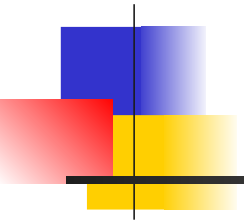


Design Patterns

James Brucker

Reusable Solutions

All engineering disciplines reuse proven good solutions

Civil Engineer

standard designs and construction methods based on experience

Circuit Designer

reuse component designs in integrated circuits

Architect

reuse design patterns in home and building design

Reusable Ideas in Software

Developers **reuse** knowledge, experience, & code

Application Level

reuse the design & code of a similar project

Design Level

apply known design principles and design patterns

Logic Level

apply known *algorithms* to implement behavior

Method Implementation (Coding) Level

use programming idioms for common tasks

A Programming Idiom

Problem: apply a function to every element in a list

Idiom:

1. Need result as a list? Use a list comprehension.

```
result = [f(x) for x in mylist]
```

2. Will result be processed further?
   Many, many values?
   Consider `map` instead.

An Algorithm

Problem:

find the shortest path from node A to node B in a graph

Solution:

apply Dykstra's Shortest Path algorithm

Reusable Code

Requirement:

Sort a List of Persons by last name, ignoring case.

Solution:

Use sorted(*iterable*, key=*function*)

```
people = [Person("Joseph","Biden", ...),
          Person("Prayut","Chan-o-cha", ...),
          Person("Jing Ping", "Xi", ...)
         ]
sorted(people, key=lambda p: p.last_name.lower)
```

Libraries of reusable code

Requirement:

Download data from the Internet in JSON format, and convert it to a dictionary of key-values.

Solution:

Use the `requests` package.

```
"""Get info about a github user."""
import requests

url = f"https://api.github.com/users/{username}"
response = requests.get(url)
# parse the response, same as json.loads(response.text)
data = response.json()
print(data)
```

What is a Design Pattern?

A *situation* that occurs over and over,

along with a *reusable* design of a solution.

Format for Describing a Pattern

Pattern Name: **Iterator**

Context

We need to access elements of a collection or data src.

Motivation (Forces)

We want to access elements of a collection without the need to know the underlying structure of the collection.

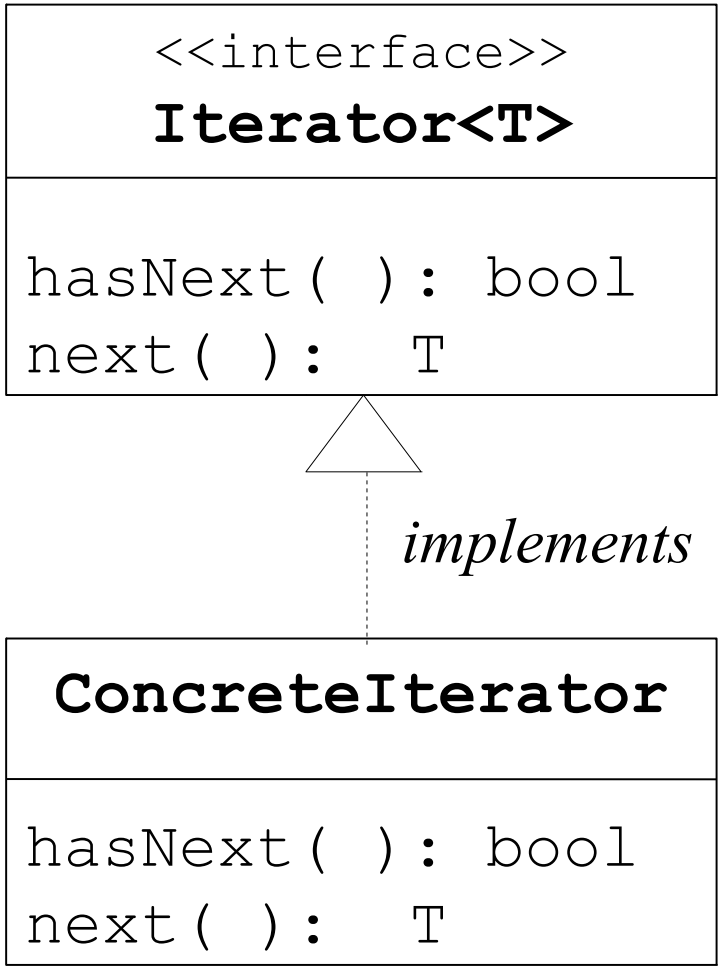
Solution

Each collection provides an iterator with a method to get the next element.

Consequences

Application is not coupled to a kind of collection.
Collection type can be changed w/o changing other code.

UML Diagram for Iterator



Interface

Python does not really have an Interface type.

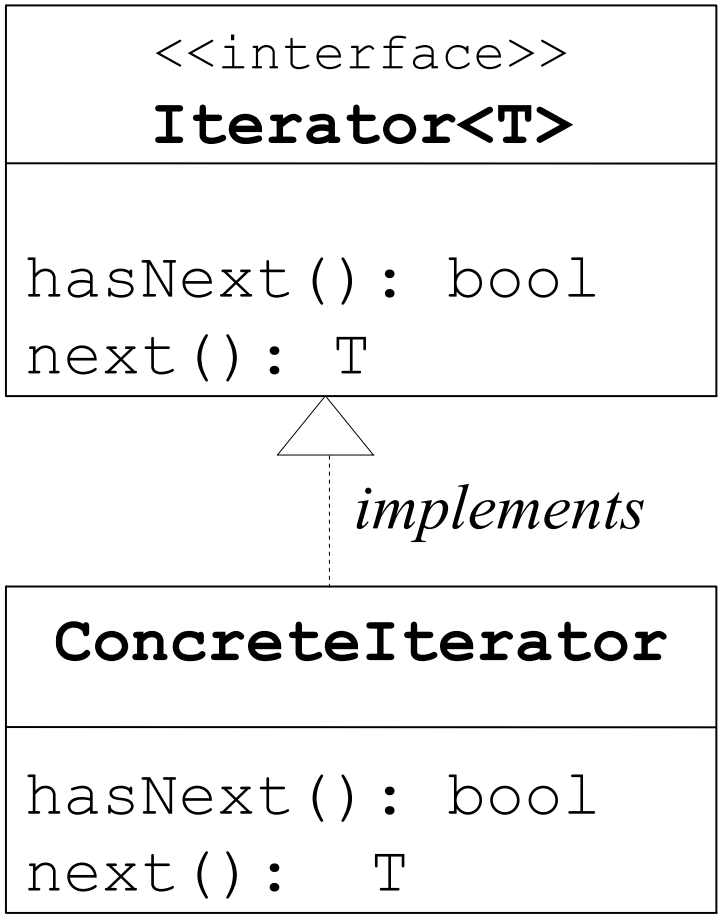
Interface = specify behavior (method signatures) but not an implementation of the behavior.

Example:

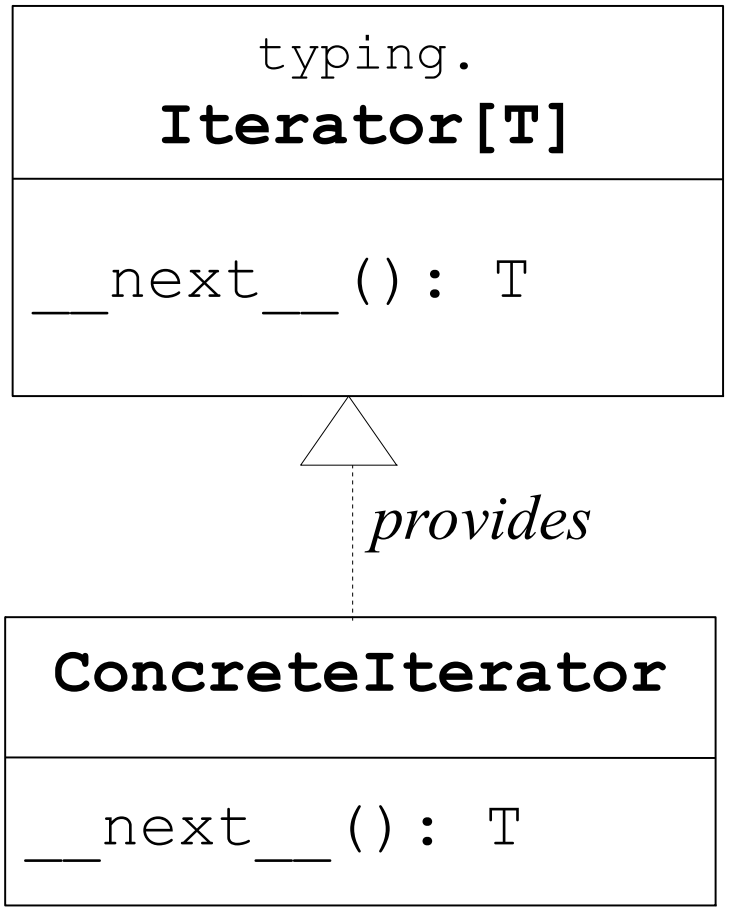
USB Interface - specifies how a USB connection should **behave**.
A manufacturer of USB devices can *implement* the USB interface any way he wants, provided it conforms to the USB specification.

Diagram for Iterator

In the Design Pattern



In Python



Examples of Iterator

What *Iterators* have you used?

- In Python you rarely use iterators directly, but you can.

```
>>> fruit = ["Apple", "Banana", "Durian", ...]
>>> it = iter(fruit)  # creates an iterator

>>> next(it)
'Apple'
>>> next(it)
'Banana'
>>> next(it)
'Durian'
```

Iterator in Python

`collections.abc.Iterator` - abstract base class

`typing.Iterator` - type hint, which accept a parameter such as `Iterator[date]` = an iterator for date objects.

How do you Get an Iterator?

Context:

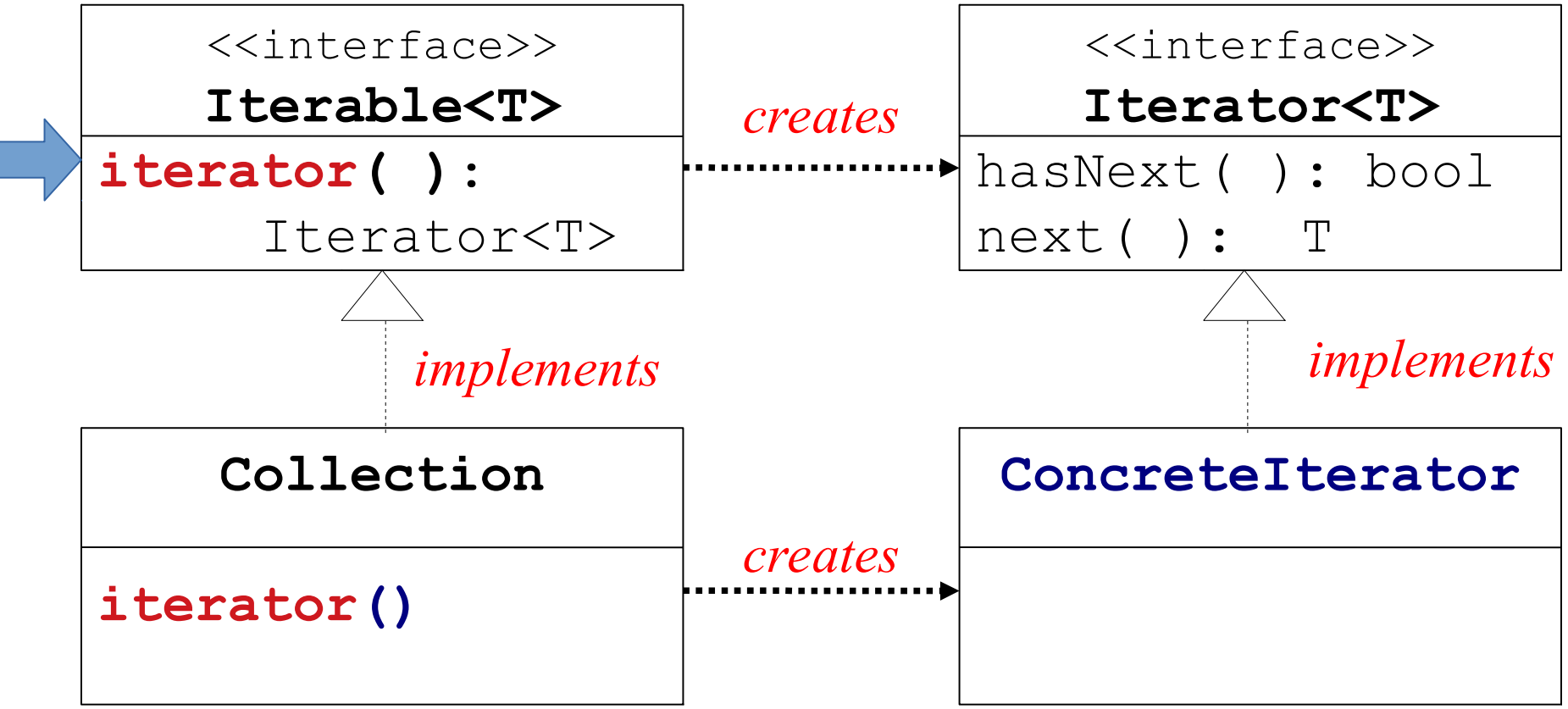
We want to create an Iterator without depending on the API for a particular collection or data source.

Forces:

We don't want the code to be coupled to a particular collection type. We want to always create iterators in the *same way* for any collection.

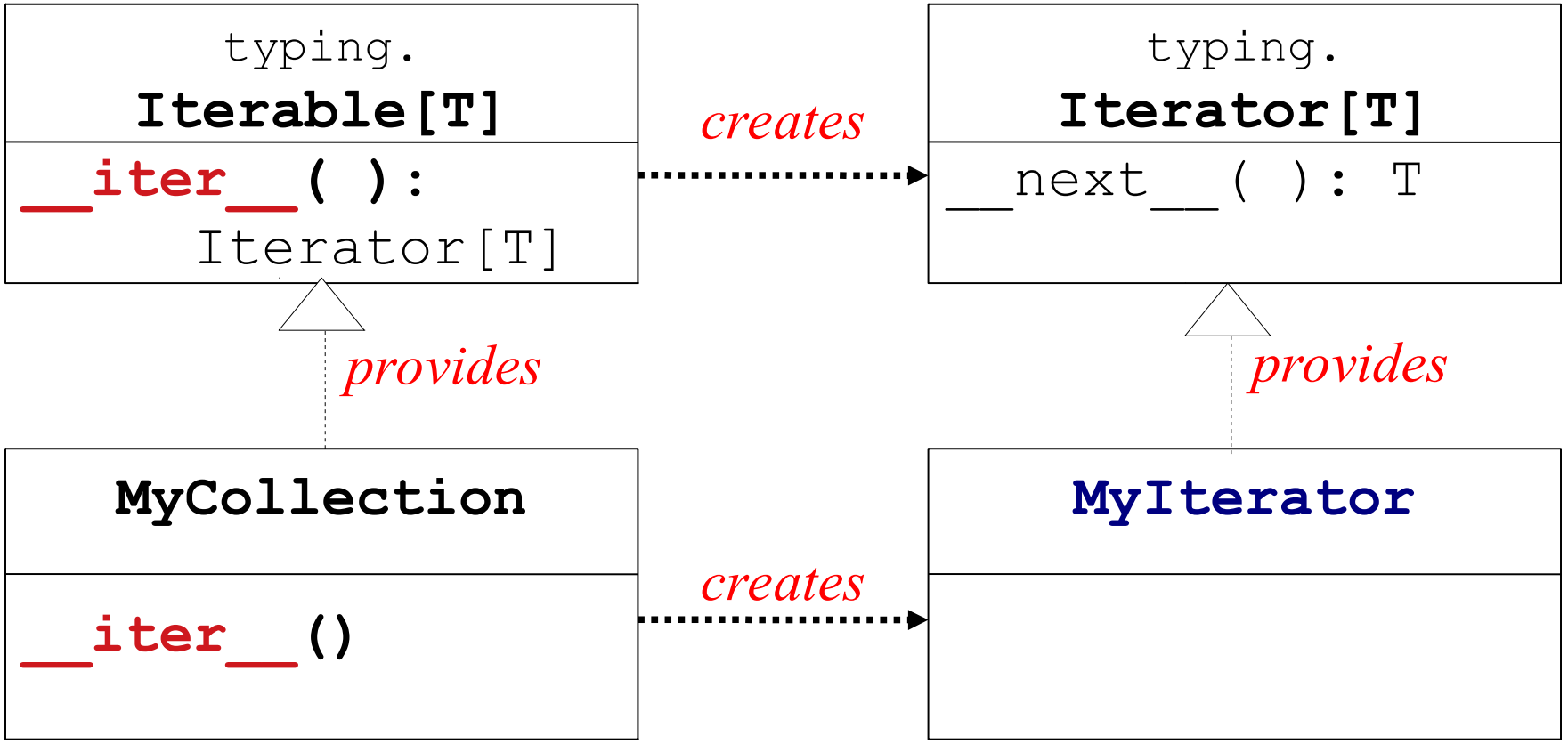
Solution: Define a *Factory Method*

Define a method that creates an Iterator.



Iterable in Python

In Python, an *Iterable* has a `__iter__` method that returns an Iterator.



What *Uses* an Iterable?

Anything that is *Iterable* or *Iterator* can be used as the data source in a "for" loop, list comprehension, or map.

for loop:

```
for x in iterable:
```

list comprehension

```
[f(x) for x in iterable if condition(x)]
```

map function:

```
map( function, iterable)
```

builtin functions:

```
max(iterable), min(iterable),
  sum(iterable), any(iterable), ...
```

What objects are Iterable?

list

set

dict (iterator over keys)

file: file = open("somefile.txt"). Iterator returns lines

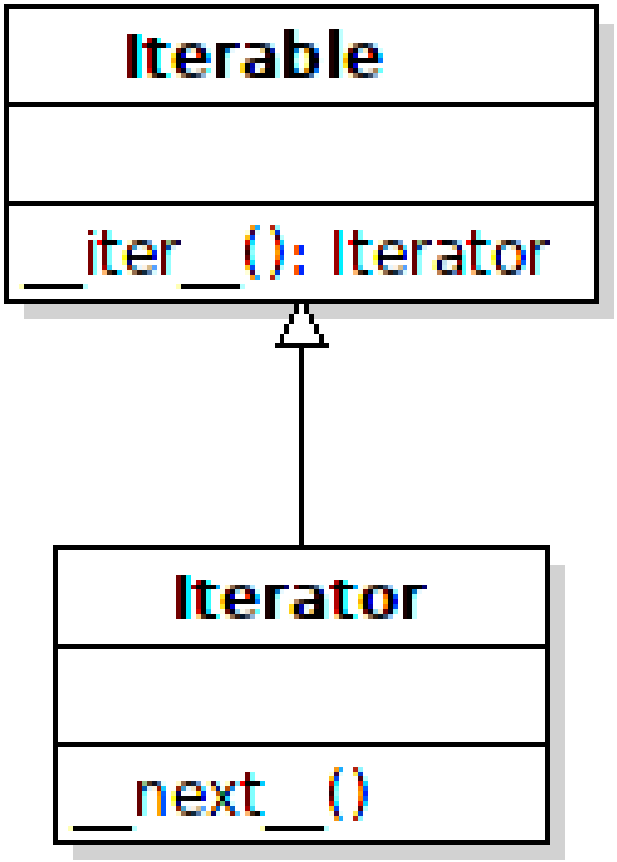
strings, generators

strings are iterable? Really???

```
>>> s = "hello there"
>>> iterator = iter(s)
>>> next(iterator)
'h'
...
```

Python Weirdness

In Python collections.abc, an *Iterable* is a subtype of *Iterator*



Iterators can create new iterators.
Just call `iter(iterator)`

Example

In the Wallet app, we want a way to view (but not modify) what is in the wallet.

Solution:

Provide an `__iter__` method that returns an iterator over Cash in the wallet.

Benefit:

Apps don't need to know the internal structure of the wallet. They use the standard Iterator interface.

Example Code

```
from typing import Iterable  # for type hints

class Wallet(Iterable[Cash]):
    def __init__(self):
        """Initialize an empty wallet."""
        self._items: List[Cash] = []

    def __iter__(self):
        """Return an iterator over items"""
        return iter(self._items)
```

List is *Iterable*, so we use iter(list) to create the iterator.

Using the Wallet Iterator

```
wallet = Wallet()
wallet.deposit(Coin(5,"Baht"), ...)
wallet.deposit(Banknote(100, "USD"), ...)

# show what is in the wallet
for cash in wallet:
    print(cash)
```

Design Patterns on the Web

Many good resources!

https://refactoring.guru/design-patterns

- Examples use pseudo-code (similar to Java)

Game Programming Patterns

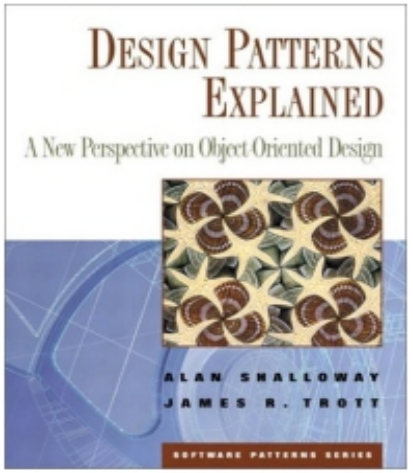
- https://gameprogrammingpatterns.com/contents.html
- Uses C++ for examples.
- I think the explanations are not so good.

Good Design Patterns Books

Good for Java programmers

Design Patterns Explained, 2E (2004)
by Allan Shallow & James Trott

also wrote: *Pattern Oriented Design*.



Head First Design Patterns (2020, 2004)
by Eric & Elizabeth Freeman

Visual & memorable examples,
code is *too simple*.

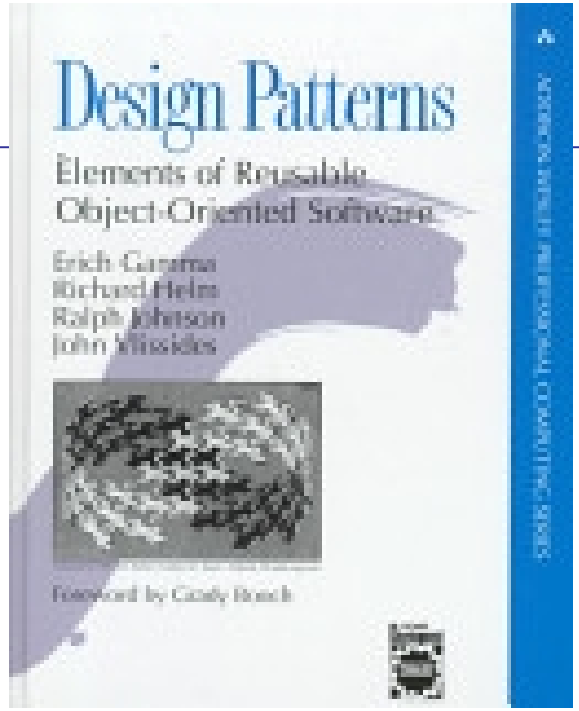
The Classic "Gang of Four" book

The "Gang of Four"

The first book to popularize the idea of software patterns:

Gamma, Helm, Johnson, Vlissides

Design Patterns: Elements of Reusable Object-Oriented Software. (1995)



Structure of Design Patterns in Gang of Four book

Name of Pattern

Intent

what the pattern does.

Motivation

Why this pattern. When to apply this pattern

Structure

Logical structure of the pattern. UML diagrams.

Participants and Collaborators

What are the elements of the pattern? What do they do?

Consequences

The benefits and disadvantages of using the pattern.

Design Patterns To Know

1. Iterator
2. Adapter
3. Factory Method
4. Decorator
5. Singleton
6. Strategy - Layout Manager, used in a Container
7. State
8. Command
9. Observer
10. Facade

SKE Favorite Design Patterns

The SKE12 *Software Spec & Design* class were asked:

"*What patterns are most instructive or most useful?*"

SKE12 Favorite Patterns

Pattern	Votes
MVC	18
State	17
Factory Method	16
Command	15
Strategy	15
Facade	12
Singleton	12
Iterator	11
Observer	11
Adapter	8
Decorator	4
Template Method	3

Categories of Patterns

Creational - how to create objects

Structural - relationships between objects

Behavioral - how to implement some behavior

Situations (Context) not Patterns

Learn the situation and the motivation (forces) that motivate the solution.

Pay attention to Applicability for details of context where the pattern applies.
(Avoid applying the wrong pattern.)

Adding New Behavior

Situation:

we want to add some new behavior to an existing class

Forces:

1. don't want to add more responsibility to the class
2. the behavior may apply to similar classes, too

Example:

Scrollbars

Changing the Interface

Situation:

we want to use a class in an application that requires interface A. But the class doesn't implement A.

Forces:

1. not appropriate to modify the existing class for the new application
2. we may have many classes we need to modify

Example:

change an Enumeration to look like an Iterator

Convenient Implementation

Situation:

some interfaces require implementing a *lot* of methods. But most of the methods aren't usually required.

Forces:

1. how can we make it *easier to implement interface?*
2. how to supply default implementations for methods?

Example:

MouseListener (6 methods), List (24 methods)

A Group of Objects act as One

Situation:

we want to be able to use a Group of objects in an application, and

the application can treat the whole group like a single object.

Forces:

There are many objects that behave similarly. To avoid complex code we'd like to treat as one object.

Example:

KeyPad in a mobile phone app.

Creating Objects without Knowing Type

Situation:

we are using a framework like OCSF.

the framework needs to create objects.

how can we change the type of object that the framework creates?

Forces:

1. want the framework to be extensible.

2. using "new" means coupling between the class and the framework.

Example:

JDBC (Java Database Connection) creates connections for different kinds of databases.

Do Something *Later*

Situation:

we want to run a task at a given time (in the future)

Forces:

we don't want our "task" to be responsible for the schedule of when it gets run.

This situation occurs a lot, so we need a reusable solution.

Example:

We're writing a digital clock. We want an alarm to sound at a specified time.