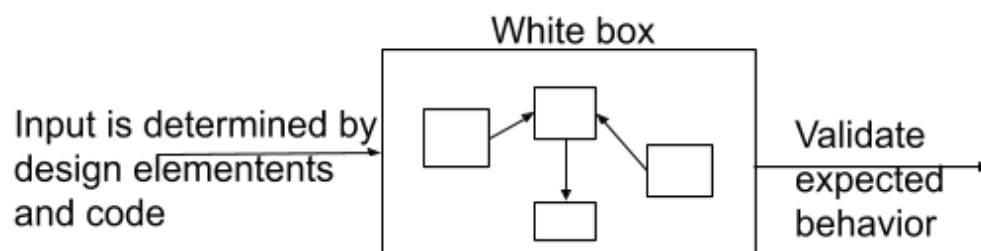
White-Box Testing

With visible design and implementation, white-box testing can do the following:

- Test all code statements

- Check the use of all called objects

- Verify the handling of all data structures

- Verify the handling of all files

- Check all paths, including both sides of all branches

- Check normal termination of all loops

- Check abnormal termination of all loops

- Check normal termination of all recursions

- Check abnormal termination of all recursions

- Verify the handling of all error conditions

- Check timing and synchronization

- Verify all hardware/platform dependencies

Code coverage is one of the common metrics used to measure how much code is executed under testing. Based on the code structure, code coverage can be categorized into the following buckets:

- **Statement coverage**—Percentage of code lines executed by a set of tests
- **Method coverage**—Percentage of method calls executed by a set of tests
- **Branch coverage**—Percentage of branches executed by a set of tests
- **Condition coverage**—Percentage of conditions executed by a set of tests
- **Path coverage**—Percentage of paths executed by a set of tests

There are a number of tools that can automatically measure the coverage, such as EclEmma, codePro, codecoverage.py Python CodeCoverage, and Codepro Analytix.

Example 1 - Character Classification

How many test cases are needed to cover all statements, all branches, and all paths, respectively? What are they?

```
if (input is in AllowedCharacterSet)
      if (input is a number)
            if (input >= 0)
                  put input into positiveNumberList
            else
                  put input into negativeNumberList
      else
            if (input is an alphabet)
                  put input into alphabetList
            else
                  put input into symbolList
else
      exception("Illegal character")
```

The above code snippet is a simple nested-if structure with only one input. We can easily come up some test cases, such as the following:

Test Case #	Input
1	a
2	1
3	!

Test Yourself

What is the minimal number of test cases we need to achieve 100% of code coverage, such as line coverage?

It is not trivial to find out the minimum number of test cases needed to achieve 100% coverage, even for this simple code snippet. A good way to solve this problem is to illustrate the code using a flowchart, as shown below:
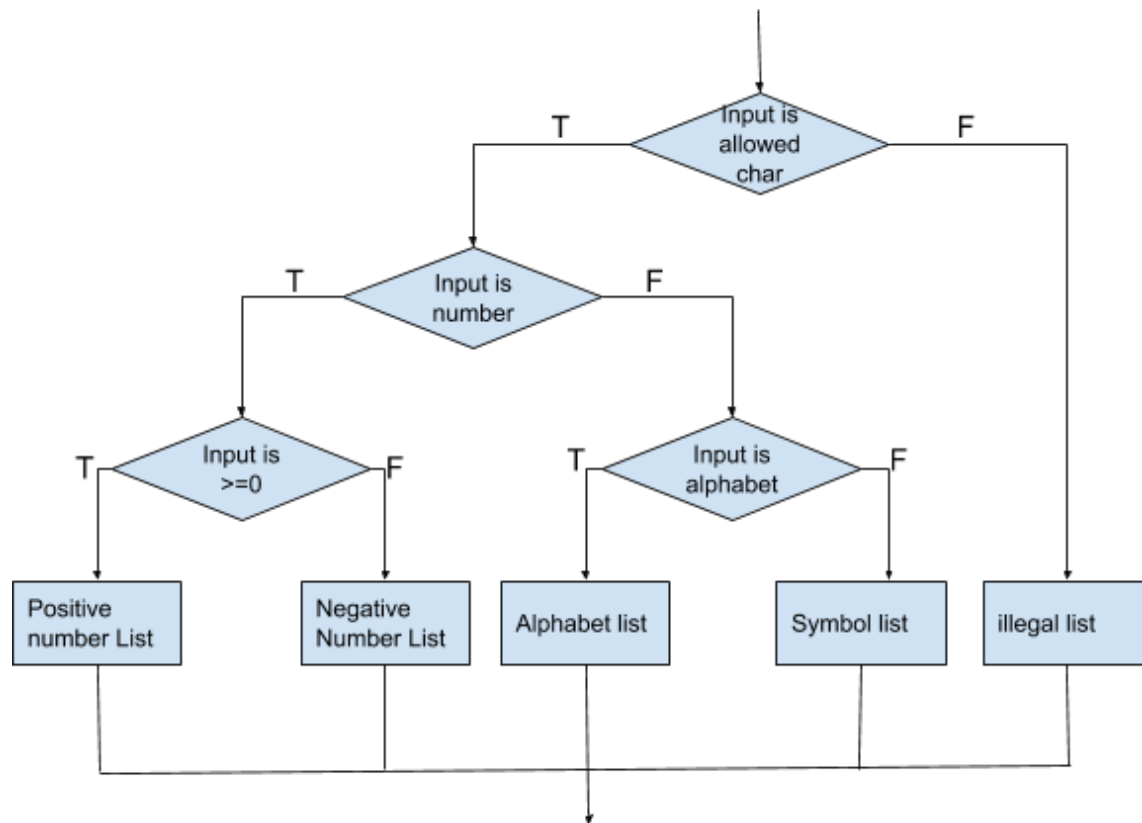
Flowchart for Example 1

From this flowchart, we can easily see that there are five execution paths. To execute all five paths, we will need at least five test cases. In addition, each path contains some statements and branches that are not covered in other paths. Therefore, we will also need at least these five test cases to cover all statements and branches. These five test cases follow:

Test Case #	Input	Example Data
1	Any positive number	5
2	Any negative number	-5
3	Any alphabet letter	a
4	Any symbol	!
5	Any disallowed character	ctrl

In this example, if the goal is only to achieve the full code coverage, we can choose any character in each category. For example, positive number 5 or 50 or even 0 will yield the same coverage. It will cover the leftmost path. In the next section, we will discuss how to choose the most representative option in each test-case partition.
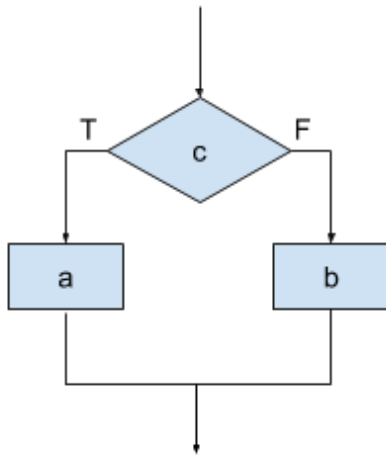
Flowchart Basics

A flowchart is a type of diagram that represents a workflow or process. It has the following basic components:
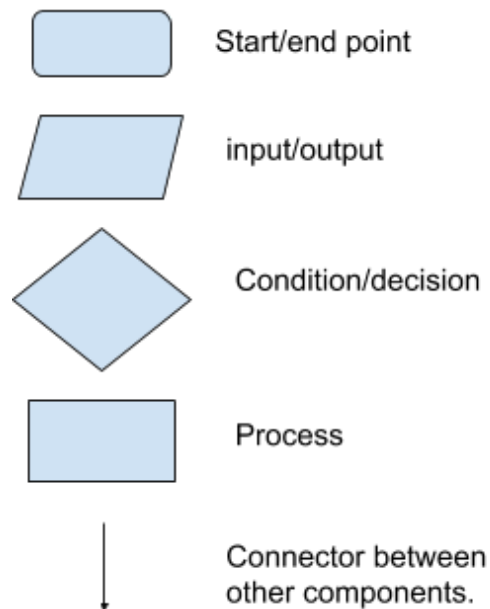
Basic Components of Flowcharts

With these basic components, a flowchart can illustrate different structural code blocks. The following diagram shows how to represent the selection code block and the loop code block:
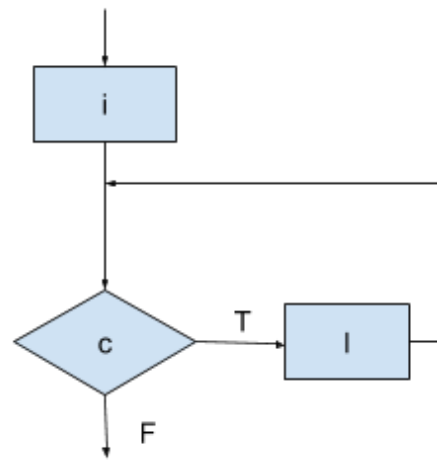
if/else selection block

if (c)
then a;
else b;

While loop block
i;
while (c)
 l;

Example 2 - Rental Fee

Now, let us take a look at another example. This example is more complicated than Example 1. It contains not only a nested-if code block, but also a for loop. In addition, there are two inputs: One is the tape array, and the other is the customer. In particular, the length of the tape array and whether the customer is premium are used in the function to decide how to calculate the total rental fee.

```
Float calcRentalFee(Tape[] tapes, Customer customer){
        float total = 0;
        for(int 1 = 0; 1 < tapes.length; 1++){
                total += tapes[1].price;
        }
        if (tapes.length > 10){
                total *= .8;
        } else if(tapes.length > 5){
        }
        if(customer.isPremium()){
                total *= .9;
        }
        return total;
}
```

The flowchart is as follows:
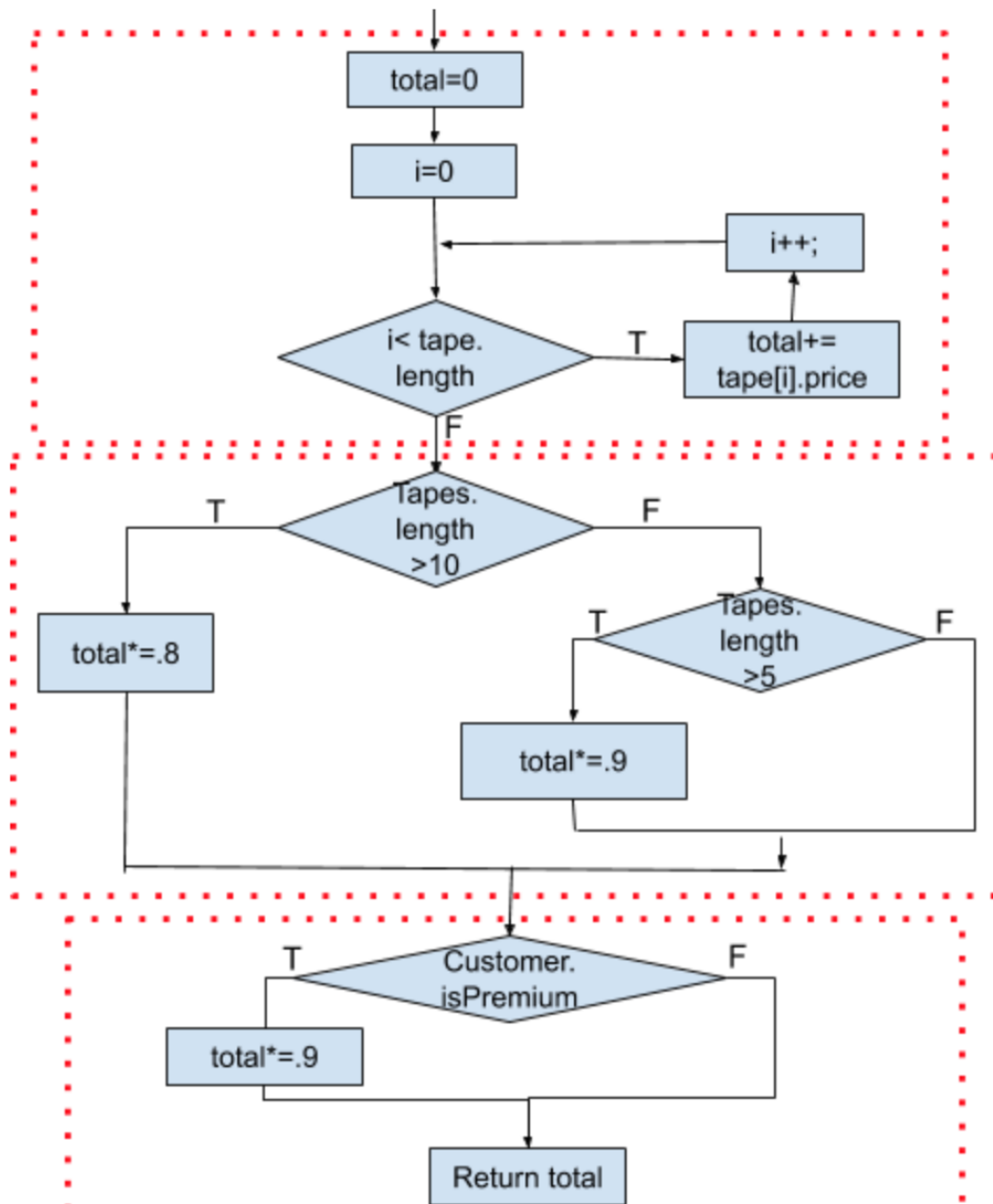
Flowchart for Example 1

This flowchart can be divided into three subblocks. The top one is a loop block. The middle one is a nested-if selection block. The bottom one is a simple-if selection block.

With a loop structure, it is hard to calculate all different paths, as it depends on the number of iterations, which can be a dynamic and unpredictable number. Instead, we only count two basis paths, based on the evaluation of the loop condition for the first time, as shown below. Let us notate them as P11 and P12. It is easy to see that the middle block has three paths (P21, P22, and P23) and the bottom block has two paths (P31 and P32). The number of branches is the same as the number of basis paths in each block.
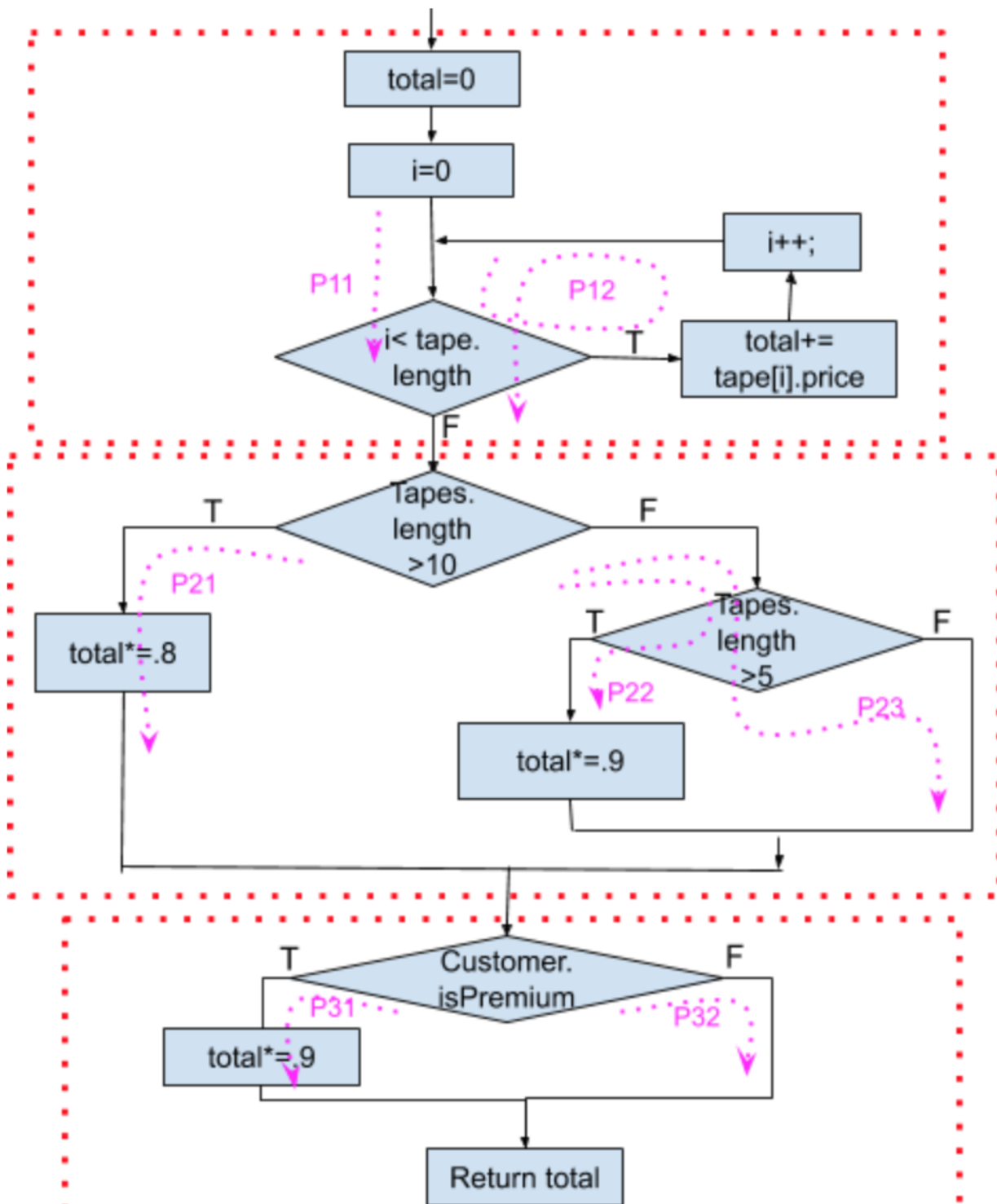
Basis Paths in Each Block

Basis Paths in Each Block

If those blocks are completely independent of each other, then the total number of basis paths can be the multiplication of all of these numbers: **2 X 3 X 2 = 1**2.

However, the top and middle blocks have data dependency. If there is no tape (*tape.length* = 0), the only path to execute in the middle block is P23. Therefore, the total number of possible basis paths is **1 X 1 X 2 + 1 X 3 X 2 =**

8.

Here are the details of the path and possible test cases.

Test Case #	Input Data	Path
1	Tape.length: > 10 (e.g. 11) Customer.isPremium: T	P12, P21, P31
2	Tape.length: > 10 (e.g. 11) Customer.isPremium: F	P12, P21, P32
3	Tape.length: [5,10] (e.g. 6) Customer.isPremium: T	P12, P22, P31
4	Tape.length: [5,10] (e.g. 6) Customer.isPremium: F	P12, P22, P32
5	Tape.length: [0,5] (e.g. 4) Customer.isPremium: T	P12, P23, P31
6	Tape.length: [0,5] (e.g. 4) Customer.isPremium: F	P12, P23, P32
7	Tape.length = 0 Customer.isPremium: T	P11, P23, P31
8	Tape.length = 0 Customer.isPremium: F	P11, P23, P32
Impossible Path		P11, P21, P31
		P11, P21, P32
		P11, P22, P31
		P11, P22, P32

While we need minimum of eight test cases to cover all basis paths. We don't need that many test cases to cover all statements or branches. For example, since there are no additional statements in P32, just executing P31 can cover all statements in the bottom block. That is true for P11 and P23, as well. P12 can cover all statements executed in P11, and P21 or P22 can cover all statements executed in P23. So, to cover all line statements, we only need two test cases: test case 1 and test case 3 (or test case 4).

Test Case #	Input Data	Path
1	Tape.length: > 10 (e.g. 11) Customer.isPremium: T	P12, P21, P31
4	Tape.length: [5,10] (e.g. 6) Customer.isPremium: F	P12, P22, P32

The largest number of branches is in the middle block. We can design three test cases that cover these three branches in the middle block, as well as the top and bottom blocks.

Test Case #	Input Data	Path
1	Tape.length: > 10 (e.g. 11) Customer.isPremium: T	P12, P21, P31
4	Tape.length: [5,10] (e.g. 6) Customer.isPremium: F	P12, P22, P32
7	Tape.length = 0 Customer.isPremium: T	P11, P23, P31

Other Testing Measurements

Besides code-structure coverage, there are also other testing measurements, such as data coverage, feature coverage, and the defect measurements. However, be aware that 100% of coverage in these measures doesn't imply complete testing at all. Actually, it may lead to a false sense of confidence, as people tend to maximize what they measure in the cost of what they don't measure. In the above example, even with eight test cases, while we can cover all basis paths, we may neglect other possible and even obvious bugs. For example, if the programmers make a mistake and input "if (tape.length>10)" as "if (tape.length >= 10)," then the above eight test cases will not be able to catch this bug, as we didn't test the case in which the length is equal to 10. **Actually, these more risky test cases should be considered besides the coverage.**

Black-Box Testing

Unlike in white-box testing, test cases in black-box testing are designed without the knowledge of the internal design or code. The only information available are the requirement specifications that describe the application's

external behavior. While white-box testing is mostly conducted by the developers, who also write the source code, black-box testing is usually conducted by separate QA engineers or third-party testers without knowledge of source code. Black-box testing is usually conducted at the system level to validate the external functional or nonfunctional behavior of the whole software system, or at the integration level to validate the behavior of the partially integrated software system—particularly in projects using continuous integration.

In black-box testing, testers design test cases (input and expected output) based on the requirement specifications. Testers may need to work closely with the clients and developers to understand both explicit and implicit, written and unwritten requirements. This can be done during the requirement-analysis activity in each iteration. Both positive tests (valid inputs) and negative tests (invalid inputs) are considered. After the execution of each test case, the actual outputs are compared with the expected outputs to check whether the software system correctly handles the inputs. If the two are not the same, the testers will send bug reports to the developers to request a bug fix. After the fix, the tests will be reconducted. Black-box testing may be conducted both automatically and manually.

Domain Testing

Domain testing is a widely taught and practiced testing technique, and it can be used in both white- and black-box testing. The idea of domain testing is to select a small number of test cases from a nearly infinite group of candidate test cases using domain knowledge. A domain is a set of values associated with a function. We can view the program as a function that has input domains and output domains. An input domain is a set of all possible values of a variable that can be input to the program. By partitioning a domain into equivalence classes and selecting only representative values in each class, we can reduce the number of test cases while achieving the same or a similar testing result (Kaner, 2004). Two techniques are used in domain testing: equivalence partition testing and selecting best representatives, which are usually boundary cases.

The following steps are involved in the domain testing (Padmanabhan, 2003):

- Identify all variables that can be analyzed. We usually focus on the input variables, but also use output variables sometimes.
- Understand the requirements and functionality with respect to the identified variables.
- Identify characteristics, values, and dimensions of each variable. A dimension of a variable is one aspect of the variable that changes. For example, a string variable has two dimensions: the length of the string

and allowed or required characters in the string. The length values are usually in a linear value space within a range. For example, the length of a password should be between 8 and 256. The required/allowed characters of a password are in a discrete set.

- Use equivalence partition and boundary (or best representative) case techniques to choose a small number of input values for each variable.
- Tests usually involve multiple variables. Finally, we will need to combine multiple variables.

Equivalence Partition

The idea is to partition the test-case space into equivalence classes, where all test cases within one equivalence class are essentially the same for the purpose of testing.

Basically, an equivalence partition contains test cases with the input values that are treated the same way by the system and produce the same kind of results. These can be identified based on the requirement specification. Valid and invalid are two common partition types for any input values. For example, the triangle problem discussed in the previous module has three input variables. Each input variable can have two types of values: a positive number as the valid side and a negative number or zero as the invalid side.

Equivalence Class	*Any Side (a or b or c)*
Valid	> 0
Invalid	≤ 0

Based on the valid- and invalid-partition idea, we have two different cases:

1. If the input is valid across a range of values, we can partition the domain into three equivalence classes: below the range, within the range, and above the range.
2. If the input is only valid when it is a member of a discrete set, we can partition the domain into two equivalence classes: valid discrete values and invalid discrete values.

Let us look at another example: a calendar. Suppose we need to show the number of valid dates in each month. The inputs of this problem are the year number and month. The output is the number of days in that month.

Month	Equivalence Class
Valid	[1, 12]
Invalid	< 1 > 12

For the year number, suppose we can only consider A.D. and within 100 years from now.

Year	Equivalence Class
Valid	[1, 2120]
Invalid	< 1 > 2120

We may also analyze the output domain. As the valid outputs here are only 28, 29, 30, and 31, we can divide the output domain into five partitions. We should have test cases in each of four valid-output classes. However, it is generally not straightforward to design test cases for the invalid-output domain.

Output Day in a Month	Equivalence Class
Valid	28 29 30 31
Invalid	All other numbers

Since the month domain is a discrete set, the day in each month is not calculated similarly. Combining both the year and the month input domain with the output domain, we may need at least 13 combined partitions.

	Year	Month	Output
Valid	Any valid	1	31
	Leap year	2	29
	Non–leap year	2	28
	Any valid	3	31
	Any valid	4	30
	Any valid	5	31
	Any valid	6	30
	Any valid	7	31
	Any valid	8	31
	Any valid	9	30
	Any valid	10	31

	Any valid	11	30
	Any valid	12	31
Invalid	Invalid	Invalid	Error

When used in white-box testing with knowledge of the source code, the input values that cause the program to take the same execution path are in the same equivalence class. For example, we can easily partition the test cases for the previous "Character Classification" equivalence classes from Example 1:

Equivalence Class	Input
Positive number	≥ 0
Negative number or 0	< 0
Alphabet letter	A to Z, or a to z
Symbol	A ist of all valid symbols
Disallowed character	Anything not in the above list

Selecting Best Representatives

While any member of an equivalence class is supposed to be the same for the purpose of testing, they may not all reveal an error to the same extent, and random sampling from equivalence-class members is not a very effective way to test. Boundary values are better representatives of each equivalence class, as they are more likely to reveal an error. For example, off-by-one errors are very common, as the developer can be easily confused between "≥ " and ">" (or between "≤" and "<"). For example, if the program has the bug of input "> 0" instead of input "≥0", then we will not capture the bug by choosing any positive number, such as five, instead of zero. For example, the boundary cases in the previous example are shown below:

Equivalence Class	Boundary Cases
Positive number	0, 1
Negative number or 0	−1
Alphabet letter	A, Z, a, z

The symbol and disallowed character classes are not linear spaces. This requires further understanding of the context and clarification from clients in order to design the most effective test cases. For example, a discrete set may be provided in the requirement specification, explicitly specifying what symbols are allowed. Another possible specification is "any nonprinting symbol in ASCII code is not allowed." In this case, we may convert the nonlinear input space into a linear input space based on the ASCII code mapping. Another clarification is needed to specify how many characters are allowed in the input, or just until "the enter key is pressed." Additional tests are needed to validate how the system handles—for example, if the input only contains an "enter key" (an empty string) or contains too many characters (very long string). Here, we can see that clear specification and understanding of the requirements are critical to testing.

Not every domain is a linear domain with boundary values clearly identified. To support the domain analysis of nonlinearly ordered space, boundary cases are generalized to the best representative cases, which are most likely to expose errors among all cases in an equivalence class.

Risk-Based Domain Testing

The idea of risk-based testing is to design the test cases based on the identified risks. In testing, a risk is a possible problem in the software system that can cause failure. By identifying and analyzing the risks, we can design and prioritize test cases that may or may not trigger the risks. A quick test is an inexpensive, ad hoc, risk-based testing, which usually requires little knowledge. An example is shoe testing. Put a shoe on the keyboard and wait to see how the system handles these random and faulty inputs. Here are some ideas to trigger possible failures:

- Invalid input value
- Invalid input type (e.g. input arbitrary characters instead of numbers for integer input)
- Incorrect number of inputs
- Input overflow (e.g., use too large a number to overflow the integer)
- Try different timing for input events to trigger possible synchronization issues
- Repeatedly execute test cases many times, which may cause memory leaks

The risk analysis can be used in equivalence-class partitioning and representative-values selection. Cem Kaner (2004) identified specific tasks involved in risk-based domain testing approach as follows:

- Identify risks in each variable dimension
- Partition the input domain into subdomains based on the identified risks by performing equivalence-class analysis
- In each class (subdomain), identify the best representative (e.g., boundary cases) that can most likely trigger the risk
- Perform combination testing

We can use the following table to show domain-testing analysis (Padmanabhan, 2003). Suppose an input variable can only be a number between [–99, 99].

Mishandling the value just below the lower boundary.

Variable	Dimension	Equivalence Classes	Best Representatives	Risks	Notes
x	value	[–99, 99]	–99, 99	Failure to process valid values correctly. Mishandling the boundary values.	
		<–99	–100	Mishandling the invalid value of fewer than –99. Mishandling the value just below the lower boundary. .	
		> 99	100	Mishandling the invalid value of greater than 99. Mishandling the value just above the upper boundary.	

Combination Testing

Since a software system function usually involves multiple variables that interact with each other, we need to combine them to design test cases. All-pairs combination is a technique used to include all pairs of values for every variable in the test cases. Each value of a variable is combined with every value of every other variable **at least once**. However, variables with dependency on each other will not have all combinations. Thus, only variables that are independent of each other are considered in the all-pairs combination. This greatly reduces the number of combinations. For example, suppose we have five variables, and each has two equivalence classes. A simple all-pairs combination can result in 2 * 2—that is, four classes. However, the number of all possible combinations is 2 ^ 5—that is, 32 classes.

Here is an example shown in [Padmanabhan, 2003]. Suppose there are 5 variables, a, b, c, d and e, where a has 3 cases, and all other variables have 2 cases each.

Test Case #	a(3)	b(2)	c(2)	d(2)	e(2)
1	A1	B1	C1	D1	E1

Test Case #	a(3)	b(2)	c(2)	d(2)	e(2)
2	A1	B2	C2	D2	E2
3	A2	B1	C2	D1	E2
4	A2	B2	C1	D2	E1
5	A3	B1	C1	D2	E1
6	A3	B2	C2	D1	E2

Example

In a web-based mail system, a user has to enter his/her username and password and then click on the "Sign Up" button to register an account. The user name can have five to fifteen characters. Only digits and lowercase letters are allowed for the user name (Padmanabhan, 2003)

1. 1. What variables could be involved in an analysis of this group of facts?

    A: username, password, sign-up button, account

2. What variable do we know enough about to perform an equivalence-class analysis and then a boundary-value analysis?

    A: username

3. Identify the relevant dimensions for this variable.

    A: length and allowed characters

4. Develop a series of tests by performing equivalence-class and boundary-value analysis on each of the identified dimensions of this variable.

Variable	Dimension	Equivalence Class	Best Representatives	Risks
username	length	[5, 15]	5, 15	Failure to process usernames with allowed length correctly. Mishandling the usernames with a length of boundary values.

Variable	Dimension	Equivalence Class	Best Representatives	Risks
		< 5	4, 0	Mishandling too-short usernames. Mishandling usernames just shorter than the lower boundary. Mishandling the empty usernames.
		> 15	16 Max value allowed in the system. Max value allowed in the system + 1.	Mishandling too-long usernames. Mishandling the usernames just longer than the upper boundary. Mishandling too long usernames and beyond.
	Allowed characters	All allowed characters, digits, and lowercase letters	A username containing: Digit: '0' (ASCII: 48) '9' (ASCII: 57) Lowercase char: 'a' (ASCII: 97) 'z' (ASCII: 122)	Mishandling of usernames that contain allowed characters. Mishandling of usernames that contain characters of boundary values.
		All values containing at least one character that is NOT allowed	A user name containing: '/' (ASCII: 47) ':' (ASCII: 58) '`' (ASCII: 96) '{' (ASCII: 123) A character corresponding to ASCII 128 of the extended ASCII set.	Mishandling of usernames that contain characters that are NOT allowed. Mishandling of characters just beyond lower and upper boundary values, respectively. Mishandling of characters beyond the upper boundary of the standard ASCII set.

Let us look at the classical triangle problem again. It is a classical problem used in software testing since 1969. The program reads three numbers as the sides of a triangle and states whether the triangle is scalene, equilateral, or isosceles.

Examples of Myers's categories follow:

1. Test case for a valid scalene triangle. (e.g. 3. 4, 5)
2. Test case for a valid equilateral triangle (e.g. 3, 3, 3)
3. Three test cases for valid isosceles triangles (e.g., [3, 3, 5], [3, 4, 4], [5, 4, 5])
4. One, two, or three sides has zero value (five cases)
5. One side has a negative
6. Sum of two numbers equal the third (e.g., 1, 2, 3) (a + b = c, a + c +b, b + c = a)
7. 7. Sum of two numbers is less than the third (e.g., 1, 2, 4) (a + b < c, a + c < b, b + c < a)
8. Non-integer
9. Too many or too few sides

Test Yourself

Can you think about any other test cases that are needed? Based on Myers's categories and what we learned in this module, can you use the previous risk-based equivalence class–analysis table to outline the test cases?

Integration Testing

Finally, let us discuss a little bit more about integration testing. There are several different methods to integrate subsystems for the testing purpose. Suppose a software system has the following three-layer design with seven subsystems. Top-layer subsystem A uses middle-layer subsystems B, C, and D. Middle-layer component B needs to use the service provided by bottom-layer E and F, and D needs to use G.
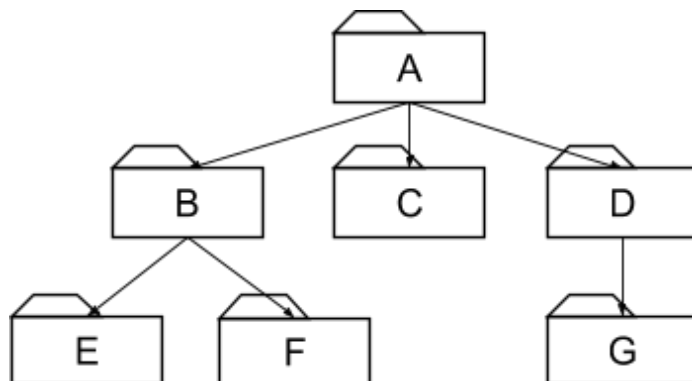

An Example of Three-Layer Software-System Architecture

The integration-testing strategy determines the order in which the subsystems are selected for testing and integration. The goal is to test all interfaces between subsystems and the interaction of subsystems.

There are several strategies:

- **Big-bang approach**—First come unit tests each of the subsystems, followed by one gigantic integration test, in which all the subsystems are immediately tested together. This is not a good method. If any defects are revealed in the integration test, it is hard to find which subsystem interfaces have caused the issues.
- **Bottom-up testing strategy**—The subsystems in the lowest layer of the call hierarchy are tested individually. Then the next subsystems are tested, those that call the previously tested subsystems. This is repeated until all subsystems have been tested. Drivers are needed to test the lower-layer components.
- **Top-down testing strategy**—Test the top layer, or the controlling subsystem, first (user interface). Then combine all of the subsystems that are called by the tested subsystem, and test the resulting collection. Do this until all subsystems have been incorporated into the test. Stubs are needed to do the testing.
- **Modified sandwich testing strategy**—First test all layers in parallel: the middle layer with drivers and stubs, the top layer with stubs, and the bottom layer with drivers. Then test all interactions in parallel, including the top layer accessing middle layer (the top layer replaces drivers), and the bottom layer being accessed by the middle layer (the bottom layer replaces stubs).
- **Continuous integration testing**—This is the strategy used in modern applications nowadays. Integration and testing start from day one. With almost daily builds, the system Jerkins, Hudson, and Teamcity. The tool set should support continuous building, automated tests with high coverage, software-configuration management, and issue tracking.
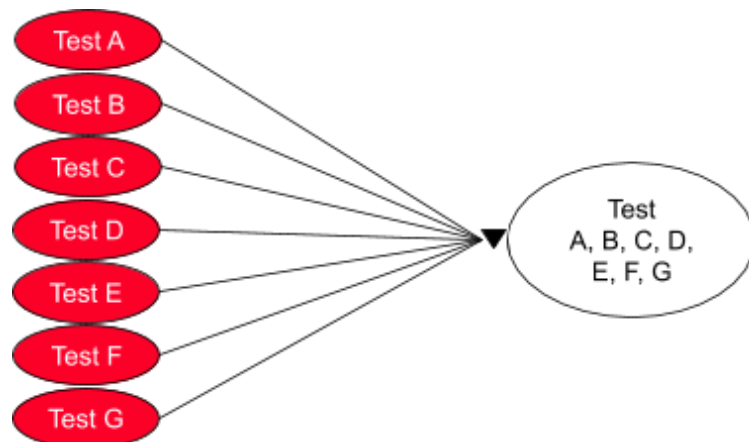
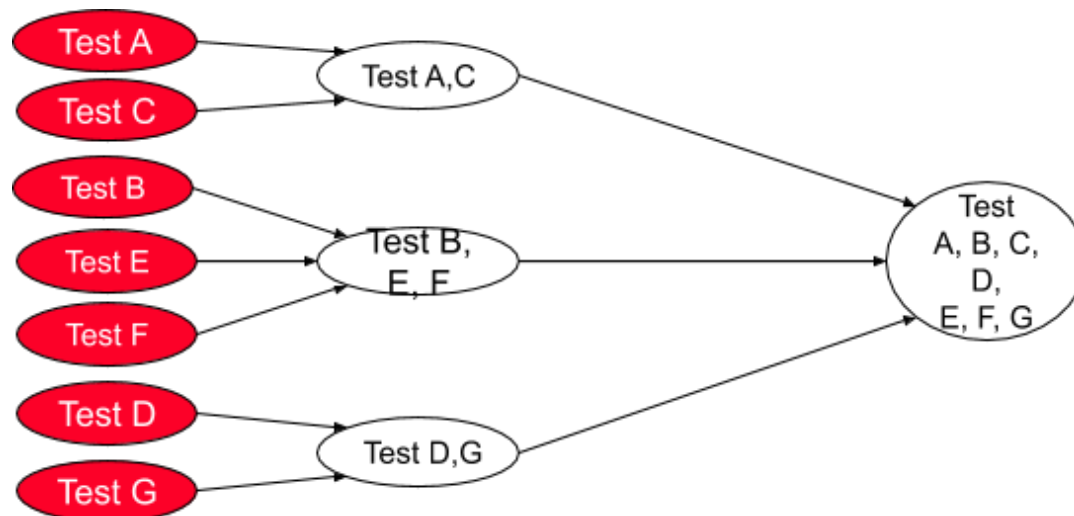Big-Bang Approach
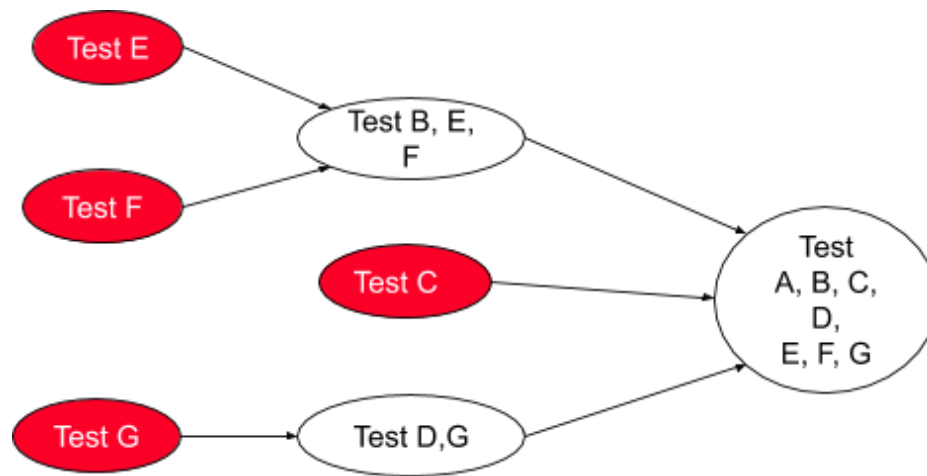
Bottom-Up Approach
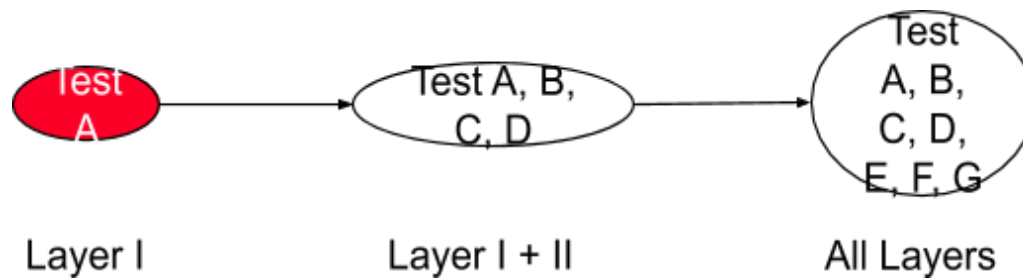
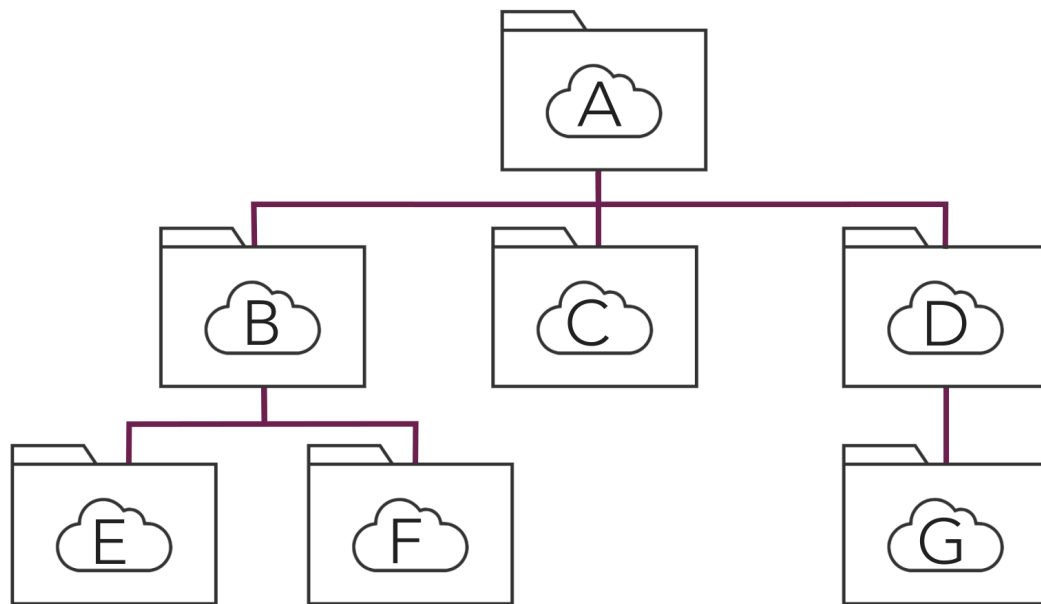Layer I   Layer I + II   All Layers

Top-Down Approach

Modified Sandwich Approach

Continuous Integration

Test Yourself

Test Yourself

What are the pros and cons using a coverage metric to evaluate the testing?

Pro: It provides a quantified approach to measuring the testing measurement.

Con: Achieving 100% coverage with these measures doesn't imply complete testing at all. Actually, it may lead to a false sense of confidence, as people tend to maximize what they measure in the cost of what they don't measure.

Test Yourself

Draw a flowchart for the following code snippet:

```java
double calTotal(ShopCart shopCart, boolean coupon, Customer customer) {
    double tot = 0;
    for (int i = 0; i< shopCart.itemNum; i++) {
        tot += shopCart.nextItem().getPrice();
    }
    tot = tot + tot * 0.06;
    if (coupon)
        tot *= 0.9;
    if (!customer.isPrime() && tot < 35)
        tot += 60;
```

```
        return tot;
    }
```

Test Yourself

What are the minimal numbers of test cases needed minimally to have 100% statement, branch, condition, and path coverage of the above code snippet, respectively?

Test Yourself

Consider the following scenario of an assignment submission on the Blackboard system, and answer the following questions:

On the CS 673 Blackboard course site, students can submit their completed assignments by clicking the corresponding assignment. The assignment submission page is then displayed to students. Students can upload their files, write comments, and click the "Submit" button to submit their assignments. Students can also cancel the submission or save it as a draft.

- *What variables could be involved in analysis of this group of facts?*
- *Identify the relevant dimensions for each variable.*
- *Identify related risks for each variable*
- *Develop a series of tests by performing equivalence-class and boundary-value analysis on each identified dimension of each variable, using the following table:*

Variable	Dimension	Equivalence Classes	Best Representatives	Risks	Notes

Conclusion

In this module, we introduced several UML diagrams used in requirement analysis and design: use-case diagrams, state diagrams, and sequence diagrams. We also discussed the differences between white- and black-box testing as well as how to measure test coverage and use domain testing techniques to design test cases.

- Kaner, C. J. D. (2004). Teaching domain testing: A status report. *Conference on Software Engineering Education & Training 2004, March 1–3.*
- Padmanabhan, S. (2003). Domain testing: Divide and conquer [M.Sc. thesis in software engineering, Florida Institute of Technology, in preparation; to be published by the Center for Software Testing Education & Research].

Boston University Metropolitan College