Lab Course / “Praktikum”:
*Project Management and Software Development for Medical Applications*

Documentation, Tests, Design Patterns & Integration Strategy – SS2021

Ardit Ramadani
Disclaimer

- This talk will not cover all aspects of SE!
- Familiarize with concepts and ideas
- Not every single detail matters
Software Engineering approaches

- Sometimes it is applied rigidly
- Many different contrasting ideas
- Do not get your attention drawn away from the problem at hand!
How Software Design and Engineering really works..
Keep the problem as small as possible!
How Software Design and Engineering really works..
Documentation
Documentation for developers

This includes:

• Your customers
• Your team
• Yourself!
Documentation for developers – Code style

• Code is written once, but read many more times

• Don’t be lazy:
  – Good variable names
  – Refactor code
  – Keep modular and generic
Documentation for developers – Comments

• No trivial comments
• Explain:
  – Assumptions
  – Corner cases
  – Non-trivial use of language features

BAD:

//Apply style.
apply(style);

GOOD:

// Unlike the others, this image needs to be drawn in the user-requested style
apply(style);
Documentation for developers – Doxygen

• Creates static docs from comments
• Close to source code, so USUALLY less out-of-date
• Useful only with non-trivial content

```cpp
class Time {

public:

 /**
  * Constructor that sets the time to a given value.
  *
  * @param timemillis Number of milliseconds passed since Jan 1, 1970.
  */
  Time (int timemillis) {
    // the code
  }

```
Documentation for developers – Doxygen

Time Class Reference

List of all members.

Public Member Functions

Time (int timemillis)

Static Public Member Functions

Time now ()

Detailed Description

The time class represents a moment of time.
Documentation for users

- Users as seen by developers:
  - Usually the cause is bad documentation!
  - You make a lot of assumptions that are clear in your head, but not to a new user.
Design Patterns (and anti-Patterns)
Design Patterns

- Reusable code structures
- Solve common problems
- Proven to work, common vocabulary
- Mostly created to work around rigid Object-Oriented type systems
- **BUT:** focus on the problem rather than where to stuff them in your program!
Some design Patterns

- **Singleton**: class with only one instance in whole program
- **Abstract factory**: allows to create an instance of several families of classes
- **Observer**: way of notifying change to a number of classes
- **Decorator**: add functionality to class without inheriting
- **Facade**: single class that represents an entire subsystem
Design anti-Patterns

• Too many classes

• Functions too long

```python
img_filter = ImageFilter()
img_filter.set_image(img)
img_filter.set_radius(2.5)
filtered_img = img_filter.get_output()

→ filtered_img = filter_img(img, radius=2.5)
```
Design anti-Patterns

• Too many classes
• Functions too long
• Mixed functionality
• Reinventing the wheel
• Premature optimization
Testing
Testing – Definitions

- Verification and Validation (V&V)
  - Verification: The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of the phase [IEEE-STD-610]
  - Validation: The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements [IEEE-STD-610]
# Testing – Definitions

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Verification</th>
<th>Validation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Definition</strong></td>
<td>The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase.</td>
<td>The process of evaluating software during or at the end of the development process to determine whether it satisfies specified business requirements.</td>
</tr>
<tr>
<td><strong>Objective</strong></td>
<td>To ensure that the product is being built according to the requirements and design specifications. In other words, to ensure that work products meet their specified requirements.</td>
<td>To ensure that the product actually meets the user’s needs, and that the specifications were correct in the first place. In other words, to demonstrate that the product fulfills its intended use when placed in its intended environment.</td>
</tr>
<tr>
<td><strong>Question</strong></td>
<td>Are we building the product <em>right</em>?</td>
<td>Are we building the <em>right</em> product?</td>
</tr>
<tr>
<td><strong>Evaluation Items</strong></td>
<td>Plans, Requirement Specs, Design Specs, Code, Test Cases</td>
<td>The actual product/software.</td>
</tr>
</tbody>
</table>
| **Activities** | • Reviews  
• Walkthroughs  
• Inspections | • Testing |
Test types

- **Runtime Test**: Sanity check for invalid program states during runtime
- **Test Run**: Developer runs the software and looks for obvious errors
- **Systematic Test**: Carefully chosen test data, comparison with expected results
- **Regression Test**: Extended and automated systematic test, run repeatedly (e.g. after every commit), test results are documented
- **Performance Test**: Testing performance of the software (runtime, memory usage, ...)

Testing may be a pain in the neck, but with the right combination of the above test types you get a good cost-return value
Test levels

- **Unit Test**: Checks a single piece of code (e.g. class) in isolation
- **Integration Test**: Verifies the interfaces between components
- **System Test**: Checks that the whole software meets the requirements
- **Operational Acceptance Test**: Put the software to test with real end users and in realistic conditions
import unittest

def fun(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(fun(3), 4)
Test Driven Development

- Write tests first, then develop until pass

Pros:
- Help focusing on objectives
- Think about corner cases
- More rewarding experience
- More confident about later changes
Testable code

- Keep functions small

```python
def add_to_cart(user, article):
    price = database.get_article(article)
    if user.age > 35 and article.category == 'food':
        price *= 0.90
    elif user.city == 'Munich' and article.category == 'electronics':
        price *= 0.85
    database.reduce_availability(article)
    user.add_to_cart(article, price)

def compute_price(user, price, article):
    if user.age > 35 and article.category == 'food':
        price *= 0.90
    elif user.city == 'Munich' and article.category == 'electronics':
        price *= 0.85
    return price

def add_to_cart(user, article):
    price = database.get_article(article)
    price = compute_price(user, price, article)
    database.reduce_availability(article)
    user.add_to_cart(article, price)
```
Bug tracker

- Help tracking defects present in software
Bug tracker
The Big-Bang Integration Strategy

• Unordered implementation of the components / all components implemented at the same time

• Problems
  – Errors are very hard to locate: Which component is the cause?
  – Design errors (errors in interfaces) not distinguishable from implementation errors

• Always prefer incremental integration strategy
Top-Down Integration Strategy

• **Start with the components from the top-most layer (e.g. GUI). Incrementally add layers further down**

• **Pros/Cons**
  – Early prototype available (with limited functionality)
  – Design errors can be detected in an early state
  – Many stubs required → cumbersome
  – No functionality until a very late stage
Bottom-Up Integration Strategy

- *Start with the components from the bottom-most layer (e.g. entity classes). Incrementally add upper layers.*

- **Pros/Cons**
  - No stubs required
  - Functionality available in early stages
  - Nothing to show to customers until the very end
  - Errors may be expensive, because they may be found late and solving them might require cumbersome changes
Version control
Version Control Systems

- Keep a history of changes to code
- Share code with others
- Integrate changes from others
- Manage concurrent versions
Version history
Changes history

2d4e9953 = streeter
2013-07-18 Add a missing quote so copy/..

989e48f7 = nickhammond
2013-05-18 Specify underscore & undersc...

3406d66b = technicalpickles
2012-06-08 Update "w" help comments

97d634da = h3h
2011-11-09 Add a Wikipedia script for p_

97d634da = h3h
2011-11-09 Add a Wikipedia script for p_

module.exports = (robot) ->
robot.respond /(wiki)( me)? (.*)/i, (msg) ->

wikiMe robot, msg.match[3], (text, url) ->

msg.send text
Branches
Centralized vs Distributed Version Control Systems

Centralized Version Control System:
- Central VCS Server
  - Version Database
    - Version 3
    - Version 2
    - Version 1
- Computer A
  - File
- Computer B
  - File

Distributed Version Control System:
- Server Computer
  - Version Database
    - Version 3
    - Version 2
    - Version 1
- Computer A
  - File
- Computer B
  - File
  - Version Database
    - Version 3
    - Version 2
    - Version 1
Centralized vs Distributed Version Control Systems

Continuous Integration

- Compile automatically on every change uploaded to VCS
Thank you

Happy coding

Ardit Ramadani, M.Sc.
Research Assistant
Deutsches Herzzentrum München des Freistaates Bayern
Klinik an der Technischen Universität München
Lazarettstr. 36
80636 München

Technische Universität München
Fakultät für Informatik - Informatik 116
Chair of Computer Aided Medical Procedures and Augmented Reality
Boltzmannstr. 3
85748 Garching bei München

campar.in.tum.de/Main/ArditRamadani
ardit.ramadani@tum.de
ramadani@dhm.mhn.de