بسم الله الرحمن الرحيم

نحو مبادئ برمجية صحيحة

تأليف/ معتز عبدالعظيم الطاهر
كود لبرمجيات الكمبيوتر

16 محرم 1439

7 أكتوبر 2017

المحتويات

مقدمة

بسم الله الرحمن الرحيم، والصلاة والسلام على أشرف الأنبياء والمرسلين، نبينا محمد وعلى آله وصحبه أجمعين.

أما بعد، فإن هذا الكتاب يهدف إلى شرح أفضل الطرق لتصميم وكتابة برامج بطريقة علمية متوافقة مع مبادئ هندسة البرمجيات مما يحقق الكفاءة في إنتاج برامج وأنظمة كمبيوتر ذات جودة عالية. والكفاءة مقصود بها جهد ووقت أقل في كافة دورة تطوير البرامج من تحليل، وتصميم، وبرمجة، وصيانة وإضافة، واختبار، وغيرها.

تمت كتابة مادة هذا الكتاب بعد قراءة ودراسة من عدة مصادر في الإنترنت والتي تتحدث عن الطريقة المثلى ومفاهيم ومبادئ البرمجة الصحيحة، ثم تم ربطها ومقارنتها مع الواقع العملي فكانت النتيجة أن جمعت مادة الكتاب بين العملي والنظري بطريقة مبسطة وقد تم اختصار بعضها لتكون كلها مفهومة ويسهل تطبيقها وليس فيها غموض.

المؤلف: معتز عبدالعظيم

عملت مبرمج كمبيوتر ومحلل ومعماري أنظمة وقد قمت بتطوير عدد كبير من البرامج طوال العشرين عام الماضية، و ما زلت اعمل إلى الآن - بفضل الله - كمطور برامج ومعماري. وفي الآونة الأخيرة أصبحت اهتم كثيراً بالتصميم الداخلي ومعمارية الأنظمة وقد درست المبادئ الصحيحة للبرمجة، وهدفي الآن هو تطبيق تلك المفاهيم في البرامج قيد التطوير.

التحليل

الهدف من التحليل هو دراسة المتطلبات بصورة مفصّلة لتحديد الأهداف المطلوبة من النظام وإيضاحها للفريق المسؤول عن التنفيذ. وهو يأتي مباشرة بعد الموافقة في أي مشروع برمجي وهو البداية الحقيقية لتطوير البرامج والأنظمة المختلفة.

لابد أن يُترك للتحليل وقتاً كافياً ويقوم به ذوو الخبرة في تطوير البرامج. ولابد أن يتناسب المجهود التحليلي مع حجم المشروع البرمجي وعمره الافتراضي، فإذا كان الناتج نظام يعمل في مؤسسة يستمر معها مع استمرارها فلابد أن يكون له حيز مقدر من التحليل، فإذا افترضنا أنه لابد أن يعمل عشرة أعوام قبل تغييره أو استبداله بنظام أحدث أو حتى إعادة تصميمه، فلابد أن يكون زمن التحليل يستمر أسابيع لتغطية كل المتغيرات والميزات التي يحتاج إليها المستخدمون والنظام طوال العشر سنوات المقبلة.

في الواقع العملي يستعجل المبرمج التحليل ليبدأ مباشرة في كتابة الكود، لكن هذا الاستعجال إما أن يؤدي إلى وصول هذا المُنتج إلى نهاية عمره الافتراضي بصورة اسرع: مثلاً يعمل لعامين فقط ثم لا يتحمل التطوير والإضافات. أو يحدث أنه بعد فترة يحتاج لإعادة هيكلة وإعادة تصميم لمواكبة المتغيرات التي ظهرت بعد الاستخدام والمتغيرات التي تم إغفالها أثناء التحليل، والأسواء من ذلك أن يعمل النظام ويستمر تطويره لكن بتكلفة عالية بسبب تصميمه غير الصحيح.

التصميم

التصميم هو عملية تخطيط لتحديد المواصفات الفنية والأجزاء البرمجية المختلفة التي سوف تحقق هدف النظام وعلاقاتها مع بعضها.
التصميم هو من أهم الخطوات قبل البداية في كتابة الكود، وتزداد أهميته مع زيادة حجم النظام المراد تطويره. ويتم إنتاج مخططات تشمل هيكل النظام والمعمارية والعلاقات بين الأجزاء المختلفة من النظام.
التصميم يحتاج لمطور برامج أو مصمم أومعماري لديه خبرة في التصميم وخبرة في نوعية النظام المراد تطويره. والتصميم الجيد للنظام يضمن نجاحه وأن يحقق الهدف الذي بُني من أجله، ويُساهم في تقليل الجهد المبذول في التطوير والدعم.
إذا لم يتم التصميم بصورة صحيحة أو مكتملة فينتج عن ذلك برامج غير ناجحة ومكلفة وتسبب إضاعة للموارد أثناء تنفيذ المشروع البرمجي وحتى بعد تشغليه.

كتابة الكود

تأتي مرحلة كتابة الكود عادة بعد الانتهاء من التصميم، لكن أحياناً تبدأ بعد التحليل أو أثناءه لغرض إثبات فكرة المشروع prove of concept وذلك باختيار أصعب جزء برمجي في النظام أو أهم جزء لتمثيله في شكل برنامج. مثلاً إذا كان النظام لمراقبة ماكينات تعمل في مصنع ما، فيمكن عمل برنامج ليتصل مع الماكينة ويعرض معلوماتها، مثلاً ماكينة بها وصلة RS-232 أو وصلة شبكة فيقوم المطورون بكتابة برنامج مصغر بعد دراسة بروتوكول الاتصال بهذه الماكينة، الهدف من هذا البرنامج إقناع العميل بأن النظام يمكن عمله وأن هذه الجهة البرمجية قادرة على إنجاز المشروع.

كذلك فإن السبب الآخر لكتابة الكود في مرحلة مبكرة هو عمل نموذج prototype وهو جزء يعمل من النظام لعرض الفكرة و الشكل النهائي لتأكيد أن المشروع قد تم تحليله بطريقة صحيحة.
يتم بعد الانتهاء من التصميم تقسيم المشروع بين عدد من المبرمجين والمطورين ليعمل كل واحد أو مجموعة على جزئية معينة يتم ربطها لاحقاً لتعمل كنظام متكامل.
هذه المرحلة تأخذ الوقت الأطول في المشروع في غالب المشروعات البرمجية.

مرحلة الاختبار

يوجد عدد من أنواع الاختبار: نذكر منها اختبار الوحدات unit testing والذي يقوم المبرمج بعمله أثناء أو بعد الانتهاء من أي وحدة برمجية، مثلاً فئة أو جزئية متكاملة، أو شاشة أو غيرها من المكونات المستقلة التي يمكن عمل اختبار مستقل لها. الغرض من هذا الاختبار التأكد من أن هذه الوحدة التي تم الانتهاء منها تعمل بطريقة صحيحة قبل دمجها مع باقي الوحدات حتى لا ينتج خطأ أو مشكلة يصعب معرفة مكانها، حيث أن الأسهل هو اختبار الوحدات المصغرة.
النوع الثاني هو اختبار التكامل integration testing وذلك بعد ربط وحدات برمجية مع بعضها لتشكل جزئية أكبر أو تشكل النظام كاملاً، فيتم عمل اختبار بتشغيل عدد من الوحدات مع بعضها للتأكد من أنها تعمل بالطريقة المطلوبة التي صممت من أجلها.

طريقة التصميم وطريقة البرمجة

في مرحلة تطوير المشروع أو بعد اكتماله فإن طريقة التصميم وطريقة والبرمجة تظهر للعيان في شكل نجاح المشروع أو فشله أو نجاحه إلى حد ما أو فشله إلى حد ما. ولقياس مدى نجاح أو فشل المشروع البرمجي قمنا بكتابة دلالات تدل على النجاح وأخرى تدل على الفشل في الفقرتين التاليتين:

دلالات سوء التصميم والبرمجة

سوء التصميم وسوء طريقة كتابة الكود تظهر أثناء تطوير البرنامج وينتج عنه تأخير في الوصول إلى النتائج المرجوة من المشروع. كذلك تظهر بشكل جلي بعد الانتهاء من التطوير وبداية التشغيل و تظهر مع استمرار التطوير والتعديلات. ومن أهم دلالات أن التصميم والبرمجة لم تتم بالطريقة الصحيحة هي:

1. تأخير وتعثر في الانتهاء من الجزئيات البرمجية أو تأخير في تطوير المشروع ككل.

2. قلة الاعتمادية: أن لا يعمل النظام وفقاً لما صمم له، مثل أن يعطي نتائج غير صحيحة في حال أن المدخلات كانت صحيحة

3. التغيرات والإضافات البسيطة تأخذ وقتاً طويلاً لتنفيذها

4. التغيير في جزئية ما تنتج عنها مشكلة أو تغيير في جزئية أخرى

5. بعض المتطلبات أو التغييرات يصعب جداً تنفيذها و أحياناً تكون مستحيلة بسبب التصميم الحالي للنظام

6. بعض التغييرات لا يمكن تنفيذها إلا بإعادة هيكلية و معمارية النظام

7. يصعب تتبع المشاكل وحلها

8. يصعب دخول مبرمجين ومطورين جدد لتطوير النظام

9. يصعب عمل اختبار جزئي unit testing

10. الاعتمادية الكبيرة للبيئة: مثل أن النظام لا يعمل إلا في نظام تشغيل معين أو نُسخة معينة منه، أو يعتمد على نظام معين، مثل أنظمة الحسابات، فإذا حدث تغيير لهذه الأنظمة يتوقف النظام عن العمل ولا يمكن ربطه بنظام بديل آخر

11. عدم القدرة على تشغيل النظام في أكثر من مؤسسة: ليس به مرونة ليعمل في بيئات استخدام مختلفة ولم يتم تصميمه ليعمل على متطلبات مختلفة.

12. الحاجة لعمل فرع مختلف من النظام يتم تطويره بمعزل عن الفرع الرئيس لخدمة أكثر من مؤسسة أو أكثر من قسم

13. عدم القدرة على تحمل زيادة الاستخدام، سواءً زيادة عدد المستخدمين أو العمليات التي تجرى عليه في الساعة، حتى مع وجود مخدمات إضافية، حيث أن بعض الأنظمة لا تستطيع الاستفادة من زيادة مخدمات جديدة للعمل في النظام. عندها نقول أن النظام not scalable

14. وجود أجزاء برمجية في غير مكان الصحيح: مثلاً وجود business log في قاعدة البيانات أو في واجهة المستخدم presentation layer حيث أن مكانها الصحيح هو الجزء الوسط middle layer من هيكل النظام.

15. تعطل النظام أو إظهاره لأخطاء أو نتائج غير صحيحة أو غير مناسبة عند حدوث استثناءات عادية أثناء التشغيل، مثل انقطاع الشبكة

وعلى العكس تماماً هو التصميم الصحيح الذي يحقق الأهداف المذكورة في الفقرة التالية:

دلالات التصميم والبرمجة بطريقة صحيحة

التصميم والبرمجة بطريقة مثالية و باستخدام مبادئ علمية ينتج عنها نظام ناجح به الميزات التالية:

1. سرعة التطوير: أن تسير مرحلة تطوير النظام بطريقة سلسة تتناسب مع حجم الأجزاء البرمجية التي يتم تطويرها

2. الاعتمادية: أن يعمل النظام وفقاً لما صمم له، ويعطي نتائج صحيحة في حال أن المدخلات كانت صحيحة

3. سهولة التعديل والإضافات في النظام

4. سرعة معرفة المشاكل وحلها

5. إمكانية إضافة أجزاء جديدة وميزات دون تغيير في هيكل النظام

6. إمكانية تشغيل نفس النظام في مؤسسات مختلفة دون تغيير في الكود الأساسي للنظام: أي يكون هُناك فرع واحد فقط من كود النظام يهدف للعمل في كل المؤسسات المستفيدة

7. إمكانية دخول مبرمجين جدد في النظام لتطوير أجزاء جديدة أو العمل على أجزاء قديمة

8. إمكانية الاستفادة من مخدمات أخرى وذلك بتوزيع حمل التشغيل عند زيادة الاستخدام، وقابليته لتنفيذ مهام أكثر أو مستخدمين أكثر، عندها نقول أن النظام scalable

9. عمل النظام للفترة المطلوبة أو لفترة أطول من الفترة المصمم أن يعمل لها

10. يقوم النظام بالتعامل مع الاستثناءات التي تحدث أثناء التشغيل بطريقة صحيحة ومتوقعة، مثل أن تظهر رسالة بأن هُناك عطل في الشبكة في حالة تعطل الشبكة.

11. سهولة عمل الاختبارات للأجزاء المختلفة من النظام

المبادئ الصحيحة للبرمجة:

توجد عدد من المفاهيم والوسائل تسمى مبادئ البرمجة programming principles تهدف لكتابة الكود بطريقة معينة وتقسيم أجزاءه المختلفة بطريقة علمية صحيحة تساهم في أن يحقق الأهداف السابقة. من هذه المبادئ:

1. التجريد abstraction
2. إعادة استخدام الكود code re-usability
3. البساطة simplicity
4. الوظيفة الواحدة single responsibility
5. إخفاء التفاصيل hide implementation details
6. الفتح والإغلاق open-closed principle
7. التماسك Cohesion
8. فك الارتباط decoupling
9. القياسية في كتابة الكود code standardization

1. مبدأ التجريد abstraction

التجريد في عالم هندسة البرمجيات من المواضيع المهمة، حيث يحقق عدة أهداف سوف نتحدث عنها لاحقاً بإذن الله في هذه الفقرة. لكن قبل أن نتكلم عنها من ناحية برمجية دعونا نشرحها كمفهوم عام للتجريد في حياتنا اليومية.

التجريد هو التبسيط، وهو عكس التعقيد، وهو تقسيم المشكلة أو التصميم، أو ألأشياء إلى عدة مكونات أكثر بساطة وبدائية. مثلاً جهاز الحاسوب إذا كان قطعة واحدة مصمتة فهو شيء معقد، إذا تلف منه جزء لابد أن يتم استبداله كاملاً، أما التجريد فهو يعني أن يتم فصل أجزاء مختلفة لتشكل مكونات مستقلة يتم تركيبها ووصلها مع بعضها بطريقة اتصال معينة لتعمل في النهاية كأنها جهاز واحد.

وكمقارنة بين جهاز حاسوب أكثر تجريداً وواحد آخر أقل تجريداً: نقارن جهاز سطح المكتب بالجهاز

المحمول (لابتوب) حيث أن الأول (جهاز سطح المكتب) يتكون من أجزاء منفصلة سهل فكها وتغييرها، مثل لوحة المفاتيح، و الفأرة، والشاشة، ووحدة المعالجة المركزية، والسماعات. فإذا تلف أي جزء منها، نستطيع شراء بديل له بسهولة، حيث يمكنك شراء أي لوحة مفاتيح من أي نوع، كذلك يمكن ترقية الشاشة لشاشة ذات مساحة عرض أكبر دون أن نقوم بتغيير الحاسوب كاملاً، كذلك يمكنك استعارة أحد أجزاء الحاسوب المكتبي لتشغيله في حاسوب آخر. أما اللابتوب فهو أقل تجريداً حيث يميل لأن يكون قطعة واحدة، لذلك لا تستطيع تغيير الشاشة إلى شاشة أكبر، ولا تستطيع استعارة الماوس المثبت في اللابتوب لاستخدامه مع جهاز آخر. لكن على الأقل يوجد نوع من التجريد وذلك بإمكانية ترقية الذاكرة، أوالقرص الصلب. أما الموبايل فهو أقل تجريداً من اللابتوب، حيث يميل أكثر ليكون جهاز واحد مصمت لذلك إذا تلف فإنه أصعب من ناحية الصيانة، مثلاً لا يمكن تغيير الذاكرة الداخلية إذا تلفت بسهولة. فنجد أن الأجهزة الأكثر تجريداً تتمتع بعمر خدمي أكبر، وذلك لسهولة الصيانة والترقية. فالجهاز المكتبي لديه عمر أكبر من اللابتوب والذي لديه عمر أكبر من الموبايل، وذلك بمفهوم التجريد وليس بمفهوم جودة الصناعة، نحن نقارن بين منتجات ذات جودة واحدة في هذا المثال.

التجريد في البرمجة هو نفس المفهوم الذي شرحناه، حيث يهدف إلى تقسيم الأنظمة الكبيرة إلى برامج أصغر ذات وظائف محددة، فبدلاً من أن يكون النظام هو عبارة عن برنامج واحد به شاشات الاستخدام، والتقارير، ويحتوي على البيانات، فإن الأفضل تقسيمه إلى طبقات: طبقة تتعامل مع المستخدم، نسميها واجهة المستخدم، وطبقة أخرى لاستقبال طلبات المستخدم وتنفيذها، وطبقة أخرى لتخزين البيانات، ويمكن تثبيت كل طبقة في مخدم منفصل حين يكون عدد المستخدمين كبير.

كذلك يمكن تقسيم البرامج إلى وحدات أكثر تخصصية من ناحية الوظائف، فمثلاً نظام حسابات وشيكات ومخازن، كل وظيفة يكون لها برنامج أو وحدة برمجية منفصلة، يمكن تطويرها بطريقة مستقلة عن الأجزاء الأخرى. يوجد أيضاً نوع من التجريد على مستوى الإجراءات وعلى مستوى كتابة الكود، فبدلاً من كتابة إجراء طويل ومعقد يمكننا تقسيم هذا الإجراء إلى عدة أجراءات أولية بسيطة، كل واحد منها يهدف إلى تحقيق وظيفة واحدة، يمكن إعادة استخدامها أكثر من مرة.

في هذه المقالة سوف نقوم بعرض مثال للنوع الأخير من التجريد، وهو على مستوى الإجراءات. هذا الإجراء (بلغة جافا) يقوم بقراءة رسائل

مستلمة والتي لم يتم معالجتها بعد من قاعدة بيانات ثم يقوم بوضعها في شكل مصفوفة. وفي المثال هذا الإجراء لا يحقق التجريد، حيث أن به بعض التعقيد ويقوم بعمل أكثر من وظيفة: 1 – تجهيز الطلب لقاعدة البينات، 2 – ثم إرسال الطلب، 3- ثم قراءة السجلات ووضعها في المصفوفة.

```
public ArrayList<SMSMessage> getNewMessages(String shortCodeText){

 success = false;
 try
 {
  ArrayList<SMSMessage> messages;
  PreparedStatement statement = connection.prepareStatement(
     "select * from inbox\n" +
     "inner join shortcodes on shortcodes.ShortCodeID = inbox.ShortCodeID\n" +
     "inner join smppconnections on smppconnections.CONNECTIONID = “ +
  “inbox.connectionid\n" +
  "where Lower(shortCode) = ? \n" +
  "and isProcessed = 0");
  statement.setString(1, shortCodeText.toLowerCase());
  ResultSet result = statement.executeQuery();
  messages = new ArrayList<>();
  while (result.next()){

   SMSMessage message = new SMSMessage();
   message.connectionID = result.getInt("connectionID");
   message.operatorID = result.getInt("operatorID");
   message.fromMDN = result.getString("fromMDN");
   message.shortMessage = result.getString("ShortMsg");
   message.shortCodeStr = result.getString("ShortCodeStr");
   message.isProcessed = result.getInt("isProcessed");
   message.messageID = result.getInt("id");
   message.shortCodeID = result.getInt("shortCodeID");
   message.msgTime; = result.getTimestamp("msgTime");
   messages.add(message);
```

```
  }


  success = true;
  return(messages);


 }
 catch (SQLException ex) {
  lastError = "Error in getNewMessages: " + ex.toString();
  General.writeEvent(lastError, "");
  return(null);
 }
}
```

وهو يعتبر مثال بسيط، لكن توجد في الحياة العملية إجراءات أكثر تعقيداً صعبة الفهم إذا لم يتم تطبيق مفهوم التجريد فيها.

سوف نقوم بعمل إعادة صياغة للكود وذلك بقسيمه إلى جزأين، حيث نقوم باستخلاص الجزء الخاص بطلب المعلومات وقراءة السجلات ووضعها في مصفوفة في إجراء منفصل، ثم نقوم بنداء الإجراء الجديد من داخل الإجراء القديم.

قمنا باستخلاص جزء القراءة من قاعدة البيانات واسميناه readInbox وهو اسم مجرد يعني قراءة أي نوع من رسائل الواردة، حيث يمكننا قراءة رسائل جديدة بها أو رسائل قديمة، بخلاف الإجراء القديم والذي لا يمكن استخدامه إلا لقراءة الرسائل الجديدة، فأصبح الكود كالتالي:

```
public ArrayList<SMSMessage> getNewMessages(String shortCodeText){

  success = false;
  try
  {
    ArrayList<SMSMessage> messages;
    PreparedStatement statement = connection.prepareStatement(
     "select * from inbox\n" +
     "inner join shortcodes on shortcodes.ShortCodeID = inbox.ShortCodeID\n" +
     "inner join smppconnections on smppconnections.CONNECTIONID = " +
     "inbox.connectionid\n" +
     "where Lower(shortCode) = ? \n" +
     "and isProcessed = 0");
    statement.setString(1, shortCodeText.toLowerCase());
    messages = readInbox(statement);
```

```
    success = true;
    return(messages);

  }
  catch (SQLException ex) {
    lastError = "Error in getNewMessages: " + ex.toString();
    General.writeEvent(lastError, "");
    return(null);
  }

}
```

وهذا كود الإجراء الجديد readInbox:

```
private ArrayList<SMSMessage> readInbox(PreparedStatement statement) throws
SQLException {
  ArrayList<SMSMessage> messages;
  ResultSet result = statement.executeQuery();
  messages = new ArrayList<>();
  while (result.next()){

    SMSMessage message = new SMSMessage();
    message.connectionID = result.getInt("connectionID");
    message.operatorID = result.getInt("operatorID");
    message.fromMDN = result.getString("fromMDN");
    message.shortMessage = result.getString("ShortMsg");
    message.shortCodeStr = result.getString("ShortCodeStr");
    message.isProcessed = result.getInt("isProcessed");
    message.messageID = result.getInt("id");
    message.shortCodeID = result.getInt("shortCodeID");
    message.msgTime = result.getTimestamp("msgTime");
    messages.add(message);
  }
  return messages;
}
```

نلاحظ أن الإجراء الأول (getNewMessages) أصبح أصغر وأبسط، و سهل القراءة . كذلك فإن الإجراء الجديد (readInbox) أيضاً سهل القراءة والفهم ويمكن إعادة استخدامه مع إجراءات أخرى، مثلاً لقراءة الرسائل القديمة، لا نحتاج لتكرار جزئية القراءة. وبذلك نكون قد سهلنا الصيانة، مثلاً لو قمنا بإضافة حقل جديد في الجدول Inbox ما علينا إلا إضافة سطر واحد لقراءته في إجراء القراءة، لكن إذا كان هُناك تكرار للكود ولم يكن هُناك تجريد، فإن أي تعديل أو إضافة لحقل، يتبعها تغييرات كثيرة في أي مكان من الكود يوجد فيه تكرار قراءة هذه الحقول والسجلات، وإذا نسينا تعديل جزء من الأجزاء المكررة ربما تحدث مشكلة في البرنامج. أهداف هندسة البرمجيات التي يحققها التجريد في هذا المثال هي:

- إمكانية إعادة استخدام الكود
- سهولة الصيانة لأي جزء من الكود

- سهولة القراءة والفهم ومن ثم سهولة إيجاد المشكلة
- سهولة التعديل والاختبار
- سهولة فهم هذا الجزء من قبل باقي الفريق البرمجي

2. مبدأ إعادة الاستخدام code re-usability

إعادة استخدام الكود- في هندسة البرمجيات- هي من المباديء الصحيحة للبرمجة، وهي تقلل من زمن تطوير البرامج وبالتالي تقليل التكلفة، وتساعد في تسهيل وتبسيط البرامج المعقدة، ومن ثم سهولة الاستمرار في تطويرها وزيادة إمكاناتها. وقد تكلمنا في الفقرة سابقة عن التجريد باعتباره أحد الوسائل المستخدمة لتسهيل أو تحقيق إعادة استخدام الكود.
تعريف إعادة استخدام الكود هو كتابة إجراء أو مجموعة إجراءات أو وحدات يمكن الاستفادة منها في أكثر من موضع. وسوف نتكلم عنها في هذه الفقرة بإذن الله من ناحية عملية.
توجد عدة طرق ومستويات لإعادة استخدام الكود، و سوف نقوم بتقسيم هذه المستويات حسب مدى استخدامها، وقد قمنا بتقسيمها إلى ثلاث مستويات: المستوى الأول هو الأعلى قابلية للاستخدام والمستوى الثالث هو الأقل. لكن دعونا نبدأ بالمستوى الرابع:

المستوى الرابع: النسخ واللصق: وهو أن يكون لديك برنامج أو وحدة أو إجراء أو مجموعة إجراءات في نظام ما، مثلاً نظام مخازن، ونحن نريد إنشاء برنامج جديد مختلف عن نظام المخازن لكن يشترك معه في بعض الأمور، فإذا كان شديد الشبه به يمكننا نسخ نسخة من نظام المخازن ثم تحويلها إلى النظام الآخر. الشكل الأخر هو وحدة في نظام ما نقوم بنسخها إلى نظام آخر ثم نقوم بتعديلها لتتناسب من النظام الآخر، كذلك بالنسبة للإجراءات. بهذه الطريقة نكون قد كسبنا الوقت، فبدلاً من البداية من الصفر نكون قد بدأنا بقاعدة معتبرة من الكود. مع أننا بهذه الطريقة اعدنا استخدام كود مكتوب سابقاً إلا أننا قمنا بتكرار الكود، فاصبح نفس الكود موجود في مشروعين برمجيين في معزل عن بعضهما، والأسواء هو أن يتم تطوير وتعديل كل جزء من الكود بطريقة مختلفة تُناسب المشروع الذي تعمل فيه، فتصبح التعديلات في النظام الأول لا نستفيد منها في النظام الثاني مع أن مصدرهما واحد، مثلاً وحدة أو مكتبة في المشروع الأول كانت هي النسخة المصدر لوحدة أو مكتبة في المشروع الثاني. فإذا اكتشفنا مشكلة مثلاً أو ثغرة أمنية في هذه الوحدة فلابد أن نقوم بمعالجتها في جميع المشروعات التي

قمنا بنسخ هذه الوحدة إليها، وبذلك تصبح هناك زيادة في الزمن وزيادة في التكلفة ويمكن أن ينتج عدم تطابق في أداء هذه الإجراءات، حيث أن التعديل في كل نسخة بطريقة مختلفة يجعل الإجراءات لها صفات مختلفة وبالتالي علاج مشكلة يمكن أن يختلف من إجراء إلى إجراء آخر حتى لو كان لها نفس الاسم، وتحتاج لعمل تجربة مختلفة للكود في كل مشروع. إذاً فإن طريقة النسخ واللصق هذه لا تعتبر ضمن وسائل إعادة استخدام الكود.

المستوى الثالث: استخدام الإجراءات: ونقصد به القيام بزيادة تجريد عدد من الإجراءات لمستوى يمكننا معه استخدامها في أكثر من موضع داخل البرنامج الواحد، فإذا كان البرنامج الناتج هو ملف تنفيذي تكون هذه الإجراءات داخل هذا الملف التنفيذي. هذه الطريقة تحقق إعادة الاستفادة من نفس الإجراء أو الوحدة داخل برنامج واحد، وتسهل صيانته حيث تمنع إعادة تكرار الكود، والتطوير لخصائص هذه الإجراءات يكون في نقطة واحدة فقط. يُستخدم هذا النوع مع الإجراءات الخاصة ببرنامج أو مشروع معين، أي أنها اكثر خصوصية بمشروع واحد وأقل عمومية ، حيث لا يمكن أن تستفيد منها باقي البرامج. مشكلة هذا النوع هو محدوديته وارتباطه بمشروع واحد، فإعادة الاستخدام طبقناها على نطاق ضيق، هو نطاق البرنامج الواحد. لكن يظل هو طريقة صحيحة يحقق مبدأ إعادة الاستخدام وننصح بتطبيقه.

المستوى الثاني: الوحدات والمكتبات: وهي كتابة وحدة أو مكتبة كاملة بطريقة مجردة ليس فيها أي ارتباط بوحدات خاصة في مشروع معين ، فيتم كتابتها بصورة عامة ليتم استخدامها في أي مشروع، ويوجد نوعان: النوع الأول هي الوحدات المصدرية الخاصة بلغة برمجة معينة، مثل header files في لغة سي، و units في لغة باسكال، و class library في لغة جافا: في هذه الحال يحتاج المبرمج المستفيد من هذه الوحدة لمصدر هذا الكود ليقوم بتضمينه داخل البرنامج المستفيد، وفي حالة إنتاج برامج طبيعية Native يتم تضمين هذه الوحدات داخل الملف التنفيذي. النوع الثاني: وهو استخدام مكتبة، حيث تتم ترجمتها إلى مكتبة بصورة ثنائية حسب نظام التشغيل ويتم استدعائها بصورة خارجية من أي برنامج، ويتم الربط في وقت التنفيذ ولا يتم تضمينها ضمن الملف التنفيذي للبرنامج المستفيد. هذا المستوى يحقق إعادة استخدام الكود بأفضل الطرق، حيث يمكن الاستمرار في تطور تلك المكتبات بصورة منفصلة، أحياناً يكون المطور لهذه المكتبات طرف ثالث -كما يُسمى- هدفه هو فقط تطوير هذه المكتبة، وكمثال لها: مكتبات التشفير، والربط مع قواع البيانات، أو الربط مع البريد. أما مبرمجو التطبيقات فيقومون بتحميل هذه المكتبة سواءً في شكل ملف مصدري أو ثنائي لاستخدامها

في برامجهم المختلف. وهي تُمثل أعلى درجات التجريد وذلك حسب نجاحها في الاستخدام في أغراض مختلفة.

يُمكن استخدام هذه الطريقة كذلك لتقسيم المشروع الواحد إلى عدة أقسام : قسم متخصص في إجراءات البيانات، وقسم آخر للربط مع الأنظمة الأخرى، وقسم متخصص في واجهة المستخدم، وبذلك يمكن إيكال مهمة تطوير كل جزئية لمبرمج مختلف بناءً على تخصصه والمجال الذي يتميز فيه. من عيوب هذا المستوى هو الحاجة لإعادة ترجمة كل البرنامج التي تستخدم تلك الوحدة في حال تغيير صفة ما في احد الإجراءات، أما إذا كان التغيير في صفة في مكتبة تم ربطها خارجياً فلا تتطلب إعادة الترجمة إلا إذا تم تغيير واجهة نداء الإجراء procedure interface. كذلك فإن هذه الطريقة بها ارتباط للغة البرمجة المستخدمة، مثلاً مكتبة للغة جافا لا يمكن استخدامها مع برامج سي أو برامج لغة Golang. لكن بالنسبة للغات الطبيعية مثل سي و باسكال فيمكنها أن تتشارك في مكتبات بعضها في حيث الأخذ بالاعتبار الطريقة القياسية لكتابة المكتبات وأنواع المتغيرات.

المستوى الأول: الإجراءات البعيدة remote procedures، مثل خدمات الويب web services: ومن أهم أشكالها خدمات ويب يمكن نداءها من أي برنامج مكتوب بأي لغة برمجة. و تُستخدم هذه الطريقة لربط الأنظمة مع بعضها البعض، أو لعزل جزئية من باقي الأجزاء. ويتم توفير الإجراءات بصورة حية online من أي برنامج مستفيد، كذلك يتم تعديل ويتم تطوير الإجراءات في هذه الخدمات دون الحاجة إلى إعادة ترجمة البرامج المستفيدة، و أحياناً لا نحتاج حتى لإغلاق تلك البرامج المستفيدة عند تحديث خدمات الويب، بل لا يتم إخبار مطور البرنامج المستفيد بهذه التغييرات في بعض الأحيان. وتُمسى هذه الطريقة بنداء الإجراءات البعيدة أو تُسمى أحياناً بالواجهة البرمجية API . تصبح خدمة الويب هذه مورد تتم صيانتها ودعمها من قبل جهة أخرى أو مبرمج أو فريق برمجي آخر مسؤوليته تطوير هذا الجزء المستقل من الكود. هذه الطريقة تمثل نقطة مركزية حية لكافة البرامج، فإذا تم أي تعديل لا نحتاج لإعادة ترجمة كما في المستوى الثاني. وهو يُمثل البرمجة متعددة الطبقات، ويحقق سرية وعزل كبيرين في تطوير البرامج - وهذا موضوع آخر ليس ضمن حديثنا في هذا الكتاب- إلا أنه أقل تجريداً من المستوى الثاني، حيث أن مطور خدمات الويب لا يكون طرف ثالث في معظم الأحيان، بل هو مطور النظام الذي نريد التخاطب معه، كذلك فأن خدمات الويب لا تُمثل إجراءات عامة، بل هي خاصة بمؤسسة معينة يتم تطوير خدمة الويب لها فقط، أو يمكن أن تكون خاصة بمشروع معين يتم تطويرها لصالح هذا النظام فقط، مثل أنظمة ERP. وبخلاف المكتبات والوحدات في المستوى الثاني فإن خدمات الويب تعتمد على مكتبات أو أجزاء كود خاصة وليست عامة. توجد أشكال أخرى

غير خدمات الويب تحقق نفس الأهداف لكنها أكثر قدما وأقل استخداماً ومن تقنياتها: CORBA, RMI, +COM
هُناك ميزة مهمة في هذا النوع من إعادة الاستخدام: وهو أنه لا يوجد ارتباط أو شرط للغة برمجة معينة لتستفيد من خدمات الويب مثلاً، حيث يمكن كتابة خدمة ويب باستخدام لغة جافا وندائها بلغة Golang أو أي لغة أخرى، وكذلك بالنسبة للمعماريات: حيث يمكن أن تكون خدمة ويب موجودة في منصة Solaries ويتم ندائها من منصة لينكس أو وندوز في معمارية AMD64.

من ناحية عملية مع أن المستوى الثاني والأول هما من أفضل المستويات إلا أنها ليست دائماً أفضل الخيارات، فكلما ارتفع المستوى كلما زادت التكلفة الأولية لتطوير البرامج، فلابد من اختيار المستوى المناسب مع التطبيق، فإذا كان التطبيق هو محرر نصوص بسيط فلا نتحاج لأن نقوم بتطويره في شكل خدمة ويب ليتم ندائه في نفس جهاز المستفيد، يكفي فقط أن نستخدم المستوى الثالث والثاني كهيكلية لهذا النوع من البرامج. وكلما زاد تعقيد وحجم المشروع كلما كانت هُناك الحاجة لاستخدام خدمات الويب لما لها من فوائد إعادة استخدام وفوائد أخرى مثل السرية وعزل البرامج من بعضها. المستوى الأول مع أنه أكثر تكلفة في تطويره الأولي، إلا أنه يحقق أفضل النتائج لإعادة الاستخدام وصيانة البرامج وتطويرها في المستقبل. ومن ناحية عملية يتم استخدام كافة هذه الأنواع في مشروع واحد، بل حتى المستوى الرابع يمكن استخدامه أثناء تطوير خدمات الويب.

يُستخدم المستوى الأول والثاني في واجهة البرمجة API والتي عن طريقها يمكننا ربط نظامين مع بعضهما. في السابق كان يُستخدم المستوى الثاني (استخدام المكتبات) لغرض ربط برامج مع بعضها، لكن الآن اصبح السائد هو المستوى الأول، حيث أصبحت واجهة ربط البرامج مع بعضها هو باستخدام خدمات الويب لما توفره من سرية وعزل كبير لبيئة تشغيل الأنظمة المراد ربطها ببعض.

3. مبدأ البساطة simplicity

مبدأ البساطة لا يقتصر على هندسة البرمجيات فقط، إنما يشمل كل مجالات الهندسة عموماً، بكل كل مجالات الحياة. وهو من أهم مقومات المبادئ الصحيحة للبرمجة. لذلك دعونا نتكلم عن البساطة كمفهوم عام أولاً ثم نخصصها لجانب هندسة البرمجيات.

بساطة الأدوات، والتصميم، وبساطة الآلات، بل وحتى بساطة العبارات، والخُطب، والمقالات، والنظريات، وغيرها من الأشياء الملموسة وغير الملموسة في حياتنا هي من المفاهيم التي تهدف وتحقق الآتي:

1. اختيار الآلية الأبسط لتحقيق هدف ما
2. الأكثر بساطة هو الأكثر اعتماداً
3. البساطة تقلل التكلفة الابتدائية وتكلفة التعديل والتطوير وتكلفة التشغيل
4. البساطة تسّهل فهم الآخرين للأمر المراد تنفيذه ومشاركتهم في إنجازه

كانت هذه أمثلة وليست كل الأهداف والفوائد. فاختيار الطريق والآلية الأبسط لتحقيق مشروع ما أو تصليح عطل أو إنشاء بناء أو حتى كتابة كتاب فهذا لا يقلل التكلفة فقط، بل يضمن تحقيق ذلك الإنجاز. أحياناً يكون الاختيار غير الموفق للأداة – مع أنها أحياناً لا تكون جزء مهم من المشروع- يمكن أن يساهم تعقيدها وعدم إتقانها الفشل المبكر للمشروع. فإذا كان الهدف ربط مسمار برغي واحد أو عدد محدود من البراغي فأفضل طريقة هو اختيار مفك مناسب وبسيط، فإذا اخترنا مثلاً مفك براغي كهربائي نكون قد زدنا التكلفة وزدنا التعقيد، حيث يمكن أن لا تتوفر كهرباء في المكان الذي نريد ربط البرغي فيه، فنفكر في اختيار مفك براغي كهربائي لديه بطارية أو شراء وصلة طويلة توصلنا إلى المكان الذي نريد ربط البرغي فيه، بذلك نكون قد بذلنا جهداً كبيراً في حل مشكلة الأداة (وهي مفك البراغي الكهربائي) بدلاً من الحل المباشر للمشكلة.

كذلك فإنه كلما زادت البساطة كلما زادت الاعتمادية، حيث أن الاعتمادية تتطلب توفر الحل في جميع الظروف، مثلاً ظرف انقطاع الكهرباء، و ظرف بعد المكان عن مصدر الطاقة، وظرف العمل لوقت طويل دون توقف، وغيرها. فكلما كانت الأداة والطريقة والآلية بسيطة لتحقيق حل مشكلة ما كلما قل احتمال تعطلها، وبالتالي زادت فرصة توفرها دائماً للعمل.

تقليل التكلفة هي شيء مهم جداً والمقصود بها التكلفة الزمنية والتكلفة المادية، حيث أنه في ظل الانفتاح وتحول العالم إلى قرية، إذا لم تقدم حلاً ذو تكلفة أقل، سوف يأتي غيرك يقدم نفس الحل

بتكلفة أقل. لذلك فإن الهدف هُنا هو حل المشكلة بطريقة أكثر كفاءة على مستوى الحل الأولي وعلى مستوى الاستمرار في تشغيل وتفعيل تلك الآلية أو المشروع.
البساطة تساهم في الفهم السريع لأعضاء الفريق لآلية الحل وبالتالي تسمح لهم بالمساهمة الفعلية و الاستمرار و الانتشار بهذا المشروع بدلاً من أن يكون لغزاً محتكراً لصاحب الفكرة أو من قام بتنفيذه فقط.
بالنسبة لتطوير البرامج فإنها أصبحت أكثر تعقيداً من الماضي، فابتداءً بأنظمة التشغيل إلى لغات البرمجة وحتى المتصفحات، ناهيك عن عدم ذكر العتاد و ما وصل إليه من تطور. كلها أصبح فيها تعقيد، مثلاً المتصفحات أصبح بها مترجمات للغات برمجة مثل جافا سكريبت. و الاتجاه الآن هو للبساطة، بأن يكون لدينا لغات أبسط، وأنظمة أبسط ومحركات قواعد بيانات أبسط. نعني أبسط من ناحية التصميم الداخلي وحتى من حيث الإمكانات. فيمكن لأدوات ذات إمكانات أقل أن تنافس الأدوات المعقدة. وهناك مفهوم أو نظرية في هندسة البرمجيات قرأتها مؤخراً أعجبتني كثيراً تقول worse is better، أي الأسواء هو الأفضل، ويُعنى بالأسواء هو الأقل ميزات. فإذا كُنت تريد كتابة نص بسيط فإن الأفضل هو الأدوات الأبسط مثلاً nano في نظام لينكس أو Notepad بدلاً من كتابة ذلك النص باستخدام أدوات أعقد مثل حزمة office، نكتب بها سطر أو سطرين نصيين ثم نبحث كيف نقوم بتحويله إلى شكل مقروء لكل الأنظمة. أما إذا كتبناه بتلك الأدوات البسيطة فالناتج هو ملف نصي يمكن فراءته في كل أنظمة التشغيل وكل المعماريات ويمكن قراءته كذلك بجميع لغات البرمجة. تخيل أنك مبرمج وتم إرسال بيانات مرجعية لاستخدامها في برنامج ما، فأيهما تختار: أن تكون تلك البيانات في شكل ملف word أم excel تحتاج لمكتبة لفراءتها وقراءة نسقها المعقد ثم استنباط المعلومات الأساسية منه، أم تتمنى أن يكون ملفاً نصياً بسيطاً خالياً من أي نسق يستطيع عمل برنامج لفراءته طالب في بداية طريقة لدارسة لغة برمجة.
البساطة في تطوير البرامج يشمل اختيار الأداة الأبسط التي تستطيع إنجاز العمل وإلا أصبحت تلك الأداة عبئاً جديداً ابتداءً من تعلمها ومروراً بالبيئة التي تحتاج إليها للعمل بصورة أساسية و إنتهاءً بإيجاد المبرمجين المتوفرين للعمل عليها أو للوقت الذي يحتاج إليه المبرمج لدراستها وإتقانها. كذلك فإن البساطة تشمل تصميم النظام من حيث الأجزاء المختلفة وتقنيات الربط التي يحتاج إليها، وإلى البيئة التشغيلية والتطويرية التي نطلبها لتنفيذ هذا المشرع. كذلك حتى على مستوى الكود والإجراءات فكلما استخدمنا الإجراءات قياسية البسيطة فذلك أفضل بدلاً من استخدام مكتبات خارجية ربما يتوقف صاحبها عن دعمها في المستقبل.

في دورة حياة أي برنامج، يبدأ بسيطاً ثم يتعقد ليلبي كافة الاحتياجات للزبائن المختلفين، فإذا زاد تعقيده من ناحية تصميمه الداخلي يشيخ ذلك البرنامج ويُحال للتعاقد لأن التطوير فيه يزيد صعوبة وبالتالي تزيد ميزانية تطويره، فيكون الحل هو عمل بديل له بطريقة أبسط بتصميم نظيف clean design. فإذا بدأت بنظام بتصميم ومتطلبات معقدة فإنك بذلك تحيله إلى المعاش مبكراً. أما إذا بدأ بسيطاً وحافظ على هذه البساطة فهذا يزيد من عمره التشغيلي ويكون قليل المشاكل البرمجية ولديه فرصة أكبر ليعمل على نطاق أوسع من المستفيدين – على الأقل المستفيدين الذين يحبون البساطة-.
الأبسط هو الأفضل بالنسبة للمطور وبالنسبة للمستفيد. فالمطور يبذل جهداً و زمناً قليلين مقارنة بالأكثر تعقيداً. وبالنسبة للمستفيد الذي يبذل جهد في التعليم والتدريب للاستخدام الأمثل لهذا النظام أو البرنامج. فكما كان بسيطاً كلما كان الوقت اقل للتعلم والمتطلبات أقل من حيث الخبرة والكفاءة للمستخدمين النهائيين الذين سوف يعملون على هذا النظام. كذلك فإن البساطة تقلل فرصة الأخطاء التي يمكن أن يرتكبوها وتسهل علاج أي خطأ يحدث.
لم ذكر أمثلة برمجية في هذه الفقرة لأن الموضوع واسع يتسع لمجالات مختلفة ولا نريد تخصيصه لجانب ونترك جانب. فكما ذكرت سابقاً فإن البساطة تشمل التصميم الداخلي للنظام، وطريقة كتابة الكود، واختيار الأدوات للتطوير، وحتى التوثيق البسيط يحقق فائدة أكبر. تخيل مثلاً وثائق للنظام بعدد أوراق معدودة ووثائق أخرى تصل إلى مئات الصفحات، فأيهما نضمن أن المستخدم النهائي سوف يقرأه؟

4. مبدأ الوظيفة الواحدة single responsibility principle

الفكرة ببساطة هي أن يكون للوحدة من الكود هدف واحد من كتابتها، مثلاً أن تكون وحدة متخصصة في قراءة البيانات من قاعدة البيانات العلائقية، مثلاً نسميها access unit وأخرى متخصصة في واجهة المستخدم فنسميها presentation unit أو class أو حتى Layer حسب التقسيم المتبع وحجم الوحدة المتخصصة. بهذا نكون قد جعلنا لكل وحدة سبباً واحداً للتغيير، مثلاً إذا كانت المهمة تغيير في واجهة المستخدم لا يؤثر ذلك على طريقة قراءة المعلومات من قاعدة البيانات، وهذا يجعل الكود أسهل تعديلاً.

نأخذ مثال لإجراء به أكثر من وظيفة ثم نقوم بتصحيحه. والمثال مكتوب بلغة برمجة افتراضية، مهمته قراءة بيانات من جدول في قاعدة بيانات ثم عرضها في المتصفح:

```
function ReadAndDisplayTable
{
  connection = Connection("localhost", "MyData", "user", "password")

  dataset = connection.query("select * from students")

  html.print("<table><tr><th>ID</th><th>Name</th></tr>")
  for record in dataset
  {
    html.print("<tr>")
    html.print("<td>", record["id"], "</td>")

    html.print("<td>", record["name"], "</td>")
    html.print("</tr></table>")
  }
  connection.close

}
```

نلاحظ أن الإجراء به قراءة من قاعدة البيانات ثم إظهارها للمتصفح، فإذا أردنا تغيير شكل المخرجات في الشاشة فإننا نقوم بتغيير هذا الإجراء ، وإذا أردنا تغيير مخدم قاعدة البيانات فنقوم بتغيير نفس الإجراء، بهذا كان هُناك سببان مختلفان للتغيير، تغيير بسبب واجهة مستخدم وآخر بسبب إعدادات قاعدة بيانات، وكلاهما موضوعان مختلفان لا يمكن جمعهما في إجراء واحد، والتغيير ربما يحدث عنه خطأ غير مقصود في الجزء الآخر الذي لا يعنينا من الكود.

التصحيح هو أن نقوم بفصلهما في إجراءين منفصلين، والأفضل أن تكون وحدات مختلفة، مثلاً كل واحد في مكتبة منفصلة أو class منفصلة تجمع نوع متشابه من الوظائف:

```
function ReadTable: Dataset
{
  connection = Connection("localhost", "MyData", "user", "password")

  dataset = connection.query("select * from students")
  connection.close
  retrn dataset
}


function displayTable(dataset: Dataset)
{

  html.print("<table><tr><th>ID</th><th>Name</th></tr>")

  for record in dataset
  {
    html.print("<tr>")
    html.print("<td>",record["id"],"</td>")
    html.print("<td>",record["name"],"</td>")
    html.print("</tr></table>")
  }
}
```

ثم نقوم بندائهما عند الحاجة بهذه الطريقة:

```
data = Readtable()
displayTable(data)
```

بهذه الطريقة يكون لدينا إجراء لقراءة البيانات فقط (readTable) وآخر لكتابته (displayTable) ونحقق أيضاً إعادة استخدام الكود، حيث يمكن استخدام الإجراء readTable في أي مكان آخر ليس له علاقة بإظهار البيانات في المتصفح، مثلاً لقر ائتها ثم إرسالها بالبريد، كذلك يمكن نداء إجراء الكتابة في المتصفح بعد قراءة البيانات من ملف في القرص الصلب مثلاً.

5. مبدأ إخفاء التفاصيل hide implementation details

المقصود بمبدأ إخفاء تفاصيل التطبيق implementation وهو إخفاء الجزء الذي به المكونات الفعلية لجزئية أو نظام ما، مثل قاعدة البيانات أو الموارد المختلفة مثل الملفات و الاتصالات وحتى الإجراءات و الدوال التي يتم نداءها داخلياً هي من التفاصيل التي يجب حمايتها. هذه هي مكونات وأجزاء هشة تغييرها أو الوصول لها يؤثر مباشرة في سلوك النظام أو جزء منه مثل الإجراءات و الدوال أو الفئات والوحدات، لذلك يجب حمايتها من الوصول الخارجي لها ومن البيئة المحيطة بها.

الهدف من إخفاء التفاصيل هو تقليل الارتباط بين البرامج المعتمدة على جزء ما من الكود، مثلاً وحدة، أو مكتبة أو فئة، و التقليل يكون بأن لا نسمح لتلك الأجزاء المستخدمة لهذا الجزء من الكود للوصول إلى التفاصيل مثل المتغيرات و بعض الإجراءات، فإذا حدث تغيير لتلك المتغيرات أو الأجزاء الداخلية لتلك الإجراءات عندها لا بد من تغيير كل أجزاء الكود المستخدمة. لكن إذا منعناها من الوصول إلى تلك البيانات والإجراءات وسمحنا لها الوصول عبر نقطة واحدة أو عدد بسيط من النقاط مثل استخدام دالة واحدة نسميها الواجهة البرمجية interface والتي تُعتبر المدخل الصحيح لاستخدام تلك الوحدة، أو المكتبة أو الفئة، فكلما كان عدد الـ interfaces أقل وذات مُدخلات (باراميترات) أقل، كلما كانت تلك الوحدة البرمجية أكثر استقلالية في تغييراتها الداخلية، وكذلك فإن الأجزاء الخارجية التي تستخدم تلك الوحدة تكون أقل تعديلاً إذا تم تعديل في تلك الوحدة البرمجية.

المثال التالي لوحدة unit في لغة برمجة هيكلية (لغة باسكال) لقراءة معلومات زبون من قاعدة البيانات، تفاصيل الوحدة تحتوي على الاتصال مع قاعدة البيانات وتفاصيل لغة SQL للبحث عن معلومات الزبون بالإضافة إلى متغيرات تُستخدم لتنفيذ الربط والـ query

```
unit Database;

interface

 function getCustomerInfo(customerID: string): TInfo;


implementation

var
    connection: TConnection;
    SQL: TSQLQuery;

function connectToDB();
begin

    // code for connecting to database
```

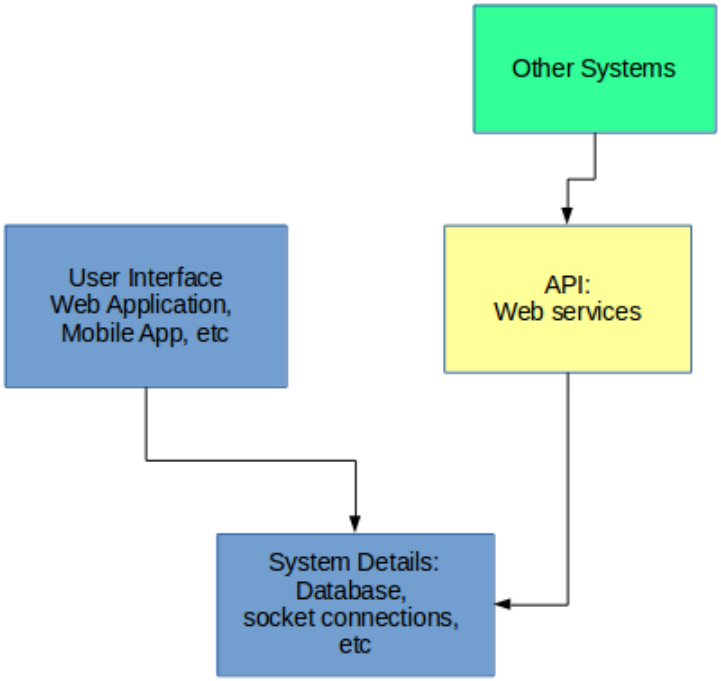
```
    connection := ....

end;


function readData(customerID: string): RecordSet;
begin
    connectoDB;
    // Query SQL
    result:= sql.Record;
end;


function getCustomerInfo(customerID: string): TInfo;
begin
   result:= readData(customerID);
end;
```

نلاحظ أننا قمنا بإظهار دالة واحدة فقط في الواجهة interface وهي *getCustomerInfo* أما باقي الدوال الخاصة مثل *connectToDB*, *readData* والمتغيرات كلها أخفيناها بحيث لا يستطيع من يستخدم هذه الوحدة الوصول لهذه الموارد مباشرة، فقط يمكن الوصول للدالة *getCustomerInfo* بهذه الطريقة فإننا إذا قمنا بتغيير جدول قاعدة البيانات الذي نقرأ منه معلومات الزبون أو حتى لو غيرنا محرك قاعدة البيانات مثلاً فإن ذلك لا يؤثر على من يستخدم هذه الوحدة ما لم نقم بتغيير الباراميترات وهي في هذه الحالة (*customerID*) أو اسم الدالة، ما عدا ذلك فإن أي تغيير في باقي التفاصيل المحمية بعد كلمة *implementation* فإن ذلك لا يغير في باقي الأجزاء من الكود المستفيدة من هذه الوحدة، بهذا نكون قد عزلنا المستفيدين من تفاصيل تلك الوحدة والطريقة التي تعمل بها.

نفس المفهوم يمكن تطبيقه باستخدام البرمجة الكائنية بواسطة الكبسلة *encapsulation*.

المثال السابق كان يتكلم عن وحدة صغيرة برمجية مثل الوحدات والفئات والمكتبات، لكن نفس المبدأ يمكن تنفيذه (لكن بطريق مختلف) مع الوحدات الأكبر مثل أجزاء أكبر تجمع عدة وحدات أو حتى أنظمة كاملة.

إذا تكلمنا عن أنظمة كاملة فإن إخفاء تفاصيلها له أهمية أكبر، فلا يمكن أن نسمح لنظام آخر أن يقوم بالوصول مباشرة لقاعدة بيانات نظامنا الحالي وإلا أصبحنا لا نستطيع تغيير هيكل البيانات – مثلاً - أو حتى تغيير المخدم دون أن يتأثر النظام الآخر، ناهيك عن مشاكل السرية أو مشاكل الأداء التي قد يتسبب بها النظام الآخر. الحل الأفضل هو عمل واجهات برمجية API في شكل خدمات ويب مثلاً web services تعمل كأنها إجراءات تعمل على النظام الحالي يتم ندائها من أنظمة خارجية بالطريقة في الشكل أدناه :



نلاحظ أننا سمحنا للأنظمة الأخرى الاتصال بنظامنا الذي قمنا بتطويره عن طريقة الواجهة البرمجة فقط API فإذا قمنا بعمل أي تغيير في أجزاء النظام الداخلية فإن ذلك لا يؤثر على طريقة الربط وبالتالي لا يؤثر على الأنظمة الأخرى، كذلك فمن ناحية السرية فإن الأنظمة الأخرى لا تستطيع الوصول إلى أي تفاصيل سوى ما هو مسموح لها عن طريق الـ API المحدودة والتي بدورها لا تتيح الوصول إلى كل النظام.

6. مبدأ الفتح والإغلاق open closed principle

احد مباديء البرمجة بطريقة صحيحة هو مبدأ الفتح والإغلاق. وهو يعني سماحية توسيع إمكانات الجزئية البرمجية والإضافة لها واستخدامها بطريقة مختلفة دون التعديل فيها.

من تعريفه يبدو أن هُناك تناقض: حيث تكون هُناك إمكانية للإضافة لكن دون التعديل. والمقصود هو أن تكون لدينا جزئية برمجية أو مكتبة أو فئة كائن لديها صفات معينة، فنقوم بإعادة استخدامها مع إضافات جديدة لها و تغيير في صفاتها الموجودة فيها دون أن نكون قد فتحنا مصدرها وقمنا بتغييرها. بذلك كل من يستخدم تلك المكتبة أو الكائن لا يحدث له تغيير في السلوك وذلك لأننا لم نمس تلك المكتبة أو فئة الكائن، تركناها كما هي لكن بطريقة ما اعدنا استخدامها بطريقة سمحت لنا إضافة ميزات جديدة لها مع استيعاب التغييرات التي نريد أن نقوم بتغييرها فيها أثناء تشغيل البرنامج.

الهدف من هذه الطريقة هي إعادة الاستخدام، لكن دون المساس بالكود الأساسي كما ذكرنا. وعدم المساس بالكود الأساسي يحقق لنا استقرار البرنامج بحيث لا تتأثر الأجزاء التي تستخدم هذه المكتبة أو البرامج الأخرى التي تستخدم نفس المكتبة، ويحقق لنا توسعة قاعدة الاستخدام للكود بين البرامج.

كذلك الهدف هو إمكانية التعديل والإضافة بطريقة سهلة لاستيعاب المطلوبات الجديدة.

ما يحقق هذا المبدأ هي عدة ميزات لابد أن تتوفر في لغة البرمجة التي نستخدمها وهي: البرمجة الكائنية OOP، والوراثة inheritance، و وتعدد الأشكال polymorphism. كذلك فإننا نستخدم التجريد لتحقيق هذا الهدف. ونلاحظ أننا أول مرة يتطلب منا المبدأ أن تكون لغة البرمجة تدعم البرمجة الكائنية. حيث أن المبادئ التي ذُكرت سابقاً في الفقرات السابقة يمكن تحقيقها في لغات هيكلية structured لكن هذه المرة البرمجة الهيكلية لا تسمح لنا بتحقيق مبدأ الفتح والإغلاق.

تعدد الأشكال Polymorphism مرتبط بالوراثة والتجريد، ومع أنه ليس الموضوع الأساسي في هذه الفقرة، لكن سوف نقوم بالتحدث عنه باختصار.

تعدد الأشكال ببساطة هو أن تكون هُناك فئة كائن بها نوع من التجريد أي تصلح أن تكون عامة، مثلاً فئة كائن اسمها UserAuthenticationClass الغرض منها التأكد من المستخدم وكلمة المرور، وبها هذه الإجراءات:

```
CheckUser(String username, String password)

ChangeUserPassword(String username, String oldPassword, String newPassword)

InsertNewUser(String username, String password)
```

نحن نريد استخدام هذه الفئة مع أي قاعدة بيانات لكن تختلف كل قاعدة بيانات حسب تصميمها، مثلاً أحد المبرمجين يسمي الجدول المحتوي على أسماء المستخدمين users وآخر يسميه logins وثالث يُسميه accounts. كذلك فإن الحقول التي تحتويها هذه الجداول تختلف أسمائها، كذلك فإن كلمة المرور تختلف طريقة تخزينها، فمنهم من يستخرج منها ما يُسمى بالـ hash مثل MD5 ومنهم من يضع كلمة المرور واضحة كما هي في قاعدة البيانات. لذلك يكون من الصعب أن يكون لدينا نفس الكود أو نفس المكتبة التي يمكن استخدامها مع جميع هذه الجداول المختلفة التصميم.
لحل هذه المشكلة أولاً لابد من تجريد الإجراءات الداخلية والتي تتعامل مع الجداول مباشرة، مثلاً لابد من استخلاص هذا الجزء من الكود في إجراء منفصل:

```
query.Execute("select username, password from users where username = ?");

query.setParameter(1) = username;
```

مثلاً نُسميه getAuthenticationFromTable لكن المشكلة إذا كتبنا فيه أي كود فإننا نكون قد ربطناه بتصميم معين، وهو أن يكون اسم الجدول users وحقل اسم المستخدم username وكلمة المرور password فإذا اختلف أي من هذه التفاصيل فإن هذا الكود سوف لن يعمل.
نقوم ببساطة عدم كتابة متن هذا الإجراء في الفئة، فقط نكتب اسمه والمدخلات والمخرجات أي نكتب رأس الإجراء header/interface ونحوله إلى إجراء افتراضي virtual أي لا يوجد له كود في هذه المكتبة أو الفئة أو الوحدة، لكن على من يستخدم هذه المكتبة أن يقوم بكتابة كود هذا الإجراء بنفسه، فمثلاً هذا تعريف لهذا الإجراء في فئة الكائن الأساسي باستخدام لغة جافا:

```
public abstract ResultSet getAuthenticationFromTable(String username);
```

نلاحظ أنه لا يوجد كود بداخله، لكن مع ذلك يمكن نداءه من باقي إجراءات الفئة، مثلاً نقوم بمناداته داخل الإجراء checkUser:

```
public boolean checkUser(String username, String password) throws SQLException{
  ResultSet result = getAuthenticationFromTable(username);
  if (result.next()){
    if (password.equals(result.getString(2))){
      return true;
    }
    else {
      return false;
    }
  }
```

```
  else {
    return false;
  }
}
```

في هذه الحال لا يمكن استخدام هذه الفئة مباشرة، إنما لابد من وراثتها ثم كتابة الإجراءات الافتراضية فيها، مثلاً في برنامج للمبيعات نقوم بوراثتها في فئة جديدة نسميها *PurchaseAuthentication* وهذا هو الكود الذي تحتويه:

```
public class PurchaseAuthentication extends UserAuthenticationClass{

  @Override
  protected ResultSet getAuthenticationFromTable(String username){

    // Do Database connection..
    PreparedStatement statement = connection.prepareStatement(
      "select username, password from users where username = ?");
    statement.setString(1, username);
    return statement.executeQuery();

  }
```

عبارة *@Override* تعني أن هذا الإجراء سوف يقوم باستبدال الإجراء الموروث، فكأنها وراثة عكسية، فبدلاً من أن ترث الفئة الابن الفئة الأب، فإن الأب في هذه الحالة يرث الابن ويستخدم إجراءه *getAuthenticationFromTable*

نلاحظ كذلك أننا فقط كتبنا إجراء واحد لكن في المقابل اعدنا استخدام كل الإجراءات الموجودة في الفئة الأصلية *UserAuthenticationClass*. عند استخدام الفئة نقوم بتعريفها ومناداتها كما في المثال التالي:

```
  UserAuthenticationClass auth = new PurchaseAuthentication();
  boolean success = auth.checkUser("mohammed", "mypassword");
```

نلاحظ أننا عرفنا الكائن على أنه من الفئة *UserAuthenticationClass* لكن عندما قمنا بعمل تهيئة له instantiation هيأناه على أنه من نوع *PurchaseAuthentication* حتى نتمكن من نداء أي إجراء من الفئة *PurchaseAuthentication*. أو من الفئة *UserAuthenticationClass*.

ويمكننا أن نقول أن الفئة *PurchaseAuthentication* هي فئة ممتدة من الفئة *UserAuthenticationClass* و الامتداد extension حققه لنا الوراثة وتعدد الأشكال بأن سمحت لنا إضافة كود جديد لكن دون أن نقوم بتغيير الكود الأساسي للفئة الرئيسة.

ويمكننا استخدام الفئة *UserAuthenticationClass* مع أي برنامج مهما كان شكل قاعدة بياناتها وجداولها.

يمكننا كذلك تجريد الاتصال مع قاعدة البيانات ونجعلها افتراضية للسماح بإمكانية الاتصال مع أي نوع من محركات قواعد البيانات مثلا أوراكل، MySQL أو FireBird حيث أن لكل واحدة طريقة مختلفة أو مكتبة مختلفة للاتصال.

بهذه الطريقة نكون قد وسعنا إعادة استخدام الكود بالنسبة للفئات المجردة، أو نقوم بتجريد الفئات ثم نستخدمها في عدد من البرامج، أي يمكننا كتابة مكتبة بها كل هذه الفئات المجردة ثم نقوم باستخدام هذه المكتبة في باقي البرامج بدلاً من إعادة كتابة نفس الكود عدة مرات مع كل برنامج جديد مع اختلاف بسيط من برنامج لآخر. ويمكن للمبرمجين الأكثر خبرة أن يقومون بكتابة هذه المكتبات لتقليل الأخطاء التي يمكن أن يقع فيها المبرمجين المبتدئون.

7. مبدأ التماسك Cohesion

التماسك مقصود به علاقة الإجراءات والبيانات في فئة أو وحدة واحدة. فكلما كانت الإجراءات لها علاقة وطيدة كلما كان ذلك جيداً وزاد من التماسك، أما إذا لم يكن هُناك علاقة مع بعضها أو ذات علاقة ضعيفة يُسمى ذلك قلة تماسك. من فوائد التماسك هو سهولة فهم الوحدة وبالتالي صيانتها وتطويرها، ويحقق إعادة الاستخدام، وسهولة التجربة، وعند التغيير فيها لا يؤثر ذلك على باقي الوحدات.
توجد عدة أنواع أو مستويات من التماسك نبدأها بأقلها تماسكاً (أسوائها) إلى أكثرها تماسكاً (أفضلها):

أ- تماسك بالصدفة Coincidental Cohesion :
وهو وجود إجراءات ودوال في وحدة أو فئة واحدة لا علاقة لها مع بعضها، مثلاً:

```
class XYZ {

  function getMD5(string text) string {

     // calculate, and return MD5
  }

  function searchFile(string afilename) bool {

      // search file, and return true/false
  }

  function downloadPage(string aurl ) string {

    // get URL
  }

}
```

نلاحظ في المثال السابق وجود ثلاث دوال أو إجراءات لا علاقة لها ببعضها، وهذا يتعارض أيضاً مع مبدأ الوظيفة الواحدة، حيث أن للفئة XYZ وظائف متعددة. فيحدث أن كل إجراء له حاجاته المختلفة وإجراءاته الثانوية المختلفة ويمكن أن لا يكون هُناك إجراءات مشتركة مما يزيد حجم الكود في هذه الوحدة البرمجية فيصعب صيانتها وفهمها. كذلك فإن من يريد إعادة استخدام هذه الوحدة ربما يحتاج فقط لإجراء واحد منها، لكن باقي الإجراءات ربما تتطلب مكتبات تمنع ترجمة البرنامج إلا بوجودها، مثلاً نفرض أن الدالة downloadPage تحتاج لمكتبة HTML والمبرمج يحتاج لهذه الفئة فقط لاستخدام

الدالة searchFile فبالتالي لا يستطيع استخدامها إلا أن يقوم بتوفير مكتبة HTML التي تتطلبها الدالة downloadPage مع أنه لا يحتاج إلى هذه الدالة.

ب- التماسك المنطقي Logical cohesion

ومقصود به اشتراك مجموعة من الدوال - مع اختلاف طبيعتها – تحت مسمى منطقي واحد، مثلاً فئة لتثبيت البرنامج في المخدم تحتوي على الدوال التالية:

```
class Installation {

  function writeConfig(string configData) bool {

      // write configuration file
  }

  function createDatabase(string schema) bool {

    // Create database schema
  }

  function copyApplicationFiles(string sourcedir) {

   // Copy application files into desired directory
  }

}
```

نجد أن كل إجراء ذو طبيعة مختلفة عن الآخر لكنها تشترك في مهمة تثبيت البرنامج. نجد أن كل إجراء يعمل على بيانات مختلفة.

ج- التماسك المؤقت Temporal cohesion

مثل أن يتم نداء عدد من الإجراءات لا علاقة لها ببعضها عند حدث معين، مثلاً حدث مشكلة في نظام فيقوم البرنامج بإرسال رسالة نصية وبريد إلكتروني ثم إعادة تشغيل النظام:

```
class SystemFailure {

  function sendSMS(string adminMSISDN) bool {
```

```
      // send SMS to adminsitartor(s)
  }

  function sendEmail(String supportEmail) bool {

    // send email to second line support
  }

  function restartSystem() {

   // Restart for application server

  }


}
```

نلاحظ أنها إجراءات جمع بينها حدوث حدث معين ومؤقت. وهي ذات طبيعة مختلفة ومُدخلات مختلفة.

د- التماسك الإجرائي procedural cohesion:

وهي دوال تم جمعها في دالة واحدة بسبب أنه سوف يتم ندائها بالتتالي لإتمام إجراء واحد مع أن كل واحد يختلف في طبيعته. مثلاً إجراء لضغط ملف ثم تشفيره ثم إرساله كما في المثال التالي:

```
class PrepareAndSendFile {

  function CompressFile(string source, string dest) string {

     // Compress file
  }

  function EncryptFile(string source, string dest, string key) string {

      // Encrypt file
  }

  function sendFile(string afilename, string toaddress ) bool {

    // send file to email address
  }

}
```

نلاحظ أن كل دالة لها طبيعة مختلفة وتحتاج لمكتبات مختلفة، لكن ما جمعها أنه يتم ندائها بالتتالي. بها مشكلة الطبيعة المختلفة والبيانات أو المدخلات المختلفة.

ه- التماسك المعلوماتي أو الاتصالي Communicational/informational cohesion:

والمقصود به وجود دوال مع بعضها لأنها تشترك في معالجة نفس البيانات مثلاً لمعالجة سجل أو مجموعة سجلات، أو جدول، كما هذا المثال:

```
class UsersData {

  function insertNewUser(string username, string password) bool {

     // insert new user into table
  }

  function modifyUserPassword(string username, string newpassword) bool {

      // modify user password
  }

  function getUserInfo(string username) record {

    // Get user record
  }

}
```

و- التماسك المتتالي Sequential Cohesion

والمقصود به أن تكون مخرجات دالة تُستخدم كمدخلات لدالة أخرى كما في هذا المثال:

```
class Users {

  function getUsers() recordset {

     // fetch table and return recordset
  }

  function searchUser(string username, recordset records) record {

      // search user in recordset
  }

  function uploadUserInfo(record user, string aurl ) bool {

    // send user info into a web service
  }

}
```

حيث أن الإجراء الأول يقوم باسترجاع كافة السجلات للمستخدمين ثم نستخدم هذه السجلات لتكون مدخلات للدالة الثانية والتي تبحث عن سجل واحد لمستخدم والذي يقوم الدالة الثالثة بإرساله إلى خدمة ويب.

ز- التماسك الوظيفي Functional cohesion

وهو من أفضل أنواع التماسك والمقصود به أن تشترك دوال وإجراءات لتحقيق هدف واحد وواضح ومحدد. مثلاً فئة لضغط عدد من الملفات:

```
class Compression {
  function AddFile(string afilename) bool {
     // Add file to prepare it for compression
  }
  function compress(string archivename) bool {
      // Compress all added files
  }
  function getCompressionSize() int {
    // Get compressed archive file size
  }
}
```

نجد أنها كلها تخدم وظيفة واحدة وهي وظيفة ضغط الملفات.

ح- التماسك الذري Perfect cohesion (atomic)

وهو أن تكون الوحدة تحتوي على كود يمثل إجراء واحد لا يمكن تقسيمه إلى وحدة أصغر، مثلاً:

```
class Encryption {
  function encryptFile(string archivename, string key) string {
      // encrypt file
  }
}
```

الأنواع التي يمكن استخدامها وتمثل المبادئ الصحيحة لطرق البرمجة هي:

- التماسك المعلوماتي
- التماسك المتتالي
- التماسك الوظيفي (أفضلها)
- التماسك الذري (عملياً يصعب تطبيقه في المهام المعقدة)

8. مبدأ فك الارتباط decoupling

قبل الكلام عن فك أو تقليل الارتباط نتكلم عن ماهو المقصود بالارتباط: الارتباط هو ارتباط وحدتين أو إجراءين بحيث لا يمكن استخدام إجراء أو وحدة أو فئة إلا باستخدام الأخرى، أو أن عمل إجراء أو وحدة أو فئة تؤثر على الأخرى، أي أن الوحدات البرمجية غير قائمة بذاتها وغير متماسكة. وهذا من التصميم غير الصحيح في البرمجة ويجعل الوحدات المرتبطة غير قابلة لإعادة الاستخدام وصعبة في التطوير وصعبة في اكتشاف الثغرات وصعبة في إجراء الاختبارات.

المثال التالي به فئتين مرتبطتين بواسطة متغيرات عامة global variables:

```
class Email {

  public static string address;
  public static string text;

}

class Send {


  public static void sendEmail(){

     // sending Email.text to Email.address

  }
}

class Alarm {

  public void sendAlarm(String to){

    Email.address = to;
    Email.text = "System has failed";
    Sender.sendEmail();

  }
}
```

نجد أنهما مرتبطات بواسطة الفئة Email في متغيراتها address, text ولا يمكن أن تعمل الفئة Send إلا بتهيئة متغيرات الفئة Email.

يمكننا إلغاء هذه الفئة الوسيطة Email وجعل الفئة Alarm تستخدم الفئة Send مباشرة:

```
class Send {

  public static string address;
  public static string text;


  public static void sendEmail(){

     // sending text to address

  }

}

class Alarm {

  public void sendAlarm(String to){

    Send.address = to;
    Send.text = "System has failed";
    Send.sendEmail();

  }
}
```

هذه المرة قمنا بتعريف المتغيرات العامة في الفئة Send، لكن ما يزال هُناك ارتباط بين الدالة sendEmail مع هذه المتغيرات وأي تغيير في هذه المتغيرات يؤثر في سلوك الدالة sendEmail.

قمنا هذه المرة بتحويل المتغيرات العامة إلى متغيرات خاصة private لا يمكن الوصول إلها إلا عن طريق نداء دالة لتحقيق الكبسلة في البرمجة الكائنية encapsulation لمنع أي فئة من تغيير هذه المتغيرات المهمة والتي تؤثر على باقي الدوال الموجودة في هذه الفئة:

```
class Send {

  static string address;
  static string text;

  public static init(String to, String atext){
```

```
    address = to;
    text = atext;
  }

  public static void sendEmail(){

     // sending text to address

  }

}


class Alarm {

  public void sendAlarm(String to){

    Send.init(to, "System has failed");
    Send.sendEmail();

  }
}
```

مع أننا قللنا الارتباط من المرات السابقة إلا أننا قمنا بتحويل الارتباط بين الدالتين init , sendEmail حيث لابد من ندائهما بالتتالي. لفك هذا الارتباط نقوم بإضافة باراميترات للدالة sendEmail لمدها بالبيانات التي تحتاج إلها لتعمل دون أن تعتمد على دالة أخرى:

```
class Send {


  public static void sendEmail(String address, String text){

     // sending text to address

  }

}

class Alarm {

  public void sendAlarm(String to){

    Send.sendEmail(to, "System has failed");

  }
}
```

هذا مثال بسيط لتقليل الارتباط بين دالتين أو فئتين، لكن في الواقع العملي نجد هناك عدة أشكال من الارتباط ومن أشهرها الارتباط ببيانات جزئية خارجية مثل قاعدة البيانات، مثلاً النظام 2 مرتبط بالنظام

1 في قاعدة البيانات الخاصة بالنظام 1، فعند تغيير معلومات في جدول ما أو تغيير في هيكله فإن ذلك يؤثر على النظام 2، وهذه من الأخطاء التصميمية الكبيرة، تجعل تعديل النظام 1 صعباً وذلك لأن أي تغيير في قاعدة بيانات يؤثر على أنظمة أخرى ليست جزءاً من النظام الأول.

لحل مشكلة الارتباط في قاعدة البيانات يتم عمل خدمات ويب في النظام 1 وإتاحتها لباقي الأنظمة، فإذا حدث تغيير في هيكل قاعدة البيانات الخاصة بالنظام 1 يقوم المطورون بالتغيير فقط في خدمة الويب الخاصة به فقط ولا تتأثر باقي الأنظمة بهذا التغيير وأحياناً لا تحتاج حتى لمعرفة أن هُناك تغيير حدث في قاعدة البيانات. كذلك يمكن تغيير في معمارية النظام 1 دون أن يتأثر النظام 2 بهذا التغيير.

9. مبدأ القياسية في كتابة الكود code standardization

مقصود به استخدام العرف القياسي في طريقة كتابة الكود سواءً للغة البرمجة المستخدمة، أو نوعية البرامج التي يتم تطويرها.
استخدام القياسية في كتابة الكود يحقق هدف كتابة كود سهل القراءة والصيانة والفهم. يمكن أن تكون لكل مؤسسة برمجية عرف قياسي convention ومنهج محدد لكتابة وتصميم البرامج، واتباع هذا العرف يُساعد على إدارة المشروعات البرمجية وصيانتها بطريقة سهلة.
وكمثال ولشرح أوفى لهذا المفهوم هو موضوع التسميات في الأجزاء البرمجية مثل المتغيرات، والإجراءات، و الكائنات، والوحدات وحتى البرامج وأجزاءها المختلفة.
التسمية الصحيحة والمعبرة والدقيقة لكل جزئية من البرامج لا تقتصر فائدتها على كتابة كود مقروء فقط، إنما تتعدى ذلك لفوائد أخرى لا تقل أهمية، وهي أنها مقياس لصحة التصميم الداخلي للبرامج، ومقياس لمهنية المبرمج وفهمه الصحيح لهذه الطرق و المبادئ للتصميم وكتابة البرامج، بل وحتى لها دلالات لفهم الفريق البرمجي للمشكلة التي من أجلها تم تطوير البرنامج أو النظام.
تسمية الإجراءات على سبيل المثال والكائنات لابد أن يكون شاملاً ومختصراً ويعبر عن الوحدة البرمجية المستهدفة بطريقة مباشرة لوظيفتها. مثلاً إذا كان المراد كتابة دالة لقراءة معلومة من ملف إعدادات يمكن أن يكون اسمها كأحد الخيارات التالية:
readConfigValue, *getConfigValue*, *readConfigParameter*, *getConfigurationParameter*
نلاحظ أنها واضحة وتتكون من ثلاث مقاطع، المقطع الأول *read* يعني أن هذه الدالة تقوم بالقراءة، والمقطع الثاني: *Config* يشرح الهدف الذي نريد قراءته، والمقطع الثالث *Value* يحدد بصورة أكثر دقة الناتج من الدالة وهو قراءة القيمة لهذا البارامیتر.
أما إذا استخدمنا احد الأسماء التالية:
configValue, *readX*, *getValue*
نجد أن الاسم الأول: *configValue* ليس فيه الفعل الذي نريد تنفيذه وهو القراءة، لذلك يجد من يريد صيانة الكود أو التطوير فيه، يجد صعوبة فهم الغاية من هذه الدالة، و الاسم الثاني *readX* لا يوضح ماذا نريد أن نقرأ، و الاسم الثالث: *getValue* أيضاً لا يوضح أننا نريد القراءة من ملف الإعدادات.

كذلك فإن الشمولية في الاسم مهمة جداً، ويُفضل أن تكون مجردة قدر الإمكان، فإذا كانت الفئة أو الوحدة مثلاً المراد منها القراءة والكتابة من قاعدة البيانات فلا يصح أن نقوم بتسميتها *DataReader* فهي لا تشمل الكتابة فالأفضل تسميتها *DataReaderAndWriter* أو الاكتفاء بالتسمية المجردة وهي *DataClass* أو *DataManipulation* أو *Database*. والتسمية المجردة لها فائدة في التطوير المستقبلي، مثلاً إذا كانت الدوال داخل تلك الوحدة البرمجية هي فقط دوال للقراءة فإن تسميتها *DataReader* يكون صحيحاً ما لم تتم إضافة دالة للكتابة، أما إذا تمت إضافة وظيفة الكتابة في قاعدة البيانات فلابد من تغيير الاسم، لكن إذا نسى المبرمج ذلك فإذن ذلك يقود إلى أن تصبح الأسماء غير مطابقة للوظائف ويبدأ البرنامج بالحيد من إتباعه للمبادئ الصحيحة للبرمجة.

في آخر برنامج كنت اعمل عليه في الأيام الماضية وجدت أني أتوقف كثيراً في تسمية الفئات وتصنيفاتها المختلفة وكان الهدف أن لا أقوم بعمل أي خطأ في بداية التصميم وكتابة الكود، لكن هذا التأخير في وجود الأسماء المناسبة جعلني أفكر في أن السبب هو عدم فهمي الكامل للتحليل الصحيح للنظام. فاستنتجت أن التحليل الكامل ثم الفهم التام للنظام يجعل التصميم أسرع وبالتالي إيجاد أسماء للفئات بطريقة أسرع. مع أني لم أتأكد من هذه النظرية من مرجع ما، إلا أن هذا الارتباط بين وضوح التحليل والتصميم الصحيح والتسميات اصبح يتضح لي أكثر فأكثر، حيث يكفي مراجعة الأسماء لأي برنامج لمعرفة هل هُناك خلل تصميمي ما أم أن البرنامج مكتوب بطريقة مطابقة للمبادئ الصحيحة. هذه الفائدة لربط الأسماء بالمبادئ تسهل لمن يقوم بمراجعة الكود من أجل تقيمه، فكلما كانت الأسماء مجردة فهي تعني أن المبرمج أو الفريق البرمجي قام باستخدام مبدأ التجريد، وكلما كانت الأسماء تدل على هدف واحد فهي تعني استخدام مبدأ الوظيفة الواحدة، فإذا كان الاسم يحتوي على أكثر من هدف مثل *DatabaseAndConfiguration* فهي تدل على تحميل الوحدة لأكثر من هدف، أما إذا كان الاسم هو *Database* ويحتوي على قراءة للإعدادات أيضاً من الملفات فهو يعني عدم شمولية الأسماء، لكن الاسم الشامل في هذه الحال يدل على مشكلة أخرى و هي عدم استخدام مبدأ الوظيفة الواحدة، لكن مع ذلك فإن الاسم الشامل أفضل في هذه الحالة لمن يريد تعديل الكود وفصل المهام عن بعضها، فيقوم بالتركيز على الوحدات التي تحتوي على أسماء لأكثر من وظيفة لفصلها في أكثر من فئة.

تعديل الأسماء أحياناً يكون سهل وذلك في الأجزاء الداخلية للبرامج، فيتم تعديل الأسماء لعدة حالات نذكر منها: الحالة الأولى إذا لم تكن دلالاتها صحيحة، في هذه الحال تكون ضمن عملية إعادة صياغة الكود، وذلك لتصحيح الهدف من تلك الوحدة البرمجية. والسبب الثاني هو توسعة الوظيفة، وكمثال ما ذكرناه سابقاً في الوحدة التي كان اسمها *DataReader* قمنا بتجريد اسمها إلى *DataClass* لإضافة وظيفة الكتابة مع القراءة في قاعدة البيانات. السبب الثالث هو التجريد لإعادة الاستخدام في برامج

أخرى أو أجزاء أخرى. مثلاً إجراء للربط مع قاعدة البيانات المحاسبة كان اسمه *AccountingDBConnection* نقوم بتسميته إلى *DatabaseConnection* لإمكانية استخدامه مع أي برنامج وأي نظام آخر دون تحديد اسمه.
تعديل الأسماء يكون صعباً في حال أنه واجهة برمجية interface مثل اسم خدمة ويب أو إجراء ضمنها أو حتى باراميتر وذلك لأنه اتفاق يجب عدم الإخلال به مع برامج أخرى تستخدم هذه الأسماء لتنفيذ تلك الدوال فإذا تم التغيير في هذه الأسماء فإن البرنامج المستفيدة من هذه الخدمات سوف تتوقف. كذلك عند كتابة مكتبة يتم استخدامها في أكثر من نظام أو منشورة في النت، فإن تغيير الدوال يجعل البرامج المعتمدة عليها ينتج عنها خطأ في حالة إعادة ترجمتها.
من والأهداف للتي نحققها باستخدام طريقة صحيحة للتسمية هي:

- سهولة قراءة وصيانة الكود
- تحقيق التجريد
- تحقيق وتسهيل إعادة الاستخدام
- سهولة توقع وظيفة أي جزء من الكود
- معرفة نقاط الضعف في أجزاء معينة من الكود مثلاً الارتباط

المحافظة على المبادئ الصحية

من التحديات الكبيرة في تطوير البرامج هو المحافظة على أن يكون التصميم صحيحاً، حيث أن الاستمرار في تطوير البرامج وإضافة طلبات جديدة من الزبائن والتغييرات المستمرة ودخول عدد من المبرمجين في دورة تطوير ذلك النظام ربما ينتج عنه انحراف في التصميم المتفق عليه، ويمكن أن يقوم احد المبرمجين بكتابة كود بطريقة غير صحيحة مثل التكرار وعدم التجريد في الإجراءات والفئات وغيرها. فإذا زادت الأخطاء التصميمية فإن ذلك النظام يتحول من نظام ذو تصميم صحيح إلى نظام ذو تصميم غير صحيح وتبدأ تظهر عليه علامات ودلالات التصميم السيئ التي تكلمنا عنها سابقاً، وربما ينتهي به المطاف لفشله.
يمكن المحافظة على التصميم الجيد للنظام بمراقبة ومراجعة الإضافات الجديدة، وفي حال أن هناك إضافة أو تعديل تم بطريقة غير صحيحة يمكن عمل إعادة صياغة لهذه الجزئية Refactoring وكلما

كان هذا التصحيح في مرحلة مبكرة كلما كان أسهل، لكن عند التوغل في الأخطاء يصعب حينها التصحيح ويصبح مهمة مكلفة.

الخاتمة

في الختام نتمنى أن تتم الاستفادة من مادة هذا الكتاب ويكون عوناً على المبرمجين والمطورين والمعماريين والمصممين لتطوير برامجهم بطريقة سليمة تحقق كل أهداف هندسة البرمجيات وتجعل برامجنا تنافس البرامج العالمية من حيث الجودة والاعتمادية، بل وتنافسها في الكفاءة الإنتاجية لصناعة البرمجيات.