0.1 Mission Statement

This is our personal collection of some important ideas in computer science and programming. It shall help us document high-level insights that can serve us in practice. We do not try to capture topics in-depth, but instead just try to make us aware, so that we know what to look for when we are in need. We are aware that when everything is important, nothing is important.

We also try to document some of our occasional nuggets of temporary insights.

0.2 Main Topics

- Nothing substantial yet

0.3 Sources to review

- Study notes
  - Compiler Construction
  - Experimental Economics: Design Elements (e.g., basline neighborhood) and the basic idea of falsification as supplement material for the algorithm engineering notes
  - Formal Systems (something in there?)
  - Advanced Data Structures
  - Parallel Algorithms
  - Parallel Machines and Parallel Programming
  - Randomized Algorithms
  - Computerarchitecture (implications of branch predictions, cache coherece, pipelining, super scalar architecutures . . .)
  - Softwareengineering II
  - Game Theory (Battle of the sexes :p)
- Lecture notes and books to skim
  - Algorithm Engineering
  - Algorithm II
  - Algorithms and Data Structures — The Basic Toolbox
  - Linder's *Things Thy Should have taught you*
- Books with potential:
  - Pragmatic Thinking and Learning
  - Notes from the Pragmatic Programmer

– Notes from Head First Software Development (Very light stuff but maybe there is something in there. I used to enjoy reading it.)

– ...

1 Desigining Abstractions

- Single Level of Abstraction
- Single Responsibility Principle
- Separation of Concerns
- Interface Segragation Principle
- Liskov Substitution Principle
- Open Closed Principle
- Law of Demeter

If I am not totally mistaken, these principles are well described within SICP.

2 Algorithm Design Toolbox

2.1 Randomized Algorithms

- Among others, randomized algorithms help to improve robustness concerning worst-case inputs (e.g., think of the pivot selection problem in `quicksort` when the sequence is provided by a malicious adversary).
- Allowing randomized algorithms to compute a *wrong* result (with a very low probability) can open many new possibilities concerning speed, space, quality and ease of implementation (e.g., think of `bloom filters`).
- Never use the `C rand()` function. If in doubt, use *Mersenne Twister*.
- Use random numbers with care. Treat them as a scarce resource.
- In certain parallel setups, *expected* bounds of randomized algorithms do no longer hold: Consider `n` processes that call operations with *expected* runtime bounds and that have to be synchronized before and afterwards. The runtime will suffer whenever at least one of the processes hits an expensive case.

3 Datastructure Toolbox

3.1 Lists

3.2 Arrays

3.3 Associative Arrays

4 Personal Development

- Always life on the edge of incompetence. When ever you have the chance, approach challanges beyond your current skill level. Be focused, determined and passioned and you will be invincible, because even if you fail (what you won't), you have learned a great deal.
- There is no speed limit in life. You don't have to move (improve, learn, ...) as slowly and overcautiously as others.
- Never presume to already know. Have a beginner's mind.
- Awareness trumps trying.
- Act or accept there is nothing in between.
- "Money is nothing more than neutral proof that you're adding value to people's lives. Making sure you're making money is just a way of making sure you're doing something of value to others." (Derek Sivers)

4.1 Studying

- Approach study tasks like you approach programming taks (see the common principles)
- Do not expect teachers to teach you (they only present information). Nobody will teach you anything. You have to teach yourself.
- Studying is not about time, it's about effort. You have to approach a difficult topics from different angles and discuss it with peers. Make the ideas your own by discovering them for yourself (e.g., with your own examples). Stupid repetition just does not cut it. If you have to memorize something, you haven't understood it.
- Don't miss the big picture by getting lost in all the details. Always zoom-out again and fish for connections and insights.

- Always write down key insights as succinctly and precicely as possible. Feel the pain. You have to do it, because it is a different thing to believe to understand something and to be able to articulate it.
- If stuck with anything, ask yourself *what exactly_* is it, that you don't understand? (See the notes on debugging).

4.2 Stress & Burnout Prehab

- Stress kills creativity and gives you a bad hangover. Don't get back to work just because you are bored. Once your creativity returns (with ideas on cool projects, the strong desire to read one the countless unread books on your shelf, people you really want to meet again, ...) you will feel more energetic than ever and can re-approach your life (and work) with a whole new mind.

4.3 Mentoring & Teaching

- Challenge the student by constantly asking questions. Let the student prove they got it. Set a high pace.

5 Sources

- There's no speed limit and [6 things I wish I knew the day I started Berklee] (http://sivers.org/berklee) by Derek Sivers

6 Programming

6.1 Common Principles

- Think hard before you start.
- Focus. Eliminate distractions and obstacles. If you cannot stay within *the flow*, you are doing something wrong.
- *Make it*, *make it run* (so that the tests pass), *make it better* (review and refactor until you can honestly say that it is good enough)
- If in doubt, leave it out (abstractions, features, ...), because *you ain't gonna need it (YAGNI)!* You can always come back later. Don't waste your time now.

6.2 Object-oriented Programming

- Using an OO-language does not imply you *'got it'* and are doing it right.
- If nothing else, do atleast try to follow the *tell, don't ask* principle. ## Testing
- Look for *inflection points* as the sweet spot between testing effort and covered functionality.

6.3 Debugging

- Again, think before you start. Relax, breath and fetch your favorite beverage. Find the bug in your head. Only then start the debugger, if at all!
- Debugging is not a reason to stay overtime. Document your thought process so that you know where to pick up the next day.
- If stuck, describe your problem to an (imaginary) colleague.
- Dig until you have found the root cause. If you are inclined to just add a `null` check, you haven't found it yet.