Lesson 9

clean code

Coding is for humans

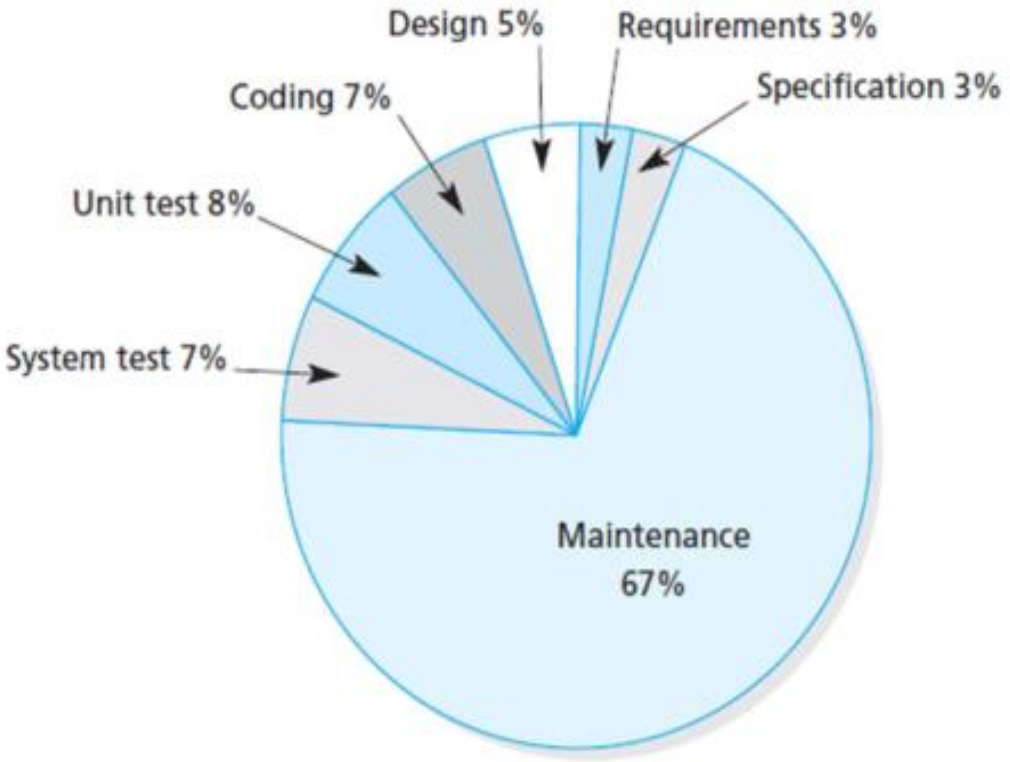
Programming is the art of telling another human what one wants the computer to do.

Donald Knuth

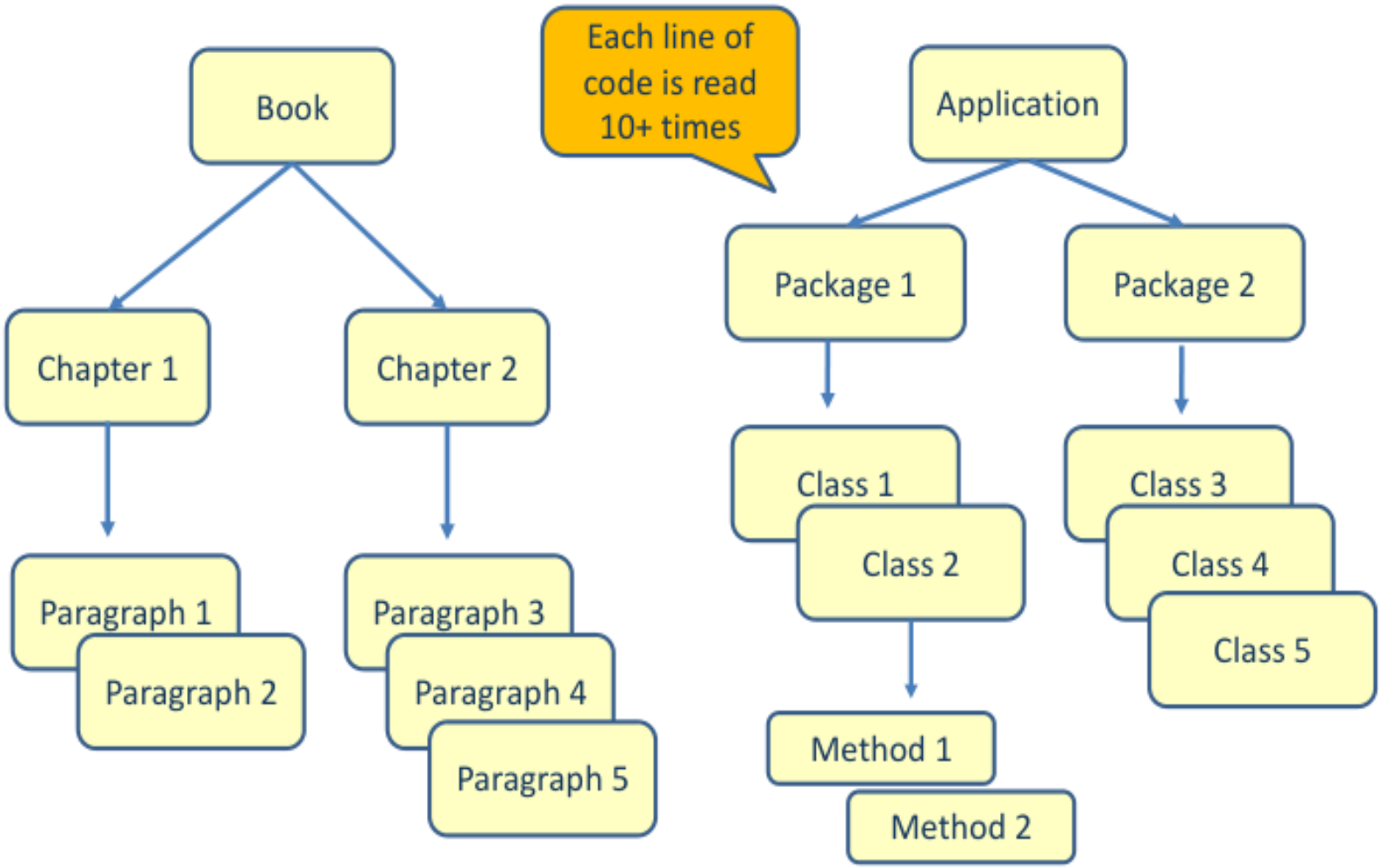
Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Martin Fowler

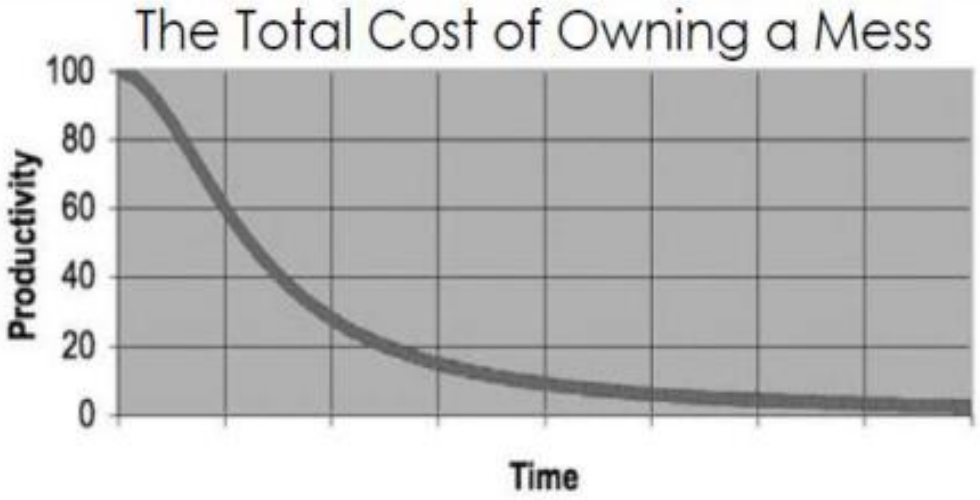
Cost of software development



You are an author!



Why clean code



"Any one can write code that a computer can understand, the trick is to write code that humans can understand."

Ugly code

```
public class Inventory {
    List<Product> sl = new ArrayList<>();

    boolean isInStock(String n) {
        Product s = new Product(n);
        int k = 0;
        int h = sl.size() - 1;

        while (k <= h) {
            int m = k + (h - k) / 2;
            int c = sl.get(m).compareTo(s);

            if (c < 0) {
                k = m + 1;
            } else if (c > 0) {
                h = m - 1;
            } else {
                return true;
            }
        }
        return false;
    }
}
```

NOT OK

Clean code

```
public class Inventory {
    List<Product> sortedList = new ArrayList<>();

    boolean isInStock(String name) {
        Product supply = new Product(name);
        int low = 0;
        int high = sortedList.size() - 1;

        while (low <= high) {
            int middle = low + (high - low) / 2;
            int comparison = sortedList.get(middle).compareTo(supply);

            if (comparison < 0) {
                low = middle + 1;
            } else if (comparison > 0) {
                high = middle - 1;
            } else {
                return true;
            }
        }

        return false;
    }
}
```

OK

MEANINGFUL NAMES

Use Intention Revealing Names

- Choosing good names takes time but saves more than it takes
- If a name requires a comment, then the name does not reveal its intent.

Use Intention Revealing Names

NOT OK

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList) {
    if (x[0] == 4)
      list1.add(x);
  }
return list1;
}
```

What is the significance of the zeroth item in the list?

What kind of things are in theList?

What is the significance of the value 4?

What does this method do?

Use Intention Revealing Names

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList) {
    if (x[0] == 4)
      list1.add(x);
  }
return list1;
}
```

NOT OK

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard) {
    if (cell.isFlagged())
      flaggedCells.add(cell);
  }
return flaggedCells;
}
```

OK

Avoid abrevations

```
int d; // elapsed time in days
int ds;
int dsm;
int faid;
```

NOT OK

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

OK

Avoid disinformation

- Avoid leaving false clues that obscure the meaning of code.
  - Do not refer to a grouping of accounts as an ***accountList*** unless it's actually a List.
    - The word **list** means something specific to programmers.
    - Plain ***accounts*** would be better.
- Beware of using names which vary in small ways.
  - XYZControllerForEfficientHandlingOfStrings
  - XYZControllerForEfficientStorageOfStrings

Avoid disinformation

```
Customer[] customerList;
Table theTable;
```

NOT OK

```
Customer[] customers;
Table customers;
```

OK

Avoid disinformation

Avoid variable names like lowercase L or uppercase O

```
int a = l;
if (O == 1)
  a = O1;
else
  l = O1;
```

NOT OK

Make meaningful distinctions

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

NOT OK

```
public static void copyChars(char source[], char destination[]) {
  for (int i = 0; i < source.length; i++) {
    destination[i] = source[i];
  }
}
```

OK

Use Pronounceable Names

- If you can't pronounce it, you can't discuss it without sounding like an idiot.
- This matters because programming is a social activity.

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private string s1 = "102";
  /*.....*/
}
```

NOT OK

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;
  private string recordId = "102";
  /*.....*/
}
```

OK

Intelligent conversation is now possible: *"Hey, Mikey, take a look at this record! The generation timestamp is set to tomorrow's date! How can that be?"*

Use searchable names

- Single-letter names and numeric constants have a particular problem in that they are not easy to locate across a body of text.
  - `MAX_CLASSES_PER_STUDENT` vs. `7`
- `e` is a poor choice for any variable

Use searchable names

```
for (int j = 0; j < 34; j++) {
  s += (t[j] * 4) / 5;
}
```

NOT OK

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j = 0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] *   realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

OK

Avoid Encodings

```
public class Part {
  private String m_dsc; // The textual description
  void setName(String name) {
    m_dsc = name;
  }
}
```

NOT OK

```
PhoneNumber phoneString;
// name not changed when type changed!
```

```
public class Part {
  private String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

```
PhoneNumber phone;
```

OK

Class names

- Should be nouns, not action: Order, Account
- Should come from the problem domain
- Should match its responsibility: OrderDAO
- Use compound names when needed: UserProfile
- Avoid meaningless names: Manager, Processor

METHODS

Method names

- Do one thing
  - calculateTotal()
  - saveOrder()
  - validateEmail()
- If it is too long, it probably needs to split, not renamed: calculateTotalAndSaveAndNotifyUser()

Method names

- Use positive concepts to name your functions

```
isConnected()
```

OK

- Do not use negative concepts

NOT OK

```
isUnconnected()


if (!isUnconnected)
```

Double negatives lead to confusion

Method names

- Use correct terms for methods that form natural pairs

```
openWindow() and closeWindow()
```

- Natural pairs

Add/Remove	Begin/End	Create/Destroy
Enable/Disable	Insert/Delete	Lock/Unlock
Next/Previous	Open/Close	Push/Pop
Send/Receive	Show/Hide	Source/Target

Methods should be small

- Methods should be easy to read
  - One clear purpose
  - Short enough to understand at a glance
  - Split If a method does too many things
- Signs a method is doing too much
  - Hard to name
  - Contains unrelated logic
  - Too many nested conditions

Method should be small

```
public void createUser(UserDto dto) {
    if (dto.name == null || dto.email == null) {
        throw new IllegalArgumentException("Missing fields");
    }

    if (!dto.email.contains("@")) {
        throw new IllegalArgumentException("Invalid email");
    }

    User user = new User();
    user.setName(dto.name);
    user.setEmail(dto.email);
    user.setCreatedAt(LocalDateTime.now());

    database.save(user);

    EmailService email = new EmailService();
    email.sendWelcomeEmail(user.getEmail());

    logger.info("User created: " + user.getEmail());
}
```

Method should be small

```
public void createUser(UserDto dto) {
    validateUserInput(dto);
    User user = buildUser(dto);
    saveUser(user);
    sendWelcomeEmail(user);
    logCreation(user);
}
```

Method should be small

```
private void validateUserInput(UserDto dto) {
    if (dto.name == null || dto.email == null)
        throw new IllegalArgumentException("Missing fields");

    if (!dto.email.contains("@"))
        throw new IllegalArgumentException("Invalid email");
}

private User buildUser(UserDto dto) {
    User user = new User(dto.name, dto.email);
    user.setCreatedAt(LocalDateTime.now());
    return user;
}

private void saveUser(User user) {
    database.save(user);
}

private void sendWelcomeEmail(User user) {
    emailService.sendWelcomeEmail(user.getEmail());
}

private void logCreation(User user) {
    logger.info("User created: " + user.getEmail());
}
```

Avoid deep nesting: return early

```
private bool ValidUsername(string username)
{
    bool isValid = false;

    const int MinUsernameLength = 6;
    if (username.Length >= MinUsernameLength)
    {
        const int MaxUsernameLength = 25;
        if (username.Length <= MaxUsernameLength)
        {
            bool isAlphaNumeric = username.All(Char.IsLetterOrDigit);
            if (isAlphaNumeric)
            {
                if (!ContainsCurseWords(username))
                {
                    isValid = IsUniqueUsername(username);
                }
            }
        }
    }
    return isValid;
}
```

NOT OK

Return late

Avoid deep nesting: return early

```
private bool ValidUsername(string username)
{
    const int MinUsernameLength = 6;
    if (username.Length < MinUsernameLength) return false;

    const int MaxUsernameLength = 25;
    if (username.Length > MaxUsernameLength) return false;

    bool isAlphaNumeric = username.All(Char.IsLetterOrDigit);
    if (!isAlphaNumeric) return false;

    if (ContainsCurseWords(username)) return false;

    return IsUniqueUsername(username);
}
```

Return early

OK

How many parameters

- Parameters add complexity to your methods
- Zero parameters is ideal
- Three parameters are allowed
  - But think twice
  - Not more than 3 parameters

```
public void SaveUser(User user, bool sendEmail, int emailFormat,
    bool printReport, bool sendBill)
```

NOT OK

```
private void SaveUser(User user)
```

OK

Method Parameters

- Use descriptive parameter names

```
char *strstr(const char *s1, const char *s2);
```

Find a substring in a string

```
char *findSubString(const char *haystack, const char *needle);
```

More descriptive

The Boolean Parameter Trap

- Boolean parameters often lead to unreadable code.
  - It's almost invariably a mistake to add a boolean parameter to an existing function.

```
widget.repaint(false);
```

It is not clear what this method does.

Expected: Don't repaint.
Real behavior: repaint, and the bool parameter specifies whether the background should be erased (the default) or not

```
widget.repaint();
widget.repaintWithoutErasing();
```

Better.

COMMENTS

Comments

- “Purpose of a comment is to explain code that does not explain itself.”
- Comments do not make up for bad code
  - Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments.
- Don‘t use a comment when you can use a method or a variable
- Comments can contain lies
  - They are not always changed with the code
  - Inaccurate comments are far worse than no comments at all

Comments

- Purpose of a comment is to explain code that does not explain itself
- Comments do not make up for bad code
- Do not use a comment when you can use a better name (method or variable)
- Comments can be outdated

Explain yourself in code

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

NOT OK

```
if (employee.isEligibleForFullBenefits())
```

OK

Create a method that says the same thing as the comment you want to write

Journal comments

NOT OK

Nowadays we have source control systems
Do not write these journal comments

```
/**
* Changes (from 11-Oct-2001)
* --------------------------
* 11-Oct-2001 : Re-organised the class and moved it to new
* package com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and
* eliminated NotableDate class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now
* that NotableDate class is gone (DG); Changed
* getPreviousDayOfWeek(),
* getFollowingDayOfWeek() and
* getNearestDayOfWeek() to correct bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate
* interface (MonthConstants) (DG);
**/
```

Noise comments

NOT OK

```
/**
* Default constructor.
*/
protected AnnualDateRule() { }

/** The day of the month. */
private int dayOfMonth;

/**
* Returns the day of the month.
* @return the day of the month.
*/
public int getDayOfMonth() {
  return dayOfMonth;
}
```

Commented-out code

NOT OK

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

Why are they commented out?
Are they important?
Can we delete it?

CODE SMELL EXAMPLES

Code smell example

```
double disabilityAmount() {
  if (seniority < 2) {
    return 0;
  }
  if (monthsDisabled > 12) {
    return 0;
  }
  if (isPartTime) {
    return 0;
  }
  // compute the disability amount
  //...
}
```

NOT OK

```
double disabilityAmount() {
  if (isNotEligableForDisability()) {
    return 0;
  }
  // compute the disability amount
  //...
}
```

OK

Code smell example

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
  charge = quantity * winterRate + winterServiceCharge;
}
else {
  charge = quantity * summerRate;
}
```

NOT OK

```
if (isSummer(date)) {
  charge = summerCharge(quantity);
}
else {
  charge = winterCharge(quantity);
}
```

OK

Code smell example

```
void renderBanner() {
  if ((platform.toUpperCase().indexOf("MAC") > -1) &&
       (browser.toUpperCase().indexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
  {
    // do something
  }
}
```

NOT OK

```
void renderBanner() {
  final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
  final boolean isIE = browser.toUpperCase().indexOf("IE") > -1;
  final boolean wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() && wasResized) {
    // do something
  }
}
```

OK

Code smell example

```
class Order {
  // ...

   public double calculateTotal() {
    double total = 0;
    for (Product product : getProducts()) {
      total += product.quantity * product.price;
    }

    // Apply regional discounts.
    switch (user.getCountry()) {
      case "US": total *= 0.85; break;
      case "RU": total *= 0.75; break;
      case "CN": total *= 0.9; break;
      // ...
    }

    return total;
  }
```

NOT OK

```
class Order {
  // ...

  public double calculateTotal() {
    double total = 0;
    for (Product product : getProducts()) {
      total += product.quantity * product.price;
    }
    total = applyRegionalDiscounts(total);
    return total;
  }

  public double applyRegionalDiscounts(double
                                        total) {
    double result = total;
    switch (user.getCountry()) {
      case "US": result *= 0.85; break;
      case "RU": result *= 0.75; break;
      case "CN": result *= 0.9; break;
      // ...
    }
    return result;
  }
```

OK

Code smell example

```
class Bird {
  //...
  double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return getBaseSpeed();
      case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
      case NORWEGIAN_BLUE:
        return (isNailed) ? 0 : getBaseSpeed(voltage);
    }
    throw new RuntimeException("Should be unreachable");
  }
}
```

NOT OK

Code smell example

```
abstract class Bird {
  //...
  abstract double getSpeed();
}
```

```
class European extends Bird {
  double getSpeed() {
    return getBaseSpeed();
  }
}
```

```
class African extends Bird {
  double getSpeed() {
    return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
  }
}
```

```
class NorwegianBlue extends Bird {
  double getSpeed() {
    return (isNailed) ? 0 : getBaseSpeed(voltage);
  }
}
```

```
// Somewhere in client code
speed = bird.getSpeed();
```

OK