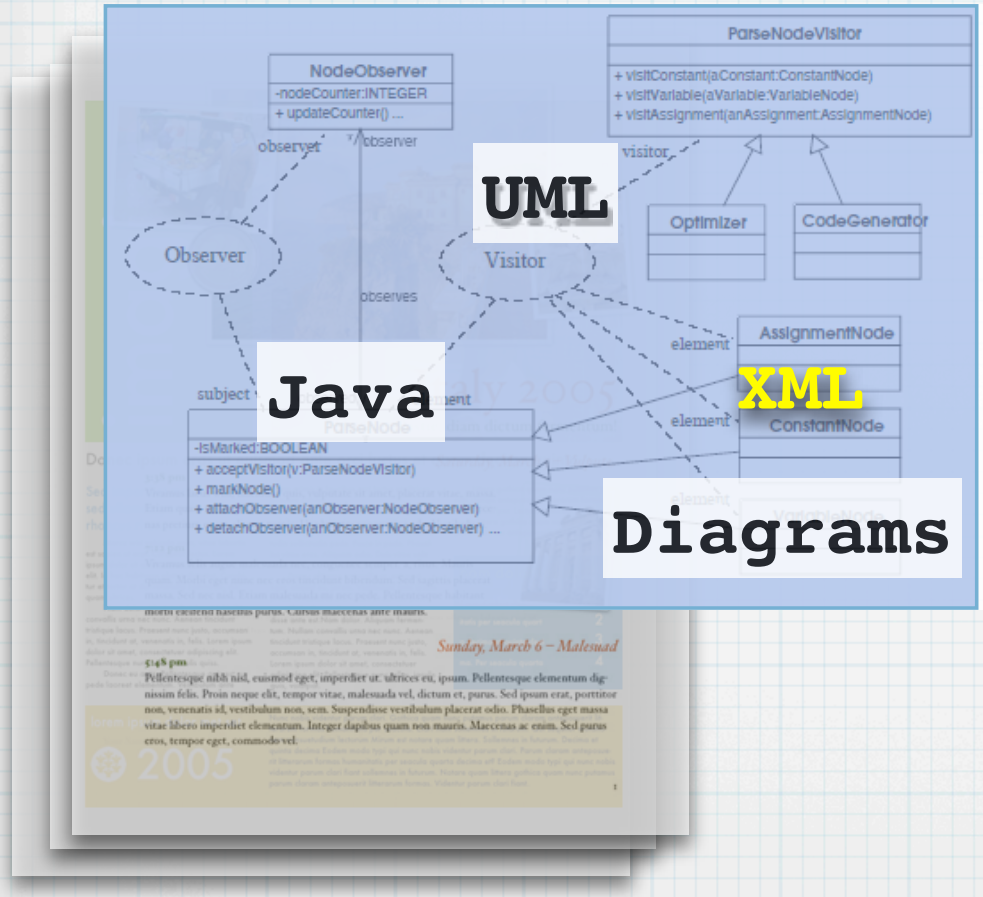
Assertion Checking for Object-Oriented Programs

Kees Huizing
June 13, 2007

What is software?



the documents or the product

Product analysis

* testing
* model checking
* run-time checks
* not our concern here

Document analysis

* documents are what the developers are working on
* documents determine flexibility
* OO has a strong view on decomposition of the software: documents
* Note that this structure need not be present in product (running code)

Facts of SE: maintenance

* corrective (debugging etc.): 17%
* adaptive (to changes in environment): 18%
* perfective (new requirements): 65%
  - Flexibility (i.e., being amenable to change) is essential
  - Claim to fame of OO is (a.o.) this flexibility

Our goal

- specification and verification on the document level
- verification is about behaviour: found in the **code** (not so much in the structure documents)
- allow in proof and specs the same flexibility as OO design
  - do not create additional dependencies

B2. Removing dependencies

- mutual dependencies can often be broken by

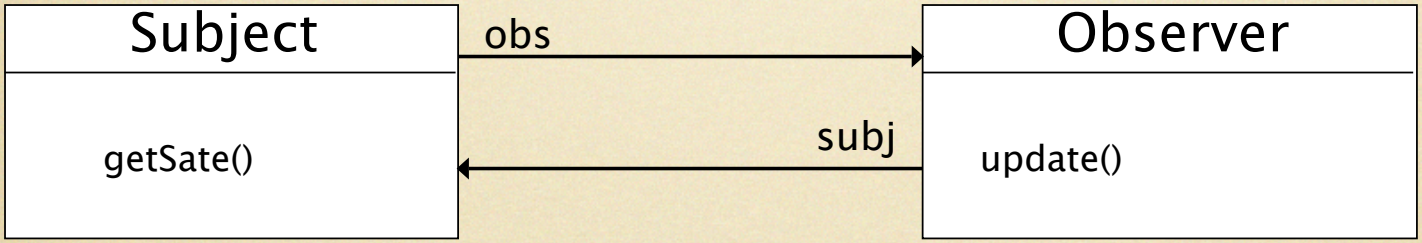
  1. splitting up components, in particular creating *interfaces*

  2. replacing dependencies on full component by weaker dependencies on interfaces

Observer pattern

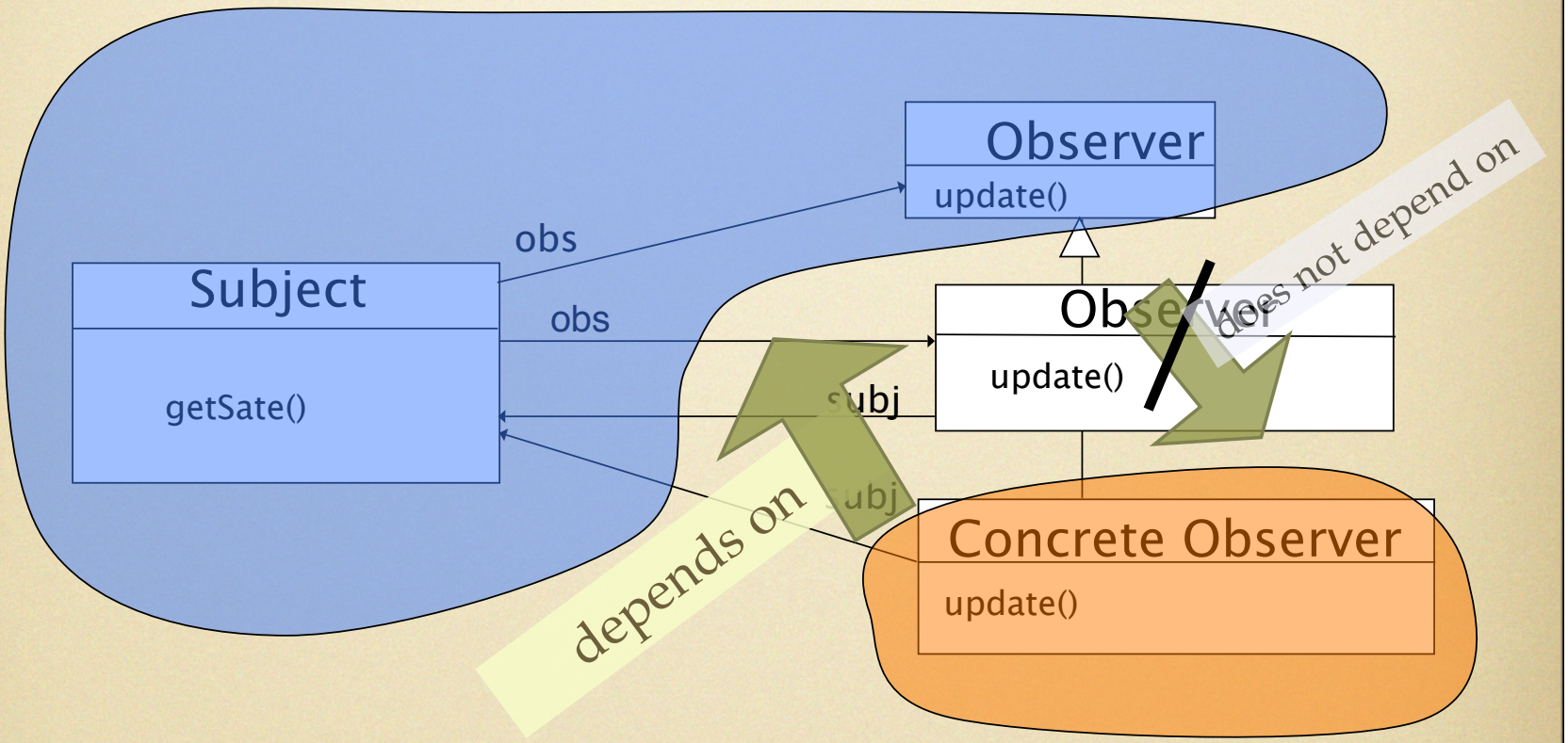
Observer keeps copy (or part of it) of state of Subject
e.g., graphical view of numerical data

data (model) graph, etc. (view)



Mutual Dependency

Breaking mutual dependency



Mutual dependency is replaced by three singular dependencies

Means

- Formal verification in the style of Ontwerp van Algoritmen
  - example:

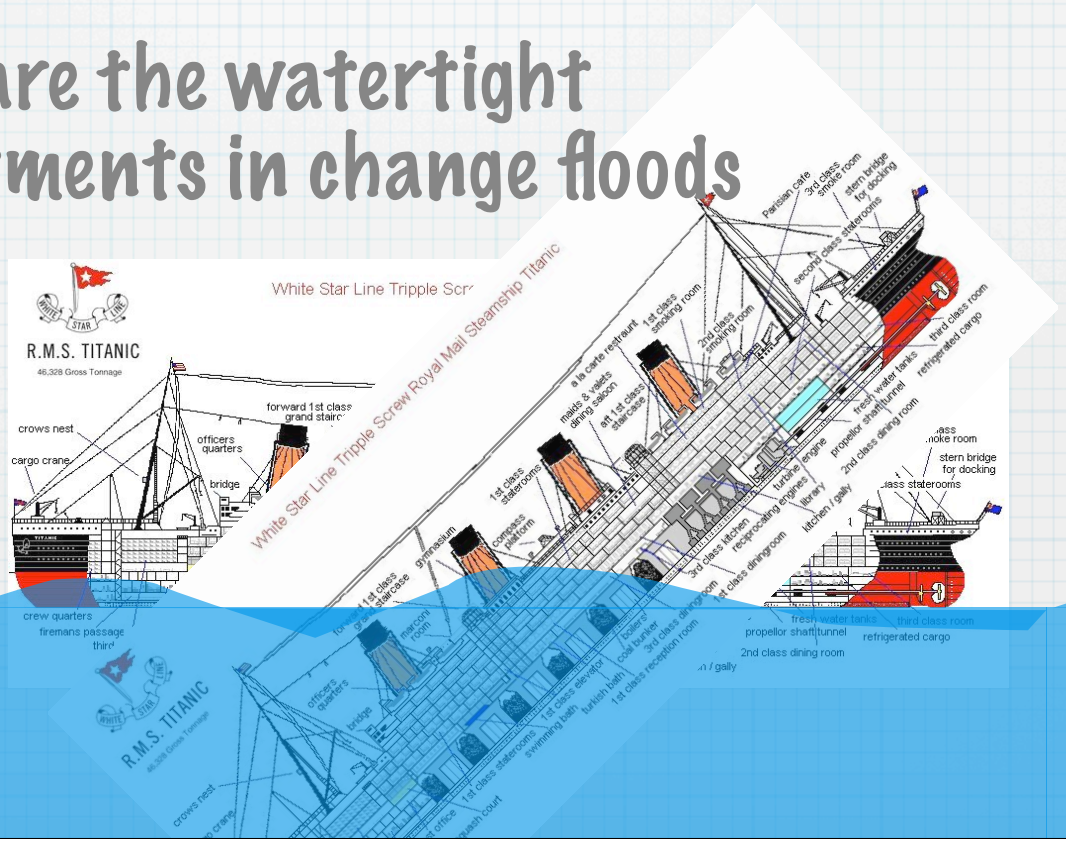
    product level: repetition involves many (unbounded) state changes during execution

    document level: repetition is one statement and has one invariant, fixed number of proof obligations

- Note: "afleiding" (program derivation) not essential; also a posteriori (= enhanced code review)

00

* objects are basic units in the product
* classes are basic units in the code
* classes are the watertight compartments in change floods



How to specify a class?

- different from code fragments (pre/post)
- class describes both:
  - behaviour
  - data

Specifying behaviour

- all methods satisfy their contracts:
  - if started in precondition, will end in postcondition
  - some data remains untouched

Specifying data

- all objects of a class should satisfy the class invariant
- why invariants?
  - documentation (what does this data mean?)
  - data definition (representation invariant)
  - ease of specification
  - put the concerns where they belong:
    - user of object should not be worried with internal consistency

How to prove invariance? (1)

* Data induction.

If for every object in a sequence:

1. **I** holds after object creation
2. for each change from state **s** to **s'** holds:

$$s \models I \Rightarrow s' \models I$$

➡ Then **I** holds for every state in the sequence

How to prove invariance? (2)

Class **C** with invariant **I**

* Prove that **I** holds at the end of each constructor
* Prove for every method **m** of **C**:

  $\{ I \wedge pre \}\ Body_m\ \{ I \wedge post \}$

Notes

Invariants hold only between method calls (so-called **observable states**)

* Caller doesn't have to prove invariant
* Invariant may be assumed after call

These are in fact requirements of the method

```
someCode() {
   Klasse o;
   ...
      // prove o.pre_m
      o.m();
      // assume o.post_m ∧ o.I_C
    ...

}
```

```
class Klasse {
  //@ invariant I_C;
 ...

  void m() {
     // assume this.pre_m ∧ this.I_C
      ...
     // prove this.post_m ∧ this.I_C
  }

   ...
}
```

Problems

* do you see one?

```
class Divider {
  //@ invariant n > 0;
  int n;
  Help h;

  int m(int k) {
    int r = k/n;
    n--;
    h.check( r );
    if (n==0) n = 100;
    return r;
  }
}
```

```
class Help {
    Divider d;
    void check(int i) {
    ...
    d.m(...);
    ...
  }
}
```

Solutions to call-back problem

- forbid call-back statically (layered architecture)
  - rigid
- forbid call-back dynamically (by recording caller)
  - creates dependency callee → caller
- prove invariant before call
  - breaks encapsulation
- prove invariants when leaving object

Problems (ctd)

- Invariants referring to other objects

  next.prev = this

  - changing one object can invalidate another
  - proving $\{ I \wedge pre \}\ Body_m\ \{ I \}$ is not enough

vulnerability

Solutions to vulnerability

* make hierarchical structure and restrict change access (ownership)
* prove all -possibly- violated invariants at critical points :-(
* make dependencies explicit in specification

Methods revisited

* Polymorphism: one word can mean different things
* object variable can hold references to different types of objects
* hence, a.m() can result in calls to different methods m (in case of overriding)
* what precondition do we prove, what postcondition do we assume?

Dynamic binding

* carry the **dynamic type** with you in the assertions and use the corresponding pre/post pair :-(
* prove the pre/post of the **static type** and make sure that
  - precondition of overriding method is not stronger
  - postcondition of overriding method is not weaker

behavioural subtyping

Method call

- naïve proof rule (non-recursive)

for every method m applicable to b

$$\frac{\{this.pre_m \wedge this.I \wedge \ldots\} \quad \text{body}_m \quad \{this.post_m \wedge this.I \wedge \ldots\}}{\{b.pre_m\} \quad \text{b.m()} \quad \{b.post_m \wedge b.I\}}$$

dynamic binding!

- choice of *pre, post, I* depends on dynamic type of *b*

Removing dynamic binding

- Notation:
  $b.P$ dynamic binding
  $b:P$ static binding

dynamic binding
no alternative

$$\frac{\{this : pre_m \wedge this : I \wedge \ldots\} \quad \text{body}_m \quad \{this : post_m \wedge this.I \wedge \ldots\}}{\{b : pre_m\} \quad \text{b.m()} \quad \{b : post_m \wedge b : I\}}$$

Removing dynamic binding

- Notation:
  $b.P$ dynamic binding
  $b:P$ static binding

dynamic binding
no alternative

$$\frac{\{this : pre_m \wedge this : I \wedge \ldots\} \quad \text{body}_m \quad \{this : post_m \wedge this.I \wedge \ldots\}}{\{b : pre_m\} \quad \text{b.m()} \quad \{b : post_m \wedge b : I\}}$$

Assertional tools for Java

- JML: a widely accepted spec language
- uses Java syntax
- added as comment to existing code or in separate files
- many tools exist

ESC/Java

* Extended Static Checking for Java
* uses subset of JML

ESC/Java (ctd)

- modular checking: classes and methods are checked in isolation
- disadvantages:
  - you can not use knowledge of specific usage in application
  - executing proofs requires insight in the code
- advantages:
  - no exponential blow-up of verif. conditions
  - robust to change, ready for reuse

Links

- ESC/Java: http://research.compaq.com/SRC/esc/download.html
- Spec#, Microsoft counterpart to ESC/Java2 tool for C# verification. See: http://research.microsoft.com/specsharp/
- Huizing, K. and Kuiper, R., (2001), Reinforcing fragile base classes, in: Proc. 3rd ECOOP Workshop on Formal Techniques for Java Programs, Budapest.
- R. Middelkoop, C. Huizing, R. Kuiper, E. Luit, Cooperation-based Invariants for OO languages, International Workshop on Formal Aspects of Component Software 2005, Macao, (Electronic Notes in Theoretical Computer Science, Elsevier).
- Sun, K. , Verifying Java Programs by Integrating ESC/Java2 and PVS, MSc, TU/e 2007.