Techniques and tOols for Programming (TOP)

Martin Quinson <martin.quinson@loria.fr>

École Supérieure d'Informatique et Applications de Lorraine – 1re année

2008-2009

Module Presentation

Module Presentation

Algorithmic and Programming

Programming? Let the computer do your work!

- How to explain what to do?
- How to make sure that it does what it is supposed to? That it is efficient?
- What if it does not?

Module content and goals:

- Introduction to Algorithmic
  - Master theoretical basements (computer science is a science)
  - Know some classical problem resolution techniques
  - Know how to evaluate solutions (correctness, performance)
- Programming Techniques
  - Programming is an engineering task
  - Master the available tools (debugging, testing)
  - Notion of software engineering (software life cycle)

Module Prerequisites

- Basics of Java (`if`, `for`, methods – *ie.*, tactical programming)
- Sense of logic, intuition

Module organization

Time organization

- 6 two-hours lectures (CM, with Martin Quinson): Concepts introduction
- 10 two-hours exercise session (TD, with staff member[1]): Theoretical exercises
- 6 two-hours labs (TP, with staff member[1]): Coding exercises
- Homework: Systematically finish the in-class exercises

Evaluation

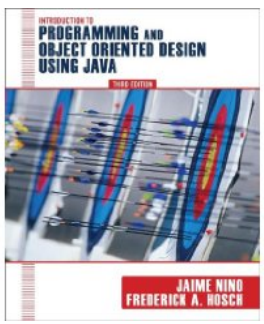
- Two hours table exam
- Quiz at the beginning of each lab
- Maybe an evaluated lab (TP noté) at the end

[1] Martin Quinson, Gérald Oster, Thomas Pietrzak or Rémi Badonnel.

Module bibliography

Bibliography

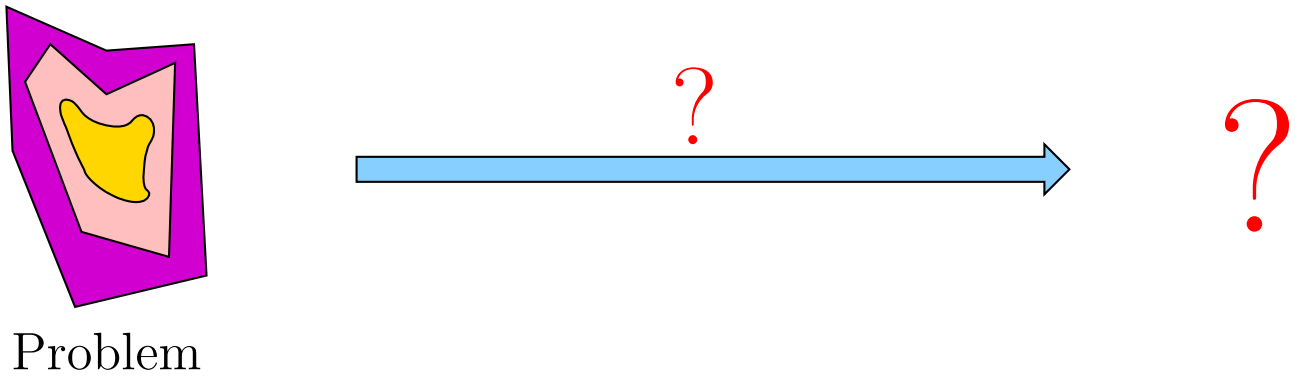
- Introduction to programming and object oriented design, Nino & Hosch.
  Reference book. Very good for SE, less for CS ($120).
- Big Java, Cay S. Horstman. Less focused on programming ($110).
- Programmer en java, Claude Delannoy.
  Bon livre de référence (au format poche – 20€).
- Entraînez-vous et maîtrisez Java par la pratique, Alexandre Brillant.
  Nombreux exercices corrigés (25€).



Webography

- IUT Orsay (in french): `http://www.iut-orsay.fr/~balkansk/`

Problems



Provided by clients (or teachers ;)

Problems

- Problems are generic

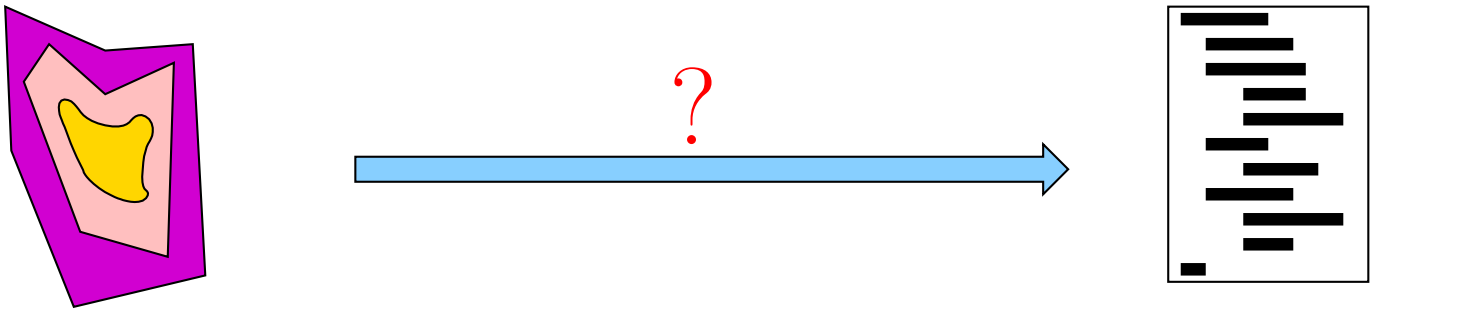
  Example: Determine the minimal value of a set of integers

Instances of a problem

- The problem for a given data set

  Example: Determine the minimal value of $\{17, 6, 42, 24\}$

Problems and Programs



Software systems (*ie.*, Programs)

- Describes a set of actions to be achieved in a given order
- Understandable (and doable) by computers

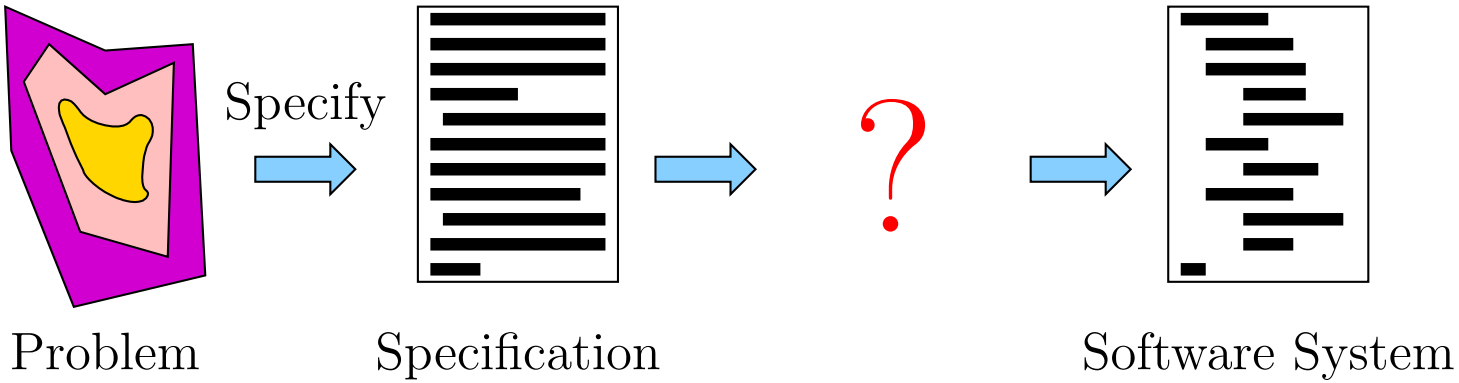
Problem Specification

- Must be clear, precise, complete, without ambiguities
  Bad example: find position of minimal element (two answers for {4, 2, 5, 2, 42})
  Good example: Let L be the set of positions for which the value is minimal.
  Find the minimum of L

Using the Right Models

- Need simple models to understand complex artifacts (ex: city map)

Methodological Principles



Abstraction think before coding (!)

- Describe how to solve the problem

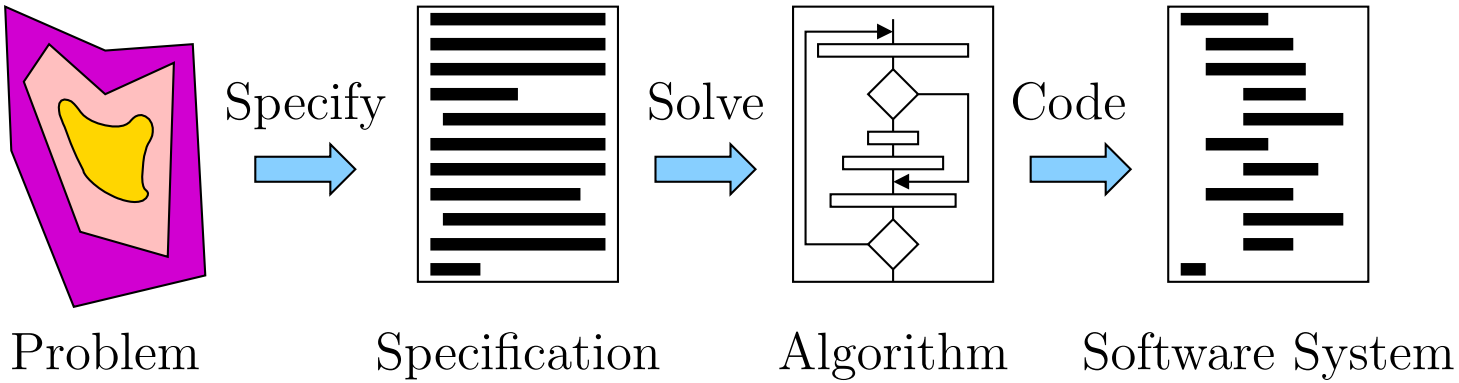
Divide, Conquer and Glue (top-down approach)

- Divide complex problem into simpler sub-problems (think of Descartes)
- Conquer each of them
- Glue (combine) partial solutions into the big one

Modularity

- Large systems built of components: **modules**
- Interface between modules allow to mix and match them

Algorithms



Precise description of the resolution process of a well specified problem

- Must be understandable (by human beings)
- Does not depend on target programming language, compiler or machine
- Can be an diagram (as pictured), but difficult for large problems
- Can be written in a simple language (called **pseudo-code**)

"Formal" definition

- Sequence of actions acting on problem data to induce the expected result

New to Algorithms?

Not quite, you use them since a long time

Lego bricks™	list of pictures →	Castle
Ikea™ desk	building instructions →	Desk
Home location	driving directions →	Party location
Eggs, Wheal, Milk	recipe →	Cake
Two 6-digits integers	arithmetic know-how →	sum

And now

List of students	sorting algorithm →	Sorted list
Maze map	appropriated algorithm →	Way out

Computer Science vs. Software Engineering

> *Computer science is a science of abstraction – creating the right model for a problem and devising the appropriate mechanizable technique to solve it.* – Aho and Ullman

NOT Science of Computers

> *Computer science is not more related to computers than Astronomy to telescopes.* – Dijkstra

- Many concepts were framed and studied before the electronic computer
- To the logicians of the 20's, a *computer* was a person with pencil and paper

Science of Computing

- Automated problem solving
- Automated systems that produce solutions
- Methods to develop solution strategies for these systems
- Application areas for automatic problem solving

Foundations of Computing

Fundamental mathematical and logical structures

- To understand computing
- To analyze and verify the correctness of software and hardware

Main issues of interest in Computer Science

- Calculability
  - Given a problem, can we show whether there exist an algorithm solving?
  - Are there problems for which no algorithm exist?
- Complexity
  - How long does my algorithm need to reach the result?
  - How much memory does it take?
  - Is my algorithm optimal, or does a better one exist?
- Correctness
  - Can we be certain that a given algorithm always reaches a solution?
  - Can we be certain that a given algorithm always reaches the right solution?

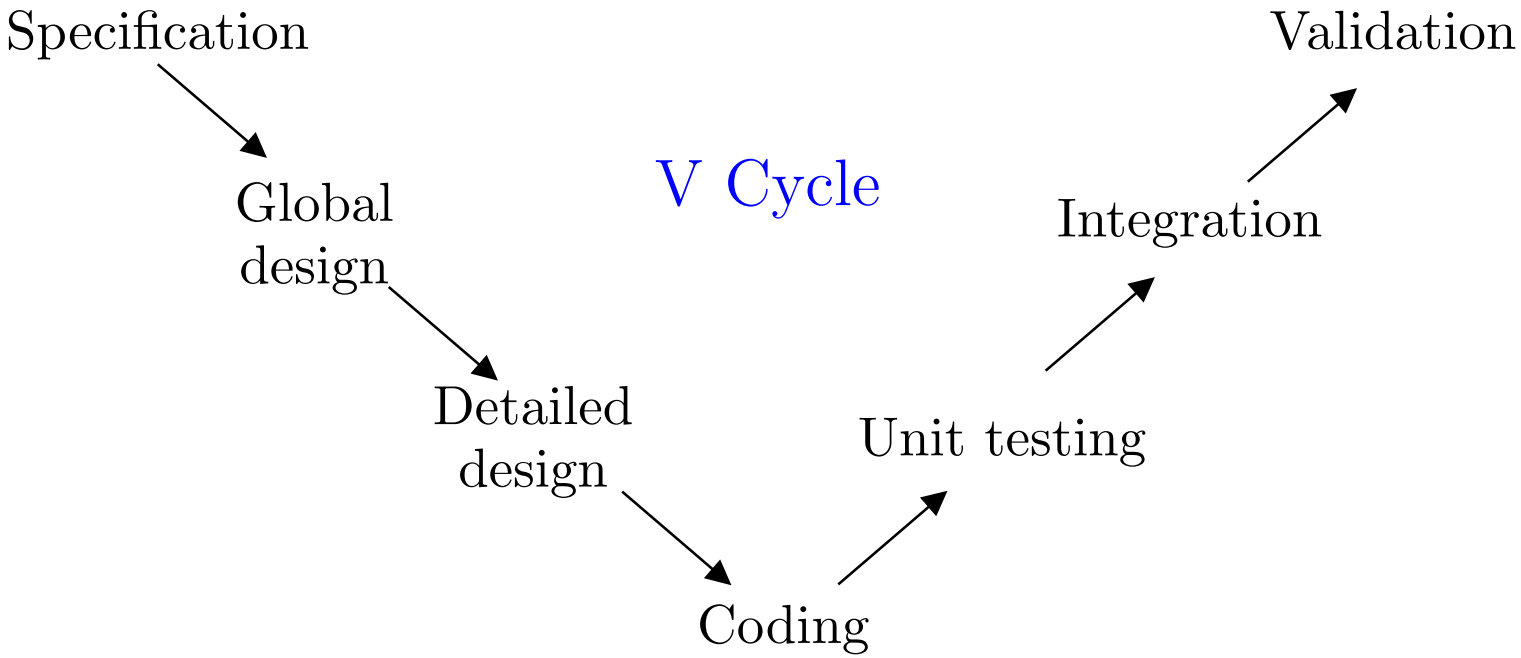
Software Engineering vs. Computer Science

Producing technical answers to consumers' needs

Software Engineering Definition

- Study of methods for producing and evaluating software

Life cycle of a software (*much* more details to come later)



- Global design: Identify application modules
- Detailed design: Specify within modules

As future IT engineers, you need both CS and SE

Without Software Engineering

- Your production will not match consumers' expectation
- You will induce more bugs and problems than solutions
- Each program will be a pain to develop and to maintain for you
- You won't be able to work in teams

Without Computer Science

- Your programs will run slowly, deal only with limited data sizes
- You won't be able to tackle difficult (and thus well paid) issues
- You won't be able to evaluate the difficulty of a task (and thus its price)
- You will reinvent the wheel (badly)

Two approaches of the same issues

- Correctness: CS $\leadsto$ prove algorithms right; SE $\leadsto$ chase (visible) bugs
- Efficiency: CS $\leadsto$ theoretical bounds on performance, optimality proof;
  SE $\leadsto$ optimize execution time and memory usage

First Chapter

Practical and Theoretical Foundations of Programming

- Introduction
  - From the problem to the code
  - Computer Science vs. Software Engineering
- Designing algorithms for complex problems
  - Composition
  - Abstraction
- Comparing algorithms' efficiency
  - Best case, worst case, average analysis
  - Asymptotic complexity
- Algorithmic stability
- Conclusion

There are always several ways to solve a problem

Choice criteria between algorithms

- Correctness: provides the right answer
- Simplicity: KISS! (jargon acronym for *keep it simple, silly*)
- Efficiency: fast, use little memory
- Stability: small change in input does not change output

Real problems ain't easy

- They are not fixed, but dynamic
  - Specification helps users understanding the problem better
    That is why they often add wanted functionalities after specification
  - Example: my text editor is v22.1 (hundreds of versions for "just a text editor")
- They are complex (composed of several interacting entities)

Dealing with complexity

- Some classical design principles help
- Composition: split problem in simpler sub-problems and compose pieces
- Abstraction: forget about details and focus on important aspects

Dealing with complexity: Composition

Composite structure

- Definition: a software system composed of manageable pieces

☺ The smaller the component, the easier it is to build and understand

☹ The more parts, the more possible interactions there are between parts

$\Rightarrow$ the more complex the resulting structure

- Need to balance between simplicity and interaction minimization

Good example: audio system

Easy to manage because:

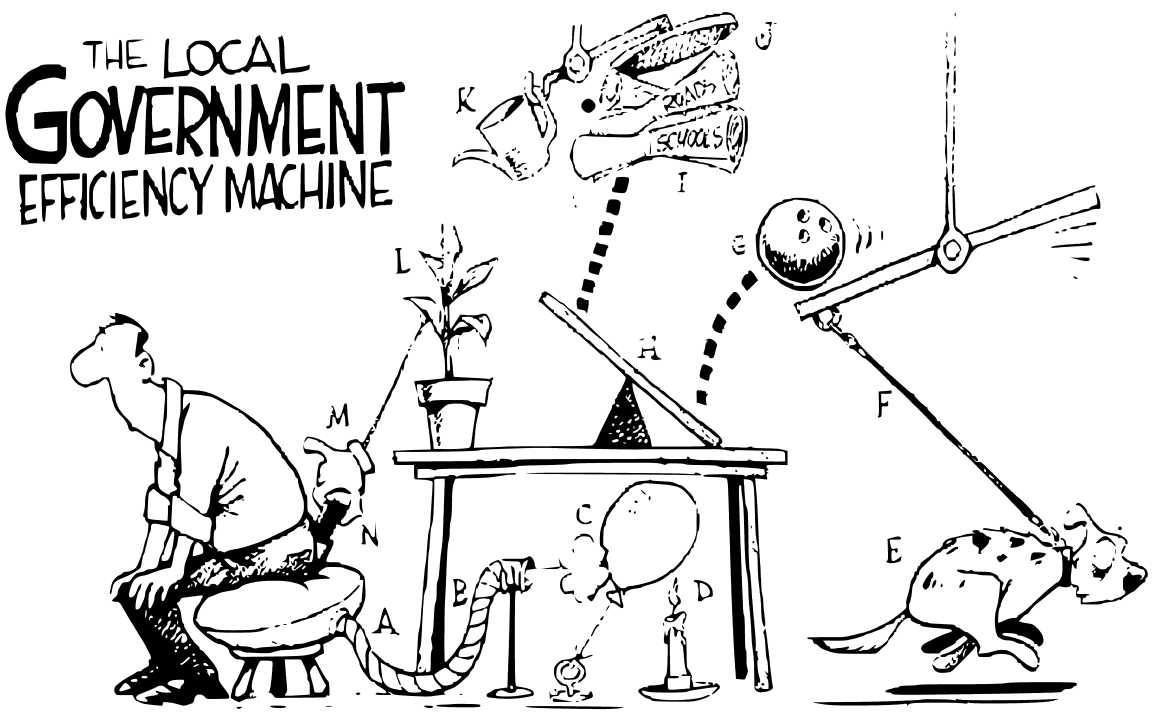
- each component has a carefully specified function
- components are easily integrated
- i.e. the speakers are easily connected to the amplifier

Composition counter-example (1/2)

Rube Goldberg machines

- Device not obvious, modification unthinkable
- Parts lack intrinsic relationship to the solved problem
- Utterly high complexity

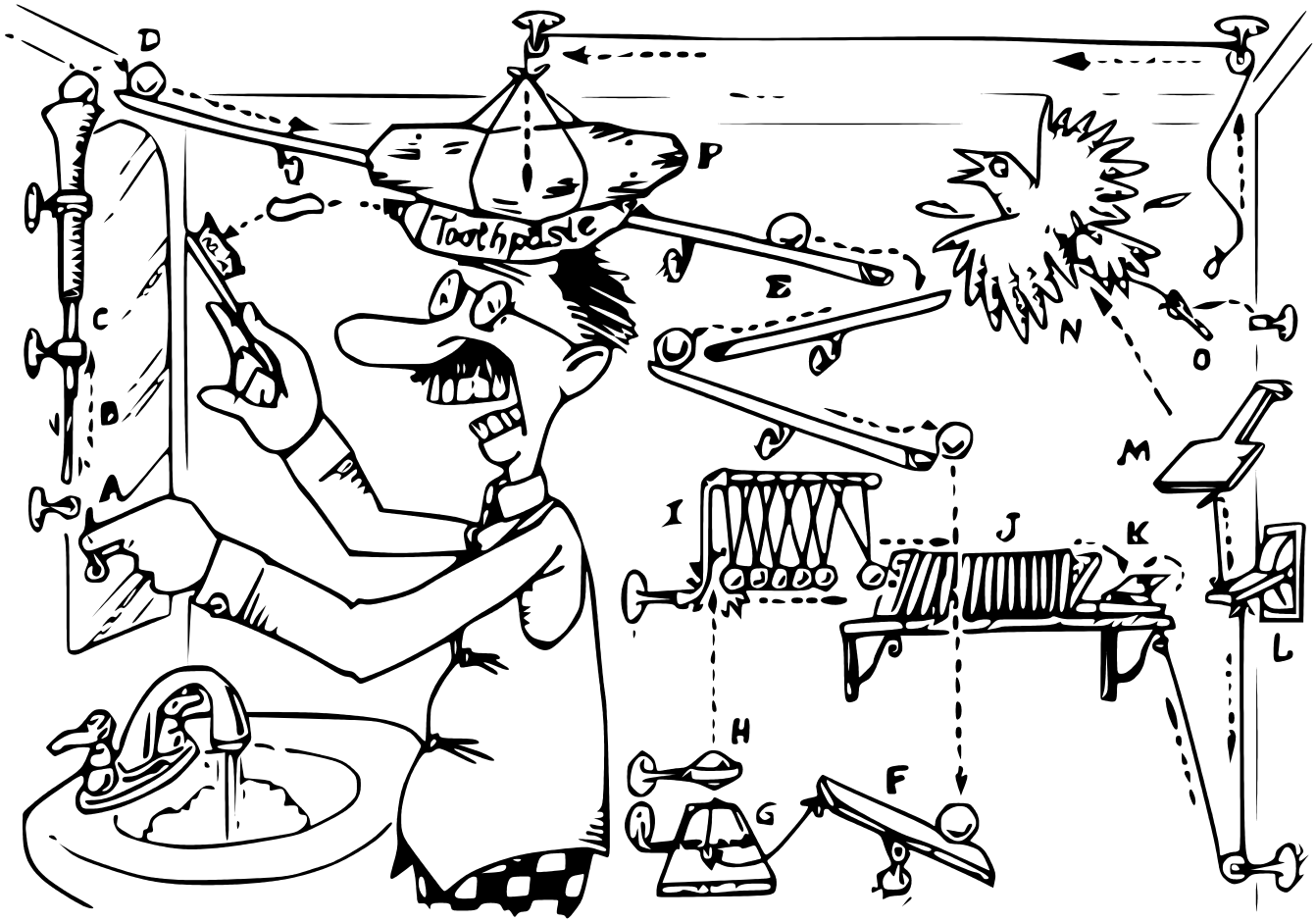
Example: Tax collection machine



A. Taxpayer sits on cushion
B. Forcing air through tube
C. Blowing balloon
D. Into candle
E. Explosion scares dog
F. Which pull leash
G. Dropping ball
H. On teeter totter
I. Launch plans
J. Which tilts lever
K. Then Pitcher
L. Pours water on plant
M. Which grows, pulling chain
N. Hand lifts the wallet

Composition counter-example (2/2)

Rube Goldberg's toothpaste dispenser

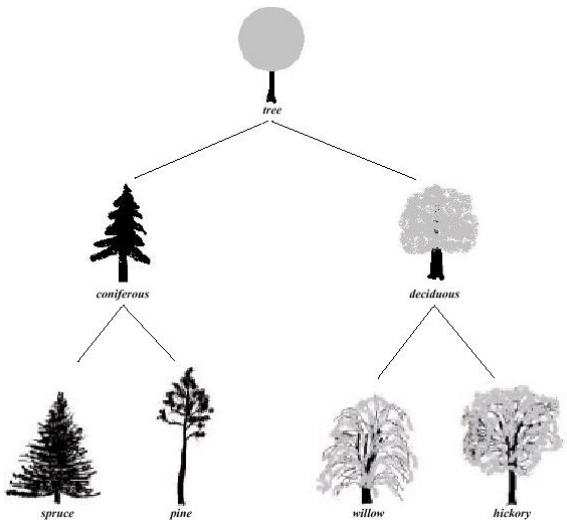


Such over engineered solutions should obviously remain jokes

Dealing with complexity: Abstraction

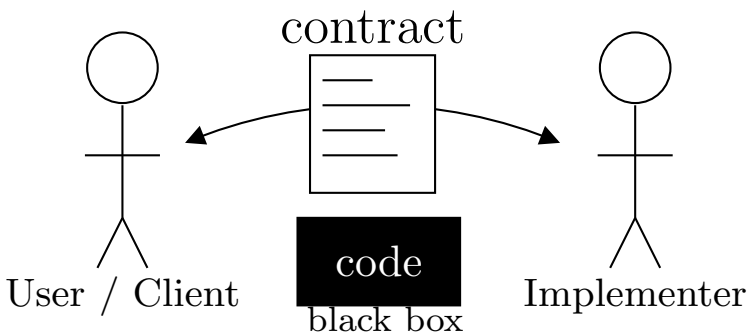
Abstraction

- Dealing with components and interactions without worrying about details
- Not "vague" or "imprecise", but focused on few relevant properties
- Elimination of the irrelevant and amplification of the essential
- Capturing commonality between different things



Abstraction in programming

- Think about what your components should do before
- Ie, abstract their **interface** before coding



- Show your interface, hide your implementation

First Chapter

Practical and Theoretical Foundations of Programming

- Introduction
  - From the problem to the code
  - Computer Science vs. Software Engineering
- Designing algorithms for complex problems
  - Composition
  - Abstraction
- Comparing algorithms' efficiency
  - Best case, worst case, average analysis
  - Asymptotic complexity
- Algorithmic stability
- Conclusion

Comparing Algorithms' Efficiency

There are always more than one way to solve a problem

Choice criteria between algorithms

- Correctness: provides the right answer
- Simplicity: *not* Rube Goldberg's machines
- Efficiency: fast, use little memory
- Stability: small change in input does not change output

Empirical efficiency measurements

- Code the algorithm, benchmark it and use runtime statistics

☹ Several factors impact performance:
machine, language, programmer, compiler, compiler's options, operating system, ...

⇒ Performance not generic enough for comparison

Mathematical efficiency estimation

- Count amount of basic instruction as function of input size

☺ Simpler, more generic and often sufficient

(true in theory; in practice, optimization necessary **in addition** to this)

Best case, worst case, average analysis

Algorithm running time depends on the data

Example: Linear search in an array

```
boolean linearSearch(int val, int[ ] tab) {
  for (int i=0; i<tab.length; i=i+1)
    if (tab[i] == val)
      return true;
  return false;
}
```

- Case 1: search whether 42 is in {42, 3, 2, 6, 7, 8, 12, 16, 17, 32, 55, 44, 12} answer found after one step
- Case 2: search whether 4 is in {42, 3, 2, 6, 7, 8, 12, 16, 17, 32, 55, 44, 12} need to traverse the whole array to decide (n steps)

Counting the instructions to run in each case

- t_{min}: #instructions for the best case inputs
- t_{max}: #instructions for the worst case inputs
- t_{avg}: #instructions on average (average of values coefficiented by probability)
  $t_{avg} = p_1 t_1 + p_2 t_2 + \ldots + p_n t_n$

Linear search runtime analysis

```
for (int i=0; i<tab.length; i=i+1)
  if (tab[i] == val)
    return true;
return false;
```

- For simplicity, let's assume the value is in the array, positions are equally likely
- Let's count tests (noted t), additions (noted a) and value changes (noted c)

Best case: searched data in first position

- 1 value change (i=0); 2 tests (loop boundary + equality)
- $t_{min} = c + 2t$

Worst case: searched data in last position

- 1 value change (i=0); {2 tests, 1 change, 1 addition (i++)} per loop
- $t_{max} = c + n \times (2t + 1c + 1a) = (n+1) \times c + 2n \times t + n \times a$

Average case: searched data in position p with probability $\frac{1}{n}$

- $t_{avg} = c + \sum_{p \in [1,n]} \frac{1}{n} \times (2t + c + a) \times p = c + \frac{1}{n} \times (2t + c + a) \times \sum_{p \in [1,n]} p$

$$t_{avg} = c + \frac{n(n-1)}{2n} \times (2t + c + a) = (n-1) \times t + \frac{n+1}{2} \times c + \frac{n-1}{2} \times a$$

Simplifying equations

$$\boxed{t_{avg} = (n-1) \times t + \frac{n+1}{2} \times c + \frac{n-1}{2} \times a} \text{ is too complicated}$$

Reducing amount of variables

- To simplify, we only count the most expensive operations
- But which it is is not always clear...
- Let's take write accesses (c)

Focusing on dominant elements

- We can forget about constant parts if there is n operations
- We can forget about linear parts if there is n^2 operations
- ...
- Only consider the most dominant elements when n is very big

$\Rightarrow$ This is called **asymptotic complexity**

Asymptotic Complexity: Big-O notation

Mathematical definition

- Let $T(n)$ be a non-negative function
- $\boxed{T(n) \in O(f(n))} \Leftrightarrow \exists$ constants c, n_0 so that $\forall n > n_0, \boxed{T(n) \leq c \times f(n)}$
- f(n) is an upper bound of T(n) ...

... after some point, and with a constant multiplier

Application to runtime evaluation

- $T(n) \in O(n^2) \Rightarrow$ when n is big enough, you need less than n^2 steps
- This gives a upper bound

Big-O examples

Example 1: Simplifying a formula

- Linear search: $t_{avg} = (n-1) \times t + \frac{n+1}{2} \times c + \frac{n-1}{2} \times a \Rightarrow \boxed{T(n) = O(n)}$
- Imaginary example: $T(n) = 17n^2 + \frac{32}{17}n + \pi \Rightarrow \boxed{T(n) = O(n^2)}$
- If T(n) is constant, we write T(n)=O(1)

Practical usage

- Since this is a upper bound, $T(n) = O(n^3)$ is also true when $T(n) = O(n^2)$
- But not as relevant

Example 2: Computing big-O values directly

array initialization

```
for (int i=0;i<tab.length;i++)
  tab[i] = 0;
```

- We have n steps, each of them doing a constant amount of work
- $T(n) = c \times n \quad \Rightarrow \quad T(n) = O(n)$

  (don't bother counting the constant elements)

Big-Omega notation

Mathematical definition

- Let $T(n)$ be a non-negative function
- $\boxed{T(n) \in \Omega(f(n))} \Leftrightarrow \exists$ constants c, n_0 so that $\forall n > n_0, \boxed{T(n) \geq c \times f(n)}$
- Similar to Big-O, but gives a lower bound
- Note: similarly to before, we are interested in big lower bounds

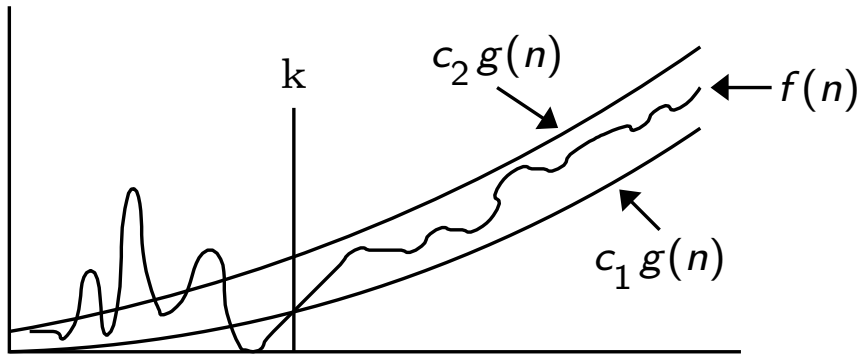
Example: $T(n) = c_1 \times n^2 + c_2 \times n$

- $T(n) = c_1 \times n^2 + c_2 \times n \geq c_1 \times n^2 \qquad \forall n > 1$
  $T(n) \geq c \times n^2$ for $c > c_1$
- Thus, $T(n) = \Omega(n^2)$

Theta notation

Mathematical definition

- $T(n) \in \Theta(g(n))$ if and only if $T(n) \in O(g(n))$ and $T(n) \in \Omega(g(n))$



Example

		n=10	n=1000	n=100000	
$\Theta(n)$	n	10	1000	10^5	seconds
	100n	1000	10^5	10^7	
$\Theta(n^2)$	n^2	100	10^6	10^{10}	minutes
	$100n^2$	10^4	10^8	10^{12}	
$\Theta(n^3)$	n^3	1000	10^9	10^{15}	hours
	$100n^2$	10^5	10^{11}	10^{17}	
$\Theta(2^n)$	2^n	1024	$> 10^{301}$	∞	...
	100×2^n	$> 10^5$	10^{305}	∞	
$\log(n)$	$\log(n)$	3.3	9.9	16.6	
	$100 \log(n)$	332.2	996.5	1661	

Classical mistakes

Mistake notations

- Indeed, we have $O(\log(n)) = O(n) = O(n^2) = O(n^3) = O(2^n)$
  Because it's an upper bound; to be correct we should write $\subset$ instead of $=$
- Likewise, we have $\Omega(\log(n)) = \Omega(n) = \Omega(n^2) = \Omega(n^3) = \Omega(2^n)$
  Because it's a lower bound; we should write $\supset$ instead of $=$
- We only have $\Theta(\log(n)) \neq \Theta(n) \neq \Theta(n^2) \neq \Theta(n^3) \neq \Theta(2^n)$

  (but in practice, everybody use O() as if it were $\Theta()$ – although that's wrong)

Mistake worst case and upper bounds

- Worst case is the input data leading to the longest operation time
- Upper bound gives indications on increase rate when input size increases

  (same distinction between best case and lower bound)

Asymptotic Complexity in Practice

Rules to compute the complexity of an algorithm

Rule 1: Complexity of a sequence of instruction: Sum of complexity of each

Rule 2: Complexity of basic instructions (test, read/write memory): O(1)

Rule 3: Complexity of `if`/`switch` branching: Max of complexities of branches

Rule 4: Complexity of loops: Complexity of content $\times$ amount of loop

Rule 5: Complexity of methods: Complexity of content

Simplification rules

- Ignoring the constant:

  If f(n) = O($k \times g(n)$) and $k > 0$ is constant then f(n) = O(g(n))

- Transitivity

  If f(n) = O(g(n)) and g(n) = O(h(n)) then f(n) = O(h(n))

- Adding big-Os

  If A(n) = O(f(n)) and B(n) = O(h(n)) then A(n)+B(n) = O(max(f(n), g(n)))
  = O(f(n)+g(n))

- Multiplying big-Os

  If A(n) = O(f(n)) and B(n) = O(h(n)) then $A(n) \times B(n)$ = O($f(n) \times g(n)$)

Some examples

Example 1: `a=b;` $\Rightarrow \Theta(1)$ (constant time)

Example 2

```
sum=0;
for (i=0;i<n;i++)
  sum += n;
```

$\Theta(n)$

Example 3

```
sum=0;
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    sum ++;
for (k=0;k<n;k++)
  A[k] = k;
```

$\Theta(1) + \Theta(n^2) + \Theta(n) = \Theta(n^2)$

Example 4

```
sum=0;
for (i=0;i<n;i++)
  for (j=0;j<i;j++)
    sum ++;
```

$\Theta(1) + O(n^2) = O(n^2)$
one can also show $\Theta(n^2)$

Example 5

```
sum=0;
for (i=0;i<n;i*=2)
  sum ++;
```

$\Theta(\log(n))$ log is due to the $i \times 2$

Practical and Theoretical Foundations of Programming

Algorithmic stability

Computers use fixed precision numbers

- 10+1=11
- $10^{10} + 1 = 10000000001$
- $10^{16} + 1 = 10000000000000001$
- $10^{17} + 1 = 100000000000000000 = 10^{17}$

What is the value of $\sqrt{2^2}$?

- Old computers though it was 1.9999999

Other example

```
while (value < 2E9)
  value += 1E-8;
```

This is an infinite loop
(because when $value = 10^9$, $value + 10^{-8} = value$)

Numerical instabilities are to be killed to predict weather, simulate a car crash or control a nuclear power plant

(but this is all *ways* beyond our goal this year ;)

Conclusion of this chapter

What tech guys tend to do when submitted a problem

- They code it directly, and rewrite everything once they understood
- And rewrite everything to improve performance
- And rewrite everything when the code needs to evolve

What managers tend to do when submitted a problem

- They write up a long and verbose specification
- They struggle with the compiler in vain
- Then they pay a tech guy (and pay too much since they don't get the grasp)

What theoreticians tend to do when submitted a problem

- They write a terse but formal specification
- They write an algorithm, and prove its optimality
  (the algorithm never gets coded)

What good programmers do when submitted a problem

- They write a clear specification
- They come up with a clean design
- They devise efficient data structures and algorithms
- Then (and only then), they write a clean and efficient code
- They ensure that the program does what it is supposed to do

Choice criteria between algorithms

Correctness

- Provides the right answer
- This crucial issue is delayed a bit further

Simplicity

- Keep it simple, silly
- Simple programs can evolve (problems and client's wishes often do)
- Rube Goldberg's machines cannot evolve

Efficiency

- Run fast, use little memory
- Asymptotic complexity must remain polynomial
- Note that you cannot have a decent complexity with the wrong data structure
- You still want to test the actual performance of your code in practice

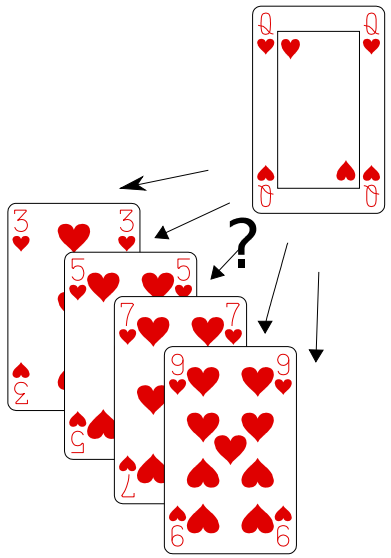
Numerical stability

- Small change in input does not change output
- Advanced issue, critical for numerical simulations (but beyond our scope)

Insertion Sort

How do you sort your card deck?

- No human would apply *selection sort* to sort a deck!



Algorithm used most of the time to sort a card deck:

1. If the cards #1 and #2 need to be swapped, do it
2. Insert card #3 at its position in the [1,2] part of the deck
3. Insert card #4 at its position in the [1,3] part of the deck

   ...

Finding the common pattern

- Step n (≥ 2) is "insert card #(n+1) into [1,n]"
- Step 1 = insert the 2. card into [1,1]
- We may add a Step 0 to generalize the pattern (that's a no-op)

This is *Insertion Sort*

U	N	S	O	R	T	E	D
U	N	S	O	R	T	E	D
N	U	S	O	R	T	E	D
N	S	U	O	R	T	E	D
N	O	S	U	R	T	E	D
N	O	R	S	U	T	E	D
N	O	R	S	T	U	E	D
E	N	O	R	S	T	U	D
D	E	N	O	R	S	T	U

Algorithm big lines

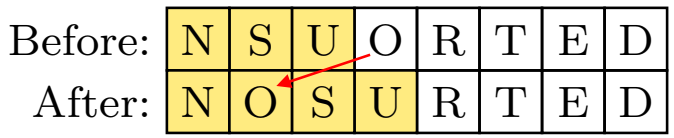
For each element
  Find insertion position
  Move element to position

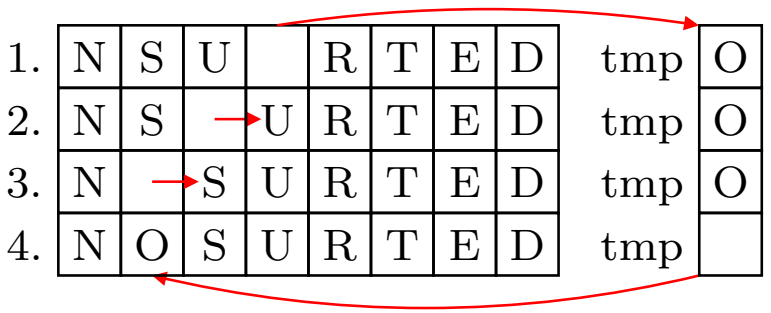
Writing the insertion sort algorithm

Fleshing the big lines

For each element
  Find insertion point
  Move element to position

- Finding the insertion point is easy (searching loop)
- Moving to position is a bit harder: "make room"
- We have to *shift* elements one after the other





- Shifting elements induce a loop also
- We can do both searching insertion point and shifting at the same time

```
/* for each element */
for (i=0; i<length; i++) {
  /* save current value */
  int value = tab[i];
  /* shift to right any element on the left being smaller than value */
  int j=i;
  while ((j > 0) && (tab[j-1]>value)) {
    tab[j] = tab[j-1];
    j--;
  }
  /* Put value in cleared position */
  tab[j]=value;
```

Bubble Sort

- All these sort algorithms are quite difficult to write. Can we do simpler?
- Like "while it's not sorted, sort it a bit"

Detecting that it's sorted

```
for (int i=0; i<length-1; i++)
   /* if these two values are badly sorted */
   if (tab[i]>tab[i+1])
      return false;
return true;
```

How to "sort a bit?"

- We may just swap these two values

```
int tmp=tab[i];
tab[i]=tab[i+1];
tab[i+1]=tmp;
```

All together

- Add boolean variable to check whether it sorted

```
boolean swapped;
do {
   swapped = false;
   for (int i=0; i<length-1; i++)
      /* if these two values are badly sorted */
      if (tab[i]>tab[i+1]) {
         /* swap them */
         int tmp=tab[i];
         tab[i]=tab[i+1];
         tab[i+1]=tmp;
         /* and remember we swapped something */
         swapped = true;
      }
} while (swapped);/* until a traversal without swapping */
```

Conclusion on Iterative Sorting Algorithms

Cost Theoretical Analysis

Amount of comparisons	Best Case	Average Case	Worst Case
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$

Which is the best in practice?

- We will explore practical performance during the lab
- But in practice, bubble sort is awfully slow and should never be used

Is it optimal?

- The lower bound is $\Omega(n \log(n))$
- Some other algorithms achieve it (Quick Sort, Merge Sort)
- We come back on these next week

Third Chapter

Recursion

- Back-tracking
- Conclusion

Third Chapter

Recursion

- Back-tracking
- Conclusion

Combinatorial Optimization

Large class of Problems with similar approaches

Problem

- Solutions are really numerous; A set of constraints make some solution invalids
- We look for the solution maximizing a function

Examples

- Knapsac: Ali-Baba searches object set fitting in bag maximizing the value
- Minimum Spanning Tree of a given graph
- Traveling Salesman: visit n cities in order minimizing the total distance
- Artificial Intelligence: select best solution from set of possibilities

Resolution Approaches

- Exhaustive Search: study *every* solutions (often exponential – ie infeasible)
  $\leadsto$ maximize value of any possible knapsack contents
- Backtracking: tentative choices + backtrack to previous decision point
  Restricting study to valid solutions $\leadsto$ if bag is full, don't stuff something else
  Factorizing computations $\leadsto$ only sum up once the N first objects' value

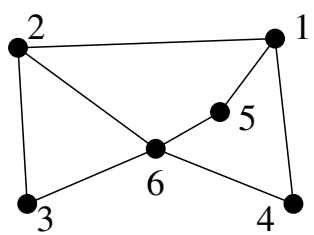
Back-tracking

Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

First example: Independent Sets

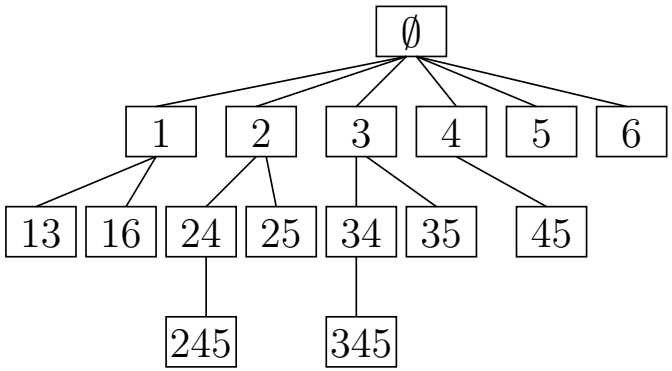
- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.
- $\{2\}$, $\{2,4\}$, $\{2,4,5\}$ (Stuck; remove 5 then 4) $\{2,5\}$
- $\{3\}$, $\{3,4\}$, $\{3,4,5\}$, $\{3,5\}$; $\{4\}$, $\{4,5\}$; $\{5\}$, $\{6\}$

Algorithm Computation Time

Solution Tree of this Algorithm



- Traverse every nodes (without building it explicitly)
- Amount of algorithm steps = amount of solutions
- Let n be amount of nodes

Amount of solutions for a given graph?

- Empty Graph (no edge) $\rightsquigarrow$ $I_n = 2^n$ independent sets
- Full Graph (every edges) $\rightsquigarrow$ $I_n = n + 1$ independent sets
- On average $\rightsquigarrow$ $I_n = \sum_{k=0}^{n} \binom{k}{n} 2^{-k(k-1)/2}$

n	2	3	4	5	10	15	20	30	40
I_n	3,5	5,6	8,5	12,3	52	149,8	350,6	1342,5	3862,9
2^n	4	8	16	32	1024	32768	1048576	1073741824	1099511627776

- Backtracking algorithm traverses I_n nodes on average
- An exhaustive search traverses 2^n nodes

Other example: n queens puzzle

Goal:

- Put n queens on a $n \times n$ board so than none of them can capture any other

Algorithm:

- Put a queen on first line
  There is n choices, any implying constraints for the following
- Recursive call for next line

Pseudo-code `put_queens(int line, board)`

If $line > line\_count$, return `board` (success)

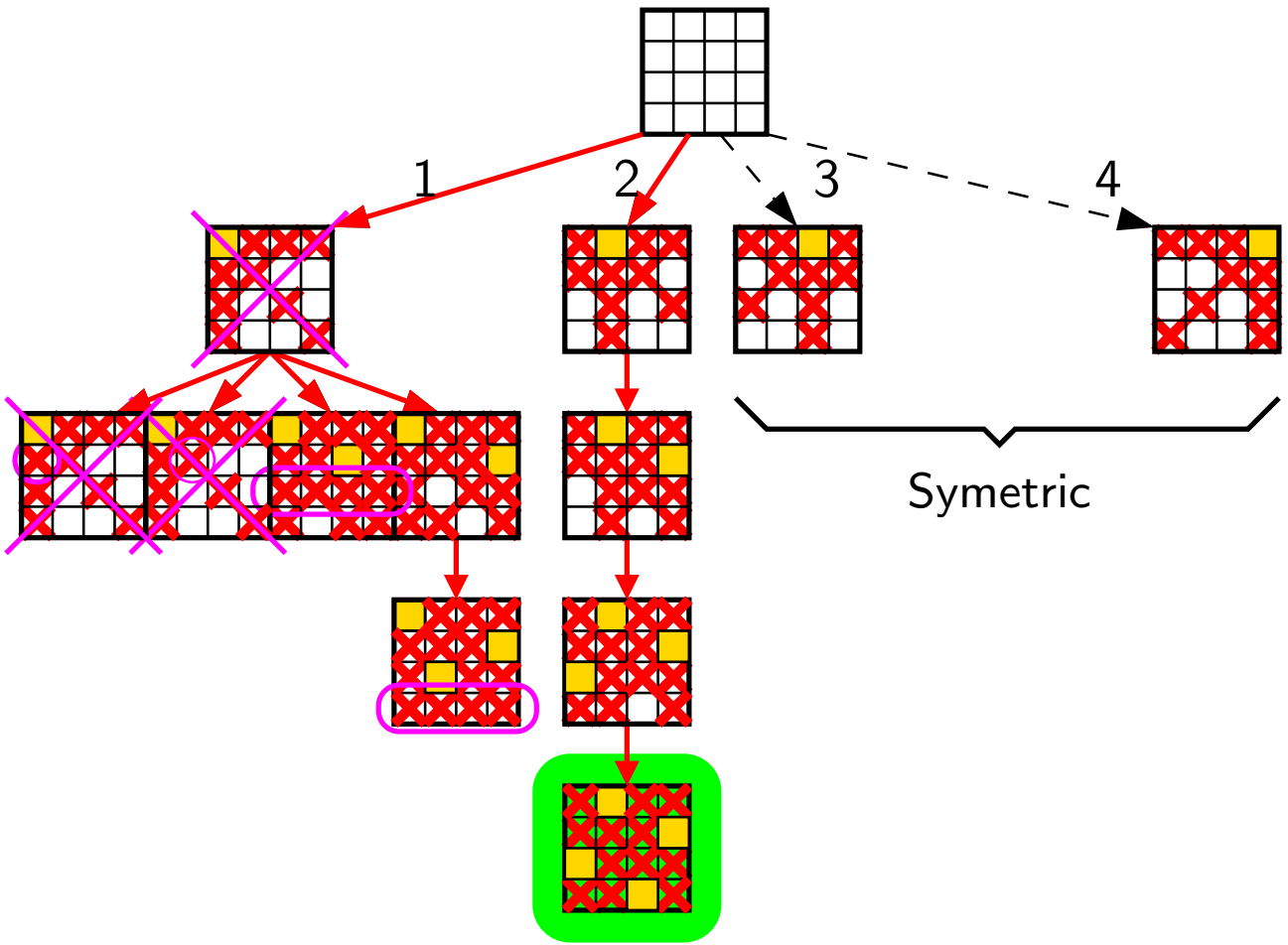
$\forall\ cell \in line$,

- Put a queen at position $cell \times line$ of `board`
- If conflict, then return (stopping descent – failure)
- (else) call `put_queens(ligne+1, board` $\cap\ \{cell, line\}$`)`

$\Rightarrow$ Recursive Call within a Loop

Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)
- Until we find a solution (or not)



Java implementation of n queens puzzle

```
boolean Solution(boolean board[][], int line) {
  if (line >= board.length) // Base Case
     return true;

  for (int col = 0; col < board.length; col++) {  // loop on possibilities
     if (validPlacement(board, line, col)) {
       putQueen(board, line, col);
       if (Solution(plateau, line + 1)) // Recursive Call
           return true; // Let solution climb back
       removeQueen(board, line, col);
     }
  }
  return false;
}
```

Some Principles on Backtracking

- Study "depth first" of solution tree
- On backtracking, restore state as before last choice
  Trivial here (parameters copied on recursive call), harder in iterative
- Strategy on branch ordering can improve things
- Progressive Construction of boolean function
- If function returns false, there is no solution

- Probable Combinatorial Explosion (4^4 boards)
  $\Rightarrow$ Need for heuristics to limit amount of tries

Conclusion on recursion

Essential Tool for Algorithms

- Recursion in Computer Science, induction in Mathematics
- Recursive Algorithms are frequent because easier to understand ...
  (and thus easier to maintain)

  ... but maybe slightly more difficult to write (that's a practice to get)
- Recursive programs maybe slightly less efficients...

  ... but always possible to transform a code to non-recursive form
  (and compilers do it)
- Classical Functions: Factorial, gcd, Fibonacci, Ackerman, Hanoï, Syracuse, ...
- BackTracking: exhaustive search in space of *valid* solutions
- Data Structure module: several recursive datatypes with associated algorithms

Merge Sort

Recursive sorting

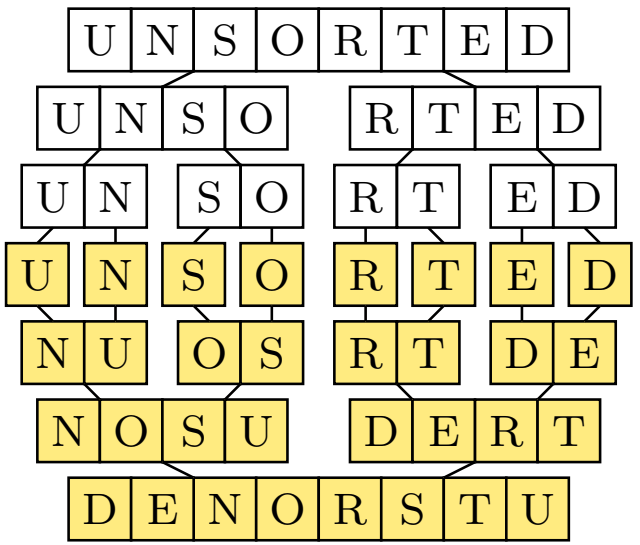
- Imagine the simpler way to sort recursively a list

1. Split your list in two sub-lists
   One idea is to split evenly, but not the only one
2. Sort each of them recursively
   (base case: size$\leq$1)
3. Merge sorted sublists back
   at each step, pick smallest remaining elements of sublists, put it after already picked

Merge Sort

- List splited evenly
- Sub-list copied away
- Merge trivial

(invented by John von Neumann in 1945)



Merge Sort

Pseudo-code

```
function merge_sort(m)
    var list left, right, result
    if length(m) <= 1
        return m

    var middle = length(m) / 2
    for each x in m up to middle
         add x to left
    for each x in m after middle
         add x to right
    left = merge_sort(left)
    right = merge_sort(right)
    result = merge(left, right)
    return result
```

(C) Wikipedia

```
function merge(left,right)
    var list result
    while length(left)>0 and length(right)>0
        if first(left) <= first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    end while
    while length(left) > 0
        append left to result
    while length(right) > 0
        append right to result
    return result
```

Complexity Analysis

- Time: $\log(n)$ recursive calls, each of them being linear $\rightsquigarrow \Theta(n \times \log(n))$
- Space: Need to copy the array $\rightsquigarrow 2n$ (quite annoying)

QuickSort

Presentation

- Invented by C.A.R. Hoare in 1962
- Widely used (in C library for example)

Big lines

- Pick one element, called *pivot* (random is ok)
- Reorder elements so that:
  - elements smaller to the pivot are before it
  - elements smaller to the pivot are after it
- Recursively sort the parts before and after the pivot

Questions to answer

- How to pick the pivot? (random is ok)
- How to reorder the elements?
  - First solution: build sub-list (but this requires extra space)
  - Other solution: invert in place (but hinders stability, see below)

Simple Quick Sort

Building sub-lists makes it easy:

- Create two empty list variables
- Iterate over the original list, and put elements in correct sublist
- Recurse
- Concatenate results

```
function quicksort(array)
    var list less, greater
    if length(array) <= 1
        return array
    select and remove a pivot value pivot from array
    for each x in array
        if x <= pivot then append x to less
        else append x to greater
    return concatenate(quicksort(less), pivot, quicksort(greater))
```

(C)wikipedia

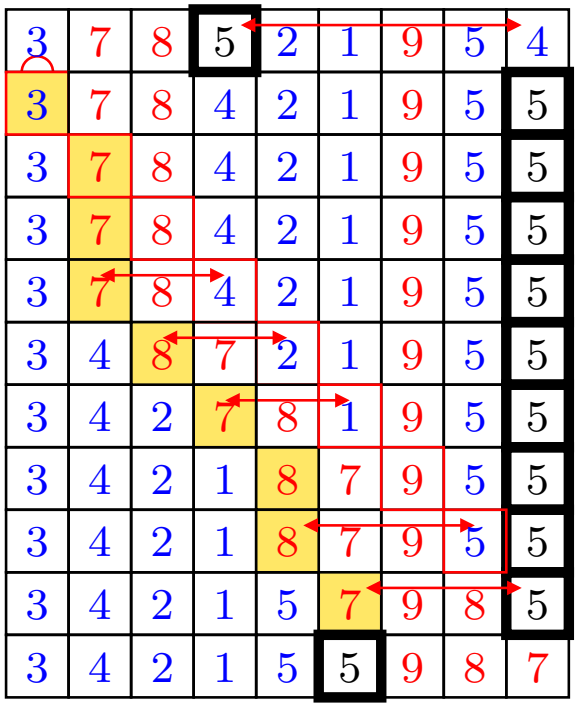
Problem

- Space complexity is about $2n + \log(n))$...
  (2n for array dupplication, log(n) for the recursion stack)

In-place Quick Sort

Big lines of the list reordering

- Put the pivot at the end
- Traverse the list
  - If visited element is smaller, do nothing
  - Else swap with "storage point"
    + shift storage right
    (storage point is on left initially)
- Swap pivot with storage point




```
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right] // Move pivot to end
   storeIndex := left
   for i  from  left to right - 1
       if array[i] <= pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1
   swap array[storeIndex] and array[right] // Move pivot to its final place
   return storeIndex
```

(C) wikipedia

Discussion

Complexity Analysis

- Space complexity: $O(\log(n))$ (recursion stack)
- Time complexity: classical recursion function? almost
  - Function `Partition` in $\Theta(n)$
  - Amount of steps depends on how evenly the list gets splitted
    - Evenly? $\leadsto \Theta(\log(n))$ steps $\leadsto$ QuickSort in $\Theta(n\log(n))$
    - 1%/99%? $\leadsto 100 \times \log(n)$ steps $\leadsto \Theta(n\log(n))$
    - Fixed amount of values on one side? $\leadsto O(n)$ steps $\leadsto O(n^2)$
- Worst case arise when every values are dupplicated

Discussion

- Merge Sort does less comparison and less moves than QuickSort
- In-Place version of both algorithms are not stable
- Both can be quite easily parallelized

Fifth Chapter

Correction of Software Systems

- Hoare Logic
- Preuves de fonctions récursives

Selection sort's performance discussion

We have shown that

- Space complexity is in $\Theta(1)$ (was part of problem specification)
- Time complexity is in $\Theta(n^2)$, regardless of the input
  (best case = worst case = average case)
- Very accurate knowledge on achieved performance

But wait a second...

How do you know this code actually sorts the array?

- Because the teacher / a friend says so
- Because it's written in a book / on the internet
- Because you see it, it's obvious (yeah, right...)

First move to convince septics: you test it

- Problem: a *whole* load of arrays exists out there. Cannot test them all...
- How much should you test to get convincing? Which ones do you pick?

To convince real septics, you have to **prove** correctness

How to prove that 'selection sort' sorts arrays?

Back to the roots: what exactly do you want to prove?

- Proper specification mandatory to proof: gives what we have, what we want
- We also need a mathematical logic to carry the proof

Hoare Logic [Hoare 1969]

- Set of logical rules to reason about the correctness of computer programs
- Central feature: description of state changes induced by code execution
- **Hoare triple:** $\boxed{\{P\}\ C\ \{Q\}}$
  - C is the code to be run
  - P is the **precondition** (assertion about previous state)
  - Q is the **postcondition** (assertion about next state)
  - This can be read as "If P is true, then when I run C, Q becomes true"
  - C is said to satisfy specification (P, Q)
- Such notation allows very precise algorithm specifications
- Axioms and Inference rules allow rigorous correctness demonstrations
- Note: other logics (temporal logic) proposed as replacement, but harder

Assertions

What exactly is an assertion?

Definition

Formula of first order logic describing relationships between algorithm's variables

Constituted of:

- Variables of algorithm pseudo-code
- Logical connectors: $\wedge$ (and) $\vee$ (or) $\neg$ (not) $\Rightarrow$, $\Leftarrow$
- Quantifiers: $\exists$ (exists), $\forall$ (for all)
- Value-specific elements (describing integers, reals, booleans, arrays, sets, ...)

Example:

- $(x \times y = z) \ \wedge \ (x \leq 0)$
- $n^2 \geq x$

Examples of specification

Solving quadratic equations ($ax^2 + bx + c = 0$)

P: $a, b, c \in \mathbb{R}$ and $a \neq 0$

Q: $(solAmount \in \mathbb{N}) \wedge (s, t \in \mathbb{R}) \wedge$
$((solAmount = 0) \vee$
$(solAmount = 1 \wedge as^2 + bs + c = 0) \vee$
$(solAmount = 2 \wedge as^2 + bs + c = 0 \wedge at^2 + bt + c = 0 \wedge s \neq t))$

Possible implementation

$\Delta = b^2 - 4ac$
if $(\Delta > 0)$
$s = \frac{-b+\sqrt{\Delta}}{2a}; t = \frac{-b-\sqrt{\Delta}}{2a};$
$solAmount = 2$
else if $(\Delta = 0)$
$s = \frac{-b}{2a}; solAmount = 1$
else (ie, $\Delta < 0$)
solAmount=0

- Here, the proof will be difficult...
- ...because it is trivial.
- Correctness comes from definitions!

Demonstration tool: inference rules

Definitions

- Inference: deducting new facts by combining existing facts correctly
- Inference rule: mechanism specifying how facts can be combined

Classical representation of each rule:

$$\frac{p_1, p_2, p_3, \ldots, p_n}{q}$$

- Can be read as "if all $p_1, p_2, p_3, \ldots, p_n$ are true, then q is also true"
- Or "in order to prove q, you have to prove $p_1, p_2, p_3, \ldots, p_n$"
- Or "q can be deduced from $p_1, p_2, p_3, \ldots, p_n$"

First axioms and rules

Empty statement axiom

$$\frac{}{\{P\}skip\{P\}}$$

Assignment axiom

$$\frac{}{\{P[x/E]\}x := E\{P\}}$$

- $P[x/E]$ is P with all free occurrences of variable x replaced with expression E
- Example:
  - P: $x = a \wedge y = b$
  - Q: $x = b \wedge y = a$
  - SWAP: algorithm achieving transition; For example: $t = x; x = y; y = t$
  - We should prove: $\{P\}SWAP\{Q\}$

Consequence rule

$$\frac{P \Rightarrow P', \{P\}\ C\ \{Q\}, Q \Rightarrow Q'}{\{P'\}\ C\ \{Q'\}}$$

- P is said to be weaker than P'
- Q is said to be stronger than Q'

Rules for algorithmic constructs

Rule of composition $$\frac{\{P\}C_1\{Q\},\ \ \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

- $C_1; C_2$ means that both code are executed one after the other.
- Can naturally be generalized to more than two codes

Conditional Rule $$\frac{\{P \wedge Cond\}\ T\ \{Q\},\ \ P \wedge \neg Cond \Rightarrow Q}{\{P\}\ \textbf{if}\ Cond\ \textbf{then}\ T\ \textbf{endif}\ \{Q\}}$$

Conditional Rule 2 $$\frac{\{P \wedge Cond\}\ T\ \{Q\},\ \ \{P \wedge \neg Cond\}\ E\ \{Q\}}{\{P\}\ \textbf{if}\ Cond\ \textbf{then}\ T\ \textbf{else}\ E\ \textbf{endif}\ \{Q\}}$$

While Rule $$\frac{\{I \wedge Cond\}\ L\ \{I\}}{\{I\}\ \textbf{while}\ Cond\ \textbf{do}\ L\ \textbf{endif}\ \{I \wedge \neg Cond\}}$$

- $\{I\}$ is said to be the **loop invariant**

How to prove algorithms?

Proving an algorithm: two steps

- Correction proof: when it terminates, the algorithm produce a valid result with regard to problem specification
- Terminaison proof: the algorithm always terminate

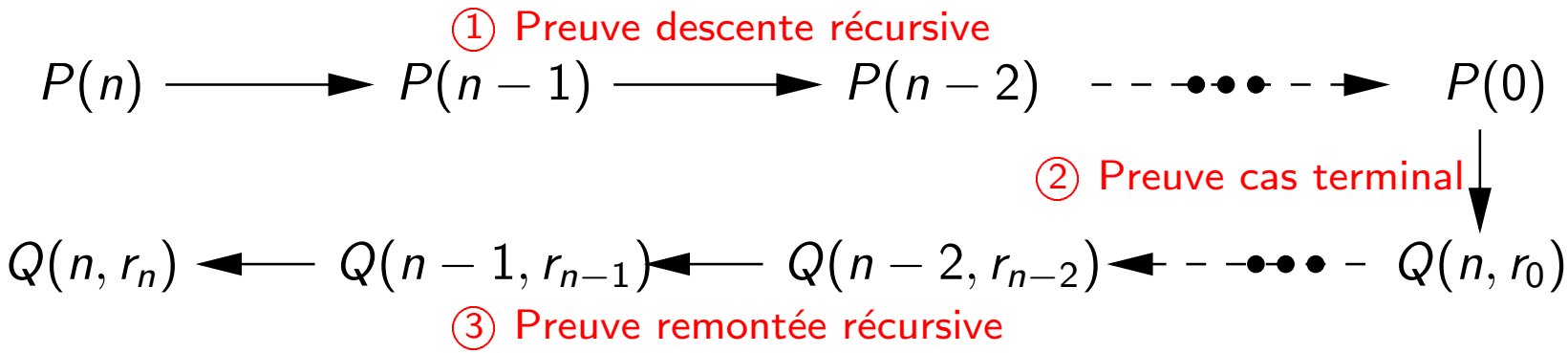
Coming now

- Application to recursive function

Idée de la correction de fonctions récursives

$P(n)$: Précondition étape n; $Q(n, r_n)$: Postcondition étape n avec résultat r_n

On veut montrer P(n) {TREC} Q(n, r_n)



Si $f(n)$ s'exprime en fonction de $f(n-1)$, il faut que:

- Dans le cas général
  - Précondition de $f(n)$ implique précondition de $f(n-1)$ ①
    Si non, le calcul est impossible
  - HdR: postcondition de $f(n-1)$ vraie. Prouver postcondition de $f(n)$ ③
- Dans le cas terminal
  - la précondition et le traitement permettent de prouver la postcondition ②

Preuve de la correction (1/2)

$$P(n)\ \{TREC\}\ Q(n, r_n) \quad (1)$$

$$P(n)\ \{\textbf{si}\ \text{cond}\ \textbf{alors}\ \text{TTER}\ \textbf{sinon}\ \text{TGEN}\}\ Q(n, r_n)$$

Cas simple: TGEN et TTER sont des affectations

TGEN: $r \leftarrow G(n, f(n_{int}))$

TTER: $r \leftarrow v(n)$

Avec n_{int} Valeur de l'appel récursif

$f(x)$ L'appel récursif

$v(n)$ Fonction sans appel à $f(n)$

$G(n, y)$ Fonction:

- Sans appel récursif à $f(n)$
- Définie $\forall n$ paramètre, $\forall y$

Exemple: Factorielle

TGEN: $r \leftarrow n \times facto(n-1)$

TTER: $r \leftarrow 1$

$n_{int} = n - 1$

$f(x)$: $facto(x)$

v(n) $= 1$

G(n,y) $= n \times y$

$P(n)$: $n \geq 0$

$Q(n, r)$: $r = n!$

cond(n) : n=0

Preuve de la correction (2/2)

Cas simple: TTER et TGEN sont des affectations

- Algorithme calculant $r = f(n)$

  si cond(n) **alors** $r \leftarrow v(x)$
  sinon $r \leftarrow G(n, f(n_{int}))$

- Pour prouver (1), il suffit de prouver:
  - Dans le cas terminal: précondition et traitement impliquent postcondition
    $$P(n) \wedge cond(n) \Rightarrow Q(n, r)$$
  - Dans le cas général:
    - Descente récursive: précondition de $f(n)$ implique précondition de $f(n-1)$
      $$P(n) \wedge \neg cond(n) \Rightarrow P(n_{int})$$
    - Remontée récursive: postcondition de $f(n-1)$ implique postcondition de $f(n)$
      $$P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n, r)$$

Cas plus général: plus dur

- Il faut combiner ceci avec les autres cours de preuve
  $$P(x) \Rightarrow P(x_{int}) \qquad\qquad Q(x_{int}, r_{int}) \Rightarrow Q(x, r)$$

Exemple de la factorielle

```
FACTORIELLE(n):
  si n = 0 alors  r ← 1                          (TTER)
           sinon  r ← n × factorielle(n − 1)     (TGEN)
  finsi
```

$P(n)$: $n \geq 0$ cond(n): $n = 0$ $Q(n, r)$: $r = n!$

- Cas terminal: $P(n) \wedge cond(n) \Rightarrow Q(n, r) \equiv n \geq 0 \wedge n = 0 \Rightarrow r = n!$
  Vrai (car $1 = 0!$ quoi qu'il arrive)
- Cas général:
  - $P(n) \wedge \neg cond(n) \Rightarrow P(n_{int}) \equiv (n \geq 0) \wedge (n \neq 0) \Rightarrow (n - 1 \geq 0)$
    Trivial
  - $P(n) \wedge \neg cond(n) \wedge Q(n_{int}, r_{int}) \Rightarrow Q(n, r)$
    $\equiv (n \geq 0) \wedge (n \neq 0) \wedge (r_{int} = n_{int}!) \Rightarrow (r = n!)$

    Vrai car:
    - $r = n \times r_{int}$ dans le cas général
    - $r_{int} = n_{int}! = (n - 1)!$ par HdR
    - $n \times (n - 1) = n!$

Preuve de terminaison

- Conditions suffisantes:
  - Valeurs successives du paramètre x: suite strictement monotone (pour un ordre éventuellement à préciser)
  - Existence d'un extrema x_0 vérifiant la condition d'arrêt
- Remarque: la suite de Syracuse semble se terminer sans ceci

- Exemple: la factorielle, bien sûr
  - $n \geq 0$
  - n strictement décroissant
  - $0 =$ condition d'arrêt

Sixth Chapter

Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Conclusion on Testing
- JUnit
- Design By Contract

Introduction

Bugs

- Bugs are inevitable in complex software system
- A bug can be very visible or can hide in your code until a much later date

Chasing bugs

- Once identified, use print statements of IDE's debugger to hunt them down
- But how to discover all bugs in the system, even those with low visibility?

$\Rightarrow$ Testing and Quality Assurance practices

Why to test?

> *Testing can only prove the presence of defects, not their absence.*
> – E. W. Dijkstra

- Perfect Excuse: Don't invest in testing: system will contain defects anyway
- Counter Arguments:
  - The more you test, the less likely such defects will *cause harm*
  - The more you test, the more *confidence* you will have in the system

Who should Test?

Fact: Programmers are not necessarily the best testers

- Programming is a constructive activity: try to make things work
- Testing is a destructive activity: try to make things fail

In practice

- Best case: Testing is part of quality assurance
  - done by developers when finishing a component (unit tests)
  - done by a specialized test team when finishing a subsystem (integration tests)
- Common case: done by rookies
  - testing seen as a beginner's job, assigned to least experienced team members
  - testing often done after completion (if at all)
  - but very difficult task; impossible to completely test a system
- Worst case (unfortunately very common too): no one does it
  - Not productive $\Rightarrow$ not done [yet], postponed "by a while"
  - But without testing, productivity decreases, so less time, so less tests

What is "Correct"?

different meanings depending on the context

Correctness

- A system is correct if it behaves according to its specification

$\Rightarrow$ An absolute property (i.e., a system cannot be "almost correct")

$\Rightarrow$ ... undecideable in theory and practice

Reliability

- The user may rely on the system behaving properly
- Probability that the system will operate as expected over a specified interval

$\Rightarrow$ Relative property (system mean time between failure (MTTF): 3 weeks)

Robustness

- System behaves reasonably even in circumstances that were not specified

$\Rightarrow$ Vague property (specifying abnormal circumstances $\rightsquigarrow$ part of the requirements)

Terminology

Avoid the term "Bug"

- Implies that mistakes somehow creep into the software from the outside
- imprecise because mixes various "mistakes"

Error: incorrect software behavior

- A deviation between the specification and the running system
- A manifestation of a defect during system execution
- Inability to perform required function within specified limits
- *Example*: message box text said "Welcome null."
- Transient error: only with certain inputs; Permanent error: for any input

Fault: cause of error

- Design or coding mistake that may cause abnormal behavior
- *Example:* account name field is not set properly.
- A fault is not an error, but it can lead to them

Failure: particular instance of a general error, caused by a fault

Quality Control Techniques

Large systems bound to have faults. How to deal with that?

Fault Avoidance: Prevent errors by finding faults before the release

- Development methodologies:
  Use requirements and design to minimize introduction of faults
  Get clear requirements; Minimize coupling
- Configuration management: don't allow changes to subsystem interfaces
- [Formal] Verification: find faults in system execution
  Maturity issue; Assumes requirements, pre/postconditions are correct & adequate
- Review: manual inspection of system by team members
  shown effective at finding errors

Fault detection: Find existing faults without recovering from the errors

- Manual tests: Use debugger to move through steps to reach erroneous state
- Automatic Testing: tries to expose errors in planned way ← **We are here**

Fault tolerance: When system can recover from failure by itself

- Recovery from failure (*example:* DB rollbacks, FS logs)
- Sub-system redundancy (*example:* disk RAID-1)

Testing Concepts

Recapping generic terms

- Error: Incorrect software behavior
- Fault: Cause of the error (programming, design, etc)
- Failure: Particular instance of a general error, caused by a fault

Component

- A part of the system that can be isolated (through *stub* and *driver*) for testing

$\Rightarrow$ an object, a group of objects, one or more subsystems

Test Case

- {inputs; expected results} set exercising component to cause failures
- Boolean method: whether component's answer matches expected results
- "expected results" includes exceptions, error codes ...

Test Stub

- Partial implementation of components on which the tested compnt depends
- dummy code providing necessary input values and behavior to run test cases

Test Driver

- Partial implementation of a component that depends on the tested part
- a "main()" function that executes a number of test cases

Tests Campaign Planing

Goal

- Should *verify* the requirements (are we building the product right?)
- NOT *validate* the requirements (are we building the right product?)

Definitions

- **Testing**: activity of executing a program with the intent of finding a defect
  $\Rightarrow$ A successful test is one that finds defects!
- **Testing Techniques**: Techniques to find yet undiscovered mistakes
  $\Rightarrow$ Criterion: Coverage of the system
- **Testing Strategies**: Plans telling *when* to perform *what* testing technique
  $\Rightarrow$ Criterion: Confidence that you can safely proceed with the next activity

Sixth Chapter

Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Conclusion on Testing
- JUnit
- Design By Contract

White Box Testing

Focuses on internal states of objects

Use internal knowledge of the component to craft input data

- *Example:* internal data structure = array of size 256
  $\Rightarrow$ test for size = 255 and 257 (near boundary)
- Internal structure include design specs (like diagram sequence)
- Derive test cases to maximize structure coverage, yet minimize # of test cases

Coverage criteria: Path testing

- every statement at least once
- all portions of control flow (= branches) at least once
- all possible values of compound conditions at least once (condition coverage)
  Multiple condition coverage $\rightsquigarrow$ all true/false combinations for all simple conditions
  Domain testing $\rightsquigarrow$ {a < b; a == b; a > b}
- all portions of data flow at least once
- all loops, iterated at least 0, once, and N times (loop testing)

Main issue: white box testing negates object encapsulation

Black Box Testing

Component $\equiv$ "black box"

Test cases derived from external specification

- Behavior only determined by studying inputs and outputs
- Derive tests to maximize coverage of spec elements yet minimizing # of tests

Coverage criteria

- All exceptions
- All data ranges (incl. invalid input) generating different classes of output
- All boundary values

Equivalence Partitioning

- For each input value, divide value domain in classes of equivalences:
  - Expects value within [0, 12] $\leadsto$ negative value, within range, above range
  - Expects fixed value $\leadsto$ below that value, expected, above
  - Expects value boolean $\leadsto$ {true, false}
- Pick a value in each equivalence class (randomly or at boundary)
- Predict output, derive test case

Sixth Chapter

Testing

Testing Strategies

Unit testing

$\rightsquigarrow$ Looks for errors in objects or subsystems

Integration testing

$\rightsquigarrow$ Find errors with connecting subsystems together

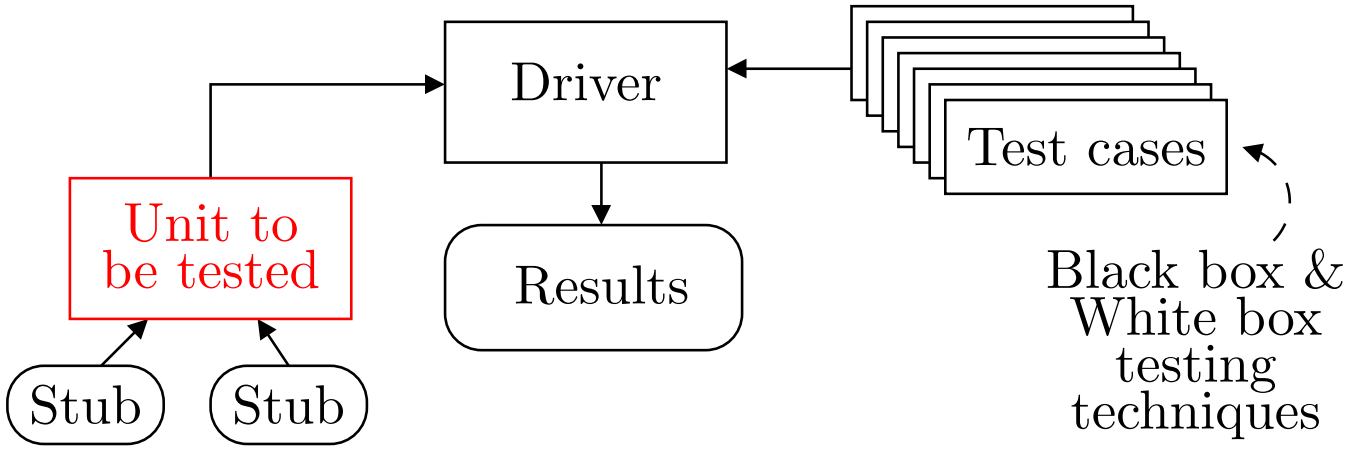
- System structure testing: integration testing all parts of system together

System testing

$\rightsquigarrow$ Test entire system behavior as a whole, wrt use cases and requirements

- functional testing: test whether system meets requirements
- performance testing: nonfunctional requirements, design goals
- acceptance testing: done by client

Unit Testing



Why?

- Locate small errors (= within a unit) fast

Who?

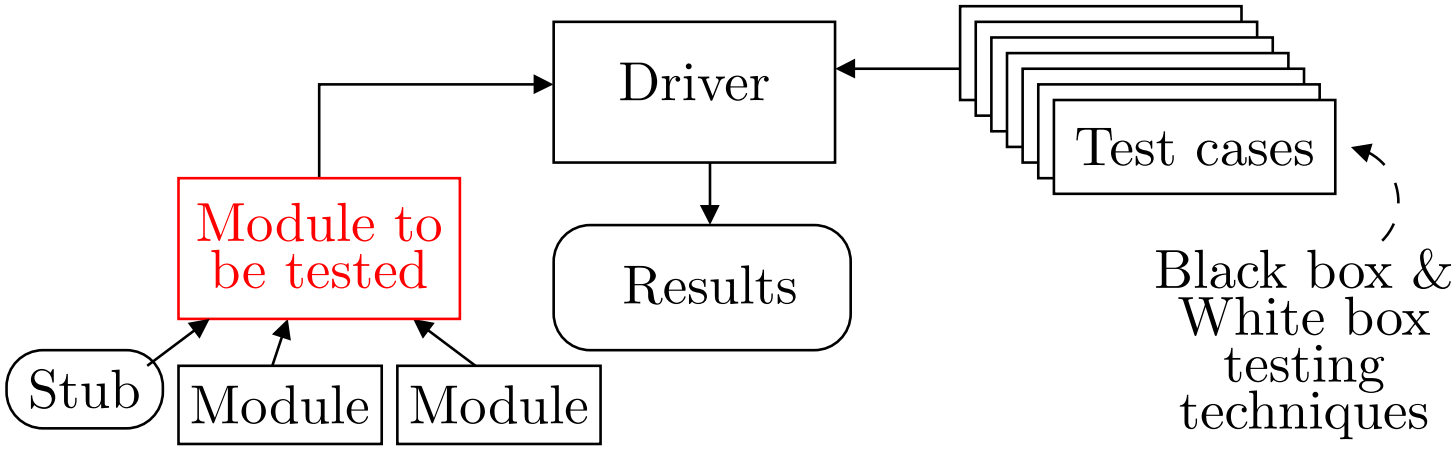
- Person developing the unit writes the tests

When?

- At the latest when a unit is delivered to the rest of the team
  - No test $\Rightarrow$ no unit
- Write the test first, i.e. before writing the unit

  $\Rightarrow$ help to design the interface right

Integration Testing



Why?

- Sum is more than parts, interface may contain faults too

Who?

- Person developing the module writes the tests

When?

- Top-down: main module before constituting modules
- Bottom-up: constituting modules before main module
- In practice: a bit of both

Remark: Distinction between unit testing and integration testing not that sharp

Regression Testing

Ensure that things that used to work still work after changes

Regression test

- Re-execution of tests to ensure that changes have no unintended side effects
- Tests must avoid regression (= degradation of results)
- Regression tests must be repeated *often*
  (after every change, every night, with each new unit, with each fix,...)
- Regression tests may be conducted manually
  - Execution of crucial scenarios with verification of results
  - Manual test process is slow and cumbersome
    $\Rightarrow$ preferably completely automated

Advantages

- Helps during iterative and incremental development + during maintenance

Disadvantage

- Up front investment in maintainability is difficult to sell to the customer
- Takes a lot of work: more test code than production code

Acceptance Testing

Acceptance Tests

- conducted by the end-user (representatives)
- check whether requirements are correctly implemented
  borderline between verification ("Are we building the system right?")
  and validation ("Are we building the right system?")

Alpha- & Beta Tests

- Acceptance tests for "off-the-shelves" software (many unidentified users)
- Alpha Testing
  - end-users are invited at the developer's site
  - testing is done in a controlled environment
- Beta Testing
  - software is released to selected customers
  - testing is done in "real world" setting, without developers present

Other Testing Strategies

Recovery Testing

- Forces system to fail and checks whether it recovers properly

$\rightsquigarrow$ For fault tolerant systems

Stress Testing (Overload Testing): Tests extreme conditions

- e.g., supply input data twice as fast and check whether system fails

Performance Testing: Tests run-time performance of system

- e.g., time consumption, memory consumption
- first do it, then do it right, then do it fast

Back-to-Back Testing

- Compare test results from two different versions of the system

$\rightsquigarrow$ requires N-version programming or prototypes
`git` version control system does so to isolate regressions

Sixth Chapter

Testing

When to stop?

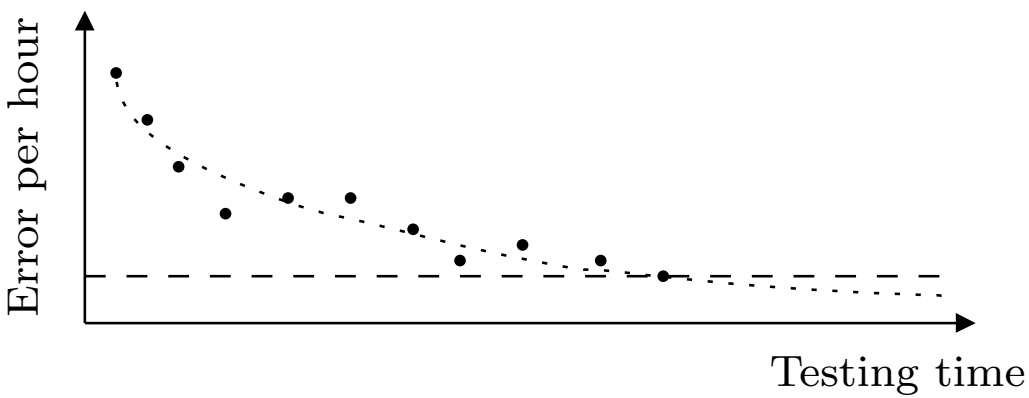
> *Testing can only prove the presence of defects, not their absence.*
> – E. W. Dijkstra

Cynical answer (sad but true)

- You're never done: each run of the system is a new test
  - $\Rightarrow$ Each bug-fix should be accompanied by a new regression test
- You're done when you are out of time/money
  - Include test in project plan and **do not give in to pressure**
  - ... in the long run, tests **save** time

Statistical testing

- Test until you've reduced failure rate under risk threshold



Why to test? (continued)

Because it helps ensuring that the system matches its specification

But not only (more good reason to test)

- Traceability
  - Tests helps tracing back from components to the requirements that caused their presence
- Maintainability
  - Regression tests verify that post-delivery changes do not break anything
- Understandability
  - Newcomers to the system can read the test code to understand what it does
  - Writing tests first encourage to make the interface really useable

Tool support

Test Harness

- Framework merging all test code in environment
- Main example for Java is called JUnit
- It inspired CppUnit, PyUnit, ...

Test Verifiers

- Measure test coverage for a set of test cases
- JCov for Java, gcov for gcc, ...

Other Techniques (somehow) Related to Testing

- Fuzzing: provide application *almost* correct data
  - Useful to ensure robustness to user- or network-provided data
  - If program fails, possible security issues (like buffer overflow)
- Model-Checking: formal method to test any execution path from given point
  - Save app. state at each branching, explore one branch, restore, explore other
  - Rarely doable without rewriting a model of the application
  - Can be seen as extensive testing

Sixth Chapter

Testing

Introduction

What is JUnit?

- It is a unit testing framework for Java.
- It provides tools for easy implementation of unit test plans
- It eases execution of tests
- It provides reports of test executions

What is NOT JUnit?

- It cannot design your test plan
- It does only what you tell it to
- It does not fix bugs for you

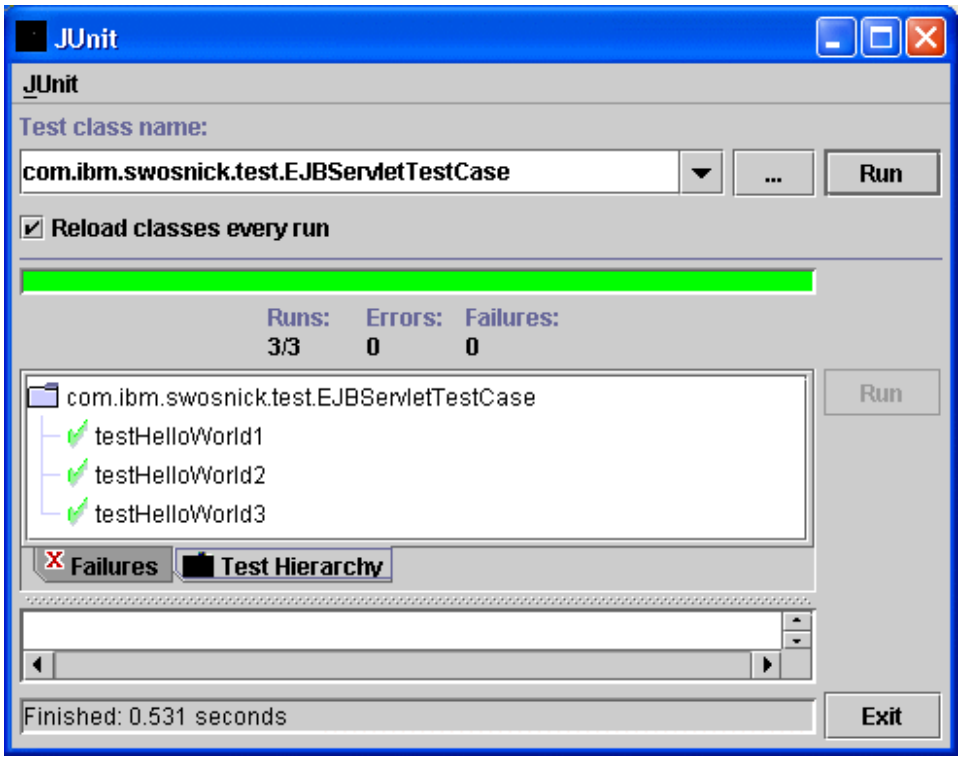
JUnit has two major versions

- JUnit 3.x: uses convention on method naming
- JUnit 4.x: uses Java 5 annotations

Structure of JUnit tests

Running a test suite consists of

- Setting up test environment
- For each test
  - Setting test up
  - Invoking test function
  - Tearing test down
- Tearing down everything
- Report result



Setting up test environment

Purpose

- Get things ready for testing.
- Create common instances, variables and data to use in tests.

Two kinds may co-exist

- Setting up before each test function
  - Named `public void setUp()` in JUnit 3.x
  - Annotated `@Before` in JUnit 4.x
- Setting up once for all
  - Placed in constructor in JUnit 3.x
  - Annotated `@BeforeClass` in JUnit 4.x

Cleaning test environment: Tearing down methods

Purpose

- Clean up after testing
- I.e., closing any files or connexions, etc
- Not used as often as setup methods

Two kinds may co-exist

- Tearing down after each test function
  - Named `public void tearDown()` in JUnit 3.x
  - Annotated `@After` in JUnit 4.x
- Tearing down once for all (JUnit 4.x only)
  - Annotated `@AfterClass`

Actually doing the tests

Test functions

- It is where the tests are performed
- Need one function per test case (which may call helper functions)
- Name must start with `test` in JUnit 3.x
- Annotated `@Test` (in JUnit 4.x)

Verifying results

- All tests are verified with assertions.
- JUnit comes with an Assert class for this purpose
  - public void assertTrue(String message, boolean condition)
  - public void assertNotNull(String message, Object obj)
  - public void assertEquals(String message, Object expected, Object actual)
  - public void assertSame(String message, Object expected, Object actual) uses ==, not .equals()
  - public void assertFalse(String message, boolean condition)
  - public void assertNotEquals(String message, Object expected, Object actual)
  - public void assertNotSame(String message, Object expected, Object actual)
  - public void fail(String message)

Example: Combination Lock (1/2)

Data and setting up

```
public class CombinationLockTest {
    // Locks with the specified combinations
    private CombinationLock lock00; // comb. 00
    private CombinationLock lock03; // comb. 03
    private CombinationLock lock12; // comb. 12
    private CombinationLock lock99; // comb. 99
    @Before
    public void setUp () {
        lock00 = new CombinationLock(0);
        lock03 = new CombinationLock(3);
        lock12 = new CombinationLock(12);
        lock99 = new CombinationLock(99);
    }
    ...
}
```

Tear down not necessary here

- object data will be deallocated automatically
- setup method overwrites instance variables

Example: Combination Lock (2/2)

Simple test method

```
@Test
public void testOpenLock () {
    lock12.enter(3);
    lock12.enter(4);
    assertTrue(lock12.isOpen());
}
```

Test method with helper

```
@Test
public void testFirstDigitTwice () {
    closeLocks();
    firstDigitTwice(lock03,0,3);
    firstDigitTwice(lock12,1,2);
}
private void firstDigitTwice(CombinationLock lock, int first, int second) {
    lock.enter(first);
    lock.enter(first);
    assertFalse(lock.isOpen());
    lock.enter(second);
    assertTrue(lock.isOpen());
}
```

Sixth Chapter

Testing

Introduction

Design by Contract

- Programming methodology trying to prevent code to diverge from specs
- Mistakes are possible
  - while transforming requirements into a system
  - while system is changed during maintenance

What's the difference with Testing?

- Testing tries to diagnose (and cure) errors after the facts
- Design by Contract tries to prevent certain types of errors

Design by Contract is particularly useful in an Object-Oriented context

- preventing errors in interfaces between classes
  incl. subclass and superclass via subcontracting
- preventing errors while reusing classes
  incl. evolving systems, thus incremental and iterative development
  Example of the Ariane 5 crash

Use Design by Contract in combination with Testing!

What is Design By Contract?

> *View the relationship between two classes as a formal agreement, expressing each party's rights and obligations.* – Bertrand Meyer

Example: Airline Reservation

	Obligations	Rights
Customer	▶ Be at Paris airport at least 3 hour before scheduled departure time ▶ Bring acceptable baggage ▶ Pay ticket price	▶ Reach Los Angeles
Airline	▶ Bring customer to Los Angeles	▶ No need to carry passeger who is late ▶ has unacceptable baggage ▶ or has not paid ticket

- Each party expects benefits (rights) and accepts obligations
- Usually, one party's benefits are the other party's obligations
- Contract is declarative: it is described so that both parties can understand *what* service will be guaranteed without saying *how*

Connecting back to Hoare logic

Pre- and Post-conditions + Invariants

- Obligations are expressed via pre- and post-conditions

> *If you promise to call me with the precondition satisfied, then I, in return promise to deliver a final state in which the postcondition is satisfied.*

pre-condition: $x >= 9$ post-condition: $x >= 13$

component: $x := x + 5$

- and invariants

> *For all calls you make to me, I will make sure the invariant remains satisfied.*

Isn't this pure documentation?

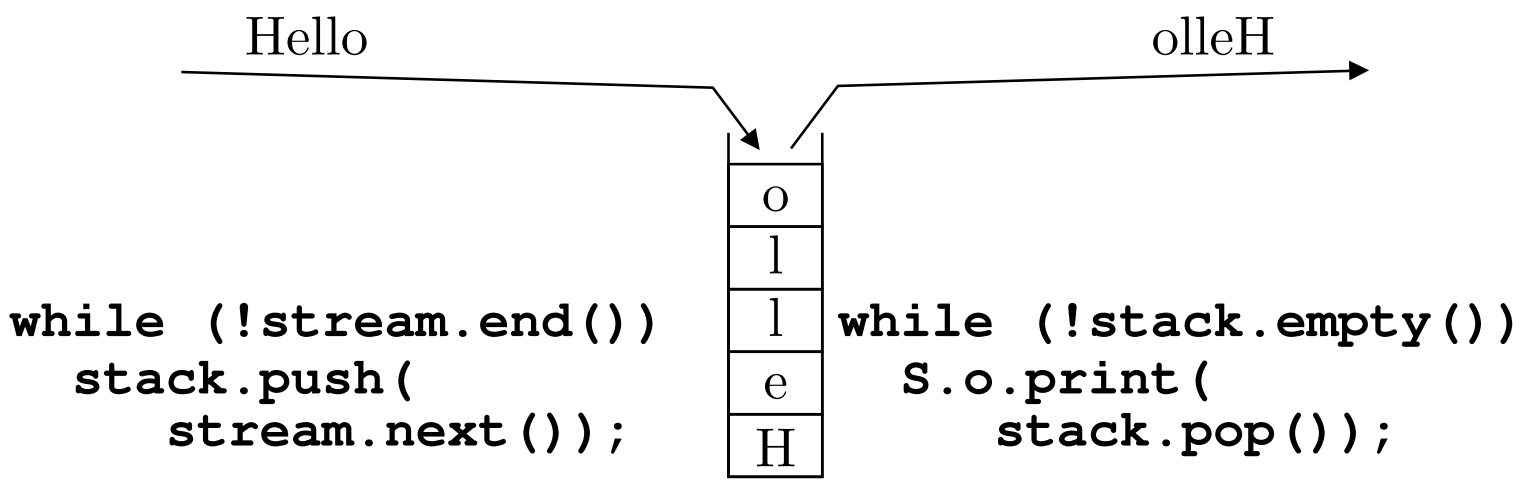
(a) Who will register these contracts for later reference (the notary)?
The source code

(b) Who will verify that the parties satisfy their contracts (the lawyers)?
The running system

Example: Stack

Specification

- Given
  - A stream of characters, length unknown
- Requested
  - Produce a stream containing the same characters but in reverse order
  - Specify the necessary intermediate abstract data structure



Example: Stack Specification

```
class stack
  invariant: (isEmpty (this)) or (! isEmpty (this))
  /* Implementors promise that invariant holds after all methods return
     (incl. constructors)*/
  public char pop ()
     require: !isEmpty(this)              /* Clients' promise (precondition) */
     ensure: true                         /* Implementors' promise (postcondition)
                                             Here: nothing */
  public void push(char)
     require: true
     ensure: (!isEmpty(this))             /* Implementors' promise:
              and (top(this)==char)          Matches specification */
  public void top (char) : char
     require: ...                         /* left as an exercise */
     ensure: ...
  public void isEmpty() : boolean
     require: ...
     ensure: ...
```

Defensive Programming

Redundant checks

- Redundant checks are the naive way for including contracts in the source code

```
public char pop () {
  if (isEmpty (this)) {
    ... //Error-handling
  } else {
    ...}
```

This is redundant code: it is the responsibility of the client to ensure the pre-condition!

Redundant Checks Considered Harmful

- Extra complexity
  due to extra (possibly duplicated) code ... which must be verified as well
- Performance penalty
  Redundant checks cost extra execution time
- Wrong context
  - How severe is the fault? How to rectify the situation?
  - A service provider cannot asses the situation, only the consumer can.
  - Again: What happens if the precondition is not satisfied?

Assertions

Any boolean expression we expect to be true at some point

Advantages

- Help in writing correct software (formalizing invariants, and pre/post-conditions)
- Aid in maintenance of documentation (specifying contracts in the source code)
  $\Rightarrow$ tools to extract interfaces and contracts from source code
- Serve as test coverage criterion (Generate test cases that falsify assertions)
- Should be configured at compile-time (to avoid performance penalties in prod)

What happens if the precondition is not satisfied?

- When an assertion does not hold, throw an exception

Assertions in Programming Languages

Eiffel

- Eiffel is designed as such ... but only used for correction (not documentation)

C++

- assert.h does not throw an exception, but close program
- Possible to mimick. Documentation extraction rather difficult

Smalltalk

- Easy to mimic, but compilation option requires some language idioms
- Documentation extraction is possible (style JavaDoc)

Java

- Assert is standard since Java 1.4 ... very limited
- JML provide a mechanism ... but not ported to Java 5 (damn genericity)
- Modern Jass seems very promising, but needs more polishing

Design by Contract vs. Testing

They serve the same purpose

- Design by contract *prevents* errors; Testing *detect* errors

$\rightsquigarrow$ One of them should be sufficient!

They are complementary

None of the two guarantee correctness ... but the sum is more than the parts

- Testing detects wide range of coding mistakes
  ... design by contract prevents specific mistakes due to incorrect assumption
- Design by contract ease black box testing by formalizing spec
- Condition testing verify whether all assertions are satisfied
  (whether parties satisfy their obligations)

Sixth Chapter

Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Conclusion on Testing
- JUnit
- Design By Contract

Bibliography for this chapter

- Lectures on Software Engineering by Serge Demeyer (U. Antwerpen)
  `http://www.lore.ua.ac.be/Teaching/SE3BAC/`
- Lecture on JUnit by Dirk Hasselbalch (U. Copenhagen)
  `http://isis.ku.dk/kurser/blob.aspx?feltid=217458`
- Test Infected: Programmers Love Writing Tests (Tutorial on JUnit)
  `http://www.cril.univ-artois.fr/~leberre/MI32001/TESTING/junit3.7/doc/testinfected/ntesting_fr.htm`