

Software design

\$5

Steps to working software

1. Design

a) Data, state

b) Behaviour/
Functionality

2. Tests

3. Implementation

Iterate!

The diagram consists of three curved arrows pointing from right to left. The top arrow starts from the right side of the 'Implementation' step and points to the 'Design' step. The middle arrow starts from the right side of the 'Tests' step and points to the 'Design' step. The bottom arrow starts from the right side of the 'Implementation' step and points to the 'Tests' step. The word 'Iterate!' is placed to the right of these arrows.

Think before you code, and write down what you were thinking.

Data + Functionality

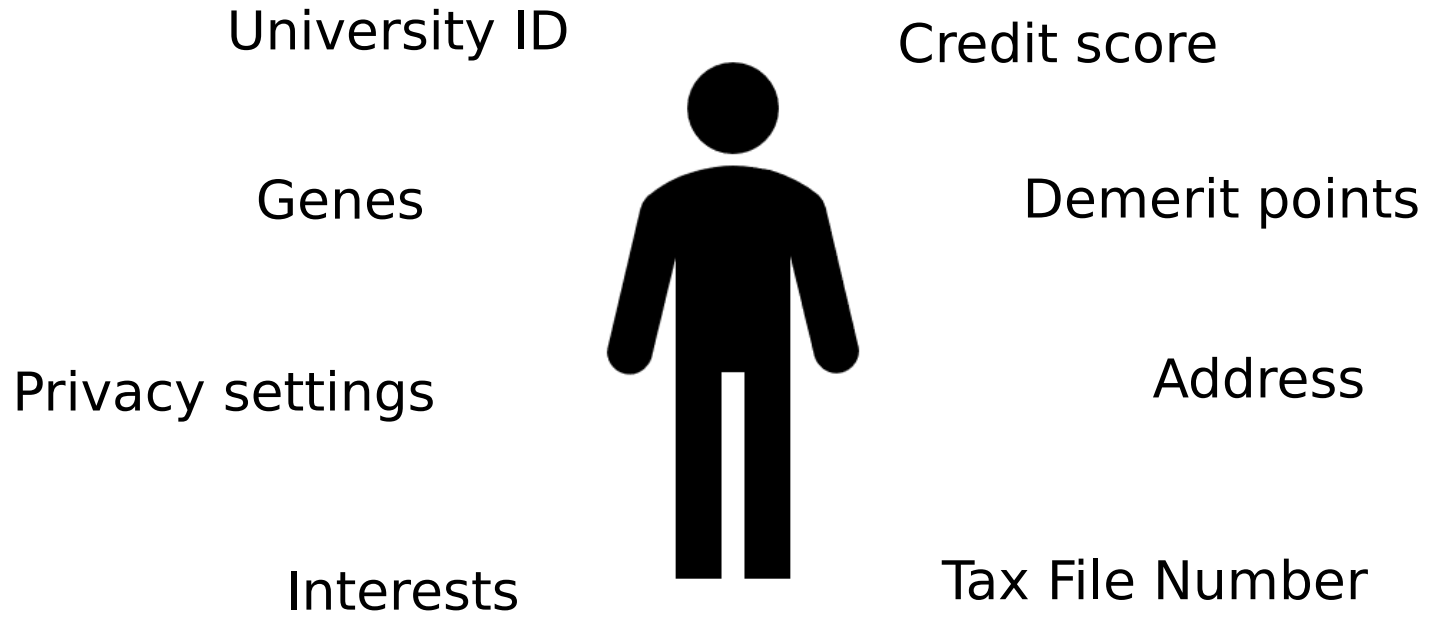
- What distinct things do we care about?
- What aspects of them do we care about?

- How can we interact with those things?
- What can they do?

- Are there any subtype relationships?

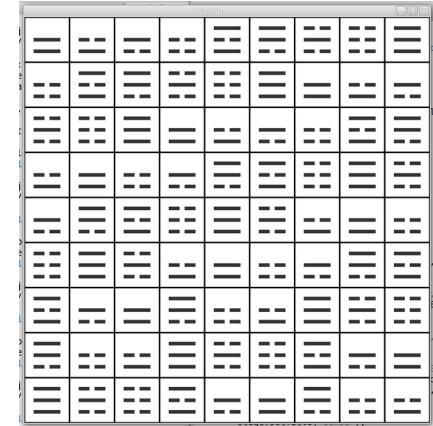
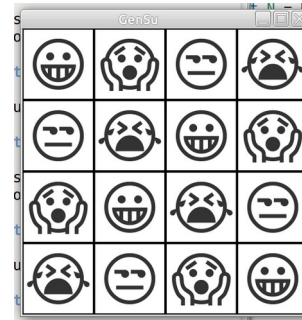
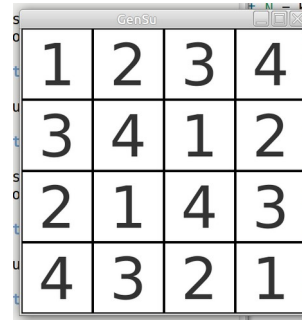
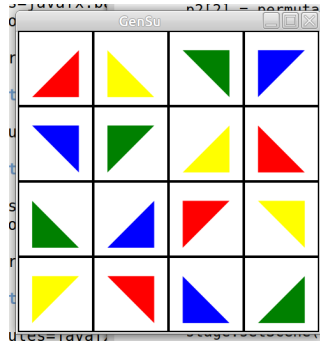


Context matters



Example

The puzzle consists of an $n \times n$ grid ($n = k^2$) of squares. The goal is to fill each square with one of n symbols, such that the symbols in each row, column and (non-overlapping) $k \times k$ box are all different.



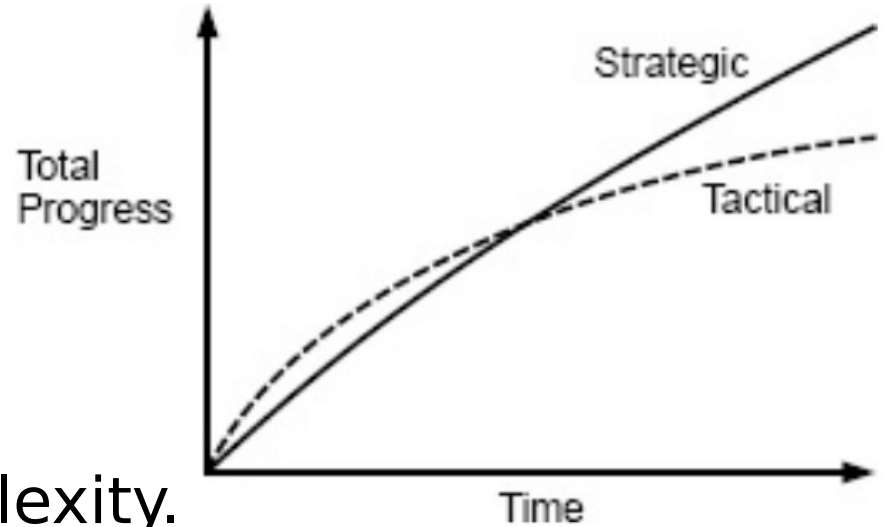
A screenshot of a 16x16 grid puzzle. Each cell contains a number from 0 to 9 or a letter from A to F. The grid is a Latin square where each row, column, and 4x4 box contains all 16 symbols exactly once.

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4	5	6	7	0	1	2	3	C	D	E	F	8	9	A	B
8	9	A	B	C	D	E	F	0	1	2	3	4	5	6	7
C	D	E	F	8	9	A	B	4	5	6	7	0	1	2	3
1	0	3	2	5	4	7	6	9	8	B	A	D	C	F	E
5	4	7	6	1	0	3	2	D	C	F	E	9	8	B	A
9	8	B	A	D	C	F	E	1	0	3	2	5	4	7	6
D	C	F	E	9	8	B	A	5	4	7	6	1	0	3	2
2	3	0	1	6	7	4	5	A	B	8	9	E	F	C	D
6	7	4	5	2	3	0	1	E	F	C	D	A	B	8	9
A	B	8	9	E	F	C	D	2	3	0	1	6	7	4	5
E	F	C	D	A	B	8	9	6	7	4	5	2	3	0	1
3	2	1	0	7	6	5	4	B	A	9	8	F	E	D	C
7	6	5	4	3	2	1	0	F	E	D	C	B	A	9	8
B	A	9	8	F	E	D	C	3	2	1	0	7	6	5	4
F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0



Simplicity

- Avoid *unnecessary* complexity.
- What is “complex” depends on perspective.
- Isolate/encapsulate complexity.
 - Classes: Interface vs. implementation.
 - Make code readable and maintainable for other programmers (including your future self).



(Figure 3.1, Ousterhout, 2018)

Some Principles (Ousterhout)

- **Deep “modules”** (method, class, package, or module)
 - Simple “interfaces”* (narrow)
 - Encapsulate lots of complexity (depth)
 - General-purpose
- Prefer **simple interface** over simple implementation
- Design **errors out of existence**
- Design for **ease of reading**, not ease of writing
- Extra: Don't Repeat Yourself (**DRY**)

* Interfaces in the broad sense, not just the Java keyword



Documentation

- The name of a class/method/variable is the (shortest) description of what it does, but that is often not enough.
- Abstract: describe what, not how.
 - The larger the unit, the more abstract the description.
- Make assumptions and limitations explicit!
 - Is the value of a field tied to some other value?
 - Is a number assumed to be non-zero/within a certain range?
 - Is a reference assumed to always be non-null?
 - Can a String be empty?