Design Patterns and Refactoring

Dario Campagna

Head of Research and Development

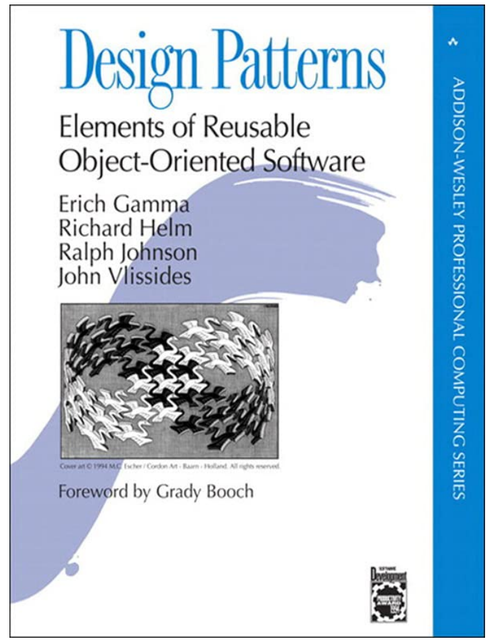
Each pattern is a three-part rule, which express a relation between a certain context, a problem and a solution.

Christopher Alexander

Design Patterns

Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

- Presents 23 patterns
- Discuss alternative patterns to consider
- Defines families of related patterns



Design Patterns classification

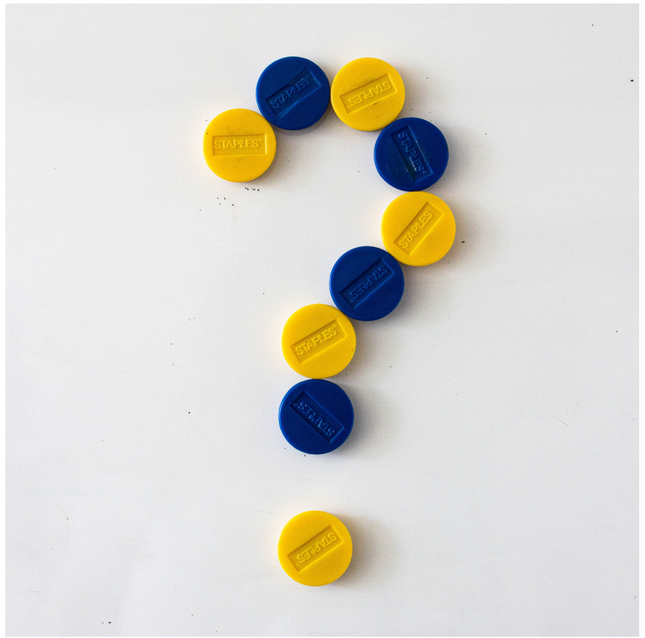
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

See https://refactoring.guru/design-patterns.

Why Design Patterns?

Patterns convey many useful design ideas.

- Designing object-oriented software is hard
- Designing reusable object-oriented software is even harder
- Good solution that worked in the past can be reused
- Record experience in designing object-oriented software



Abstract Factory

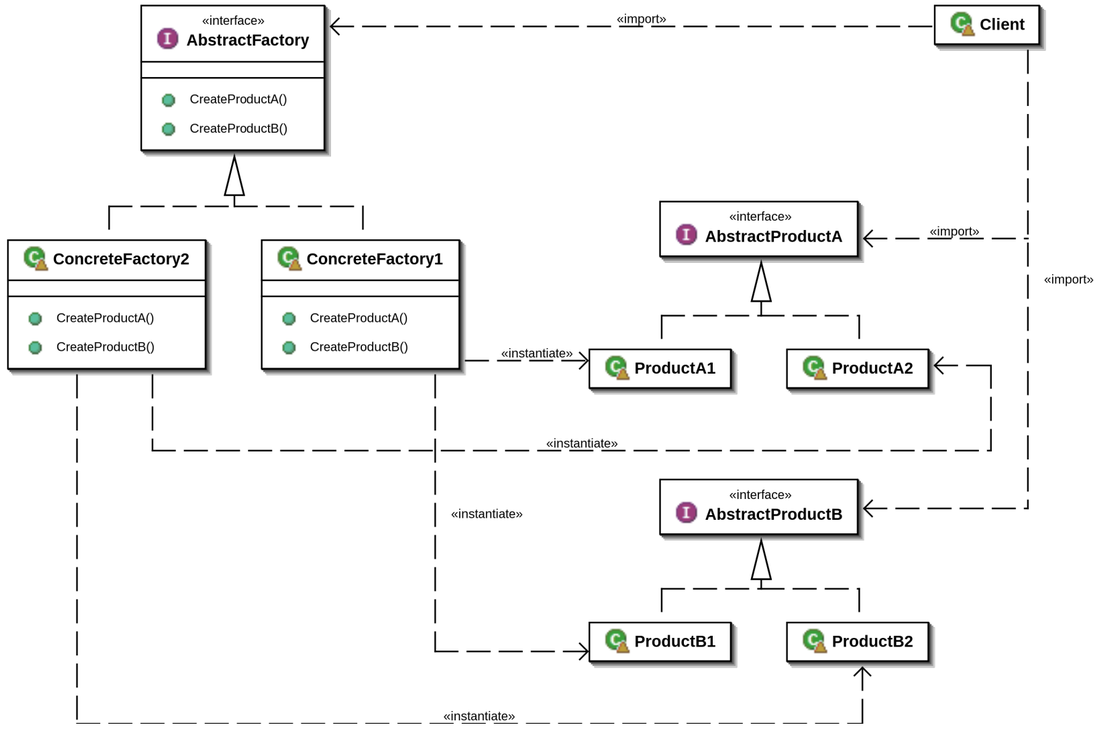
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Motivation

- User interface toolkit supporting multiple look-and-feels.
- Application that needs to be portable and hence encapsulate platform dependencies.

Applicability

- A system that should be independent from object creation, composition, representation.
- A system that should be configured with one of multiple families of objects.
- ...



What does this code do?

```
public interface MessageStrategy {
  public void sendMessage();
}
```

```
public abstract class AbstractStrategyFactory {
  public abstract MessageStrategy createStrategy(MessageBody mb);
}
```

```
public class MessageBody {

  Object payload;

  public Object getPayload() {
  return payload;
  }

  public void configure(Object obj) {
  payload = obj;
  }

  public void send(MessageStrategy ms) {
  ms.sendMessage();
  }
}
```

```
public class DefaultFactory extends AbstractStrategyFactory {

  private DefaultFactory() {;}
  static DefaultFactory instance;

  public static AbstractStrategyFactory getInstance() {
    if (instance==null) instance = new DefaultFactory();
    return instance;
  }

  public MessageStrategy createStrategy(final MessageBody mb) {
    return new MessageStrategy() {
      MessageBody body = mb;

      public void sendMessage() {
        Object obj = body.getPayload();
        System.out.println((String)obj);
      }
    };
  }
}

public class HelloWorld {

  public static void main(String[] args) {
    MessageBody mb = new MessageBody();
    mb.configure("Hello World!");
    AbstractStrategyFactory asf = DefaultFactory.getInstance();
    MessageStrategy strategy = asf.createStrategy(mb);
    mb.send(strategy);
  }
}
```

From https://developers.slashdot.org/comments.pl?sid=33602&cid=3636102.

This “Hello World” code is…

```
public interface MessageStrategy {
  public void sendMessage();
}

public abstract class AbstractStrategyFactory
{
  public abstract MessageStrategy
  createStrategy(MessageBody mb);
}

public class MessageBody {

  Object payload;

  public Object getPayload() {
  return payload;
  }

  public void configure(Object obj) {
  payload = obj;
  }

  public void send(MessageStrategy ms) {
  ms.sendMessage();
  }
}

public class DefaultFactory extends
AbstractStrategyFactory {

  private DefaultFactory() {;}
  static DefaultFactory instance;

  public static AbstractStrategyFactory
  getInstance() {
    if (instance==null) instance = new
    DefaultFactory();
    return instance;
  }

  public MessageStrategy createStrategy(final
  MessageBody mb) {
    return new MessageStrategy() {
      MessageBody body = mb;

      public void sendMessage() {
        Object obj = body.getPayload();
        System.out.println((String)obj);
      }
    };
  }
}

public class HelloWorld {

  public static void main(String[] args) {
    MessageBody mb = new MessageBody();
    mb.configure("Hello World!");
    AbstractStrategyFactory asf =
    DefaultFactory.getInstance();
    MessageStrategy strategy =
    asf.createStrategy(mb);
    mb.send(strategy);
  }
}
```

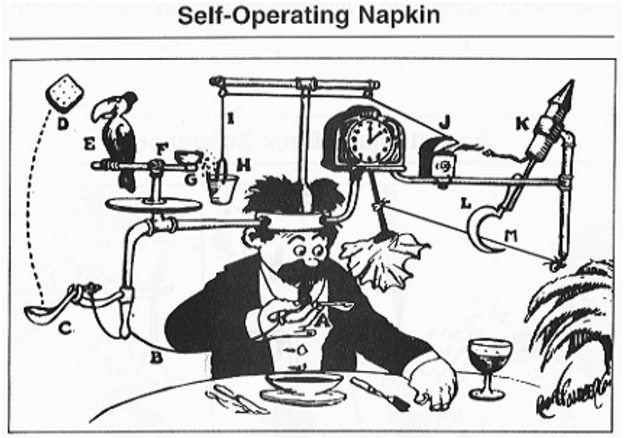
1. Flexible

2. Complex

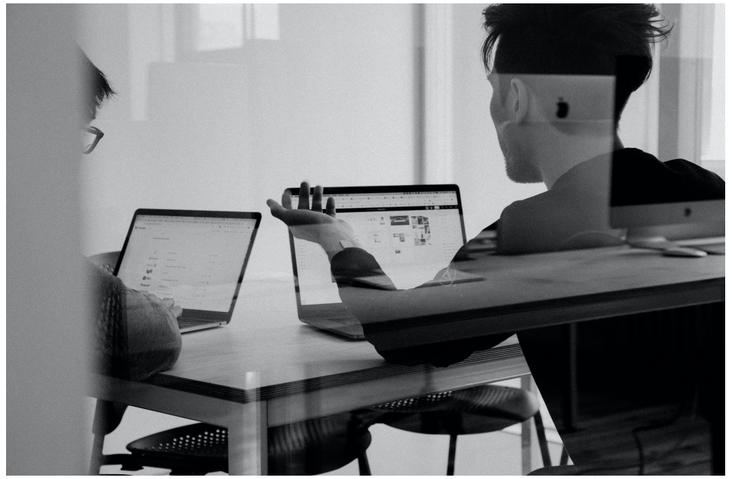
3. Poorly designed

Over-engineered code

A consequence of pattern overuse



More complicated that it needs to be



Higher learning costs

Reduced productivity

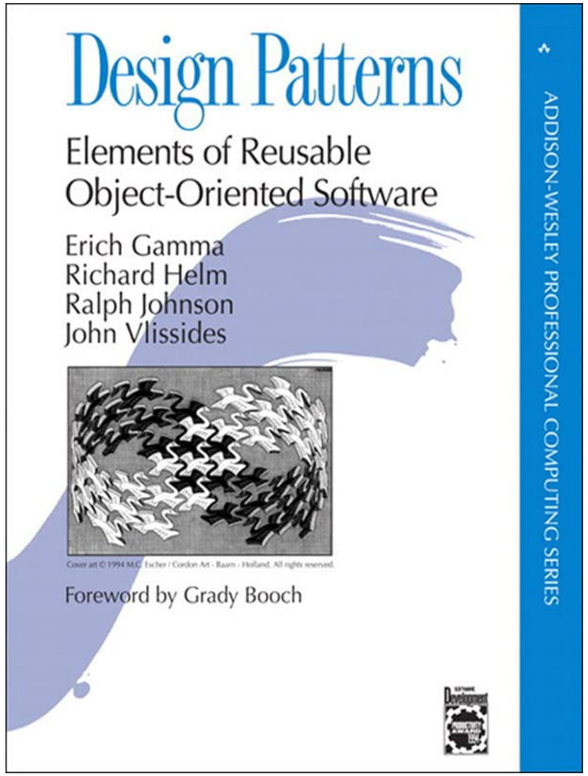
Pattern should be used wisely

Refactoring can help us do that by focusing our attention on…

- Removing duplication
- Simplifying logic
- Communicating intention
- Increasing flexibility

Our design patterns capture many of the structures that results from refactoring… Design patterns thus provide targets for your refactorings.

The authors of “Design Patterns”



Patterns are where you want to be; refactorings are ways to get there from somewhere else.

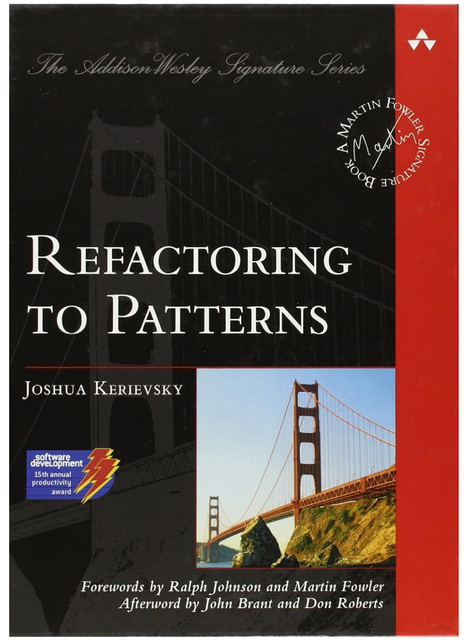
Martin Fowler in “Refactoring”

Refactoring to Patterns

A book about the marriage of refactoring with patterns.

Suggests that using patterns to improve an existing design is better than using patterns early in a new design.

- Catalog of 27 refactorings.
- Examples of different ways to implement the same pattern.
- Advice for when to refactor to, towards, or away from patterns.



Refactoring to, towards, and away from Patterns

Directions of refactoring

To

You refactor the code to implement a Design Pattern.

Towards

You take some steps toward a Design Pattern and stop when you made a good enough design improvement.

Away

If your design has not improved, you backtrack or refactor in another direction.