Patterns

Jeszenszky, Péter
University of Debrecen, Faculty of Informatics
jeszenszky.peter@inf.unideb.hu

Kocsis, Gergely (English version)
University of Debrecen, Faculty of Informatics
kocsis.gergely@inf.unideb.hu

Last modified: 06.04.2017

Based on the material by Kollár Lajos

Patterns

- The concept is from Christopher Alexander (1936–) from the field of building architecture.
- Some software engineers also started to use the concepts
- Become widely known in SE (Software Engineering) by the book written by the „Gang of Four” (GoF):
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides

What is a pattern? (1)

- „**Each pattern describes a problem** which occurs over and over again in our environment, **and then describes the core solution to that problem**, in such a way that you can use the solution a million times over, without ever doing it the same way twice”
  - Christopher Alexander. A Pattern Language: Towns, Buildings, Construction. Oxford University Press. 1977.

What is a pattern? (2)

- Christopher Alexander:

“Each pattern is a three-part rule, which expresses a relation between a certain **context**, a **problem**, and a **solution”**

  - The Timeless Way of Building. Oxford University Press. 1979.

What is a pattern? (3)

- Martin Fowler:

  „**A pattern is an idea that has been useful in one practical context** and will probably be useful in others.”

  - Analysis Patterns: Reusable Object Models. Addison-Wesley, 1996.

What is a pattern? (4)

- Scott W. Ambler:

  A pattern is a description of a general solution to a common problem or issue from which a detailed solution to a specific problem may be determined.

  - http://www.ambysoft.com/books/processPatterns.html

Parts of patterns

- ***Context***:
  - In which situations the problem occurs
- ***Problem***:
  - A problem that emerges repeatedly in a given context.
  - ***Force***: A point of view that has to be counted in during the solution of the problem.
    - E.g. constraints and desired properties of the solution
    - Forces analyse the problem from different points of view. They may complete each other or may contradict each other
      - Example of contradicting forces: scalability of the system and minimizing the code length
- ***Solution***:
  - How to solve a repeating problem? How to offset the appearing forces?
  - A solution provides only a scheme and not a detailed plan.
  - It is a “Mental building block”

Pattern Catalogues and Pattern Languages (1)

- ***Pattern Catalogue*/*Pattern Collection***
  - A group of patterns
  - A collection can be heterogeneous or it can focus on a specific field, problem or abstraction level.
  - It may be structured or not structured.
  - The description of the patterns are usually independent.
  - Example: the GoF catalogue

Pattern Catalogues and Pattern Languages (2)

- ***Pattern Language***: A pattern collection of related patterns that define a systematic process for the solution of software engineering problems
  - E.g.: user interface design patterns

Pattern Catalogues and Pattern Languages (3)

- The description of patterns is always done using a given template (pattern template).
  - The different catalogues and languages may use different templates in practice.

Pattern types

„Patterns can exist at all scales.” — C. Alexander

- **Architectural patterns/styles**
- **Design patterns**
- **Programming idioms/Implementation patterns**
- **Process patterns**
- **Analysis patterns**
- **Testing patterns**
- **Process patterns**
- **User interface desidn patterns**
- **Antipatterns**

Architectural patterns (1)

Architectural patterns/styles

- They contain best practices and pre-defined subsystems for the basic structure of a software structure.
- E.g.: MVC architecture

Architectural patterns (2)

- Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
- Frank Buschmann et al. *Pattern-Oriented Software Architecture Vol. 1–5*. Wiley, 1996, 2000, 2004, 2007.

Example of an architectural pattern: MVC (1)

Name: Model–View–Contrroller (MVC)

Context: interactive applications with a flexible human-machine interface

Problem: frequent change requests for user interfaces

- The same information is to be displayed differently (e.g., bar chart, pie chart)
- Presentation and behavior of the application should immediately reflect data manipulation
- User Interfaces that can be changed dinamically (even in run-time) are needed
- Support of various look-and-feel standards are needed
- Porting an application should not affect the core functionalities

Solution:

- model encapsulates data and functionality and is independent from the input behavior and the representation of the output;
- view displays information;
- controller receives and transforms the input to service requests (towards model and view).

Example of an architectural pattern: MVC (2)

- The separation of the model from the view makes it possible to to have more views for the same model
  - The same data can be presented in different ways
- The separation of the view from the control component is less important
  - This makes possible the use of more controllers for the same view
    - Classical problem: an editable and a non-editable version of the same view by the use of two different controllers
  - In practice there is often only one control/view

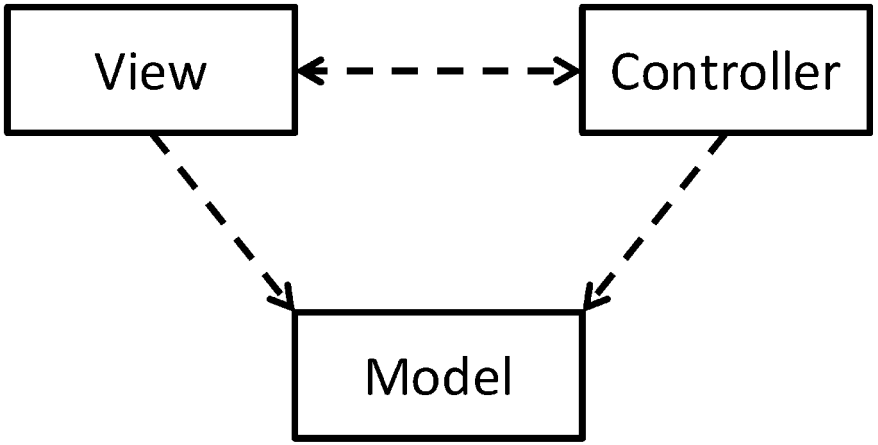
Example of an architectural pattern: MVC (3)

- The model is an object representing some information of the domain that encapsulates data
  - It has application-specific processing procedures that are called by the controllers in the name of the user
  - Provides functions to enable the access of data that are used by views
  - Registers the dependent objects (views and controllers) and inform them about changes in the data

Example of an architectural pattern: MVC (4)

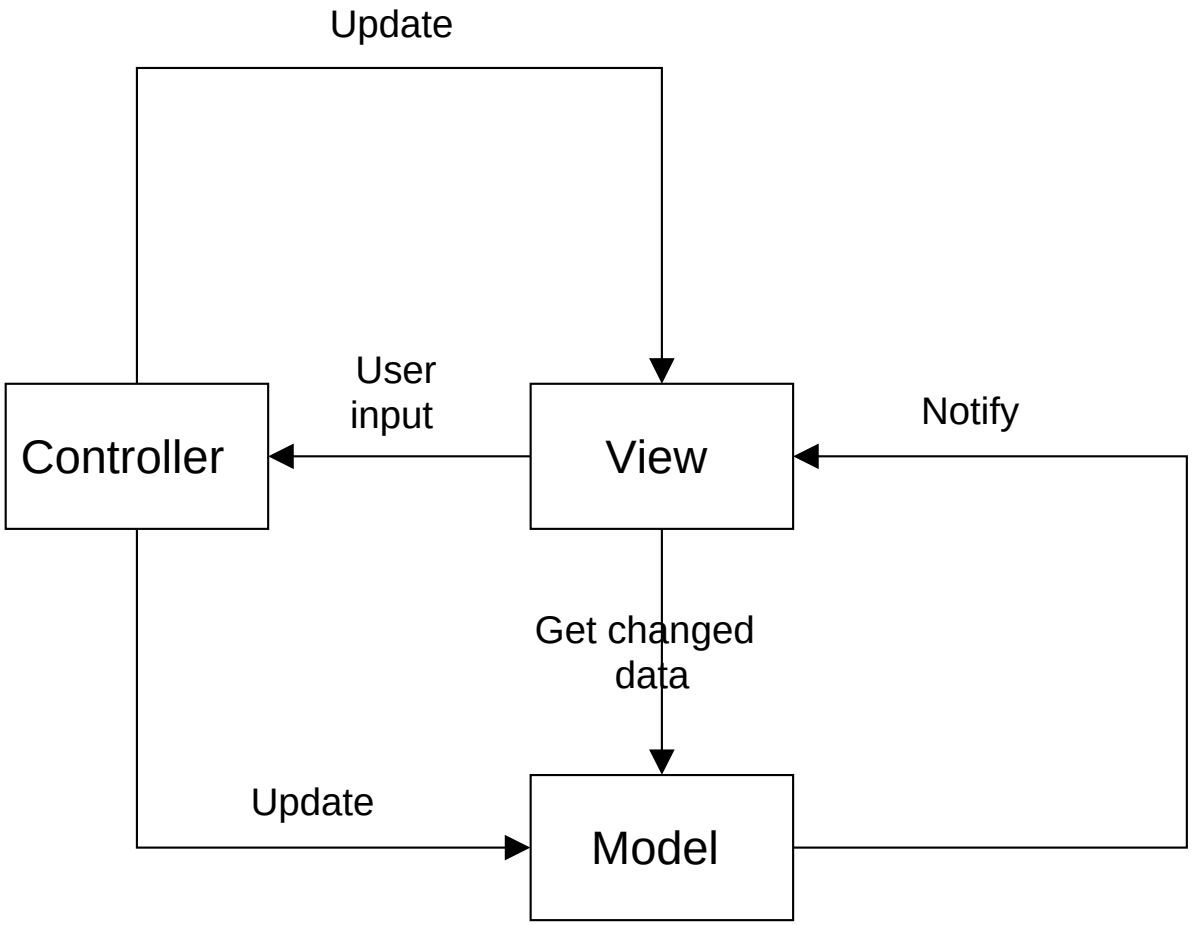
View represents the display of the model on a user interface.

Controller receives user inputs, handles the model and asks the view to refresh accordingly.



Model is a nonvisual object representing some information of the problem domain.

Example of an architectural pattern: MVC (5)



Architectural styles (1)

An architectural style defines a family of software systems in terms of their structural organization. An architectural style expresses components and the relationships between them, with the constraints of their application, and the associated composition and design rules for their construction. [POSA1]

- E.g.: client-server, layered, REST – *Representational State Transfer*, …

Architectural styles (2)

- Architectural styles are very similar to architectural patterns, however they differ at many points
  - Patterns give solutions for given repeatedly appearing problems from the actual viewpoint of the context
  - Styles provide design methods independent of the actual context

Design patterns (1)

- Design patterns are middle-scale patterns. They are less comprehensive than architectural patterns
- The use of them does not affect the overall structure off the system, but they highly affect the structure of the subsystem
- They are independent of programming languages and paradigms

Design patterns (2)

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
  - GoF – *Gang of Four*

Design patterns (3)

- GoF:
  - Design patterns are descriptions of cooperating objects that solve a given design problem in a given context in a customized form.

Design patterns (4)

- Language specific further reading:
  - **C#**:
    - Steven John Metsker. *Design Patterns in C#*. Addison-Wesley, 2004.
    - *.NET Design Patterns in C#* http://www.dofactory.com/net/design-patterns
  - **C++**:
    - Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001.
    - Alexander Shvets. *Design Patterns Explained Simply*. 2014. https://sourcemaking.com/design-patterns-ebook
  - **Java**:
    - Alexander Shvets. *Design Patterns Explained Simply*. 2014. https://sourcemaking.com/design-patterns-ebook
    - Ilkka Seppälä. *Design patterns implemented in Java*. http://java-design-patterns.com/ https://github.com/iluwatar/java-design-patterns
  - **Python**:
    - Bruce Eckel et al. *Python 3 Patterns, Recipes and Idioms*. http://python-3-patterns-idioms-test.readthedocs.io/

Design patterns (5)

- Design patterns grouped by their goals (GoF):
  - ***Creational Patterns***
  - ***Structural Patterns***
  - ***Behavioral patterns***

Design patterns (6)

- GoF defines 23 design patterns but since the publicattion of the book of them many new patterns have born
  - E.g.: a*bstract document*, m*onad*, r*epositor*, m*ultition*, t*win*, ...
- New category: ***Concurrency Patterns***
  - E.g.: a*ctive object*, *guarded suspension*, t*hread pool*, …

Design patterns (7)

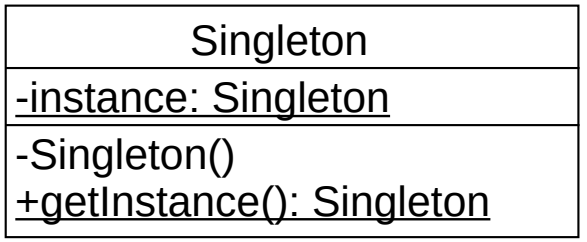
- Pattern template (GoF):
  - ***Name and Classification***:
  - ***Intent***:
  - ***Also Known As***:
  - ***Motivation***:
  - ***Applicability***:
  - ***Structure***:
  - ***Participants***:
  - ***Collaborations***:
  - ***Consequences***:
  - ***Implementation***:
  - ***Sample Code***:
  - ***Known Uses***:
  - ***Related Patterns***:

Design pattern sample: Singleton (1)

- **Intent**: Let only one instance of a class to be exist and provide a global access point for it.
- **Motivation**: In some cases it is important to have at most one instance of them
- **Applicability**: The patterns can be used in the following cases:
  - We need exactly on e instance of the class and that instance has to be reachable for the clients through well-known access points
  - This one instance has to be extensible by subclasses and the clients have to be able to use the extended instance without modification of their code

Design pattern sample: Singleton (2)

- **Structure**:



Design pattern sample: Singleton (3)

- Java implementation (1): greedy initilaization, thread safe

```
public class Singleton {

    private static Singleton instance = new Singleton();

    private Singleton() {
    } // private constructor!

    public static Singleton getInstance() {
        return instance;
    }

}
```

Design pattern sample: Singleton (4)

- Java implementation (2): lazy initialization, thread safe

```
public class Singleton {

    private static Singleton instance;

    private Singleton() {
    } // private constructor!

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

}
```

Design pattern sample: Singleton (5)

- Java implementation (3): lazy initialization, thread safe (without explicit synchronization)

```
public class Singleton {

    private static class Holder {
        private static Singleton instance = new Singleton();
    }

    private Singleton() {
    } // private constructor!

    public static Singleton getInstance() {
        return Holder.instance;
    }

}
```

Design pattern sample: Singleton (6)

- Java implementation (4): enum (J2SE 5.0–)

```
public enum Singleton {

    INSTANCE;

}
```

Design pattern sample: Singleton (7)

- Java implementation (5): lazy initialization, non thread safe (*double-checked locking*)

```
public class Singleton {

    private static Singleton instance;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

Design pattern sample: Singleton (8)

- Java implementation (6):
  - About thread safety see:
    - Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006. http://jcip.net/
      - A *The Java Memory Model* – Chapter 16. Section 16.2

Programming idioms, implementation patterns (1)

- An idiom is a low-level programming language specific pattern.
- It describes how to implement components and some spects of the relations between them with the tools of the given language.
- They include existing programming experience.

Programming idioms, implementation patterns (2)

- Kent Beck. *Implementation Patterns*. Addison-Wesley, 2008.
- Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

Programming idioms, implementation patterns (3)

- Programming language specific further reading:
  - **C#**:
    - Bill Wagner. *Effective C#: 50 Specific Ways to Improve Your C#*. Third Edition. Addison-Wesley, 2016.
    - Bill Wagner. *More Effective C#: 50 Specific Ways to Improve Your C#*. Addison-Wesley, 2008.
  - **C++**:
    - Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Third Edition. Addison-Wesley, 2008.
    - Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, 2014.
  - **Java**:
    - Joshua Bloch. *Effective Java*. Second Edition. Addison-Wesley, 2008.
  - **Python**:
    - Brett Slatkin. *Effective Python: 59 Specific Ways to Write Better Python*. Addison-Wesley, 2015. http://www.effectivepython.com/
    - Luciano Ramalho. *Fluent Python*. O'Reilly Media, 2015. https://github.com/fluentpython

Programming idioms, implementation patterns (4)

Example: Whenever overriding equals, hashCode should also be overridden!:

```
public final class Pet implements Cloneable {
    private String name;
    private int age;

    public Pet(String name, int age) {
        if (age < 0)
            throw new IllegalArgumentException();
        this.name = name;
        this.age = age;
    }

    @Override
    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (! (o instanceof Pet))
            return false;
        Pet that = (Pet) o;
        return (name == that.name || (name != null && name.equals(that.name)))
             && age == that.age;
    }
    // Error: there are no hashCode() and clone() methods!
}
```

Programming idioms, implementation patterns (5)

- Example (2):

```
Set<Pet> s = new HashSet<Pet>();
Pet p = new Pet("Tardar Sauce", 4);
s.add(p);

System.out.println(s.contains(p));

System.out.println(s.contains(new Pet("Tardar Sauce", 4)));

System.out.println(s.contains(p.clone()));
```

Output:

true
false
false

Programming idioms, implementation patterns (6)

- Example (3):

```
public final class Pet implements Cloneable {

    // ...

    @Override
    public int hashCode() {
        int result = 17;
        result = 31 * result + age;
        result = 31 * result + (name == null ? 0 : name.hashCode());
        return result;
    }

    @Override
    public Pet clone() {
        try {
            return (Pet) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }

}
```

Process patterns (1)

- Scott W. Ambler. *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press, 1998.
- Scott W. Ambler. *More Process Patterns: Delivering Large-Scale Systems Using Object Technology*. Cambridge University, 1999.

Process patterns (2)

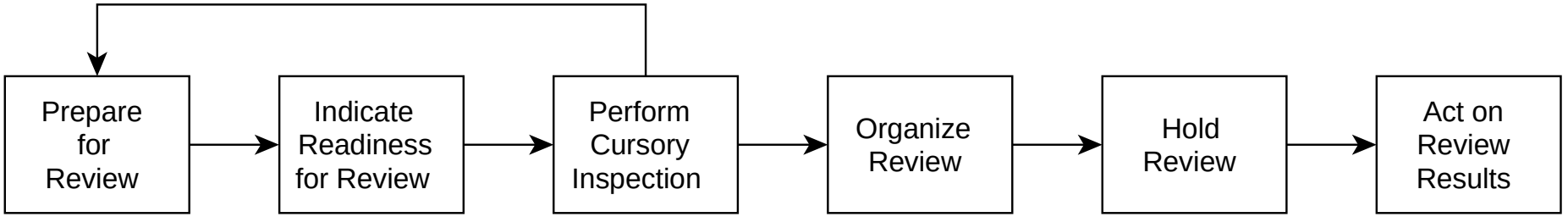
- **Process**: A sequence of activities that provides one or more outputs from one or more inputs
- **Process pattern**: A collection of general methods, operations and/or tasks that are used for object-oriented software development
  - By applying them in a structured way the process of an organization can be defined
    - Since they do not describe how exactly the task is to be done, they are reusable building blocks customizable for the needs of the given organization
  - Form of them: task, stage and phase process pattern

Process patterns (3)

- ***Task Process Pattern***: Describes the needed steps of doing a task in details
  - Example: **Technical review**
- ***Stage Process Pattern***: A higher level process pattern that usually builds up from more task process patterns. Describes those steps that are to be done interactively in a project stage
  - Example: **Program**
- ***Phase Process Pattern***: Describes interactions between stage process patterns of a project phase. The execution of phase process pattern is sequential.
  - Example: **Creation**

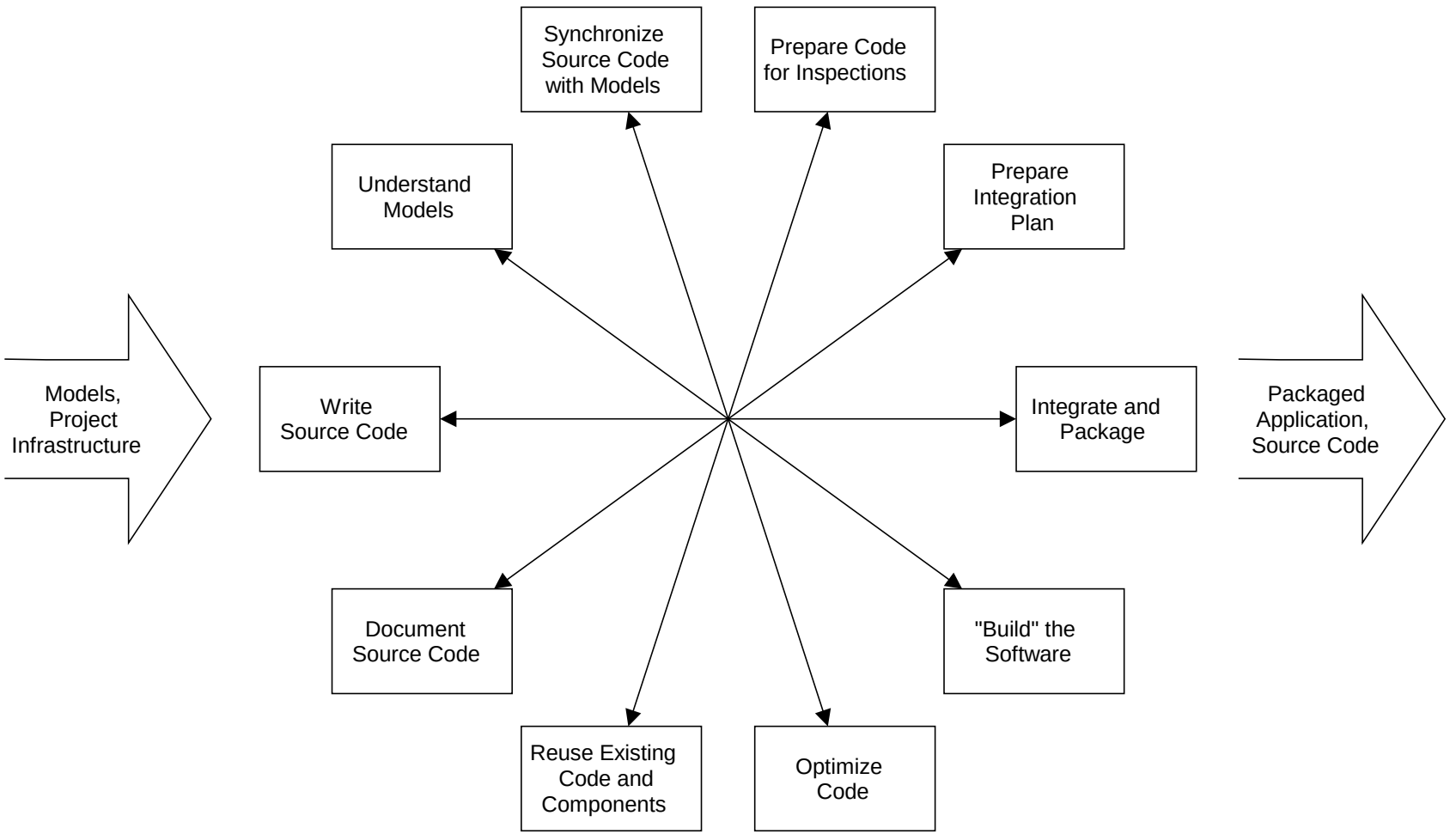
Process patterns (4)

- Example of task process pattern: ***Technical Review***



Process patterns (5)

- Example of stage process pattern: ***Program***



Analysis patterns (1)

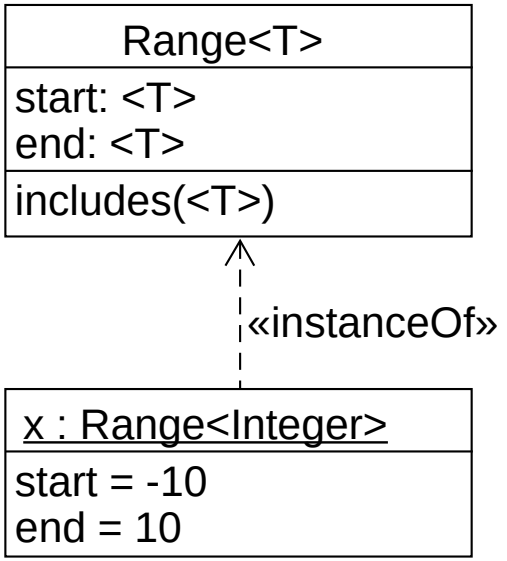
- They follow the conceptual structure of business processes
- They are groups of concepts representing often used constructions in business modeling
- What are they used for?
  - They offer design patterns and solutions for frequent problems in order to make it easier to a design model from the analysis model.
  - They help to get abstract analysis models as soon as possible. These models describe the most important specifications of the exyct problem.

Analysis patterns (2)

- Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1996.
- *tagged by: analysis patterns* https://martinfowler.com/tags/analysis%20patterns.html

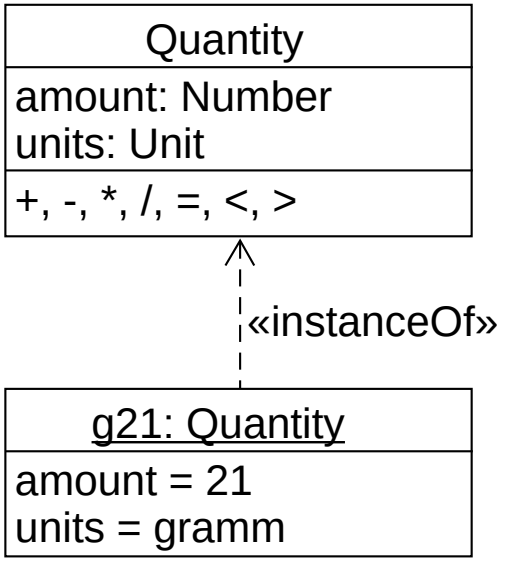
Analysis patterns (3)

- Example of analysis patterns: domain https://martinfowler.com/eaaDev/Range.html
  - Describes a value domain.



Analysis patterns (4)

- Example of analysis patterns: quantity https://www.martinfowler.com/eaaDev/quantity.html
  - Represents a numeric value with a quantity unit



Testing patterns (1)

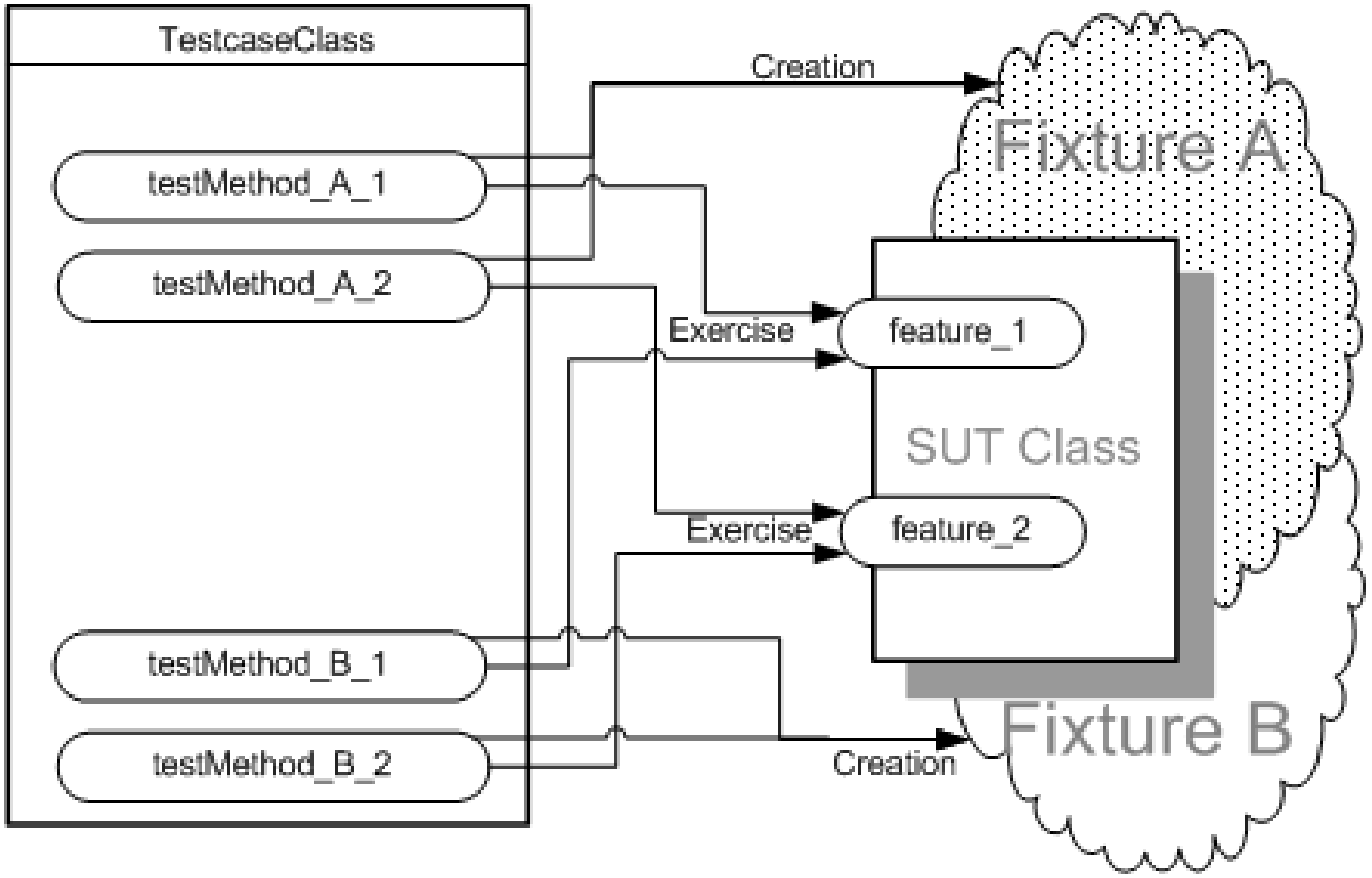
- Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007. http://xunitpatterns.com/

Testing patterns (2)

- ***Testcase Class per Class*** (page 617.):
  - How to organize our test methods to testcase classes?
    - Put all the test methods testing a given class to one testcase class
  - When to use?
    - If there are not so many test methods or we have just started to add the test cases to the test system.
    - As the number of tests increases and the test fixtures requirements are getting more clear the class can be cut to more classes. (see: *Testcase Class per Fixture*, *Testcase Class per Feature*).

Testing patterns (3)

- Source: *Testcase Class per Class* http://xunitpatterns.com/Testcase%20Class%20per%20Class.html



User interface design patterns (1)

- Reusable solutions for often appearing UI design problems
  - Applied for desktop, web, and mobile interfaces

User interface design patterns (2)

- Jenifer Tidwell. *Designing Interfaces: Patterns for Effective Interaction Design*. Second Edition. O'Reilly Media, 2010. http://designinginterfaces.com/
- Christian Crumlish, Erin Malone. *Designing Social Interfaces: Principles, Patterns, and Practices for Improving the User Experience*. Second Edition. O'Reilly Media, 2015. http://www.designingsocialinterfaces.com/

User interface design patterns (3)

- Further design pattern catalogues:
  - Martijn van Welie. *Interaction Design Pattern Library.* http://www.welie.com/patterns/
  - Anders Toxboe. *UI-Patterns.com*. http://ui-patterns.com/
  - *The Endeca User Interface Design Pattern Library* http://www.oracle.com/webfolder/ux/applications/uxd/endeca/content/library/en/home.html
  - *USPTO UI Design Library* https://uspto.github.io/designpatterns/
  - …

User interface design patterns (4)

- Pattern template (Tidwell):
  - **What**:
  - **Use when**:
  - **Why**:
  - **How**:
  - **Examples**:

User interface design patterns (5)

- **Mély háttér** (Tidwell, 499. oldal):
  - **What**: Place an image or gradient into the page's background that visually recedes behind the foreground elements.
  - **Use when**: Your page layout has strong visual elements (such as text blocks, groups of controls, or windows), and it isn't very dense or busy. You want the page to look distinctive and attractive; you may have a visual branding strategy in mind. You'd like to use something more interesting than flat white or gray for the page background
  - **Why**: Backgrounds that have soft focus, color gradients, and other distance cues appear to recede behind the more sharply defined content in front of them. The content thus seems to "float" in front of the background. This pseudo-3D look results in a strong figure/ground effect—it attracts the viewer's eye to the content. Fancy explanations aside, it just looks good.
  - **How**: Use a background that has one or more of these characteristics: Soft focus, Color gradients, Depth cues, No strong focal points
  - **Examples**: https://www.mozilla.org/hu/firefox/new/

Antipatterns (1)

- General solutions of a given problem that provide expressly negative consequences
- They can appear at any level

Antipatterns (2)

- William H. Brown, Raphael C. Malveau, Hays W. McCormick III, Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.
- https://sourcemaking.com/antipatterns
- Phillip A. Laplante, Colin J. Neill, Joanna F. DeFranco. *AntiPatterns: Managing Software Organizations and People*. 2nd Edition. Auerbach, 2011.

Antipatterns (3)

- The three categories based on the viewpoint:
  - Software development antipatterns
  - Software architectural antipatterns
  - Software project management antipatterns

Software development antipatterns (1)

- ***The Blob***: The procedural design leads to an object that holds most of the responsibilities while other objects only store data or execute simple processes
- ***Continuous Obsolescence***: Technology is ever changing and it is hard for the developers to stay up to date with the current versions and find the proper combinations of software versions that can cooperate.
- ***Lava Flow***: Dead code and forgotten design informations freeze into the dynamic plan

Software development antipatterns (2)

- ***Functional Decomposition***: Experienced programmers of procedural languages develop object oriented applications. The resulting code looks similar to sources of procedural languages (FORTRAN, C, PASCAL). It can be very complex since these exxperienced programmers usually find really clever ways of the replication of the old tools in OO architecture.
- ***Poltergeist***: Classes with too restricted roles. They frequently start processes for other objects.
- ***Boat Anchor***: A part of the program or the hardware that has no useful role in the project.

Software development antipatterns (3)

- ***Golden Hammer***: The use of a well-known technology or concept for all problems
- ***Dead End***: The modification of a reusable component if the component is not maintained and supported later by the provider of the technology
- ***Spaghetti Code***: *ad hoc* program structure makes it harder to extend or optimize the code

Software development antipatterns (4)

- ***Input Kludge***: *Ad hoc* algorithms to manage the input of the program
- ***Walking through a Minefield***: Use of the newest technologies is dangerous since they may be full of errors
- ***Cut–and–Paste Programming***: Reuse of code through copy-paste implies serious maintenance problems
- ***Mushroom Management***: The isolation of developers. As a result they have only second-hand information about the specifications (through engineers, managers, analysts.

Software architectural antipatterns (1)

- ***Autogenerated Stovepipe***: This pattern appears at the time of migrating an existing software system on a shared infrastructure. The problem is when the interfaces are copied without redesign.
- ***Stovepipe Enterprise***: Separately designed systems reduce the interoperability and reuse and also may increase costs
- ***Stovepipe System***: Integration of subsystems is done in an ad hoc way using more integration strategies and mechanisms. The same as stovepipe enterprise for one system.

Software architectural antipatterns (2)

- ***Cover Your Assets***: In case of document-driven software processes the authors often pose useless specifications and requirements to avoid responsibilities.
- ***Vendor Lock-In***: Depending on an architecture of a given producer too much
- ***Wolf Ticket***: A product that poses its free property without any forced activities

Software architectural antipatterns (3)

- **Implicit architecture**: There is no architectural plan since the developers pose that based on their previous experience they can solve the problem without it
- ***Design by Committee***: A plen by a committee is often too complex and has no coherent structure
- ***Swiss Army Knife***: A too complex class with which the designer tries to solve all the possible uses. This is hard to understand by other programmers.
- ***Reinvent the Wheel***: The lack of reuse of technologies in different software projects

Software development antipatterns (4)

- ***Input Kludge***: *Ad hoc* algorithms to manage the input of the program
- ***Walking through a Minefield***: Use of the newest technologies is dangerous since they may be full of errors
- ***Cut–and–Paste Programming***: Reuse of code through copy-paste implies serious maintenance problems
- ***Mushroom Management***: The isolation of developers. As a result they have only second-hand information about the specifications (through engineers, managers, analysts.

Antipattern template (1)

- ***AntiPattern Name***:
  - The name of the antipattern
- ***Also Known As***:
  - Other known names
- ***Most Frequent Scale***:
  - On which level of the software development process this antipattern appears. The following keywords may be used here: idiom, micro-architecture, framework, application, system, entzerprise, global/industry
- ***Refactored Solution Name***:
  - Identifies the refactored solution name
- ***Refactored Solution Type:***
  - Marks the type of activity needed as the solution of the antipattern. The following keywords may be used here: software, technology, process, role

Antipattern template (2)

- ***Root Causes***:
  - Shows the root causes of the antipattern. Keywords may be here: Sloth (lazyness), Apathy (lack of the mod for being creative), Pride (not being open to learn the right solution), Haste (trying to solve sg. too fast), Avarice (not willing to spend the required resources), Ignorance (not knowing the right solution)
- ***Unbalanced Forces***:
  - Keywords specifying those factors that were not counted in or were user too many times in the pattern. Possible choices may be: managing functionality, managing performance, managing complexity, handling changes, managing IT resources, managing technology transfer
- ***Anecdotal Evidence***:
  - Optional part describing known anecdotes related to the antipattern
- ***Background***:
  - Optional. It contains more example places where the antipattern appears and more interesting general information
- ***General Form***:
  - A general description of the antipattern. Often contains figures. It is not an example but a general version.

Antipattern template (3)

- ***Symptoms and Consequences***:
  - The symptoms and the consequences of the antipattern
- ***Typical Causes***:
  - A list of the exact tipical casuses of the problem.
- ***Known Exceptions***:
  - Describes those exceptional cases when the antipattern is not harmful
- ***Refactored Solution***:
  - The detailed step-by-step solution described at the general form
- ***Variations***:
  - Optional part. Lists the possible other variations of the antipatern.

Antipattern template (4)

- ***Applicability to Other Viewpoints and Scales*:**
  - How the antipattern afftects the other viewpoints: control, architectual, development. It also describes how relevant the antipattern is from the point of other levels.
- ***Example***:
  - This part shows an example use of the solution for the antipattern.
- ***Related Solutions***:
  - Design patterns and antipatterns that are closely related to the antipattern. The differences are also detailed here.

Software development antipattern: The Blob (1)

- ***AntiPattern Name***: *The Blob*
- ***Also Known As***: *Winnebago*, *The God Class*
- ***Most Frequent Scale***: application
- ***Refactored Solution Name***: Refactoring of Responsibilities
- ***Refactored Solution Type***: software
- ***Root Causes***: Sloth, Haste
- ***Unbalanced Forces***: Management of Functionality, Performance, Complexity
- ***Anecdotal Evidence***: "This is the class that is really the heart of our architecture."

Software development antipattern: The Blob (2)

- ***Background***: *The Blob* (1958)
  http://www.imdb.com/title/tt0051418/
- ***General Form***:
  - The Blob is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data.
  - This AntiPattern is characterized by a class diagram composed of a single complex controller class surrounded by simple data classes.
  - In general, the Blob is a procedural design even though it may be represented using object notations and implemented in object-oriented languages.
  - Frequently a result of iterative development where proof-of-concept code evolves over time into a prototype, and eventually, a production system.

Software development antipattern: The Blob (3)

- ***Symptoms and Consequences***:
  - Single class with a large number of attributes, operations, or both. A class with 60 or more attributes and operations usually indicates the presence of the Blob
  - A collection of unrelated attributes and operations encapsulated in a single class.
  - The Blob Class is typically too complex for reuse and testing.
  - The Blob Class may be expensive to load into memory, using excessive resources, even for simple operations
- ***Typical Causes***:
  - The lack of OO architecture
  - The lack of any architecture
  - Lack of architecture enforcement.
  - Too limited intervention. In iterative projects, developers tend to add little pieces of functionality to existing working classes, rather than add new classes.
  - Specified disaster.(wrong requirement specification).

Software development antipattern: The Blob (4)

- ***Known Exceptions***: Acceptable when wrapping legacy systems, when encapsulating a previous system for compatibility reasons
- ***Refactored Solution***: The solution involves code refactoring
  - Identify or categorize related attributes and operations
  - Look for "natural homes" for these contract-based collections of functionality and then migrate them there
  - Remove redundant associations

Software development antipattern: The Blob (5)

- ***Variations***:
  - Behavioral form: An object that contains a centralized process that interacts with most other parts of the system („central brain class”).
  - Data form: A class that holds attributes that are used by most of the other objects of the system („global data class”).
- ***Applicability to Other Viewpoints and Scales***: Both architectural and managerial viewpoints play key roles in the initial prevention of the Blob AntiPattern.
- ***Example***:

Software development antipattern: *Spaghetti Code* (1)

- **AntiPattern Name**: Spaghetti Code
- **Most Applicable Scale:** Application
- **Refactored Solution Name**: Software Refactoring, Code Cleanup
- **Refactored Solution Type:** Software
- **Root Causes:** Ignorance, Sloth
- **Unbalanced Forces:** Management of Complexity, Change
- **Anecdotal Evidence:** "Ugh! What a mess!" "You do realize that the language supports more than one function, right?" "It's easier to rewrite this code than to attempt to modify it." "Software engineers don't write spaghetti code." "The quality of your software structure is an investment for future modification and extension."

Software development antipattern: *Spaghetti Code* (2)

- **Background**: The classic and most famous AntiPattern; it has existed in one form or another since the invention of programming languages.
- **General Form**:
  - Spaghetti Code appears as a program or system that contains very little software structure.
  - If developed using an object-oriented language, the software may include a small number of objects that contain methods with very large implementations.

Software development antipattern: *Spaghetti Code* (3)

- **Symptoms And Consequences**:
  - Methods are very process-oriented; frequently, in fact, objects are named as processes.
  - The flow of execution is dictated by object implementation, not by the clients of the objects.
  - Minimal relationships exist between objects.
  - Many object methods have no parameters, and utilize class or global variables for processing.
  - Code is difficult to reuse, and when it is, it is often through cloning.
  - Benefits of object orientation are lost.
  - Follow-on maintenance efforts contribute to the problem.
  - The effort involved in maintaining an existing code base is greater than the cost of developing a new solution

Software development antipattern: *Spaghetti Code* (4)

- **Typical Causes**:
  - Inexperience with object-oriented design technologies
  - No mentoring in place; ineffective code reviews
  - No design prior to implementation
  - Frequently the result of developers working in isolation.
- **Known Exceptions**:
  - Reasonably acceptable when the interfaces are coherent and only the implementation is spaghetti.
- **Refactored Solution**: Software refactoring.

Software development antipattern: *Spaghetti Code* (5)

- **Example**:
- **Related Solutions**: Analysis Paralysis, Lava Flow

Software architecture antipattern: *Vendor Lock−In* (1)

- **AntiPattern Name**: *Vendor Lock−In*
- **Also Known As**: Product-Dependent Architecture, Bondage and Submission, Connector Conspiracy
- **Most Frequent Scale**: System
- **Refactored Solution Name**: Isolation Layer
- **Refactored Solution Type**: Software
- **Root Causes**: Sloth, Apathy, Pride/Ignorance
- **Unbalanced Forces**: Management of Technology Transfer, Management of Change

Software architecture antipattern: *Vendor Lock−In* (2)

- **Anecdotal Evidence**: We have often encountered software projects that claim their architecture is based upon a particular vendor or product line. Other anecdotal evidence occurs around the time of product upgrades and new application installations:
  - "When I try to read the new data files into the old version of the application, it crashes my system."
  - "The old software acts like it has a virus, but it's probably just the new application data."
- **General Form**: A software project adopts a product technology and becomes completely dependent upon the vendor's implementation.

Software architecture antipattern: *Vendor Lock−In* (3)

- **Symptoms And Consequences**:
  - Commercial product upgrades drive the application software maintenance cycle
  - Promised product features are delayed or never delivered.
  - The product varies significantly from the advertised open systems standard.
  - If a product upgrade is missed entirely, a product repurchase and reintegration is often necessary.
- **Typical Causes**:
  - The product varies from published open system standards because there is no effective conformance process for the standard
  - The product is selected based entirely upon marketing and sales information,
  - There is no technical approach for isolating application software from direct dependency upon the product.
  - The complexity and generality of the product technology greatly exceeds that of the application needs

Software architecture antipattern: *Vendor Lock−In* (4)

- **Known Exceptions**: Acceptable when a single vendor's code makes up the majority of code needed in an application.
- **Refactored Solution**: The refactioned solution is called isolation layer. An isolation layer separates software packages and technology. It can be used to provide software portability from underlying middleware and platform-specific interfaces.
- **Example**: