# Chapter 13-14
# Software Testing Techniques

- Testing fundamentals
- White-box testing
- Black-box testing
- Object-oriented testing methods

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

# Karakteristik Perangkat Lunak Yang Dapat Diuji

- **Operable**
  - Semakin baik dia bekerja, semakin efisien dia dapat diuji
- **Observable**
  - Apa yang anda lihat adalah apa yang anda uji
- **Controllable**
  - Semakin baik kita dapat mengontrol perangkat lunak semakin banyak pengujian yang dapat diotomatisasi dan dioptimalkan
- **Decomposable**
  - Dengan mengontrol ruang lingkup pengujian, kita dapat dengan lebih cepat mengisolasi masalah dan melakukan pengujian kembali secara lebih halus

(more on next slide)

# Karakteristik Perangkat Lunak Yang Dapat Diuji (continued)

- **Simple**
  - Semakin sedikit yang diuji, semakin cepat kita dapat mengujinya
- **Stable**
  - Semakin sedikit perubahan, semakin sedikit gangguan dalam pengujian
- **Understandable** (*Kemampuan untuk dapat dipahami* )
  - Semakin banyak informasi yang kita miliki, semakin halus pengujian yang akan di lakukan

# Test Characteristics

- Pengujian yang baik memiliki probabilitas yang tinggi untuk menemukan kesalahan.
- Pengujian yang baik tidak redundan.
- Pengujian yang baik seharusnya “jenis terbaik”
- Pengujian yang baik tidak boleh terlalu sederhana atau terlalu kompleks

# Two Unit Testing Techniques

- Black-box testing
  - Mengetahui fungsi tertentu yang telah dirancang untuk melakukan produk, Tes untuk melihat apakah fungsi itu sepenuhnya operasional dan bebas dari kesalahan
  - Termasuk tes yang dilakukan di antarmuka perangkat lunak
  - Tidak peduli dengan struktur logis internal perangkat lunak
- White-box testing
  - Mengetahui cara kerja internal suatu produk, uji bahwa semua operasi internal dilakukan sesuai dengan spesifikasi dan semua komponen internal telah dilakukan
  - Melibatkan tes yang berkonsentrasi pada pemeriksaan detail prosedural yang ketat
  - Jalur logis melalui perangkat lunak diuji
  - Uji kasus menggunakan rangkaian kondisi dan loop tertentu

# White-box Testing

# White-box Testing

- Menggunakan struktur kontrol bagian dari desain tingkat komponen untuk mendapatkan kasus uji
- Kasus uji ini
  - Memberikan jaminan bahwa semua jalur independen pada modul telah digunakan paling tidak satu kali;
  - Menggunakan semua keputusan logis pada sisi true dan false;
  - Mengeksekusi semua loop pada batasan mereka dan pada batas operasional mereka;
  - Menggunakan struktur data internal untuk menjamin validitasnya.

"Bugs lurk in corners and congregate at boundaries"

# Pengujian Basis Path

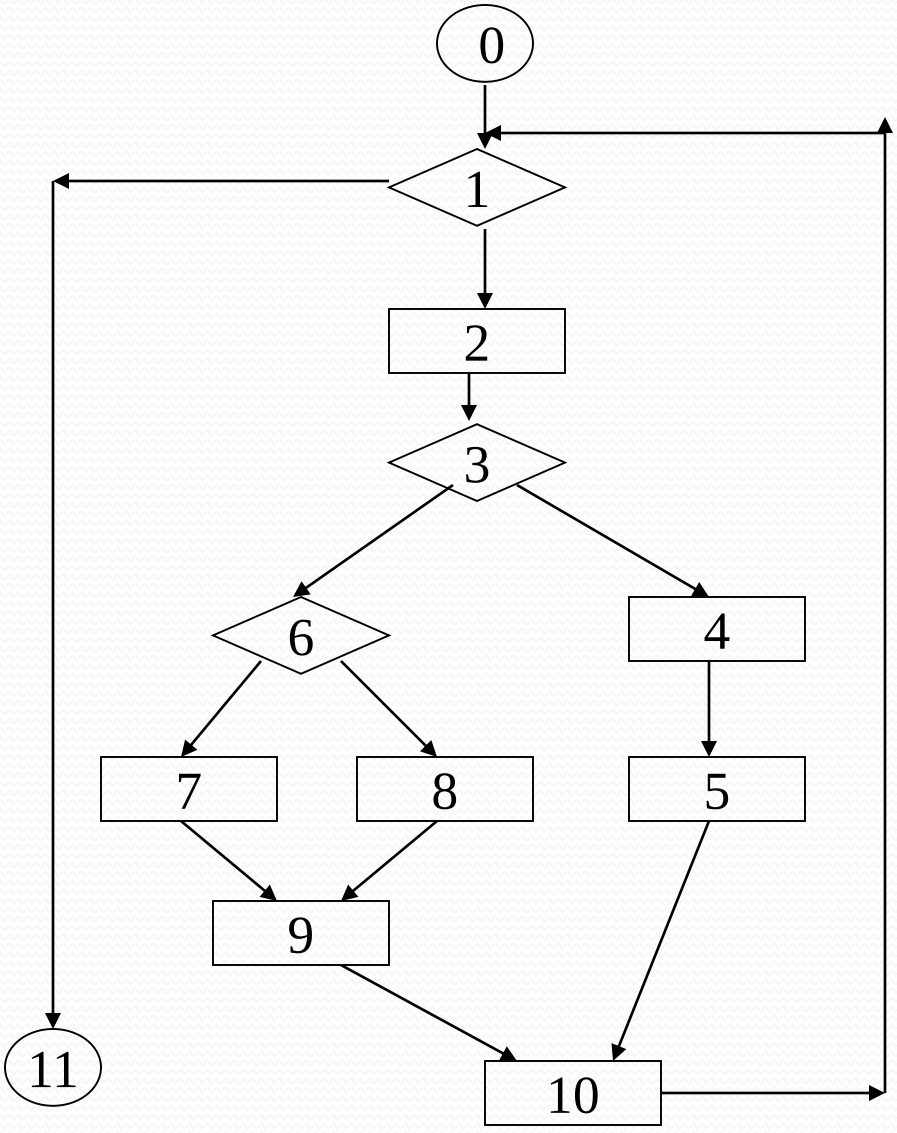
- Adalah teknik pengujian White-Box yang diusulkan pertama kali oleh Tom McCabe.
- Metode basis path ini memungkinkan desainer test case mengukur kompleksitas logis dari desain prosedural dan menggunakannya sebagai pedoman untuk menetapkan basis set dari jalur eksekusi.
- Test case yang dilakukan untuk menggunakan basis set tersebut dijamin untuk menggunakan setiap statamen di dalam program paling tidak sekali selama pengujian.

# Flow Graph Notation

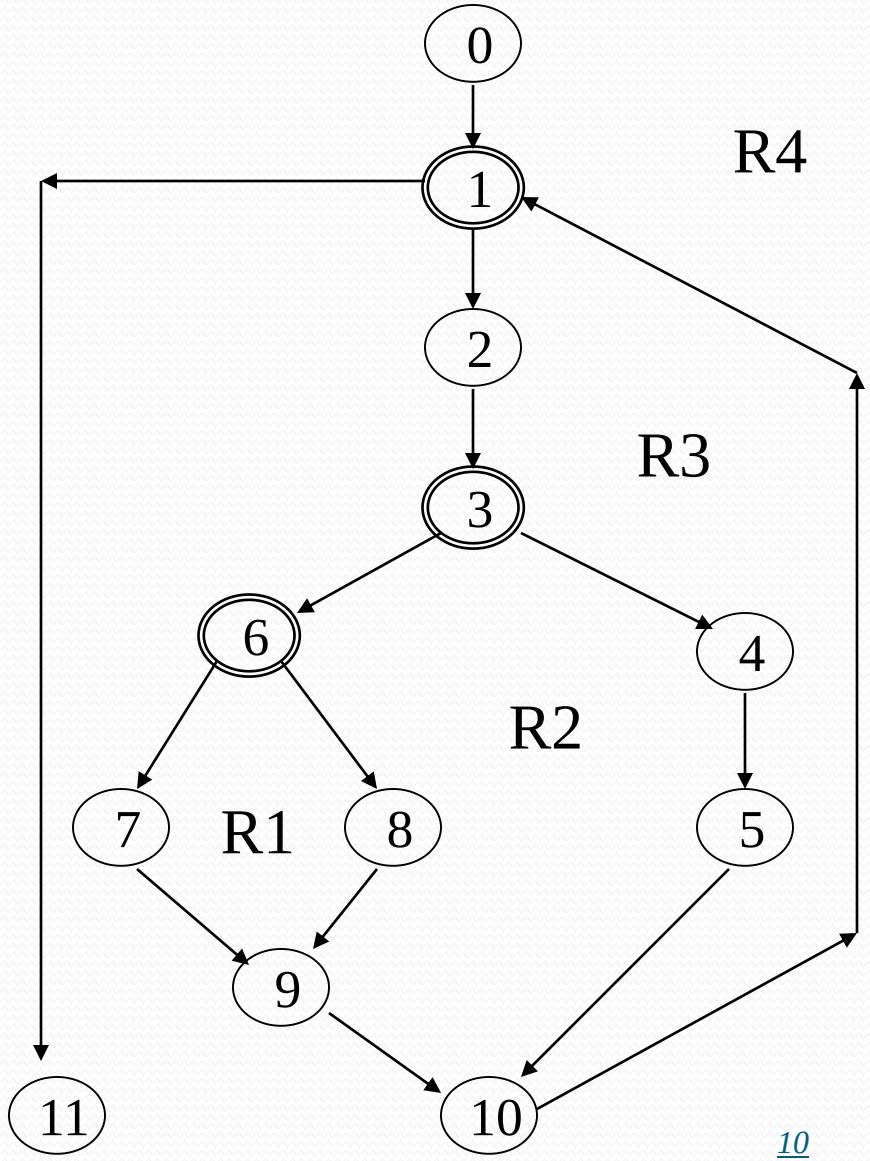
- Lingkaran dalam grafik mewakili simpul, yang merupakan urutan urutan satu atau lebih pernyataan prosedural
- Sebuah simpul (node) yang mengandung ekspresi kondisional sederhana disebut sebagai predikat simpul
  - Setiap kondisi senyawa dalam ekspresi kondisional yang mengandung satu atau lebih operator Boolean (e.g., and, or) diwakili oleh simpul predikat yang terpisah
  - Sebuah simpul predikat memiliki dua tepi yang menonjol darinya (True and False)
- Edge, atau link, adalah sebuah panah yang mewakili aliran kontrol ke arah yang spesifik
  - Sebuah edge harus dimulai dan berakhir pada sebuah simpul
  - Sebuah edge tidak bersinggungan atau menyilang sisi lain
- Daerah yang dibatasi oleh satu set tepi dan simpul disebut Regions
- Saat menghitung regions, sertakan area di luar grafik sebagai regions juga

# Flow Graph Example

**FLOW CHART**




**FLOW GRAPH**



# Independent Program Paths

- Ditetapkan sebagai jalur melalui program dari simpul awal sampai simpul akhir yang memperkenalkan setidaknya satu rangkaian pernyataan pemrosesan atau kondisi baru yang baru. (i.e., new nodes)
- Must move along at least one edge that has not been traversed before by a previous path
- Basis set for flow graph on previous slide
  - Path 1: 0-1-11
  - Path 2: 0-1-2-3-4-5-10-1-11
  - Path 3: 0-1-2-3-6-8-9-10-1-11
  - Path 4: 0-1-2-3-6-7-9-10-1-11
- The number of paths in the basis set is determined by the cyclomatic complexity

# Kompleksitas Siklomatis

- Adalah metriks perangkat lunak yang memberikan pengukuran kuantitatif terhadap kompleksitas logis suatu program.
- Bila metriks ini gunakan dalam konteks metode pengujian basis path, maka nilai yang terhitung untuk kompleksitas Siklomatis menentukan jumlah jalur independen dalam basis set suatu program dan memberi batas bagi jumlah pengujian yang harus dilakukan untuk memastikan bahwa semua statemen telah dieksekusi sedikit satu kali.

- Jalur independen adalah jalur yang melalui program yang mengintroduksi sedikitnya satu rangkaian statemen proses baru atau suatu kondisi baru.
- Sebagai contoh gambar b: adalah :
  - Jalur 1: 1-11
  - Jalur 2: 1-2-3-4-5-10-1-11
  - Jalur 3: 1-2-3-6-8-9-10-1-11
  - Jalur 4: 1-2-3-6-7-9-10-1-11

# Kompleksitas dihitung dalam salah satu dari tiga cara berikut:

1. Jumlah region grafik alir sesuai dengan kompleksitas siklomatis.
2. Kompleksitas siklomatis, *V(G)*, untuk grafik alir G ditentukan sebagai *V(G)*=E-N+2 di mana E adalah jumlah edge grafik alir dan N adalah jumlah simpul grafik alir.
3. Kompleksitas siklomatis, V(G), untuk grafik alir G juga ditentukan sebagai

$$V(G) = P+1$$

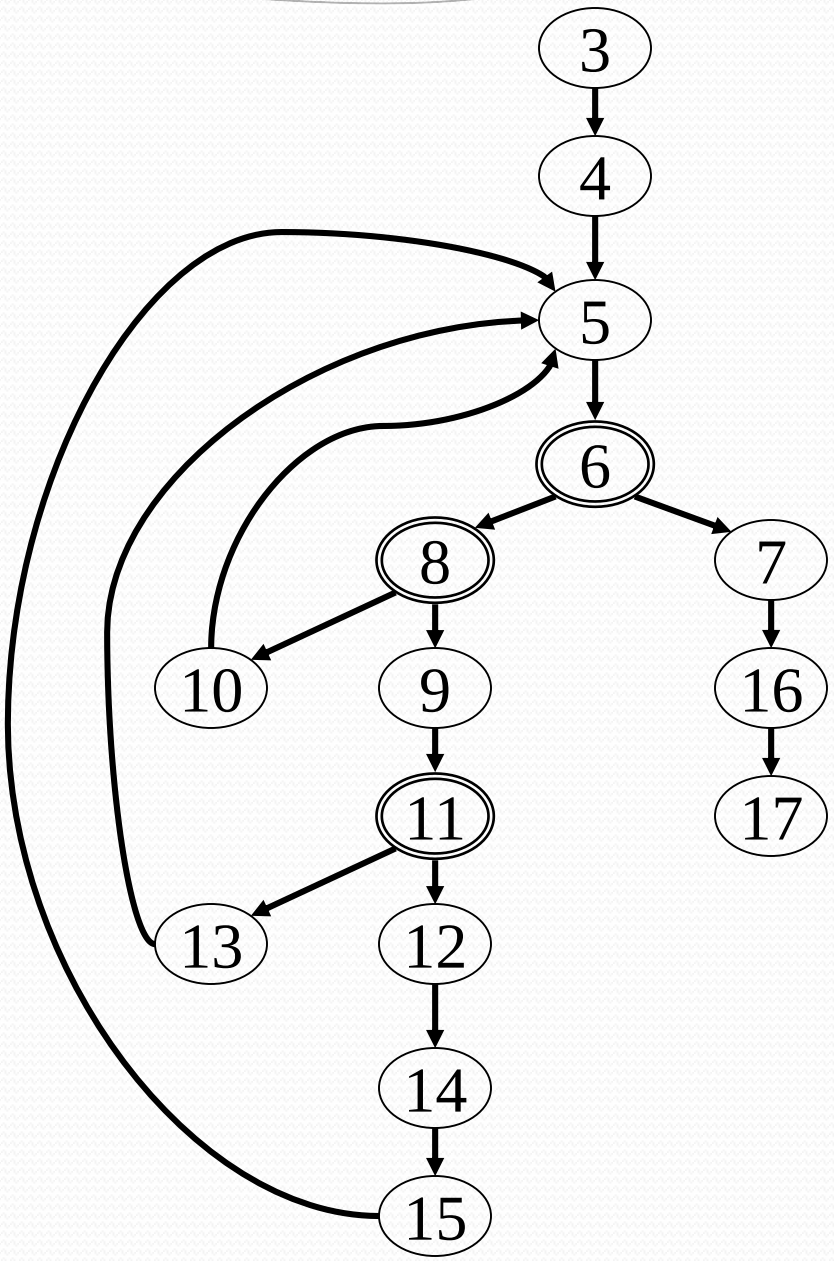
- Dimana **P** adalah jumlah simpul predikat yang diisikat dalam grafik alir V(G).
- Lihat sekali lagi grafik alir pada Gambar (b). Kompleksitas siklomatis dapat dihitung dengan menggunakan masing-masing dari algoritma yang ditulis diatas:
  1) Grafik alir mempunyai 4 region
  2) *V(G)* = 11 edge - 9 simpul + 2 = 4
  3) *V(G)* = 3 simpul yang diperkirakan + 1 = 4

# Deriving the Basis Set and Test Cases

1) Dengan menggunakan desain atau kode sebagai pondasi, gambarlah grafik alir yang sesuai
2) Tentukan kompleksitas siklomatik dari grafik alir resultan
3) Tentukan satu set dasar jalur bebas linear
4) Siapkan uji kasus yang akan memaksa eksekusi setiap jalur di basis set

# A Second Flow Graph Example

```
 1  int functionY(void)
 2  {
 3     int x = 0;
 4     int y = 19;

 5  A: x++;
 6     if (x > 999)
 7        goto D;
 8     if (x % 11 == 0)
 9        goto B;
10     else goto A;

11  B: if (x % y == 0)
12        goto C;
13     else goto A;

14  C: printf("%d\n", x);
15     goto A;

16  D: printf("End of list\n");
17     return 0;
18  }
```

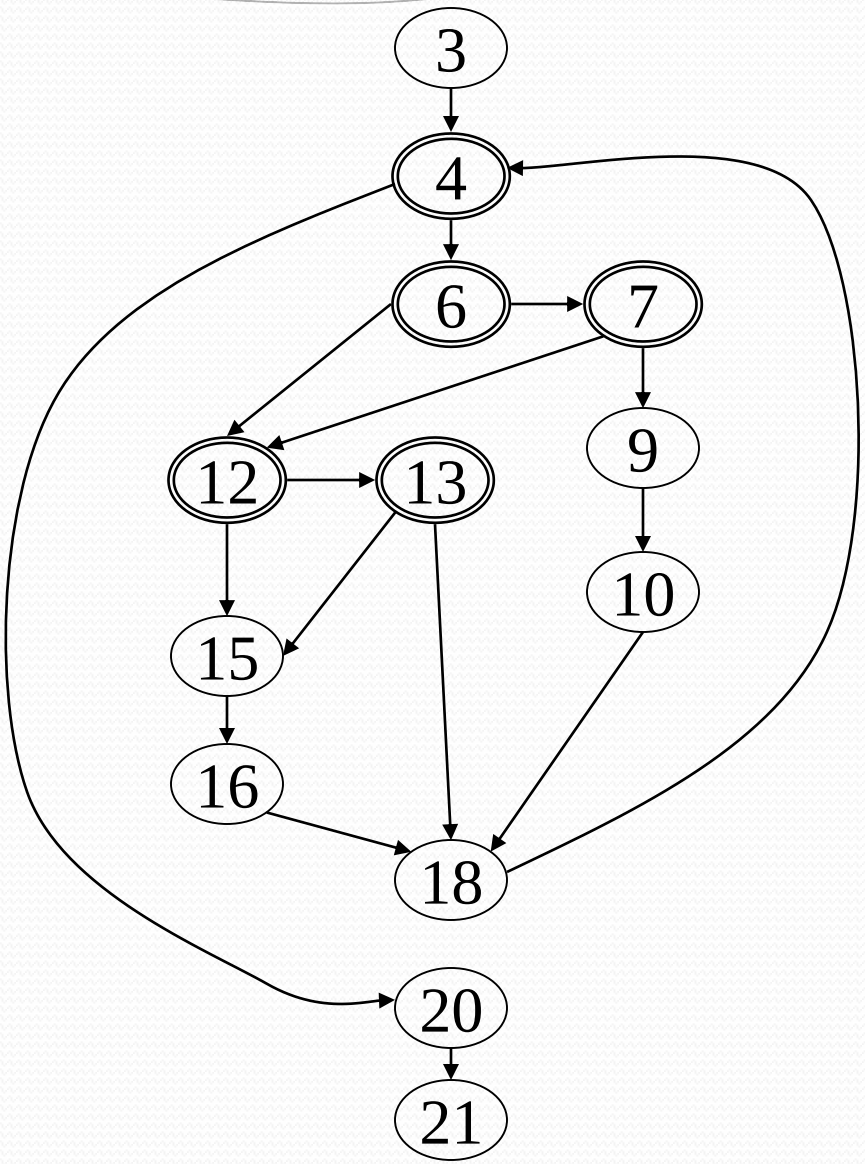


# A Sample Function to Diagram and Analyze

```
 1  int functionZ(int y)
 2  {
 3  int x = 0;

 4  while (x <= (y * y))
 5     {
 6     if ((x % 11 == 0) &&
 7          (x % y == 0))
 8        {
 9        printf("%d", x);
10        x++;
11        } // End if
12     else if ((x % 7 == 0) ||
13               (x % y == 1))
14        {
15        printf("%d", y);
16        x = x + 2;
17        } // End else
18     printf("\n");
19     } // End while

20  printf("End of list\n");
21  return 0;
22  } // End functionZ
```

# A Sample Function to Diagram and Analyze

```
 1  int functionZ(int y)
 2  {
 3  int x = 0;

 4  while (x <= (y * y))
 5     {
 6     if ((x % 11 == 0) &&
 7          (x % y == 0))
 8        {
 9        printf("%d", x);
10        x++;
11        } // End if
12     else if ((x % 7 == 0) ||
13               (x % y == 1))
14        {
15        printf("%d", y);
16        x = x + 2;
17        } // End else
18     printf("\n");
19     } // End while

20  printf("End of list\n");
21  return 0;
22  } // End functionZ
```



# Loop Testing - General

- Teknik white-box yang berfokus secara eksklusif pada validitas konstruksi loop
- Empat kelas loop yang berbeda ada
  - Simple loops
  - bersarang loops
  - bersambung loops
  - tidak terstruktur loops
- Pengujian terjadi dengan memvariasikan nilai batas loop
  - Examples:

```
for (i = 0; i < MAX_INDEX; i++)

while (currentTemp >= MINIMUM_TEMPERATURE)
```

# Testing of Simple Loops

1) Skip the loop entirely
2) Only one pass through the loop
3) Two passes through the loop
4) m passes through the loop, where m < n
5) n –1, n, n + 1 passes through the loop

'n' is the maximum number of allowable passes through the loop

# Testing of Nested Loops

1) Mulai dari lingkaran terdalam; Atur semua loop lainnya ke nilai minimum
2) Lakukan tes loop sederhana untuk loop terdalam sambil menahan loop luar pada nilai parameter iterasi minimumnya; Tambahkan tes lain untuk nilai out-of-range atau dikecualikan
3) Bekerja ke luar, melakukan tes untuk loop berikutnya, namun menjaga semua loop luar lainnya pada nilai minimum dan loop nested lainnya menjadi nilai "khas"
4) Lanjutkan sampai semua loop telah diuji

# Testing of Concatenated Loops

- Untuk loop independen, gunakan pendekatan yang sama seperti loop sederhana
- Jika tidak, gunakan pendekatan yang diterapkan untuk loop bersarang

# Testing of Unstructured Loops

- Mendesain ulang kode untuk mencerminkan penggunaan praktik pemrograman terstruktur
- Bergantung pada desain yang dihasilkan, gunakan pengujian untuk loop sederhana, loop bersarang, atau loop concatenated

# Black-box Testing

# Black-box Testing

- Melengkapi pengujian white-box dengan mengungkap berbagai kelas errors
- Berfokus pada persyaratan fungsional dan domain informasi perangkat lunak
- Digunakan pada tahap pengujian selanjutnya setelah pengujian white box telah dilakukan
- Penguji mengidentifikasi seperangkat kondisi masukan yang akan sepenuhnya melaksanakan semua persyaratan fungsional untuk sebuah program.
- Kasus uji memuaskan berikut ini:
  - Kurangi, dengan hitungan lebih dari satu, jumlah kasus uji tambahan yang harus dirancang untuk mencapai pengujian yang masuk akal
  - Beritahu sesuatu tentang adanya atau tidak adanya classes of errors, bukan kesalahan yang terkait hanya dengan tugas spesifik yang ada

# Black-box Testing Categories

- Fungsi salah atau hilang
- Kesalahan antarmuka
- Kesalahan dalam struktur data atau akses basis data eksternal
- Perilaku atau kesalahan kinerja
- Inisialisasi dan kesalahan penghentian

# Questions answered by Black-box Testing

- Bagaimana validitas fungsional diuji?
- Bagaimana perilaku dan kinerja sistem yang diuji?
- Kelas masukan apa yang akan membuat kasus uji yang bagus?
- Apakah sistem sangat sensitif terhadap nilai input tertentu?
- Bagaimana nilai batas kelas data terisolasi?
- Berapa tarif data dan volume data yang dapat ditoleransi sistem?
- Apa efek kombinasi data spesifik terhadap pengoperasian sistem?

# Equivalence Partitioning

- Metode pengujian black-box yang membagi domain masukan dari sebuah program ke dalam kelas data dari mana kasus uji diturunkan
- Kasus uji yang ideal dengan mudah mengungkap kelas kesalahan yang lengkap, sehingga mengurangi jumlah kasus uji yang harus dikembangkan
- Desain kasus uji didasarkan pada evaluasi kelas kesetaraan untuk kondisi masukan
- Kelas kesetaraan mewakili satu set keadaan yang valid atau tidak benar untuk kondisi masukan
- Dari setiap kelas kesetaraan, kasus uji dipilih sehingga jumlah atribut kelas ekuivalen terbesar adalah latihan sekaligus

# Guidelines for Defining Equivalence Classes

- If an input condition specifies a range, one valid and two invalid equivalence classes are defined
  - Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}
- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
  - Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}
- If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined
  - Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}
- If an input condition is a Boolean value, one valid and one invalid class are define
  - Input: {true condition} Eq classes: {true condition}, {false condition}

# Boundary Value Analysis

- A greater number of errors occur at the boundaries of the input domain rather than in the "center"
- Boundary value analysis is a test case design method that complements equivalence partitioning
  - It selects test cases at the edges of a class
  - It derives test cases from both the input domain and output domain

# Guidelines for Boundary Value Analysis

1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* as well as values just above and just below *a* and *b*
2. If an input condition specifies a number of values, test case should be developed that exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested

- Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above
- If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its minimum and maximum boundaries

# Object-Oriented Testing Methods

# Introduction

- Hal ini diperlukan untuk menguji sistem berorientasi objek pada berbagai tingkat yang berbeda
- Tujuannya adalah untuk mengungkap kesalahan yang mungkin terjadi saat kelas berkolaborasi satu sama lain dan subsistem berkomunikasi melintasi lapisan arsitektural
  - Pengujian dimulai " in the small" pada metode dalam kelas dan kolaborasi antara kelas
  - Seiring integrasi kelas terjadi, pengujian berbasis penggunaan dan pengujian berbasis kesalahan diterapkan
  - Akhirnya, use case digunakan untuk mengungkap kesalahan selama tahap validasi perangkat lunak
- Desain kasus uji konvensional didorong oleh tampilan input-process-output perangkat lunak
- Pengujian berorientasi objek berfokus pada perancangan urutan metode yang tepat untuk menggunakan keadaan kelas

# Testing Implications for Object-Oriented Software

- Karena atribut dan metode dienkapsulasi di kelas, metode pengujian dari luar kelas umumnya tidak produktif
- Pengujian membutuhkan pelaporan tentang keadaan suatu objek, namun enkapsulasi dapat membuat informasi ini agak sulit didapat
- Metode built-in harus diberikan untuk melaporkan nilai-nilai atribut kelas untuk mendapatkan gambaran dari keadaan objek
- Warisan memerlukan pengujian ulang setiap konteks penggunaan baru untuk kelas
  - Jika subkelas digunakan dalam konteks yang sama sekali berbeda dari kelas super, kasus uji kelas super akan memiliki sedikit penerapan dan serangkaian tes baru harus dirancang.

# Applicability of Conventional Testing Methods

- Pengujian White-box dapat diterapkan pada operasi yang didefinisikan di kelas
  - Basis path testing and loop testing dapat membantu memastikan bahwa setiap pernyataan dalam metode telah diuji
- Black-box testing methods are also appropriate
  - Use case dapat memberikan masukan yang berguna dalam desain black box test

# Fault-based Testing

- Tujuan pengujian berbasis kesalahan adalah merancang tes yang memiliki kemungkinan tinggi untuk menemukan kesalahan yang masuk akal
- Pengujian berbasis kesalahan dimulai dengan model analisis
  - Penguji mencari kesalahan yang masuk akal (yaitu, aspek penerapan sistem yang dapat menyebabkan cacat)
  - Untuk menentukan apakah kesalahan ini ada, kasus uji dirancang untuk menggunakan desain atau kode
- Jika model analisis dan perancangan dapat memberikan wawasan tentang apa yang mungkin salah, pengujian berbasis kesalahan dapat menemukan sejumlah kesalahan yang signifikan

# Fault-based Testing (continued)

- engujian integrasi mencari kesalahan yang masuk akal dalam pemanggilan metode atau koneksi pesan (i.e., client/server exchange)
- Tiga jenis kesalahan ditemui dalam konteks ini
  - Hasil yang tidak diharapkan
  - Salah metode atau pesan yang digunakan
  - Permintaan salah
- Perilaku suatu metode harus diperiksa untuk menentukan terjadinya kesalahan yang masuk akal saat metode dipanggil
- Pengujian harus menggunakan atribut suatu objek untuk menentukan apakah nilai yang tepat terjadi untuk jenis perilaku objek yang berbeda
- Fokus pengujian integrasi adalah untuk menentukan apakah ada kesalahan dalam kode panggilan, bukan kode yang disebut

# Fault-based Testing vs. Scenario-based Testing

- Fault-based testing misses two main types of errors
  - Incorrect specification: subsystem doesn't do what the user wants
  - Interactions among subsystems: behavior of one subsystem creates circumstances that cause another subsystem to fail
- A solution to this problem is scenario-based testing
  - It concentrates on what the user does, not what the product does
  - This means capturing the tasks (via use cases) that the user has to perform, then applying them as tests
  - Scenario-based testing tends to exercise multiple subsystems in a single test

# Random Order Testing (at the Class Level)

- Certain methods in a class may constitute a minimum behavioral life history of an object (e.g., open, seek, read, close); consequently, they may have implicit order dependencies or expectations designed into them
- Using the methods for a class, a variety of method sequences are generated randomly and then executed
- The goal is to detect these order dependencies or expectations and make appropriate adjustments to the design of the methods

# Partition Testing (at the Class Level)

- Similar to equivalence partitioning for conventional software
- Methods are grouped based on one of three partitioning approaches
- State-based partitioning categorizes class methods based on their ability to change the state of the class
  - Tests are designed in a way that exercise methods that change state and those that do not change state
- Attribute-based partitioning categorizes class methods based on the attributes that they use
  - Methods are partitioned into those that read an attribute, modify an attribute, or do not reference the attribute at all
- Category-based partitioning categorizes class methods based on the generic function that each performs
  - Example categories are initialization methods, computational methods, and termination methods

# Multiple Class Testing

- Class collaboration testing can be accomplished by applying random testing, partition testing, scenario-based testing and behavioral testing
- The following sequence of steps can be used to generate multiple class random test cases
  1) For each client class, use the list of class methods to generate a series of random test sequences; use these methods to send messages to server classes
  2) For each message that is generated, determine the collaborator class and the corresponding method in the server object
  3) For each method in the server object (invoked by messages from the client object), determine the messages that it transmits
  4) For each of these messages, determine the next level of methods that are invoked and incorporate these into the test sequence

# Tests Derived from Behavior Models

- The state diagram for a class can be used to derive a sequence of tests that will exercise the dynamic behavior of the class and the classes that collaborate with it
- The test cases should be designed to achieve coverage of all states
  - Method sequences should cause the object to transition through all allowable states
- More test cases should be derived to ensure that all behaviors for the class have been exercised based on the behavior life history of the object
- The state diagram can be traversed in a "breadth-first" approach by exercising only a single transition at a time
  - When a new transition is to be tested, only previously tested transitions are used