Software Architectures

From Design Patterns to Enterprise Architecture

Principles of software design

Last week on “Software Architectures”

- Principles
  - Low Coupling - High Cohesion
  - Liskov Substitution Principle
  - Open-Closed Principle
- Software Quality
  - Testing for Correctness
  - Tests for other quality criteria

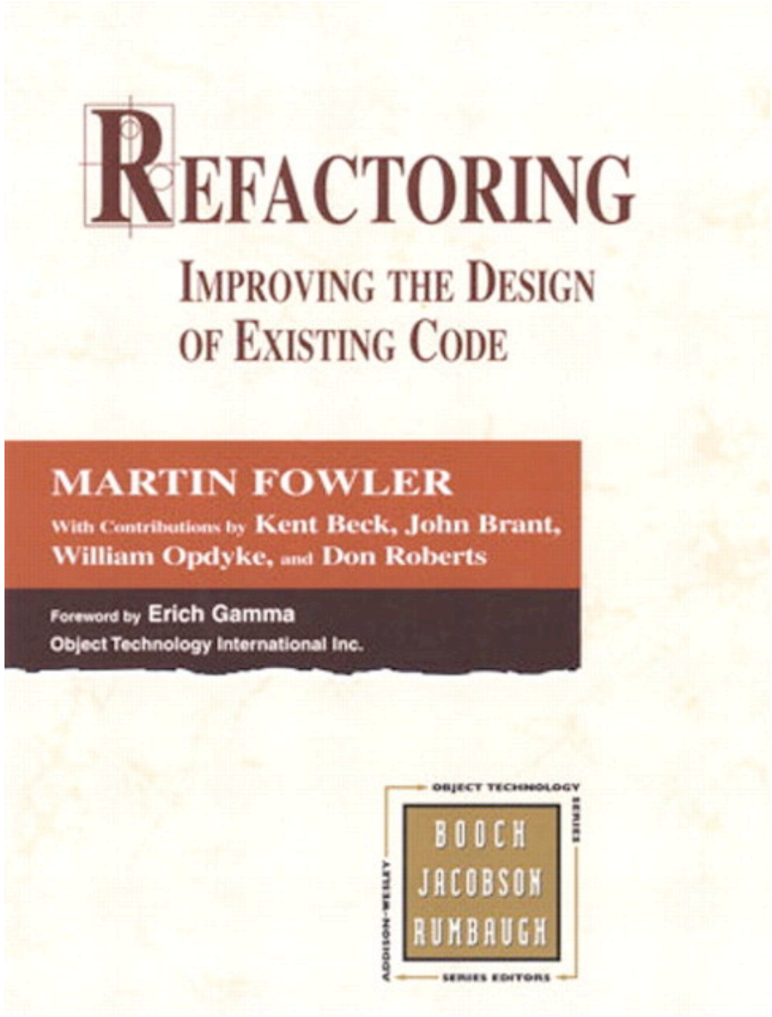
Testing for correctness

Unit tests

- We talked about unit tests, but we didn’t see them yet.
- So let’s write some.

But first…

- This code doesn’t look so good, now does it…
- So let’s do a little bit of refactoring
- Apply the learned principles to the code



REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International Inc.

OBJECT TECHNOLOGY SERIES

BOOCH
JACOBSON
RUMBAUGH

SERIES EDITORS

ADDISON-WESLEY

JUnit Tests

- Annotations: @Before,@After, @Test
- Assert Statements
- Test categories
- Test re-use

Further Principles

Think back to the refactoring we did

- Did everything really we did relate to the principles we learned already?
- To what is the separation of the multiply* methods related?

Responsibility

- UML says: “A contract or obligation of a classifier.”
- Robert Martin says: “Each responsibility is an axis of change. When the requirements change, a change will manifest through a change in responsibility amongst the classes. If a class has multiple responsibilities, it has multiple reasons to change.”
- Well what is valid here?
- Responsibility of DOING and KNOWING: What does a class do and what does a class know.

Evaluation

- Dimensions of Design
  - beautiful vs. ugly
  - good vs. poor
  - efficient vs. inefficient
  - understandable vs. mysterious
- What are your dimensions?
- How do you describe a system?
- Well, there are more formal methods out there.

Discussion

System evaluation

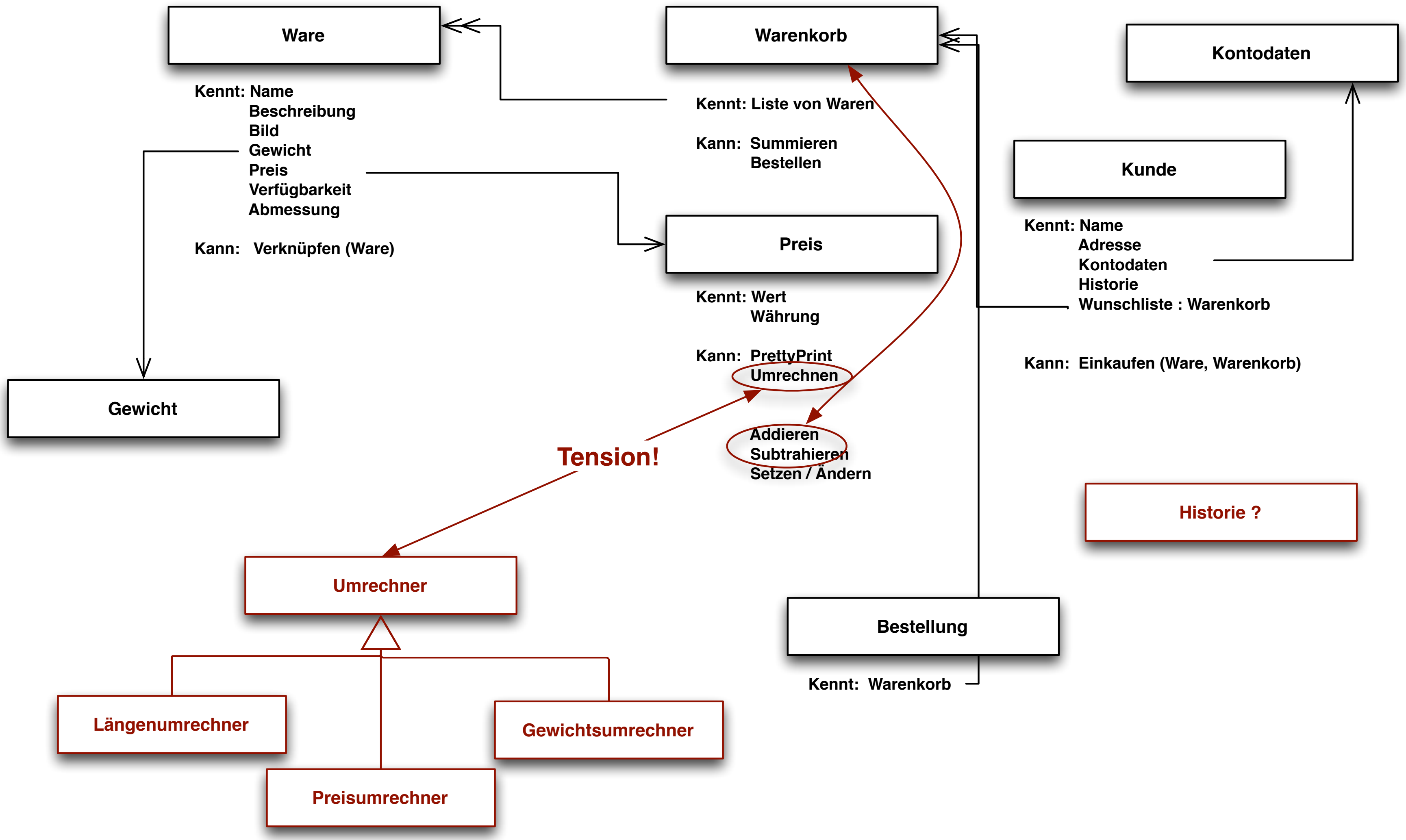
Single-Responsibility Principle

- Every class should have a single responsibility
- That responsibility should be entirely encapsulated by the class
- All its services should be tightly aligned with that responsibility
- What are typical violations?

Interface Segregation Principle

- No client should be forced to depend on methods it does not use
- Role interfaces
- Example: Split interfaces for reading and writing to a resource
- What are typical violations?

Group Exercise: Designing for Responsibility



Ware

Kennt: Name
Beschreibung
Bild
Gewicht
Preis
Verfügbarkeit
Abmessung

Kann: Verknüpfen (Ware)

Warenkorb

Kennt: Liste von Waren

Kann: Summieren
Bestellen

Kontodaten

Kunde

Kennt: Name
Adresse
Kontodaten
Historie
Wunschliste : Warenkorb

Kann: Einkaufen (Ware, Warenkorb)

Preis

Kennt: Wert
Währung

Kann: PrettyPrint
Umrechnen
Addieren
Subtrahieren
Setzen / Ändern

Gewicht

Tension!

Historie ?

Umrechner

Längenumrechner

Gewichtsumrechner

Preisumrechner

Bestellung

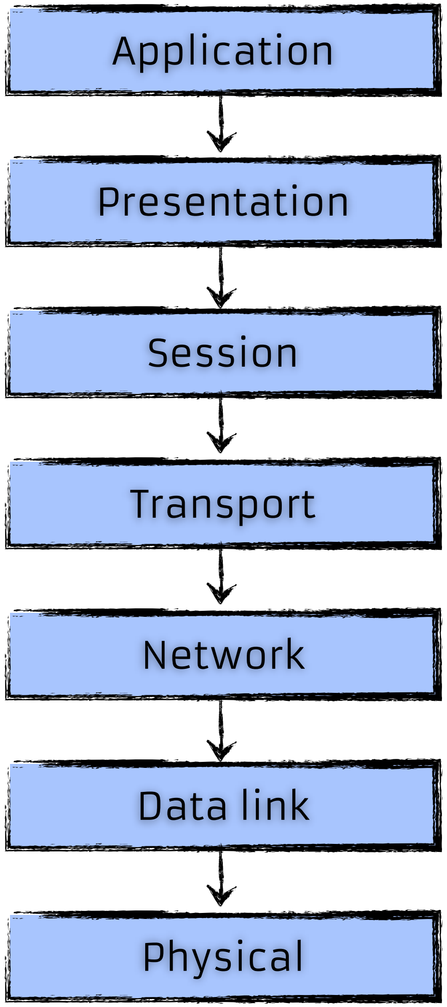
Kennt: Warenkorb

Super principle: Abstraction

- There is nothing that cannot be solved with another layer of abstraction
- There is also nothing that cannot be made more inefficient by another layer of abstraction

Layers and dependencies

- Benefit: Layers have strong responsibility and abstract lower layer away
- Downside: Strict, sometimes efficiency
- What another layered systems do you know?



Dependency Inversion Principle

- High level modules should not depend on concrete low-level modules, both should depend on abstractions
- Abstractions should not depend upon details. Details should depend upon abstractions
- What does such a design look like?

Command-Query Separation

- What is wrong with the upper example?
- Is the lower example well designed?
- Every method should be either a command or a query. Not both at the same time.
- Questions should not change the answer. Schrödinger be gone!

```
public class Counter {
   int i = 0;
   public int getValue() {
      return i++;
   }
}
```

```
public class Counter {
   int i = 0;
   public int getValue(){
      return i;
   }
   public void inc(){
      i++;
   }
}
```

Developing for robustness/reliance?

- Exception handling
  - Checked vs. Unchecked exceptions
  - What to catch?
- Exception throwing
  - Good principles also guide this activity!
- Checks
  - Trade-off between robustness and efficiency
- Assertions
  - Debug-time checks

Wrap up

- Testing for correctness: Unit tests
- Refactoring
- Notion of responsibility
- Architecture evaluation
- Single-Responsibility Principle
- Interface Segregation Principle
- Super principle: Abstractions!
- Dependency Segregation Principle
- Robustness and reliance