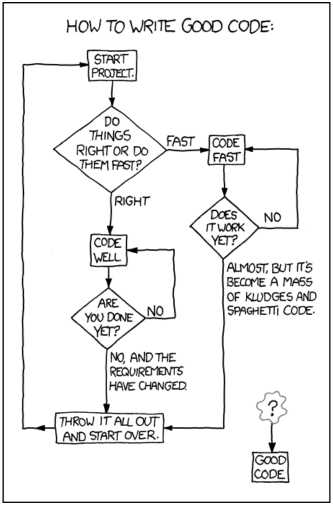
Writing Maintainable Code

Cynthia Kop

14 February, 2022


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Source: `http://www.xkcd.com`

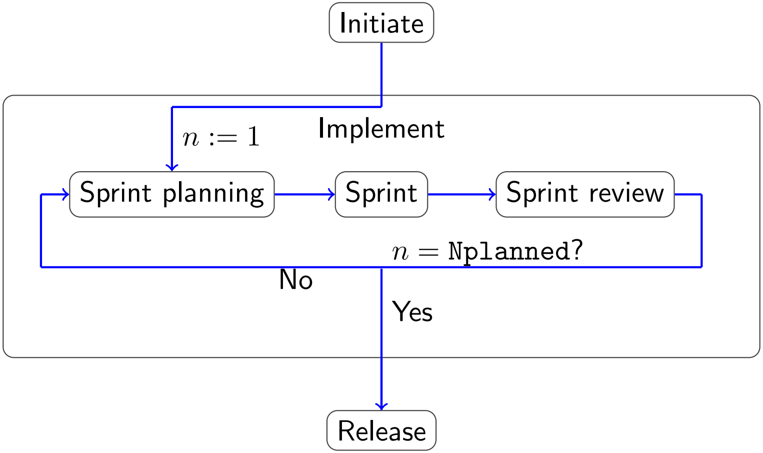
Recap: agile development

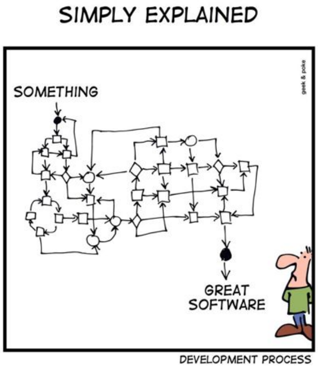
We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Recap: Scrum


Initiate
Implement
$n := 1$
Sprint planning
Sprint
Sprint review
$n =$ `Nplanned`?
No
Yes
Release


SIMPLY EXPLAINED
SOMETHING
GREAT SOFTWARE
geek & poke
DEVELOPMENT PROCESS

Which decisions should you take early on?

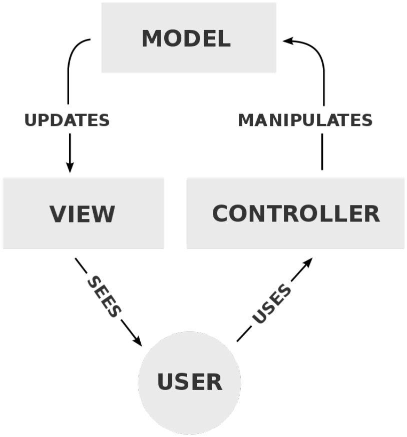
- the programming language ✓
- the coding standards in your team ✓
- the overall architecture ✓
- the structure of your classes X

Architecture

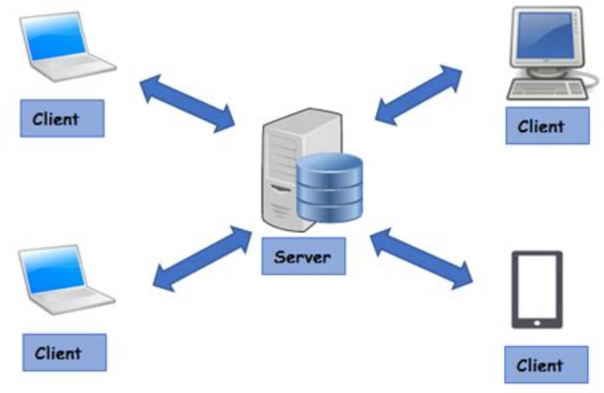
Architecture: how the system is structured overall, decomposed and organised into components, and interfaces between them

- includes decisions like programming language and platform
- various architecture patterns to make it easier to achieve high-quality code
- change later is hard (but not impossible)

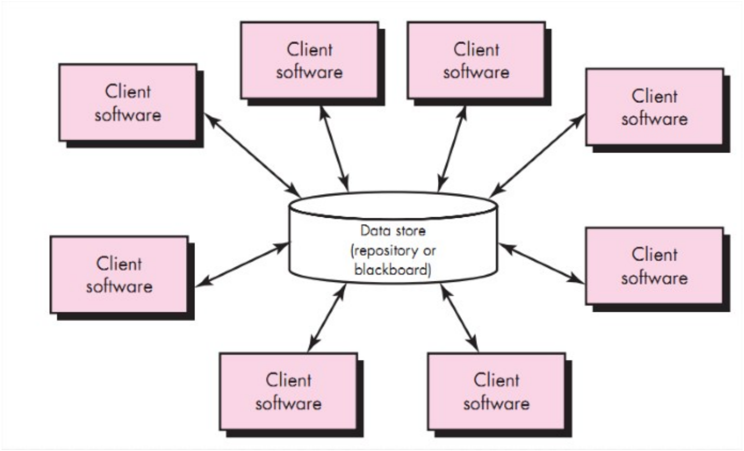
Model-View-Controller pattern



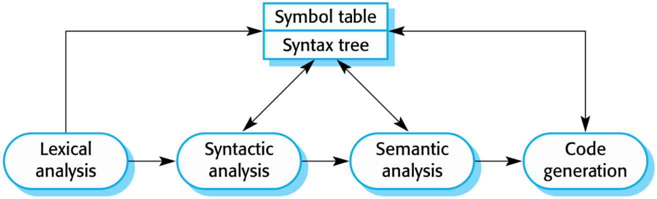
Client-server pattern



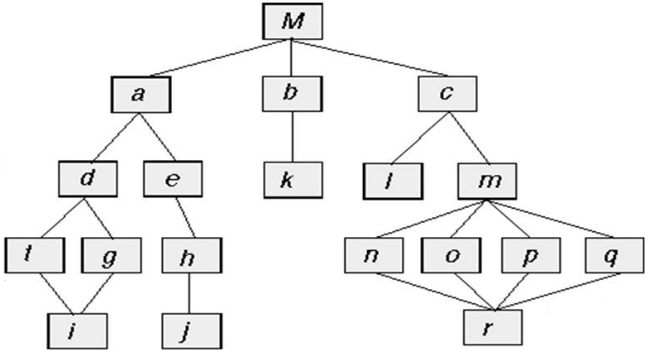
Data-centered pattern



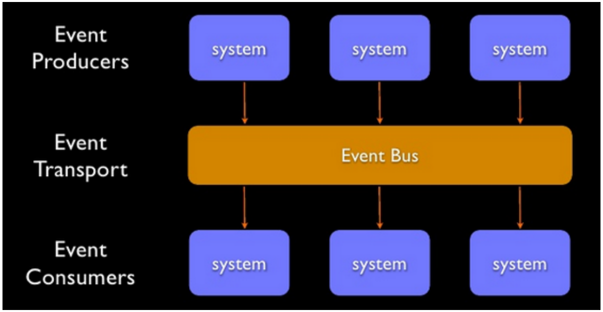
Pipe-and-filter pattern



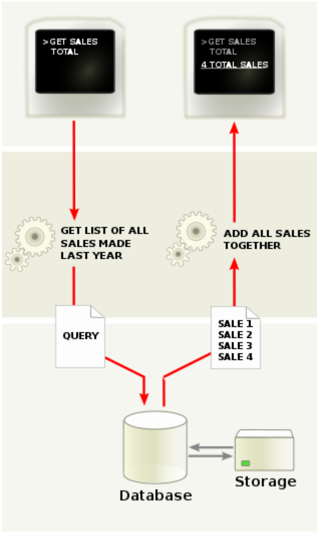
Call-and-return pattern



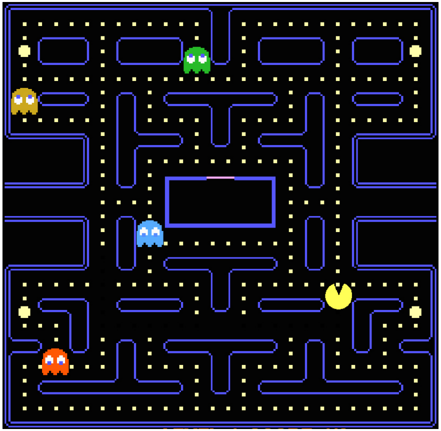
Even-driven pattern



Three-tier pattern



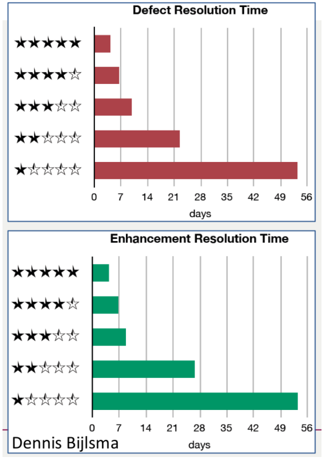
Challenge: design an architecture



A warning

Avoid too much up-front design!

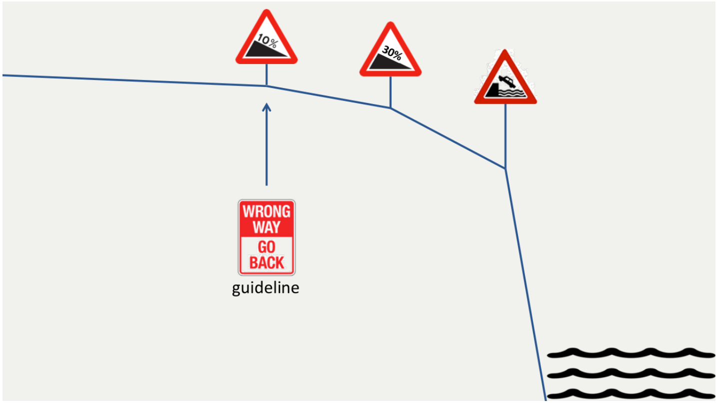
"Current development speed is a function of past development quality."
– Brian McAllister



Technical debt

"With borrowed money, you can do something sooner than you might otherwise, but then until you pay back that money you'll be paying interest. I thought borrowing money was a good idea, I thought that rushing software out the door to get some experience with it was a good idea, but that of course, you would eventually go back and as you learned things about that software you would repay that loan by refactoring the program to reflect your experience as you acquired it."

Ward Cunningham


10%
30%
WRONG WAY GO BACK
guideline

"Improving maintainability does not require magic or rocket science. A combination of relatively simple skills and knowledge, plus the discipline and environment to apply them, leads to the largest improvement in maintainability." [p. xi]

"Maintainability is not an afterthought, but should be addressed from the very beginning of a development project. Every individual contribution counts." [p. 4]

O'REILLY
Building Maintainable Software
TEN GUIDELINES FOR FUTURE-PROOF CODE
Joost Visser

Writing high-quality code

- **Coupling:** how strong are the connections between separate parts of your code?
- **Cohesion:** how well does code that is put together really belong together?

Coupling

- strength of interconnections, measure of interdependence
- the more we must know about A to understand or work with B, the higher their coupling
- increases with complexity and obscurity of interfaces
- **Goal:** keep coupling low
    - high coupling means greater cost to making changes
    - high coupling makes it harder to test separate parts, and decreases readability

Types of coupling

From high coupling to low coupling:

- content coupling
- common coupling
- control coupling
- stamp coupling
- data coupling
- message coupling

Types of coupling

From high coupling to low coupling:

- content coupling
- common coupling
- control coupling
- stamp coupling
- data coupling
- message coupling

Content coupling

One module relies on the internal workings of another module.

Content coupling

```
public void addUserProps(string user, string *props){
  ⟦actions to load the user into variable _myUser⟧
  _myUser.properties.append(props);
}

public string *queryUserProperties(string user){
  addProperty(playername, {});
  return copy(_myUser.properties);
}
```

Content coupling

```
class A {
  int arr[3];
  int []get_arr() { return arr; }
}

class B {
  void myfun(A a) {
    int brr[] = a.get_arr();
    brr[1] = 2;
  }
}
```

Types of coupling

From high coupling to low coupling:

- content coupling
- common coupling
- control coupling
- stamp coupling
- data coupling
- message coupling

Common coupling

Two or more modules share some information by using global data.

Common coupling

```
void f() { if (settings_screen == BIG) { ...} }
...
void g(int kind) { ...settings_screen = kind; ...}
```

Common coupling

```
class A {
  C mydata;
  void set_data(C data) { mydata = data; }
}
class B {
  C mydata;
  void set_data(C data) { mydata = data; }
}
void setup() {
  A a;
  B b;
  C c;
  ...
  a.set_data(c);
  b.set_data(c);
}
```

Common coupling in Discworld

Job market in Ankh-Morpork

- "Buy 2 red dresses and deliver them to Ms. Cosmopilite."

Shop in Djelibeybi

- Is the only place in the game that sells red dresses.

Types of coupling

From high coupling to low coupling:

- content coupling
- common coupling
- control coupling
- stamp coupling
- data coupling
- message coupling

Control coupling

One module controls the flow of another.

Control coupling

```
void a(boolean flag) {
  // do some shared preparation stuff
  if (flag) { // do thing 1 }
  else { // do thing 2 }
}

void b() {
  ...
  a(true);
}
```

Control coupling

```
cleanupConnections(boolean force) {
  Connection *remainder = new Array();
  foreach (Connection c in connections) {
    int errcode = c.close();
    if (errcode == 1) {
      if (force) c.forceClose();
      else remainder.add(c);
    }
  }
  connections = remainder;
}
```

Control coupling

```
bool updateBirth(string user, Date bd, bool verify){
  int age = calculateAge(bd, time());
  if (verify) {
    if (age < 18) {
    popup("You are too young to participate!\n");
    return false;
  }
  Account account = loadUser(user);
  user.setBirthday(bd);
}
```

Types of coupling

From high coupling to low coupling:

- content coupling
- common coupling
- control coupling
- stamp coupling
- data coupling
- message coupling

Stamp coupling

A composite data structure is shared between modules.
Note: this is already pretty low coupling. But if several functions only use parts of the same data structure, it may be worth splitting the structure.

Stamp coupling

```
typedef struct rectangle {
  int length, width, area, perimeter, color;
} RECTANGLE;

int calcArea(RECTANGLE r) {
  return r.length * r.width;
}

void main() {
  RECTANGLE rect;
  rect.length = 7;
  rect.width = 6;
  rect.color = RED;
  rect.area = calcArea(rect);
}
```

Stamp coupling

```
class GameElements {
  GameObject *board[WIDTH][HEIGHT];
  boolean minesVisible;
  int timeOfNextReset();
  ...
}

class Player {
  void init(GameElements ge) {
    // find empty place on the board
    // and put the player there
  }
}
```

Types of coupling

From high coupling to low coupling:

- content coupling
- common coupling
- control coupling
- stamp coupling
- data coupling
- message coupling

Data coupling

Elementary data is passed between modules (for example: through parameter passing).

Data coupling

```
class A {
  int k;

  void f() {
    ...
    int tmp = Util.sqrt(k);
    ...
  }
}
```

Data coupling

```
typedef struct rectangle {
  int length, width, area, perimeter, color;
} RECTANGLE;

int calcArea(int length, int width) {
  return r.length * r.width;
}

void main() {
  RECTANGLE rect;
  rect.length = 7;
  rect.width = 6;
  rect.color = RED;
  rect.area = calcArea(rect.length, rect.width);
  ...
}
```

Types of coupling

From high coupling to low coupling:

- content coupling
- common coupling
- control coupling
- stamp coupling
- data coupling
- message coupling

Message coupling

One module sends a message to another without passing any data.

Message coupling

```
void keyPressed(Key k) {
  if (k.isEscapeKey()) {
    foreach (KeyListener kl in listeners) {
      kl.escapePressed();
    }
  }
}
```

Cohesion

- strength of inner bonds, relationships
- concept of whether elements belong together or not, measure of how focused the responsibilities are
- generally: the higher the cohesion within each module, the looser the coupling between the modules
- high cohesion gives greater reusability and readibility, and lower complexity
- in an OO setting:
  - method cohesion
  - class cohesion
  - inheritance cohesion

Class cohesion

- Why different attributes and methods are together
- Do they contribute to supporting exactly one concept?
- Or can the methods be partitioned into groups, each accessing (almost only) a distinct subset of attributes?
- Splitting could introduce more coupling, but is still preferable.

Inheritance cohesion

- Why classes are together in a hierachy.
- Main reasons: generalisation-specialisation, code reuse
- More about this next week!

Types of cohesion

Why code is together within a method/module/class, from worst to best:

- coincidental
- logical
- temporal
- procedural
- communicational
- sequential
- functional
- atomic

Types of cohesion

Why code is together within a method/module/class, from worst to best:

- coincidental
- logical
- temporal
- procedural
- communicational
- sequential
- functional
- atomic

Coincidental cohesion

Elements in a module are grouped together arbitrarily, with no relationship between them.

Coincidental cohesion

```
class Utilities {
  pretty_print(string format, Object[] data) { ... }
  int average(int a, int b) { ... }
  int maximum(int a, int b) { ... }
}
```

Types of cohesion

Why code is together within a method/module/class, from worst to best:

- coincidental
- logical
- temporal
- procedural
- communicational
- sequential
- functional
- atomic

Coincidental cohesion

Elements in a module are grouped together because they are in some way logically related, although their functionality is very different.

Logical cohesion

```
module PolygonFunctionality() {
  void areaOfTriangle(int a, int b, int c) { ...  }
  void perimeterOfRectangle(int a, int b) {...}
  ...
}
```

Types of cohesion

Why code is together within a method/module/class, from worst to best:

- coincidental
- logical
- temporal
- procedural
- communicational
- sequential
- functional
- atomic

Temporal cohesion

Elements in a module are grouped together because they are used at the same time.

Temporal cohesion

```
void init() {
  count = 0;
  open_student_file();
  error = null;
}

void error_recovery() {
  ⟦close open files⟧
  ⟦reset some variables⟧
  ⟦restart main loop⟧
}
```

```
void tetris_block_fall(int block_id) {
  update_timer();
  move_block_one_down(block_id);
  if (block_has_landed(block_id)) {
    PAUSE(100);
    handle_landing(block_id);
  }
}
```

Types of cohesion

Why code is together within a method/module/class, from worst to best:

- coincidental
- logical
- temporal
- procedural
- communicational
- sequential
- functional
- atomic

Procedural cohesion

Elements in a module are grouped together because they are executed sequentially to perform a certain task.

Procedural cohesion

```
void store_address(string address, string user) {
  ⟦verify that the user exists⟧
  ⟦verify that the address is valid⟧
  ⟦establish connection to the database⟧
  ⟦execute appropriate sql query⟧
}
```

Types of cohesion

Why code is together within a method/module/class, from worst to best:

- coincidental
- logical
- temporal
- procedural
- communicational
- sequential
- functional
- atomic

Procedural cohesion

Two elements work on the same input data and/or produce the same output data.

Communicational cohesion

```
void determine_customer_details(int accountno) {
  ⟦do some work to find the name⟧
  ⟦do some work to find the loan balance⟧
  return new c_details(name, balance)
}
```

Types of cohesion

Why code is together within a method/module/class, from worst to best:

- coincidental
- logical
- temporal
- procedural
- communicational
- sequential
- functional
- atomic

Sequential cohesion

Parts of a module are grouped together because they should be executed sequentially, with the output from one part serving as the input to the next.

Sequential cohesion

```
void handle_record(RECORD record) {
  record.user = _my_user;
  record.valid = check(record.user, record.account);
  return record;
}
```

Types of cohesion

Why code is together within a method/module/class, from worst to best:

- coincidental
- logical
- temporal
- procedural
- communicational
- sequential
- functional
- atomic

Functional cohesion

In a single component, all the essential elements are combined together for performing a single task, and only those.

Functional cohesion

```
float calculate_sine(int angle) { ... }
RECORD read_transaction_record(int trans_id) { ... }
void assign_seat(int user_id, int seat_id) { ... }
```

Types of cohesion

Why code is together within a method/module/class, from worst to best:

- coincidental
- logical
- temporal
- procedural
- communicational
- sequential
- functional
- atomic

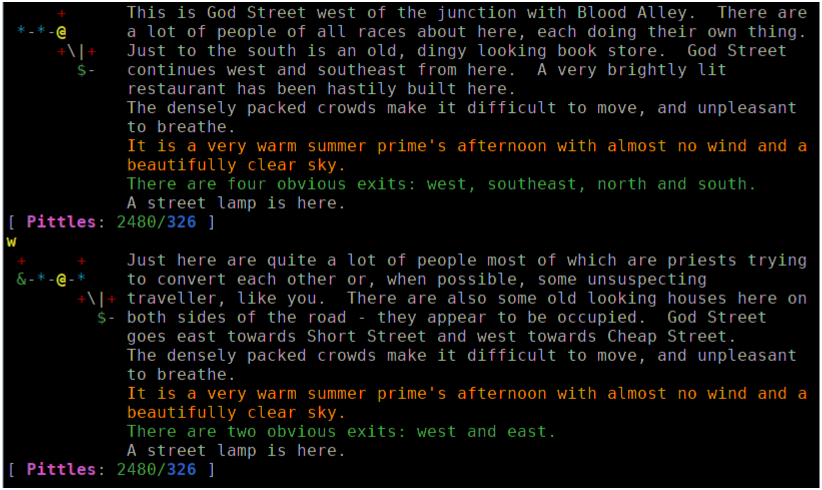
Atomic cohesion

A single component that cannot be reduced any further.

Atomic cohesion

```
int myfun(x) {
  return 5 * x + 3;
}
```

Challenge: room information in Discworld



```
     +       This is God Street west of the junction with Blood Alley.  There are
 *-*-@       a lot of people of all races about here, each doing their own thing.
     +\|+    Just to the south is an old, dingy looking book store.  God Street
       $-    continues west and southeast from here.  A very brightly lit
             restaurant has been hastily built here.
             The densely packed crowds make it difficult to move, and unpleasant
             to breathe.
             It is a very warm summer prime's afternoon with almost no wind and a
             beautifully clear sky.
             There are four obvious exits: west, southeast, north and south.
             A street lamp is here.
[ Pittles: 2480/326 ]
w
 +     +     Just here are quite a lot of people most of which are priests trying
 &-*-@-*     to convert each other or, when possible, some unsuspecting
       +\|+  traveller, like you.  There are also some old looking houses here on
         $-  both sides of the road - they appear to be occupied.  God Street
             goes east towards Short Street and west towards Cheap Street.
             The densely packed crowds make it difficult to move, and unpleasant
             to breathe.
             It is a very warm summer prime's afternoon with almost no wind and a
             beautifully clear sky.
             There are two obvious exits: west and east.
             A street lamp is here.
[ Pittles: 2480/326 ]
```

Challenge: room information in Discworld

```
varargs int move_with_look( mixed dest, string
      messin, string messout ) {
  return_to_default_position(1);
  if ( (int)this_object()->move( dest, messin,
         messout ) != MOVE_OK )
    return 0;
  room_look();
  return_to_default_position(1);
  return 1;
}
```

Challenge: room information in Discworld

```
int room_look() {
  if ( !( interactive( this_object() ) ) )
    return 0;
  this_object()->ignore_from_history( "look" );
  this_object()->bypass_queue();
  command( "look" );
  return 1;
}
```

Challenge: room information in Discworld

The look command:

- calculates the degree of darkness (for visibility)
- checks the lookmap setting for the player
- calls `environment(this_player())->long_lookmap(dark, lookmap)`
- prints the result to the player

Challenge: room information in Discworld

```
string long_lookmap(int dark, int lookmap_type) {
  if( dark )
    return 0;

  return lookmap_text(long(dark), lookmap_type);
}
```

Challenge: room information in Discworld

```
string lookmap_text(string text, int lookmap_type) {
  string ret = text;
  string map = lookmap(this_player()->map_setting());
  send_room_info(this_player(), map);
  switch(lookmap_type) {
    case NONE: return text;
    case TOP: return map + text;
    case LEFT: return combine(map, text);
  }
}
```

Challenge: room information in Discworld

```
void send_room_info(object player, string map) {
  // send metadata "room.info":  room and city name
  if (player->map_setting() == ASCIIMAP) {
    string writmap = lookmap(WRITTENMAP);
    player->send_metadata("room.map", map);
    player->send_metadata("room.writmap", writmap);
  } else {
    string asciimap = lookmap(ASCIIMAP);
    player->send_metadata("room.map", asciimap);
    player->send_metadata("room.writtenmap", map);
  }
}
```

Challenge: room information in Discworld

- Identify where coupling and cohesion are bad.
- Suggest improvements to the design of the “player enters a room, is given a room description and metadata” code.

Goal: functional cohesion in all modules (functions, units)

Method: move cohesive sub-functionality into separate modules!

Goal: functional cohesion in all modules (functions, units)

Method: move cohesive sub-functionality into separate modules!

```
void error_recovery() {
  ⟦close open files⟧
  ⟦reset some variables⟧
  ⟦restart main loop⟧
}
```

Goal: functional cohesion in all modules (functions, units)

Method: move cohesive sub-functionality into separate modules!

```
void error_recovery() {
  close_all_open_files();
  reset_file_data();
  restart_main_loop();
}
```

Goal: functional cohesion in all modules (functions, units)

Method: move cohesive sub-functionality into separate modules!

```
void error_recovery() {
  close_all_open_files();
  reset_file_data();
  restart_main_loop();
}

void store_address(string address, string user) {
  ⟦verify that the user exists⟧
  ⟦verify that the address is valid⟧
  ⟦establish connection to the database⟧
  ⟦execute appropriate sql query⟧
}
```

Goal: functional cohesion in all modules (functions, units)

Method: move cohesive sub-functionality into separate modules!

```
void error_recovery() {
  close_all_open_files();
  reset_file_data();
  restart_main_loop();
}

void store_address(string address, string user) {
  int user_index = find_user(user);
  validate_address();
  class db = open_db_connection();
  db.store_address(user_index, address);
}
```

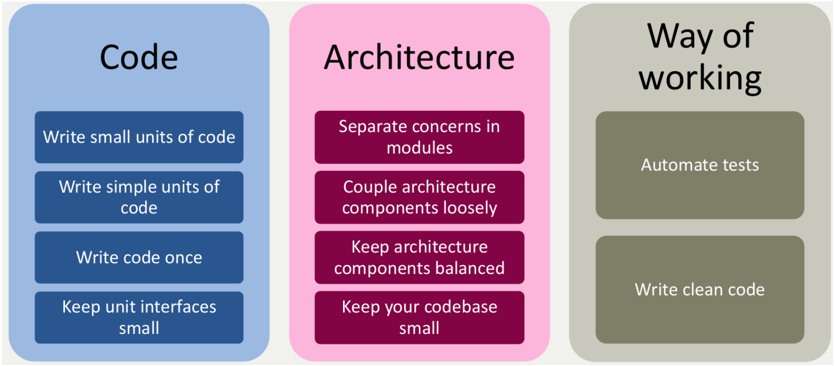
Goal: functional cohesion in all modules (functions, units)

Note: moving functionality into separate modules is not always enough!

```
void tetris_block_fall(int block_id) {
  update_timer();
  move_block_one_down(block_id);
  if (block_has_landed(block_id)) {
    PAUSE(100);
    handle_landing(block_id);
  }
}
```

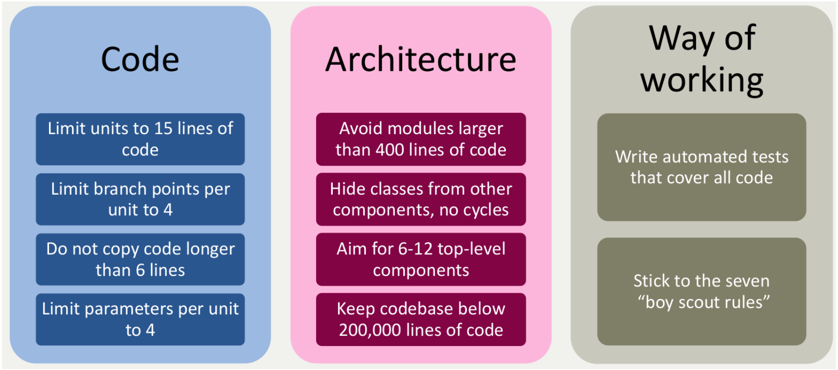
However: having the separate functions helps with restructuring!

Basic Guidelines


Code
Write small units of code
Write simple units of code
Write code once
Keep unit interfaces small
Architecture
Separate concerns in modules
Couple architecture components loosely
Keep architecture components balanced
Keep your codebase small
Way of working
Automate tests
Write clean code

– Software Improvement Group

Basic Guidelines



Code

- Limit units to 15 lines of code
- Limit branch points per unit to 4
- Do not copy code longer than 6 lines
- Limit parameters per unit to 4

Architecture

- Avoid modules larger than 400 lines of code
- Hide classes from other components, no cycles
- Aim for 6-12 top-level components
- Keep codebase below 200,000 lines of code

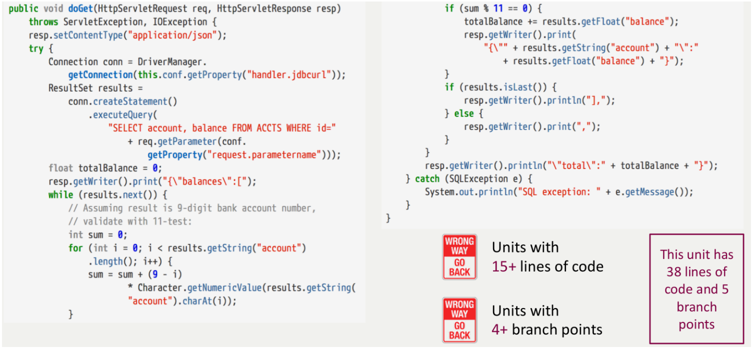
Way of working

- Write automated tests that cover all code
- Stick to the seven "boy scout rules"

– Software Improvement Group

Keep your functions (units) manageable.

- Functions should not be too long (guideline: $\leq$ 15 lines)
- Functions should not have too many decisions in them (guideline: $\leq$ 4 branch points)



```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
    resp.setContentType("application/json");
    try {
        Connection conn = DriverManager.
            getConnection(this.conf.getProperty("handler.jdbcurl"));
        ResultSet results =
            conn.createStatement()
                .executeQuery(
                    "SELECT account, balance FROM ACCTS WHERE id="
                        + req.getParameter(conf.
                            getProperty("request.parametername")));
        float totalBalance = 0;
        resp.getWriter().print("{\"balances\":[");
        while (results.next()) {
            // Assuming result is 9-digit bank account number,
            // validate with 11-test:
            int sum = 0;
            for (int i = 0; i < results.getString("account")
                .length(); i++) {
                sum = sum + (9 - i)
                    * Character.getNumericValue(results.getString(
                    "account").charAt(i));
            }
            if (sum % 11 == 0) {
                totalBalance += results.getFloat("balance");
                resp.getWriter().print(
                    "{\"" + results.getString("account") + "\":"
                        + results.getFloat("balance") + "}");
            }
            if (results.isLast()) {
                resp.getWriter().println("],");
            } else {
                resp.getWriter().print(",");
            }
        }
        resp.getWriter().println("\"total\":" + totalBalance + "}");
    } catch (SQLException e) {
        System.out.println("SQL exception: " + e.getMessage());
    }
}
```

WRONG WAY GO BACK Units with 15+ lines of code

WRONG WAY GO BACK Units with 4+ branch points

This unit has 38 lines of code and 5 branch points

Limit number of parameters

```
int *parry_modifier(object defender, object attacker,
       object defense_weapon, object attack_weapon,
       int parry_defense_bonus, int distance,
       int give_feedback) {
  ...
}
```

Fine when used as a stand-alone occurrence (although it may be indicative that your function is too large / complex), but often the same large set of parameters occurs in multiple places. Should some of these parameters be combined into a single structure, e.g., attack_data?

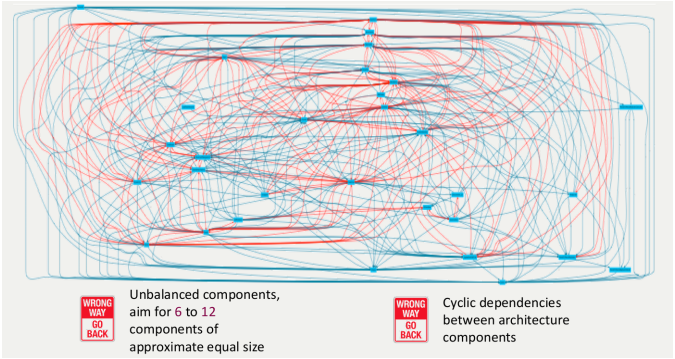
Duplicate code

If you copy N lines of code
with $N \geq 3$
then you should do $(N-3) * 5$ pushups!

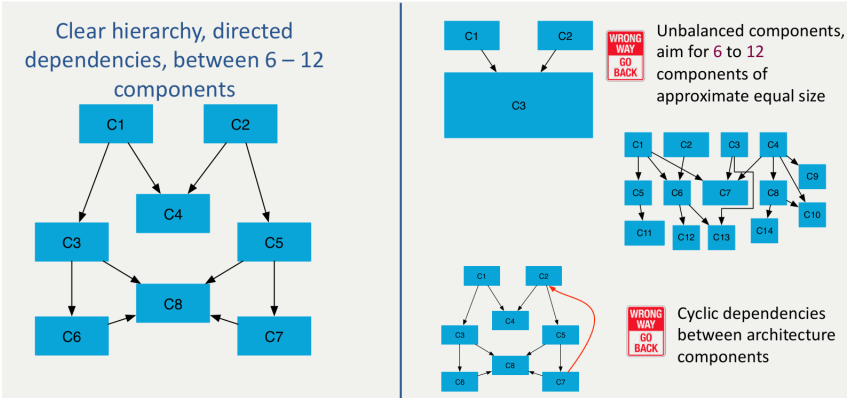
- an idea should only be expressed in one place (ease of change)
- you already need it more than once – make it more reusable!

Keep your architecture manageable.

- Modules should not be too long (guideline: $\leq$ 400 lines)
- Overall codebase should not be too large (guideline: $\leq$ 200k lines)
- Few top-level components, minimise connections between them


WRONG WAY GO BACK
Unbalanced components, aim for 6 to 12 components of approximate equal size
WRONG WAY GO BACK
Cyclic dependencies between architecture components

Keep your architecture manageable.


Clear hierarchy, directed dependencies, between 6 – 12 components
C1
C2
C3
C4
C5
C6
C7
C8
WRONG WAY GO BACK
Unbalanced components, aim for 6 to 12 components of approximate equal size
C9
C10
C11
C12
C13
C14
WRONG WAY GO BACK
Cyclic dependencies between architecture components

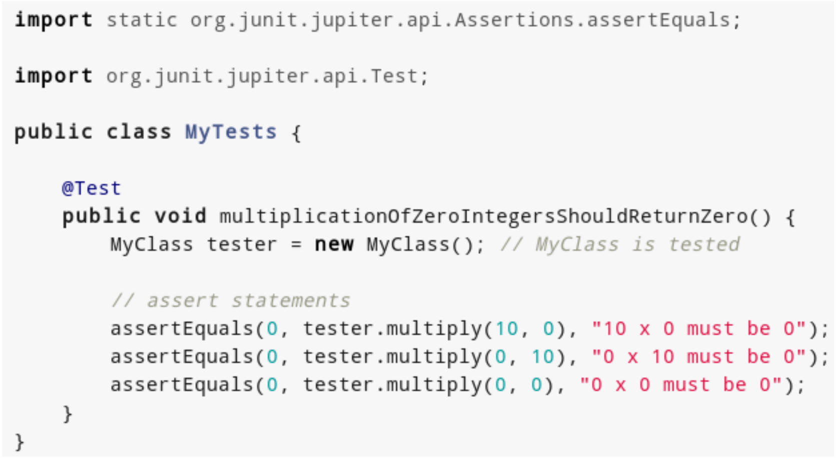
Test all non-trivial code automatically

Manual:

```
void test_validation() {
  do {
    string num = input("Type account number:  ");
    println("Result:  " + validate_account(num));
  }
}
```

Test all non-trivial code automatically

Automatic:



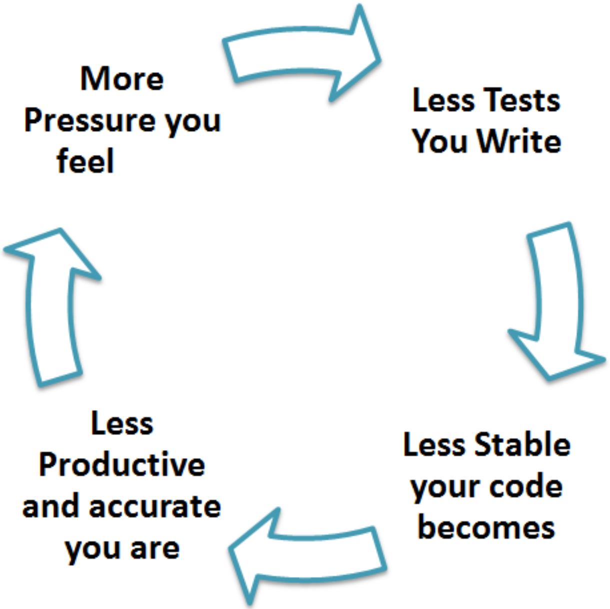
```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class MyTests {

    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {
        MyClass tester = new MyClass(); // MyClass is tested

        // assert statements
        assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");
        assertEquals(0, tester.multiply(0, 10), "0 x 10 must be 0");
        assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");
    }
}
```


More Pressure you feel
Less Tests You Write
Less Stable your code becomes
Less Productive and accurate you are

Boy Scout Rule

Leave the campground cleaner than you found it.

- Leave no unit level code smells behind.
- Leave no bad comments behind.
- Leave no code in comments behind.
- Leave no dead code behind.
- Leave no long identifier names behind.
- Leave no magic constants behind.
- Leave no badly handled exceptions behind.

Maintaining a lowly coupled, highly cohesive design that can adapt to change

~~The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so they can evolve accordingly. – Gang of Four~~

Avoid premature generalisation!

Design in eXtreme Programming

When implementing a new feature:

1. write a test
2. write code that satisfies the test
3. look back and realise if a change in design is required
4. refactor

If design documents are required, make them afterwards.

Incremental design

During/after implementing, ask questions:

- Is this code similar to other code in the system?
- Are class responsibilities clearly defined?
- Are concepts clearly represented?
- How well does this class interact with other classes?

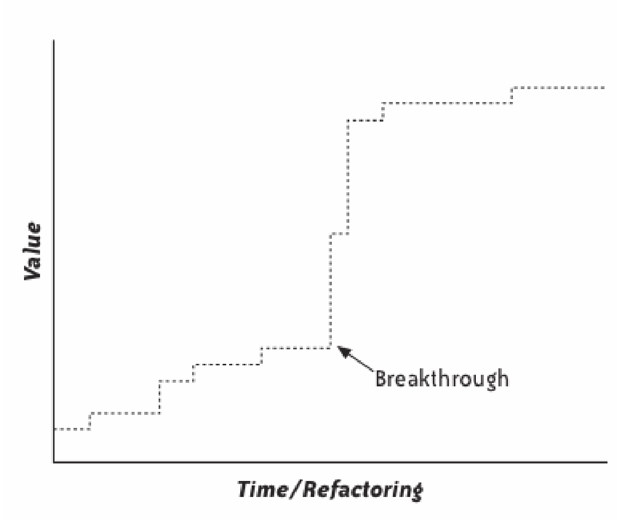
If there is a problem:

- Jot it down, and finish what you're doing.
- Discuss with teammates (if needed).
- Follow the ten-minute rule.

Incremental design

- The first time you create a design element, be completely specific.
- The second time you work with an element, make it general enough to solve both problems.
- The third time, generalise it further.
- By the fourth or fifth time, it's probably perfect!

Incremental design



Simplicity in agile design

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

– Antoine de Saint-Exupéry

Any intelligent fool can make things bigger, more complex and more violent. It takes a touch of genius and a lot of courage to move in the opposite direction.

– Albert Einstein

Keep It Simple, Stupid

– U.S. Navy

Simplicity in agile design

Keep It Simple, Stupid

The system should be:

- appropriate for the intended audience
- communicative
- factored
- minimal

Agile and incremental design

See also:

- `http://www.jamesshore.com/Agile-Book/simple_design.html`
- `http://www.jamesshore.com/Agile-Book/incremental_design.html`