Software Engineering

Design Principles

Stefan Hallerstede, Carl Schultz

Aarhus University Department of Engineering

20 November 2019

Contents

Introduction

Architectural Design
- Design Principles
- Example: The “Layers” Pattern
- More Patterns

Towards Code

Design Patterns
- Architectural Styles Again

Concluding Remarks

Contents

Introduction

Architectural Design
- Design Principles
- Example: The “Layers” Pattern
- More Patterns

Towards Code

Design Patterns
- Architectural Styles Again

Concluding Remarks

How To Design A Good Architecture?

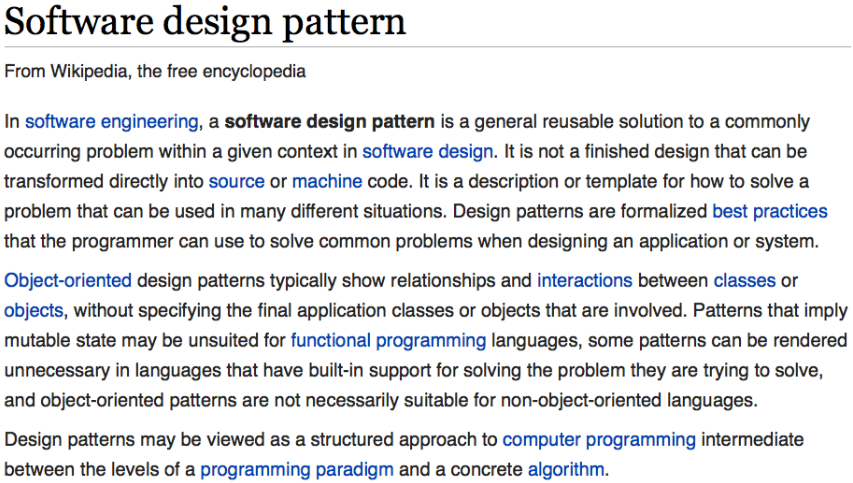
- Have we found a satisfactory answer to this question?
  → *Design principles* can guide us

- Can we do better? → *Design patterns*:

  reusable standard solutions
  to common problems
  encountered in software design

- Reuse *experience* of others

Description and Criticism



Software design pattern

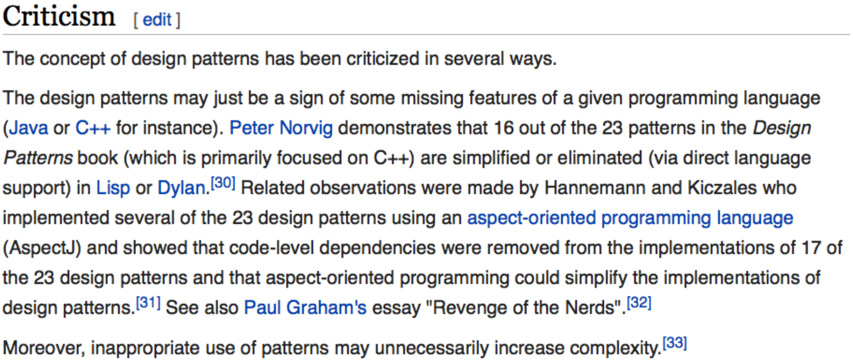
From Wikipedia, the free encyclopedia

In software engineering, a **software design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply mutable state may be unsuited for functional programming languages, some patterns can be rendered unnecessary in languages that have built-in support for solving the problem they are trying to solve, and object-oriented patterns are not necessarily suitable for non-object-oriented languages.

Design patterns may be viewed as a structured approach to computer programming intermediate between the levels of a programming paradigm and a concrete algorithm.

Description and Criticism



Criticism [edit]

The concept of design patterns has been criticized in several ways.

The design patterns may just be a sign of some missing features of a given programming language (Java or C++ for instance). Peter Norvig demonstrates that 16 out of the 23 patterns in the *Design Patterns* book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.[30] Related observations were made by Hannemann and Kiczales who implemented several of the 23 design patterns using an aspect-oriented programming language (AspectJ) and showed that code-level dependencies were removed from the implementations of 17 of the 23 design patterns and that aspect-oriented programming could simplify the implementations of design patterns.[31] See also Paul Graham's essay "Revenge of the Nerds".[32]

Moreover, inappropriate use of patterns may unnecessarily increase complexity.[33]

Contents

Introduction

Architectural Design

- Design Principles
- Example: The “Layers” Pattern
- More Patterns

Towards Code

Design Patterns

- Architectural Styles Again

Concluding Remarks

Similar Problem: Design Patterns

- Codify *design decisions* and best practices
- *Template* for how to solve a design problem
- *Not* a complete solution to a specific problem
- *Reusable* in many situations
- Design patterns applied to design problems:

  Design patterns
- How to apply a design pattern?

  How can we know whether a pattern will solve a specific problem?
- Read the *pattern documentation*

Pattern Documentation (roughly)

Context	The general situation in which the pattern applies
Problem	A short sentence or two raising the main difficulty
Forces	Issues or concerns to consider when solving the problem
Solution	A (graphical) representation of the solution to the problem
Antipatterns	Inferior or inappropriate solutions
Related patterns	Comparison to similar patterns
References	Acknowledgement of those who developed the pattern

http://msdn.microsoft.com/en-us/library/ff649977.aspx

Contents

Introduction

Architectural Design

- Design Principles
- Example: The “Layers” Pattern
- More Patterns

Towards Code

Design Patterns

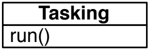
- Architectural Styles Again

Concluding Remarks

Some Design Principles

- Divide and conquer
  (small problems are better than big problems)
- Increase cohesion
  (keep related things together)
- Reduce coupling
  (especially avoid unwanted side effects)
- Abstraction
  (information hiding)
- Reuse existing designs
  (do not copy!)
- Ensure testability
  (specify!)
- Defensive design
  (design by contract)

Divide And Conquer

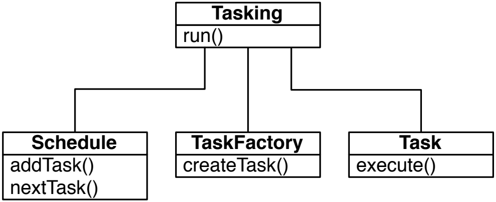


Tasking

run()

- Smaller components easier to
  - understand
  - implement
  - replace
- Opportunities for reuse
- Well-supported: packages, classes, methods

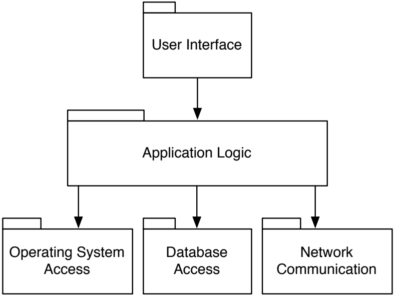
Divide And Conquer



- Smaller components easier to
  - understand
  - implement
  - replace
- Opportunities for reuse
- Well-supported: packages, classes, methods

Increase Cohesion

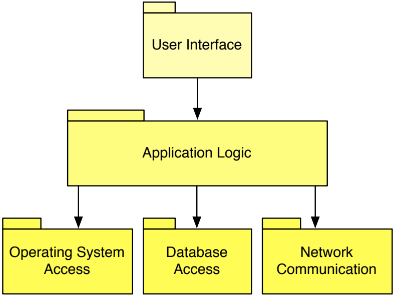
- Functional cohesion *One computation without side effects*
- Layer cohesion *Related services in a strict hierarchy*



- Temporal cohesion *Phase of execution, e.g., initialisation*
- Utility *Related utilities (no strong cohesion among them)*

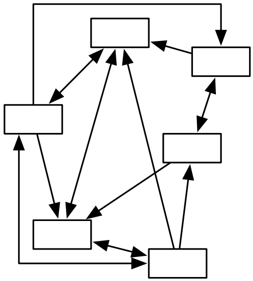
Increase Cohesion

- Functional cohesion *One computation without side effects*
- Layer cohesion *Related services in a strict hierarchy*



- Temporal cohesion *Phase of execution, e.g., initialisation*
- Utility *Related utilities (no strong cohesion among them)*

Reduce Coupling

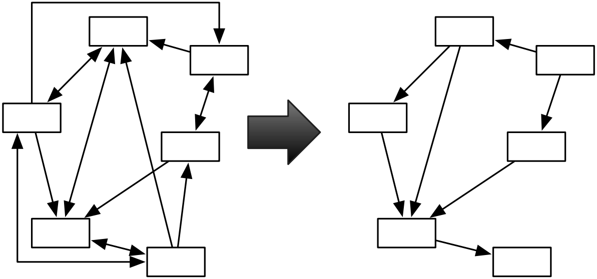


- Common coupling *use of global variables*
- Control coupling *use of control flag:*

```
void draw(int f) { if (f==1) ci(); else if (f==2) sq(); }
```

- Routine call coupling *encapsulate repeated sequences*
- Content coupling *surreptitious modification of (internal) data*

Reduce Coupling



- Common coupling *use of global variables*
- Control coupling *use of control flag:*

```
void draw(int f) { if (f==1) ci(); else if (f==2) sq(); }
```

- Routine call coupling *encapsulate repeated sequences*
- Content coupling *surreptitious modification of (internal) data*

Content Coupling

Class `java.awt.Point` of the Java Core API

```
public class Line
{
  private Point start, end;
  ...
  public Point getStart() {
   return start;
  }
  public Point getEnd()  {
   return end;
  }
}
```

```
public class Arch
{
  private Line baseline;
  ...
  void slant(int newY)
  {
   Point p = baseline.getEnd();
   p.setLocation(p.getX(),newY);
  }
}
...
```

Content Coupling

Class `java.awt.Point` of the Java Core API

```
public class Line
{
  private Point start, end;
  ...
  public Point getStart() {
   return start;
  }
  public Point getEnd()  {
   return end;
  }
}
```

```
public class Arch
{
  private Line baseline;
  ...
  void slant(int newY)
  {
   Point p = baseline.getEnd();
   p.setLocation(p.getX(),newY);
  }
}
...
```

The problem is not easy to spot!

Content Coupling

Class `java.awt.Point` of the Java Core API

```
public class Line
{
  private Point start, end;
  ...
  public Point getStart() {
   return start;
  }
  public Point getEnd()  {
   return end;
  }
}
```

```
public class Arch
{
  private Line baseline;
  ...
  void slant(int newY)
  {
   Point p = baseline.getEnd();
   p.setLocation(p.getX(),newY);
  }
}
...
```

Side effect!

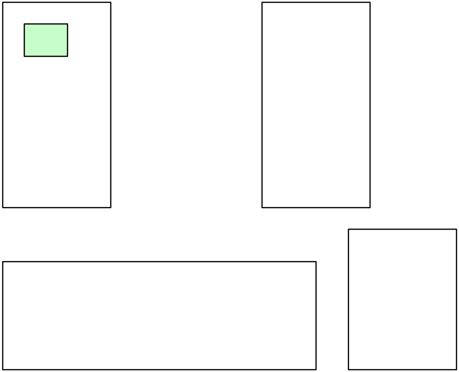
Content Coupling Antidotes

- *Encapsulate* all instance variables
  - Declare them *private* to the containing class
  - Design class API cautiously
- Make instance variables *immutable*
  - But does not always help:
  - *instance variables of instance variables!*
- Use *design patterns*
  - Has someone else already solved your problem?

Abstraction

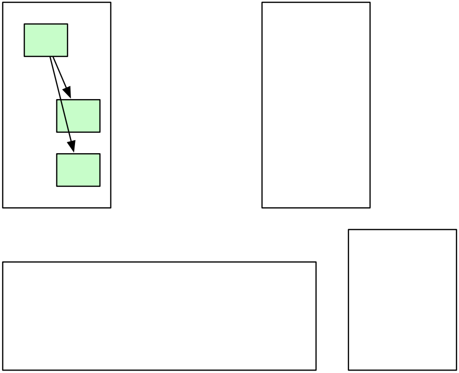
- Hide implementation detail
- Procedural abstraction
- Data abstraction
- Some abstractions are only available during modelling
- Underspecification

Reuse Existing Designs



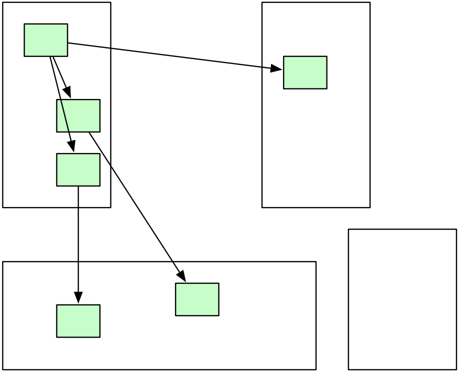
- Cloning is not a form of reuse
- Modification and fixing errors becomes very difficult

Reuse Existing Designs



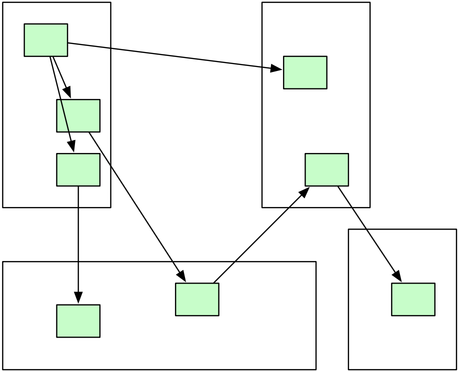
- Cloning is not a form of reuse
- Modification and fixing errors becomes very difficult

Reuse Existing Designs



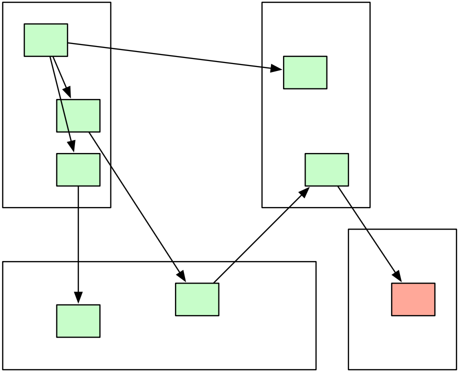
- Cloning is not a form of reuse
- Modification and fixing errors becomes very difficult

Reuse Existing Designs



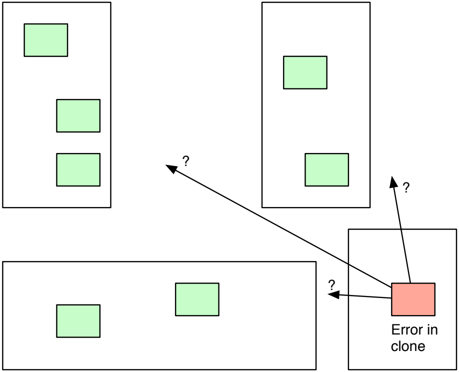
- Cloning is not a form of reuse
- Modification and fixing errors becomes very difficult

Reuse Existing Designs



- Cloning is not a form of reuse
- Modification and fixing errors becomes very difficult

Reuse Existing Designs



- Cloning is not a form of reuse
- Modification and fixing errors becomes very difficult

Ensure Testability

- Make sure all functionality can be tested
- Design functionality independently of UI and other IO
- Is there a proper behavioral description of the component?
    - Good documentation indicates this!
- Test proper components
    - Do not add (redundant) testing APIs
        - You won’t test the real API
        - You will expose otherwise private instance variables

Defensive Design

- Specify contracts
- Preconditions
- Postconditions
- Assertions
- Invariants

- (Formal Methods!)

Contents

Introduction

Architectural Design

- Design Principles
- Example: The “Layers” Pattern
- More Patterns

Towards Code

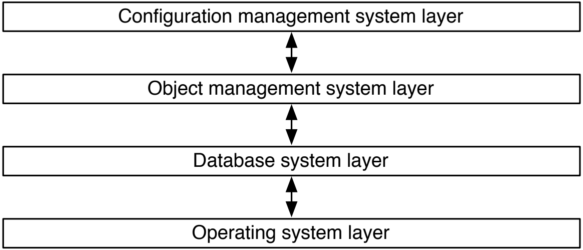
Design Patterns

- Architectural Styles Again

Concluding Remarks

Layering

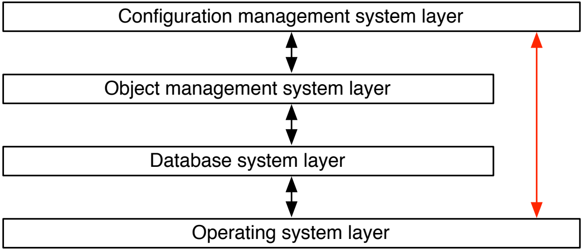
E.g. a version management system


Configuration management system layer
Object management system layer
Database system layer
Operating system layer

- Each layer provides services for the next layer.
- A layer may be considered an *abstract machine* defining a “language”.
- The next level is implemented in terms of this language.
- Easily changeable and portable.
- It may be difficult to adhere to strictly because of *poor performance*.

Layering

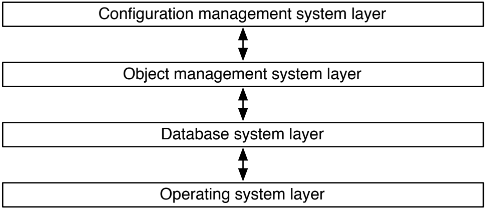
E.g. a version management system


Configuration management system layer
Object management system layer
Database system layer
Operating system layer

- Each layer provides services for the next layer.
- A layer may be considered an *abstract machine* defining a “language”.
- The next level is implemented in terms of this language.
- Easily changeable and portable.
- It may be difficult to adhere to strictly because of *poor performance*.

The “Layers” Architectural Style

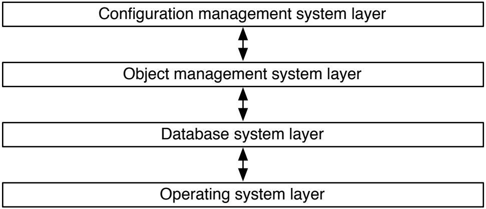
- Context?
- Problem?
- Forces?
- Solution? *we sketched this before:*


Configuration management system layer
Object management system layer
Database system layer
Operating system layer

- Antipatterns?
- Related patterns?
- References?

The “Layers” Architectural Style

- Context?
- Problem?
- Forces?
- Solution? *we sketched this before:*


Configuration management system layer
Object management system layer
Database system layer
Operating system layer

- Antipatterns?
- Related patterns?
- References?

Use pattern libraries, e.g.

http://msdn.microsoft.com/en-us/library/ee658117.aspx#LayeredStyle

Do “Layers” Adhere To The Design Principles?

- Divide and conquer
  (separate layers can be developed independently)
- Increase cohesion
  (layers provide sets of related services)
- Reduce coupling
  (layers only rely on lower layers)
- Abstraction
  (higher layers are designed independently of lower layer details)
- Reuse existing designs
  (often layers designed by someone else are available)
- Ensure testability
  (layers can be tested independently)
- Defensive design
  (layer APIs are natural places for rigorous checks)

Contents

Introduction

Architectural Design
- Design Principles
- Example: The “Layers” Pattern
- More Patterns

Towards Code

Design Patterns
- Architectural Styles Again

Concluding Remarks

Architectural Styles[1]

Pattern	Description
Client/Server	Segregates the system into two applications, where the client makes requests to the server. In many cases, the server is a database with application logic represented as stored procedures.
Component-Based Architecture	Decomposes application design into reusable functional or logical components that expose well-defined communication interfaces.
Layered Architecture	Partitions the concerns of the application into stacked groups (layers).
Message Bus	An architecture style that prescribes use of a software system that can receive and send messages using one or more communication channels, so that applications can interact without needing to know specific details about each other.
N-Tier / 3-Tier	Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer.

[1] http://msdn.microsoft.com/en-us/library/ee658117.aspx

Contents

Introduction

Architectural Design
- Design Principles
- Example: The “Layers” Pattern
- More Patterns

Towards Code

Design Patterns
- Architectural Styles Again

Concluding Remarks

Patterns: From Architecture To Code

- **Architectural styles:** Higher-level design
- **Design patterns:** Lower-level design, e.g.
  - Singleton
  - Adapter
  - Observer
- **Programming idioms:** Coding, e.g. increment
  - “`i = i + 1`” BASIC
  - “`i += 1`’’, “`i++`’’, “`++i`’’ C

Contents

Introduction

Architectural Design

- Design Principles
- Example: The “Layers” Pattern
- More Patterns

Towards Code

Design Patterns

- Architectural Styles Again

Concluding Remarks

The “Singleton” Pattern

- **Context:**

  Need class for which only one instance exists (singleton).

- **Problem:**

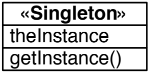
  How to enforce creation of at most one instance of some class?

- **Forces:**

  Public constructor cannot provide this guarantee.

  The singleton instance must be publicly accessible.

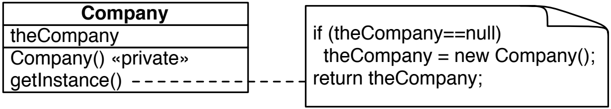
- **Solution:**



«Singleton»

theInstance

getInstance()



Company

theCompany

Company() «private»

getInstance()

```
if (theCompany==null)
  theCompany = new Company();
return theCompany;
```

The “Adapter” Pattern

- **Context:**

  You are building an *inheritance hierarchy* and want to incorporate into it an existing class.

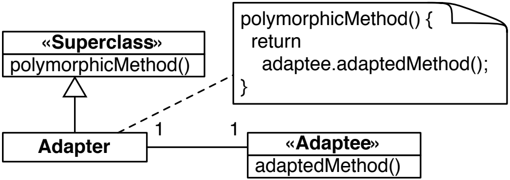
- **Problem:**

  How to obtain the power of polymorphism when reusing a class whose methods have the sought function but *not the fitting signature*

- **Forces:**

  No access to multiple inheritance or do not want to use it.

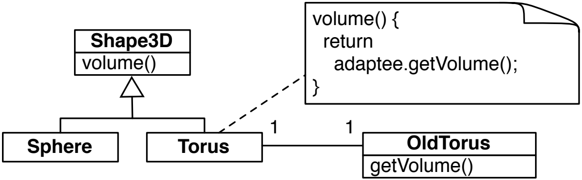
- **Solution:**



```
polymorphicMethod() {
  return
    adaptee.adaptedMethod();
}
```

The “Adapter” Pattern

- **Context:**
  You are building an *inheritance hierarchy* and want to incorporate into it an existing class.
- **Problem:**
  How to obtain the power of polymorphism when reusing a class whose methods have the sought function but *not the fitting signature*
- **Forces:**
  No access to multiple inheritance or do not want to use it.
- **Solution:**



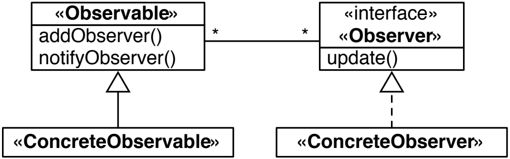
```
volume() {
  return
    adaptee.getVolume();
}
```

The “Adapter” Pattern

- **Context:**
  You are building an *inheritance hierarchy* and want to incorporate into it an existing class.
- **Problem:**
  How to obtain the power of polymorphism when reusing a class whose methods have the sought function but *not the fitting signature*
- **Forces:**
  No access to multiple inheritance or do not want to use it.
- **Solution: —**
- **Related patterns:**
  - *Façade:*
    Provides a single class for easy access to larger collection of classes
  - *Proxy:*
    Provides a lightweight class that hides a heavyweight class

The “Observer” Pattern

- **Context:**
  When an association is created between two classes, the code for the classes becomes inseparable.
- **Problem:**
  How to reduce the interconnection between classes?
- **Forces:**
  Maximize the flexibility of the system.
- **Solution:**



The “Observer” Pattern

- **Context:**

  When an association is created between two classes, the code for the classes becomes inseparable.

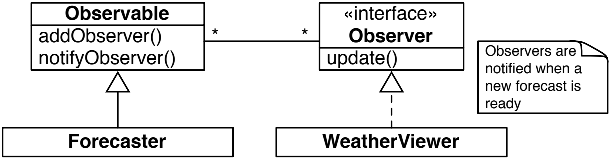
- **Problem:**

  How to reduce the interconnection between classes?

- **Forces:**

  Maximize the flexibility of the system.

- **Solution:**



Observers are notified when a new forecast is ready

The “Observer” Pattern

- **Context:**

  When an association is created between two classes, the code for the classes becomes inseparable.
- **Problem:**

  How to reduce the interconnection between classes?
- **Forces:**

  Maximize the flexibility of the system.
- **Solution: —**
- **Antipatterns:**
  - Connect an observer directly to an observable
    so that they both have references to each other.
  - Make the observers subclasses of the observable.

We Can Group Design Patterns Roughly

- **Creational Patterns** *(object creation)*

  Example: *Singleton*

- **Structural Patterns** *(object relationships)*

  Example: *Adapter*

- **Behavioral patterns** *(communication between objects)*

  Example: *Observer*

Contents

Introduction

Architectural Design
- Design Principles
- Example: The “Layers” Pattern
- More Patterns

Towards Code

Design Patterns
- Architectural Styles Again

Concluding Remarks

Architectural Styles

- Standard solutions to *architectural problems*
- Applies to *subsystems*
- Design patterns or idioms on a *large scale*
- Only capture the *essentials* of an architecture
- Are *not* themselves architectures
- May have *many* architectures as “implementations”

Contents

Introduction

Architectural Design
- Design Principles
- Example: The “Layers” Pattern
- More Patterns

Towards Code

Design Patterns
- Architectural Styles Again

Concluding Remarks

Concluding Remarks

How to design a good architecture?

- Follow *design principles*
- Design patterns capture *experience* and good practice
- Architectural styles are *large* design patterns
- Use design and architectural *patterns* and *styles* when possible

But,

- There is not a pattern/style for *every* problem
- Do not *overuse* patterns/styles (similarly to inheritance)
- Study patterns/styles *before* applying them
- Wrong usage will have *adverse* effect on quality indicators
- Use patterns/styles *with moderation*