LOG8430: Architecture logicielle et conception avancée

Modeling, OO Concepts, and Design Patterns

Automne 2017

Fabio Petrillo
Chargé de Cours

1

Page 2

Graphical Modeling Languages

2

Page 3

Graphical Modeling Languages to SA

- A **modeling language** is any **artificial language** that can be used to express information or knowledge or systems in a **structure** that is defined by a **consistent** set of **rules**. The rules are used for interpretation of the **meaning** of components in the structure.
- **Graphical** modeling languages use a **diagram technique** with named **symbols** that represent **concepts** and lines that connect the symbols and represent **relationships** and various other **graphical notation** to represent constraints.

Page 4

Graphical Modeling Languages to SA

- Unified Modeling Language (UML)
- Business Process Model and Notation (BPMN)
- ArchiMate

Page 5

Unified Modeling Language (UML)

- Visual modeling
- Language
- Specify, visualize, construct, document
- Standardization (OMG)
- 1995 ~ 20 years
- Better than nothing!

Page 6

Page 7

Unified Modeling Language

- Structural
  - Static view - Class diagram
  - Use case view – Use case diagram
  - Design View – Component diagram
- Dynamic
  - State machine diagram
  - Activity diagram
  - Sequence/communication diagram
- Physical
  - Deployment view – Deployment diagram

Use case diagrams

- Use case diagrams describe what a system does from the standpoint of an external observer.
- The emphasis is on what a system does rather than how.
- Use case diagrams are closely connected to scenarios.

http://edn.embarcadero.com/article/31863

Page 9

Use case diagrams

http://edn.embarcadero.com/article/31863

Page 10

Class diagram

- Main OO diagram
- Static aspects - and relationships
- Classes and Relationships (including inheritance, aggregation, and association)
- Methods and attributes

Page 11

Class diagram - Domain Model

http://www.uml-diagrams.org/class-diagrams-overview.html

Page 12

Class diagram - Implementation Classes

http://www.uml-diagrams.org/class-diagrams-overview.html

Page 13

Class diagram

http://edn.embarcadero.com/article/31863

Page 14

Class diagram

http://agilemodeling.com

Page 15

Class diagram

http://agilemodeling.com

Class diagram - Relationships

Component Diagram

- architecture-level artifact
- high-level software components
- interfaces to those components
- Components -> Services -> microservices
- Very useful and powerful

Page 18

Component Diagram

http://www.uml-diagrams.org/component-diagrams.html

Page 19

Component Diagram

http://agilemodeling.com

Page 20

Sequence Diagram

- flow of logic within your system
- most popular artifact for dynamic modeling

- temporal sequence
- useful to represent **layers**

Page 21

Sequence Diagram

http://edn.embarcadero.com/article/31863

Page 22

Sequence Diagram

http://www.uml-diagrams.org/sequence-diagrams.html

Page 23

Sequence Diagram

http://agilemodeling.com

Page 24

Sequence diagram (robustness)

http://agilemodeling.com/artifacts/sequenceDiagram.htm

Page 25

Sequence Diagram - Issues

http://agilemodeling.com

Package Diagram

- organize model elements into groups
- organize classes, data entities, or use cases

http://edn.embarcadero.com/article/31863 http://holub.com/uml/

Page 27

Package Diagram

http://www.uml-diagrams.org/package-diagrams-overview.html

Page 28

Package Diagram - Model or Structure Diagram

http://www.uml-diagrams.org/package-diagrams-overview.html

Page 29

Package Diagram

http://agilemodeling.com

Deployment Diagram

- Deployment diagram is a structure diagram which shows architecture of the system as deployment (distribution) of software artifacts to deployment targets.
- physical view
- run-time configuration
- nodes and the components that run on those nodes
- technological decisions
- hardware, software and networking

Page 31

Deployment Diagram - Components by Artifacts

http://www.uml-diagrams.org/deployment-diagrams-overview.html

Page 32

Deployment Diagram - Specification Level

http://www.uml-diagrams.org/deployment-diagrams-overview.html

Page 33

Deployment Diagram - Instance Level

http://www.uml-diagrams.org/deployment-diagrams-overview.html

Page 34

Deployment Diagram

http://edn.embarcadero.com/article/31863

Page 35

UML - Some online resources

- **http://www.uml-diagrams.org**
- http://edn.embarcadero.com/article/31863
- http://holub.com/uml

Page 36

Business Process Model and Notation (BPMN)

- Business Process comprehension - important step on SA modeling
- Business Process Model and Notation (BPMN) is a **standard** for business process modeling
- Maintained by the Object Management Group since 2005
- Graphical notation for specifying business processes
- A **flowcharting** technique very similar to **activity diagrams** from Unified Modeling Language (UML), adding a **rich semiotic** (symbols) to express business processes

Page 37

Basic flow objects and connecting objects

Event

Activity

Connection

Gateway

Business Process Modeling Notation (BPMN)

End event

Swimlane

Start event

Annotation

Data object

Page 39

BPMN - Some online resources

- **http://www.omg.org/spec/BPMN/2.0**
- http://www.bpmb.de/images/BPMN2_0_Poster_EN.pdf

Page 40

ArchiMate modeling language

- The ArchiMate modelling language is an open and independent Enterprise Architecture standard
- Supports the description, analysis and visualisation of architecture within and across business domains.
- Open standards hosted by The Open Group and is fully aligned.
- Open source modeling tool - Archi
- Three main layers in a single diagram
    - Business layer
    - Application layer
    - Technology layer
- Archi can be a good **alternative** to UML/BPMN

ArchiMate - Business layer

ArchiMate - Application Layer

Page 43

ArchiMate - Technology layer

Page 44

Process and
business layer

Application Layer

Technology Layer

https://www.archimatetool.com

ArchiMate - Some online resources

- **http://pubs.opengroup.org/architecture/archimate2-doc**
- https://www.archimatetool.com

Page 47

Architectural Modeling

- Use a **standard notation** or describe **our** notation in the diagrams
- **Decide your project modeling language notation and follow it!**
- Just enough modeling -> Modeling versus Documenting
- CASE tools
  - Several tools: ArgoUML, Astah, Visual Paradigm, Archi Tool, ….
  - Documenting
- Suggestion: work with two monitors. :-)

Page 48

Object-Oriented Design

Page 49

Object-Oriented Design

- Object oriented concepts
- Principles of Modular Design
- SOLID principles

Page 51

Topology First/Second Languages

Page 52

Topology Object-Oriented

Page 53

Topology Object-Oriented (Large)

Object-oriented Concepts

- Programs are organized as cooperative collections of objects
- Class – representation of object
- Hierarchy of classes united via inheritance

relationships

OO Concepts

- Abstraction
  - type (taxonomy)
  - interface
  - contract
- Encapsulation
- Inheritance (reuse of code, subtyping)
  - Class+imperative+Strongly-typed = Java
  - Prototype+Functional+Dynamically = Javascript
- Polymorphism

Page 56

Polymorphism

- More than one form
- Powerful
- Flexibility
- Extensibility
- Reusability
- Central of OO design
- 21 of 23 GoF Design Patterns use polymorphism!

Page 57

Polymorphism

- Implementation of abstract methods
- Overriding
  - specific implementation over already provided by super/parent class
- Overloading
  - Same name, but different signature

Page 58

Abstract Class or Interface?

- What is the difference?
- When use Interface?
- When use Abstract Class?

Abstract Class or Interface?

- They are different mechanisms
- When use Interface?
  - Taxonomy and contract
- When use Abstract Class?
  - package reusable data and behaviours
  - propose an “hereditary” contract

Page 60

Duck typing - problem

Page 61

Duck typing - Solution

- Favour composition over inheritance
- Allow changing implementation
- Allow safe inheritance
- **Add one level of indirection**

Page 62

Add one level of indirection

Principles of Modular Design

- Cohesion
- Coupling

Page 64

Coehsion

- Cohesion is a measure of how strongly-related or focused the responsibilities of a single module
- Higher is better

Page 65

Coupling

- Coupling or dependency is the degree to which each program module relies on each one of the other modules
- Coupling – lower is better

Principles of Modular Design

- Our main OO design goal
  - Increase cohesion

- Decrease coupling

SOLID Principles

- “First five principles” - Robert C. Martin

- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle

Interface segregation principle

Dependency inversion principle

67

Page 68

68

Page 69

Single responsibility principle

- Every module/class should have responsibility over a single part of the functionality provided by the software
- Responsibility should be entirely encapsulated by the class.
- All its services should be narrowly aligned with that responsibility.
- **A class should have only one reason to change.**
- Principle of cohesion

http://www.javabrahman.com/programming-principles/single-responsibility-principle-with-example-in-java/

Page 71

Close/Open Principle

- Classes, modules, functions, etc. should be open for extension, but closed for modification
- Entity can allow its behaviour to be extended without modifying its source code
- Composition and inheritance
- Abstract base classes
- Duck type discussion

Liskov Substitution Principle

- Barbara Liskov

Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.

- Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Page 74

Liskov Substitution Principle

-
-

http://www.javabrahman.com/programming-principles/liskov-substitution-principal-java-example/

Page 75

Interface Segregation Principle

- Many client-specific interfaces are better than one general-purpose interface.
- No client should be forced to depend on methods it does not use
- Decrease coupling

- Increase cohesion

75

Page 76

76

http://www.javabrahman.com/programming-principles/interface-segregation-principle-explained-examples-java/

Page 77

Dependency inversion principle

- Depend upon Abstractions
- Do not depend upon concretions
- High-level modules should not depend on low-level module
- Decoupling software modules

https://en.wikipedia.org/wiki/Dependency_inversion_principle

Dependency Injection

http://www.codeproject.com/Articles/592372/Dependency-Injection-DI-vs-Inversion-of-Control-IO

Design Patterns

OO Design Patterns

- Design patterns are formalized best practices
- to solve **common** problems
- record **experience** in designing
- Focus -> **designing**
- "Gang of Four" (GoF) book

http://java-design-patterns.com/

OO Design Patterns - Principles

- KISS – Keep it Simple S...
- Do the simplest thing that could possibly work
- YAGNI - You aren't gonna **need** it (XP)

- Code for the **Maintainer**
- Avoid premature optimization

Page 85

OO Design Patterns - Principles

- Avoid Premature Optimization

*"Programmers **waste** enormous amounts of time thinking about, or worrying about, the speed of **noncritical parts of their programs**, and these attempts at efficiency actually have a strong negative impact when **debugging** and **maintenance** are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%."*

https://en.wikiquote.org/wiki/Donald_Knuth

Page 86

OO Design Patterns (GoF)

- Design patterns are expressed in terms of **classes** and **interfaces**
- Four essential elements
  - Pattern name
  - Problem
  - Solution
  - Consequences

Page 87

OO Design Patterns (GoF)

- GoF Catalog: **23** design patterns
- Classified by purpose:
  - Creational
  - Structural
  - Behavioral

Page 88

Creational Patterns

- **object** creation mechanisms
- trying to create objects in a manner **suitable** to the situation

- **controlling** this object creation

Page 89

Creational Patterns (5)

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

Page 90

Creational Patterns (5)

- Abstract Factory
  - **families** of product objects
- Factory Method
  - **subclass** of object that is instantiated
- Builder
  - how a **composite** object gets created
- Prototype
  - create object based on an **existing object** through cloning.
- Singleton
  - the **sole** instance of a class

Page 91

Creational Patterns (5)

- Abstract Factory

- **families** of product objects
- Factory Method
  - **subclass** of object that is instantiated
- **Builder**
  - how a **composite** object gets created
- Prototype
  - create object based on an **existing object** through cloning.
- **Singleton**
  - the **sole** instance of a class

Creational Pattern - Builder

- **Intent**:separate the **construction** of a complex object from its representation so that the same construction process can create different **representations**.

- **Applicability**
  - Building is **independent** of the parts

Page 93

Creational Pattern - Builder

Page 94

Creational Pattern - Singleton

- **Intent**:Ensure a class **only** has **one instance**, and provide a global point of access to it.
- **Applicability**:must be exactly one instance of a class, and it must be accessible to clients from a **well-known access point**

Page 95

Creational Pattern - Singleton

Page 96

Structural Patterns

- identifying a simple way to realize **relationships** between entities
- **composition** of classes

Structural Patterns (7)

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

- Proxy

Page 98

Structural Patterns (7)

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

Page 99

Structural Pattern - Composite

- **Intent**: Compose objects into tree structures to represent part-whole hierarchies.
- **Applicability**:
  - you want to represent part-whole hierarchies of objects
  - clients to be able to ignore the difference between compositions of objects and individual objects

Page 100

Structural Pattern - Composite

Structural Pattern - Decorator

- **Intent**: Attach additional **responsibilities** to an object **dynamically**. Decorators provide a flexible **alternative** to **subclassing** for extending functionality.
- **Applicability**
  - to add responsibilities to individual objects dynamically and transparently
  - for responsibilities that can be withdrawn
  - extension by subclassing is impractical

Page 102

Structural Pattern - Decorator

Page 103

Structural Pattern - Façade

- **Intent:** Provide a **unified interface** to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Applicability**
  - simple interface to a complex subsystem
  - decouple the subsystem from clients
  - layer your subsystems

Structural Pattern - Façade

Behavioral Patterns (11)

- assignment of responsibilities between objects
- encapsulating behavior in an object and delegating requests to it
- identify common communication patterns between objects

Page 106

Behavioral Patterns (11)

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Page 107

Behavioral Patterns (11)

- Chain of responsibility
- **Command**
- Interpreter
- Iterator
- Mediator
- Memento
- **Observer**
- State
- **Strategy**
- Template method
- Visitor

Page 108

Behavioral Pattern - Command

- Intent
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, **queue** or log

requests, and support **undoable** operations.

- Applicability
  - parameterize objects by an action
  - specify, queue, and execute - different times
  - support undo
- javax.swing.Action

Page 110

Behavioral Pattern - Observer

- **Intent**: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Applicability**
  - object should be able to notify other objects
  - a change to one object requires changing others
- java.util.EventListener

Page 111

Behavioral Pattern - Observer

Page 112

Behavioral Pattern - Strategy

- **Intent**: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- **Applicability**
  - need different variants of an algorithm
  - a class defines many behaviors, no conditionals

Page 113

Behavioral Pattern - Strategy

Page 115

Publish Subscribe

- **Intent**: **Broadcast** messages from sender to all the interested receivers.
- **Applicability**
  - two or more applications need to **communicate** using a **messaging** system for **broadcasts**.

Page 116

Publish Subscribe

Persistence Design Patterns

- Data Access Object (DAO)
- Data Mapper
- Active Record
- Repository

Data Access Object (DAO)

- Object provides an abstract interface to some type of database or other persistence mechanism.

Page 119

Data Mapper

- A layer of mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.
- Composite POJO + implementation class

Active Record

- Active Record is the M in MVC - the model - which is the layer of the system responsible for representing business data and logic.
- Active Record was described by Martin Fowler in his book Patterns of Enterprise Application Architecture.
- Data and persistence operations in the same class.
- Ruby on Rails

Page 122

Repository

- Repository layer is added between the domain and data mapping layers to isolate domain objects from details of the database access code and to minimize scattering and duplication of query code.
- The Repository pattern is especially useful in systems where number of domain classes is large or heavy querying is utilized.
- Usually, use Repository pattern in your applications
- Implementation example - Spring Data

Page 123

Repository

Page 124

Anti-pattern

*“**Anti-patterns** are **poor design** choices that are conjectured to make object-oriented systems **harder to maintain**.”*

Page 127

Anti-patterns (or Code Smells)

- An Anti-pattern is a **literary form** that describes a commonly occurring solution to **a problem** that generates **negative consequences**.
- Anti-patterns provide real-world **experience** in recognizing **recurring problems** in the software industry.
- Anti-patterns provide a **common vocabulary** for identifying problems and discussing solutions.
- More than 30 antipatterns were catalogued, but we will focus on the 12 most discussed by academia and it has automate detection tools.

Page 128

Study of the impact of anti-patterns

- Classes participating in antipatterns are **more change- and fault-prone** than others.
- **Extent** classes participating in antipatterns have **higher odds to change** or to be subject to **fault-fixing** than other classes.
- **Size alone** cannot explain the higher odds of classes with antipatterns to underwent a (fault-fixing) change than other.
- Structural changes **affect more** classes with antipatterns than others.
- We now have **empirical evidence** that code containing code smells or participating in anti-patterns is **significantly more change-prone** than “clean” code.

Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc and Guiliano Antoniol, An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness, Journal of Empirical Software Engineering (EMSE), 2011.

Page 129

Anti-patterns/code smells - research findings

- **Newcomers** are **NOT necessary responsible** for introducing bad smalls, while developers with **high workloads and release pressure** are more prone to introducing smell instances.
- Findings showed that most of the smell instances are introduced **when an artifact is created and not as a result of its evolution**.
- **80% of smells survive in the system**. Code Smells **have a high**

survivability and are **rarely removed** as a direct consequence of **refactoring** activities.
- Among the 20% of removed instances, only 9% is removed as a direct consequence of refactoring operations.

M. Tufano; F. Palomba; G. Bavota; R. Oliveto; M. Di Penta; A. De Lucia; D. Poshyvanyk, "When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)," in IEEE Transactions on Software Engineering , vol.PP, no.99, pp.1-1

Page 130

Anti-patterns - research findings

- All these findings suggest that code smells and anti-pattern **need to be carefully detected and monitored** and, whenever necessary, **refactoring** operations should be **planned and performed** to deal with them.
- The **identification** and the **correction** of anti-patterns/code smells in large and non-trivial software systems can be **very challenging**.
- Anti-patterns can be **defined** in terms of thresholds imposed on **metric values** (Moha et al., 2008).

Fabio Palomba, Andrea De Lucia, Gabriele Bavota, Rocco Oliveto: Anti-Pattern Detection: Methods, Challenges, and Open Issues. Advances in Computers 95: 201-238 (2015)

Page 131

Antipatterns

- More than 30 antipatterns were catalogued by Fowler and Brown, but we will focus on the 12 most discussed by academia, providing **automate detection tools**.
  - Blob or God Class, Feature Envy
  - Duplicate Code, Refused Bequest
  - Divergent Change, Shotgun Surgery
  - Parallel Inheritance, Functional Decomposition
  - Spaghetti Code, Swiss Army Knife
  - Type Checking, Poltergeists

Page 132

Blob or God Class

- A class having **huge size** and implementing different **responsibilities**.
- Blob is a class that **centralizes** most of the system's behavior
- Presence of a **high number of attributes and methods**, which implement **different functionalities**, and by many **dependencies** with **data** classes.
- "Selfish behavior" because it works for itself.
- Detection strategies
  - LCOM (Lack of Cohesion Of Methods) higher than 20
  - Number of methods and attributes higher than 20
  - It has an one-to-many association with data classes
  - Name that contains a suffix in the set {*Process, Control, Command, Manage, Drive, System*}

Page 133

Feature Envy

- A method making **too many calls** to methods of **another class** to obtain data and/or functionality.
- It is often characterized by a **large number of dependencies** with the envied class.
- **Negatively** influences the **cohesion and the coupling** of the class in which

the method is implemented.

- Detection strategies
    - traverse the Abstract Syntax Tree (AST) of a software system in order to identify, for each field, the set of the referencing classes.
    - define a threshold to discriminate the fields having too many references with other classes.

Duplicate Code

- Classes that show the **same code structure** in more than one place.
- Code duplication is a potentially serious problem that **affects** the **maintainability** of a software system, but also its **comprehensibility**.
- The problem of the **identification** of code duplication is **very challenging**, because, during the evolution, **different copies** of a feature suffer different changes and this affect the possibility of the identification of the common functionality provided by different copied features.
- Detection strategies
    - detection of syntactic or structural similarity of source code
    - to find matches of sub-trees (using AST)

Page 135

Refused Bequest

- A class **inheriting** functionalities that it **never uses**.
- A subclass **overrides** a **lot of methods** inherited by its parent.
- Relationship of **inheritance is wrong**, namely the subclass is not a specialization of the parent.
- Methods are **never called** by the clients of the subclass
- Detection strategies
  - the class overrides more than x% of the methods defined by the parent.
  - combination of static source code analysis and dynamic unit test execution.
  - intentionally override these methods introducing an error in the new implementation (e.g., division by zero) in the superclass. If the error never comes, the subclass is a Refused Bequest antipattern.
  - Hard to detect automatically.

Page 136

Divergent Change

- A class commonly **changed in different ways** for different reasons.
- Classes affected by this anti-pattern generally have **low cohesion**.
- Detection strategies
    - using coupling information it is possible to build a Design Change Propagation Probability (DCPP) matrix. The DCPP is an *n x n* matrix where the generic entry *Aij* is the probability that a design change on the artifact *i* requires a change also to the artifact *j*.
    - using the historical information that a system can have (i.e., change log)
    - detect a subsets of methods in the same class that often change together

Page 137

Shotgun Surgery

- A class where a change implies **cascading changes** in several related classes.
- Every time you make a kind of change, you have to make a **lot of little changes** to a lot of different classes.

- Eg: Changes in UI (View) -> Controller -> Model -> Persistence
- Detection strategies
  - using coupling information it is possible to build a Design Change Propagation Probability (DCPP) matrix. The DCPP is an *n x n* matrix where the generic entry *Aij* is the probability that a design change on the artifact *i* requires a change also to the artifact *j*.
  - using the historical information that a system can have (i.e., change log)
  - detect a subsets of methods in the same class that often change togethe

Page 138

Parallel Inheritance

- Pair of classes where the **addition of a subclass** in a hierarchy implies the addition of a **subclass in another hierarchy**.
- Every time you make a subclass of one class, you also have to make a subclass of another.
- A special case of Shotgun Surgery
- Detection strategy
  - relies on historical information

Page 139

Functional Decomposition

- A class implemented following a **procedural-style**.
- A class in which inheritance and polymorphism are poorly used, declaring **many private fields and implementing few methods**.
- Procedural-style programming -> a **main routine** that **calls** many **subroutines**.
- Detection strategies
  - many dependencies with classes composed by a very few number of methods addressing a single function

Page 140

Spaghetti Code

- A class **without structure** that declare **long methods** without parameters.
- Characterized by **complex methods**, with no parameters, interacting between them using instance variables.
- Procedural-style programming. It usually have a **procedural name**.
- Detection strategies
  - looking for classes having at least one long method
  - namely a method composed by a large number of LOC and declaring no parameters
  - class does not present characteristics of Object Oriented design
  - the class does not use the concept of inheritance and should use many global variables

Swiss Army Knife

- A class that exhibits **high complexity** and offers a **large number** of different **services**.
- Swiss Army Knife is a class that provides **answer** to a **large** range of **needs**.
- Swiss Army Knife is a class that provides **services to other classes**,

exposing high complexity, while a **Blob** is a class that **monopolizes** processing and data of the system.

- The anti-pattern arises when a class has **many methods with high complexity** and the class has a **high number of interfaces**.
- Detection strategies
  - complexity metrics
  - semantic checks in order to identify the different services provided by the class

Page 142

Type Checking

- A class that shows **complicated conditional statements**.
- This problem manifests itself especially when a developer uses **conditional statements to dynamically dispatch** the behavior of the system instead of polymorphism.
- Lack of knowledge on how to use OO mechanisms such as **polymorphism**.
- Detection strategies
  - see long and intricate conditional statements
  - a conditional statement involves RunTime Type Identification (RTTI)

Poltergeists

- Poltergeists are classes with **limited responsibilities** and roles to play in the system.
- Poltergeists create **unnecessary abstractions**; they are excessively complex, hard to understand, and hard to maintain.
- Attention on Controllers: it can be poltergeists.
- Detection strategies
  - several redundant navigation paths
  - stateless classes
  - temporary, short−duration objects and classes

Linguistic Anti-Patterns

Linguistic Antipatterns (LAs) in software systems are ***recurring poor practices in the naming****, documentation, and* ***choice of identifiers*** *in the implementation of an entity, thus possibly impairing program understanding.*

V. Arnaoudova, M. Di Penta, G. Antoniol and Y. G. Guéhéneuc, "A New Family of Software Anti-patterns: Linguistic Anti-patterns," *2013 17th European Conference on Software Maintenance and Reengineering*, Genova, 2013, pp. 187-196.

Page 145

“Get” - more than an accessor

In Java, accessor methods, also called getters, provide a way to access class attributes. As such, **it is not common that getters perform actions other than returning the corresponding attribute**. Any other action should be documented, possibly naming the method differently than “getSomething”.

Page 146

“Get” - more than an accessor

Example of “Get” - more than an accessor (Eclipse-1.0).

Page 147

“Is” returns more than a Boolean

When a method name starts with the term “is” one would expect a Boolean as a return type, thus having two possible values for the predicate, i.e., “true” and “false”. Thus, having an “is” method that does not return a Boolean, but returns more information is counterintuitive.

Example of “Is” returns more than a Boolean (Cocoon 2.2.0).

Page 148

“Set” method returns

A set method having a return type different than void should document the return type/values with an appropriate comment or should be named differently to avoid any confusion.

Example of “Set” method returns (ArgoUML-0.10.1).

Page 149

Expecting but not getting a single instance

When a method name indicates that a single object (and not a collection) is returned, this shall be consistent with its return type. If, instead, the return type is a collection, the method shall be renamed or appropriate documentation is needed.

Example of Expecting but not getting a single instance (Eclipse-1.0)

Page 150

Linguistic anti-patterns (cont.)

- **"Get" method does not return**
    - protected **void get**MethodBodies(CompilationUnitDeclaration unit ...
- **Not answered question**: the method name is in the form of predicate whereas the return type is not Boolean.
    - public **void is**Valid (...
- **Transform method does not return**: the method name suggests the transformation of an object, but there is no return value.
    - public **void** java**To**Native (...

- **Expecting but not getting a collection**: the method name suggests that a collection should be returned, however a single object, or nothing, is returned.
  - public **boolean** getStat**s** ( ) { return _stats ; }

Linguistic anti-patterns (cont.)

- **Says one but contains many:** an attribute name suggests a single instance, while its type suggests that the attribute stores a collection of objects.
  - **Vector** _target;
- **Name suggests Boolean but type does not**
  - **int**[ ] **is**Reached;
- **Says more than it contains**
  - private static **boolean** _stat**s** = **true**;

Page 152

Antipattern detection tools

- God Class
  - http://pmd.sourceforge.net
- Duplication code
  - http://pmd.sourceforge.net/pmd-4.3.0/cpd.html
- SmellDetector (Ptidej - several smells)
  - git clone https://github.com/ptidej/SmellDetectionCaller.git
- Linguistic Anti-pattern Detector (LAPD)
  - http://www.veneraarnaoudova.ca/linguistic-anti-pattern-detector-lapd/
- JDeodorant (several smells)
  - https://github.com/tsantalis/JDeodorant

Page 153

Are anti-patterns really harmful?

Despite the existing evidence about the negative effects of anti-patterns, it is still **unclear** whether developers would actually consider **all anti-patterns as actual**

symptoms of wrong design/implementation choices, or whether some of them are simply a **manifestation of the intrinsic complexity** of the designed solution.

Empirical studies indicated that (i) developers perceived different instances of Blob as **not particularly dangerous** for the system maintainability, especially because they change these classes sporadically; and (ii) some developers, in particular junior programmers, work better on a version of a system having some classes that **centralized the control**, i.e., Blob.

Presence of anti-patterns in source code is sometimes **tolerable**, and part of developers' design choices.

Page 154

Refactoring

- Refactoring (noun): a change made to the internal structure of software to make it
- easier to understand and cheaper to modify **without changing its observable**

behavior.

Page 155

Why Should You Refactor?

- Refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster

Page 156

When should we refactor?

- Refactor When You Add Function
- Refactor When You Need to Fix a Bug
- Refactor As You Do a Code Review

Page 157

Refactoring process

- Identify an opportunity

- Apply **Test-driven development** (TDD)
- Have you a unit test?
  - If not, **create an unit test** to test the code
- Perform an improvement
  - Patterns, Antipatterns, code smells
- Run your test
  - Green - okay -- Red – improve your solution
- Evaluate the result
  - Write tests and improvements just to achieve the goal

Page 158

http://www.gargoylesoftware.com/practices/tdd

Page 160

Refactoring to patterns

- A list of identified opportunities to refactoring
- In general use design patterns
- Catalog of smells -> refactoring

Page 161

Refactoring to patterns (examples)

- Conditional Complexity
  - Replace Conditional Calculations with Strategy
- Duplicated Code
  - Introduce Polymorphic Creation with Factory Method
- Replace Constructors with Creation Methods

Page 162

Replace Constructors with Creation Methods