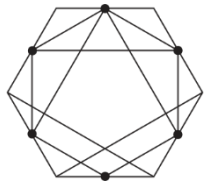
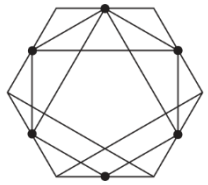
# TEST DRIVEN DEVELOPMENT

And More



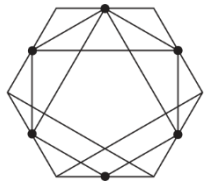
# MAIN IDEA BEHIND TDD

- Use tests to **drive design** and **feature development**
- Each test describes a feature increment of the program
- Conversely, each feature increment is first described by a test
- Testable and well-tested code is an inevitable byproduct of this process

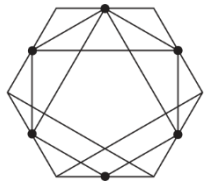


# PROBLEM

How can tests drive the design of a program?



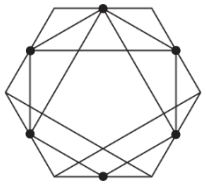
# Refactoring



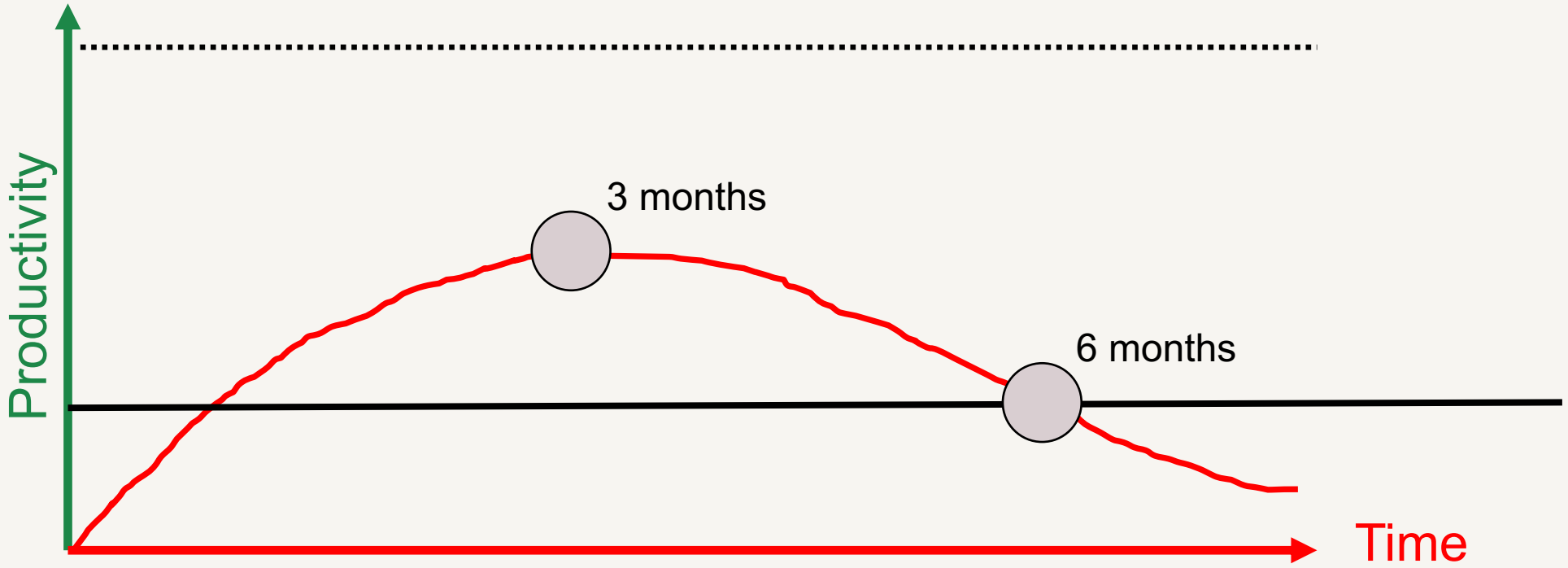
# REFACTORING

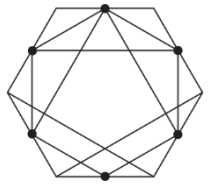
- Improve the design of the program in small increments
- The correctness of these steps is ensured by our test suite

**Small increments are important!**



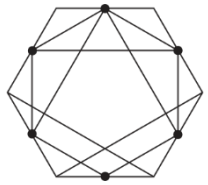
# SO WHAT?





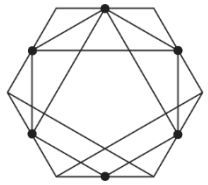
# TEST-DRIVEN DEVELOPMENT

- High test coverage is not the goal of TDD
  (No tests for methods we are convinced cannot fail, such as getters/setters)
- Main goal is to discover a good design
  - By writing tests first we can design the interface of our class so that tests are easy to write
  - This makes production code easy to write
  - Since we have tests, we can refactor without danger of introducing errors



# THE TDD CYCLE

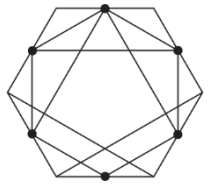
- Write a (minimal) test
- Implement the minimal functionality to get the test working
- Improve the code



# THE TDD CYCLE

- **Write a (minimal) test**
- Implement the minimal functionality to get the test working
- Improve the code

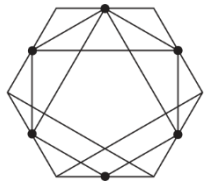
- Test only a single feature: **Baby Steps**
- This test is **supposed to fail**



# THE TDD CYCLE

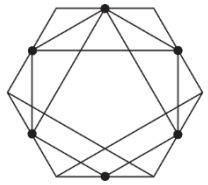
- Write a (minimal) test
- **Implement the minimal functionality to get the test working**
- Improve the code

- This code does **not** have to be clean or well-designed
- **Solve simply!**



# THE TDD CYCLE

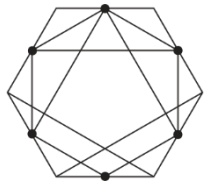
- Write a (minimal) test
- Implement the minimal functionality to get the test working
- **Improve the code**
  - Clean up the code introduced in the previous step
  - Generalize the implementation if it contains too much repetition
  - **This step is not optional!**



# THE TDD CYCLE

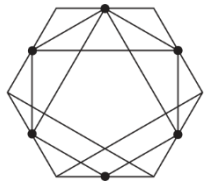
- Write a (minimal) test
- Implement the minimal functionality to get the test working
- **Improve the code**

**Improving the code really isn’t optional!**



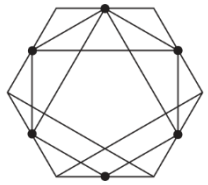
# THE TDD CYCLE

- Red: Failing test
- Green: All tests are green again, code is not clean
- Clean / refactor: All test still pass, code is clean again



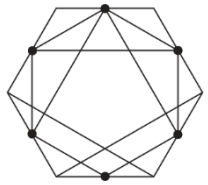
# WHY SOLVE SIMPLY?

- A flexible, generic solution often increases the complexity of the system
- This is useless if the flexibility is not needed
- Developers are often bad at predicting where flexibility or extensibility is needed
- A simple solution for a specific use case is often simple to implement
- This simple solution is often easier to understand and cleaner than generic code



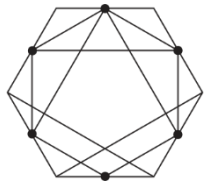
# ASSUMPTIONS MADE BY SOLVE SIMPLY

- Refactoring allows us to clean up code without changing its functionality
- It is possible to iteratively extend code and to add the necessary flexibility to do so
- It is simpler to iteratively refactor and extend code than to come up with the final solution right away
- These assumptions are not fulfilled if we don’t have a good test harness!



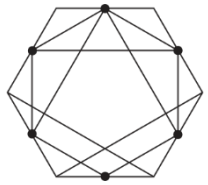
# BABY STEPS

- The system is almost always in a buildable, executable state
- Therefore we get immediate feedback from every code change
- Frequently merging to master and CI/CD become possible



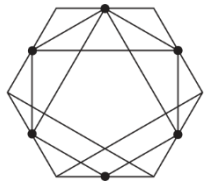
# EVEN BETTER: TDD CYCLE + PREPARATION

- Refactor the code so that the feature increment will be easy to add
  - This is often not simple!
  - But you have a working test suite while doing so
- Implement the feature increment in a TDD cycle
- Repeat this cycle *ad infinitum* (or until the program is completed)



# GUIDED KATA: PRIME FACTORIZATION

- TDD exercise that demonstrate how to come up with a clean implementation of an algorithm by following the TDD steps
- Focus is proceeding according to the TDD cycle: Have tests drive the design
- Goals:
  - Experience the TDD cycle
  - Learn to work iteratively and incrementally



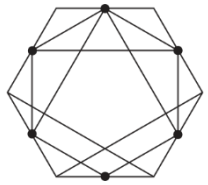
# GUIDED KATA: PRIME FACTORIZATION

Write a function

```
std::vector<long> compute_prime_factors(long n)
```

that returns the prime factors of n in ascending order. Multiple prime divisors occur multiple times in the result.

For example, `compute_prime_factors(12)` returns the vector {2, 2, 3}.



# KATA: FIZZBUZZ

Write a program that simulates a simple game for children:

- The program outputs the numbers from 1 to 100 to the standard output
- If a number is divisible by 3, it is replaced by the string “Fizz”
- If a number is divisible by 5, it is replaced by the string “Buzz”
- If a number is divisible by both 3 and 5 it is replace by the string “FizzBuzz”

Work incrementally, in small steps.

*Hint: How can you structure your code to make it easier to test?*

# Recap: Effective Unit Testing

- Unit tests must be **effective** and **efficient**
  - Effective: Find as many mistakes as possible (ideally: all of them)
  - Efficiency: Use as few and as simple tests as possible to achieve this
- Fundamentals
  - Controllability
  - Observability
  - Size
- Rules of thumb
  - Test functionality, not implementation
  - Prefer values over state
  - Prefer state over behavior
  - Test small units
  - Use test doubles if *and only if* the real dependency launches a missile

  - (Your code has to allow tests to be written that way)

# Workshops

- https://github.com/hoelzl/StackCpp
  - Simple integer stack:
  - Push(), Pop() methods
  - Extensions
- https://github.com/hoelzl/Employee
  - Employee class:
  - Difficult to test (despite its small size)
  - How can we improve testability/tests