

Writing strong object-oriented applications

Copenhagen, November 2009

Version: 1.0
Status: Final
Author: Thomas Koefoed
tsk@netcompany.com

Agenda

- Introduction and motivation
- Coupling
- Cohesion
- Test-driven development
- Advanced topics
- Questions and evaluation

Personal introduction

- Thomas Koefoed, 27, Senior Architect at Netcompany
- Graduated as Master of Informatics from Technical University of Denmark, March 2007
- Roles on Netcompany projects:
 - Lead Developer (primarily on .NET)
 - Software architect
 - Technical architect
- Netcompany delivers business-critical applications and system integration for medium-or-large companies, primarily in Denmark.

Why object-orientation?

- Encapsulating complexity and reuse of code
 - Switch of technology is easily implemented
 - Adding or optimizing code-units can be done with little effort
- Applications are easier to test
 - Object-orientation promotes cohesion → Test complexity is reduced
 - Test-frameworks exist and simplifies
- Patterns and best-practices provide common development terminology
 - Reduces complexity of reading and understanding of new code
 - Proven solutions for a wide range of known problems
- Etc.

Agenda

- Introduction and motivation
- Coupling
- Cohesion
- Test-driven development
- Advanced topics
- Questions and evaluation

What is coupling?

"The ability of the design to support replacement of components, without having to change the implementation of related components"

Real-life example 1

Tightly coupled application component

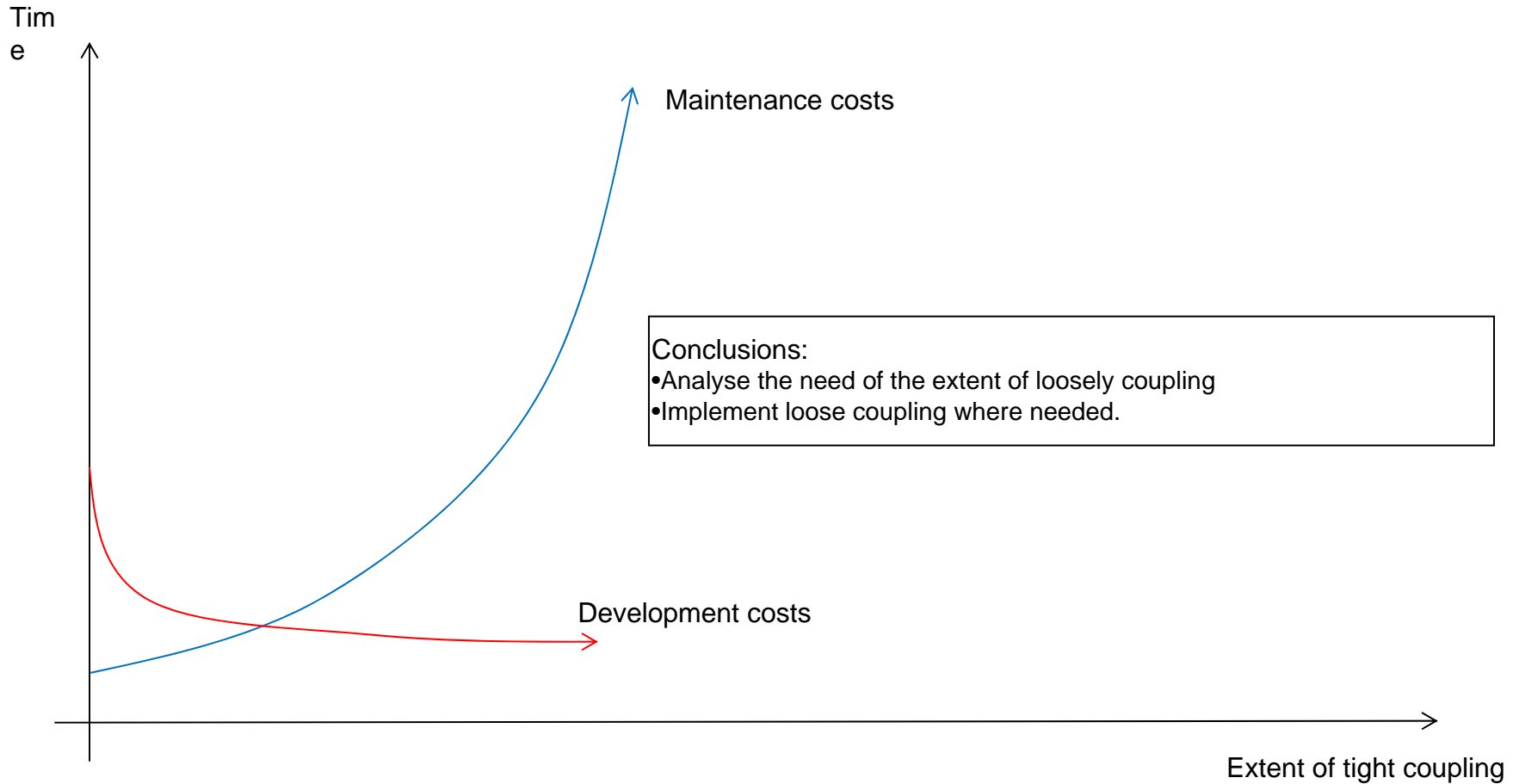
- One (business-critical) use-case implemented by one(!) method in a single class
 - Approximately 10000 lines of code
 - Size of source-code file was the same as a one-minute-MP3 file
 - No custom datatypes, only programming language simple types (int, string etc.)
- The requirements to the component changed
 - Functionality should now support new type of business and have the old-type of business removed
 - All code in method subject to change
 - Estimate of build (assuming OO-design): 1500 man hours
 - Reality of build (no OO-design): > 7000 man hours

Real life example 2

Loosely coupling overkill

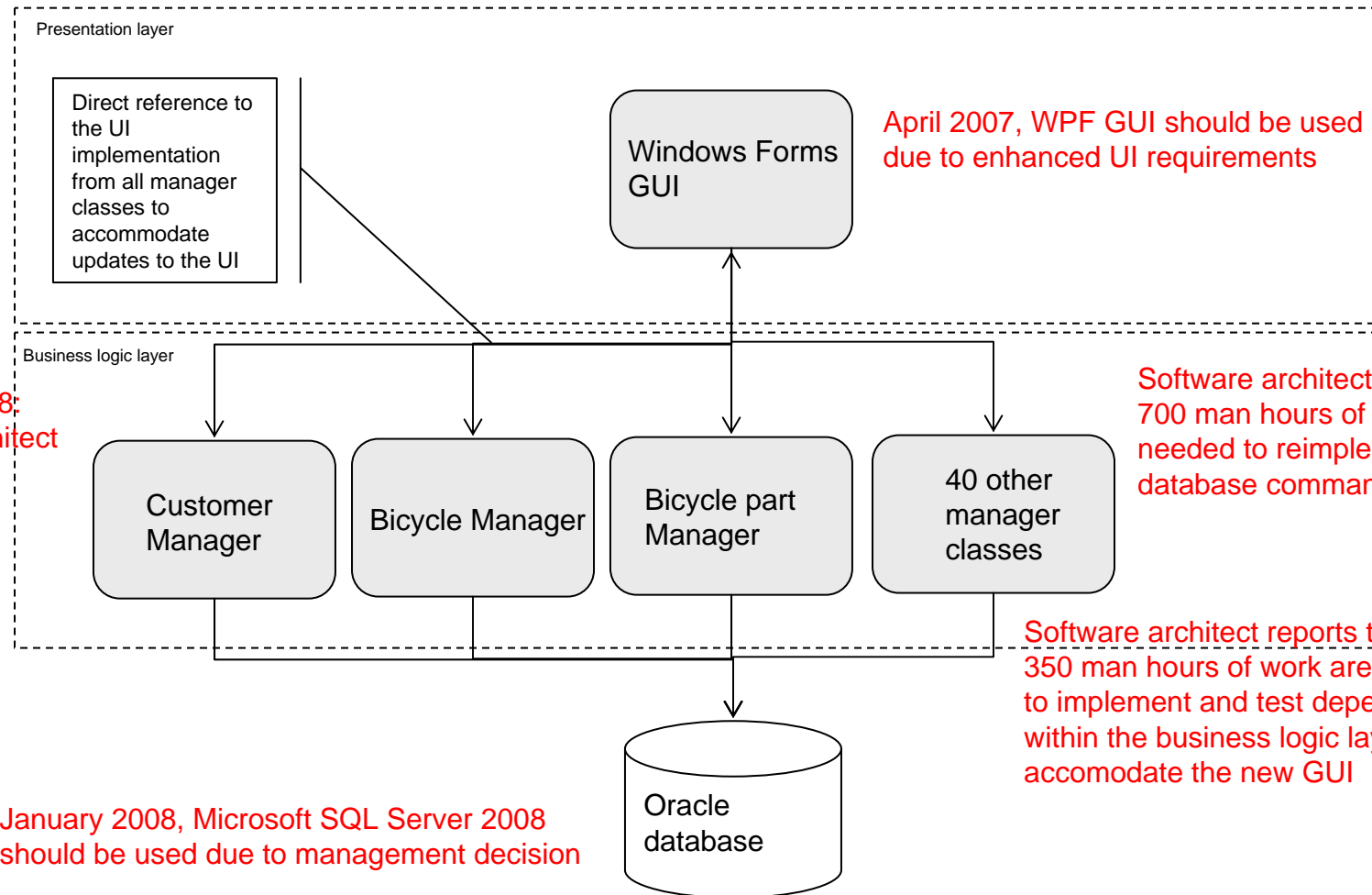
- Development of case-worker portal
 - To be used for one client only.
 - Limited functionality
 - Functionality is almost certainly not going to change radically for the next 10 years
 - Layout has been decided to be web-based (by the management)
- Solution
 - VERY generic case-worker framework
 - Fully configurable and loosely coupled – ready for new clients, technologies
 - Many many abstract classes and interfaces defines the layers of the architecture
 - Estimated man hours: 40000 (estimated specialized solution: 10000 hours)

Cost of tightly coupled systems



Tightly coupled system (bicycle dealer class diagram)

Made-up example



April 2007, WPF GUI should be used due to enhanced UI requirements

Software architect reports that 700 man hours of work is needed to reimplement the database commands

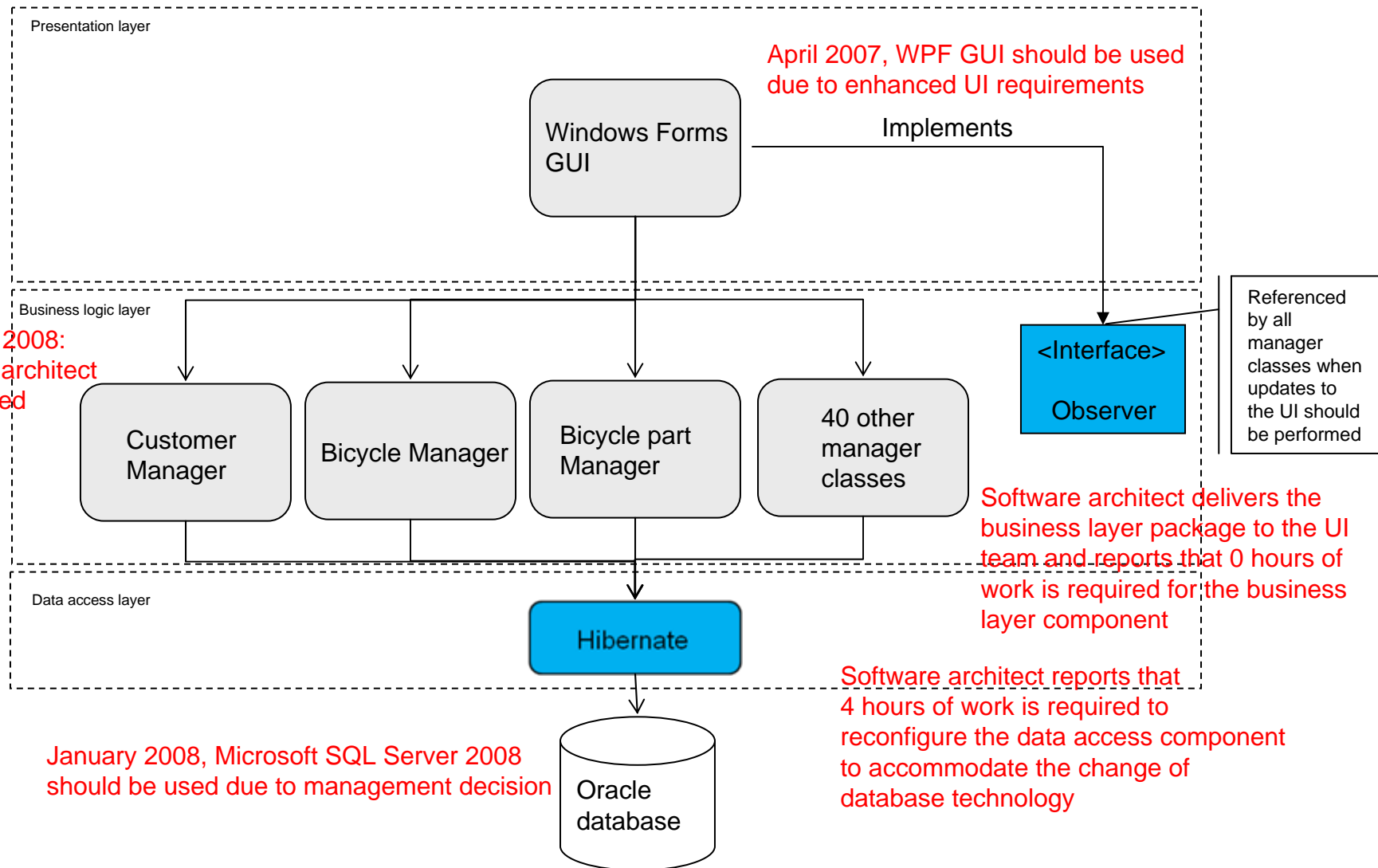
Software architect reports that 350 man hours of work are needed to implement and test dependencies within the business logic layer to accommodate the new GUI

January 2008, Microsoft SQL Server 2008 should be used due to management decision

February 2008: Software architect is fired

Loosely coupled system (bicycle dealer class diagram)

Made-up example



Loose coupling

Things to look for

- References to a class from a large number of client classes
 - Use interfaces to define the contract that the single class should fulfill and let the client classes reference the interface rather than the implementation
- Technology specific code, such as DB commands, scattered in many classes
 - Encapsulate code in classes in a separate package of the application
 - Even better: Use 3rd party or standard frameworks/components
- References to methods in packages in a higher layer of the architecture
 - Use the observer pattern shown in the bicycle dealer example
 - There should ALWAYS only be references from a layer to its immediate successor down the architecture (see bicycle dealer example).

Agenda

- Introduction and motivation
- Coupling
- Cohesion
- Test-driven development
- Advanced topics
- Questions and evaluation

What is cohesion?

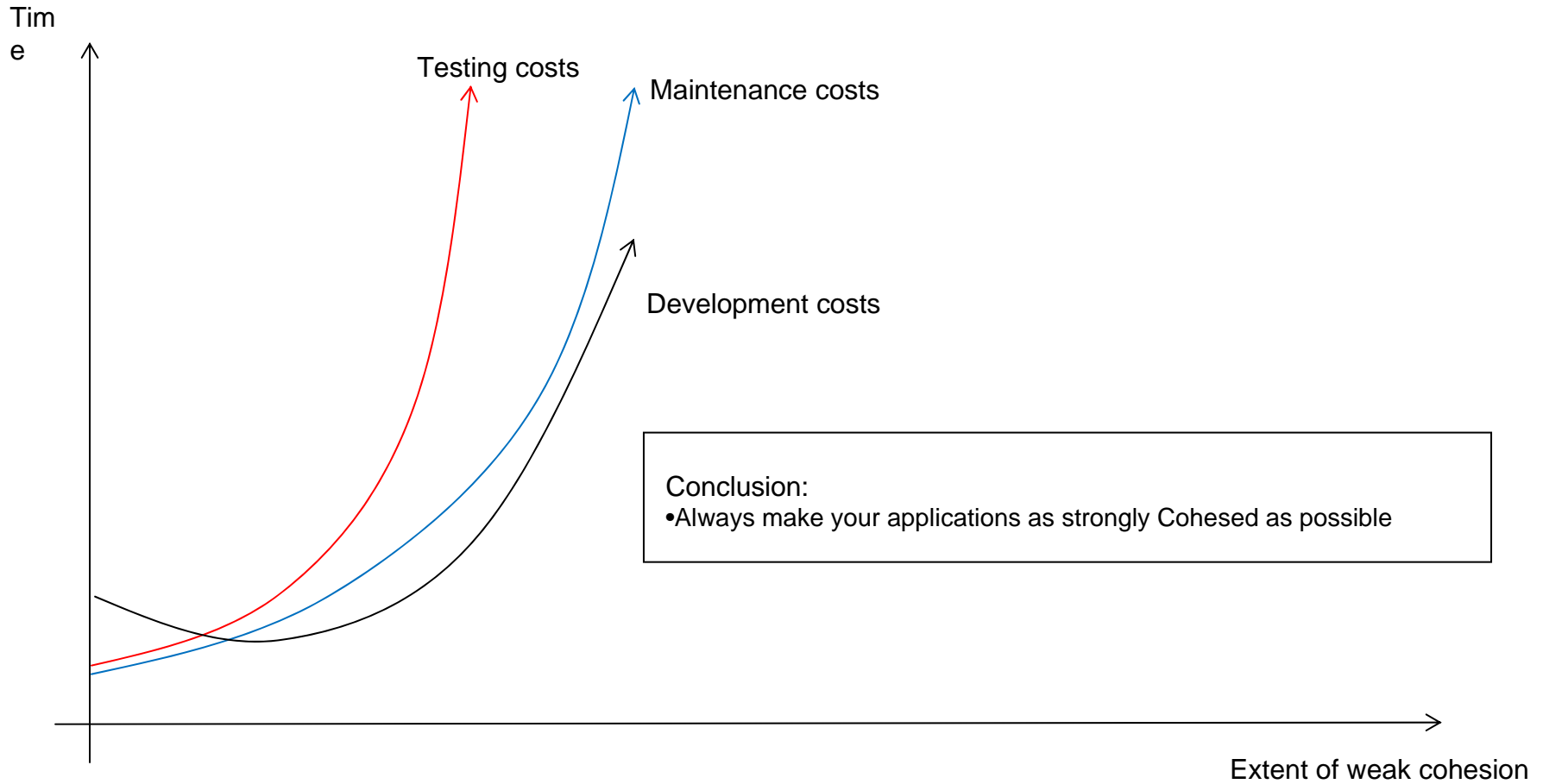
"Assigning the proper responsibilities to classes and methods, ensuring as much reusability and as little redundancy as possible"

Real-life example (same as earlier)

Weakly cohessed application component

- One (business-critical) use-case implemented by one(!) method in a single class
 - Approximately 10000 lines of code
 - Size of source-code file was the same as a one-minute-MP3 file
 - No custom datatypes, only programming language simple types (int, string etc.)
- The requirements to the component changed and the component has been reimplemented
 - All uses of the method should be tested
 - Input and expected output is almost impossible to analyse and produce
 - Estimate of test (based on OO-assumption): 750 hours
 - Reality of test: ? 😊

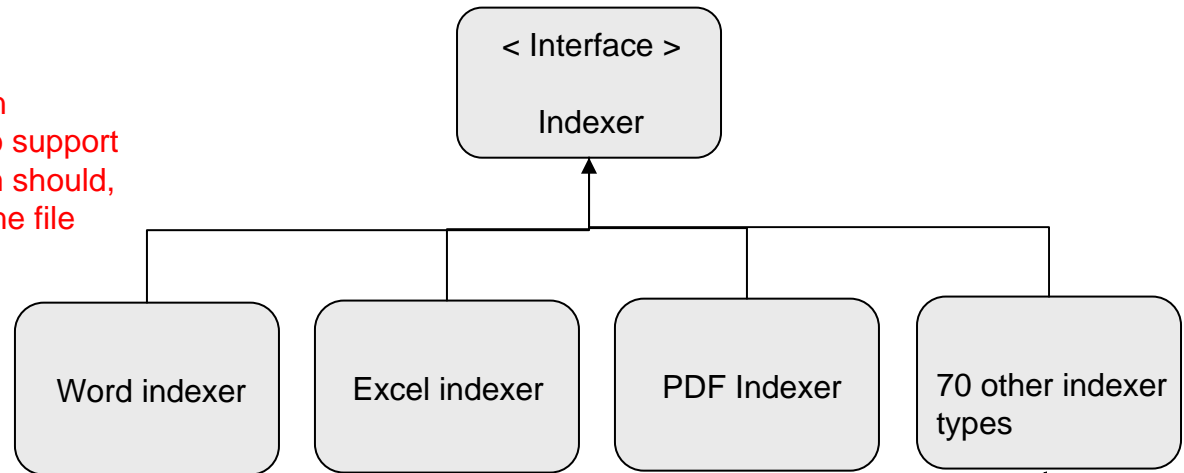
Cost of weakly cohesed applications



Weakly cohesed system

February, 2009:

Due to changed security and presentation Requirements, the indexer must now also support Security trimming. The indexing algorithm should, Therefore be able to access the ACL of the file And incorporate this into the algorithm



The software architect reports that 100 man hours should be allocated for the enhancement

```

public class WordIndexer Implements Indexer
  public ArrayList<ContentKey> IndexContent(String folder) {
    HashMap<WordDocument,Integer> rankings = new HashMap<WordDocument,Integer>();
    ArrayList<ContentKey> keys = new ArrayList<ContentKey>();
    for(WordDocument w : GetDocuments(folder)) {
      int rank = RankDocument(w);
      rankings.put(w, rank);
      keys.add(ProcessContents(w,rank));
    }
    return keys;
  }
  }
  
```

Each implementing class implements a similar algorithm, where only the WordDocument type is replaced by similar specialized types (ExcelDocument, PDF Document etc.)

Strongly cohesed system

```

public abstract class Indexer {
    public ArrayList<ContentKey> IndexContent(String folder) {
        HashMap<FileType, Integer> rankings = new HashMap<FileType, Integer>();
        ArrayList<ContentKey> keys = new ArrayList<ContentKey>();
        for(FileType w : GetFiles(folder)) {
            int rank = RankFile(w);
            rankings.put(w, rank);
            keys.add(ProcessContents(w,rank));
        }
        return keys;
    }

    public FileType[] GetFiles(String folder) {
        <Implementation>
    }

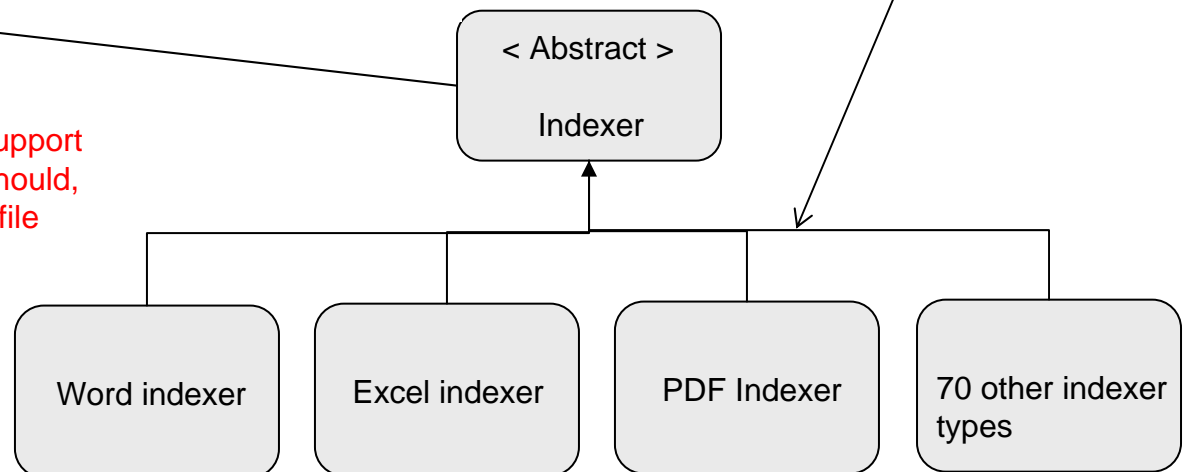
    public abstract int RankFile(FileType d);

    public abstract ContentKey ProcessContents(FileType w, int ranking);
}
  
```

The software architect reports that 12 man hours should be allocated for the enhancement

All specific indexers implements just the operations that are specific to their file types. The index algorithm is implemented in the abstract base class

February, 2009:
 Due to changed security and presentation Requirements, the indexer must now also support Security trimming. The indexing algorithm should, Therefore be able to access the ACL of the file And incorporate this into the algorithm



Keep methods short

```
public abstract class Indexer {  
    public ArrayList<String> GetEmailAccountNames() {  
        ArrayList<String> emails = new ArrayList<String>();  
  
        for(String email : GetEmails("http://exchangeserver1.com")) {  
            emails.add(email.split("@")[0]);  
        }  
        for(String email : GetEmails("http://exchangeserver2.com")) {  
            emails.add(email.split("@")[0]);  
        }  
        for(String email : GetEmails("http://exchangeserver3.com")) {  
            emails.add(email.split("@")[0]);  
        }  
        return emails;  
    }  
}
```



What happens if the processing logic for email accounts are changed?
How many places in the code should be changed?

This design is easier to test and read, as its responsibilities are clearer defined

```
public abstract class Indexer {  
    public ArrayList<String> GetEmailAccountNames() {  
        ArrayList<String> emails = new ArrayList<String>();  
  
        AddEmails(emails, "http://exchangeserver1");  
        AddEmails(emails, "http://exchangeserver2");  
        AddEmails(emails, "http://exchangeserver3");  
        return emails;  
    }  
  
    public void AddEmails(ArrayList<String> emails, String server) {  
        for(String email : GetEmails(server)) {  
            emails.add(email.split("@")[0]);  
        }  
    }  
}
```



Cohesion

Things to look for

- Redundant implementations among classes
 - Usually created by copy-pasting between classes
 - Use object oriented concepts like base- and abstract classes to eliminate redundancy between classes.
- Classes and methods are difficult to read and understand
 - Make classes and methods responsible only for what they are intended to
- Long methods exist in one or more classes
 - Extract code-pieces to private methods to improve readability
- Implementations of an algorithm are scattered within the class.
 - Can be refactored to private methods to empower stronger cohesion and thereby better testing

Agenda

- Introduction and motivation
- Coupling
- Cohesion
- Test-driven development
- Advanced topics
- Questions and evaluation

What is test driven development?

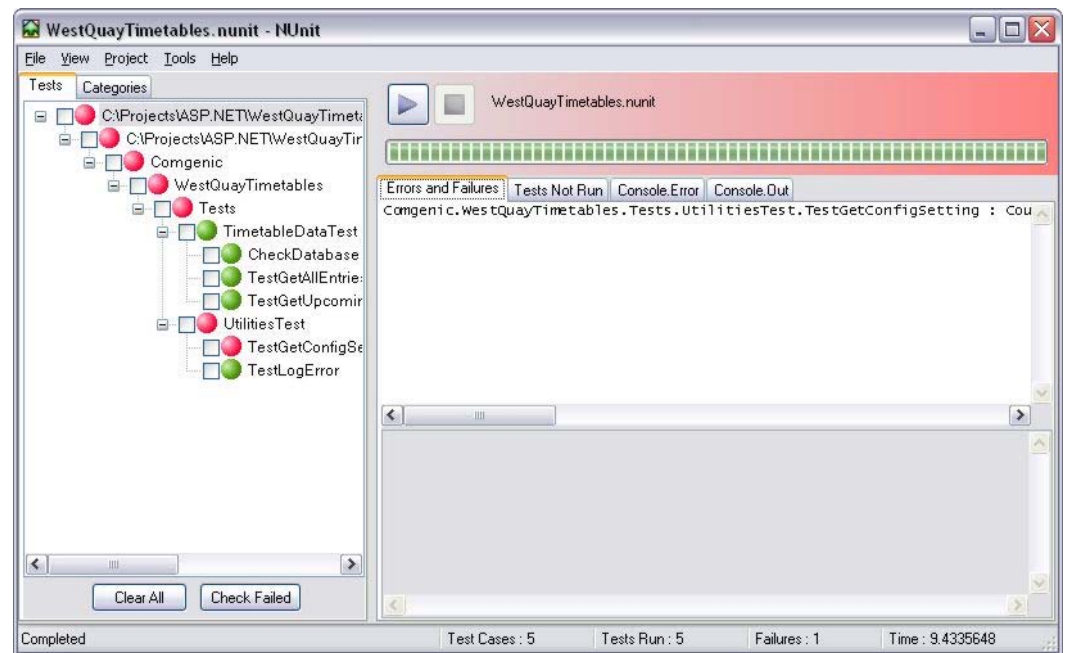
- Unit tests are created for all business logic
 - Unit tests are tests that are automatically run and verified by a test-engine
 - Usually there is a 1-1 relationship between a method and a unit test
- Unit tests are written *before* the business logic is implemented
 - Seems irrational and irritating at first
 - Enforces the programmer to think about each methods responsibility
 - Promotes the cohesion of the design (if the programmer cannot write the test before the implementation, the business logic method is probably to complex and should be broken down into more methods)
- Unit tests verifies the validity of methods that are reimplemented
 - Methods tend to be reimplemented due to performance bottlenecks, change of technology or other unexpected issue.

Why test driven development?

- Is proven to raise the level of cohesion
- Enforces the developer to think in method signatures rather in method implementation
- Ensures the quality of the developed software. Also when methods are reimplemented
- Identifies who, in a team of developers, has introduced errors.

Implementation of test driven development

- Use test frameworks such as Junit
- Design unit tests before method implementations
- Unit tests must be completely independent of existing data.
 - Use mocking if necessary
- Ensure that unit-tests are run at each checkin of the code to a source control
- Use IDE plugins for unit-testing



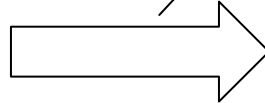
Example of test driven developmen

```

public Boolean IsPrime(int n) {
    boolean prime = true;
    for (long i = 3; i <= n; i += 2) {
        if (n % i == 0) {
            prime = false;
            break;
        }
    }
    if ((n%2 !=0 && prime && n > 2) || n == 2) {
        return true;
    } else {
        return false;
    }
}

```

The minus inserted by accident would be very hard to catch, if no unit test existed for the method



```

public Boolean IsPrime(int n) {
    boolean prime = true;
    for (long i = 3; i <= -Math.sqrt(n); i += 2) {
        if (n % i == 0) {
            prime = false;
            break;
        }
    }
    if ((n%2 !=0 && prime && n > 2) || n == 2) {
        return true;
    } else {
        return false;
    }
}

```

Agenda

- Introduction and motivation
- Coupling
- Cohesion
- Test-driven development
- Advanced topics
- Questions and evaluation

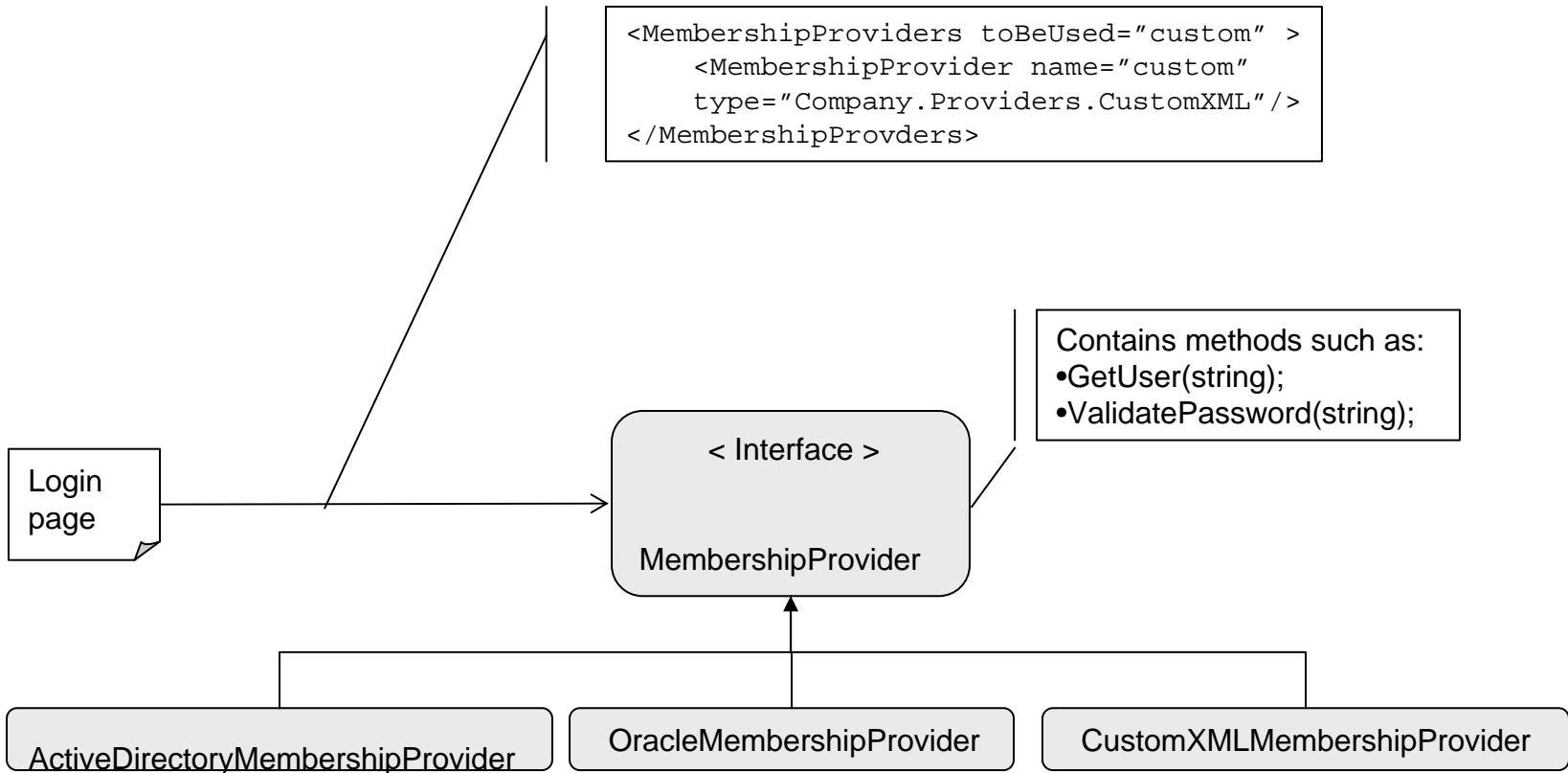
Reflection

What is reflection?

- Types are normally loaded when the runtime (JVM) loads the application.
- Members are normally invoked on the loaded types by the JVM
- Reflection makes dynamically loading of types and invoking type-members possible.
 - `person._name = "Marc"`
 - `Typeof(person).GetField("_name").SetValue(person, "Marc");`
- Reflection performs worse than typesafe loading and cannot be verified by the compiler, and should thus be used appropriately.
- Reflection can be used to "hack" 3rd party component, whose scope denies access to needed functionality.
 - Reflection can be used to invoke private and protected members.
 - Use with extreme caution!
- Reflection is ideal for provider frameworks.

Example of reflection

A membership provider framework



Design patterns

- Design patterns are templated ways of solving a given functional or technical design challenge
- Design patterns are proven ways of designing components in an application
- Design patterns enforces loose coupling and/or strong cohesion
- Design patterns should only be applied where necessary
 - Sometimes the flexibility they provide is not necessary (case-worker example)
- Design patterns ease the process of new developers reading and understanding code in the application

Examples on design patterns

- Observer (shown in first bicycle dealership example)
- Template method (shown in second bicycle dealership example)
- Visitor (can be used to apply different rendering logic to the application)
- Abstract factory (can be used to create sets of related objects. An example is rendering controls in different environments)
- Singleton (used to create an application wide single-instance of an object. An example is a cache).

- Design pattern bible: "Gang of four"-book
 - Solid foundation to improve OO skills
 - Handbook assist when facing OO-challenges

Agenda

- Introduction and motivation
- Coupling
- Cohesion
- Test-driven development
- Advanced topics
- Questions and evaluation

Evaluation (handouts in danish)

- "Det faglige indhold" (The educational content)
 1. The lecture matches the educational level of the course
 2. The subject was relevant to the course
 3. The lecture helps my understanding of the course
 4. The lecture is relevant according to my wishes for a future job

- "Formidling" (Presentation technique)
 1. The form and agenda of the lecture
 2. The amount of content of the lecture
 3. The ability of the lecturer to explain the subject
 4. The motivation of the lecturer
 5. The possibility to ask questions during and after the lecture

- "Gæsteforelæsningsen generelt" (General)
 1. Your general evaluation of the lecture

- Netcompany
 1. Would you be interested in meeting Netcompany in other contexts?
 - If yes, then which?

- Other comments and suggestions

- Grade between 1 and 5, where
 - 1 is very unsatisfied
 - 5 is very satisfied