Design Patterns and Test Driven Development

Jörg Faschingbauer

www.faschingbauer.co.at

jf@faschingbauer.co.at

Table of Contents

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

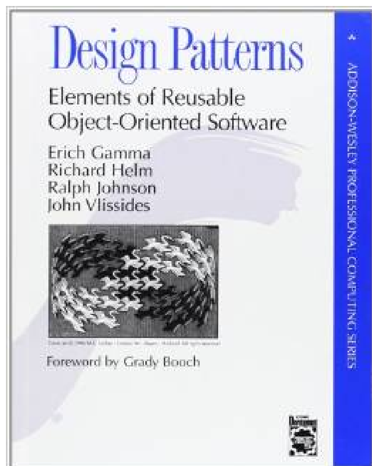
Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

The Book on Patterns: *Gang of Four (GoF)*

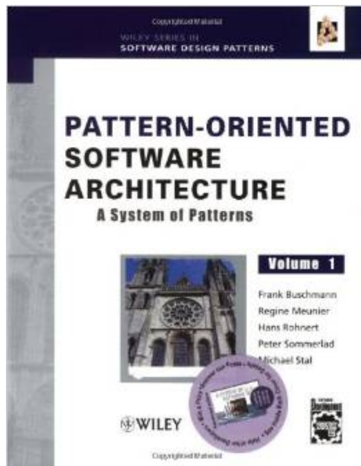
FASCHINGBAUER

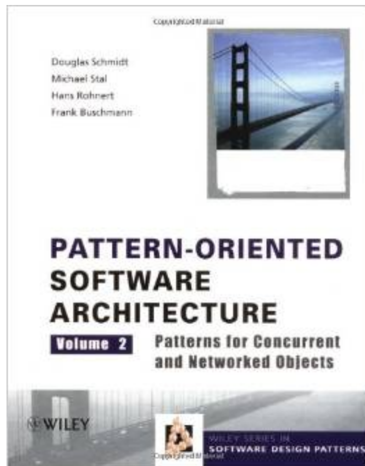


- Groundbreaking in 1995, until today
- Collection of then-existing patterns
- ... only giving them names
- Concise and to the point
- Well-structured
- (Relatively) easy to read
  - ... provided you understand the problems

The Other Books on Patterns

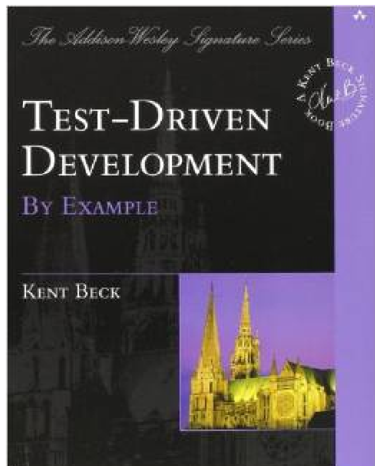
FASCHINGBAUER





The Book on Test Driven Development

FASCHINGBAUER



- Groundbreaking in 2003
- Revolutionary though simple (has only 200 pages)
- "New" methodologies
  - Test-first development
  - Refactoring, guided by automatic tests
  - ...
- Basis for all *agile* software development processes

Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Design Patterns: What?

FASCHINGBAUER

What is a design pattern?

- A solution to a design problem
  - *Beware*: there is no solution without a problem
- "Design" means *Object Oriented Design*
  - Inheritance and polymorphism aren't patterns — at least not in OO
- There are many different problems
  - *Object creation*: objects are not always created directly
  - *Structure*: who knows who, and what does he look like?
  - *Behavior*: how do my objects talk to each other?

Design Patterns: Why?

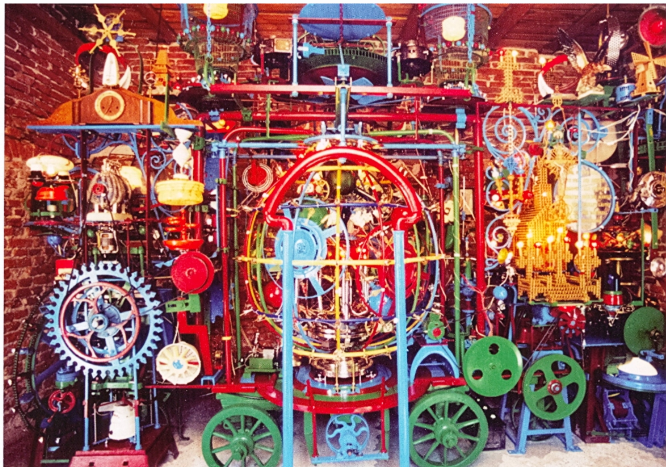
Why use design patterns?

- Code is a solution to a problem
  - Solution/code needs to be readable and understandable
  - Solutions to similar problems tend to be similar
  - ... at least, should!
- Design patterns ...
  - Give names to solutions → important in communication
  - Encourage solution similarity
  - Are well understood → documentation need only give the pattern's name
  - Solutions become obvious

Non-Obvious Problem

FASCHINGBAUER

Gsellmann's Weltmaschine



Non-Obvious Solution

FASCHINGBAUER

Design Patterns: Caveats

FASCHINGBAUER

Design patterns are no silver bullet

- *Overengineering*: artificial/unnecessary code complexity
  - Solution without a problem
  - Not easy to understand — not at all obvious what's being solved
  - One of the biggest mistakes in software design
  - It's like the pest
- Pattern usage does not automatically ensure sound OO design

What is sound design?

- Nobody knows
- ... but fortunately there is *Test Driven Development*

Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Test Driven Development

FASCHINGBAUER

A simple idea ... but first the problem ...

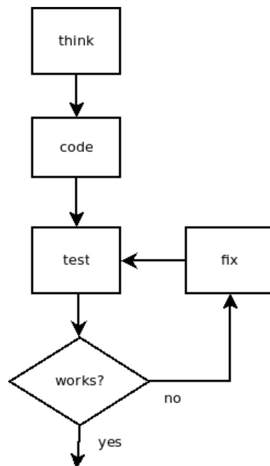
- New code is written and tested since ages
  - Bugs are fixed until it works
  - Testing mainly done manually
  - Standalone test programs, or ...
  - ... mostly the entire target application
- Existing code breaks once it is modified (law of nature)
  - Breakage not easily detected
  - *Fear!*
  - $\Longrightarrow$ nobody ever modifies existing code
  - $\Longrightarrow$ software starts to rot once it has been written

Development — Traditional Approach

FASCHINGBAUER

Traditional Approach

- Think about the design
- Come up with a decision
- Code it
- See if it works
- Fix
- (etc.)



Traditional Approach — Problems

FASCHINGBAUER

So what are the core problems?

- Before a modification ...
  - How do I know my solution will be ok?
  - How will it feel? Will it be usable?
  - Am I (and others) comfortable with it?
- After a modification ...
  - It is impossible to decide if everything still works
  - What is the definition of *everything*?
  - What is the definition of *works*?
  - What are the costs to decide that?
  - *What are the costs if we do only manual testing*?
  - *What is the state of the code? What about refactoring?*
- After the release ...
  - We curse at the testers that they do a bad job!

Test Driven Development — Principles (1)

FASCHINGBAUER

What if we were able to test everything automatically?

- Modifications could be done *without any fear*
  - *"Regression"*: new term for that kind of bug
  - Something that worked before a modification but doesn't afterwards
- Ongoing refactoring possible → no code smells
- New features would bring new tests
  - The *Everything* grows over time

But: the Everything is now defined as ...

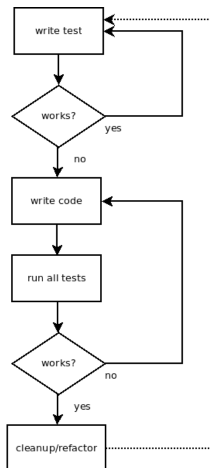
- Production code
- Test code

Test Driven Development — Principles (2)

FASCHINGBAUER

Test Driven Development

- New “development process”
- Tests come first
- → “Requirements phase”
- Have you ever read a requirements document *after* coding was done?
- → Tests fail initially



Test Driven Development — Benefits? Caveats?

FASCHINGBAUER

What does it bring, what does it cost?

- More work initially — so much for sure
- Investment into the future
- More code can be done
- Not at all easy to convince people of it

Big caveat

- Tests belong to the code
- *No way* moving on without!
- $\Longrightarrow$ Have to take care of the tests

Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Overview

FASCHINGBAUER

1 Introduction
- Literature
- Design Patterns
- Test Driven Development

2 Test Driven Development
- xUnit — How it Works
- Test Driven Development

3 OO Basics
- Members and Methods
- Inheritance

4 OO Principles — SOLID
- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

5 Design Patterns

6 Creational Patterns
- Abstract Factory
- Singleton

7 Structural Patterns
- Adapter
- Bridge
- Composite
- Proxy

8 Behavioral Patterns
- Command
- Interpreter
- Observer
- Strategy
- Visitor

Origins

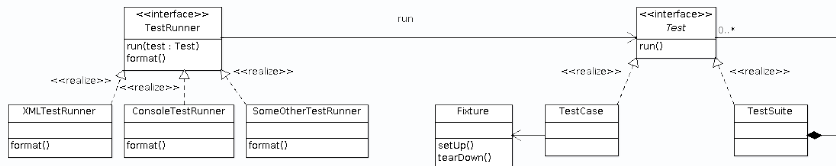
FASCHINGBAUER

Unittest frameworks — where they come from

- `SUnit`, 1998. By Kent Beck in Smalltalk.
- `JUnit`, 2001. Ported from Smalltalk to Java, by Kent Beck and Erich Gamma.
  - Gained wide popularity by Kent Beck's book
- From then on ported to almost every language — commonly known as `xUnit`
  - Python: `PyUnit`, then became part of the Python library, module `unittest`
  - C++: `Boost.Test`, `CppUnit`, Google Test, ...
  - All the newer languages: Ruby, Rust, Go, ...
  - COBOL

xUnit Structure — Overview

FASCHINGBAUER



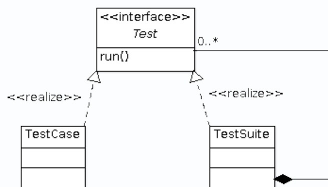
- TestCase: one test that is written. Here's the most code.
- TestSuite: composition of many test cases, for structural purposes.
- Fixture: defined environment of a TestCase
- TestRunner: runs a Test (Suite or Case), collects and presents results.

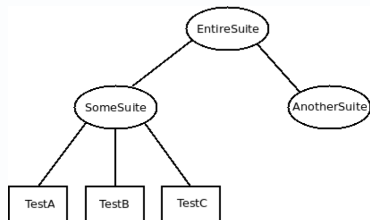
xUnit: TestCase and TestSuite

FASCHINGBAUER

Suites: recursive test structure

- Derive from `TestCase` to *implement* tests
- Use `TestSuite` objects to structure tests hierarchically
- Run a subset of all tests
- The *Composite Pattern* in use ...
- Not available in every xUnit incarnation



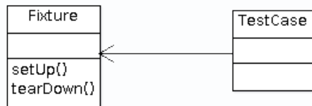


xUnit: TestCase and Fixture

FASCHINGBAUER

Fixture: defined test environment

- Multiple tests start from the same state → common *Fixture*
- Method `setUp()` — establishes known state to start tests from. Examples: well-known/required database content, files have to be present, ...
- Method `tearDown()` — deallocates resources. For example: cleanup database, remove files, ...



Implementation:

- Python: class that contains test methods
- C/C++: weird macros to setup objects and associations

xUnit: TestCase and *Assertions*

FASCHINGBAUER

Test code checks for failure: Assertions

- Varying multitude of assertions to draw from
- Records test failure in some test result, for later reporting
- Abort the test case → failure
- Variation: *non-fatal* assertions

```
container.insert(100)
container.insert(200)
self.assertEqual(len(container), 2)
```

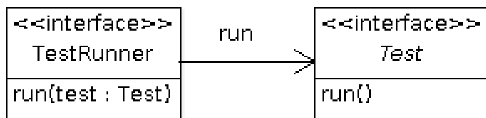
```
self.assertAlmostEqual(1/3, 0.333, 2)
```

xUnit: TestRunner

FASCHINGBAUER

Running all tests: `TestRunner`

- `TestRunner` usually instantiated in main programs
- During running a test ...
  - Fixtures are prepared (`setup()`, `tearDown()`)
  - Results are collected
  - Failure or success
- After all tests have run ...
  - The result has to be presented
- (Sidenote: do you know the *Strategy Pattern*?)



Overview

FASCHINGBAUER

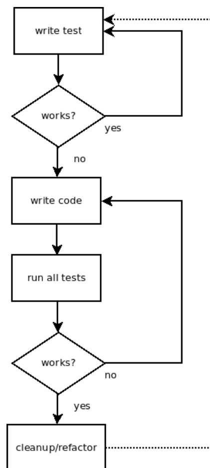
1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

The “Process”

FASCHINGBAUER

Test Driven Development is … well …

- Not a full process
- The basis of all “agile” processes
  - Anybody doing Scrum these days?
- It’s *Software done right*
- It’s about continuous investment and taking out



The “Requirements Phase”, New Code

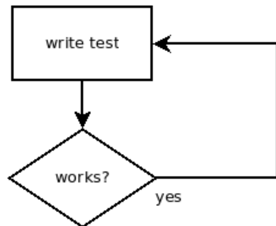
FASCHINGBAUER

Writing new code in a test driven way ...

- Nothing is clear from the beginning
- ... not even the problem

To get hold of the problem ...

- Write code that wouldn’t compile (there’s no solution yet)
- ... but gives you an impression of how a solution could look like
- Talk to people about proposed solution
- $\rightarrow$ “Finding the interface”
- *This is the first test*
- “Test First Development”



The "Requirements Phase", Existing Code

FASCHINGBAUER

Modifying existing code, to add features or change behavior ...

- Find the test suite for the module in question
    - $\rightarrow$ *structure* is important
- Add a new test for the new feature, making clear exactly what is wanted
    - The new test naturally fails, as always
- Modify code
- Run *all* tests
- Repeat

Caveats (1)

FASCHINGBAUER

Take care of your tests! If your tests are suddenly gone, your code is alone ...

Caveats (2)

FASCHINGBAUER

- Tests are what ensure your code's value
- *You can do more valuable code with tests and TDD*
- Test code is no different from "real" code
  - → Subject to bitrot
- *"Lost Tests Syndrome"*: keep your hands off manual test suite arrangement
  - → Varying support from frameworks

 Test Driven Development

Caveats (3)

FASCHINGBAUER

But:

- Nobody tests the tests
  - *false impression*: "it's only tests"
- *Structure* is important
- *Easy running* is important — everybody has to know how
- *Easy running*: avoid big dependencies — nobody will want to setup database infrastructure

Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Object Oriented Programming and Design (1)

FASCHINGBAUER

> “Perfection is attained not when there is nothing more to add, but when there is nothing more to remove”
>
> Antoine de Saint-Exupéry

- To adhere to this principle is possible even in assembly code
- ... it’s just that it’s a bit harder

Object Oriented Programming and Design (2)

FASCHINGBAUER

What OO does for us:

- *Things can be programmed like we talk about them*
- Enforces encapsulation
  - Members (private)
  - Methods
- Lets us separate out dependencies
  - Interfaces
  - Inheritance

Overview

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Example: Members and Methods

FASCHINGBAUER

Encoding and decoding: the *Julius Caesar* “Encryption” method

UML

JuliusCaesarCodec
shift : Integer
encode(data : String) : String decode(data : String) : String

C++ Code

```
class JuliusCaesarCodec
{
public:
    string encode(string data);
    string decode(string data);
private:
    int shift;
};
```

Example: Constructor and Destructor

FASCHINGBAUER

Constructor (and Destructor): controlled initialization

- There's only one way to do it — unlike in C
- `struct` initialization
- Explicit assignment
- Define a function `init_jc_codec(int shift)`
- (literally hundreds more)

C++ Code

```
class JuliusCaesarCodec
{
public:
    JuliusCaesarCodec(int shift);
    ~JuliusCaesarCodec();
// ...
};
```

Overview

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Motivation: Interfaces

FASCHINGBAUER

What if ...

- There were multiple such codecs available (Base64?)
- I don't care which one I am using
- I want to write code that just makes use of *any* codec

Solution: Interfaces

- Define the *interface* of an entire set of implementations
- Implementations *implement* interfaces
- User code is then written against *interfaces* rather than concrete implementations

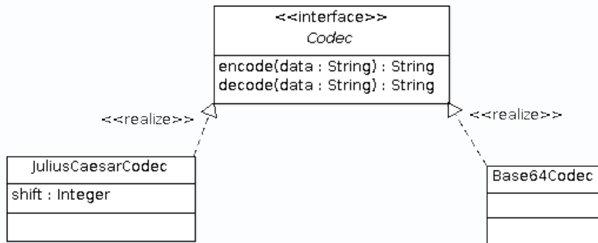
Example: Interfaces (1)

FASCHINGBAUER

Interfaces ...

- Don't implement anything
- Only force implementors into a corset for uniform usage

UML



Example: Interfaces (2)

FASCHINGBAUER

Interfaces don't implement ...

- C++: no dedicated `interface` keyword (as there is in Java)
- *Abstract Methods*

Codec *Interface*: C++ Code

```
class Codec
{
public:
    virtual ~Codec() {}

    virtual string encode(string data) = 0;
    virtual string decode(string data) = 0;
};
```

Example: Interfaces (3)

FASCHINGBAUER

Concrete code implements ...

- C++: no dedicated `implements` keyword (as there is in Java)
- Plain inheritance

Implementing an Interface: C++ Code

```
class JuliusCaesarCodec : public Codec
{
public:
    // ...
    virtual string encode(string data);
    virtual string decode(string data);
};
```

Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Design Principles

FASCHINGBAUER

Principle vs. Dogma

- Every handcraft has rules, on every single level, which everybody agrees upon
- Our handcraft is no exception
- On the *design level*: Design Principles
  - **S**ingle Responsibility
  - **O**pen/Closed
  - **L**iskov Substitution
  - **I**nterface Segregation
  - **D**ependency Inversion
- → SOLID (for people who find it hard to remember rules)
- *Antipattern*: a pattern that violates any of these principles

Overview

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Single Responsibility Principle

FASCHINGBAUER

> "Every class must have responsibility over a single part of the program"
>
> Robert C. Martin, at around 2000

> "Every class must do one thing and should do that well."
>
> Jörg Faschingbauer, all the time

Consequences:

- Defining/writing tests is easier
- Documenting is easier
- Understanding is easier

Overview

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Open/Closed Principle

FASCHINGBAUER

> "Software entities must be open for extension, but closed for modification."
>
> Bertrand Meyer, 1988

Interpretations/consequences:

- Adding functionality not by modifying but by adding (e.g. "plugins")
- Heavy use of an *interface*

Overview

FASCHINGBAUER

1 Introduction
- Literature
- Design Patterns
- Test Driven Development

2 Test Driven Development
- xUnit — How it Works
- Test Driven Development

3 OO Basics
- Members and Methods
- Inheritance

4 OO Principles — SOLID
- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

5 Design Patterns

6 Creational Patterns
- Abstract Factory
- Singleton

7 Structural Patterns
- Adapter
- Bridge
- Composite
- Proxy

8 Behavioral Patterns
- Command
- Interpreter
- Observer
- Strategy
- Visitor

Liskov Substitution Principle (1)

FASCHINGBAUER

> "It must be possible in a program to exchange two implementations of an interface *without* compromising the correctness of the program."
>
> Barbara Liskov, 1995

Is this true for our Codec "design"?

Liskov Substitution Principle (2)

FASCHINGBAUER

Classical violation of Liskow's principle: square/rectangle

- A rectangle is defined as a pair (width, height), each of which is modifiable separately
- Can a square be seen as a rectangle then?

Consequences:

- No special cases in user code
- Polished interfaces

Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Interface Segregation Principle

FASCHINGBAUER

> "No client of an interface should be forced to depend on methods it does not use."
>
> Robert C. Martin (again), at around 2000

Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Dependency Inversion Principle (1)

FASCHINGBAUER

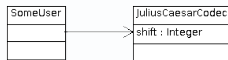
1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

Robert C. Martin (again), at around 2000

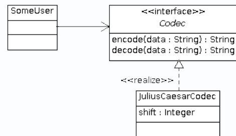
Dependency Inversion Principle (2)

FASCHINGBAUER

Bad



Good



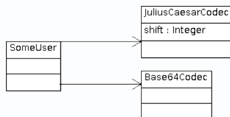
Does that really pay off?

- There is only one concrete implementation
- With *dependency inversion* applied there's one more class
- Not easily readable

Dependency Inversion Principle (3)

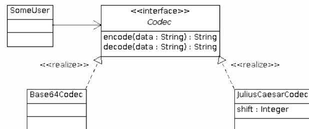
FASCHINGBAUER

Really Bad



- Typical scenario: long `if-else-if-...` chains
- Each association used based on, say, the value of an integer variable

Really Good



- Shorter code — no long chains
- (Ideally) does exactly one thing, and *delegates* encoding

Does anybody know the Strategy Pattern?

Overview

FASCHINGBAUER

1 Introduction
- Literature
- Design Patterns
- Test Driven Development

2 Test Driven Development
- xUnit — How it Works
- Test Driven Development

3 OO Basics
- Members and Methods
- Inheritance

4 OO Principles — SOLID
- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

5 Design Patterns

6 Creational Patterns
- Abstract Factory
- Singleton

7 Structural Patterns
- Adapter
- Bridge
- Composite
- Proxy

8 Behavioral Patterns
- Command
- Interpreter
- Observer
- Strategy
- Visitor

Design Patterns — The Legend

FASCHINGBAUER

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
>
> Christopher Alexander, 1977

- Christopher Alexander is an *architect*
- *Gang of Four* got heavily inspired by his work

Design Patterns — Definition

FASCHINGBAUER

A Design Pattern has the following attributes:

- **Name.** We use it to identify and talk about problems and their solutions.
- **Problem.** A pattern is a solution, and there is no solution without a problem. The problem must be clearly defined.
- **Solution.** A description of the solution — design, responsibilities, collaborations, ...
- **Consequences.** Benefits, trade-offs. Needed for evaluation/selection.

Design Patterns — *The Book*

FASCHINGBAUER

Creational Patterns

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Structural Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioral Patterns

- Interpreter
- Template Method
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor

Overview

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Creational Patterns — What for? (1)

Direct object creation ...

Using concrete type

```
void use_codec(JuliusCaesarCodec *codec) { /*...*/ }

JuliusCaesarCodec *codec = new JuliusCaesarCodec(5);
use_codec(codec);
```

- Hard dependency on `JuliusCaesarCodec`, introduced by
  1. codec being of *concrete* type
  2. Instantiation of concrete type
- *Is it necessary to use concrete type?*
- *Does `use_codec()` care?*

Creational Patterns — What for? (2)

Using interface type

```
void use_codec(Codec *codec) { /*...*/ }

Codec *codec = new JuliusCaesarCodec(5);
use_codec(codec);
```

- Still hard dependency on `JuliusCaesarCodec`, introduced by
  1. Instantiation of concrete type

Creational Patterns — What for? (3)

FASCHINGBAUER

Naive solution: external function

```
Codec *create_codec();
void use_codec(Codec *codec) { /*...*/ }

Codec *codec = create_codec();
use_codec(codec);
```

- Dependency has been moved to `create_codec()`
- We don't care which `Codec` incarnation we use → *Liskow Substitution Principle*
- Decided externally, by the implementation of `create_codec()`

Creational Patterns — What for? (4)

FASCHINGBAUER

Now how could `create_codec()` **look like?**

```
enum CodecType {
  JULIUS_CAESAR,
  BASE64
};
int jc_shift = 5;
// modify if you want different type
CodecType type_instantiated = BASE64;

Codec *create_codec() {
  switch (type_instantiated) {
    case JULIUS_CAESAR: return new JuliusCaesarCodec(jc_shift)
    case BASE64: return new Base64Codec;
  }
}
```

Creational Patterns — What for? (5)

FASCHINGBAUER

There are prettier solutions!

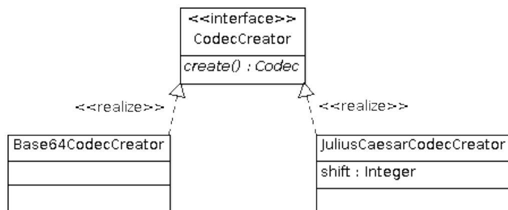
→ Creational Patterns

Overview

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Abstract Factory

FASCHINGBAUER



Setup (near `main()` function?)

```
codec_factory = new JuliusCaesarCodecFactory(5);
```

Usage

```
Codec *codec = codec_factory->create();
use_codec(codec);
```

Abstract Factory — Discussion

FASCHINGBAUER

How does using (instantiating) code get to the factory?

- Pass factory in
  - Explicit: everyone can see that module makes use of it
  - $\rightarrow$ Dependency is obvious
- Global Variable $\rightarrow$ bad smell
  - Hidden dependency
- Singleton ...

Pass via Constructor

```
class SomeCodecUser
{
public:
    SomeCodecUser(CodecFactory *codec_factory);
};
```

Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Singleton

FASCHINGBAUER

Ensure a class has only one instance,
and provide a global access point to it.

SomeClassWithOneInstance
get_instance() : SomeClassWithOneInstance do_something() do_more()

- `get_instance()` is not called on an instance
- C++, Java: `static`

Singleton — Example

FASCHINGBAUER

Example: `Base64Codec` ...

- Everybody needs it
  - Email attachments
  - HTTP transport
  - ...
- There need not be multiple instances
  - It has no data of its own
  - Only the algorithm (`encode()`, `decode()`)

Singleton — Example, Class Definition (1)

base64.h

```
class Base64Codec
{
public:
    // return (and on-demand instantiate)
    // the only Base64Codec object in the world
    static Base64Codec &get_instance();

    string encode(string data);
    string decode(string data);

private:
    ...
};
```

Singleton — Example, Class Definition (2)

base64.h

```
class Base64Codec
{
public:
    ...
private:
    // THE object
    static Base64Codec *instance;

    // inhibit public instantiation
    Base64Codec();
};
```

Singleton — Example, Class Implementation

base64.cc

```
Base64Codec *Base64Codec::instance;

Base64Codec &Base64Codec::get_instance()
{
    if (instance == NULL)
        instance = new Base64Codec;
    return *instance;
}
```

Singleton — Example, User Code

FASCHINGBAUER

user.cc

```
string binary_data = ...;
string email_attachment =
    Base64Codec::get_instance().encode(binary_data);
```

Singleton — Discussion

FASCHINGBAUER

Singleton: Pros

- Nicely encapsulates global data
- Saves one from passing parameters instead

Singleton: Cons

- Nicely encapsulates global data
- Saves one from passing parameters instead
- It's still *global*
    - Unit testing?
- Makes the design less obvious
    - Singleton access hidden deep in implementation
    - *Hidden dependency!*
- Anti-Pattern?

Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Structural Patterns — What for?

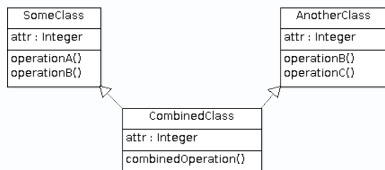
FASCHINGBAUER

Any non-trivial program has an object structure ...

- Multiple objects are combined → structure
- Some structures *and motivations* are immediately obvious
- ... others aren't
- There are no billions of different motivations
- → A handful of *Patterns* is sufficient to describe most

Combining Objects — Multiple Inheritance (1)

FASCHINGBAUER



Issues:

- `CombinedClass` is a *union* of both of its bases
- *Contains* boths methods and data, *without* “namespace” qualification
    - Conflicts
    - Ambiguities
    - Prone to bugs
- Situation very similar to global variable usage

Combining Objects — Multiple Inheritance (2)

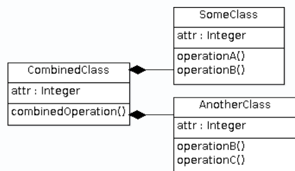
Massacre in C++

```
class CombinedClass : public SomeClass, public AnotherClass
{
public:
    void combinedOperation() {
        attr = SomeClass::attr + AnotherClass::attr;
        operationA();
        SomeClass::operationB();
        AnotherClass::operationB();
        operationC();
    }

private:
    int attr;
};
```

Combining Objects — Composition (1)

FASCHINGBAUER



Better, because ...

- Relationships are more obvious
- No ambiguities
- **Speech is clear**: "*Uses* `SomeClass` and `AnotherClass` to implement its operations" (if anybody cares at all)
- **And not**: "*Is both a* `SomeClass` and a `AnotherClass`, and adds a little to both"

Combining Objects — Composition (2)

FASCHINGBAUER

```
class CombinedClass
{
public:
    void combinedOperation() {
        attr = Some.attr + another.attr;
        some.operationA();
        some.operationB();
        another.operationB();
        another.operationC();
    }
private:
    SomeClass some;
    AnotherClass another;
    int attr;
};
```

Combining Objects — There's More To It

FASCHINGBAUER

So much for trivial object combinations

$\rightarrow$ Structural Patterns

Overview

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Adapter — Sample Problem

FASCHINGBAUER

Sample Problem: `Base64Codec` **(again) ...**

- I have an implementation based on C++ `iostream`
- Want to implement C++ `string` based interface
    - ... as dictated by interface `Codec`

```
class Base64Codec
  : public Codec
{
public:
  string encode(string);
  string decode(string);
};
```

```
class IOBase64Codec
{
public:
  static void encode(
    istream&, ostream&);
  static void decode(
    istream&, ostream&);
};
```

Adapter — Motivation

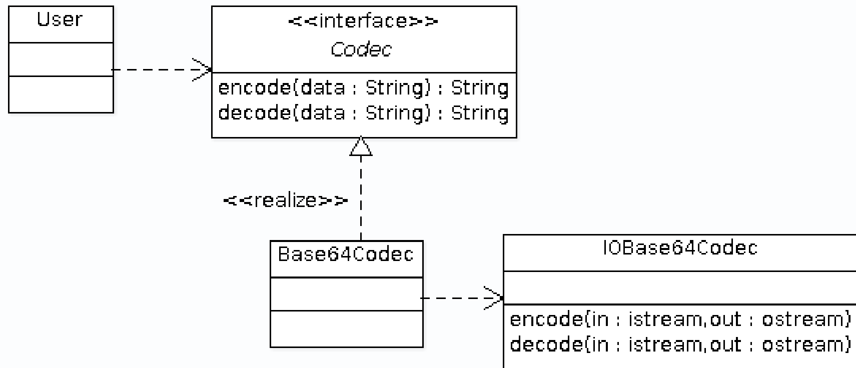
FASCHINGBAUER

Pattern: apply ultralight glue ...

- Interfaces are similar
  - It is obvious that their *intention* is the same
- Interfaces are incompatible, compiler-wise
- *Adapt* the `iostream` implementation into our `string` based `Codec` hierarchy
- Fortunately there's C++'s `istringstream` and `ostringstream` which turns a string into a stream and back

Adapter — Graphics

FASCHINGBAUER



Adapter — Implementation

A typical adapter implementation ...

- Usually very short, to the point, obvious
- Inlineable in most cases
- No big deal — it's the *name* that is important for communication

```
string Base64Codec::encode(string in)
{
    istringstream sin(in);
    ostringstream sout;
    IOBase64Codec::encode(sin, sout);
    return sout.str();
}
```

Overview

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Bridge — Example: String (1)

FASCHINGBAUER

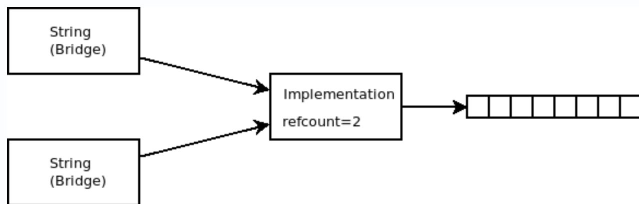
Straightforward bridge example: String

- Goal: *transparent* sharing of objects
- User code has an innocent looking object $\rightarrow$ *bridge*
  - *Handle* to the real stuff
  - ... possibly augmented with some additional higher level methods
- E.g. a naive String class
  - Implementation: reference counting, low level memory management
  - Bridge: cute methods *find*, *cut*, *splice*, ...

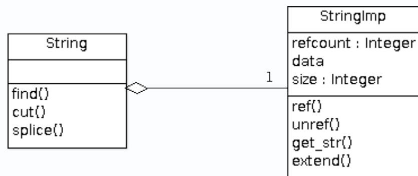
Bridge — Example: String (2)

FASCHINGBAUER

String: Object Diagram



String: Class Diagram



Bridge — Definition

FASCHINGBAUER

Bridge: GoF Definition

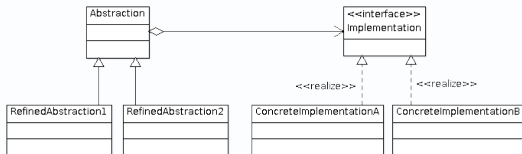
Decouple an abstraction from its implementation so that the two can vary independently.

So what could that mean?

- `String` is an easy application of the Bridge pattern
  - `String` is an abstraction: nobody sees low level memory issues, but rather useful methods
- Definition leaves much more room for interpretation
  - Abstraction side can vary
  - Implementation side can vary

Bridge — General Case

FASCHINGBAUER



- Focus is more on independent evolution of both sides
- *Abstract Factory* can be used to setup bridges

As for the `String` example ...

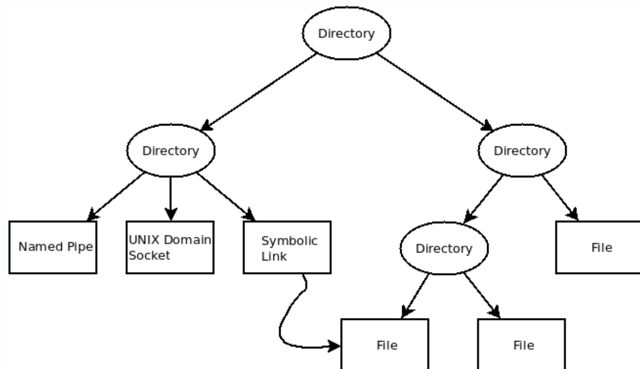
- Abstractions: `ASCIIString`, `UTF8String`, ...
- Implementations: `StringImpMalloc`, `StringImpChunked`

Overview

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

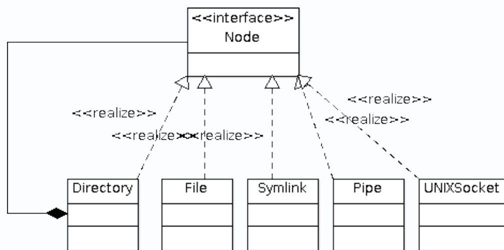
Composite — Example: Unix Filesystem (1)

FASCHINGBAUER



Composite — Example: Unix Filesystem (2)

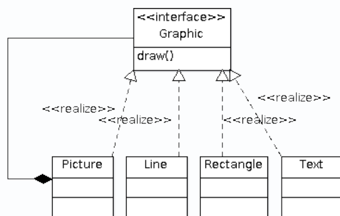
FASCHINGBAUER



- Hierarchy is in place with this design
- But there is more which is lacking from this example
  - Directory is different
  - Application (`tar`, `tree`, ...) needs to *know* the concrete types
  - → Complexity

Composite — Example: Graphics

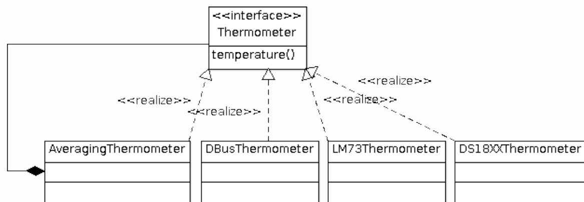
FASCHINGBAUER



- *Common interface*
- Composite `Graphic` draws all it contains
- *Recursive*: `Graphic` can contain `Graphic` can contain ...

Composite — Example: Thermometer

FASCHINGBAUER



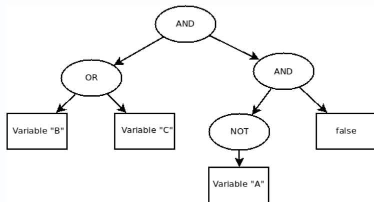
- `AveragingThermometer`: average out of several temperatures
- Design variation: *weighted* average

Composite — Example: Boolean Expression

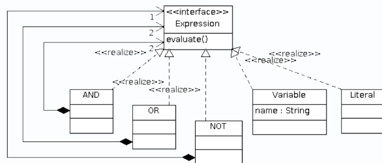
FASCHINGBAUER

- Abstract Syntax Trees (AST)
- Simple AST: Boolean Expression
- Evaluating (executing) ASTs is a different story
- *Interpreter Pattern*

Object Structure



Class Diagram



Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

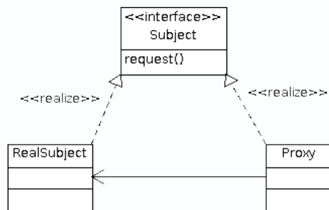
Proxy — Definition

FASCHINGBAUER

GoF Definition

Provide a surrogate or placeholder for another object to control access to it.

GoF Diagram

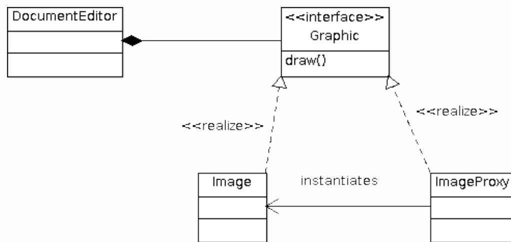


A-ha ...

Proxy — GoF Example: Image in Text

FASCHINGBAUER

- Image should not be loaded/calculated when not in visible area
- $\rightarrow$ Demand loading




```
ImageProxy::draw()
{
    if (image == NULL)
        image = Image(filename);
    image->draw();
}
```

Proxy — Example: Plugin Interface

FASCHINGBAUER

Loading code from a file ...

- Unix/Linux: `dlopen()`, `dlsym()`
- Windows: `LoadLibraryEx()`, `GetProcAddress()`

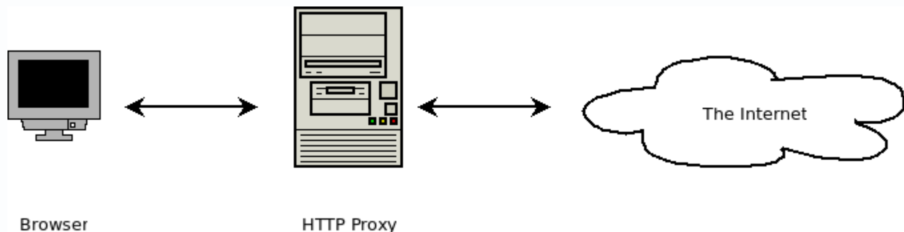
Defining a plugin scheme: `Codec` **...**

- A plugin (DLL, sharded library) brings one `Codec` object
  - Well defined name: `the_object`
  - We only know the interface (implementation buried in plugin)
- Proxy `Codec` ...
  - Load library
  - Use `dlsym()` to find `the_object`
  - Use that as `RealSubject`

Proxy — Remoting (1)

FASCHINGBAUER

The word Proxy as everybody knows it ...



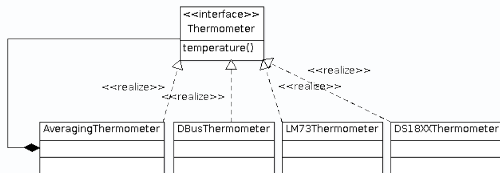
Proxy — Remoting (2)

FASCHINGBAUER

Distributed applications ...

- Use a remote object as if it were remote
- Local *Proxy* object — satisfying the interface
- Remote concrete implementation
- Wire protocol in between

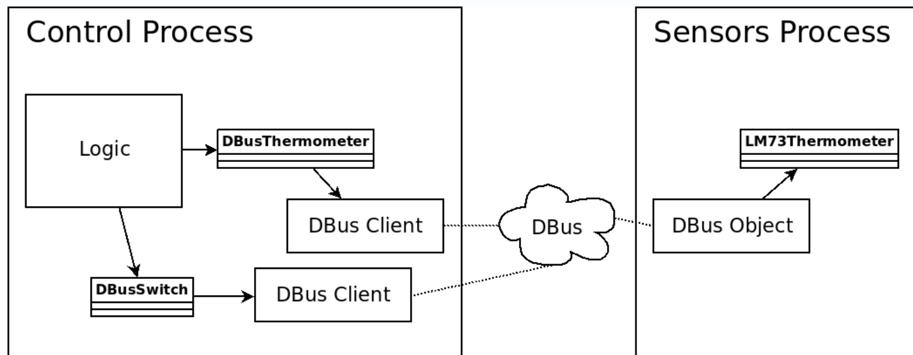
DBusThermometer



Proxy — Remoting (3)

FASCHINGBAUER

Distributed Application



Overview

FASCHINGBAUER

1 Introduction
- Literature
- Design Patterns
- Test Driven Development

2 Test Driven Development
- xUnit — How it Works
- Test Driven Development

3 OO Basics
- Members and Methods
- Inheritance

4 OO Principles — SOLID
- Single Responsibility
- Open/Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

5 Design Patterns

6 Creational Patterns
- Abstract Factory
- Singleton

7 Structural Patterns
- Adapter
- Bridge
- Composite
- Proxy

8 Behavioral Patterns
- Command
- Interpreter
- Observer
- Strategy
- Visitor

Behavioral Patterns

FASCHINGBAUER

Structure versus Functionality

- Structure implies functionality (sometimes)
  - Composite: boolean expressions
- Structure implies only little functionality (sometimes)
  - Adapter
  - Bridge
- Functionality implies structure (mostly)

Behavioral Patterns:

- Focus on object interactions
- Parameters, return values
- $\rightarrow$ Semantics

Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

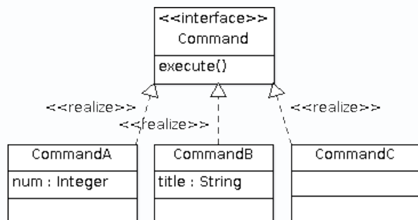
Command

FASCHINGBAUER

Command: the problem ...

- Imagine a large framework, passing and executing requests of some sort
- Requests are not fixed, but rather extensible/anonymous

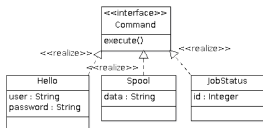
- Request $\iff$ function call
- Variable parameters
- $\rightarrow$ Encapsulate parameters in object

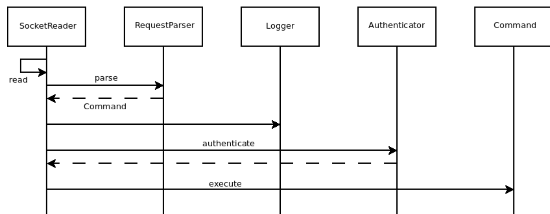


Command — Example: Remote Requests

FASCHINGBAUER

Large framework, handling remote requests ...





- Which pattern is used for `RequestParser`?
- Command's execution maximally decoupled from the rest
- $\rightarrow$ *Highly dynamic*

Command — Problems

FASCHINGBAUER

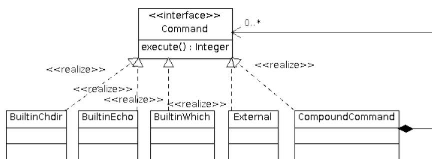
Command has problems (as everything) ...

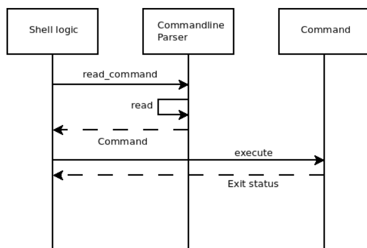
- Non-uniform return value — `Hello`, `Spool`, `JobStatus` are quite different
  - How is that serialized back onto the line?
  - → Probably not appropriate!
- `Command` classes are largely unrelated (→ structural problems)
- Many `Command` classes
- `Command` implementation tend to become complex
- Gets unhandy *really soon* if applied unappropriately → watch out, and change!

Command — Example: Shell

FASCHINGBAUER

UNIX Shell: parsing and executing user's command line ...





- Uniform outcome (exit status) → *perfect!*
- GoF example: menu items (no outcome at all)

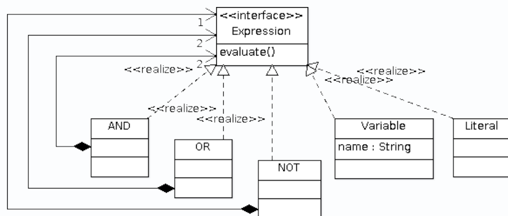
Overview

FASCHINGBAUER

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Interpreter — Example: Boolean Expression (1)

FASCHINGBAUER



- *Composite* structure
- *Evaluation* is not so simple
- Storage of variable's values → *Context*

Interpreter — Example: Boolean Expression (2)

Interpreter vs. Composite: hard to tell the difference ...

- Languages are best represented by trees
- Trees are best represented using Composite
- $\Longrightarrow$ Interpreter likely is Composite
- Not necessarily vice versa

```
class Context {
public:
  bool lookup_value(string);
};
class Expression
{
public:
  virtual bool
    evaluate(Context&) = 0;
};
```

Interpreter — Discussion

FASCHINGBAUER

Ups

- Grammar is easy to extend
- Easily implemented

Downs

- Complex grammars hard to maintain
- $\rightarrow$ Parser generators probably a better alternative

See also

- Visitor pattern, to extend functionality

Overview

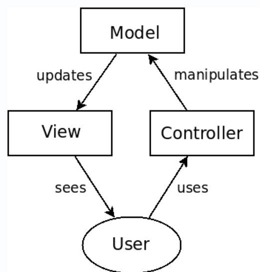
1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Observer — Model-View-Controller (MVC) (1)

FASCHINGBAUER

Model-View-Controller: revolution in GUI design in the late 70s

- *Model*: application data, business logic
- *View*: visible representation
  - *observes* the model → callback
- *Controller*: specifies actions to manipulate the model
  - Triggered by button clicks, for example



Observer — Model-View-Controller (MVC) (2)

FASCHINGBAUER

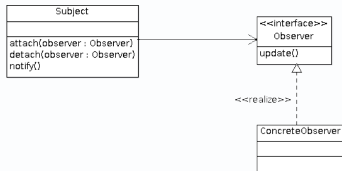
MVC is not among the GoF patterns ...

- Subdivided into more generalized patterns
    - MVC was discovered for GUI design
    - Integrated into Smalltalk
    - Adopted by Apple for their GUIs
- *Controller* → *Strategy*
    - "Do something!", and not caring what
- *View* → *Composite*
    - Recursive decomposition of graphics
- Interaction (notification) between *Model* and *View* → *Observer*

Observer — Collaborations (1)

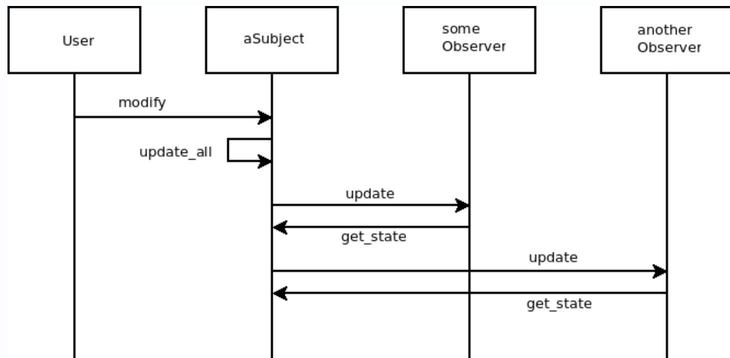
FASCHINGBAUER

- Subject has zero or more Observers registered
- On modification, all of them are notified
- On notifications, they update their ... whatever



Observer — Collaborations (2)

FASCHINGBAUER



Observer — Discussion (1)

FASCHINGBAUER

- Observing multiple subjects
  - `update()` needs to pass a reference to changed subject
- *When* is the update triggered?
  - Every single modification?
  - → Inconsistent state
  - "Transactional integrity"?
- *Who* triggers the update?
  - User after *he* is done? → unhandy
  - Subject? Transactional integrity?
  - → Be careful during design! Change if smell detected!

Observer — Discussion (2)

FASCHINGBAUER

- *Push* versus *pull*
  - Does subject push modification information?
  - Or does observer query for it?
- Multithreading
  - *pull* has to *lock* into subject
  - → Deadlock danger during callback (?)

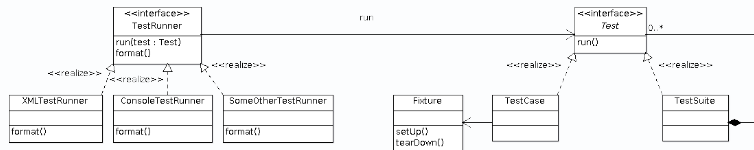
Overview

1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Strategy — Example: TestRunner (1)

FASCHINGBAUER

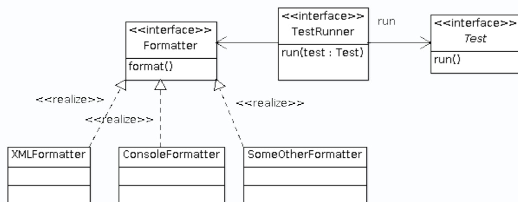
The **Template Method** pattern, as applied to `xUnit` ...



- **Template Method** doesn't scale (as everybody knows)
  - It is a straightforward solution to long `if-else` chains
  - One dimension of variability (here, `format()`) is manageable
  - Number of implementors grows exponentially with the number of variations
  - Extension is not reusable
- Different solution necessary

Strategy — Example: `TestRunner` (2)

FASCHINGBAUER



Consequences ...

- `TestRunner` instances are parameterizeable
  - `TestRunner` is a *concrete* class
  - Receives a `Formatter` during construction (or at runtime, or ...)
- Formatting is Unit-testable *without prior test run*

Strategy — Final Words

FASCHINGBAUER

Strategy is one of the most important patterns, because ...

- *Delegation* is perfectly/clearly/cleanly expressed
- Runtime parameterization possible
  - Delegee need not be passed to constructor — can also be done dynamically
- Perfect alternative to most long `if-else` chains where functionality is chosen

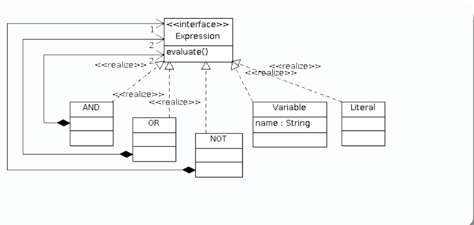
Overview

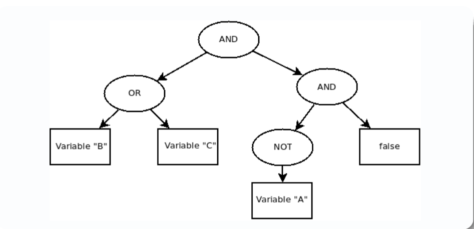
1. Introduction
   - Literature
   - Design Patterns
   - Test Driven Development
2. Test Driven Development
   - xUnit — How it Works
   - Test Driven Development
3. OO Basics
   - Members and Methods
   - Inheritance
4. OO Principles — SOLID
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
5. Design Patterns
6. Creational Patterns
   - Abstract Factory
   - Singleton
7. Structural Patterns
   - Adapter
   - Bridge
   - Composite
   - Proxy
8. Behavioral Patterns
   - Command
   - Interpreter
   - Observer
   - Strategy
   - Visitor

Visitor — Extending Boolean Expressions (1)

FASCHINGBAUER

- Main method: `evaluate()`
- What about `prettyprint()`?
- What about `myfavoritemethod()`?
- Add all to `Expression`?
- → Unhandiness, uncovered by the *Interface Segregation Principle*

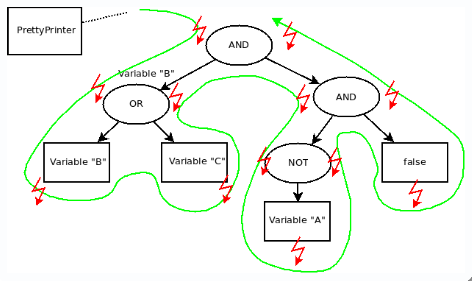


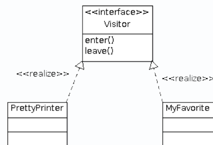


Visitor — Extending Boolean Expressions (2)

FASCHINGBAUER

- Add `visit()` to `Expression`
- Nodes: `enter()`, walk children, `leave()`
- Leaves: `enter()`, `leave()`
- ... or so





Notes

FASCHINGBAUER