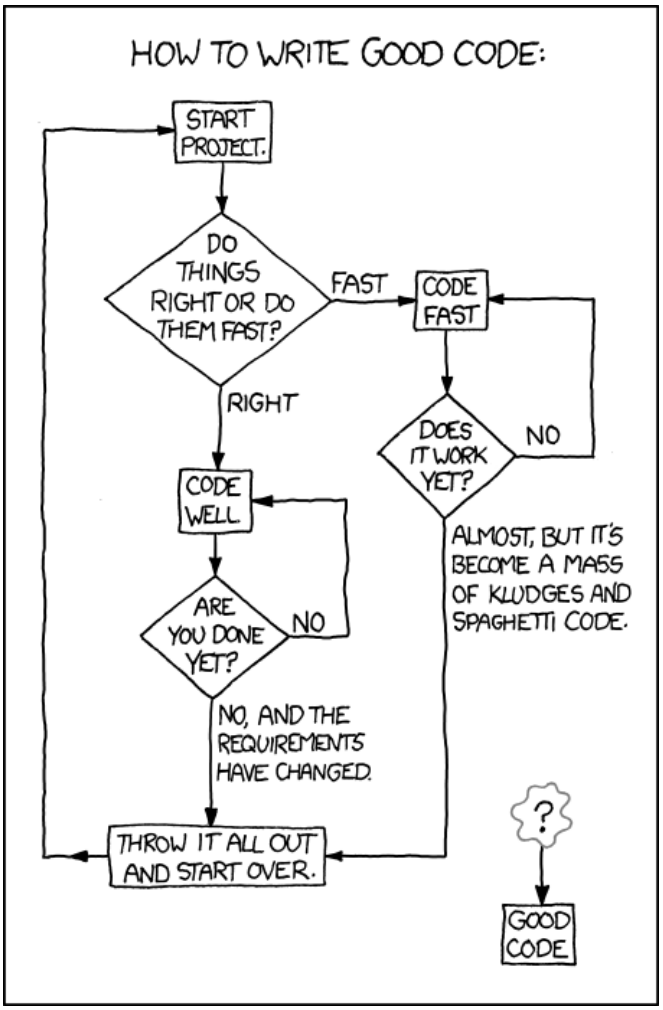
1

Handout for lecture 3 (SE):

Writing Maintainale Code

2



http://www.xkcd.com

1. Recap

3

Recap: agile development

Recall from the last lecture the agile manifesto.

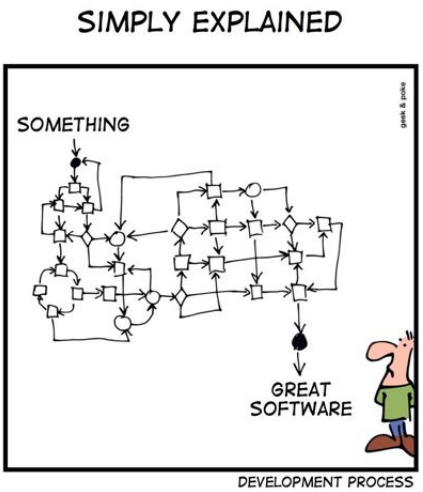
> We are uncovering better ways of developing
> software by doing it and helping others do it.
> Through this work we have come to value:

- 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.

In this lecture, we will consider how to write code in a way that supports agile development. In particular, the working software and responding to change parts are relevant here. Code should be written so as to quickly provide a working version, yet be ready for regular big changes, while still not turning into a great tangled mess. Doing this requires both a good understanding of what makes code maintainable, and good practices to maintain a high standard.

4



Fortunately, in practice the creation of good, highly maintainable code is not so difficult as demonstrated here. It is largely about following simple rules.

2. Early decisions

5

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

To start, consider what decisions should be made at the start. You will need to agree on a programming language, and the standards in your team. If some people use two-space indents and some use three-space indents, it can already create a lot of confusion – so make these agreements!

Since development should be agile, intricate decisions about the structure of the code should not be made up-front, but rather discovered during refinement sessions or while coding. However, in Scrum at least the overall architecture *is* a decision typically taken at the start.

6

Architecture

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

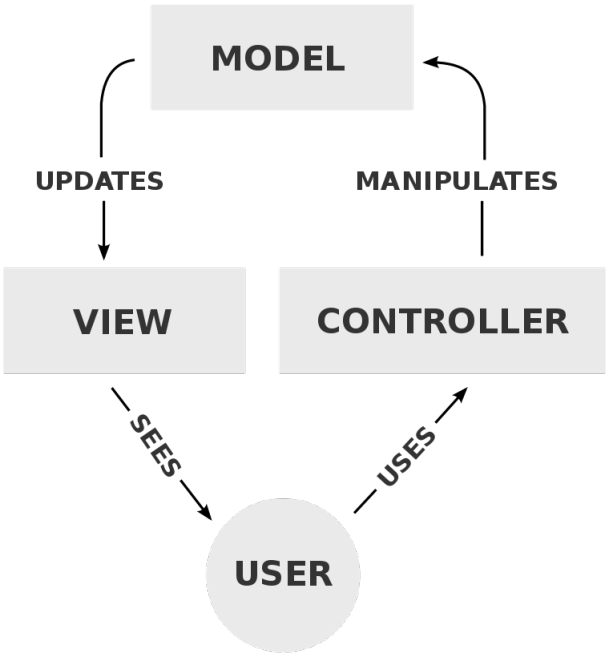
- includes decisions like programming language and platform
- various architecture patterns to make it easier to achieve high-quality code
- change later is hard

While it is possible to discover an architecture as you code (and this is done for instance in eXtreme Programming), changing the architecture can take quite a lot of time, which at least in this course is not feasible.

We will go through some examples of architecture *patterns*: typical ways to design an overall architecture of your software. There is plenty of material available on each of these patterns if you are interested in learning more more about a specific one.

7

Model-View-Controller pattern

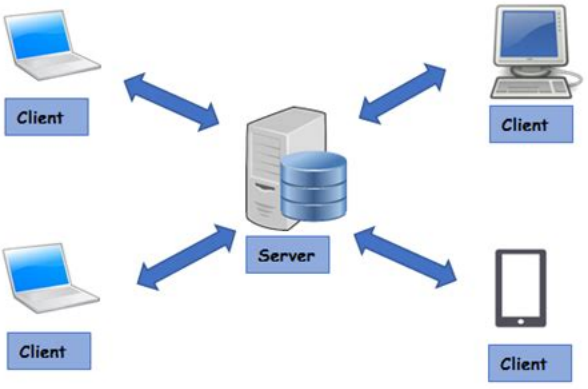


In the model-view-controller pattern, the application is split into three components. The *model* is the central component, which contains the logic of the application. The *view* shows things to the user, while the *controller* handles user input. Of course, each component can be further split up into separate sub-components.

Aside from separating concerns, the MVC pattern has the advantage of making your code very *testable*. Because all the logic, data manipulation etc. is in the model, separated from user actions and visualisation, it can typically be tested separately, and automatically, while the user interaction is kept simple and less prone to error. As the controller is typically only given *events*, and not interacted with directly by the user, this also tends to be testable.

8

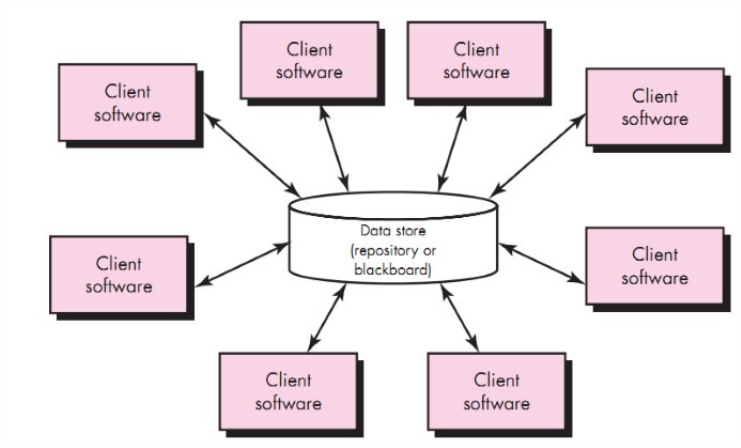
Client-server pattern



In the client-server pattern, the server is the central component, which contains the logic of the application. There are multiple clients, connected to the server over the internet, which rely on the server to do the primary computation. Some clients can still have complex logic of their own, but they do not interact with each other.

9

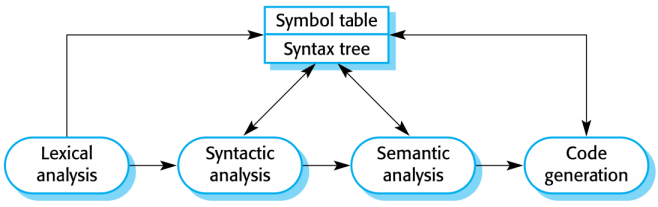
Data-centered pattern



In the data-centered pattern, a database or repository is the central component. Like in the client-server pattern, the other compnents only interact with the data, not with each other. These could be very different components; for instance an administrative component which is allowed to do updates, or a reports component which only reads the data.

10

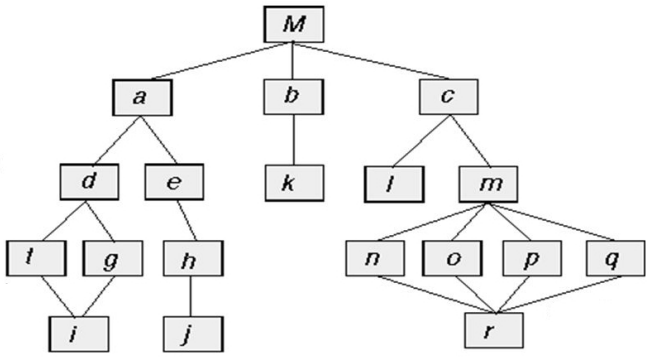
Pipe-and-filter pattern



A very different kind of pattern, pipe-and-filter is used for manipulating data. A typical example (also demonstrated in the picture above) is a compiler, which starts with a user-written program and manipulates it in several steps to turn into machine code.

11

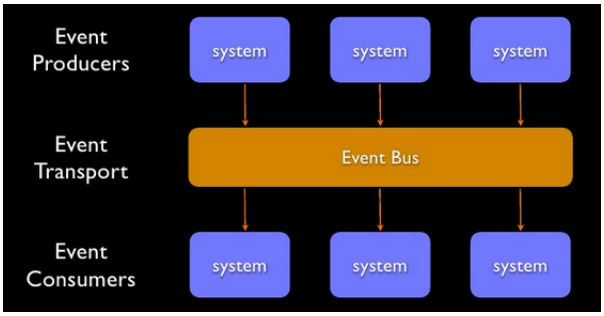
Call-and-return pattern



A very traditional pattern is call-and-return, where execution starts in a main function, which calls helper functions, each of which call functions to do a smaller part of the functionality. Put differently, there is a main program, which calls several sub-programs, each of which have sub-programs have their own.

12

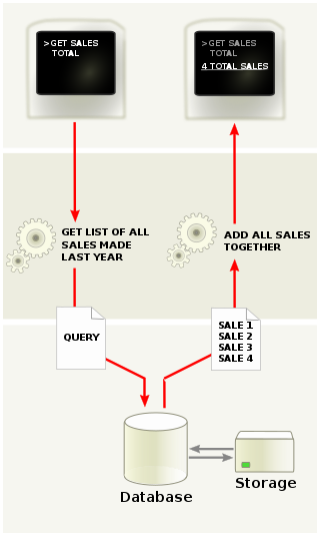
Event-driven pattern



If it works for your application, an event-driven architecture can storngly support separation of concerns. In this architecture, the central component is the *event bus*, but this is a very *small* components. The only function of the event bus is that other components can either inform the bus that they want to listen for a specific kind of event, of that a specific event happened. In the latter case, the event is passed on to the listening component, which handles it. Consequently, the main components of the program are not aware of each other and do not directly talk to each other. We will talk a bit more about this pattern in the next lecture, since it comes with specific implementation patterns as well.

13

Three-layer pattern



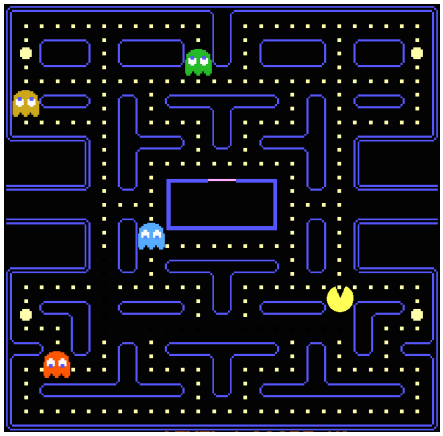
In the three-layer pattern, the program is split up into three levels, which can also be thought of as abstraction layers. The top layer is the *presentation* layer, which interacts with the user. The middle layer is the *application* layer, which handles the program logic. The bottom layer is the *data* layer, which manages the database or other representation of the program's information.

Similarly, the n-layer pattern has n layers, where n can be any number ≥ 2. The defining feature is that each layer can only interact with (do calls on) the layers directly above and below it. For instance, in the three-layer pattern, the presentation layer does not talk to the data layer.

Closely related, the *n-tier* architecture is an n-layer architecture where each layer is on a different device, and can be changed and scaled by different people, and without reference to the other tiers.

14

Challenge: design an architecture



Suppose we want to make a multiplayer Pacman game. What would be the possible architectures, and how would each choice affect the development?

Client-server: An obvious choice for a multiplayer game. The server would handle the game board and all the movements, while the clients would mostly interact with the user: displaying the board, accepting input, etcetera. However, this pattern does not fully define such choices, since the clients could be given more or less responsibilities. For example, the clients could keep track of each player's number of lives, since this does not directly affect the other players, so does not need to be handled by the server. Alternatively, the server could handle this too, and thereby prevent cheating by players who can access their clients' code (but not the server).

Model-View-Controller: In this pattern, the choices would be more clearly defined: the model is responsible for all program logic, after all. Each user would be presented with a *view*, the events of their actions would be sent to a specific *controller*, and the controller would translate this to commands to the central *model*. Of course, this could well be combined with a client-server infrastructure, where the model is handled by the server, and the view and controller by the client. (There is no limitation that we can only use one architecture – many are very combinable!)

Data-centered: The central data would be the game board, which keeps track of all the ghosts, players, edible objects and other features the game might define. The ghosts could query the game board for the nearest player to move in its location, while the player modules would query the whole board to display to the player, and – depending on how the data board is designed, either pass on their new location, or just a movement direction.

Call-and-return: This would work well with a program tick. Every second, the main function calls different subprograms for calculating all the moves, checking for collisions, handling deaths, and printing the screen updates. Each of those different routines could be split up into separate sub-routines too, for instance having different functions for different kinds of collisions (e.g., player-ghost, player-edible-ghost, player-player, player-food).

Event-driven: Aside from handling the events of "user pressed a key", in this architecture we could define many more events. For example, the movement of each player or ghost would give off an event "X moved to place Y", which other objects could list for. For example, if the "game view" object receives such an event, it could print the new status; other objects could listen for this event, check if they are at the same location, and if so, throw a "collision" event. In this particular example, some decisions still need to be taken on how to handle the collisions, since a naive implementation would have the risk of two objects handling the same event in incompatible ways; but there are good solutions here.

Three-layer: In this pattern, the data layer would have the game board, the application layer would have the program's logic (movements, collisions and lives), and the presentation layer would be used to show the game's status to the users, and receive user input. The user layer could *not* directly query the status of the game board, which would make this a more indirect approach.

Pipe-and-filter: For this application, sequential data transformation does not seem application, so I do not believe this would be suitable. (Though if you see a good way, I would love to be corrected!)

All these choices would give (subtly or less subtly) different kinds of implementations. Hence, it is worth having a good discussion at the start of the project on what architectures seem the most suitable for your application.

15

A warning

Avoid too much up-front design!

While it might be necessary to choose the architecture early, you should not go too far beyond that when developing agilely. You will discover design ideas as you develop; leave space to make these choices as they arise. Even the architecture can be changed if needed. James Shore has the following quote in his book *The Art of Agile Development*:

> *In my experience, breakthroughs in architecture happen every few months. (This estimate will vary widely depending on your team members and code quality.) Refactoring to support the breakthrough can take several weeks or longer because of the amount of duplication involved. Although changes to your architecture may be tedious, they usually aren't difficult once you've identified the new architectural pattern. Start by trying out the new pattern in just one part of your design. Let it sit for a while—a week or two—to make sure the change works well in practice. Once you're sure that it does, bring the rest of the system into compliance with the new structure. Refactor each class that you touch as you perform your everyday work and use some of your slack in each iteration to fix other classes.*

In fact, James Shore recommends *not* choosing the architecture up front, but using incremental design (which we will discuss later in this lecture) to discover the best architecture. However, this is arguably a method for XP, not so much Scrum. In this course, we unfortunately do not have the time to handle architecture breakthroughs, so we do recommend *some* up-front choices in this regard.

3. Technical debt

16

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

This is a graph from the *Software Improvement Group*, who have done some research into measuring code quality automatically, and the effect of code quality on development speed. As you can see, the impact of ignoring code quality can be great.

This is also very noticeable in Giphouse, where sometimes customers return with extensions on the same project for several years. In the past, we have had to tell a client that we could no longer develop their project, because the code had become impossible to maintain – let alone further expand – over several iterations of Giphouse projects. Of course, we want to avoid this, so we do ask you all to please pay attention to the advice in this lecture!

17

Technical debt

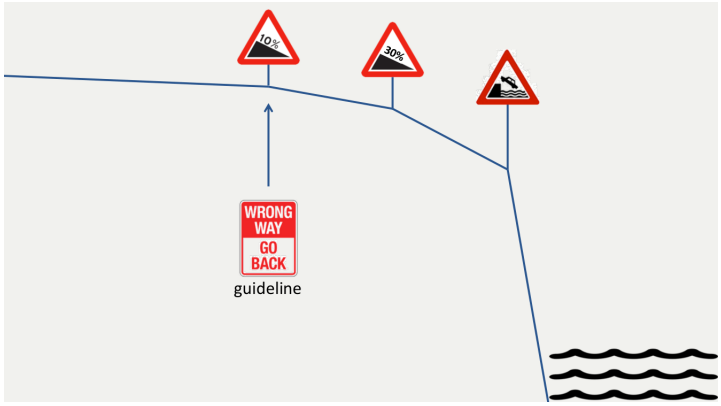
"With borrowed money, you can do something sooner than you might otherwise, but then until you pay back that money you'll be paying interest. I thought borrowing money was a good idea, I thought that rushing software out the door to get some experience with it was a good idea, but that of course, you 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

The phrase *technical debt* is often used to describe code that has become hard to maintain, but not that many people seem to know the initial quote. What Ward Cunninham indicates here is that it is sometimes necessary or useful to write bad code – maybe copy some code, put in a hack here or there or some uncomfortable dependencies... perhaps you need to do this because the sprint ends on the next day and you still want to have *something* to show to your client, or perhaps another team member depends on being able to call your back-end code to be able to test the front-end.

You can do this, but you should be aware that you *borrowed* something. And until you repay that debt (read: fix the code to not be hacky) you're paying interest, in that your code is harder to understand, your development speed is going down, and you might be spending more time fixing bugs. What is worse, if no time is allocated to repay the debt, paying that interest may *increase* the debt, as working with bad code often introduces new hacks and dependencies. And when that happens, your code may pass the tipping point where maintenance eventually becomes impossible.

18



When code only has a few bad design issues, it is still fixable – you should actually allocate some time to repay that debt, but you can go back. But once it reaches a certain point, maintaining it becomes so painful that the whole project gets out of hand; and while many companies do still continue with such code, it might actually be better to restart from scratch and get a proper design.

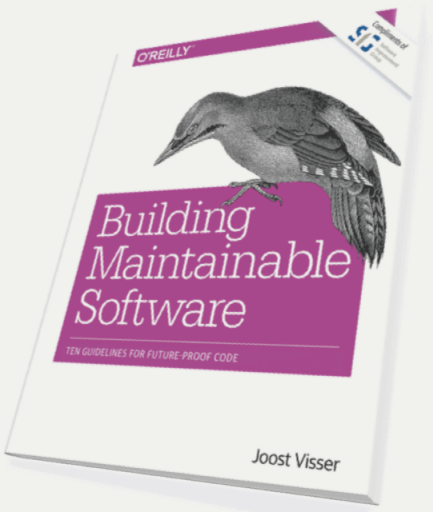
So, when you incur technical debt, put its repayment as a high priority issue on the project backlog. This should ideally be planned for the next sprint, to avoid incurring further debt and losing too much time on dealing with it.

19

If you want to learn more, you can find it in the following book by Joost Visser from the Software Improvement Group:

“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]



4. Coupling and Cohesion

20

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?

4.1 Coupling

21

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

22

Types of coupling

From high coupling to low coupling:

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

The styles of coupling at the top are bad and should be avoided. The styles at the bottom are good and should be aimed for. The styles in the middle are okay, but it is often possible to improve the code by noticing that you have these kinds of coupling, and considering if it is really needed.

23

Types of coupling

From high coupling to low coupling:

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

24

Content coupling

In content coupling, one module relies on the internal workings of another module.

```
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);
}
```

In the example above, the function addUserProps loads a player into a variable, and then manipulates an internal variable. The function queryUserProperties needs the player to be loaded, so it abuses addUserProps: by calling this, the function is loaded into _myUser, and appending the empty array has no effect anyway! The problem is that this relies on the exact implementation of addUserProps.

The proper solution is to separate the "load a player" functionality into its own function. While this may seem obvious in this example, this sort of coupling happens surprisingly often in real-life software, where the core functionality that we want to separate out is mixed in with other functionalities, and therefore harder to split off.

25

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;
  }
}
```

Another example that is often seen is when one class directly modifies a data structure held by a different object. Class B here "knows" that it is manipulating the internal data of class A; but class A cannot know what other objects are doing with it, and redesigns to A that do not take B into account might unexpectedly break things.

26

Types of coupling

From high coupling to low coupling:

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

27

Common coupling

We move on to *common coupling.* This happens when two or more modules share some information by using global data.

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

An example of common coupling that is *not* bad is a settings module. This module contains global variables that a variety of other classes interact with. Arguably it would be better to access this module using functions (e.g., if (Settings.screenSize() == Settings.BIGSCREEN) rather than variables, but this does not have wide-scale effects.)

```
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);
}
```

This is a worse example. Here, two classes are given a pointer to the same data. They are not necessarily aware of each other; certainly future developers of class A might not know that class B shares the same data, and therefore accidentally do something that messes up the functionality of B.

28

Common coupling in Discworld

Examples that fit on a slide are typically rather artificial, so let us consider a *concept* that does the same thing. This happens in Discworld, where the game offers a kind of find-and-deliver mission to players.

Job market in Ankh-Morpork

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

Clothing shop in Djelibeybi

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

The global data here is the availability of red dresses in the game world. Both the job market and the clothing shop interact with this data. The design of the job market expects the dresses to be available, but moderately hard to get. The clothing shop, however, does not know about the job market; and a coder interested only in the city of Djelibeybi might replace the red dresses by more culturally appropriate blue dresses – and hence make the mission in Ankh-Morpork impossible. Alternatively, as the dresses exist and are available, different

clothing shops may also choose to sell red dresses, therefore making the the mission much easier.

How does one fix such a thing? The trick is to realise already when designing the job market that it will introduce common coupling – a dependency that is undesirable. Is it truly necessary to have this particular mission? Or could the job market, perhaps, have a list of accommodating stores, and *give them new items* at the moment the mission is generated? In this way, the dependency is directly between the job market and the shops (or perhaps between both parties and a separate "find-and-retrieve mission" handler), and will not surprise any developers.

29

Types of coupling

From high coupling to low coupling:

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

30

Control coupling

Control coupling happens when one module controls the flow of another.

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

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

Here, depending on the flag, different parts of the function `a` are executed. This choice directly depends on the flag; not on calculations done on it by `a`.

In general, control coupling is not so bad, and can even be good; it is among the middle kinds of coupling. How good or bad it is depends on how it is done. It is bad if a component needs to be aware of the internal structure and logic of another module to use it. It is kind of bad

if the function could easily be split into a true case and a false case. It can be good if the parameter actually avoids code duplication and allows for reuse of functionality in a natural way.

31

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;
}
```

This is an example of good control coupling: the flag has a clear name and function, and it only affects one line of control. It would not make sense to split this function into a version for true and a version for false, as that would require a lot of code copying.

32

Control coupling

```
void printReport(boolean manager) {
  string text;   if (manager) {
    ⟦set text to values for management reports⟧
  }
  else {
    ⟦set text to values for director reports⟧
  }
  sendToPrinter(text);
}
```

On the other hand, this is pretty bad. The parameter largely decides what the function does, and it would make a lot more sense to split the function into a printManagerReturn and a printDirectorReport function – after all, these two parts do not share much functionality.

33

Control coupling

```
void handleError(int code) {
  if (code == 1) handleItemTooLargeError();
  if (code == 2) handleItemTooHeavyError();
```

```
  if (code == 3) handleItemStolenError();
  if (code == 0) {
    ⟦handle correct movement of the item⟧
  }
```

This kind of control coupling is also bad. To call this function with the right code requires knowledge of the internal functionality: if the error codes change, both the functions that generate the error codes and this function need to change. This would actually be very easy to change by using global constants or an Error clas, which would make the code more understandable, and reduces the risk of an incorrect error code being returned by some code.

34

Types of coupling

From high coupling to low coupling:

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

35

Stamp coupling

Stamp coupling is when a composite data structure is shared between modules.

This is a pretty good kind of coupling. However, there is one disclaimer: if several functions only use parts of the same data structure, then it may be worth splitting the structure instead.

```
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);
}
```

In this example, the RECTANGLE struct has multiple fields, and is passed around as a whole. However, `calcArea` only uses two of these fields. This is fine, except for two things.

First, in this case there is a danger that a future developer may try to return `r.area` instead, if they do not know how the function is used. After all, if you have access to the full RECTANGLE, you may be tempted to use it, which is clearly not intended here.

A second, though smaller, problem occurs when there are several functions which only use the dimensions of the rectangle. In this case, it would be better to create a new struct DIMENSIONS with only the length and width fields, and let `calcArea` take an argument of type DIMENSIONS instead. This is still control coupling, but of the most innocuous kind.

36

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⟧
  }
}
```

Another example, where a full GameElements class is passed on, and only one part is affected. This is not necessarily bad, but it would be better if this functionality was moved into a method of the GameElements class.

37

Types of coupling

From high coupling to low coupling:

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

- message coupling

38

Data coupling

Data coupling is when elementary data is passed between two modules, for example through parameter passing. This is one of the lowest kinds of coupling, and what we should generally aim for.

```
class A {
  int k;

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

39

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);
  ...
}
```

Here we see how the example of stamp coupling could be turned into data coupling. However, there is a disclaimer on this: if some arguments truly belong together – such as dimensions – it is often *better* to couple them into a structure than separate them into their elementary components. Otherwise we risk creating *data clumps*: the same set of arguments which are passed to various functions. This for instance increases the risk of mixing up the order (for example switching length and width in some functions), and makes the code less readable.

40

Types of coupling

From high coupling to low coupling:

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

41

Message coupling

Finally, there is one kind of coupling even more elementary than data coupling: message couple. There, one module sends a message to another without passing any data.

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

Unlikely data coupling and stamp coupling, message coupling can typically not be achieved for many interactions: we typically pass around some kind of data.

4.2 Cohesion

42

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

43

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.

44

Inheritance cohesion

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

45

Types of cohesion

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

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

- atomic

As before, the kinds of cohesion at the top (coincidental, logical, temporal) are rather bad (*low cohesion*), while the ones at the bottom (sequentail, functional, atomic) are quite good, with the ones in the middle varying.

46

Types of cohesion

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

47

Coincidental cohesion

Coincidental cohesion is when elements in a module are grouped together arbitrarily, with no relationship between them. A typical example of this is a "utilities" kind of class.

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

In a game such as the Discworld game, another example might be the weapon system, where weapons are grouped together by categories such as "dagger", "sword", "axe", and "misc" – with misc functioning as a catch-all"anything that doesn't fit elsewhere". When this is built into the structure of the codebase, it can make it hard to find things, and when connections later appear, they often are not taken out into separate modules. (For example, in the utilities class above, the `average` and `maximum` functions could be taken together into a "NumberUtilities" module, which would at least give logical cohesion.)

Types of cohesion

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

49

Logical cohesion

Logical cohesion occurs when elements in a module are grouped together because they are in some way logically related, although their functionality is very different and they do not share code or data.

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

For humans this is certainly more intuitive than coincidental cohesion, and it is easier to find where to look for certain functionality in the codebase, but it would still be better if such functionality could be in a more appropriate place. For example, in the codebase above, there clearly is some use for polygons, so it is very likely that there are Triangle and Rectangle classes – the functionality to calculate respective area and perimeters would make more sense in these dedicated classes.

50

Types of cohesion

- coincidental
- logical

- temporal
- procedural
- communicational
- sequential
- functional
- atomic

51

Temporal cohesion

Temporal cohesion occurs when elements in a module (typically a method or function) are grouped together because they are used at the same time.

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

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

The init function is an example of temporary cohesion that is not bad. This occurs at the start of an algorithm, where some unrelated things are set up. However, there is not really another way to do it, and (importantly) the separate functionalities are all in their own subfunctions.

A worse example is the `error_recovery` function: again, this function does three different things which need to happen at the time of error recovery, but which otherwise have nothing to do with each other. The difference is that here, they are all handled in the same function. This makes the function unnecessarily long and complex, and makes the functionalities not reusable.

52

Temporal cohesion

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

```
  }
}
```

A particularly bad example of temporal cohesion is given above. The clearly named function `tetris_block_fall` *sounds* like it should handle the falling of a block in the tetris game, but then it also updates the timer – this doesn't belong to the same functionality, but has been placed there because it should always happen at the same time as the falling of the block.

This sort of thing rarely happens when the code is first written, but does often occur when code is edited to add new functionality afterwards. Likely, the person who implemented this function already had a working `tetris_block_fall` function and added a timer; they struggled to find a good place to put the update, and added it among functionality that happens at the same time, perhaps because they wanted to show the block moving before the timer updates.

53

Types of cohesion

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

54

Procedural cohesion

Procedural cohesion happens when elements in a module are grouped together because they are executed sequentially to perform a certain task.

```
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⟧
}
```

Sequential cohesion is not necessarily bad, as the actions are arguably cohesive to achieve a shared goal. However, when there is otherwise no connection between the actions, it would be

better to isolate them into separate subfunctions, which are called by the main functionality. For example, the function above would be significantly improved if it simply consisted of four function calls rather than four (possibly inter-mingled) pieces of code.

55

Types of cohesion

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

56

Communicational cohesion

Communicational cohesion occurs when two elements work on the same input data and/or produce the same output data. This could also be a reason why functionalities are together in a class, as they manipulate the same underlying data.

```
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)
}
```

57

Types of cohesion

- coincidental
- logical
- temporal
- procedural
- communicational

- sequential
- functional
- atomic

58

Sequential cohesion

Sequential cohesion is when 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.

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

Sequential cohesion is one of the good kinds of cohesion, although in many cases it can still be improved by isolating subfunctionalities into a function that takes arguments and/or returns a value (as done in this example by the `check` function).

59

Types of cohesion

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

60

Functional cohesion

Functional cohesion is what we should strive for as an ideal. This is when, in a single component, all the essential elements are combined together for performing a single task, and only those. Ideally, you should be able to recognise functionally cohesive modules by a name which exactly describes their function:

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

We will talk more about functional cohesion in the next lecture.

61

Types of cohesion

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

62

Atomic cohesion

Some sources also mention *atomic cohesion*: a single component that cannot be reduced any further.

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

This is of course not feasible for most functionalities, and functional cohesion is the norm to aim for.

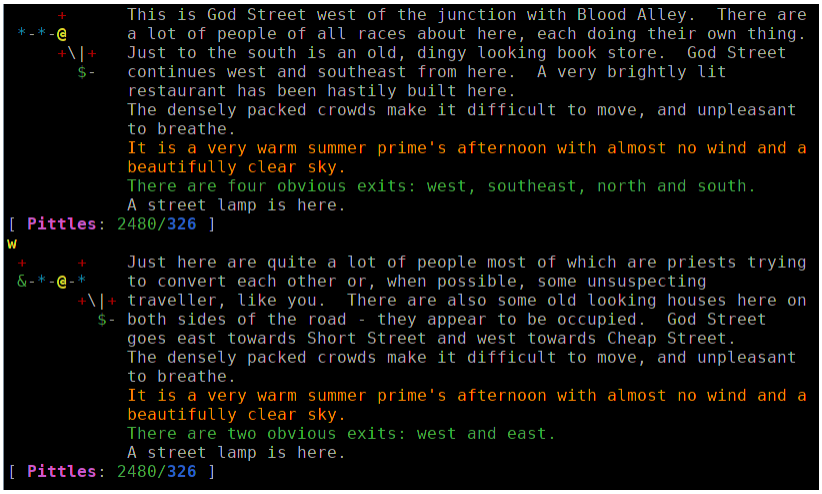
4.3 Coupling and cohesion: an example

63

Challenge: room information in Discworld

As a practical challenge, let us look at some realistic code. This chain of functions was taken from the Discworld game (though slightly adapted to fit on a slide and not require too much additional explanation, and not including recent updates and code improvements :)). This series of calls contains many instances of bad coupling and cohesion – can you recongise them? Even if you cannot directly recognise, for instance, control coupling, it is useful to identify red flags in the code (also known as *code smells*) that suggest something might be wrong.

The code chain is what is executed when a player moves to a different room. Every room has a description, with a line for the weather, exits, and other things in the same room, as well as a little ascii-art map that can be printed to the side or above of the room description. When the player sends the move command, she is moved to the corresponding room, and the description (with all other parts) for that room is printed. In addition, some metadata is sent, which her client can use to for instance keep track of locations where she has been, or print the map in a separate window. To accommodate visually impaired players, instead of an ascii-art map the player can also receive a written map with text like "a brown dog is two rooms to the east".




```
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;
}
```

This function executes the move, and if it is successful, does a look in the room.

```
int room_look() {
  if ( !( interactive( this_object() ) ) )
```

```
    return 0;
  this_object()->ignore_from_history(``look'');
  this_object()->bypass_queue();
  command(``look'');
  return 1;
}
```

This function executes a look in the room by (a) checking if the current object is actually a player (if not, there's no need to do it, since non-player objects won't use the look information); and (b) by using the "look" command to execute a look – but taking extra steps to avoid some of the side effects of executing that command.

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

(This command is rather too long to print on a slide.)

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

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

If it is not dark, this function passes on control to the `lookmap_text` function and takes over its return value.

```
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);
  }
  return ret;
}
```

In this function, the little map is generated (in ascii art or written text depending on the player's settings), then the room info is sent to the player including that map, and then the text and map are combined based on the requested look map setting.

```
void send_room_info(object pl, string map) {
  [[send metadata "room.info": room & city name]]
```

```
  if (pl->map_setting() == ASCIIMAP) {
    string wmap = lookmap(WRITTENMAP);
    pl->send_metadata("room.map", map);
    pl->send_metadata("room.writmap", wmap);
  } else {
    string asciimap = lookmap(ASCIIMAP);
    pl->send_metadata("room.map", asciimap);
    pl->send_metadata("room.writmap", map);
  }
}
```

This function sends the two kinds of maps, as well as additional information on the room and city, to the player as metadata, so their client can handle it in whatever way they wish.

64

Challenge: room information in Discworld

Your challenge:

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

Some suggestions are given below. These are not exhaustive; there may be other red flags which are not mentioned here.

move_with_look:

Although we do not necessarily see instances of bad coupling or cohesion here, the fact that `return_to_default_position(1);` is called twice is a bit of a red flag, and it might be worth checking the code to see if the 1 parameter is not indicative of control coupling (especially since, in this language, there are no booleans – so 0 operates as false, and any other integer, usually 1, is used as true). Beyond this, the name could be a red flag, since it roughly says "thing 1 and thing 2", which is not a good sign for functional cohesion (as it suggests more than one responsbility). But, if moving and looking very often happens together (which in this game is certainly the case) it would make sense to have a *short* function that does both, provided they are handled through separate function calls – which they are.

Note that the use of the MOVE_OK constant is actually a positive example: if the function used a constant integer here instead, it would be an instance of control coupling, but the clearly named MOVE_OK constant makes it understandable what happens here. (However, an argument could be made that if "move" has a chance to fail, as suggested by the return value, then it might not be correctly named – especially since a developer who takes the name at face value might ignore the return value and simply assume that the move was successful. But, this is not a problem with coupling or cohesion, and rather has to do with naming choices.)

room_look:

Here we actually see content coupling. The function uses knowledge of the internal working of the "look" command to bypass the specific side effects. This could be avoided by having the core functionality of the "look" command in a separate function, that could be called here.

The check at the start whether the current object is interactive (that is, if it is a player) could also be a red flag. It seems likely that in a language with objects and inheritance, this could be more appropriately handled by only implementing this function is the player object. However, to know if this is actually an issue we would have to know more about the surrounding code, which is beyond the scope of this exercise.

long_lookmap:

Looking at this function should raise some red flags – it seems likely that this function initially did something more with the dark flag than just use it to abort right away, but was changed at some point in the past. This is often how strange or confusing code arises in practice. Another possibility is that someone figured we might eventually wish to do *more* with the darkness status, and added a currently-mostly-unused argument to the function. This is premature generalisation: adding complexity that is not currently needed.

In the language of coupling and cohesion, this is a clear case of control coupling that could easily be avoided.

lookmap_text:

A red flag we might notice here: why is the `lookmap_type` passed as an argument, wile the `map_setting` is just directly requested from the player? Couldn't both be recovered from the settings here? There might be reasons for this in other code that is not shown, though.

More importantly: what is happening with the room info, and why is that happening in a function whose name suggests that it will only compute some text, and not have side effects? This looks suspiciously like temporal cohesion: sending the room information has nothing to do with the actual task of calculating the text-with-map, but is done there because it needs to be done pretty much every time that this calculation is done. It is not *exactly* temporal cohesion, because the room info function does use a variable that is calculated in the lookmap_text function, but it is not really sequential cohesion either.

However we call it, the call to send_room_info clearly does not belong in this place. So why did it end up here? This is a question of efficiency: computing the map takes a relatively long time, and moving the call to send_room_info into the look command (where it would be more appropriate) would mean doing this calculation twice. This also makes it challenging to refactor. There are good ways to do it, but these take some time, and in development people sometimes take shortcuts. In this case, the price is that the lookmap_text function is less reusable, since it comes with an unexpected side effect.

5. General advice

65

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⟧
}

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⟧
}
```

These functions are not too cohesive: while their actions all contribute to a common goal, they have several parts which do separate things, which are not related to each other. However, we can obtain functional cohesion by separating out these subfunctionalities into their own functions:

```
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(address);
  class db = open_db_connection();
  db.store_address(user_index, address);
}
```

66

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

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

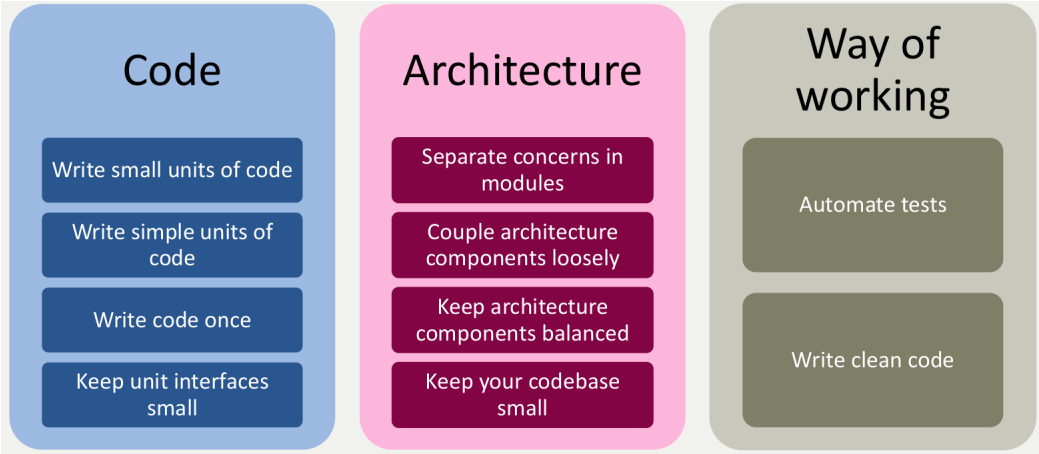
```
void tetris_block_fall(int block_id) {
  update_timer();
  move_block_one_down(block_id);
  if (block_has_landed(block_id)) {
    pause(100);
    handle_landing(block_id);
  }
}
```

However: having the separate functions helps with restructuring!

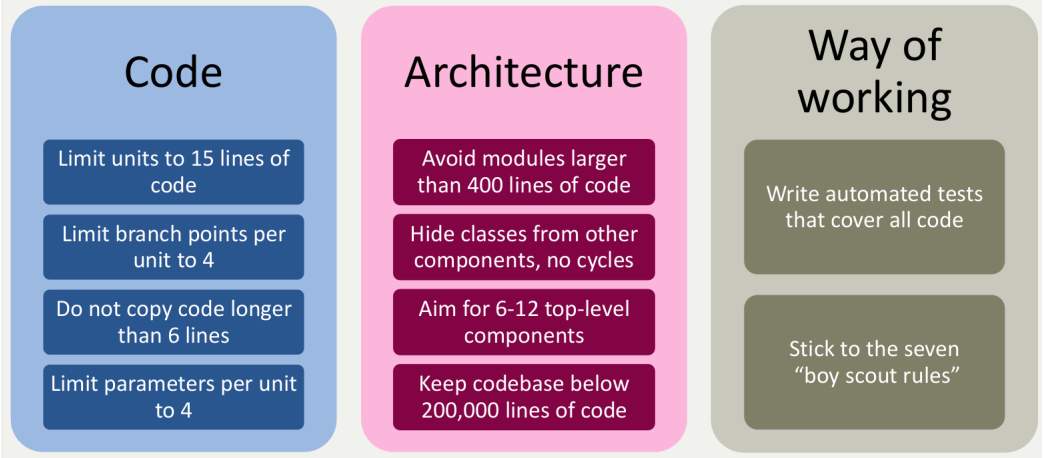
67

Basic Guidelines

The Software Improvement Group suggests the following guidelines to keep your code clean, which – when handled in a natural way – are likely to lead to low coupling and high cohesion:



Concretely, the software improvement group translates this into very specific guidelines:



– Software Improvement Group

Of course, there is a difference between programming languages in, for instance, the expressiveness of 15 lines of code. The importance is not so much this specific number, but the

existence of a reasonable number, which is agreed upup by the team and enforced in *most* of the codebase. There may be some exceptions where it makes sense to, say, have a longer module or a few extra parameters, but it should not be common in the code.

68

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)

69

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) {
  ...
}
```

Note that a function like this is 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. Then it is worth checking: should some of these parameters be combined into a single structure, e.g., `struct attack_data`?

70

Duplicate code

The Software Improvement Group suggests not copying more than 6 lines of code. This is about copying code within the same project; copying too much code from StackOverflow or other projects should also be avoided for different reasons – especially when it is code that you do not fully understand – but the issue here is code *duplication*, where the same lines of code appear in multiple places, which means that if they need to be changed, this needs to be done in multiple places, and you risk inconsistencies when some are changed and some are not.

I actually use a different guideline:

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! (For example by moving the shared code into a global function (with parameters) or inheritable class.) Do watch out for introducing bad kinds of coupling, though!

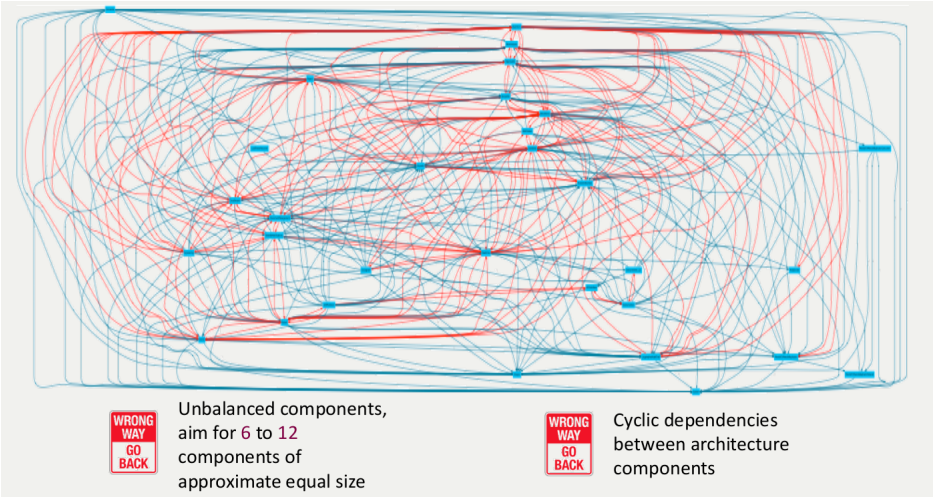
The idea of the pushup rule is: sometimes you do need to copy code, because it is genuinely the best thing for the project, or changing the code to avoid the copying would require a large refactoring that is simply not possible in the near future / worth it for 10 lines of copying. But when you do need to copy code, there should be a price to pay. The pushup rule assumes that the programmer is comfortable with 10 pushups, and does not like doing 20, but can if they must (the pushups need not be done all in one go). Hence, if you are genuinely willing to do 100 pushups to be able to copy those 23 lines of code rather than changing the codebase to increase reusability, then you may. (But you do actually need to do them!)

(Of course, this is university and not the military – choose for yourself if you want to try sticking to this rule.)

71

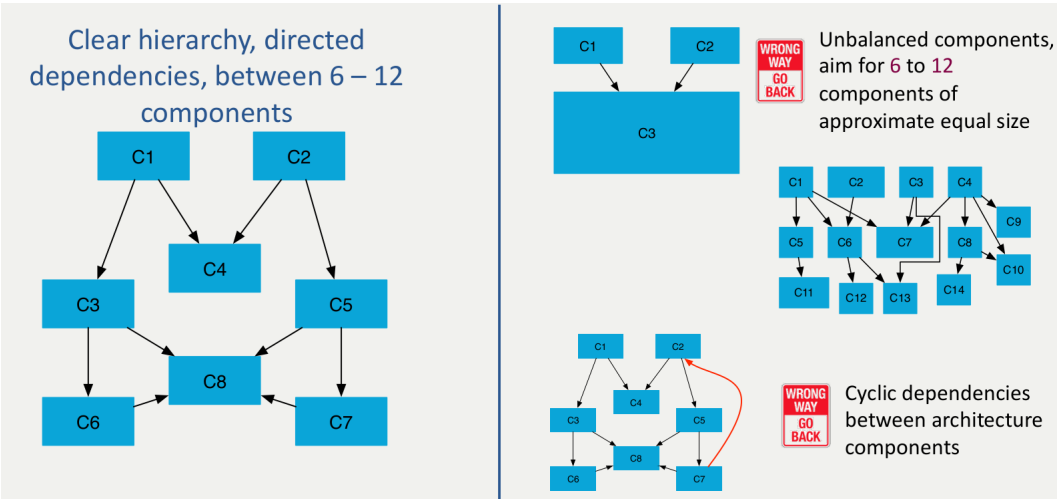
Keep your architecture manageable.

- 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



72

Keep your architecture manageable.



73

Test all non-trivial code automatically

When writing code, you may have occasionally tested it using a manual function like this:

```
void test_validation() {
  do {
    string num = input("Type account no:  ");
    println("Result:  " + validate_account(num));
  }
}
```

This is nice for one time, where you come up with a few numbers and check if the result is as you expect, but if you change the code at one point, you'll have to run the test again and

do the creative work of coming up with good account numbers and seeing if the result is as desired again. If you regularly edit that code, chances are you will slack on doing the tests every time.

This is where automatic testing comes in. You can *save* your tests, and have them automatically rerun whenever your code has changed. This way, if a problem comes up, you are immediately notified, and you have to do the creative work of thinking of good tests only once.

```
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");
    }
}
```

Testing code that involves user interaction or code with side effects on databases is more tricky. While there are unit testing frameworks that support this, they have a learning curve, and some can be quite impractical to use.

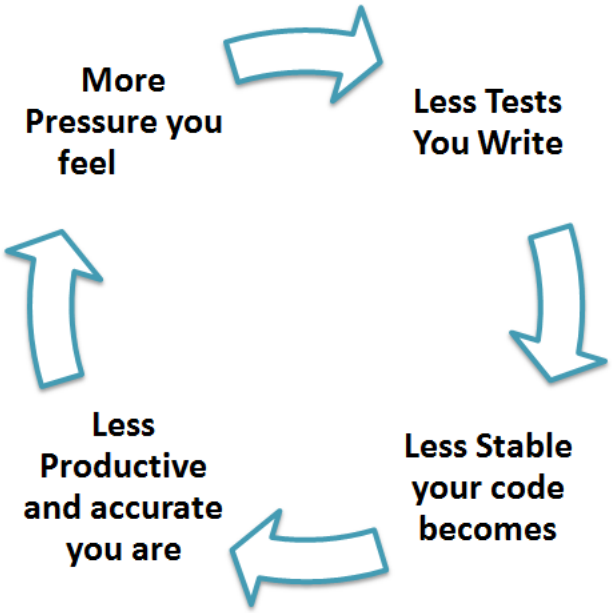
In the next lecture we will discuss some ways to design your code so as to minimise this issue, and make the vast majority of your logic testable. In a very short summary: keep the logic inside functions and/or classes that do not directly interact with the outside world, and test those classes.

For now, if you are unsure how to test something automatically, *write down the test and save it inside your project repository.* Design it as you would if the tester was a robot, only following instructions – so not "fill in some real names" but rather "fill in 'Hans Smit', 'Tessa d'Uilenberg' and 'Jan-Willem Zeidersteen van Assen' ". You can reuse those manual tests later, or turn them into automatic tests once you have an appropriate framework / implementation set up.

74

Test all non-trivial code automatically

Often, when there is little time, people end up omitting the testing. But beware: this is likely to end up costing *more* time, and at more inconvenient moments. Time spent writing tests now saves a greater amount of time debugging later!



75

Boy Scout Rule

Leave the campground cleaner than you found it.

When you edit some code, clean it up as you go along. This doesn't mean doing a major refactoring every time you make a tiny change in a module, but simple changes to make the code more understandable are good to take along as you work.

- **Leave no unit level code smells behind.**

  When you see "red flags" in a function/method you are editing, check if it is actually doing something bad, and if so, if you can fix it; if not, can you make it more clear what is happening so that it doesn't look like a red flag?

- **Leave no bad comments behind.**

  Very often, code gets updated but the comments along with the code do not. So, there are comments which do not properly describe the code. Take them out. (In fact, for this reason it can often be better to *not* write a lot of comments, but rather to aim for producing self-documenting code: code with appropriate naming that does not need much explanation.)

  Other examples of bad comments are "I have no clue what is happening here but the printer starts jamming if you take it out". If you can find what is happening, either improve the code or the comment. :)

- **Leave no code in comments behind.**

  Often when code is replaced, the old code is left behind in comments. This makes the code less readable, and as the code is further changed that old code is not going to be usable anymore even if uncommented. Moreover, we have version control! Why keep the outcommented stuff in there, when we can always get it back using some simple git commands?

  When you remove the outcommented code, make this a separate commit, though – do not do it together with your other changes. By making this a separate commit, it is easy to recover the code on its own.

- **Leave no dead code behind.**

  Very similarly, if code is not reachable though the program flow (for example because it occurs after a return statement), it is clearly not used. Remove it (but indicate clearly where this happened in your git commit message).

- **Leave no long identifier names behind.**

  Identifier names should be descriptive, but not insanely long. In particular names that express multiple responsibilities (such as `generateConsoleAnnotationScriptAndStylesheet`) or contain too many technical terms (such as `GlobalProjectNamingStrategyConfiguration`) violate this rule.

- **Leave no magic constants behind.**

  Magic constants are literal values that are used in the code without explanation. For example, `cost *= 0.90;` – what is this multiplication about? What is the meaning of 0.90? Instead, *name* your literals using constants or macros; `cost *= DISCOUNTFACTOR;` is much clearer.

- **Leave no badly handled exceptions behind.**

  Exceptions should be handled; not caught-and-ignored, nor just shown to the user in all their technical detail without good explanation.

6. Agile design

76

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!

77

Design in eXtreme Programming

The following advice comes from James Shore's book *The Art of Agile Development*. See for instance `http://www.jamesshore.com/Agile-Book/simple_design.html` and `http://www.jamesshore.com/In-the-News/Evolutionary-Design-Illustrated-Video.html` .

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.

Of course, this is about XP while you are using Scrum, but similar ideas apply.

78

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. This rule states: if, after ten minutes of discussion, you cannot agree on the best choice for design, start to implement both your favourite options. (Use different branches!) Often implementation shows what works and what does not.

79

Incremental design

- The first time you create a design element, be completely specific. That is, only write something that solves the exact problem you want right now, no more.

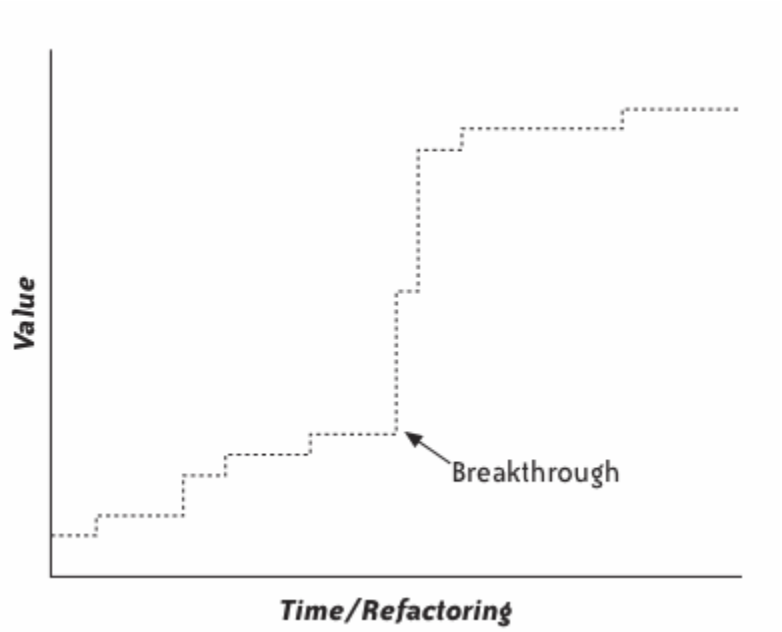
  (This is actually really hard to do: humans tend to think ahead, and try to generalise. But coding for a requirement that isn't there yet is actually also quite a bit harder than coding for the completely specific requirements that we have right now, and this causes unnecessary complications.)

- 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!

80

Incremental design

With incremental design, after making a few changes to the same element, there is often a *breakthrough* moment, where you realise the perfect shape, or a design pattern (see next lecture) that fits what the code needs to do. At this point, the code has been made to fit multiple use cases, and is likely to become significantly more reusable to new use cases that subsequently arise.



81

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

82

Simplicity in agile design

Keep It Simple, Stupid

The system should be:

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

83

Risk-driven design

Positing this rule of "do *only* what you need right now, nothing more" often leads to protests, such as:

But I already have a strong suspicion of what I will want to do in future iterations and I can see that this is going to be a really big problem...

This is fair: it is a rule from eXtreme Programming, after all, and as such is likely to be a bit extreme for many people. However, there are solutions to this, both from an XP perspective, and from the slightly more planned Scrum perspective.

- **Remove duplication around the risky code.**

  This is arguably the true XP solution to the problem, but also very recommendable in general. The word "duplication" does not refer to copied code (although that could be part of it), but also to the duplication of an *idea*. Consider for instance the case of a project that is written for the Dutch market, when we know the client hopes to expand to an international market in the future. The way numbers, dates and other variables are formatted varies between countries, so we might be inclined to already add a way to check the current locale and put in some checks based on it.

  Don't. That would be premature generalisation.

  Instead, note that the *duplication* here is the *idea* that, for instance, a date is written as YYYY/MM/DD, and the *idea* that the unit for money is eurocents. Neither of these ideas requires a lot of code, but they are used throughout the code. We can remove duplication around this concept by creating Date and Money objects, and instead of manipulating strings and integers, the code performs calls on these objects. If we eventually decide to internationalise, the changes can be made in just these objects. Perhaps this also requires changes to everything that does calls on them, but the compiler (or IDE) can easily find these places.

- **Schedule risky features early on!**

  Arguably more of a Scrum approach, where we do a bit more advance planning: if we can see that an issue is genuinely going to be problematic if we postpone it, it should probably be considered as a high-priority item on the product backlog. This way, the potentially tricky code is implemented before it becomes an issue.

Example: I only need RED/GREEN/BLUE now, but eventually will want colours that depend on user settings.
Bad: class Colour with unused functions setRGB(), setDisplay(), ...
Good: class Colour with options for RED/GREEN/BLUE.

84

Agile and incremental design

See also:

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