تولید نرم افزار مبتنی بر آزمون

تست

تاکید بر یافتن اشتباهات، ناسازگاری ها و نواقص است.

verification و **validation** میتوان از طریق تست بررسی شود.

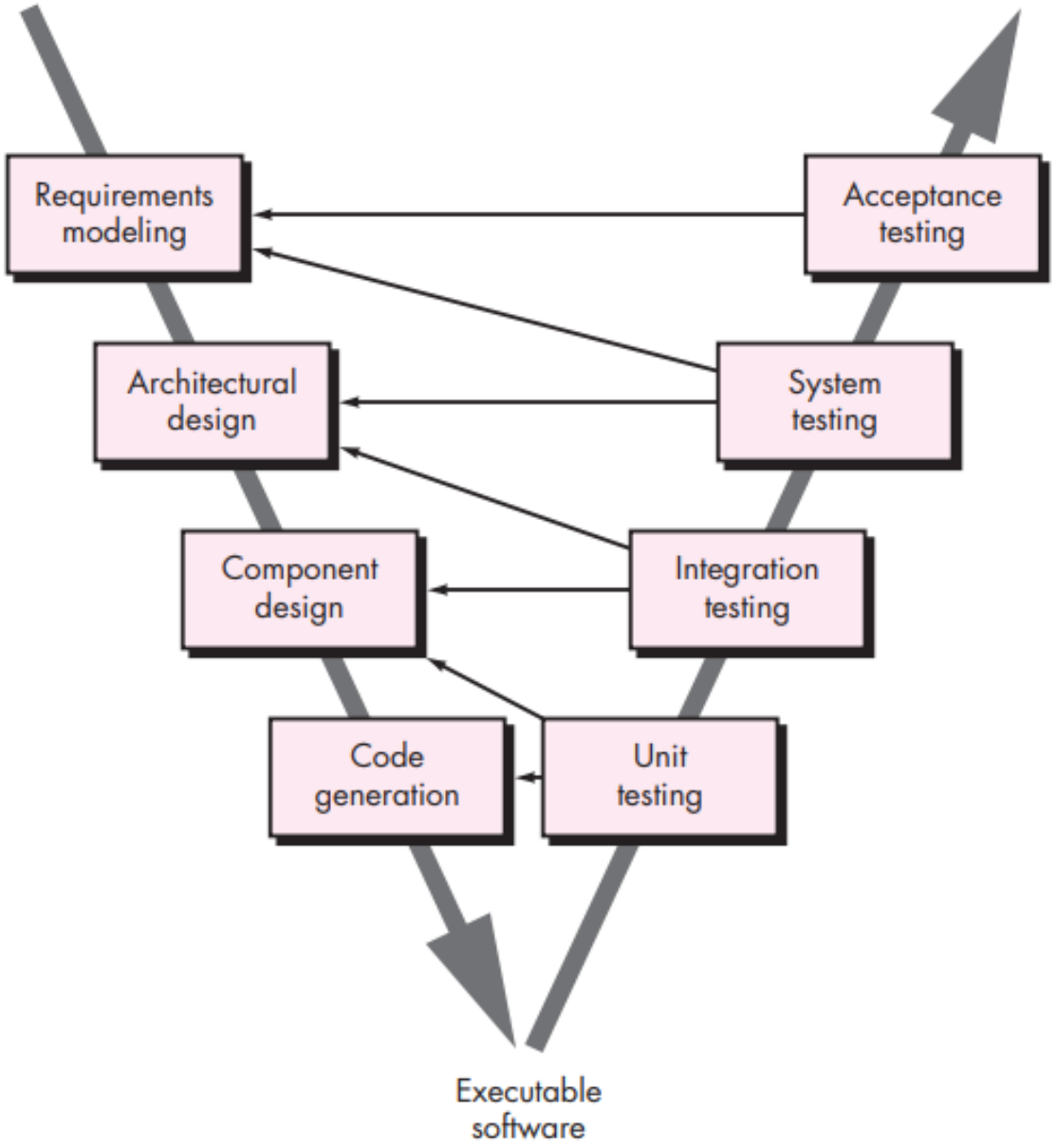
verification: "آیا ما محصول را درست می سازیم؟"
validation: "آیا ما محصول مناسبی می سازیم؟"

اهمیت تست

- تشخیص زود هنگام عیب های نرم افزاری
- افزایش کیفیت محصول
- افزایش اعتماد و رضایت کاربران
- شناسایی آسیب پذیری های امنیتی
- به مقیاس پذیری کمک می‌کند
- صرفه جویی در هزینه های پرداختی

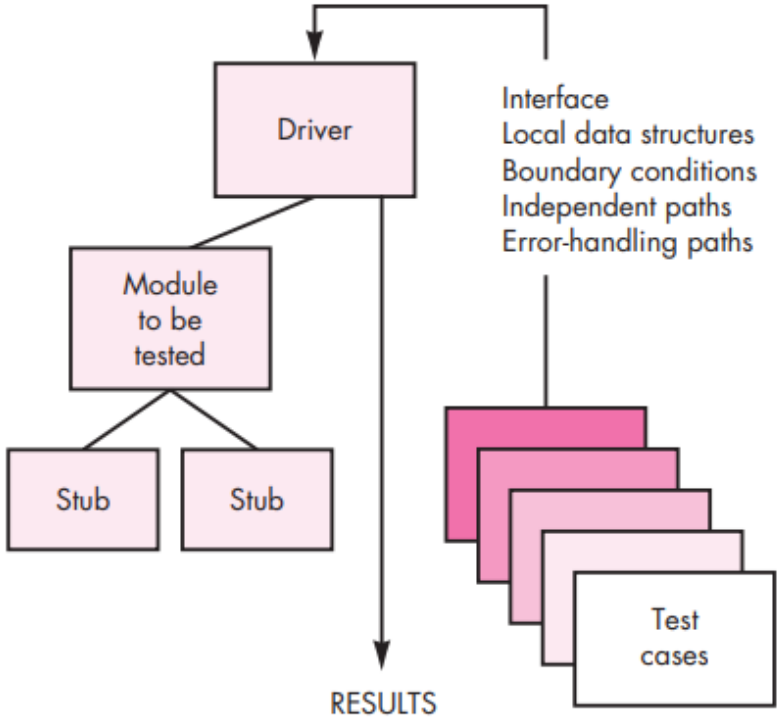
سطوح تست

v-model



Unit test - تست واحد

- کوچکترین عنصر قابل تست (مولفه یا ماژول) از سیستم را تست میکند. در این تست مسیرهای اجرایی برنامه چک میشود. مثلا قسمتی از برنامه ماژولی است که عمل sort را انجام میدهد، حال میتوان فقط همین را بررسی کرد که کار خود را به درستی انجام میدهد یا نه؟



تست واحد معمولاً به عنوان یک مرحله کمکی برای مرحله کدگذاری در نظر گرفته می شود.
طراحی آزمون های واحد می تواند قبل از شروع کدنویسی یا پس از تولید کد منبع اتفاق بیفتد.
از آنجا که یک جزء یک برنامه مستقل نیست، درایور و/یا زیربرنامه اغلب باید برای هر تست واحد توسعه داده شود.
در اکثر برنامه‌ها، درایور چیزی نیست جز یک «برنامه اصلی» که داده‌های مورد آزمایش را می‌پذیرد، چنین داده‌هایی را به مؤلفه ارسال می‌کند (برای آزمایش)، و نتایج مرتبط را چاپ می‌کند.
زیربرنامه ساختگی از رابط ماژول زیرمجموعه استفاده می‌کند، ممکن است حداقل دستکاری داده‌ها را انجام دهد، تأیید ورودی را چاپ می‌کند و کنترل را به ماژول در حال آزمایش برمی‌گرداند.

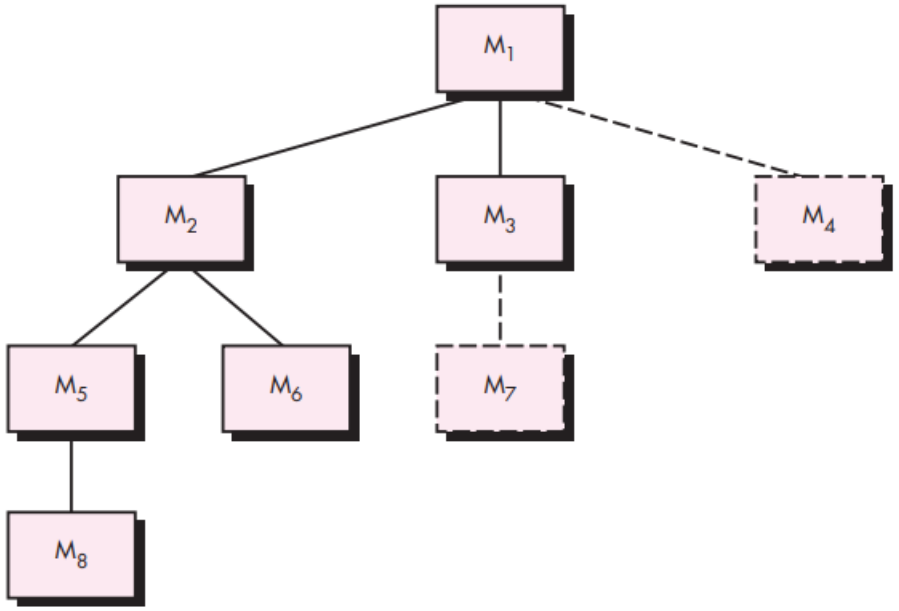
تست یکپارچگی Integration test

هدف تست ماژولها بعد از قرار گیری در کنار یکدیگر است یکی از مشکلات ممکن این است که در هنگام قرارگیری ماژولها در کنار هم، ممکن است برخی داده ها در واسط بین ماژولها حذف شوند.

یکی از روش های حل این مشکل، big bang می باشد. در این روش همه ماژولها را کنار یکدیگر قرار دهیم و یک جا تست کنیم. که روش سختی است و بررسی حالتهای مختلف را دشوار میکند.

یکی دیگر از روشها رویکرد افزایشی است.

رویکرد افزایشی



۱. Top-down integration

روش بالا به پایین:

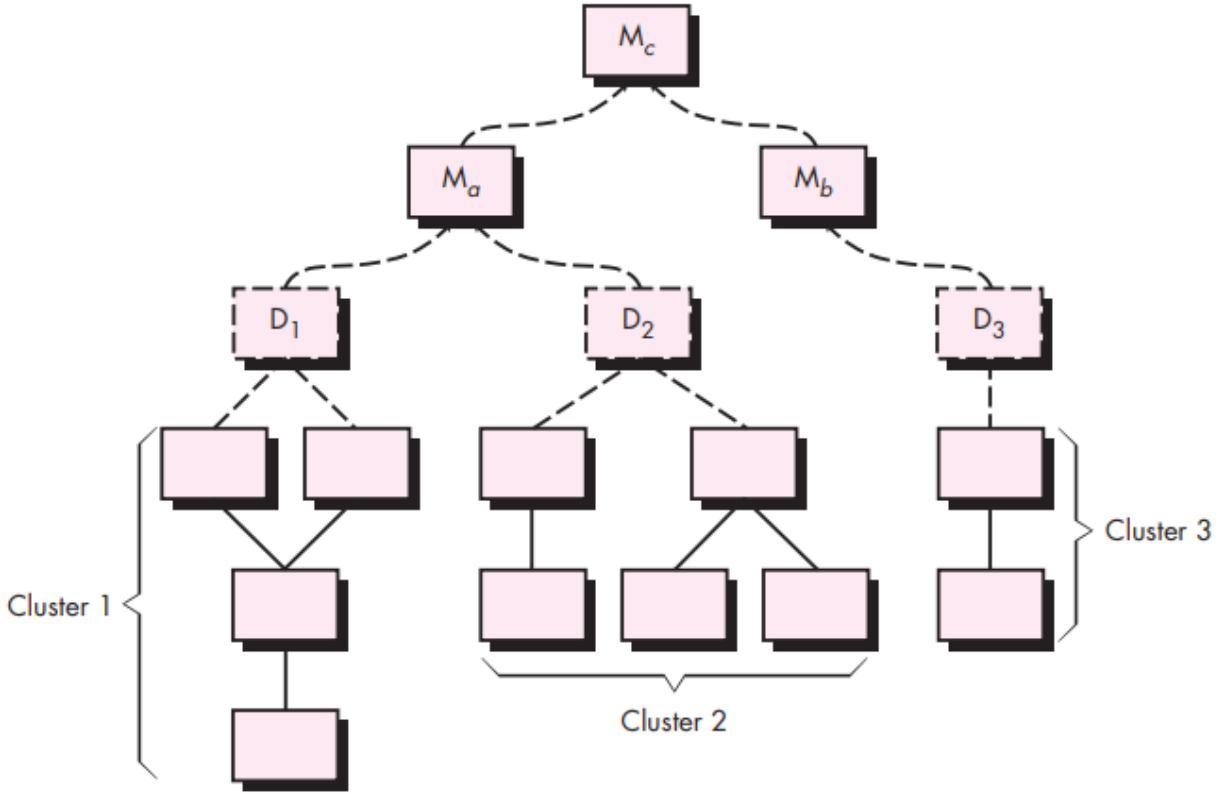
در این حالت در ابتدا به صورت عمقی پیش میروند و یا یک سطح را در ابتدا تست میکنند. مشکلی که این روش دارد این است که احتمال این وجود دارد که زیر ماژولهای مربوط به یک ماژول بزرگتر هنوز پیاده سازی نشده باشد.

رویکرد افزایشی

۲. Bottom-up integration

روش پایین به بالا:

ابتدا از زیرماژولها شروع کرده و با اتصال آنها به یکدیگر ماژولها و clusterهای بزرگتر ایجاد میشود.



رویکرد افزایشی

- **Regression testing**

با اضافه شدن هر ماژول به نرم افزار در مرحله یکپارچه سازی، تغییرات پیش امده در سایر زیربخش ها تست شود.

تست رگرسیون به معنای انجام دوباره بخشی از تست های قبلی است.

این test مشکل دو روش قبلی را برطرف میکند و لازم است به ازای هر تغییر مجددا محلهایی که تحت تاثیر قرار میگیرند بررسی شوند.

رویکرد افزایشی

- **Smoke testing**

با بزرگ شدن نرم افزار ممکن است قسمتهایی از برنامه که تحت تاثیر قرار میگیرند، بسیار زیاد شود و تست رگرسیون بسیار زمان بر است. از این روش استفاده میشود.

یک زیر مجموعه از موارد تست که کارکردهای اصلی سیستم را شامل میشود و مسیرهایی که احتمال خطا بیشتری دارد را تست میکند.

ایده این روش از سخت افزار میاید، که همه قطعات را وصل کرده و اجرا میکنیم. اگر از ان دود بلند شد، مشکل پیدا میشود.

تست سیستم system test

نرم افزار کامل شده مورد تست قرار میگیرد. تست سیستم در واقع مجموعه ای از تست های مختلف است که هدف اصلی آنها اجرای کامل سیستم مبتنی بر کامپیوتر است. اگرچه هر آزمایش هدف متفاوتی دارد، اما همه برای تأیید اینکه عناصر سیستم به درستی ادغام شده‌اند، کار می‌کنند.

تست پذیرش Acceptance test

- نرم افزار اماده شده به وسیله کاربران نهایی سیستم به منظور تعیین آمادگی برای استقرار نرم افزار تست میشود.

انواع دیگر تست :

- **Validation test**: هدف بررسی اعتبار نرم افزار از دید کاربر نهایی است.
- **Alpha test** : در محل تیم توسعه دهنده انجام میشود و کاربران با حضور اعضای تیم توسعه نرم افزار را تست میکنند و خطاها یادداشت و ثبت شده و برطرف میشوند. این نوع تست در محیط کنترل شده است.
- **Beta test** : در محل سازمان مشتری انجام میشود، برخلاف تست الفا، اعضای تیم توسعه حضور ندارند. کاربران مشکلات را ثبت کرده و بعدا گزارش میکنند.

انواع دیگر تست :

- **Configuration test**: تست پیکربندی – هدف تست در پیکربندی های مختلف و ارزیابی چگونگی عملکرد. پیکربندی یعنی در سیستم از چه ابزارهایی استفاده میشود (مثلاDBMS ، MySQL و ...) و متناسب با شرایط جدید پیکربندی مناسب تعیین میشود.
- **Recovery test**: بعد از بروز fault، نرم افزار بتواند دوباره فعالیت خود را آغاز کند. یا اینکه fault- tolerant باشد.
- در این تست شرایط failure برای سیستم فراهم شده و رفتار آن از نظر recovery تست میشود.

انواع دیگر تست :

- **Security test** : تست امنیت است و افراد متخصص انواع حملات را بررسی میکنند.
- **Stress testing** : تست ارزیابی، هدف تست کارایی سیستم در شرایط غیر معمول و افراطی یا بحرانی، مانند افزایش بیش از حد کاربران، ۱۰۰ تا کاربر را با ۱۰۰۰ کاربر شبیه سازی میکنند.
- **Performance test**: هدف ارزیابی قابل قبول بودن کارایی است. تست با استفاده از پیکربندی های مختلف در شرایط عملیاتی ثابت انجام میشود.
- با فرض اینکه ۱۰۰ تا کاربر یکسان داریم، یک بار برنامه را با oracle 10g بررسی کرده و بار دیگر با oracle 11g و کارایی سیستم را بررسی میکنیم.

انواع دیگر تست :

- **Loud test :** تست بار – ارزیابی قابل قبول بودن کارایی است. هدف، تست سیستم در شرایط متغییر عملیاتی مانند تعداد کاربران در حالتی که پیکربندی ثابت است، می باشد.
- **Illustration test :** تست نصب – ارزیابی نحوه نصب نرم افزار در پیکر بندی های مختلف و شرایط مختلف (کمبود حافظه اصلی). مثلا در هنگام نصب ابتدا وضعیت سیستم را بررسی میکنند.

- **Black box testing تست جعبه سیاه**

در این روش کاری با نحوه پیاده سازی ندارند و رفتار سیستم را در سطح واسط کاربری بررسی میکنند. بدون توجه به نحوه پیاده سازی و منطق داخلی نرم افزار

تست جعبه سفید White box testing

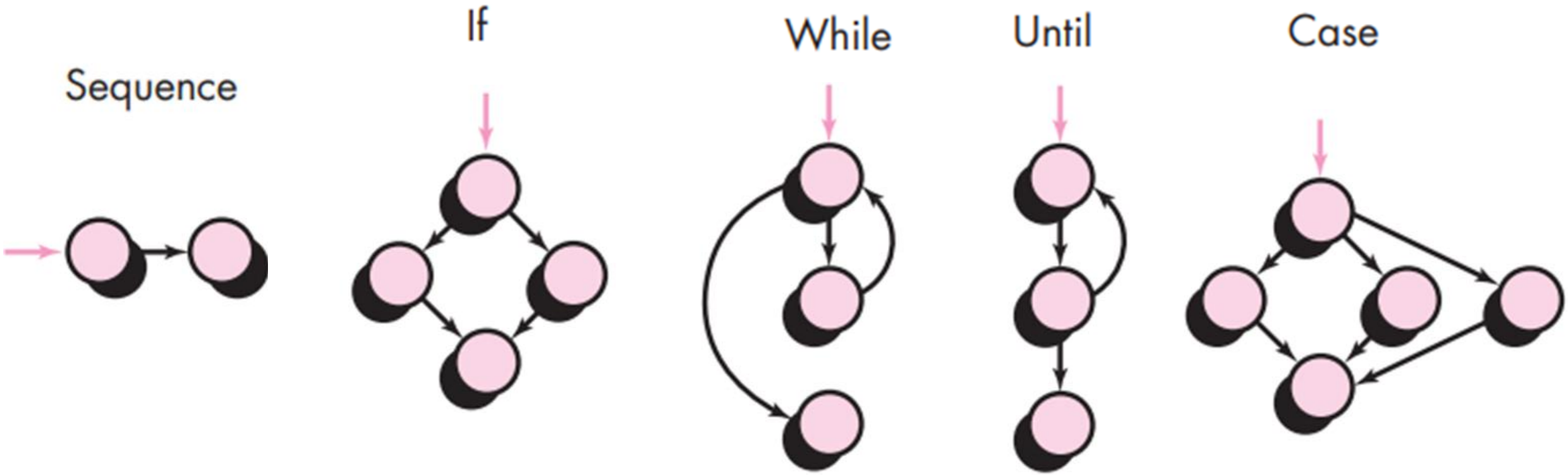
منطق داخلی نرم افزار و مسیرهای منطقی مختلف تست میشود. برای هر usecase لازم است.

همه مسیرهای ممکن را بررسی کنیم که حالت دهی بسیار زیادی ایجاد میشود و عملی نیست.

بنابراین میتوان قسمت های مهم و حساس با این روش تست شود و فضا را محدود ساخت.

Basic path testing

- هدف: یافتن یک مجموعه پایه از مسیرهای اجرایی و تهیه test case برای هر مسیر



- **قدم دوم : تعیین cyclomatic complexity**

تعداد نواحی داخلی که بسته هستند

cc : #region + 1

تعداد شرط

cc : #decision + 1

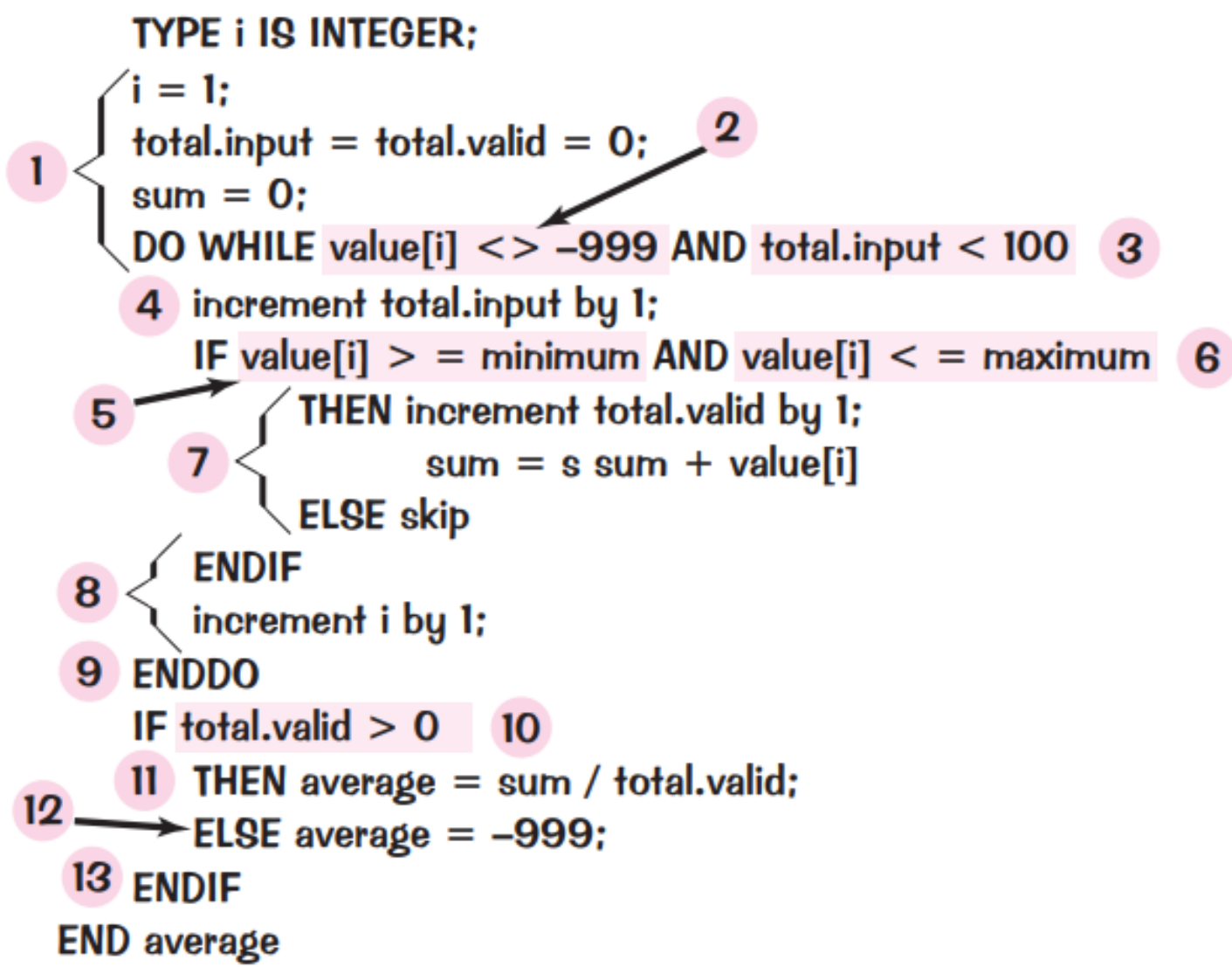
cc : L - N + 2

تعداد یال ها تعداد نودها

- **قدم سوم:** تعیین مسیرهای مستقل در برنامه

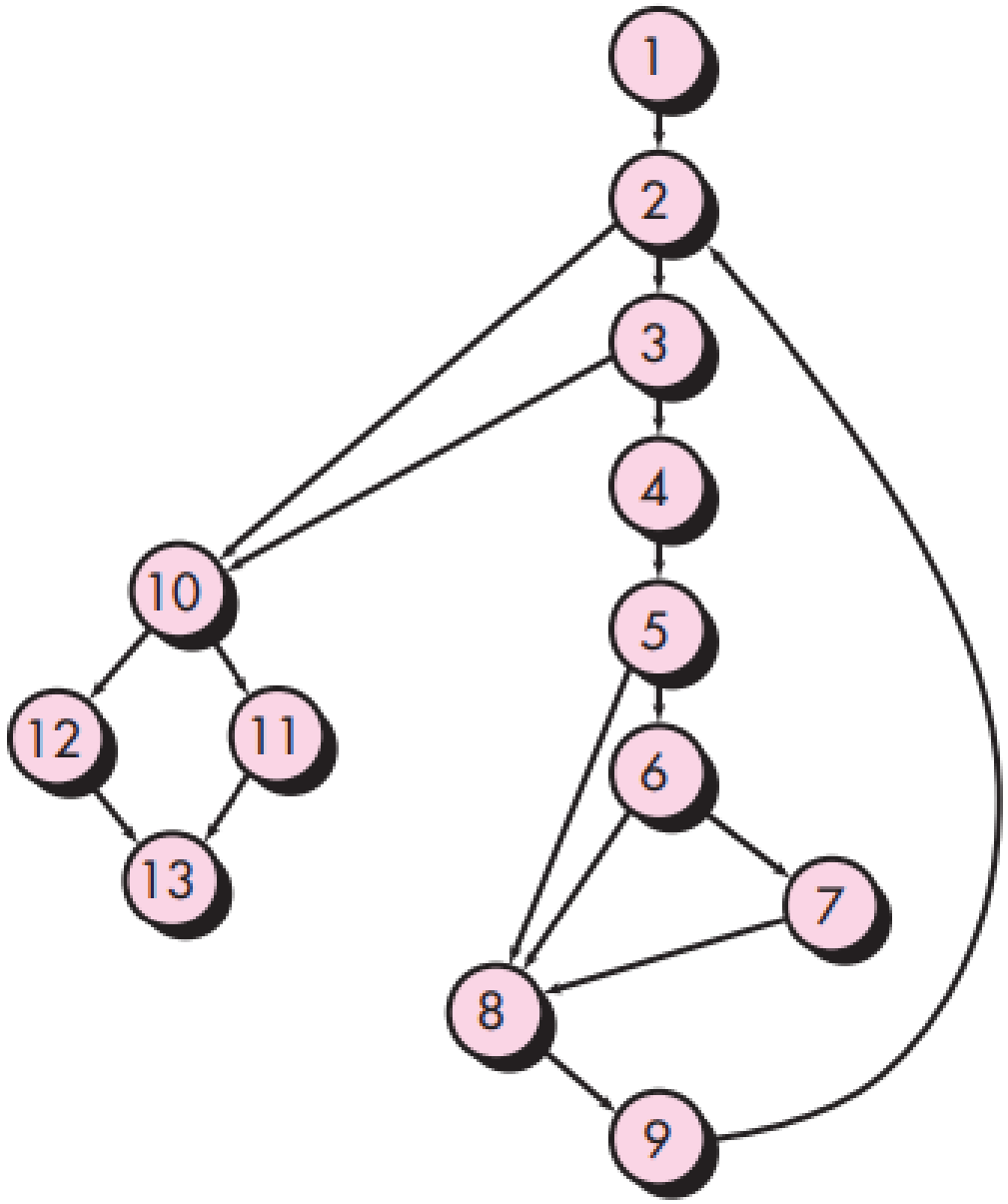
- **قدم چهارم :** استخراج test caseها

مجموعه test case باید به گونه ای باشد که همه مسیرها در مجموعه پایه حتما اجرا شود.


TYPE i IS INTEGER;
i = 1;
total.input = total.valid = 0;
sum = 0;
DO WHILE value[i] <> –999 AND total.input < 100
increment total.input by 1;
IF value[i] > = minimum AND value[i] < = maximum
THEN increment total.valid by 1;
sum = s sum + value[i]
ELSE skip
ENDIF
increment i by 1;
ENDDO
IF total.valid > 0
THEN average = sum / total.valid;
ELSE average = –999;
ENDIF
END average
1
2
3
4
5
6
7
8
9
10
11
12
13

```
     TYPE i IS INTEGER;
     i = 1;
1    total.input = total.valid = 0;
     sum = 0;
     DO WHILE value[i] <> -999 AND total.input < 100   3
4      increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum   6
5        THEN increment total.valid by 1;
7                sum = s sum + value[i]
         ELSE skip
8      ENDIF
       increment i by 1;
9    ENDDO
     IF total.valid > 0   10
11     THEN average = sum / total.valid;
12     ELSE average = -999;
13   ENDIF
   END average
```

2



$$V(G) = 6 \text{ regions}$$
$$V(G) = 17 \text{ edges} - 13 \text{ nodes} + 2 = 6$$
$$V(G) = 5 \text{ predicate nodes} + 1 = 6$$

Path 1: 1-2-10-11-13

Path 2: 1-2-10-12-13

Path 3: 1-2-3-10-11-13

Path 4: 1-2-3-4-5-8-9-2-. . .

Path 5: 1-2-3-4-5-6-8-9-2-. . .

Path 6: 1-2-3-4-5-6-7-8-9-2-. . .