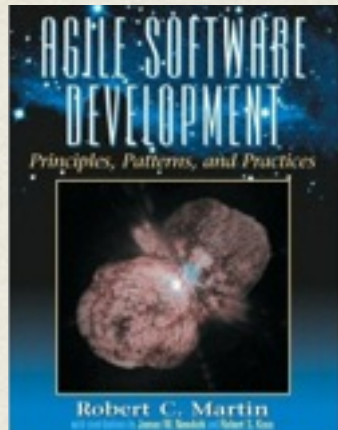
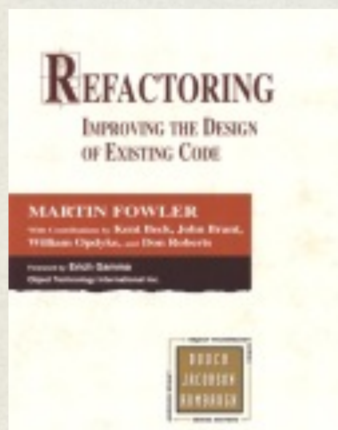
INTRODUCTION

Advanced Programming Concepts 2014

CREDITS



- Robert C. Martin et al.
  Agile Software Development. Principles, Patterns and Practices
  Prentice Hall, 2003
  http://www.objectmentor.com/omSolutions/oops_what.html
  (OO design principles catalogue, incl. design smells)



- Martin Fowler et al.
  Refactoring. Improving the Design of Existing Code
  Addison-Wesley, 1999
  http://www.refactoring.com (incl. catalogue of refactorings)

- Gang of Four (E. Gamma, R. Helm, R. Johnson, J. Vlissides)
  Design Patterns. Elements of Reusable Object-Oriented Software
  Addison-Wesley, 1995

AIM OF THIS LECTURE

- Give an overview of the school program
  - Present the common ideas and concepts behind a large fraction of the topics
  - Show you how the different lectures link together
- Help you to better digest the rest of the school (provide the right boxes for the knowledge that is presented)

OUTLINE

- Motivation/Introduction
  - Why APC?
  - Brief outline of the school
- Code Design Basics
  - What *is* good design
  - A minimal example and its lessons
  - Designing in practice
  - Design vs. Performance
- The outline of the school revisited

PART I: MOTIVATION & INTRODUCTION

A SCHOOL ON APC — WHY?

- In HEP, programming makes up a large fraction of our daily work
- In contrast, there's not much education in programming
  - Mostly learn from colleagues or teach yourself
- This school is an attempt to help closing that gap
  - … but is it really worthwhile?

BUT WHY CARE ABOUT CODE DESIGN AT ALL?

- We are physicists — programming is just a
  … tax we have to pay …(?)
  … tool we have to use … (!)
  in order to do physics … much like statistics.

- The better we master this tool,
  the more efficiently we can use it.
  - … and the more time we can spend on physics!

- Studying APC makes sense if it helps us spend less time on programming in the long run.

TYPICAL PHYSICIST CODING SCENARIOS

- **Small studies** („…just quickly make a few plots…“)
  - Often start from scratch, sometimes take over existing pieces
  - Limited lifetime and complexity (but can grow…)
- **Analysis software**
  - Medium complexity, lifetime and #developers (can vary a lot)
  - Take over existing code or start from scratch
- **Reconstruction software**
  - Complex packages, long lifetime (desirably), many developers
  - Normally don’t write from scratch but extend existing code

HOW DO WE SPEND OUR CODING TIME?

- Write code from scratch
- Modify/extend existing code
  - Our own code
  - Other peoples' code (need to understand first)
- Debugging

(Anyone who never spent days/weeks on endless debugging?)

„WHY CARE ABOUT APC?“ — CONCLUSIONS —

- We want to spend time on physics, not coding
- We can save time, if we
  - learn to extend/modify existing code efficiently
  - learn to work efficiently with legacy code
  - produce less bugs
  - produce well-designed code (by applying APC)
- Importance of good design is proportional to lifetime, complexity and number of developers

OUTLINE OF THE SCHOOL

- Lectures can be grouped into 2-3 topics:
  - UML
  - **Code design**
  - **Performance**

UNIFIED MODELING LANGUAGE (UML)

- Notation to visualize aspects of software design
  (class structure, program flow, use cases, ...)
- Comprises a variety of different diagram types
  (class-, object-, sequence-, activity-, state- , ...)
- Main purpose: human communication
  (e.g. explain your code to your colleagues)
  - Diagrams can help you design your code,
    but they can’t replace it

CODE DESIGN — LECTURE TOPICS —

- **Refactoring**
  - „The big brother of code cleaning“
  - Improve the design of code after it has been written (but without altering the external behavior!)
- **Principles of class design and of package design**
  - How to write good classes
  - How to group your classes into packages/libraries
- **Design patterns**
  - Well-proven solutions to frequently occurring problems

PERFORMANCE — LECTURE TOPICS —

- **Performance and design**
  - Performance vs. Design or
    „Is ‚clean‘ code necessarily slow?“
- **Meta-template programming**
  - Cool tricks for faster code

PART II:
CODE DESIGN — BASICS

WHAT *IS* WELL-DESIGNED CODE?

According to „Uncle“ Bob Martin, „every software module has three functions:“

1. „**First, there is the function it performs while executing.** This function is the reason for the module's existence.“

- In other words: good code is code that works.
- Assert that your code works at all times using tests.
  (—> lecture(s) on refactoring)

CHANGE

2. „**The second function of a module is to afford change.** Almost all modules will change in the course of their lives, and it is the responsibility of the developers to make sure that such changes are as simple as possible to make.“

- When code is simple to change, you will be faster when doing so and introduce less bugs.
  (—> all code design lectures)

TALK TO THE CODE

3. „**The third function of a module is to communicate to its readers.** Developers unfamiliar with the module should be able to read and understand it without undue mental gymnastics.“

- When code is easy to understand, you can work with it more efficiently (i.e. be faster and introduce less bugs).
- Code can (almost) look like spoken language!

(—> refactoring, OOD principles)

„Any fool can write code
that a computer can understand.

Good programmers write code
that humans can understand.“

-Martin Fowler

OTHER PROPERTIES OF GOOD CODE DESIGN

- Maintainability (<—> *change*!)
- Modularity/Flexibility (—> *change*)
- Reuse/Reusability (as opposed to code duplication…) (—>*enforced by modularity/flexibility*)
- These are frequently claimed as „the“ benefits of Object-Oriented Design (OOD)

The Devil take your axioms, rather show us an example!

DISCLAIMER

- The program in the example *way* too small
to be worth any designing effort

- But the mechanics that we will see work just the same in a large-scale system

  - Take it as a representative (though minimal) example illustrating a variety of aspects that will be detailed in the course of the school

THE „COPY“ PROGRAM

LESSONS FROM THE „COPY“ PROGRAM

CHANGE AT WORK

- We saw an example of changing requirements:
  - We needed to *extend* the behavior of the program in order to be able to read from the paper tape
- Typical scenario, happening all the time in real life
- **Lesson:**
  The requirements *will* change.
  (So you better know how to react.)

ROTTING CODE

- In the first version of the example, we saw the design of the program degraded due to the modifications that were made to the code.

- **Lesson:**
  Code *rots* in the presence of change,
  if you don’t react to it properly.
  - The code begins to „smell“!
    (we’ll come back to that later)

REACTING TO CHANGE

- How did we react to the change? In two steps:
  1. Adjust the design (refactor)
  2. Add the new behavior
- After refactoring, the new behavior was achieved by only *adding* new code, not change existing code.
- **Lessons:**
  - When a change is hard to make, refactor until the change is simple to make; then make the change.
  - The simplest change is if you only need to add code

ENCAPSULATE CHANGE

- What was the key to success when we refactored? (Why did we only have to add new code in the end?)
  - Hide the new behavior behind the Reader class
  - Further changes of the same sort will always be easy, now! (Simply add more Reader derivatives.)
- **Lesson:**
  „*Encapsulate the concept that varies!*“
  - Common theme of numerous design patterns

USE ABSTRACTIONS

- How did we manage to encapsulate the change?
  - We used an *abstraction* for it!

- **Lesson:**
  Abstraction is *the* mechanism to encapsulate change.

ABSTRACTION ON CODE LEVEL

- How did we realize the abstraction in the code: Abstract interface class with several implementations
  - *(Dynamic) Polymorphism*

- **Lesson:**
  Abstraction, realized by abstract interface classes …

  … is *the* code-level mechanism
  behind many design patterns

  … is used to satisfy several of the design principles

ON DESIGN PATTERNS

- We have introduced a simple design pattern:
  Abstract Server
- Got there by general considerations about our design (without even knowing about the pattern)
- **Lesson:**
  This is how to generally apply patterns:
  get there by applying general design considerations.

REDUCING DEPENDENCES

- Introducing the Abstract Server pattern removed the dependencies on printer and paper tape. Only depend on the (abstract) interface instead.
- **Lesson:**
  Reducing dependencies like this is *the* low-level mechanism behind many of the benefits claimed for OO technology.

INTERFACE VS. IMPLEMENTATION

- Initially, the copy program used the *implementation* of how to read from the keyboard: invoke „RdKbd()“
  - This the way how most physicists code (and think)
- In the final version, the program only knows about the „Reader“ *interface*
- **Lesson:**
  Program to an interface, not to an implementation.

ONLY THE READER?!

- Why didn't we also implement a Writer class?
  - An axis of change is an axis of change
    only if the changes really occur!
- Only implement an abstraction when actually needed.

  „Fool me once, shame on you.
  Fool me twice, shame on me."
- **Lessons:**
  - Resisting premature abstraction is as important as abstraction itself.
  - Always keep your design as simple as possible.

DESIGNING IN PRACTICE

(UNIT) TESTS

```
class TestTVector2 : public TestCase {
public:
  TestTVector2(string name): TestCase(name) { }

  void runTest() {
    TVector2 vector(1, 2);
    CPPUNIT_ASSERT( vector.GetX() == 1);
    CPPUNIT_ASSERT( vector.GetY() == 2);
  }
};
```

- Use your code in a minimal setup, make sure that what you get is what you expect.
- Run automatically when you compile (—> framework), show either green (success) or red (fail)
  —> provide *immediate* feedback
- „Unit“ = the smallest testable part of an application

(UNIT) TESTS: BENEFITS

- Valuable tool to ensure your code works at all times
  - E.g. add a test after each fixing a bug to make sure it never occurs again
- Serve as documentation which is:
  unambiguous, accurate, reliable, never outdated
- Essential for refactoring
  (make sure you *really* don't change the behavior)

TEST-DRIVEN DEVELOPMENT (TDD)

- Not all code is easily *testable*
  (in particular in physics software)
  - Hard/tedious to equip existing code with unit tests
- Test-Driven Development (TDD)
  - First write the test, then the code to make it pass
  - Automatically enforces testability
    —> leads to decoupled, modular (i.e. good) design
  - Comprehensive set of tests grows with the code

RED - GREEN - REFACTOR

- Start by coding as usual
  - Add some code, compile, run, watch it fail *(TDD: add test)*
  - Debug until it works *(TDD: make test pass)*
  - This may well be rather quick & dirty
- Once the code works: Refactor!
  - Simplify where possible (loops/conditionals/…)
  - Rename variables/functions for maximum readability
  - Extract functions or create classes as needed
  - …
- Useful paradigm for small projects or single changes

THE KITCHEN METAPHOR

REMOVING THE SMELLS

- We have seen the notion of code (or design) **smells** for „symptoms of bad design“

- In this picture, code design is the act of removing…
  - … code smells by applying appropriate refactorings
  - … design smells by applying design principles

- **CAUTION:** Don’t use design like a perfume!
  - No smell —> No principles or refactorings (else: overdesign)
  - Always keep your design as simple as possible

DESIGN SMELLS

Design smells:

- Rigidity
- Viscosity
- …

(see the full list in the design principles lecture)

- Abstract, high-level concepts
- Describe different aspects of „The code is hard to change (and hence error prone)“
- Straight violations of the „three functions of a software module“ (—> bad by definition)

CODE SMELLS

Code smells:

- Code duplication
- Excessive use of long, nested loops/conditionals
- Long functions
- ...

(see a longer list in the refactoring lecture(s))

- More concrete, code-related than the design smells
- Why are these bad? Because their presence leads to (or directly implies) design smells!

DESIGN VS. PHYSICS

- You may find it difficult to apply textbook design concepts to your real-life physics code
  - Physics code is special in several regards
    (many computations, less „business rules“, not much about „features“, complex and volatile data structures, ...)
  - Adopt concepts & transfer ideas to physics needs
- But also need to adjust your coding style!

DESIGN VS. PERFORMANCE

DESIGN VS. PERFORMANCE

- There is a tension between design and performance:
  - Code that is optimized for speed is likely to be less well-readable and harder to change
  - Code that is optimized for design may be slow
  - Tradeoff between design and performance
- There are several ways to approach this problem

STRATEGIES FOR PERFORMANCE TUNING

- Constant attention approach:
  Optimize every line you write for performance
  - But: typically >90% CPU time spent in <10% of the code —> 90% of time spent on tuning are wasted
  - Intuitive approach, but inefficient
- Better: 1st care about design and ignore performance, then find the hotspot and fix them
  - Focus performance tuning to where it’s needed
  - Well-designed code often is even easier to optimize
- Never tune for performance before running a profiler!

STATIC VS. DYNAMIC POLYMORPHISM

```
<<interface>>
Algorithm
+initialize(): void
+execute(): void
+finalize(): void
```

```
AlgA
inheriths from Algorithm
+initialize(): void
+execute(): void
+finalize(): void
```

```
AlgB
inherits from Algorithm
+initialize(): void
+execute(): void
+finalize(): void
```

```
AlgRunner
using dynamic polymorphism
-_theAlg: Algorithm*
+run(): void
```

```
AlgRunner runner(new AlgA());
runner.run();
```

```
void run() {
  _theAlg->initialize();
  _theAlg->execute();
  _theAlg->finalize();
}
```

Dynamic version

- Interface class with several implementations — just as „usual“
- AlgRunner can run different Algorithms and switch between them at run time
- Can make a collection of AlgRunners with different Algorithms (always the same class)

STATIC VS. DYNAMIC POLYMORPHISM

```
AlgType:
AlgRunner
using static polymorphism
-_theAlg: AlgType*
+run(): void
```

```
AlgRunner<AlgA> runner;
runner.run();
```

```
AlgA
+initialize(): void
+execute(): void
+finalize(): void
```

```
AlgB
+initialize(): void
+execute(): void
+finalize(): void
```

```
template<class AlgType>
class AlgRunner {
...
};
```

```
void run() {
  _theAlg->initialize();
  _theAlg->execute();
  _theAlg->finalize();
}
```

Static version

- Doesn't need the interface class
- Defined at compile time (AlgRunners for different Algorithms are different classes!)
- Faster, but less flexible (can't change Alg at run time, collections are difficult)
- Only use static polymorphism if needed, usually prefer dynamic polymorphism

PART III:
OUTLINE OF THE SCHOOL REVISITED

OUTLINE OF THE SCHOOL

- UML
- **Code design**
  - Refactoring
  - Principles of class design and of package design
  - Design patterns
- **Performance**
  - Performance and design
  - Meta-template programming

REFACTORING — CONTENTS & CONCEPTS —

Contents (simplified)

- List of code smells
- List of refactorings and how to apply them
- Refactoring and Testing
- Refactoring vs. Physics

Concepts — when to refactor?

- When you read/understand foreign/legacy code
- To recover rotten code —> remove code smells
- To deal with Change: first refactor, then extend
- Constantly, every day: after any changes to your code

CLASS DESIGN PRINCIPLES — CONTENTS & CONCEPTS —

Contents

- List of design smells — List of design principles

Concepts — when/how to apply principles?

- to prevent code from rotting —> remove design smells
- to keep the design as simple as possible (no smell, no action)
- to manage change:
  - encapsulate change using abstractions (—> polymorphism)
  - extend behavior by adding code
- to manage/reduce dependencies
  - program to interfaces, not implementations

PACKAGE DESIGN PRINCIPLES — CONTENTS & CONCEPTS —

Contents

- List of design principles

Concepts — when/how to apply principles?

- to manage change:
  - group changes that occur together
- to manage dependencies
  - depend interfaces, not on implementations

DESIGN PATTERNS — CONTENTS & CONCEPTS —

Contents

- List of design patterns and how to use them

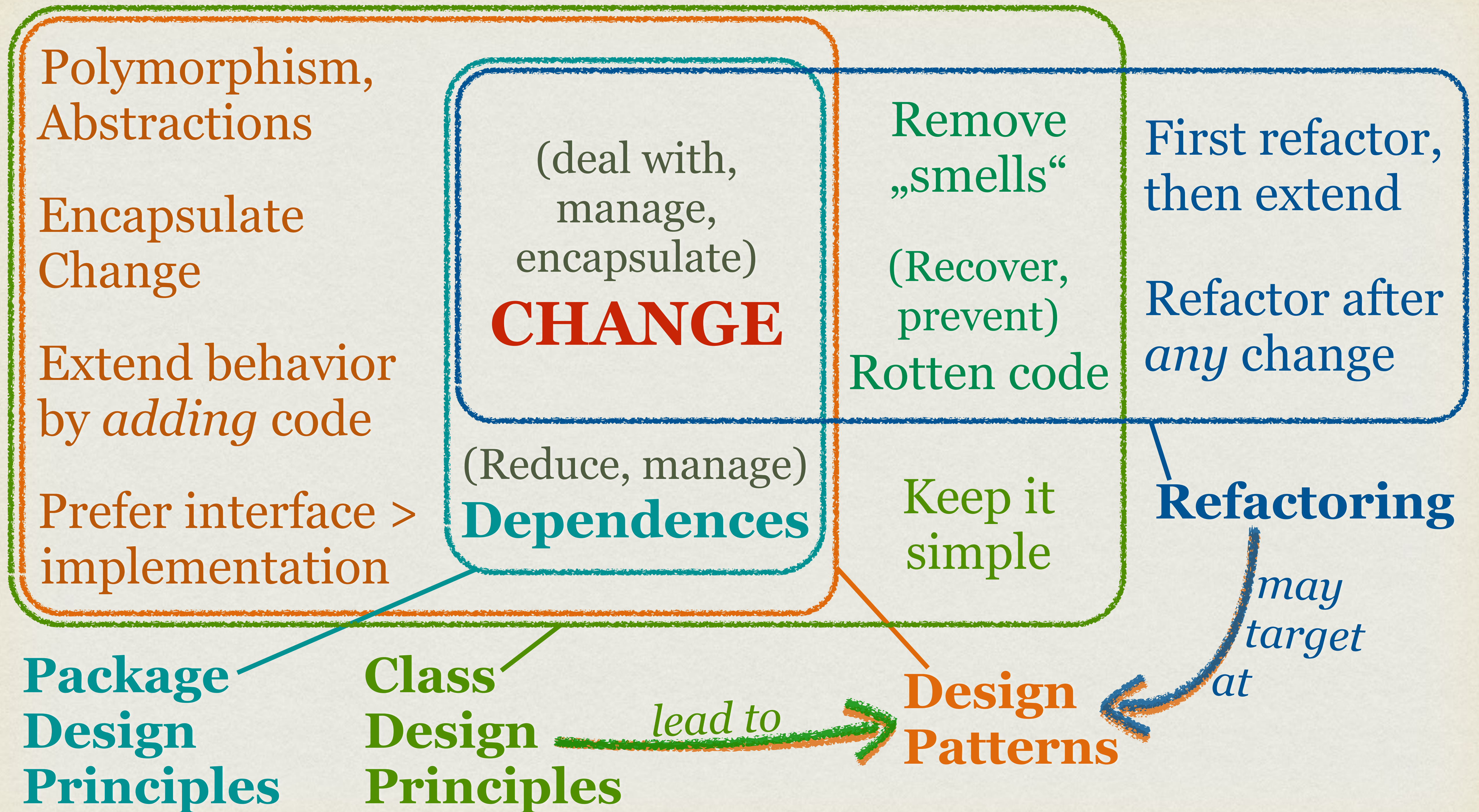
Concepts — How to apply patterns and how they work

- As the results of applying design principles
- As targets for refactoring
- They encapsulate change
- They provide proper abstractions (—>polymorphism)
- They focus on interfaces, not implementations
- They lead to loosely coupled systems (few dependences)

PERFORMANCE — LECTURE TOPICS —

- **Performance and design**
  - How to measure performance and find hotspots
  - The interplay between performance and design
- **Meta-template programming**
  - Various techniques to write fast code using templates
  - Template „design patterns“

HOW IT ALL GOES TOGETHER



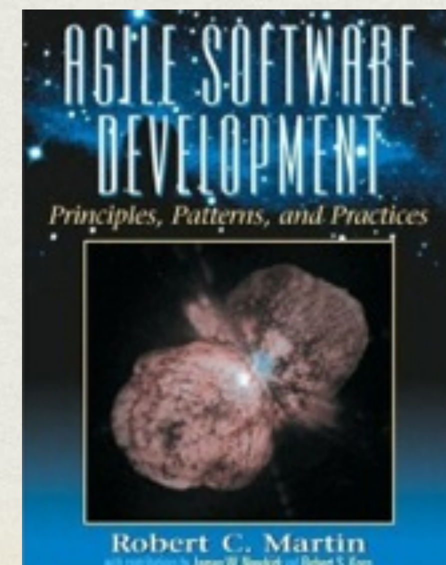
AGILE DEVELOPMENT

Agile Development:

„*A group of s/w development* [i.e. project management] *methods based on iterative and incremental development where requirements and solutions evolve* [...]“ (Wikipedia)

- Refactoring, TDD, OOD principles, Design Patterns all belong to the shared toolkit that is common to these methods

- „Agile Software Development“ (Robert C. Martin) (see slide 2)



HOW IT ALL GOES TOGETHER