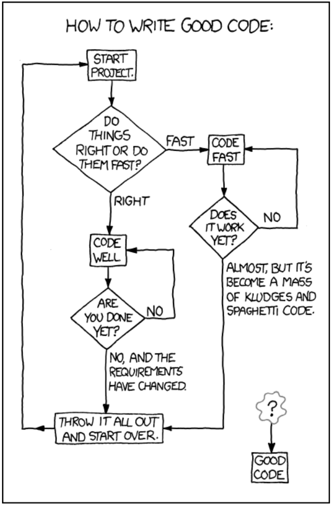
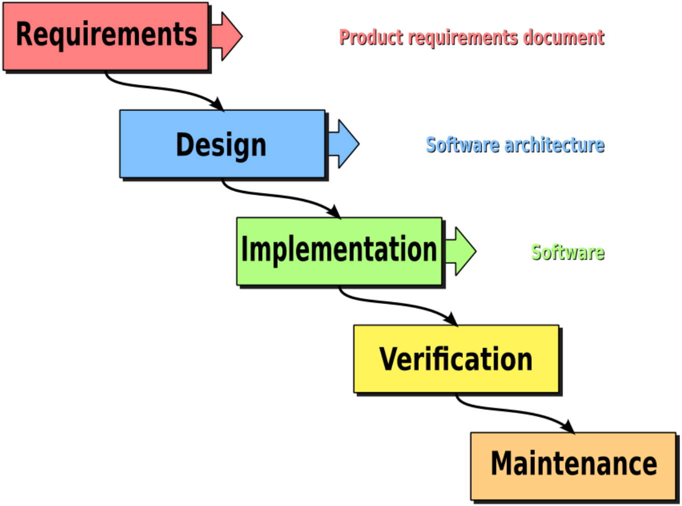
Agile Software Design

19 February, 2020


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`

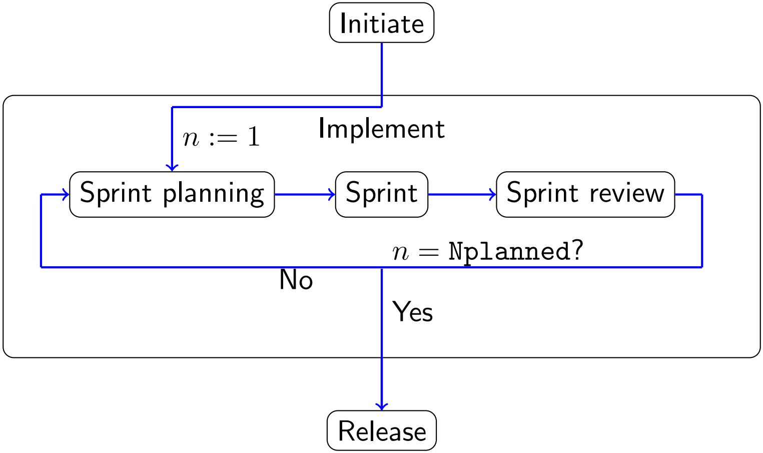
What can "design" mean?



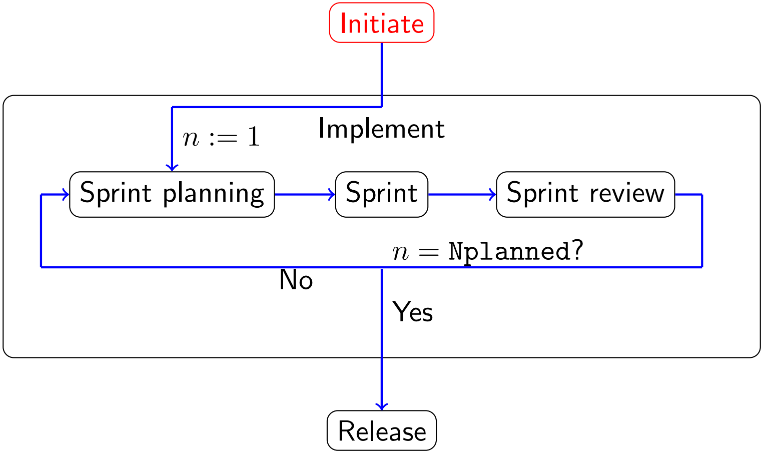
What can “design” mean?

- a process: a phase between *requirements* and *implementation*
- the result of this process, embodied in documents or in code

Design in an agile setting?



Design in an agile setting?



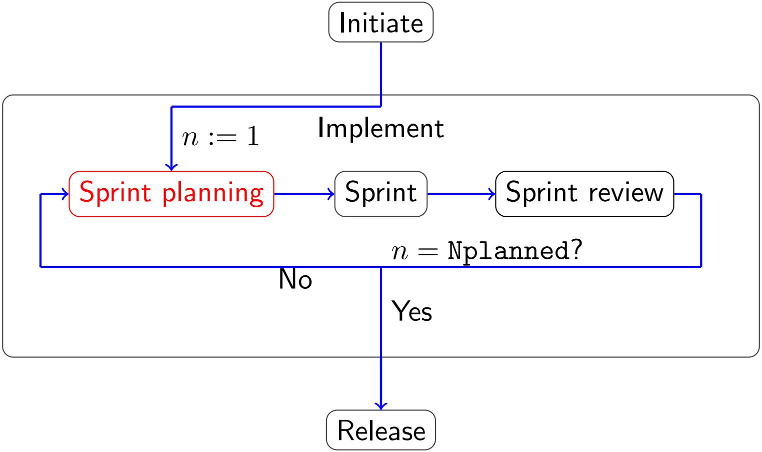
Design in an agile setting?

Scrum Initiation

- create project vision
- create initial product backlog: prioritised requirement list
- create initial user stories
- designate Product Owner and Scrum Master
    - Product Owner: product visionary (often: key stakeholder)
    - Scrum Master: removing blocks, making decisions
- assemble development team (3-9 people, not necessarily just programmers!)
- make agreements on development process (tool use, definition of done, code reviews, etc.)
- create initial time estimates

To estimate times, you need to have some idea of what you're going to do.

Design in an agile setting?



Design in an agile setting?

Sprint Planning

- refine and reprioritise product backlog
- improve relevant user stories
- (re-)estimate times for features in product backlog
- create Sprint backlog of tasks (4–16 hours)
- estimated time may not exceed available time!

What can “design” mean?

- a process: a phase between *requirements* and *implementation*
- the result of this process, embodied in documents or in code

Aspects:

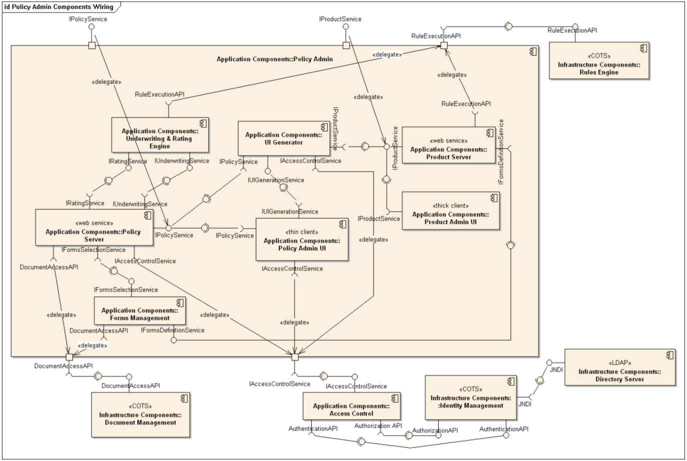
- methodology, modeling languages
- structure of modules and code
- principles and patterns
  - principle: rules you adhere to
  - pattern: a general, reusable solution to a commonly occurring problem
- appropriate use of programming language features
- notions of code quality

What can “design” mean?

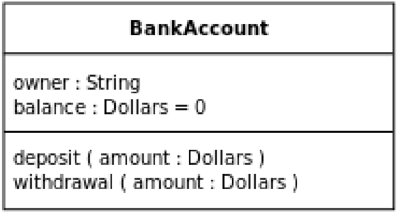
- a process: a phase between *requirements* and *implementation*
- the result of this process, embodied in documents or in code

Aspects:

- methodology, modeling languages
- structure of modules and code
- principles and patterns
  - principle: rules you adhere to
  - pattern: a general, reusable solution to a commonly occurring problem
- appropriate use of programming language features
- notions of code quality



Structure diagram

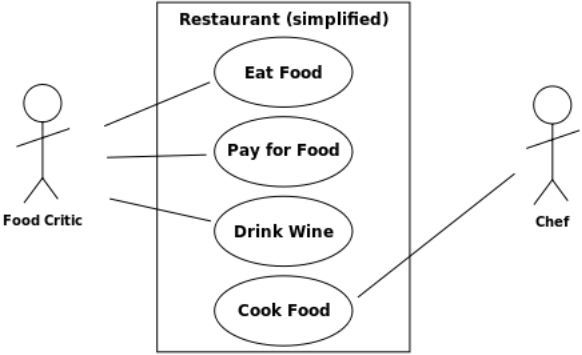


BankAccount

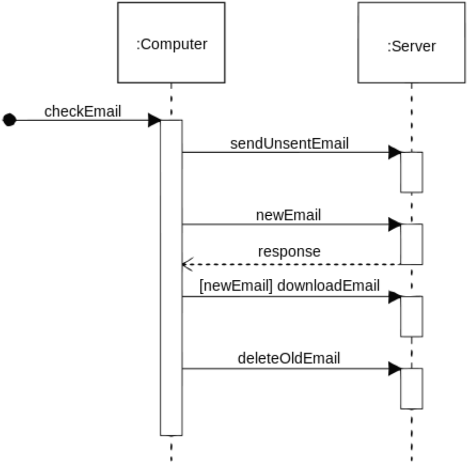
owner : String
balance : Dollars = 0

deposit (amount : Dollars)
withdrawal (amount : Dollars)

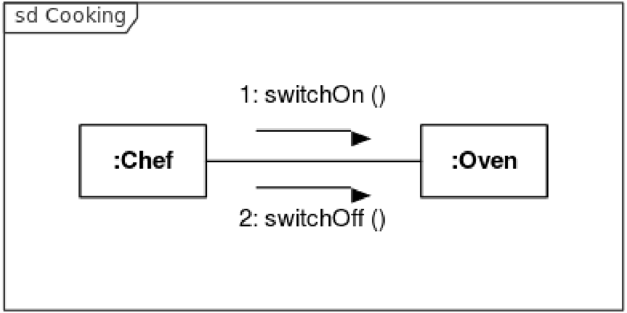
Class diagram



Use case diagram



Sequence diagram



Communication diagram

In an agile setting:

- minimal up-front design
- possibly: use flowcharts, bullet lists or just a textual description
- *can* use UML, but also to document design as given by the current code

What can “design” mean?

- a process: a phase between *requirements* and *implementation*
- the result of this process:
  - decisions about the shape of your product
  - the actual shape of your product

Aspects:

- methodology, modeling languages
- structure of modules and code – this lecture
- principles and patterns – next lecture
  - software design principles: rules you adhere to
  - software design patterns: solutions you use
- appropriate use of programming language features
- notions of code quality – throughout, and in three weeks

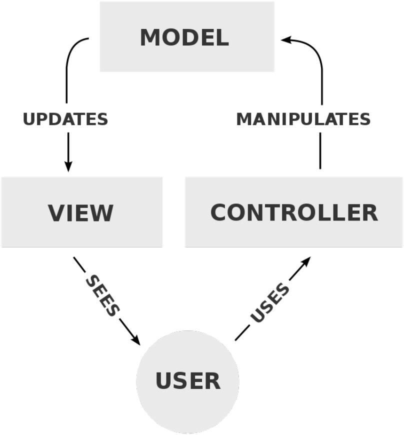
Design considerations: which are priorities?

- Extensibility – easily adding new capabilities
- Modularity – natural split in independent components
- Reusability – being able to reuse parts elsewhere
- Security – able to withstand attacks
- Privacy – able to protect (sensitive) user data
- Performance – performing quickly and/or in low memory
- Usability – easily used by a majority of likely users
- Robustness – able to operate under unpredictable conditions
- Fault-tolerance – able to recover from component failure
- Portability – works across different environments
- Scalability – adapts well to increasing user counts

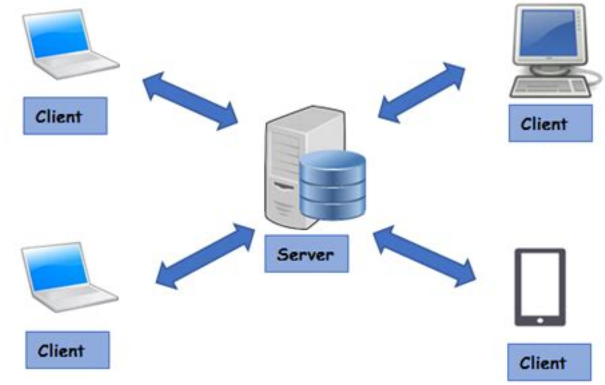
Levels in software design

- Architecture: how the system is overall structured, decomposed and organised into components, and interfaces between them
  - includes decisions like: programming language and platform
  - various architecture patterns
  - change later is hard (but not impossible)
- Module/component level: how the individual modules/classes are structured and operate (Not early!)
- Others: for instance database design

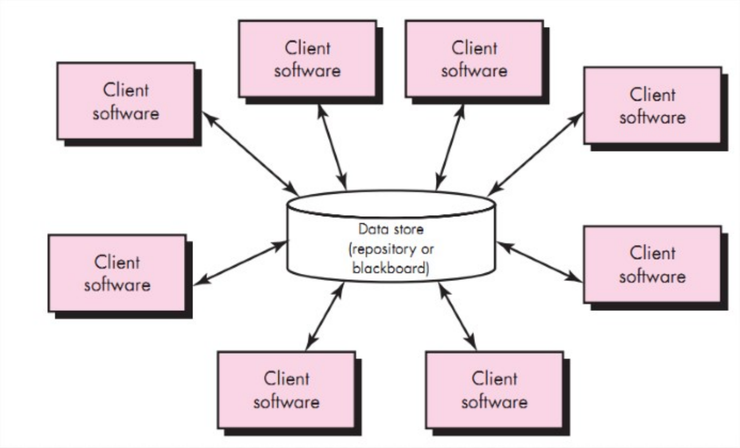
Model-View-Controller pattern



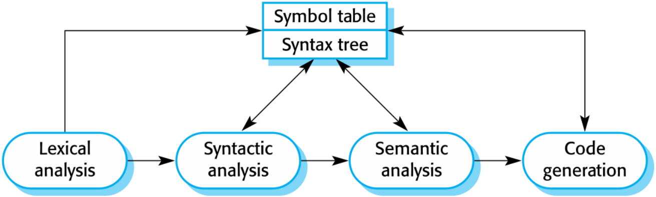
Client-server pattern



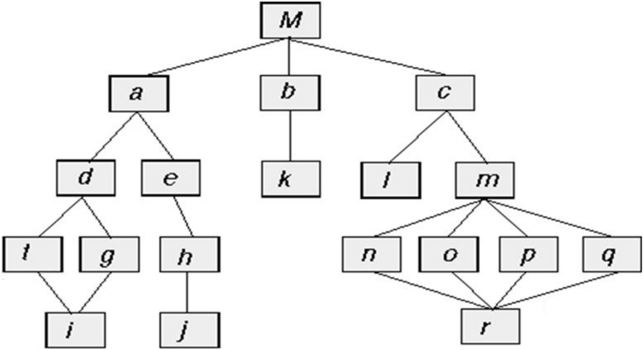
Data-centered pattern



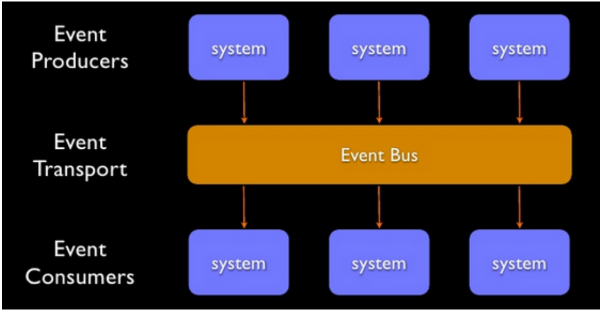
Pipe-and-filter pattern



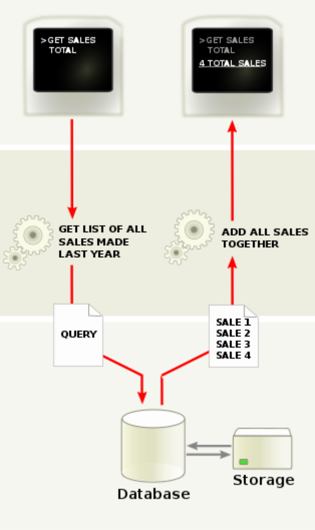
Call-and-return pattern



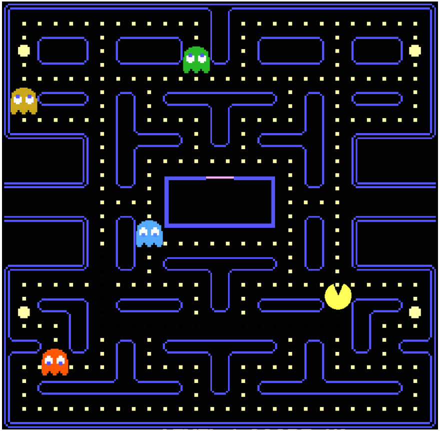
Even-driven pattern



Three-tier pattern



Challenge: design an architecture



A warning

Avoid too much up-front design!

Levels in software design

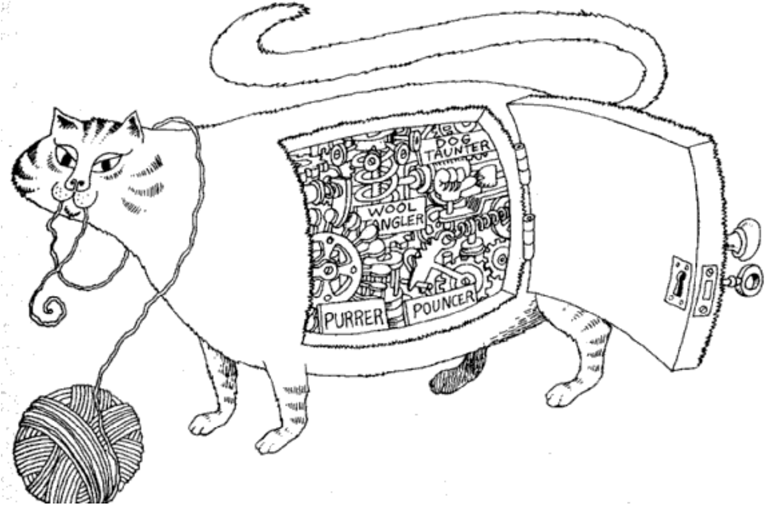
- Architecture: how the system is overall structured, decomposed and organised into components, and interfaces between them
- Module/component level: how the individual modules/classes are structured and operate
- Others: for instance database design

Modular design

- aims: each module (or, class) can be implemented separately; change to one has minimal impact on others
- this cannot be achieved by simply chopping code into pieces
- some criteria are needed

- information hiding, loose coupling, high cohesion
- supporting principles: open-closed principle, substitution principle, . . .

Information hiding



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
- high coupling means greater cost to making changes
- also 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

```
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

```
class A {
  int a;
  void f() { print(a); }
  void g(int b) { a = 2 * b; }
}
```

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);
}
```

Types of coupling

From high coupling to low coupling:

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

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

```
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

```
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.length = 6;
  rect.color = RED;
  rect.area = calcArea(rect);
  ...
}
```

Types of coupling

From high coupling to low coupling:

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

Message coupling

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

Coupling in object-oriented programming

- inheritance coupling
- interaction coupling

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
- Which is the “better” reason?

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

```
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

Logical cohesion

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

```
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

```
void main(string name) {
  read_data();
  read_user_input();
  generate_insults();
}
```

Types of cohesion

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

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

Communicational cohesion

```
void determine_customer_details(int accountno) {
  // do some work to find the name
  // do some work to find the loan balance
  return 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

```
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

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

Types of cohesion

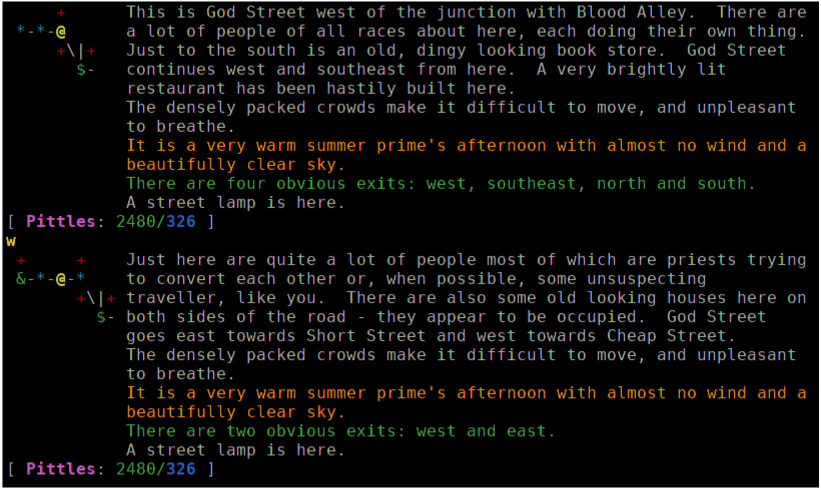
Why code is together within a method/module/class, from worst to best:

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

Atomic cohesion

```
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);
  swwitch(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.

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?

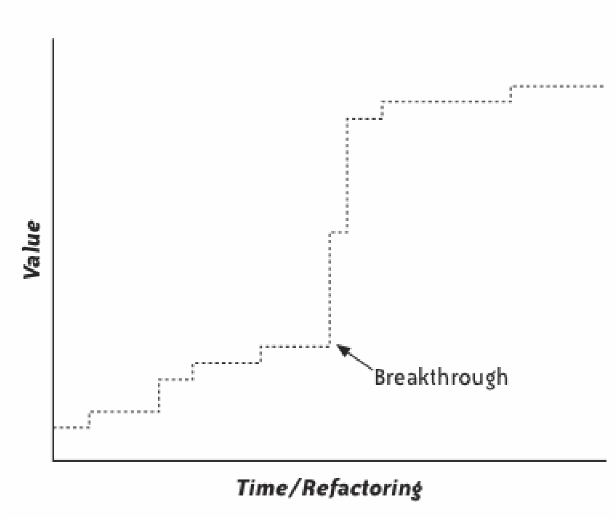
If there is a problem:

- 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

Design and workflow principles to maintain simplicity

- Principle of Least Astonishment
- You Aren't Gonna Need It
- Once and Only Once
- Fail Fast
- Self-Documenting Code
- Isolate Third-Party Components
- Limit Published Interfaces

Risk-driven design

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...

- Remove duplication around the risky code.
- Schedule risky features early on!

Agile and incremental design

See also:

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

Assignment

Tell me, in 300–500 words, what you – personally! – have done to improve the code quality and testability in your product. (Or to maintain an existing high standard.)

Deadline: 18 May.