## استراتيجيات فحص البرمجيات

## Software Testing Strtegies

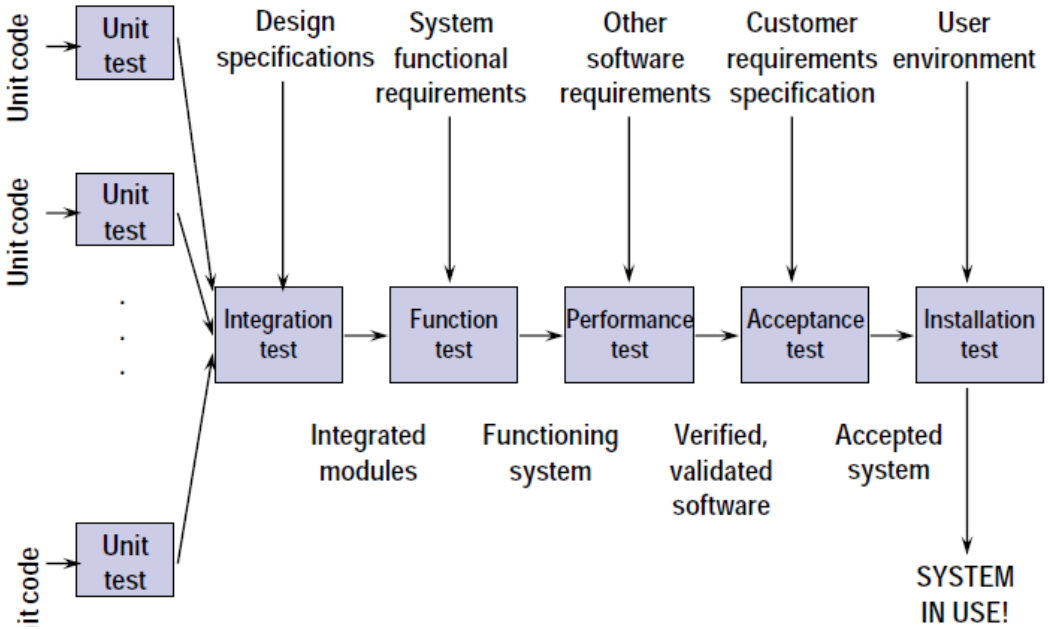
**مقدمة:**

غاية استراتيجية اختبار البرمجيات تكاملُ طرائق تصميم حالات اختبار برمجية في متتالية من المراحل المخططة جيداً، ينتج منها بناء ناجح للبرمجيات. وبنفس المستوى من الأهمية، تُعدُّ استراتيجية اختبار البرمجيات خريطة الطريق لكل من مطوّر البرمجيات، ومؤسسة ضمان الجودة، والزبون — وهي خريطة تصف المراحل المتبعة في الاختبار: متى تخطط تلك المراحل ومتى تنفّذ؟ وكم يلزم من الزمن والجهد والموارد؟ لذا يجب على استراتيجية اختبار البرمجيات، أيّاً كانت، أن تتضمن تخطيط الاختبار، وتصميم حالات الاختبار، وتنفيذ الاختبار، وتجميع المعطيات الناتجة وتقييمها.

لنتذكر معاً ما هو اختبار البرمجيات بعد الدروس السابقة، الاختبار هو تشغيل للبرنامج وتسجيل النتائج بهدف إيجاد الأخطاء والاختلافات مع المتطلبات. وقد يتم التشغيل تحت ظروف معينة لتقييم بعض جوانب النظام كالأداء والأمن.

**تنظيم اختبار البرمجيات:**

الاختبار هو مجموعة فعاليات يمكن أن تُخطط مقدماً وتُجرى بصورة نظامية. لهذا يجب تعريف *قالب template* لاختبار البرمجيات –وهو جملة مراحل يمكننا أن نضع فيها طرائق تصميم حالات اختبار محددة– في حالة إجرائية هندسة البرمجيات.




الشكل السابق يحدد التسلسل المقترح لاختبار النظام: حيث يتم اختبار كل جزء للنظام على حدة باستخدام اختبار الوحدة وبعدئذ نختبر بأن مختلف وحدات النظام تعمل مع بعضها البعض. ومن ثم يتم اختبار وظائف النظام مقارنة بالمتطلبات

الوظيفية ومن ثم يتم اختبار المتطلبات غير الوظيفية (الأداء، الأمن، .. حسب طبيعة النظام المطور) وفي النهاية يتم اختبار قبول النظام من المستخدم.

### 1 - اختبار الوحدة (Unit Test)

تُختبر *واجهة* المختزأ لضمان صحة تدفق المعلومة من (وإلى) وحدة البرنامج في قيد الاختبار. تُفحص بنية المعطيات المحلية لضمان أن المعطيات المختزنة مؤقتاً، تحافظ على تكاملها خلال كل مراحل تنفيذ خوارزمية معينة. تُختبر *الشروط الحدية* لضمان أن المختزآت تعمل كما ينبغي عند الحدود المعيَّنة، بغرض حدّ أو تقييد المعالجة. تُجرَّب جميع المسارات المستقلة (المسارات الأساسية) في بنية التحكم لضمان تنفيذ كل عبارات مختزأ معيَّن مرة واحدة على الأقل. أخيراً تُختبر مساراتُ تَولّي الأخطاء كافة.

يتم اختبار مكونات البرنامج كل على حدة للتأكد من صحتها وجودتها وتركز على:

- التأكد من بنية المعطيات اللازمة للاحتفاظ بالمعلومات التي يعالجها المكون.
- منطق عمل المكون (اختبار الصندوق الأبيض والأسود).
- واجهة إدخال البيانات (عدد الوسائط وأنواعها).

ماهي الوحدة أو مكون من مكونات البرنامج:
يمكن أن تكون أصغر قطعة قابلة للترجمة لوحدها.

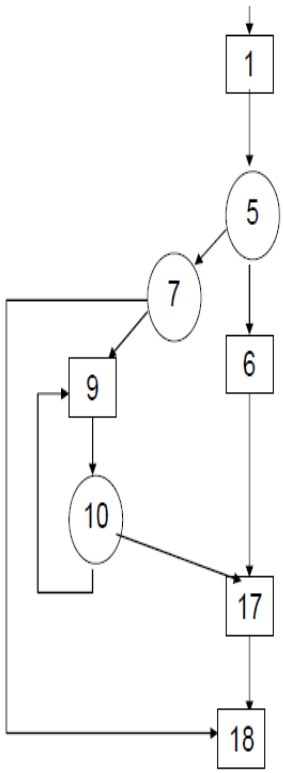
- يمكن أن تكون تابع برمجي.
- يمكن أن تكون أصغر كود مطور من قبل شخص واحد.

وغالباً ما يقوم بالاختبار الشخص الذي طور المكون. ويقوم بتصميم حالات اختبار لما يأتي:

- تحديد صفوف التكافؤ لمعطيات الدخل.
- تحديد الحدود صفوف التكافؤ لمعطيات الدخل.
- تحليل مسارات خوارزمية عمل الوحدة الرئيسية.

لنستعيد معاً تقنيات اختبار البرمجيات (اختبارات الصندوق ألأبيض وألأسود) من الدرس الماضي من خلال مثال لاختبار الوحدة (Unit Testing) لبرنامج إيجاد القاسم المشترك الأكبر.

```
1.  function gcd (int a, int b) {
2.     int temp, value;
3.     a := abs(a);
4.     b := abs(b);
5.     if (a = 0) then
6.          value := b;    // b is the GCD
7.     else if (b = 0) then
8.          raise exception;
9.     else
10.          loop
11.               temp := b;
12.               b := a mod b;
13.               a := temp;
14.           until (b  =  0)
15.           value := a;
16.     end if;
17.     return value;
18.  end gcd
```



لنحدد صفوف التكافؤ (الصندوق الأسود)
خوارزمية إيجاد القاسم المشترك الأكبر يجب أن تقبل أي عددين صحيحين كدخل :كل واحد ممكن يكون 0، موجب، سالب (نجرب كل التركيبات التي يمكن أن نصادفها):

- classes:
  - a < 0 and b < 0, a < 0 and b > 0, a > 0 and b < 0
  - a > 0 and b > 0, a = 0 and b < 0, a = 0 and b > 0
  - a > 0 and b = 0, a < 0 and b = 0, a = 0 and b = 0

لنحدد القيم الحدية (على الحدود) (الصندوق الأسود)

- Boundary Values
  - a = $-2^{31}$, -1, 0, 1, $2^{31}-1$ and b = $-2^{31}$, -1, 0, 1, $2^{31}-1$

لنحدد المسارات الرئيسية للخوارزمية (الصندوق الأبيض)

- Basic Path Set
  - V(G) = 4

(1,5,7,9,10,9,10,17,18) (1,5,7,9,10,17,18) (1,5,7,18) (1,5,6,17,18)

فتكون حالات الاختبار هي:

| Test Description / Data | Expected Results | Test Experience / Actual Results |
|---|---|---|
| Basic Path Set | | |
| path (1,5,6,17,18) ➔ (0, 15) | 15 | |
| path (1,5,7,18) ➔ (15, 0) | 15 | |
| path (1,5,7,9,10,17,18) ➔ (30, 15) | 15 | |
| path (1,5,7,9,10,9,10,17,18) ➔ (15, 30) | 15 | |
| Equivalence Classes | | |
| a < 0 and b < 0 ➔ (-27, -45) | 9 | |
| a < 0 and b > 0 ➔ (-72, 100) | 4 | |
| a > 0 and b < 0 ➔ (121, -45) | 1 | |
| a > 0 and b > 0 ➔ (420, 252) | 28 | |
| a = 0 and b < 0 ➔ (0, -45) | 45 | |
| a = 0 and b > 0 ➔ (0 , 45) | 45 | |
| a > 0 and b = 0 ➔ (-27, 0) | 27 | |
| a > 0 and b = 0 ➔ (27, 0) | 27 | |
| a = 0 and b = 0 ➔ (0 , 0) | exception raised | |
| Boundary Points | | |
| (1 , 0) | 1 | |
| (-1 , 0) | 1 | |
| (0 , 1) | 1 | |
| (0 , -1) | 1 | |
| (0 , 0) (redundant) | exception raised | |
| (1, 1) | 1 | |
| (1, -1) | 1 | |
| (-1, 1) | 1 | |
| (-1, -1) | 1 | |

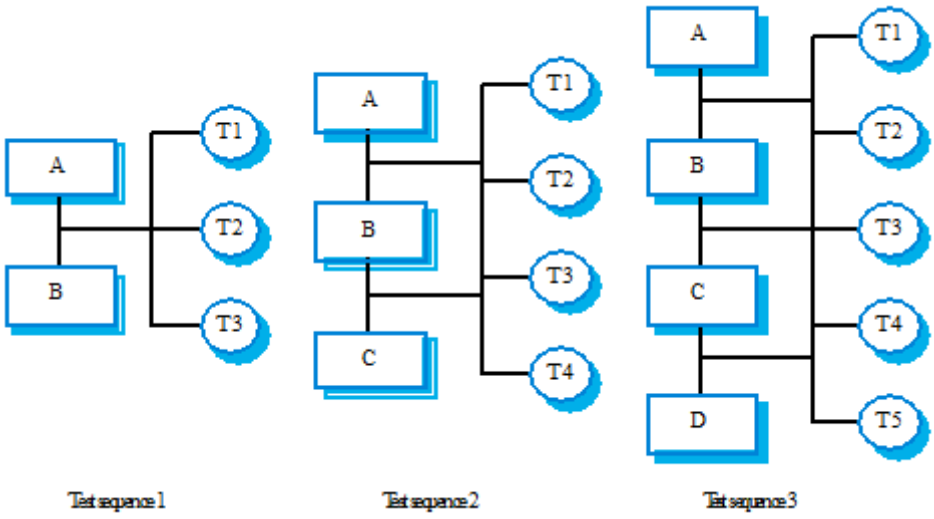
**2 - اختبار التكامل (Integration Test)**

يتم اختبار عمل مكونات النظام مع بعضها البعض. ويتم التركيز على اكتشاف الأخطاء :

- تصميم وبنية النظام
- تفاعل عمليات واجهات المكونات
- استخدام وتشارك موارد بيئة العمل

الاختبارات الانكفائية Regression Test:

عند إضافة مجتزأ جديد في اختبار تكامل، تتغير البرمجيات. وتنشأ مسارات تدفق معطيات جديدة، وقد تحدث عمليات دخل/خرج جديدة، ويستدعى منطق جديد للتحكم. قد تسبب هذه التغيرات مشاكل للوظائف التي كانت تعمل من دون خلل سابقاً. يعني *الاختبار الانكفائي regression testing* في سياق استراتيجية اختبار التكامل: إعادة تنفيذ بعض المجموعات الجزئية من الاختبارات المنفّذة سابقاً، لضمان عدم تسبب التعديلات بانتشار آثار جانبية غير مقصودة.




يتم أولاً اختبار تكامل المكونين ألأهم في البرمجية A وB بثلاثة حالات اختبار (T1, T2, T3) –اصبر قليلاً وسأشرح ما تعنيه حالة الاختبار-.

ومن ثم يتم اختبار تكامل المكونات التالية في البرمجية A وB و C بأربعة حالات اختبار (T1, T2, T3, T4). تم إعادة حالات الاختبار T1, T2, T3 بالرغم من أن T4 هي التي تخص مكاملة المكون C مع المجموعة السابقة. وذلك لربما إن إضافة المكون C أثرت على ترابط A وB فيجب التأكد من الاختبارات T1, T2, T3 مازالت صحيحة

ومن ثم يتم اختبار تكامل المكونات التالية في البرمجية A وB و C و D بخمس حالات اختبار T1, T2, T3, (T4, T5). تم إعادة حالات الاختبار T1, T2, T3, T4 بالرغم من أن T5 هي التي تخص مكاملة المكون D مع المجموعة السابقة. وذلك لربما إن إضافة المكون D أثرت على ترابط A وB و C فيجب التأكد من الاختبارات T1, T2, T3, T4 مازالت صحيحة.

في سياق أوسع، ينجم عن الاختبارات الناجحة (من أي نوع كانت) اكتشاف الأخطاء، وهذه الأخطاء يجب أن تُصحّح. عندما تُصحّح البرمجيات تتغير بعض مكونات تشكيلة البرمجية (البرنامج أو توثيقه أو المعطيات التي يتقبّلها). *الاختبار الانكفائي* هو فعالية تساعد على ضمان عدم تسبب التغيرات (الناجمة عن الاختبارات أو أسباب أخرى) في إدخال سلوك غير مقصود أو أخطاء إضافية.

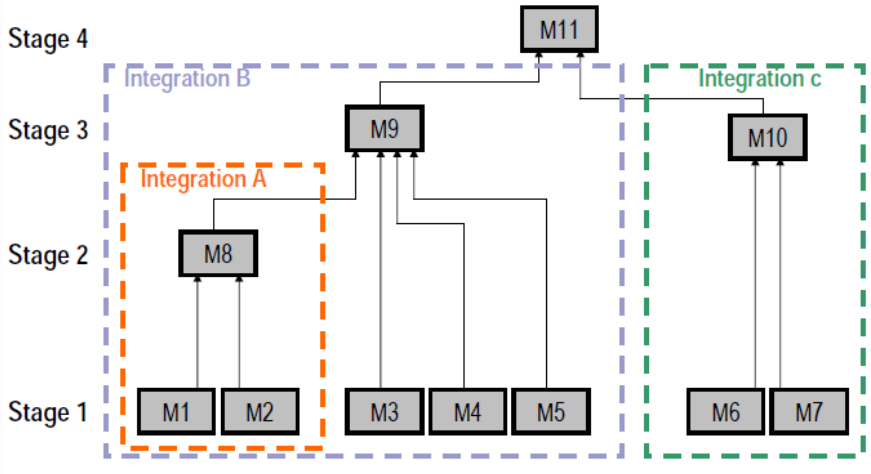
ما إنْ يُستكمل اختبار الوحدة للمجتزآت، حتى يمكن لمبتدئ في عالم البرمجيات أن يسأل سؤالاً، مشروعاً من حيث الظاهر: "إذا كان كل مجتزأ يعمل وحده، فلماذا نشكّك في أن المجتزآت لن تعمل مجتمعة؟". المشكلة بالطبع هي "جمع المجتزآت معاً" –مشكلة التواجُه. قد تضيع المعطيات في واجهة ما، إذ قد يؤثّر مجتزأ، ودون قصد، عكسياً في مجتزأ آخر. وقد لا تعطي الإجراءات الجزئية، عندما تُضمُّ، الوظيفةَ الأساسية المطلوبة. وقد يتضخم عدم الدقة، المقبولة على حدة، لتبلغ مستويات غير مقبولة. وقد تتسبب بنى المعطيات العامة بمشاكل –ويا للأسف، فهذه القائمة مستمرة ...

اختبار التكامل هو تقنية نظامية لبناء بنية برنامج، في أثناء إجراء الاختبارات لكشف الأخطاء المتعلقة بالتواجه. الغاية هي الانطلاق من مجتزآت مختبرة بطريقة اختبار الوحدة، وبناء بنية برنامج التي يمليها التصميم.

ملاحظة: أحياناً، نحتاج عند اختبار مكون يستخدم مكون آخر لاستدعاء عمليات من المكون الآخر (المنفذ بحده الأبسط: يعني لنفرض أن المكون الأصلي يستدعي عملية عليه للقيام بعمل ما نتيجته معروفة بالنسبة لهذه العملية فيقتصر التنفيذ على عملية return للقيمة) ويدعى المكون المنفذ بحده الأبسط ب Stub.

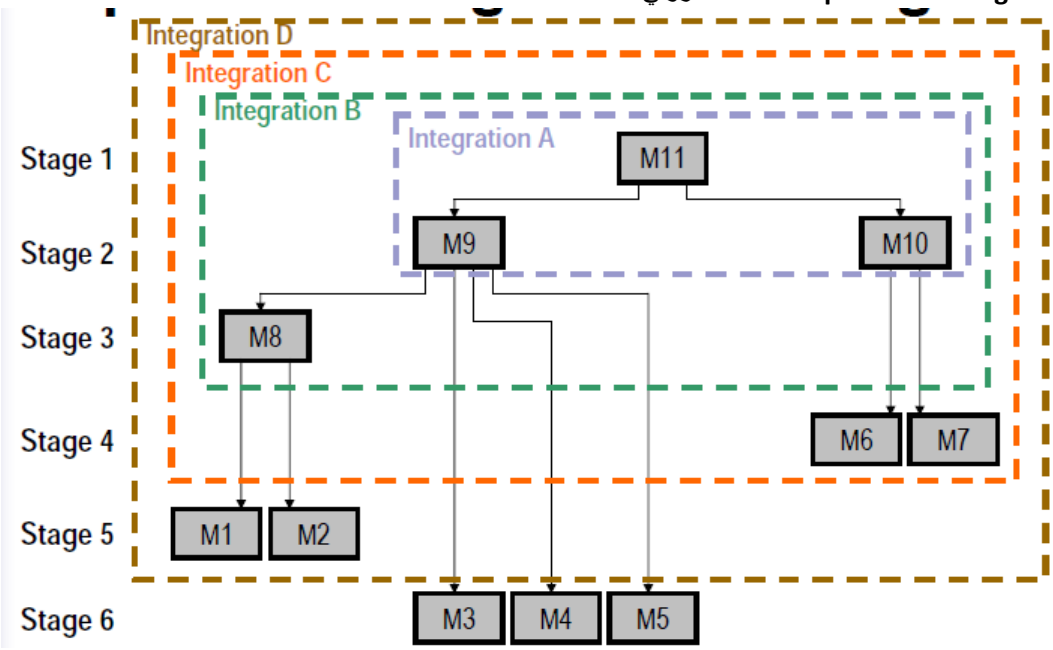
يوجد ثلاث طرق لاختبار تكامل مكونات النظام:

- الطريقة التزايدية Incremental integration:
  - التكامل الصعودي: **Bottom-up testing**



يتم في البداية اختبار المكون M8 مع ما يحتاج من مكونات M1, M2. ثم نختبر المكون M9 مع ما يحتاج من مكونات M8, M3, M4, M5. ثم نختبر المكون M10 مع المكونين M6 و M7. في النهاية نختبر المكون M11 مع ما يحتاج من مكونات M10, M9.

- **Top-down testing** التكامل النزولي:




في البداية يتم اختبار المكون M11 مع stub للمكون M9 و stub للمكون M10، بعد نجاح الاختبار يتم اختبار المكون M9 مع stub للمكون M8 من ثم اختبار المكون M10 مع stub للمكون M6, M7 ومن ثم نختبر المكون M8 مع stub للمكونيين M1, M2 وفي النهاية نختبر المكون M9 مع stub للمكون M3, M4, M5.

يتم استخدام هذه الطريقة عندما يكون التطوير مقاد بالاختبار أو في المشاريع الضخمة. حيث يتم اختبار الوظائف النهائية للنظام حتى قبل أن يتم تطوير النظم الفرعية وإنما يساعد هذا الاختبار على التأكد من صحة واجهات التخاطب بين المكون الأساسي والنظم الفرعية ونكرر العمل بنفس الطريقة بعدئذ في كل نظام فرعي ننفذ المكون الرئيسي له ونختبر تفاعله مع واجهات مكوناته وعند نجاح الاختبار يتم تحقيق مكوناته واختبار تفاعلها مع مكوناتها الجزئية.... وهكذا.

- الطريقة اللاتزايدية non-incremental integration:
  - **Big bang testing**:

يتم اختبار تفاعل مكونات النظام مع بعضها البعض دفعة واحدة وهو يناسب للأنظمة الصغيرة حيث لا بنية هرمية عميقة.