Lecture 13: Better Living with Patterns

SE313, Software Design and Architecture

Damla Oguz

Design Pattern defined

A Pattern is a solution to a problem in a context.

- The **context** is the situation in which the pattern applies. This should be a recurring situation.
  - Example: You have a collection of objects.
- The **problem** refers to the goal you are trying to achieve in this context, but it also refers to any constraints that occur in the context.
  - Example: You need to step through the objects without exposing the collection's implementation.
- The **solution** is what you're after: a general design that anyone can apply which resolves the goal and set of constraints.
  - Example: Encapsulate the iteration into a separate class.

Design Pattern defined (cont.)

- Here’s a little mnemonic you can repeat to yourself to remember it:
  - “If you find yourself in a context with a problem that has a goal that is affected by a set of constraints, then you can apply a design that resolves the goal and constraints and leads to a solution.”
- After all, you already know that a Design Pattern gives you a solution to a common recurring design problem. What is all this formality getting you?
  - Well, you’re going to see that by having a formal way of describing patterns we can create a catalog of patterns, which has all kinds of benefits.

Example

- We need a problem, a solution and a context:
  - Problem: How do I get to work on time?
  - Context: I've locked my keys in the car.
  - Solution: Break the window, get in the car, start the engine and drive to work.
- We have all the components of the definition:
  - We have a problem, which includes the goal of getting to work, and the constraints of time, distance and probably some other factors.
  - We also have a context in which the keys to the car are inaccessible.
  - And we have a solution that gets us to the keys and resolves both the time and distance constraints.
- We must have a pattern now! Right?

Looking more closely at the Design Pattern definition

- Our example does seem to match the Design Pattern definition, but it isn't a true pattern. Why?
  - We know that a pattern needs to apply to a recurring problem.
  - While an absent-minded person might lock his keys in the car often, breaking the car window doesn't qualify as a solution that can be applied over and over (or at least isn't likely to if we balance the goal with another constraint: cost).

Looking more closely at the Design Pattern definition (cont.)

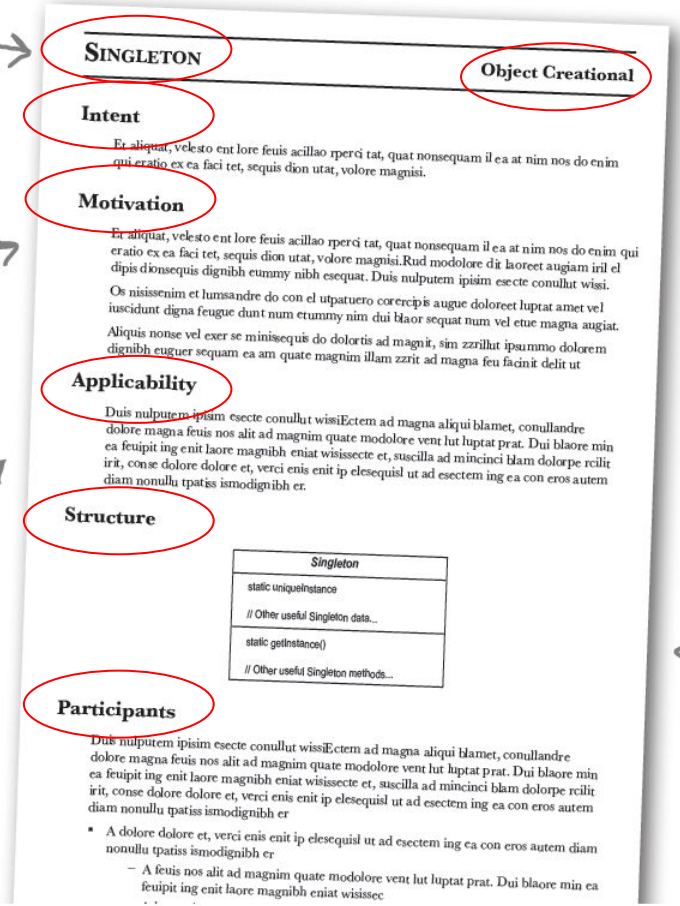
- It also fails in a couple of other ways:
  - First, it isn't easy to take this description, hand it to someone and have him apply it to his own unique problem.
  - Second, we've violated an important but simple aspect of a pattern: we haven't even given it a name!
    - Without a name, the pattern doesn't become part of a vocabulary that can be shared with other developers.
- Luckily, patterns are not described and documented as a simple problem, context and solution; we have much better ways of describing patterns and collecting them together into patterns catalogs.

All patterns in a catalog start with a name. The name is a vital part of a pattern – without a good name, a pattern can't become part of the vocabulary that you share with other developers.

The motivation gives you a concrete scenario that describes the problem and how the solution solves the problem.

The applicability describes situations in which the pattern can be applied.

The participants are the classes and objects in the design. This section describes their responsibilities and roles in the pattern.



SINGLETON — **Object Creational**

Intent

Et aliquat, velesto ent lore feuis acillao rperci tat, quat nonsequam il ea at nim nos do enim qui eratio ex ea faci tet, sequis dion utat, volore magnisi.

Motivation

Et aliquat, velesto ent lore feuis acillao rperci tat, quat nonsequam il ea at nim nos do enim qui eratio ex ea faci tet, sequis dion utat, volore magnisi.Rud modolore dit laoreet augiam iril el dipis dionsequis dignibh eummy nibh esequat. Duis nulputem ipisim esecte conullut wissi.

Os nisissenim et lumsandre do con el utpatuero corercipis augue doloreet luptat amet vel iuscidunt digna feugue dunt num etummy nim dui blaor sequat num vel etue magna augiat.

Aliquis nonse vel exer se minisequis do dolortis ad magnit, sim zzrillut ipsummo dolorem dignibh euguer sequam ea am quate magnim illam zzrit ad magna feu facinit delit ut

Applicability

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Structure

Participants

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er

- A dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er
  - A feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissec

This is the pattern's classification or category. We'll talk about these in a few pages.

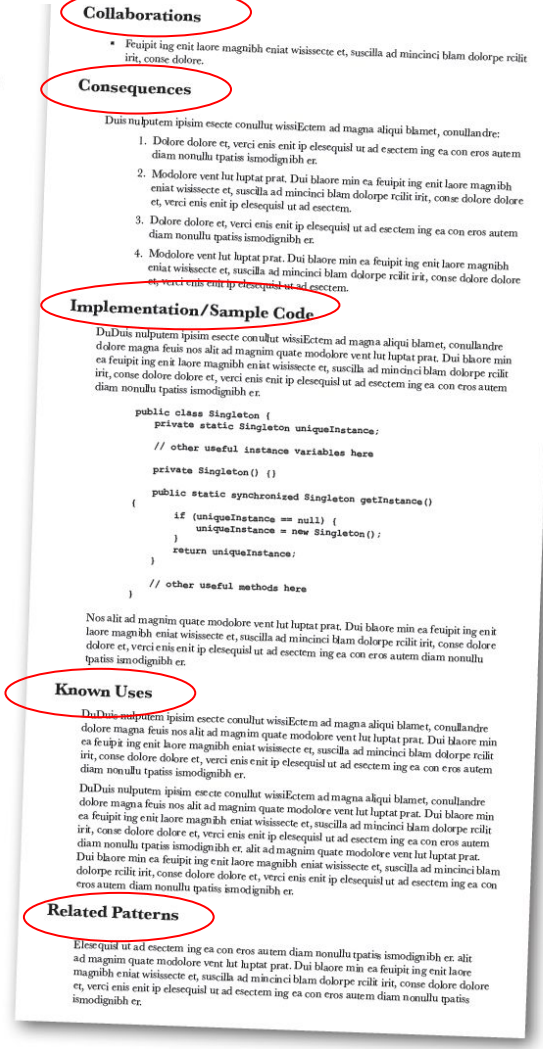
The intent describes what the pattern does in a short statement. You can also think of this as the pattern's definition (just like we've been using in this book).

The structure provides a diagram illustrating the relationships among the classes that participate in the pattern.

The consequences describe the effects that using this pattern may have: good and bad.

Implementation provides techniques you need to use when implementing this pattern, and issues you should watch out for.

Known Uses describes examples of this pattern found in real systems.



Collaborations

- Feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore.

Consequences

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre:

1. Dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.
2. Modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem.
3. Dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.
4. Modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem.

Implementation/Sample Code

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

```
public class Singleton {
   private static Singleton uniqueInstance;

   // other useful instance variables here

   private Singleton() {}

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

   // other useful methods here
}
```

Nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Known Uses

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er. alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Related Patterns

Elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er. alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Collaborations tells us how the participants work together in the pattern.

Sample Code provides code fragments that might help with your implementation.

Related Patterns describes the relationship between this pattern and others.

A Question

Q: Is it possible to create your own Design Patterns? Or is that something you have to be a “patterns guru” to do?

A: First, remember that patterns are discovered, not created. So, anyone can discover a Design Pattern and then author its description; however, it’s not easy and doesn’t happen quickly, nor often. Being a “patterns writer” takes commitment.

You should first think about why you’d want to—the majority of people don’t author patterns; they just use them. However, you might work in a specialized domain for which you think new patterns would be helpful, or you might have come across a solution to what you think is a recurring problem, or you may just want to get involved in the patterns community and contribute to the growing body of work.

So you wanna be a design patterns star?

Well, listen now to what I tell.

Get yourself a patterns catalog,

Then take some time and learn it well.

And when you've got your description right,

And three developers agree without a fight,

Then you'll know it's a pattern alright.

To the tune of "So you wanna be a Rock'n Roll Star."

So you wanna be a Design Patterns writer

- **Do you homework.**
  - You need to be well versed in the existing patterns before you can create a new one.
  - Most patterns that appear to be new, are, in fact, just variants of existing patterns.
  - By studying patterns, you become better at recognizing them, and you learn to relate them to other patterns.
- **Take time to reflect, evaluate.**
  - Your experience (the problems you've encountered, and the solutions you've used) are where ideas for patterns are born.
  - So take some time to reflect on your experiences and comb them for novel designs that recur.

So you wanna be a Design Patterns writer (cont.)

- **Get your ideas down on paper in a way others can understand.**
  - You need to document your pattern candidates so that others can read, understand, and apply them to their own solution and then supply you with feedback.
  - Luckily, you don’t need to invent your own method of documenting your patterns.
  - As you’ve already seen with the GoF template, a lot of thought has already gone into how to describe patterns and their characteristics.

So you wanna be a Design Patterns writer (cont.)

- **Have others try your patterns; then refine and refine some more.**
  - Don’t expect to get your pattern right the first time. Think of your pattern as a work in progress that will improve over time.
  - Have other developers review your candidate pattern, try it out, and give you feedback. Incorporate that feedback into your description and try again.
  - Your description will never be perfect, but at some point it should be solid enough that other developers can read and understand it.
- **Don’t forget the Rule of Three.**
  - Remember, unless your pattern has been successfully applied in three real-world solutions, it can’t qualify as a pattern.
  - That’s another good reason to get your pattern into the hands of others so they can try it, give feedback, and allow you to converge on a working pattern.

Organizing Design Patterns

- As the number of discovered Design Patterns grows, it makes sense to partition them into classifications.
- So that we can organize them, narrow our searches to a subset of all Design Patterns, and make comparisons within a group of patterns.
- The most well-known scheme was used by the first patterns catalog and partitions patterns into three distinct categories based on their **purposes**:
  - **Creational**
  - **Behavioral**
  - **Structural**

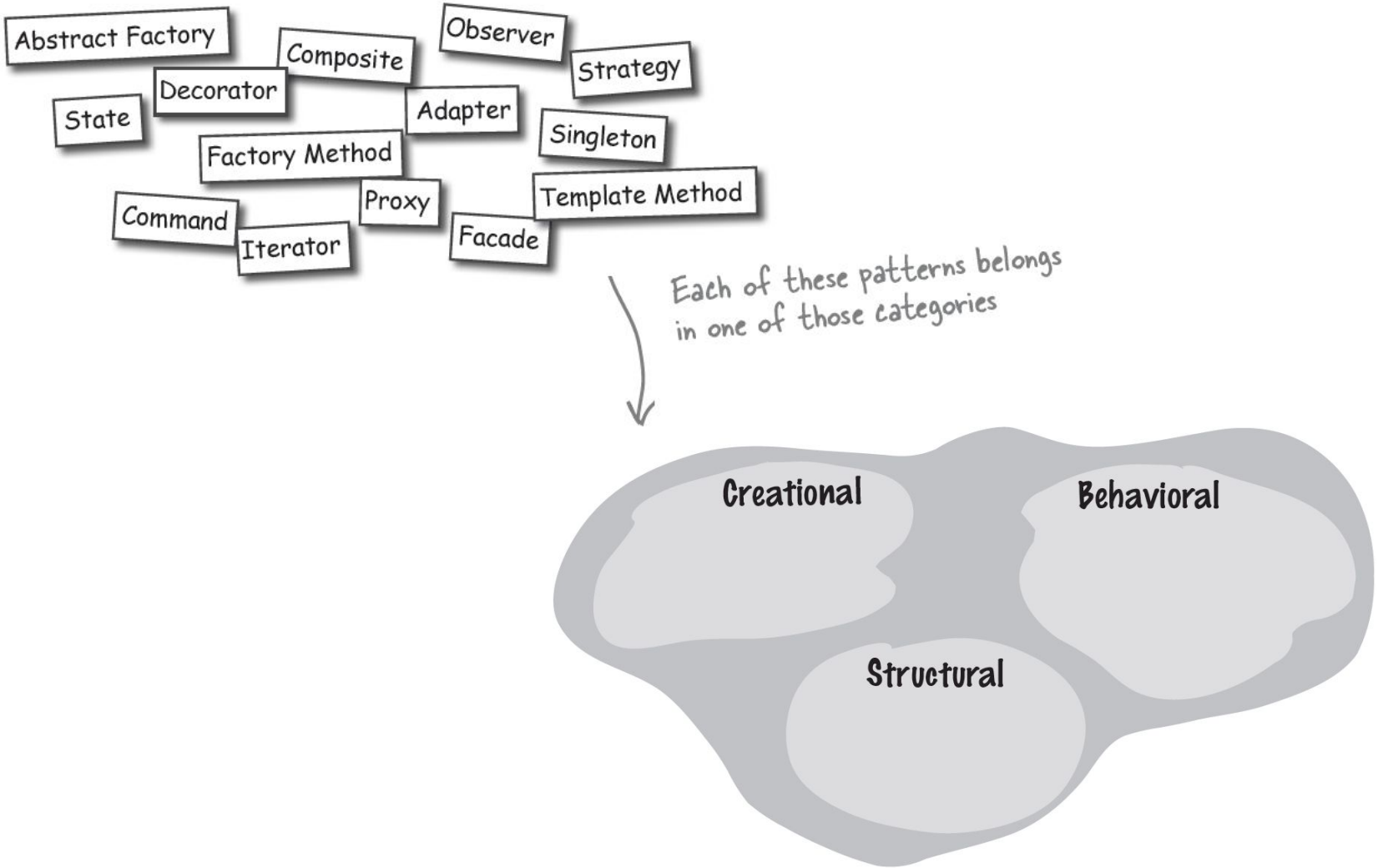
Organizing Design Patterns (cont.)

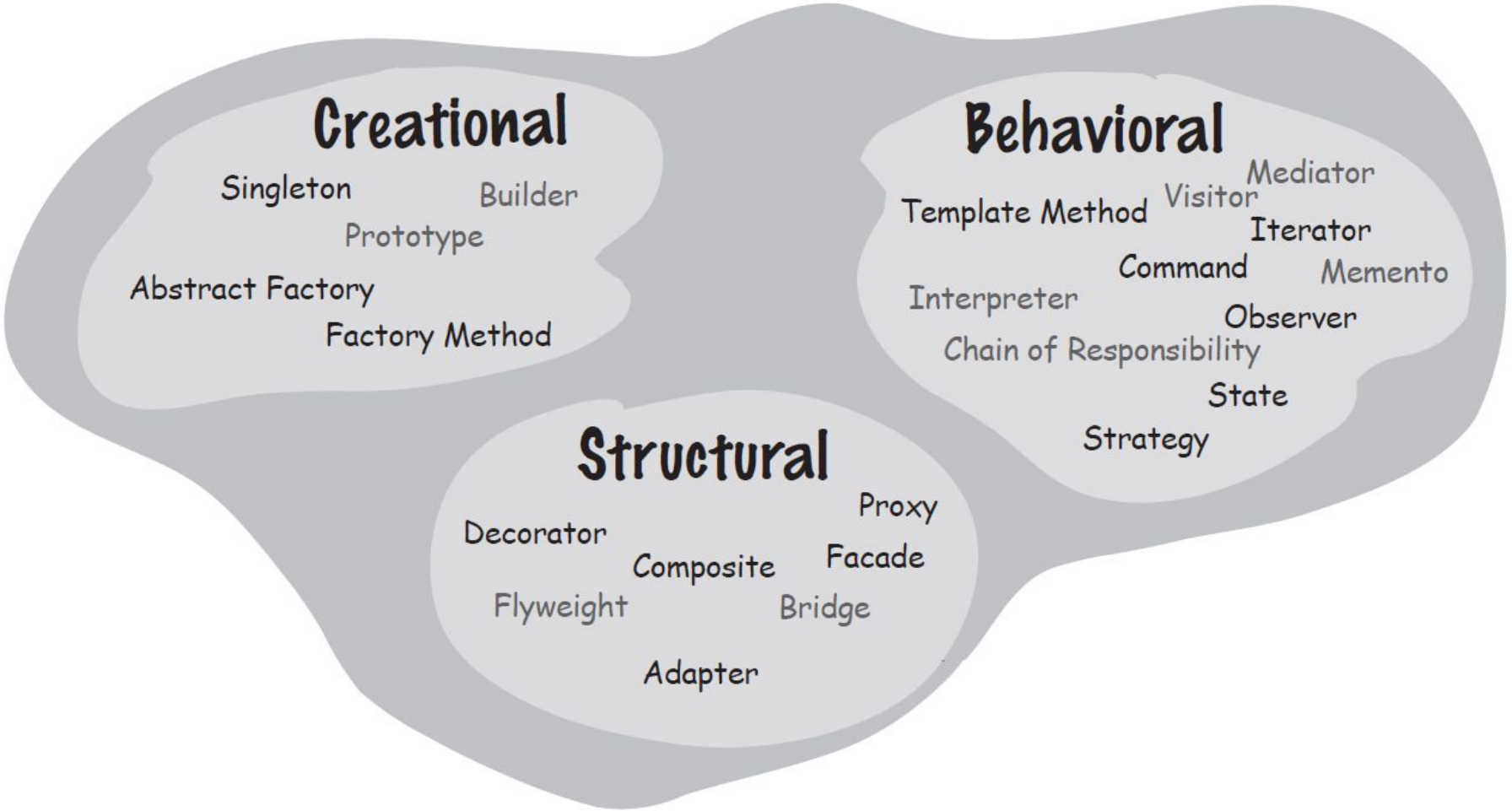
- **Creational Patterns** involve object instantiation and all provide a way to decouple a client from the objects it needs to instantiate.
- Any pattern that is a **Behavioral Pattern** is concerned with how classes and objects interact and distribute responsibility.
- **Structural Patterns** let you compose classes or objects into larger structures.

Organizing Design Patterns (cont.)

- **Creational Patterns:** These design patterns provide ways to create objects while hiding the *creation logic*, instead of instantiating objects directly using the new operator. This gives the program more flexibility in deciding which objects need to be created for a given *use case*.
- **Behavioral Patterns:** These design patterns are specifically concerned with communication between objects.
- **Structural Patterns**: These design patterns deal with class and object composition. The concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionality.

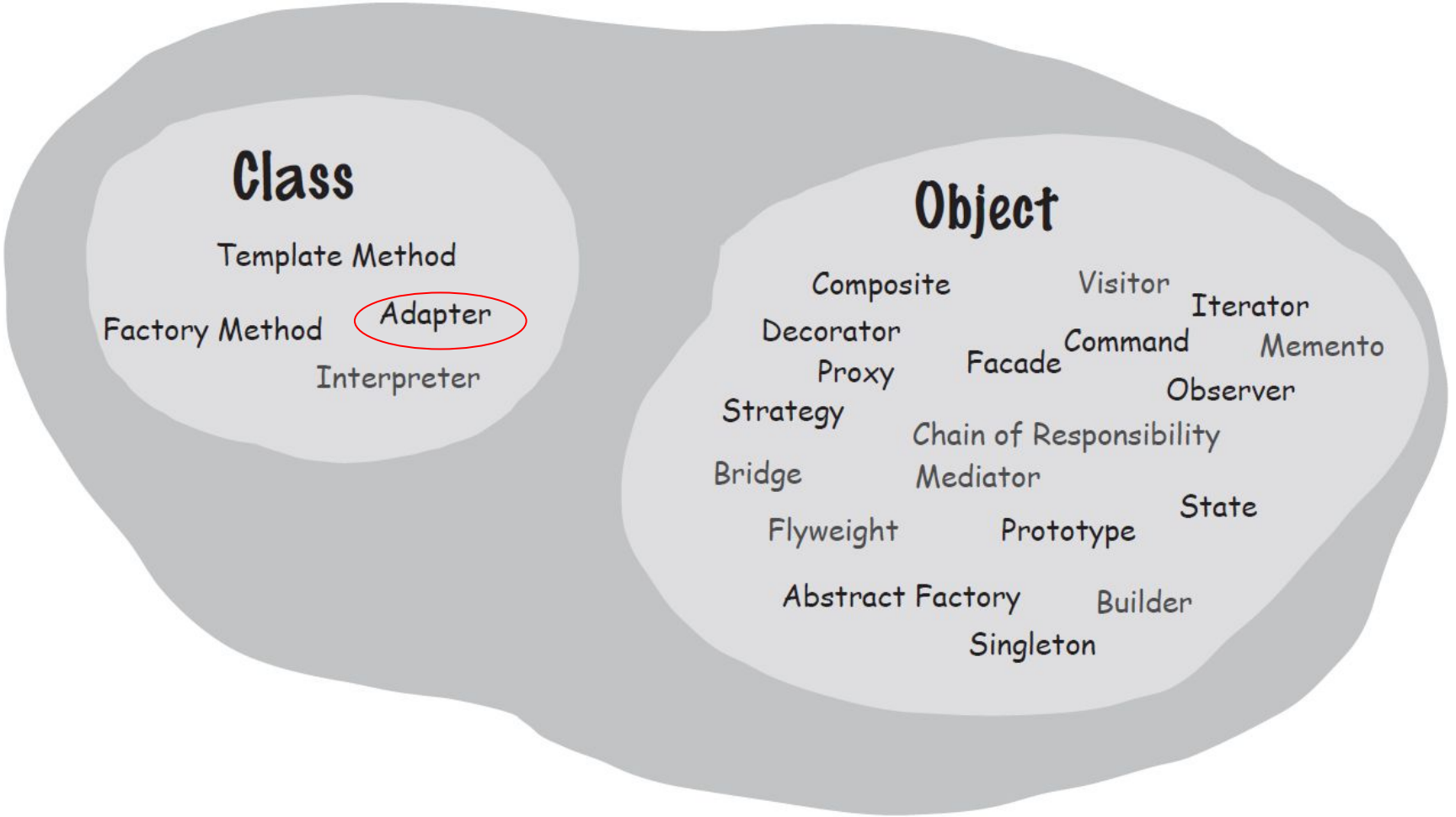
https://www.gofpatterns.com/design-patterns/module2/behavioral-creational-structural.php


Abstract Factory
Composite
Observer
Strategy
Decorator
State
Adapter
Singleton
Factory Method
Proxy
Template Method
Command
Iterator
Facade
Each of these patterns belongs in one of those categories
Creational
Behavioral
Structural


Creational
Singleton
Builder
Prototype
Abstract Factory
Factory Method
Behavioral
Mediator
Visitor
Template Method
Iterator
Command
Memento
Interpreter
Observer
Chain of Responsibility
State
Strategy
Structural
Proxy
Decorator
Composite
Facade
Flyweight
Bridge
Adapter

Organizing Design Patterns (cont.)

- Patterns are often classified by a second attribute: whether or not the pattern deals with classes or objects.
- **Class Patterns** describe how relationships between classes are defined via inheritance. Relationships in class patterns are established at compile time.
- **Object Patterns** describe relationships between objects and are primarily defined by composition. Relationships in object patterns are typically created at runtime and are more dynamic and flexible.


Class
Template Method
Factory Method
Adapter
Interpreter
Object
Composite
Visitor
Iterator
Decorator
Command
Memento
Proxy
Facade
Observer
Strategy
Chain of Responsibility
Bridge
Mediator
State
Flyweight
Prototype
Abstract Factory
Builder
Singleton

Thinking in Patterns

- Contexts, constraints, forces, catalogs, classifications... this is starting to sound mighty academic.
- Okay, all that stuff is important and knowledge is power. But, let's face it, if you understand the academic stuff and don't have the experience and practice using patterns, then it's not going to make much difference in your life.
- Here's a quick guide to help you start to think in patterns. What do we mean by that? We mean being able to look at a design and see where patterns naturally fit and where they don't.

Thinking in Patterns (cont.)

- **Keep it simple (KISS)**
  - First of all, when you design, solve things in the simplest way possible.
  - Your goal should be simplicity, not “how can I apply a pattern to this problem?”
  - Don’t feel like you aren’t a sophisticated developer if you don’t use a pattern to solve a problem. Other developers will appreciate and admire the simplicity of your design.
  - That said, sometimes the best way to keep your design simple and flexible is to use a pattern.

Thinking in Patterns (cont.)

- **Design Patterns aren’t a magic bullet; in fact, they’re not even a bullet!**
  - Patterns are general solutions to recurring problems.
  - Patterns also have the benefit of being well tested by lots of developers.
  - So, when you see a need for one, you can sleep well knowing many developers have been there before and solved the problem using similar techniques.
  - However, patterns aren’t a magic bullet. You can’t plug one in, compile and then take an early lunch.
  - To use patterns, you also need to think through the consequences for the rest of your design.

Thinking in Patterns (cont.)

- **You know you need a pattern when…**
  - The most important question: when do you use a pattern? As you approach your design, introduce a pattern when you're sure it addresses a problem in your design. If a simpler solution might work, give that consideration before you commit to using a pattern.
  - Knowing when a pattern applies is where your experience and knowledge come in.
  - Once you're sure a simple solution will not meet your needs, you should consider the problem along with the set of constraints under which the solution will need to operate—these will help you match your problem to a pattern.

Thinking in Patterns (cont.)

- **You know you need a pattern when... (cont.)**
  - If you've got a good knowledge of patterns, you may know of a pattern that is a good match.
  - Otherwise, survey patterns that look like they might solve the problem.
  - The intent and applicability sections of the patterns catalogs are particularly useful for this.
  - Once you've found a pattern that appears to be a good match, make sure it has a set of consequences you can live with and study its effect on the rest of your design. If everything looks good, go for it!

Thinking in Patterns (cont.)

- **You know you need a pattern when... (cont.)**
  - There is one situation in which you'll want to use a pattern even if a simpler solution would work: when you expect aspects of your system to vary.
  - As we've seen, identifying areas of change in your design is usually a good sign that a pattern is needed.
  - Just make sure you are adding patterns to deal with practical change that is likely to happen, not hypothetical change that may happen.

Reminder

- The first of many design principles:
  - Design Principle: **Identify the aspects of your application that vary and separate them from what stays the same.**
  - What do you understand from this design principle?
  - Why do we apply this principle?

Reminder (cont.)

- If you've got some aspect of your code that is changing, say with every new requirement, then you know you've got a behavior that needs to be pulled out and separated from all the stuff that doesn't change.
- Here's another way to think about this principle: *take the parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.*
- **Take what varies and "encapsulate" it so it won't affect the rest of your code.**
- **The result? Fewer unintended consequences from code changes and more flexibility in your systems.**
- As simple as this concept is, it forms the basis for almost every design pattern.
- All patterns provide a way to let some part of a system vary independently of all other parts.

Thinking in Patterns (cont.)

- **Refactoring time is Patterns time!**
  - Refactoring is the process of making changes to your code to improve the way it is organized.
  - The goal is to improve its structure, not change its behavior. This is a great time to reexamine your design to see if it might be better structured with patterns.
  - For instance, code that is full of conditional statements might signal the need for the State Pattern.
  - Or, it may be time to clean up concrete dependencies with a Factory.
  - Entire books have been written on the topic of refactoring with patterns, and as your skills grow, you'll want to study this area more.

Thinking in Patterns (cont.)

- **Take out what you don’t really need. Don’t be afraid to remove a Design Pattern from your design.**
  - When do you remove a pattern? When your system has become complex and the flexibility you planned for isn’t needed.
  - In other words, when a simpler solution without the pattern would be better.

Thinking in Patterns (cont.)

- **If you don’t need it now, don’t do it now.**
  - Design Patterns are powerful, and it’s easy to see all kinds of ways they can be used in your current designs.
  - Developers naturally love to create beautiful architectures that are ready to take on change from any direction.
  - Resist the temptation. If you have a practical need to support change in a design today, go ahead and employ a pattern to handle that change.
  - However, if the reason is only hypothetical, don’t add the pattern; it is only going to add complexity to your system, and you might never need it!

Power of the shared vocabulary

- Don’t underestimate the power of a shared vocabulary, it’s one of the biggest benefits of Design Patterns.
- When your fellow developers know patterns and use a shared vocabulary as well, it leads to better designs, better communication, and, best of all, it’ll save you a lot of time that you can spend on cooler things.

Top five ways to share your vocabulary

- In design meetings
  - Discussing designs from the perspective of Design Patterns and OO principles keeps your team from getting bogged down in implementation details and prevent many misunderstandings.
- With other developers
  - Use patterns in your discussions with other developers. This helps other developers learn about new patterns and builds a community. The best part about sharing what you've learned is that great feeling when someone else "gets it"!
- In architecture documentation
  - When you write architectural documentation, using patterns will reduce the amount of documentation you need to write and gives the reader a clearer picture of the design.

Top five ways to share your vocabulary (cont.)

- In code comments and naming conventions
  - When you're writing code, clearly identify the patterns you're using in comments.
  - Also, choose class and method names that reveal any patterns underneath. Other developers who have to read your code will thank you for allowing them to quickly understand your implementation.
- To groups of interested developers
  - Share your knowledge.
  - Many developers have heard about patterns but don't have a good understanding of what they are.
  - Volunteer to give a brown-bag lunch on patterns or a talk at your local user group.

Thank you

References

Material in this lecture is taken from Freeman, E., Robson, E., Bates, B., & Sierra, K., *Head First Design Patterns: A Brain-Friendly Guide*, O'Reilly Media, Inc., 2014.