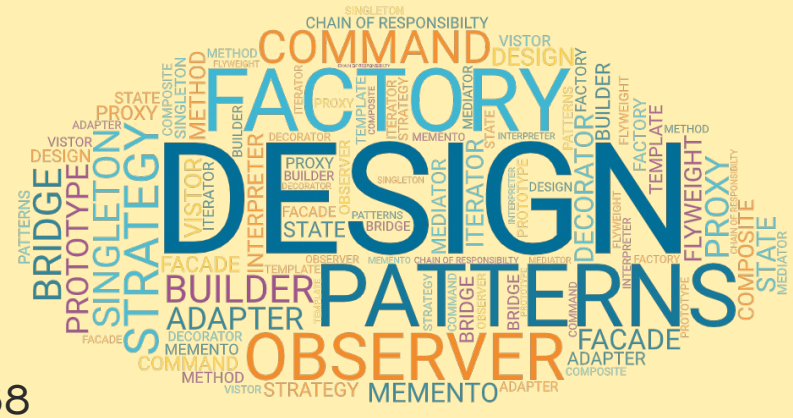
Technische Universität Braunschweig

Suresoft
SUSTAINABLE RESEARCH SOFTWARE

Standing on the shoulders of giants

Hands on Software Design Patterns

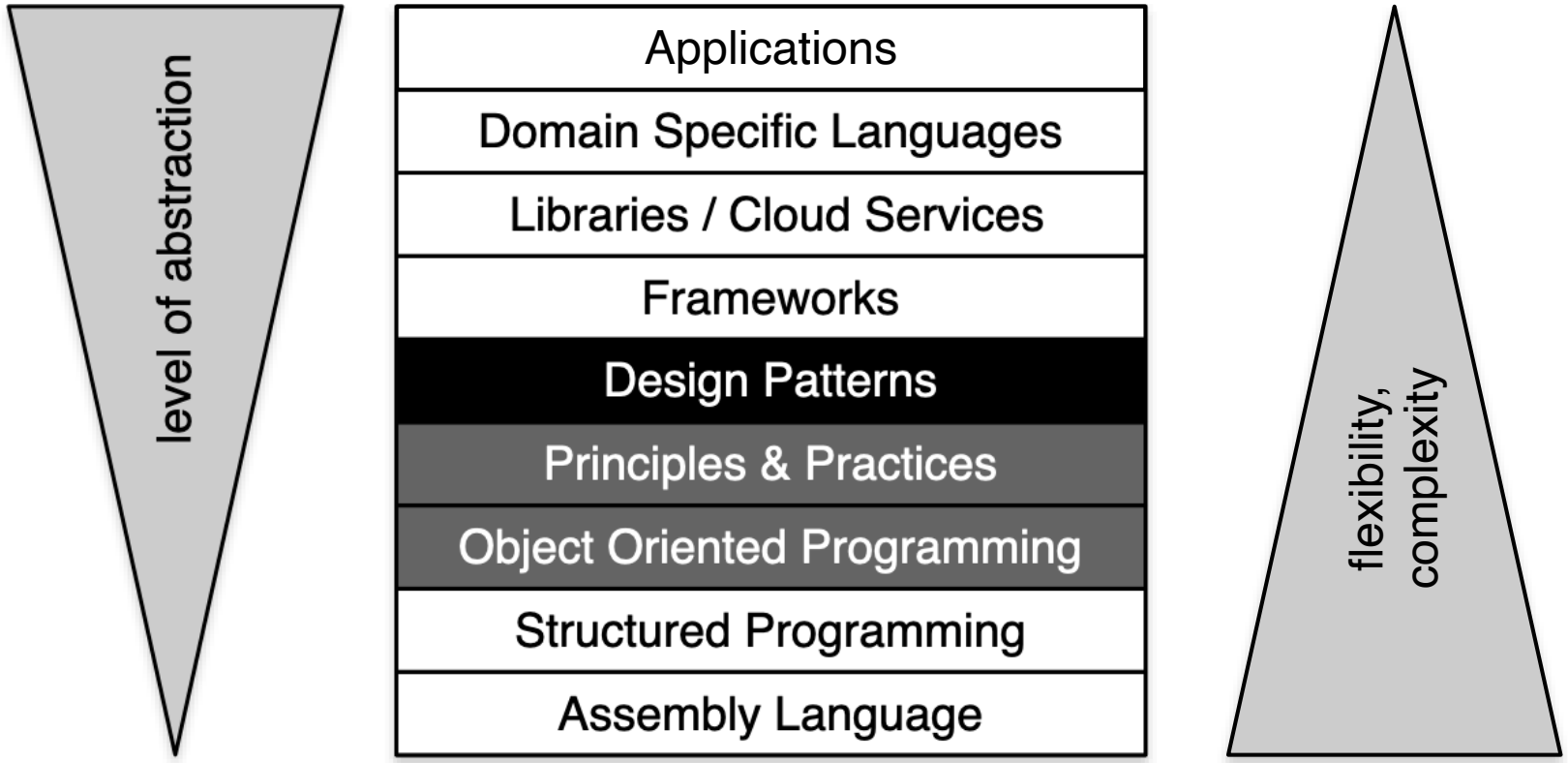


“The dissemination of knowledge is of obvious value —
the massive dissemination of error-loaded software is frightening.”
Edsger W. Dijkstra at the NATO Software Engineering Conference, 1968

Who are we?

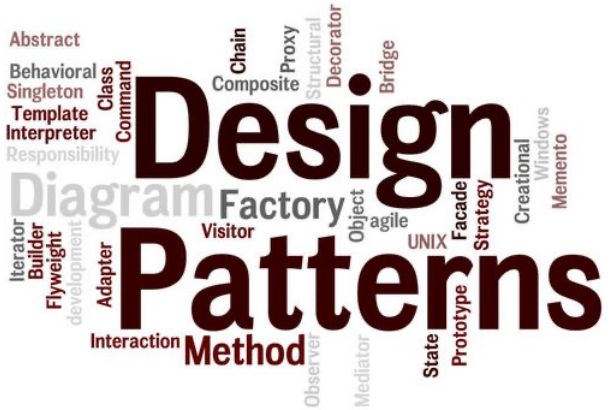
Dr. Jan Linxweiler
j.linxweiler@tu-braunschweig.de

Sören Peters
soe.peters@tu-braunschweig.de


level of abstraction
Applications
Domain Specific Languages
Libraries / Cloud Services
Frameworks
Design Patterns
Principles & Practices
Object Oriented Programming
Structured Programming
Assembly Language
flexibility, complexity

Design Patterns

- Reliable solutions / strategies for repeating design problems in software engineering
  - don’t reinvent the wheel, but instead build upon experiences of experts
- Independent of a specific programming language
- Enables communication on a more abstract level (discussions)
- Enhances understanding of the solutions for other developers (documentation)
- Origin in architecture (Similarity of structures and styles)
  - Christopher Alexander: The Timeless Way of Building, 1979

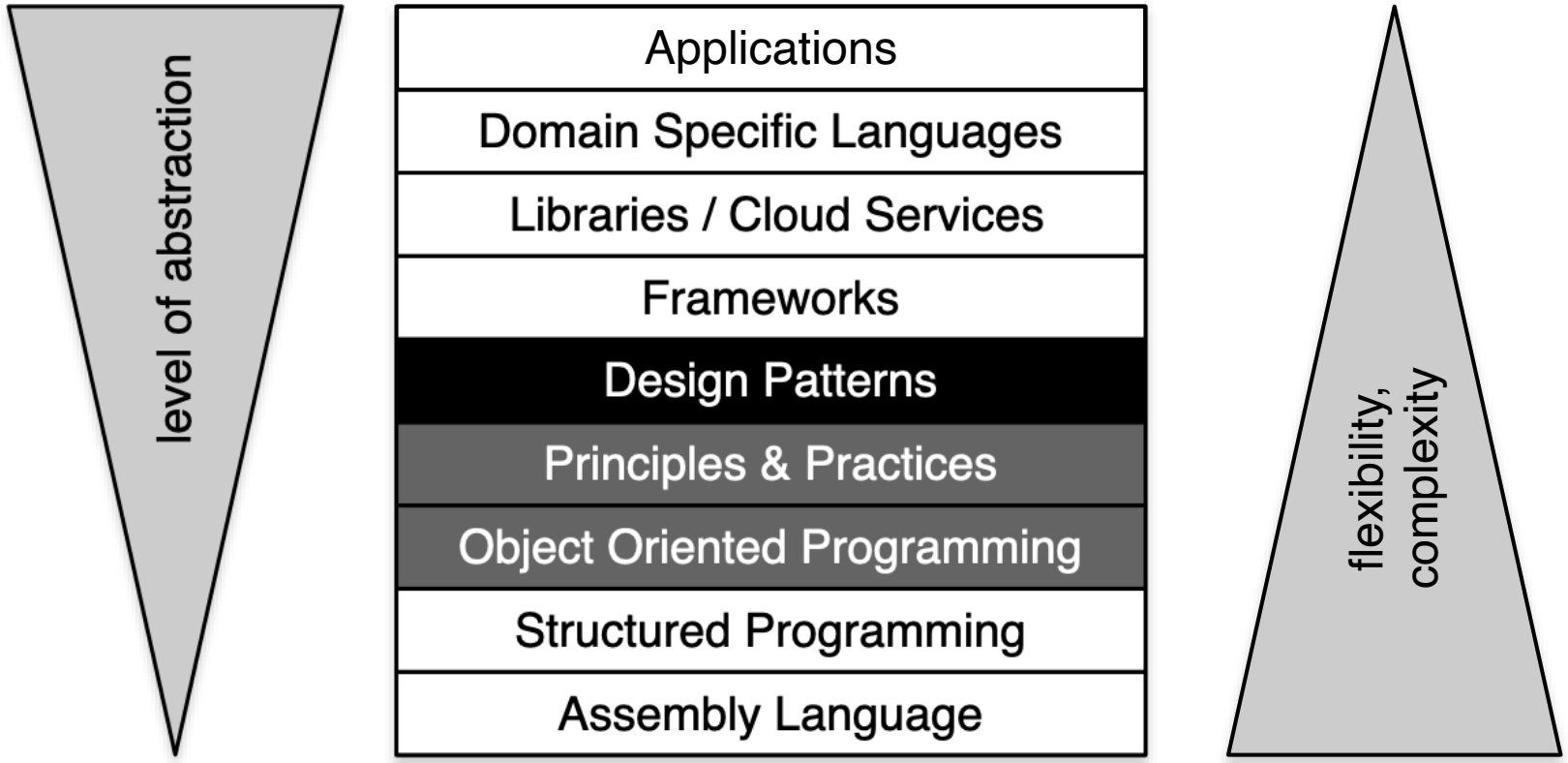


* Erich Gamma, Richard Helm, Ralph Johnson, John Vlissedes, Design Patterns, 1995

How familiar are you with Design Patterns?

a) I never heard of them
b) I’ve heard of them, but never used them
c) I use them from time to time
d) I continuously keep my eyes open to spot them
e) The “Gang of Four” are my personal heros
f) other

Join at
slido.com
#123 444


level of abstraction
Applications
Domain Specific Languages
Libraries / Cloud Services
Frameworks
Design Patterns
Principles & Practices
Object Oriented Programming
Structured Programming
Assembly Language
flexibility, complexity

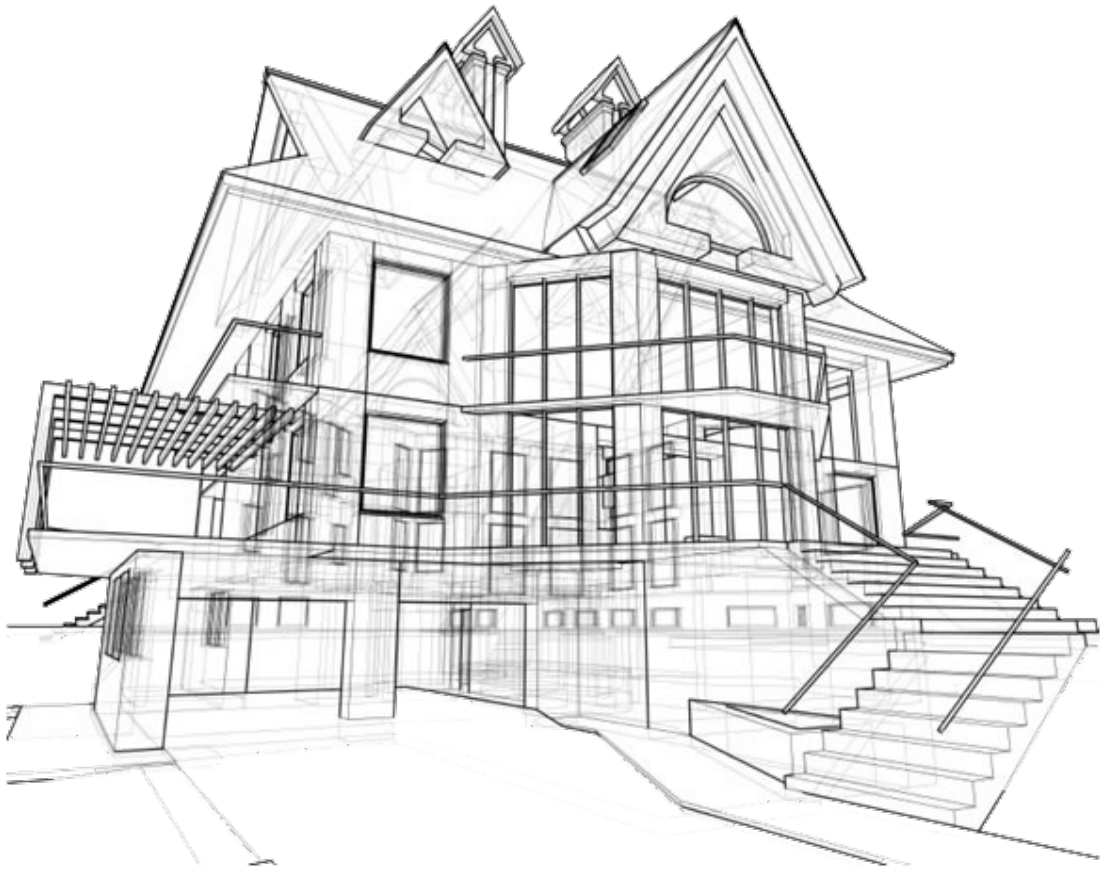
Design Patterns

Origin in architecture (Similarity of structures and styles)*

From architectural patterns to patterns of software design:

1995 GoF: “Design Patterns: Elements of Reusable Object-Oriented Software”**

1. Transferred the idea of patterns to software design.
2. Defined a structure to categorize the patterns.
3. 23 patterns were categorized.
4. Presented object-oriented strategies and approaches based on design patterns.



* Christopher Alexander: The Timeless Way of Building, 1979

** Gang of Four (GoF) Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Design Patterns

- Reliable solutions / strategies for repeating design problems in software engineering
- The description of a design pattern follows certain rules*:
  - description of a **problem**
  - description of a **solution**
    - structure, elements, interactions
  - **discussion**
    - advantages / disadvantages / dependencies
  - source code **examples**
  - related design patterns
- independent of a specific programming language

* Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, 1995

What Design Patterns are not:

- Not a “one-size-fits-all” solution. You need to adapt it to your context.
- No algorithms - algorithms solve problems (search, sort etc.) and are less flexible in their implementation
- Design patterns are no magical cure! When applying a pattern creative talent is still needed
- Design patterns are no frameworks or libraries. Frameworks and libraries provide reusable code. Design patterns only provide templates for solutions.

Why would you care about design patterns?

Tyranny of the Detail: The Carpenters and the drawers

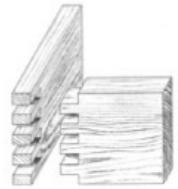
Try to figure out, what they are talking about!!

How do you think should we build the drawers?

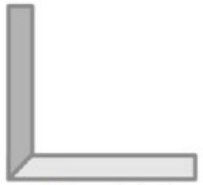
Well, I think we should make the joint by cutting straight down the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight back down, and then …

Details may cause confusion!

Why would you care about design patterns?



Dovetail Joint



Miter Joint

Tyranny of the Detail: The Carpenters and the drawers

Try to figure out, what they are talking about!!

How do you think should we build the drawers?

Should we use a *dovetail joint* or a *miter joint*?

They are discussing differences in the quality of solutions to the problem. And they avoid getting bogged down in the details of a particular solution.

Their discussion is at a higher level, a more abstract level!

Benefits of Design Patterns

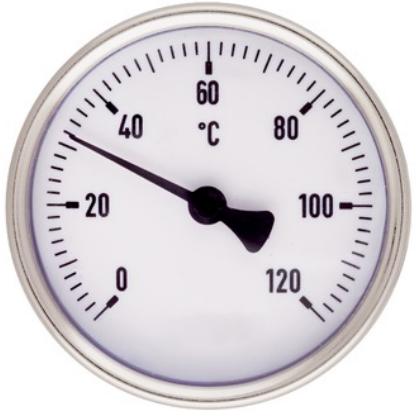
Developer	• **help with design decisions** • use of **proven solutions** of experienced developers • code examples can **make it easier to get started** • help in passing on and holding on to your own **knowledge**
Team	• forming a **standardized language** • standardized **documentation** • **knowledge transfer** between colleagues
Community / Organisation	• standardized **documentation** of knowledge • **reuse** of proven solutions • **common structure** of systems

Classification of design patterns

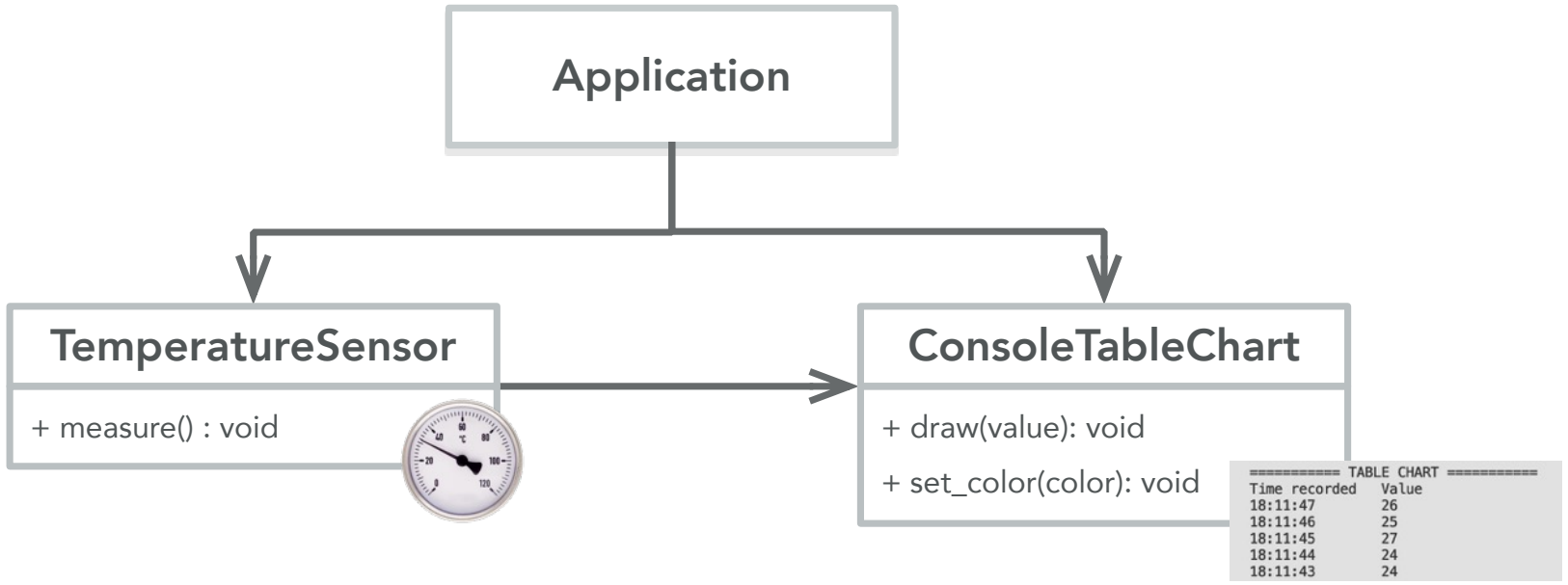
Categories	Task
Structural patterns	Composition of classes and objects
Creational patterns	Encapsulation of the creation process of objects
Behavioral patterns	Manage control flows and interactions between classes and objects

A weather station story

Our company creates a popular weather station software. It controls a temperature sensor that measures the temperature every second. Whenever the temperature is measured we want to update a temperature display.



Starting point



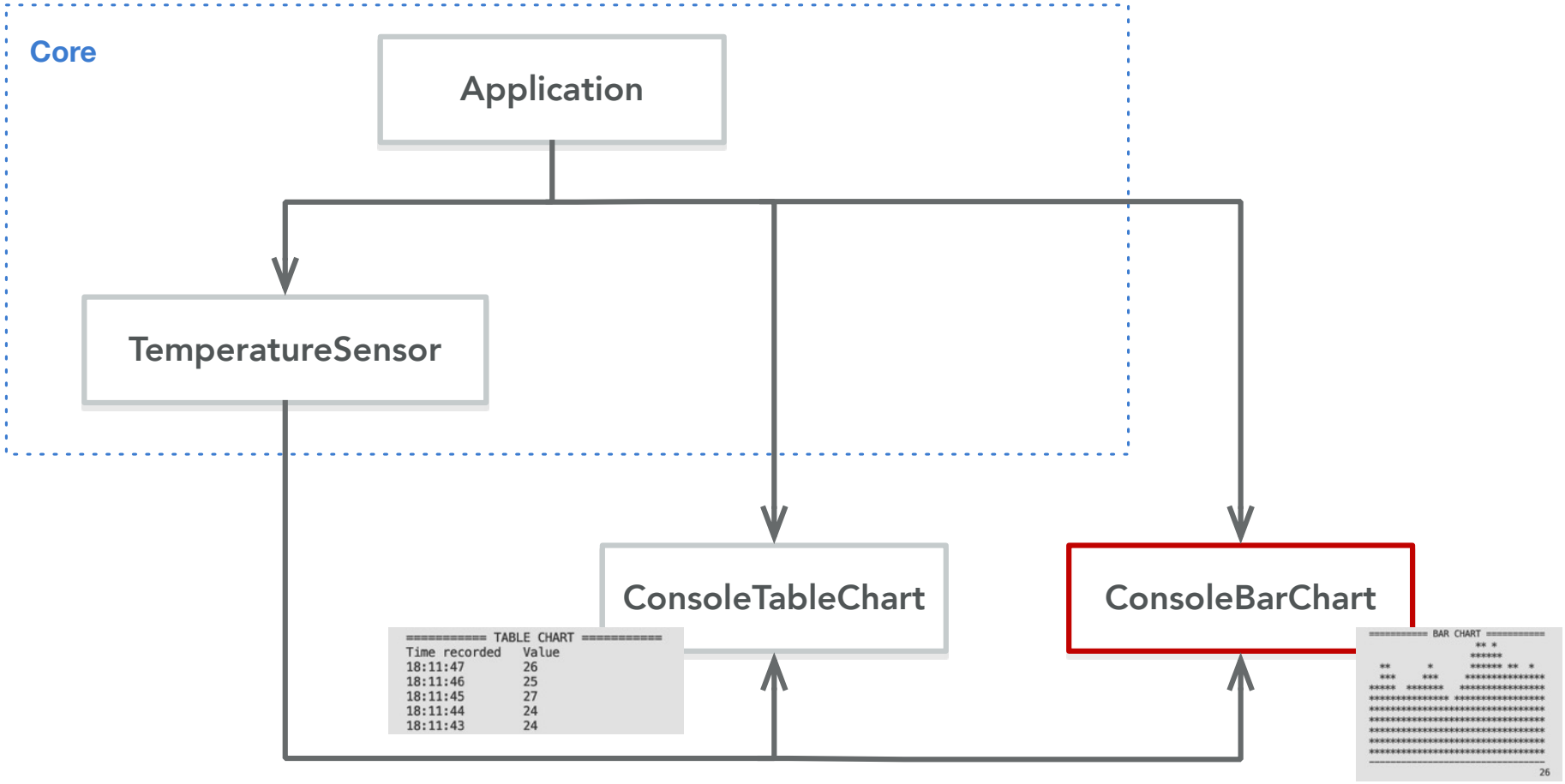
*The **Application** class contains an instance of the **TemperatureSensor** and a **ConsoleTableChart**. The sensor knows the chart and will call the chart's **draw()** method whenever it measures a new value.*

Stage 1: A second chart

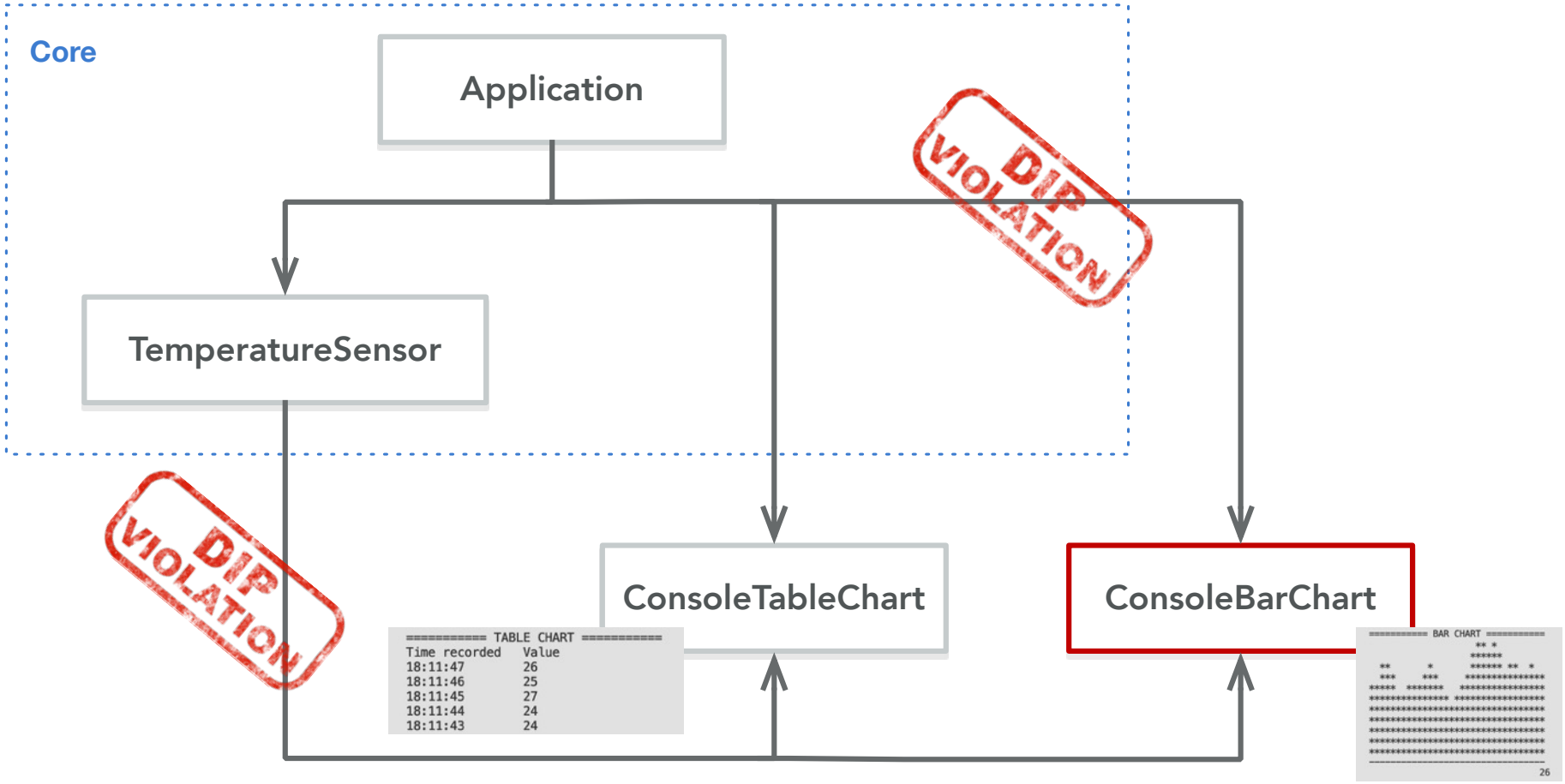
Our glorious overlords (managers) have graced us with a new feature request to keep up with the competition: ***adding a bar chart*** *to the application.*

The problem
In the current state of the application the sensor can only deal with one specific type of chart (ConsoleTableChart).

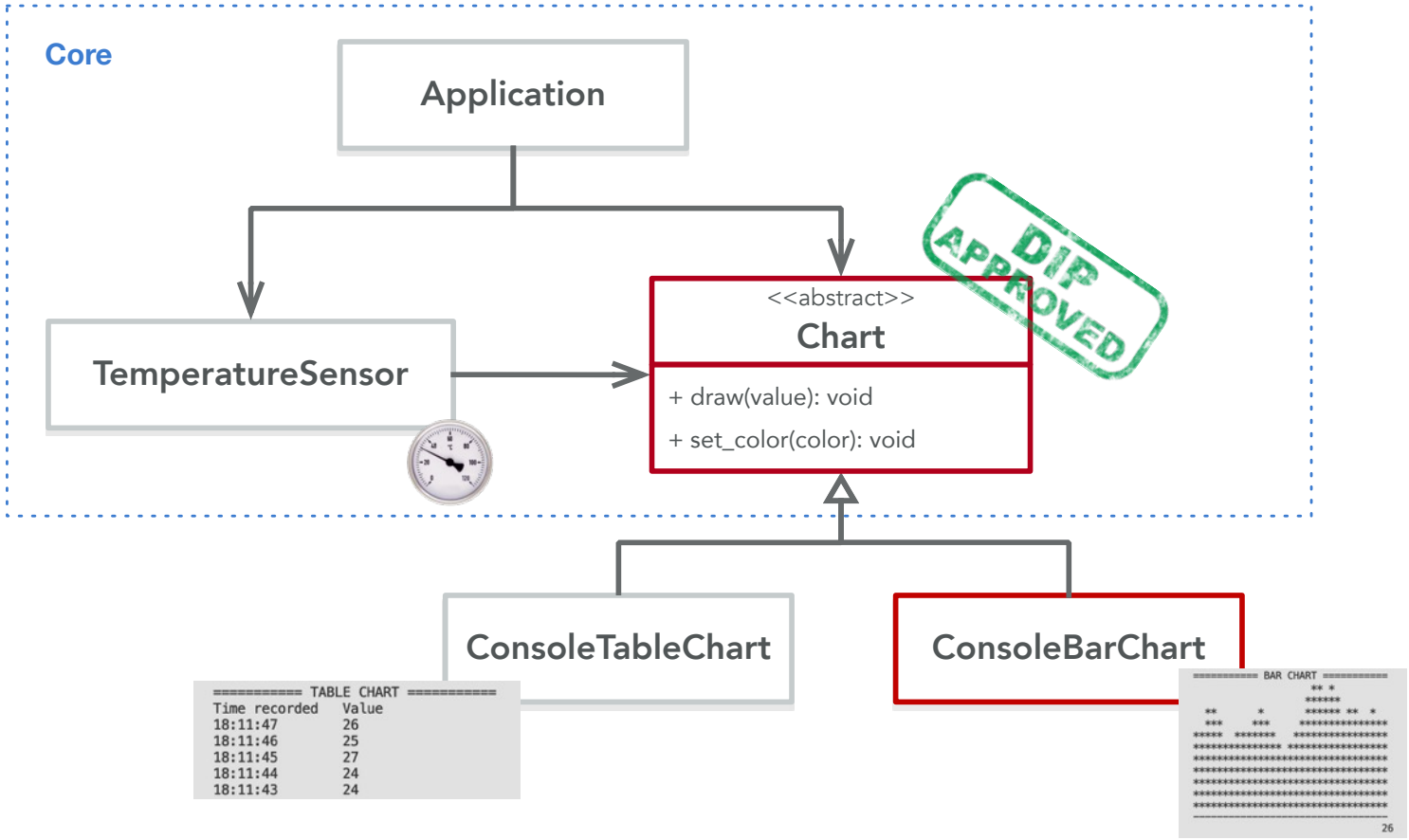
A first solution



A first solution



The solution



*After implementing the **ConsoleBarChart** we extract a common **Chart** interface with an abstract **draw()** method that both Charts must implement.*
*Now the sensor does not depend on a concrete chart type, but only the Chart interface, therefore adhering to the **Dependency Inversion Principle (DIP)**.*

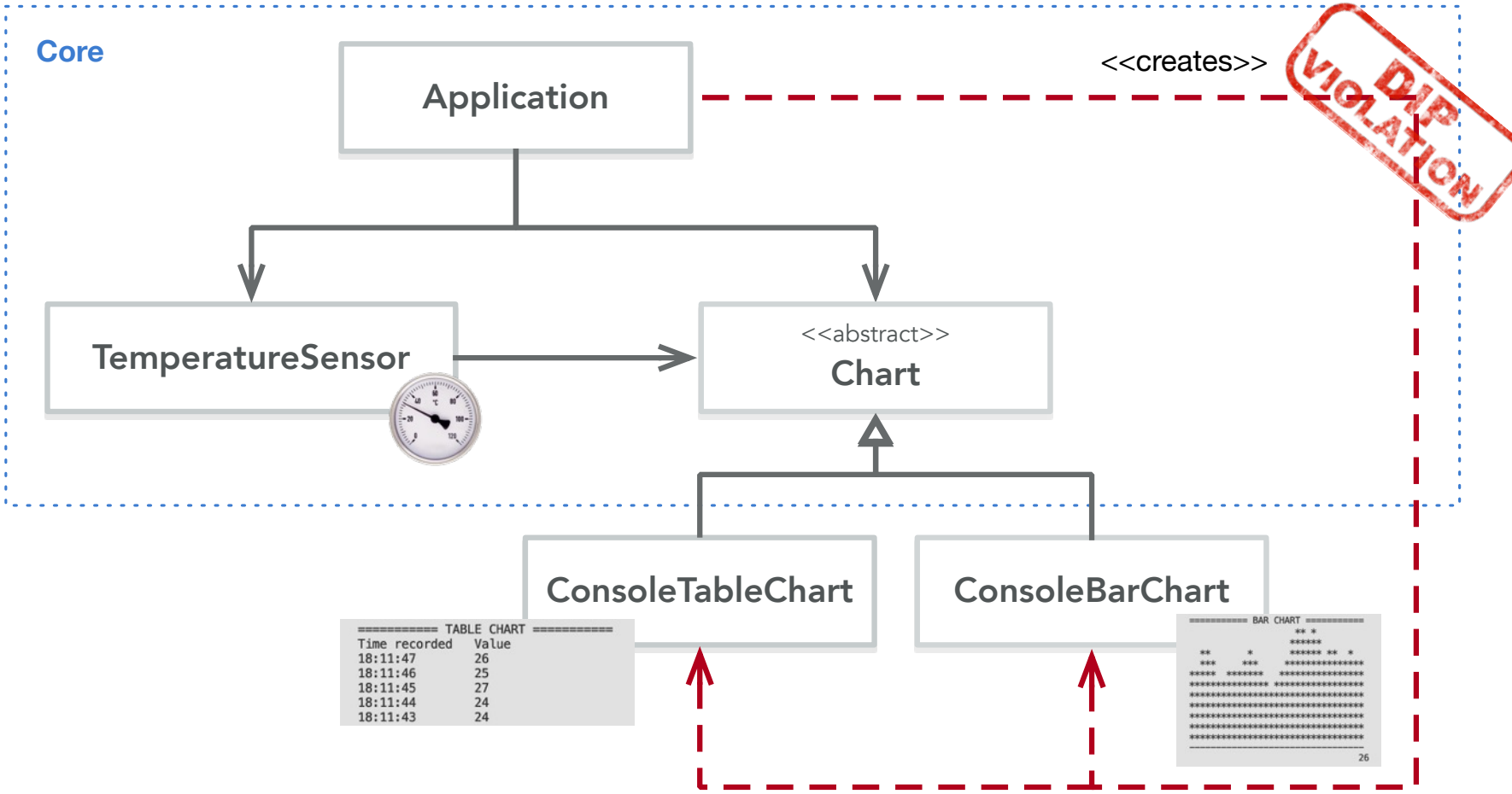
Stage 2: Mass producing charts

*During the next code review session we show our implementation to our colleagues. They point out that in the **Application** class we still depend on the concrete chart types **ConsoleTableChart** and **ConsoleBarChart** that change frequently. Also there will be new chart variants in the future so they ask us to improve our solution.*

*While we are at it we can also implement a **feature request from our clients for a menu** that allows the users to choose a chart from a list of all available chart types.*

The problem

*Our **Application** creates instances of the concrete chart types to pass them on to our sensor.*

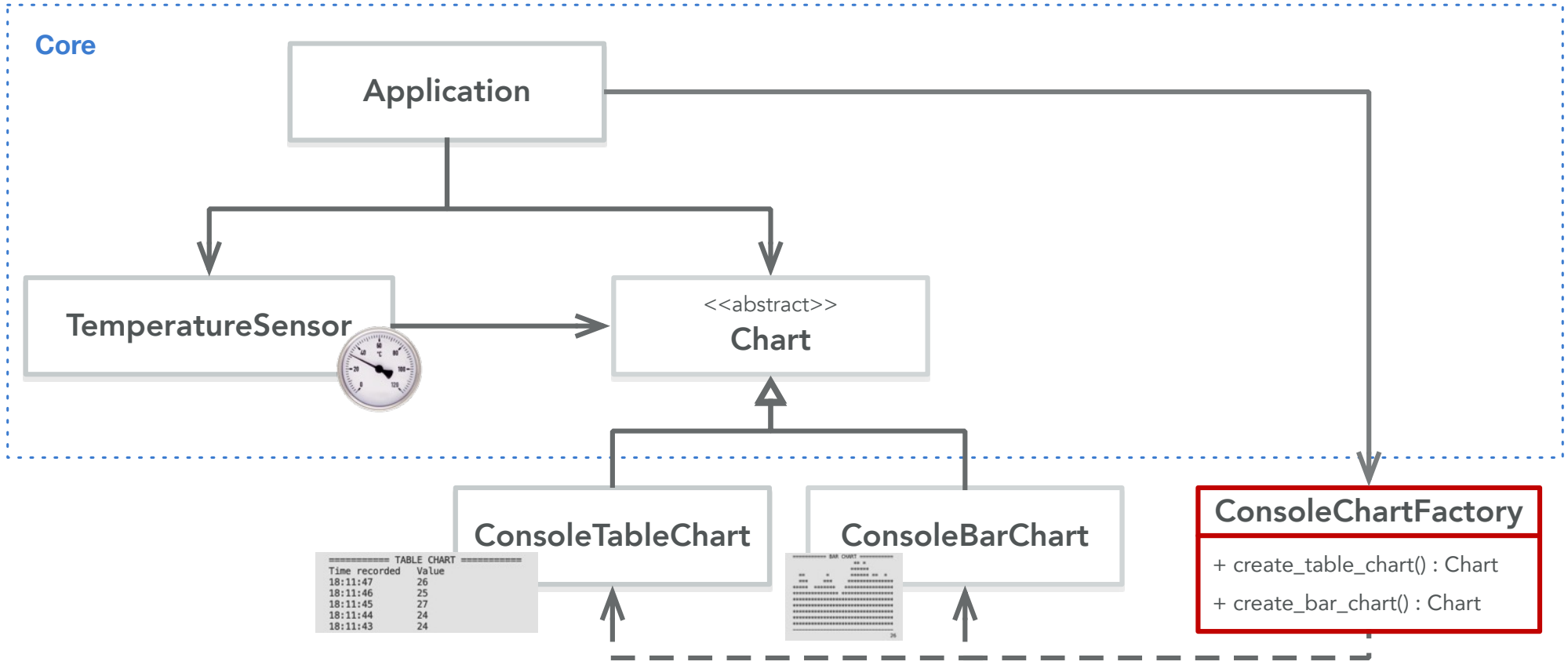


The solution

*We've heard about the **Abstract Factory Pattern** that encapsulates the creation of objects and exposes them only through their common interface afterwards.*
*Therefore, we decide to implement an abstract **ChartFactory** that defines abstract **create_table_chart**() and **create_bar_chart**() methods that both return objects of the abstract type **Chart**.*
*The **ConsoleChartFactory** implements this interface and creates and returns **ConsoleTableChart** and **ConsoleBarChart** instances.*

*We inject this factory into the constructor of the **Application** class leading to an application core that does not depend on any concrete chart types anymore.*

Towards the solution…



Stage 3: Reduce coupling

*Our boss informs us that there is still a weakness in our design. Since we will add more charts in the future, we would have to **change the interface of the factory each time**. Since interfaces should be stable we need to come up with a better solution.*

The problem

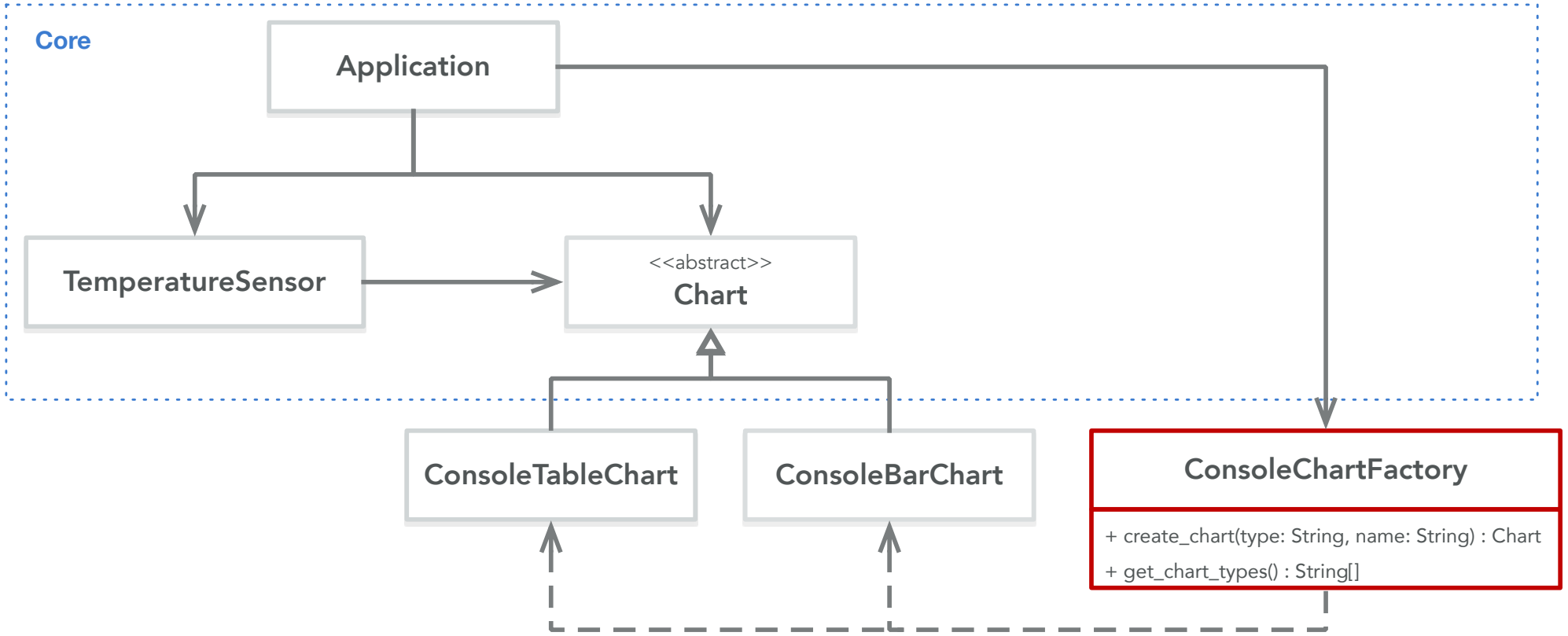
*We notice that with our current implementation of the **ChartFactory** the **Application** class still has a dependency to the specific method name of the specific chart we want to create. This means we have to add a method call for every new chart type in our **Application** class.* This *violates the **Open Closed Principle (OCP)**.*

```
chart = chart_factory.create_table_chart()
self.sensor.add_chart(chart)

chart = chart_factory.create_bar_chart()
self.sensor.add_chart(chart)

# . . .  more to come??
```

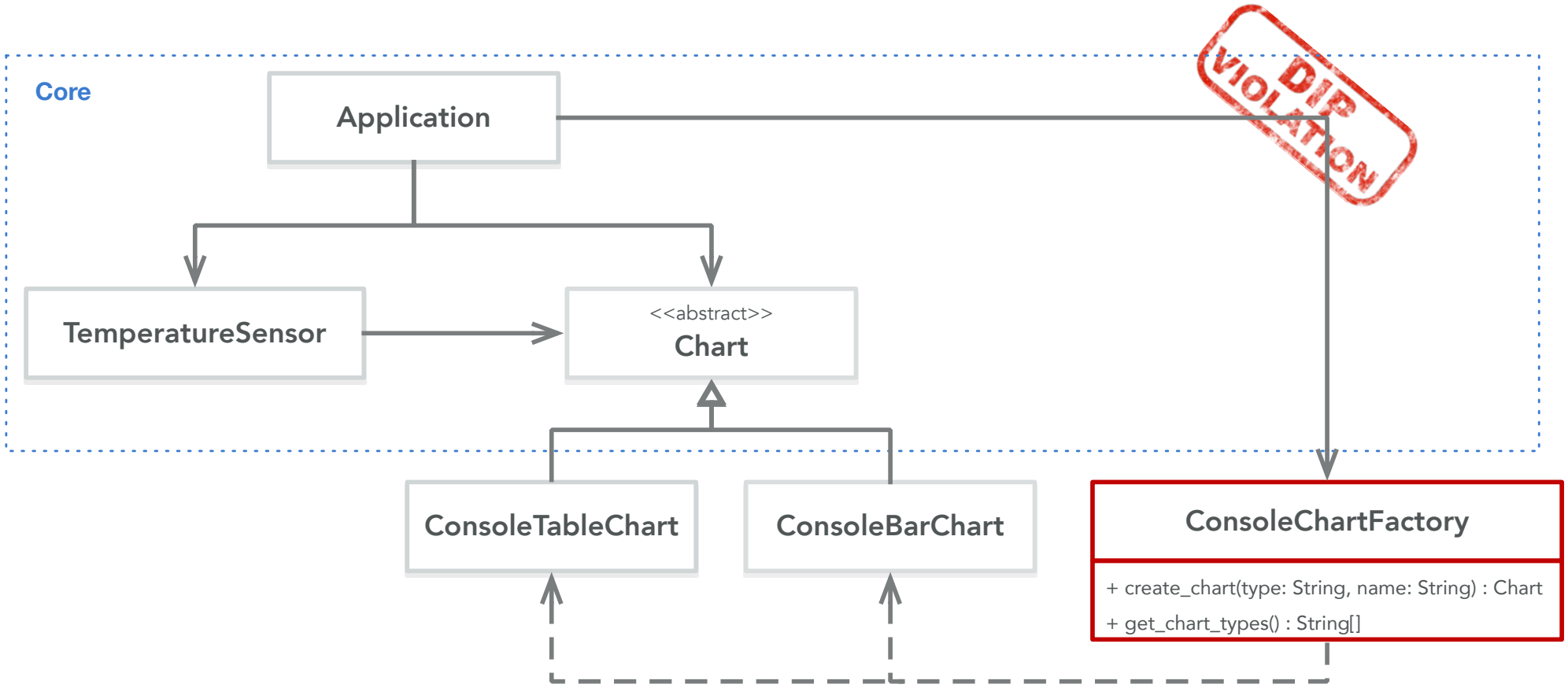
Towards the solution …



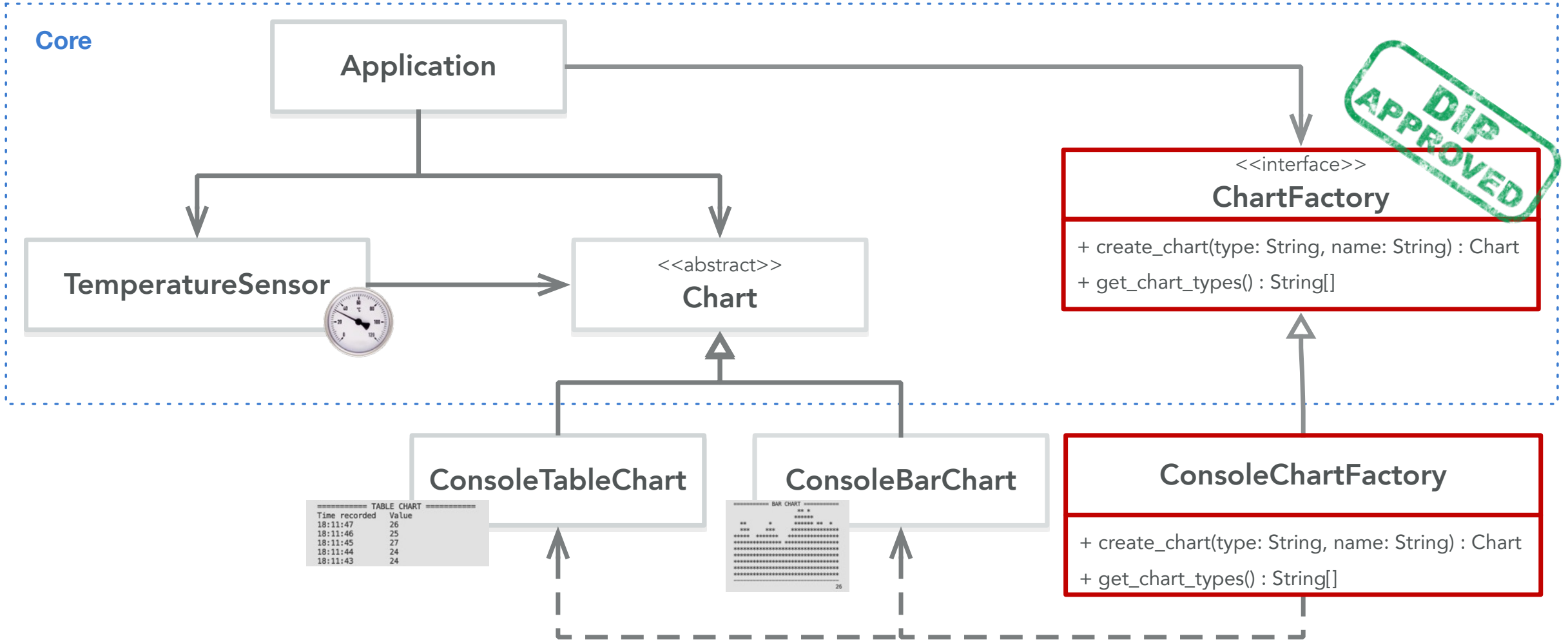
The solution

*We change our **ChartFactory** interface to contain a single **create_chart(chart_type: str)** method that accepts the chart type as a string. By doing this we loose some safety because the methods accepts any string valid or not. But, we also add another method **get_chart_choices()** that returns a list of strings with all possible chart types.*
In our Application class we can now display the possible choices in our selection menu and select a chart type without changing anything in that class.

Towards the solution …



The solution



Question

Which of the following lines in the core of your application are a violation of the Dependency-Inversion Principle?

a) `names: List[str] = ["Lucy", "Paul"]`

b) `chart = ConsoleTableChart()`

c) both

d) none

Hint: *The ConsoleTableChart class changes frequently*

Factory - Question

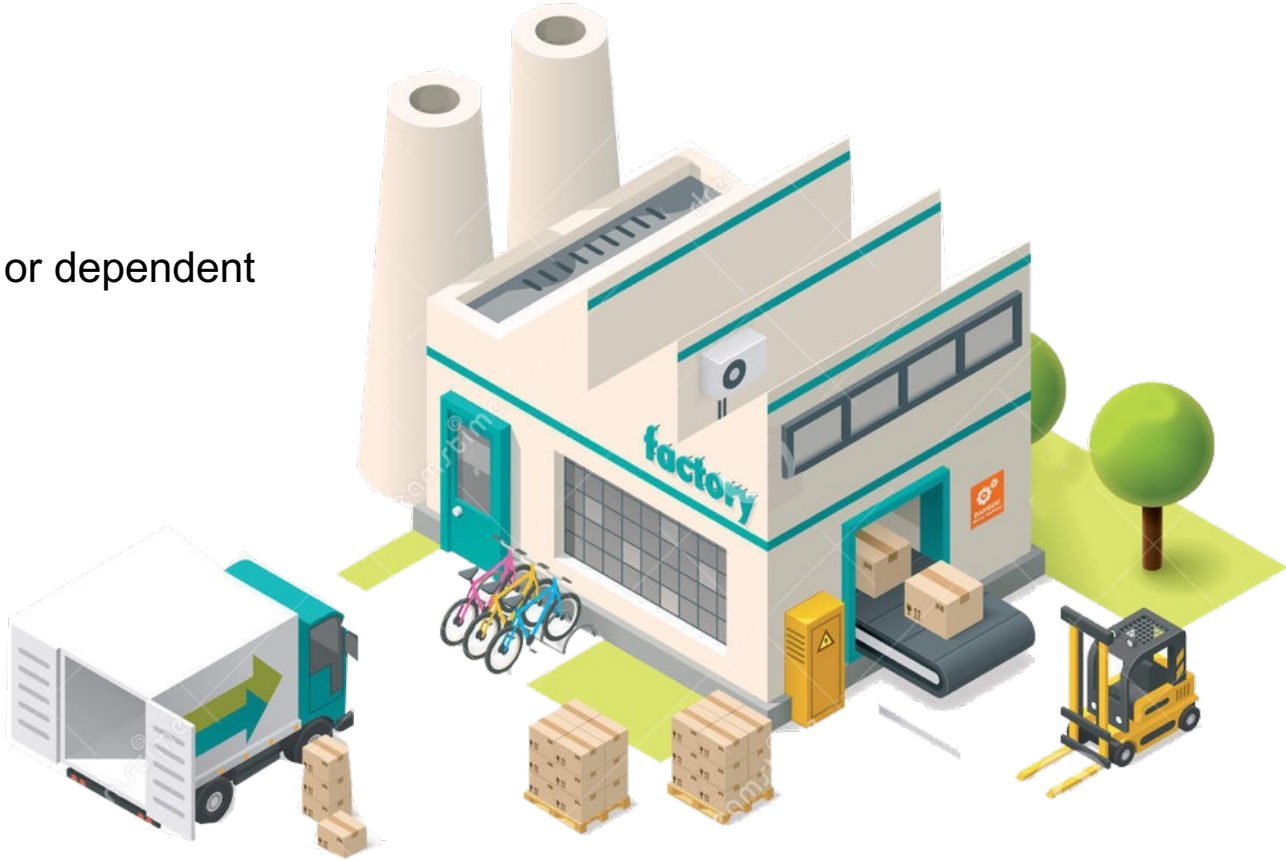
Is the following source code violating the Dependency-Inversion Principle?

```
names: List[str] = ["Lucy", "Paul"]
```

No, it's not violating DIP…
The `List` class is very unlikely to change (not volatile)?
Depending on the concrete `List` class is therefor safe.

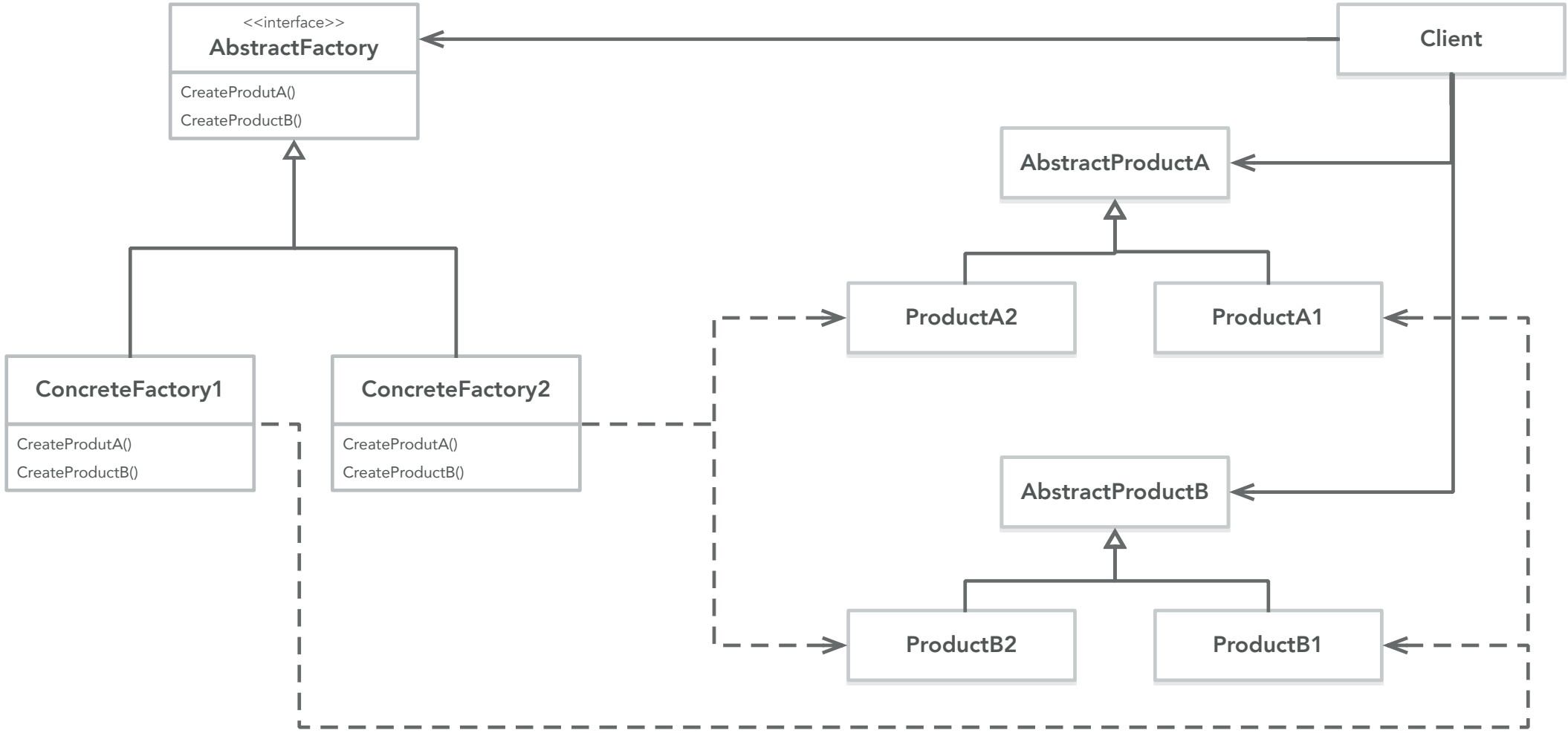
Abstract Factory Pattern

- **Object Creational pattern**
- “Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”



* Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, 1995

Abstract Factory Pattern



Abstract Factory Pattern

Benefits and liabilities

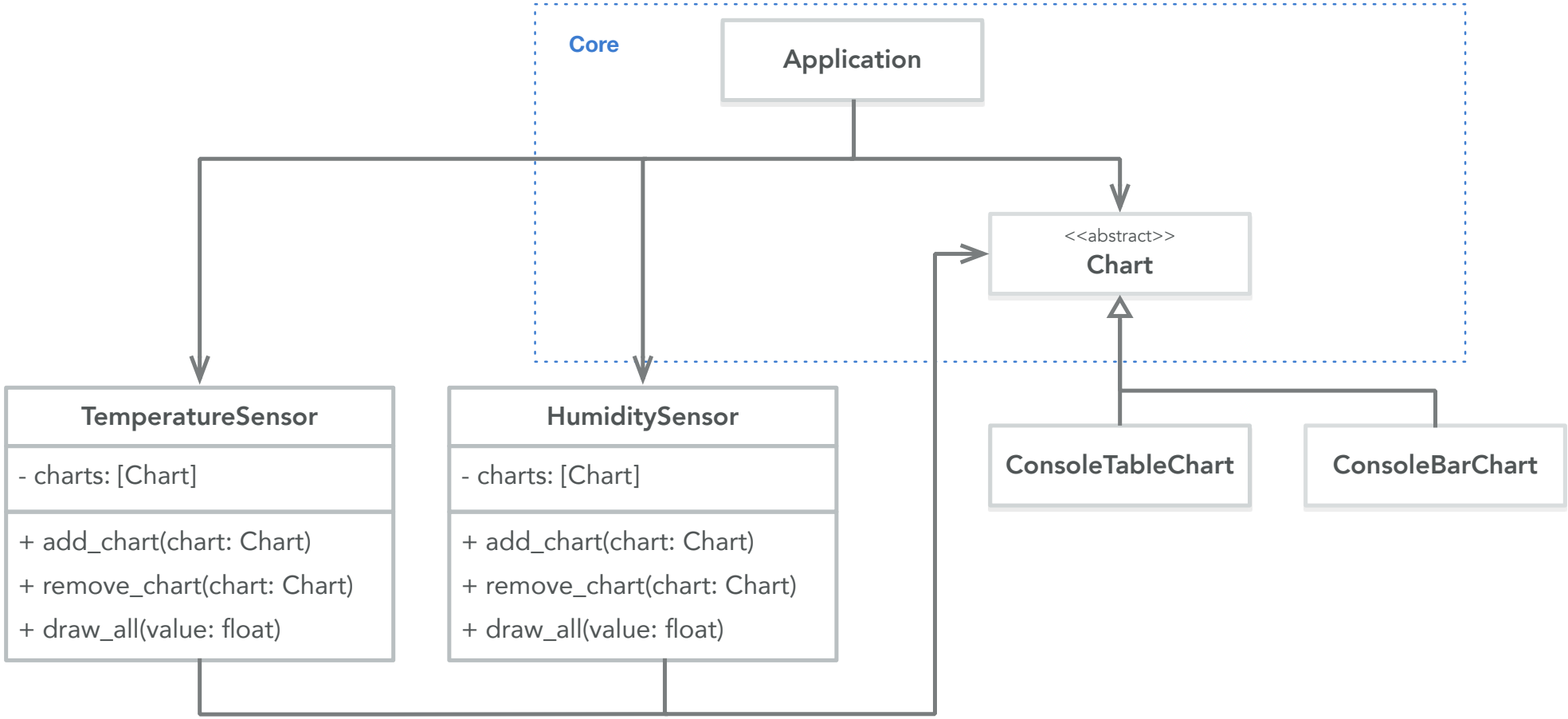
- Isolate concrete classes
- Make exchanging product families easy
- Promote consistency among products
- Supporting new kinds of products is difficult

Stage 4: A second sensor

*To stay ahead of the competition we decide to **add a humidity sensor** as a second sensor type to our application. Our boss reassures us that the sensors are known up front and will not be created dynamically at runtime, so no need for an additional factory. However, both sensors need to be able to send updates to a chart.*

https://www.freeimages.com/

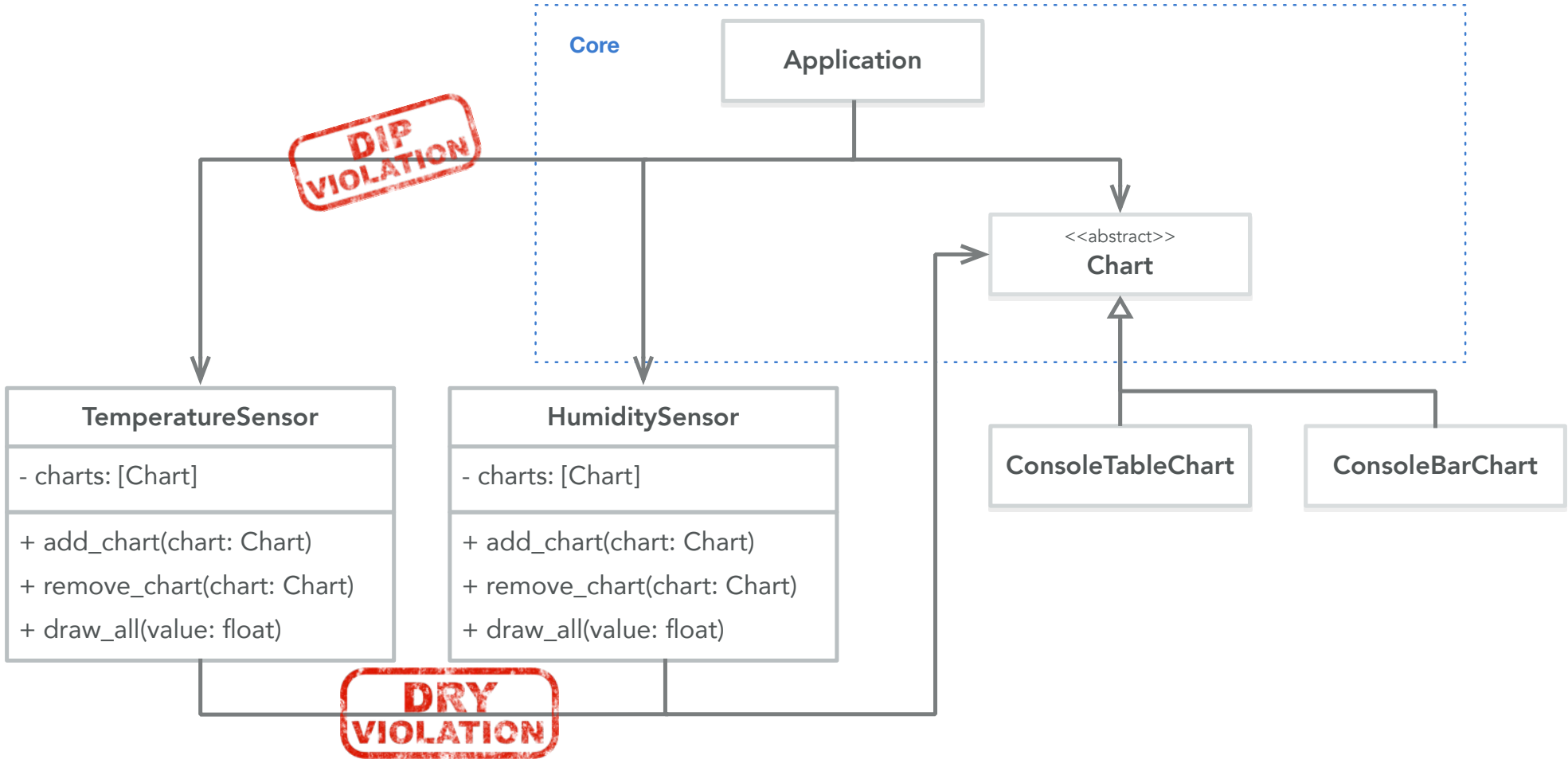
The problem …


Core
Application
<<abstract>>
Chart
TemperatureSensor
- charts: [Chart]
+ add_chart(chart: Chart)
+ remove_chart(chart: Chart)
+ draw_all(value: float)
HumiditySensor
- charts: [Chart]
+ add_chart(chart: Chart)
+ remove_chart(chart: Chart)
+ draw_all(value: float)
ConsoleTableChart
ConsoleBarChart

The problem

*We implement the **HumiditySensor** class and notice some striking similarities to the TemperatureSensor class. Also, the Application class depends directly on the concrete sensor implementations, violating the Dependency Inversion Principle (DIP).*

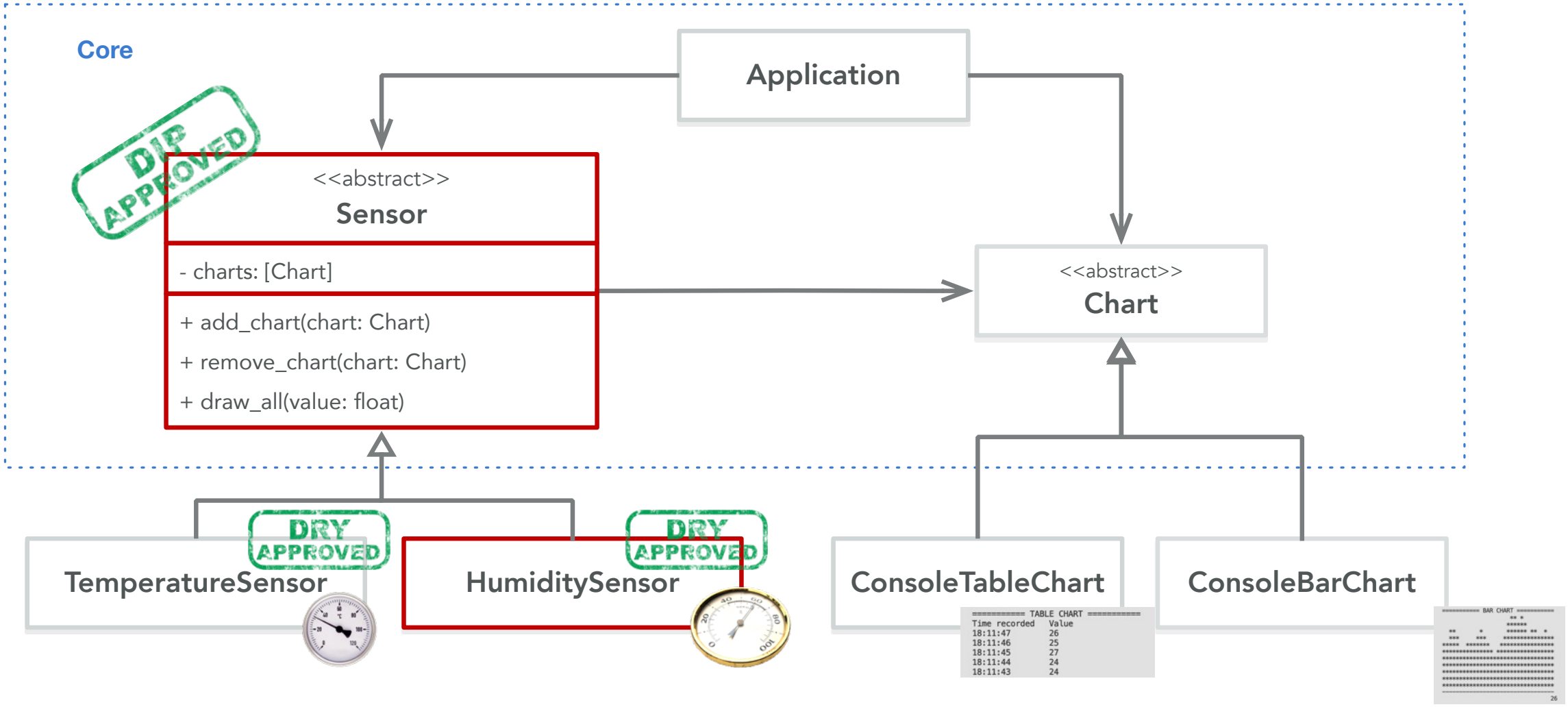
The problem …


Core
Application
DIP VIOLATION
<<abstract>>
Chart
TemperatureSensor
- charts: [Chart]
+ add_chart(chart: Chart)
+ remove_chart(chart: Chart)
+ draw_all(value: float)
HumiditySensor
- charts: [Chart]
+ add_chart(chart: Chart)
+ remove_chart(chart: Chart)
+ draw_all(value: float)
ConsoleTableChart
ConsoleBarChart
DRY VIOLATION

The solution

*We decide to extract a common abstract **Sensor** base class that both the **TemperatureSensor** and **HumiditySensor** inherit from and therefore adhering to the **Don't Repeat Yourself Principle (DRY)** by eliminating duplication. The Sensor class defines an abstract **measure**() method and contains the logic to **add/remove** charts and a **draw_all**() method that calls the **draw**() method of every chart attached to the sensor. The **TemperatureSensor** and **HumiditySensor** can now call this method whenever they measure a new value without needing to implement the logic themselves.*

The solution



Stage 5: Logging

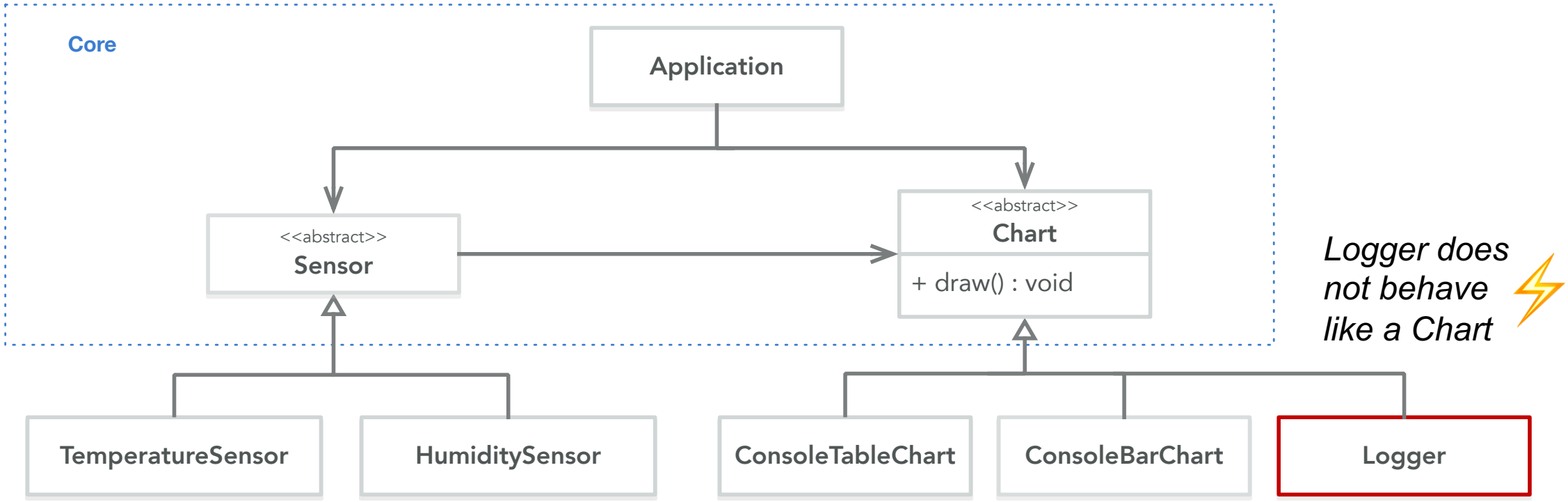
Our charts have been showing some weird data. To find out what's happening under the hood we want to log the values produced by the sensors to a file.

https://www.freeimages.com/

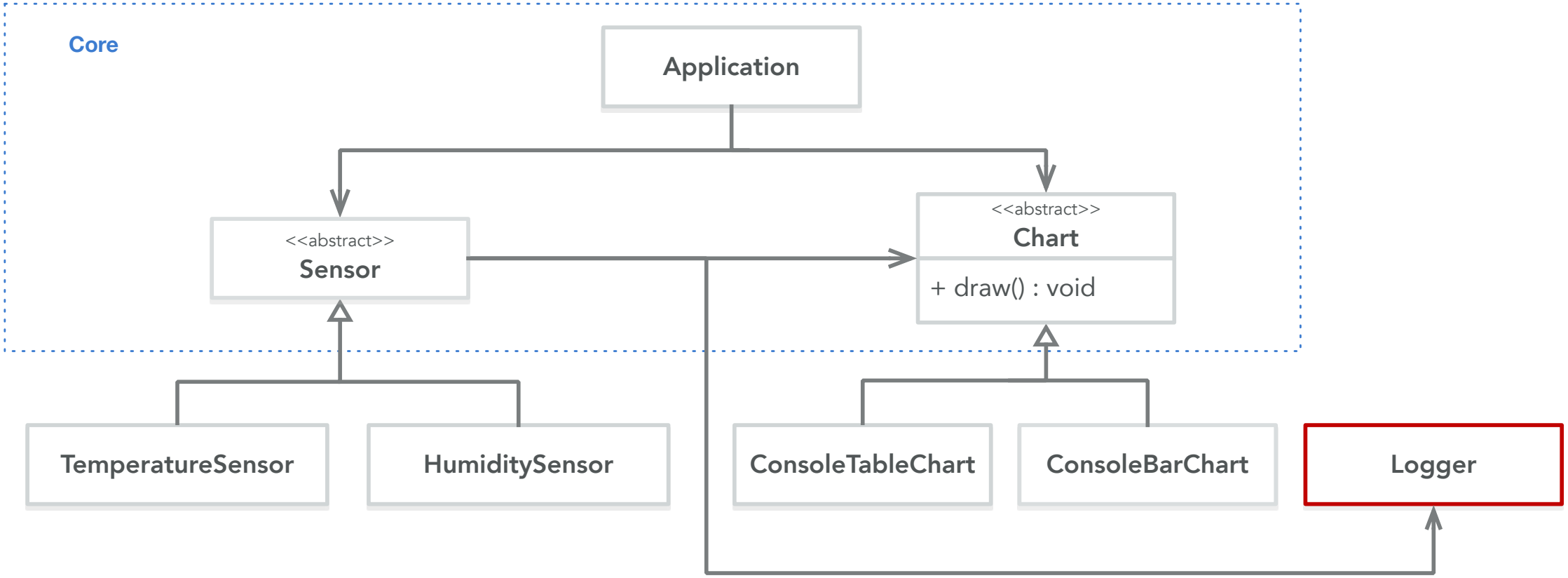
The problem

We need to implement a file logger that is updated along with the charts without implementing the Chart interface.

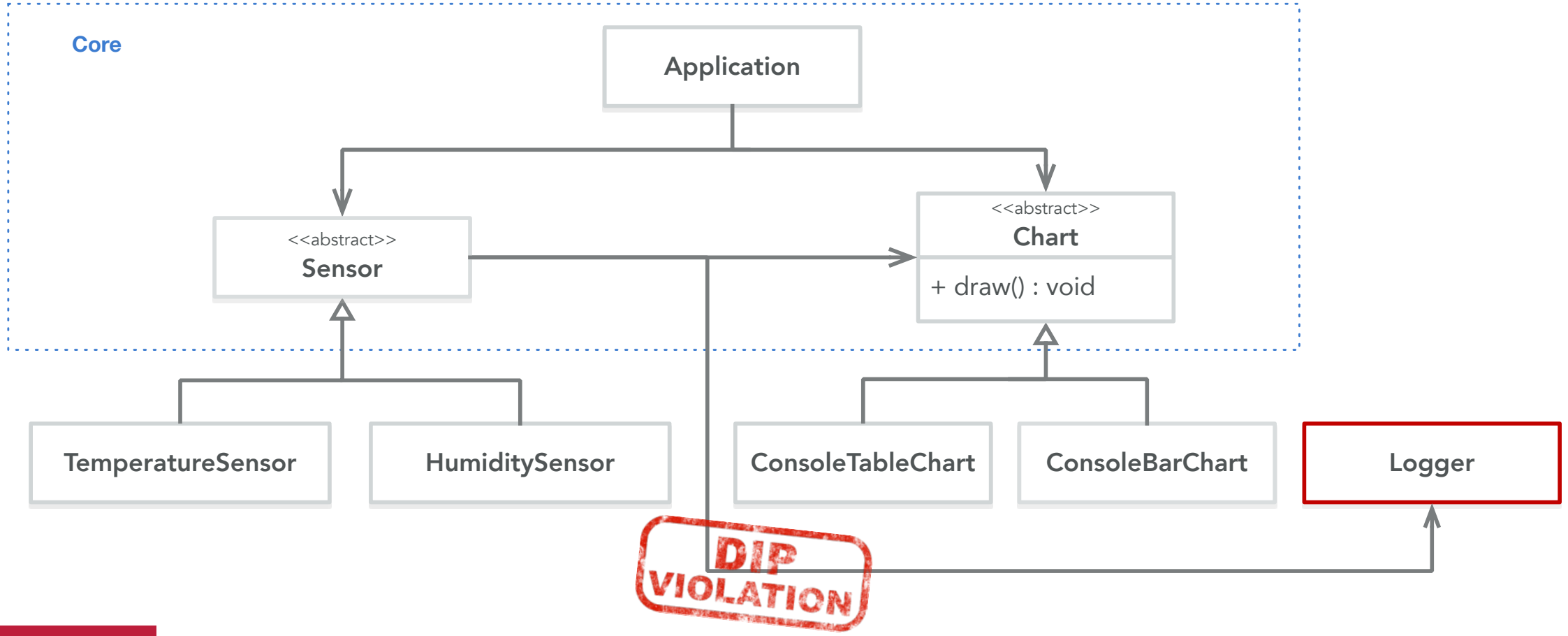
A questionable solution …



The alternative?



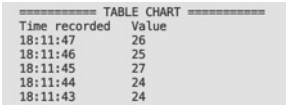
The alternative?

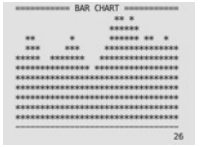


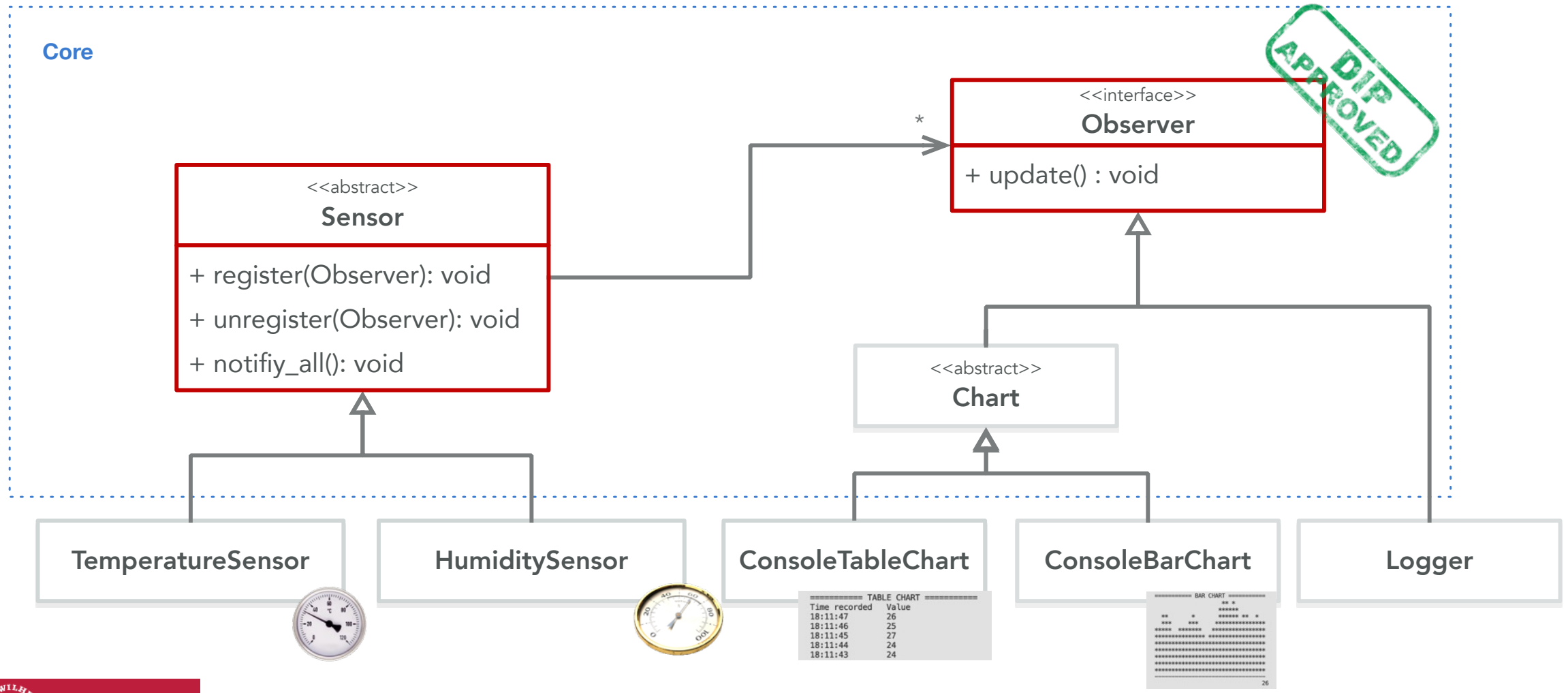
The solution

*We decide to introduce a new interface called **Observer** that contains an **update**() method that both the charts and the new file logger will implement.*
*Instead of using **Charts** our sensors will now talk to **Observer** instances. At this point we notice that managing **Observers** and measuring data are different responsibilities as well. To separate these responsibilities we extract an abstract **Subject** class that only contains the code to manage Observers while keeping the **measure**() method in the **Sensor** class.*
*Finally we make **Sensor** class inherit from the **Subject** class.*
*We now have a generic mechanism to notify interested components of our application about updates. This is called the **Observer Pattern**.*

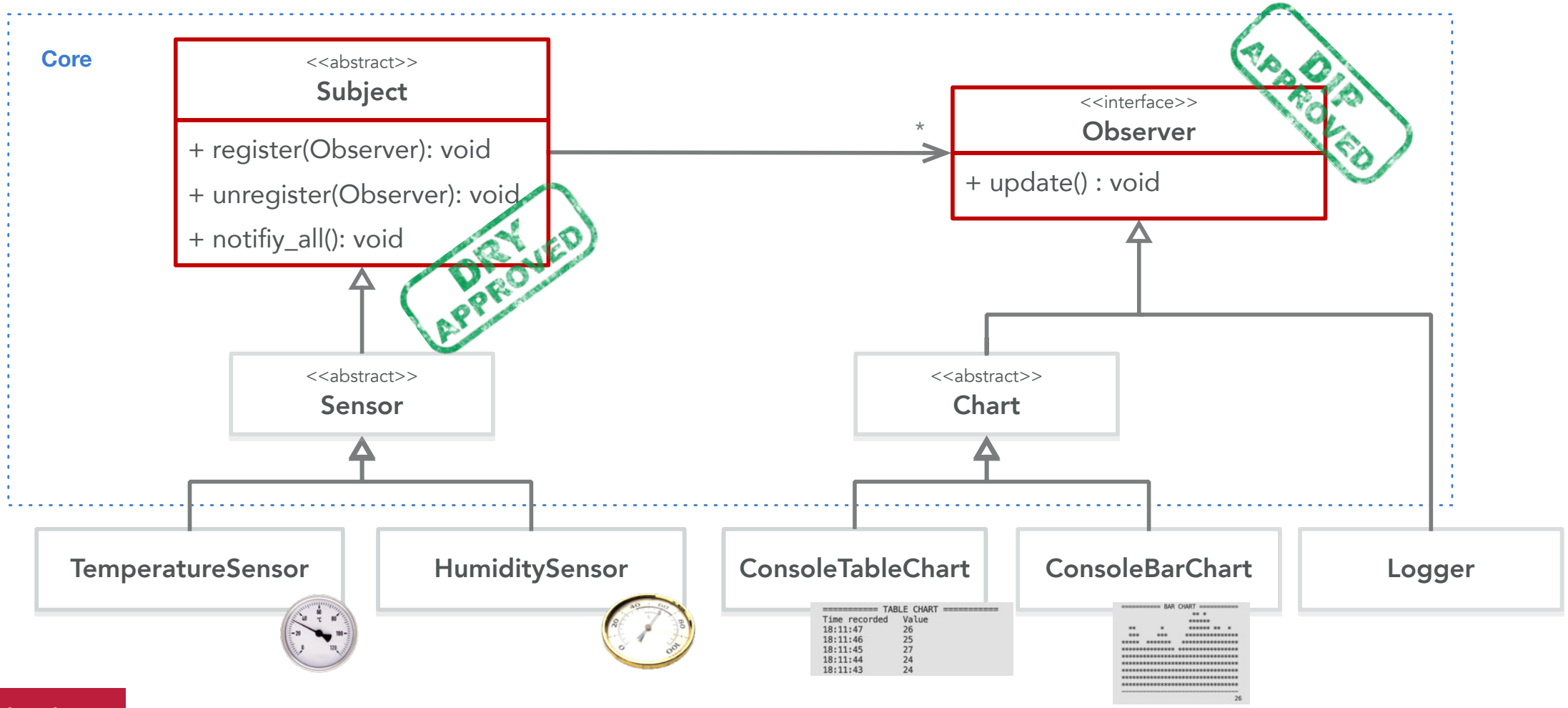
The final solution using Observer Pattern






Core
DIP APPROVED
<<interface>>
Observer
+ update() : void
*
<<abstract>>
Sensor
+ register(Observer): void
+ unregister(Observer): void
+ notifiy_all(): void
<<abstract>>
Chart
TemperatureSensor
HumiditySensor
ConsoleTableChart
ConsoleBarChart
Logger
========== TABLE CHART ==========
Time recorded Value
18:11:47 26
18:11:46 25
18:11:45 27
18:11:44 24
18:11:43 24
========== BAR CHART ==========

The final solution using Observer Pattern



Observer Pattern

- **Object Behavioral pattern**
- “Intent: Define a one-to-many dependency between objects so that when objects change state, all its dependents are notified and updated automatically.”*

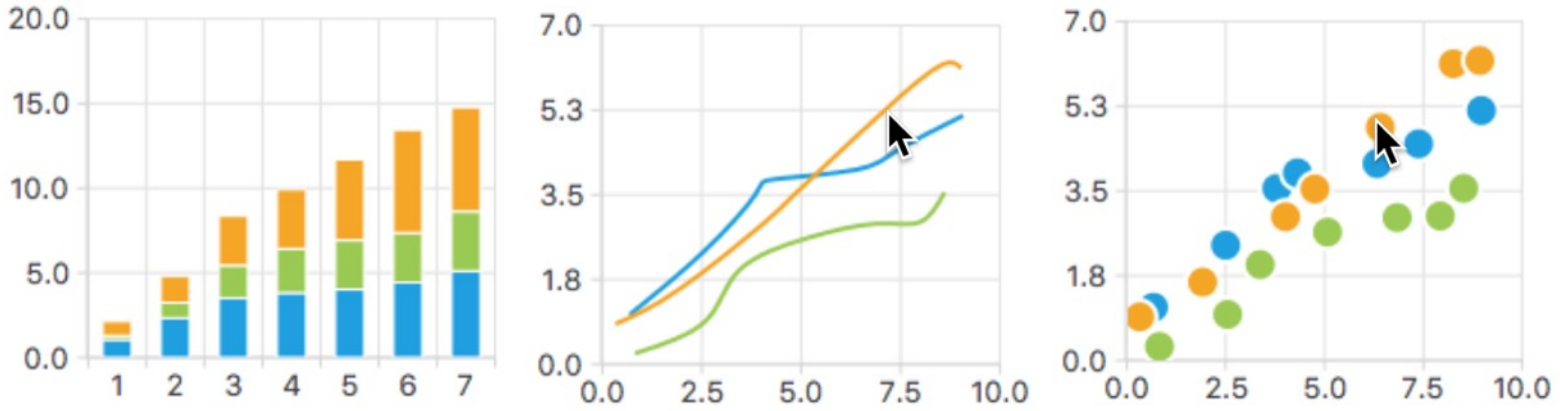
* Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, 1995

Observer

Purpose

1-to-n-Dependencies between Objects: An object can notify all dependent objects

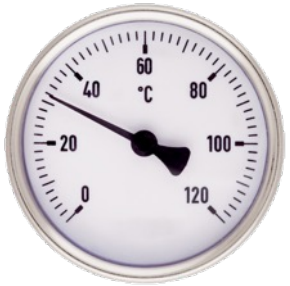
GUI



?

More to come!

volatil



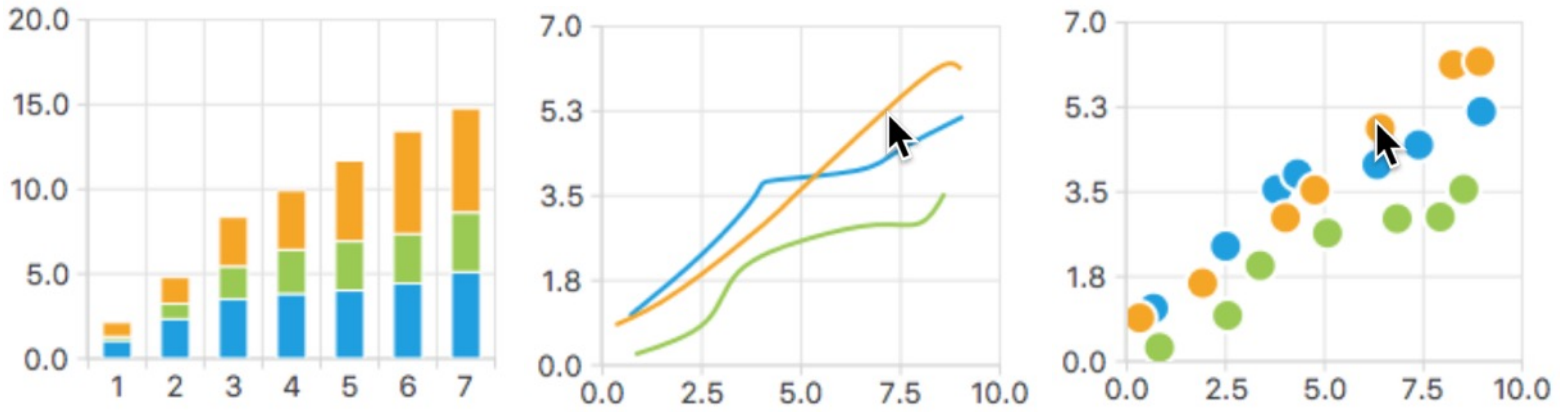
non volatil

Observer

Solution 1

Pull data instead of pushing it

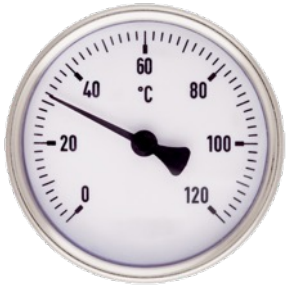
GUI



?

More to come!

volatil



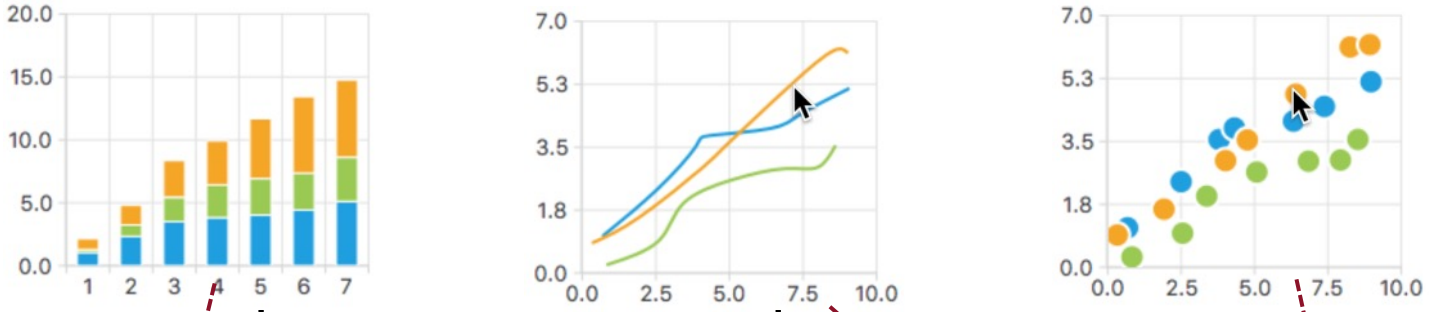
non *volatil*

Observer

Solution 2

Dependency Inversion

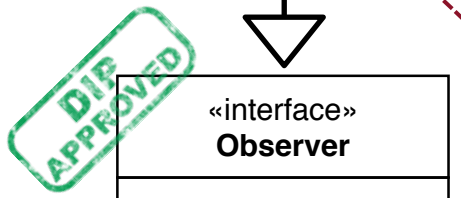
GUI



?

More to come!

volatil

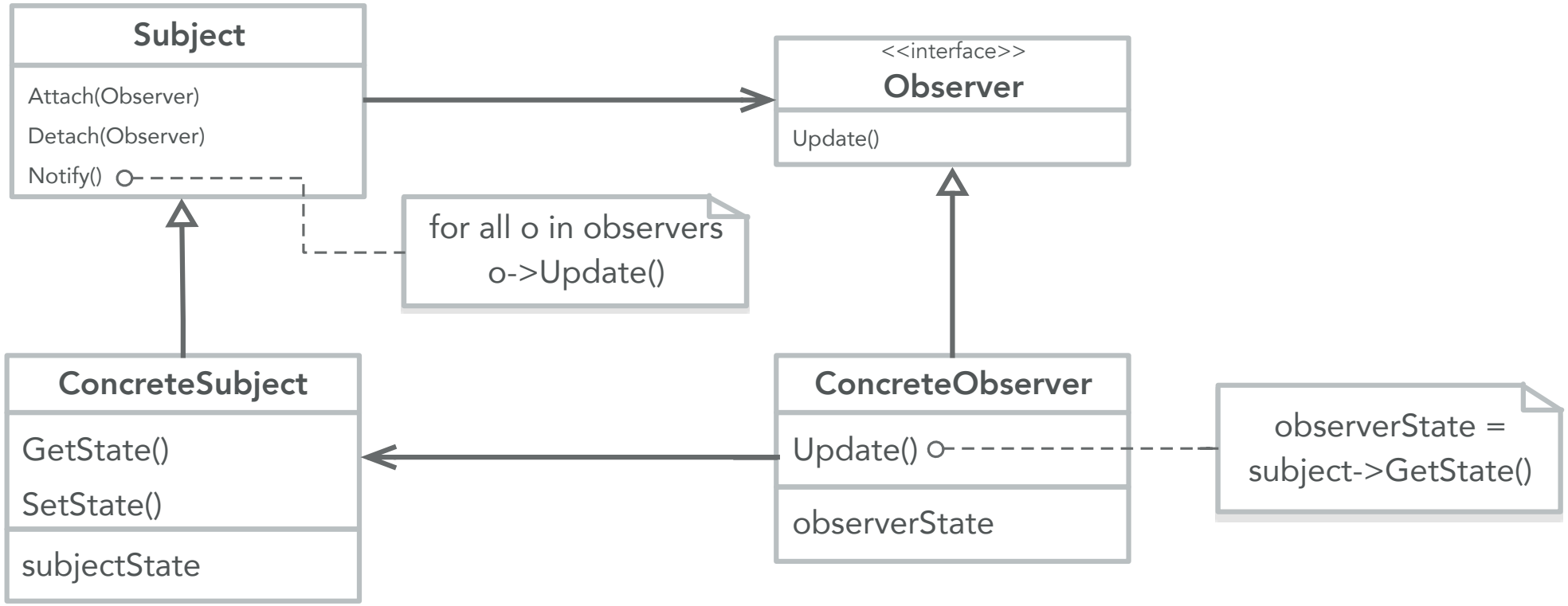


«interface»
Observer

+ update() : void = 0

non volatil

Observer Pattern



Observer Pattern

Benefits and liabilities

- Abstract coupling between Subject and Observer
- Support for broadcast communication
- Push instead of pull mechanism

Stage 6: Another strategy

*Our clients are complaining again. This time it's the resource consumption of our application. Some of them are using it on **very low power machines**. Measuring the sensors and updating the charts every second is too much for those tiny CPUs. Some want to have the application only **display the data once and then exit**, others would like to **trigger an update manually** and the rest wants to keep the **continuous updating** we already have.*

The problem

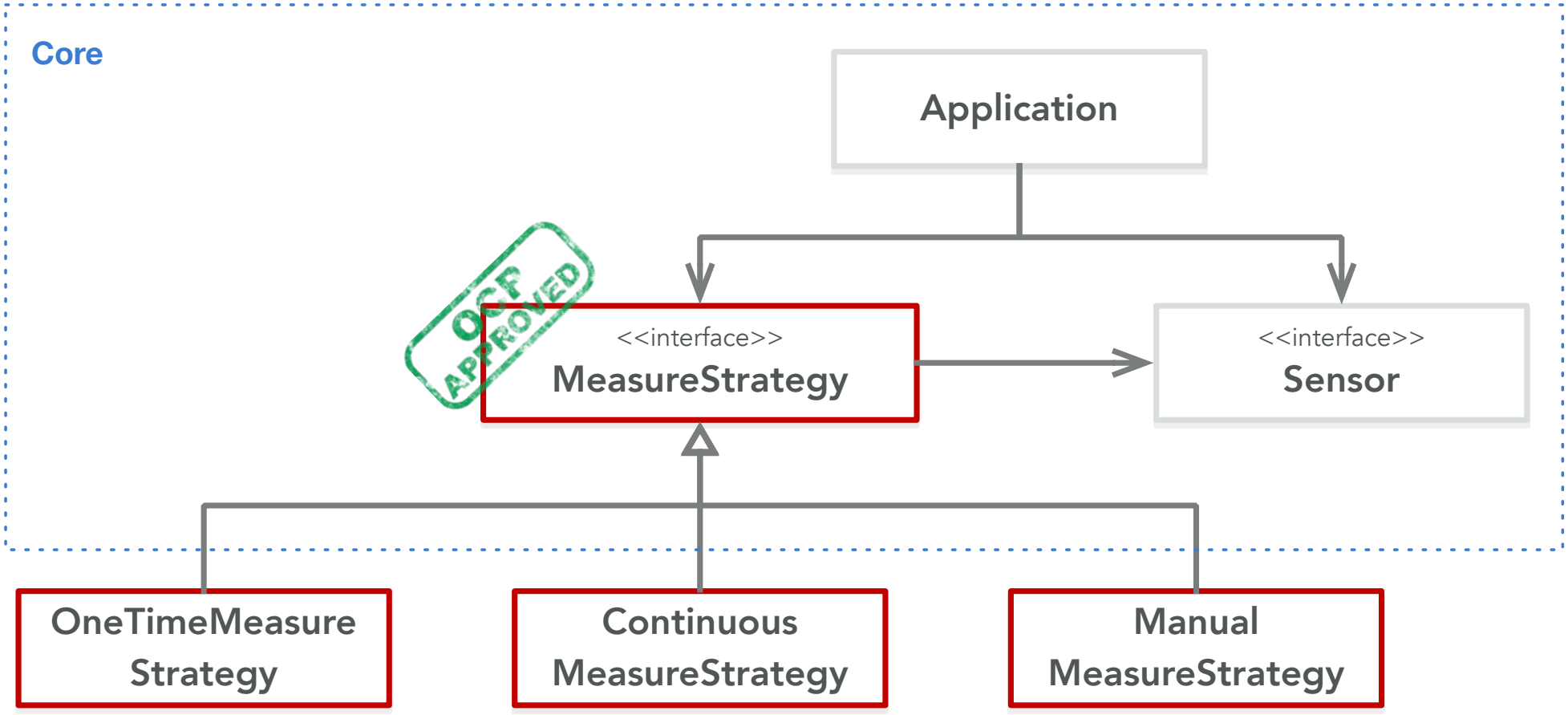
*We need to implement a **mechanism to swap out the behaviour** that triggers the measurement. Since some of our clients want to trigger updates manually we can't just set a different timer value.*

The solution

We could have an if statement in the Application's ***run()*** *method to determine when to measure and update our charts. However, that would violate the* ***Open Closed Principle (OCP)*** *since we'd have to extend that if statement whenever the requirements for measuring change. Instead we decide to apply the* ***Dependency Inversion Principle (DIP)*** *and encapsulate the algorithm that determines when we want to call the* ***measure()*** *method into different classes that all implement a common interface that we'll call* ***MeasureStrategy****.*

When starting our program we inject either a ***OneTimeMeasureStrategy****, a* ***ManualMeasureStrategy*** *or a* ***ContinuousMeasureStrategy*** *into the Application class. Instead of calling the sensor's measure method itself, the Application class will delegate that responsibility to the respective Strategy. We call this the* ***Strategy Pattern****.*

The solution



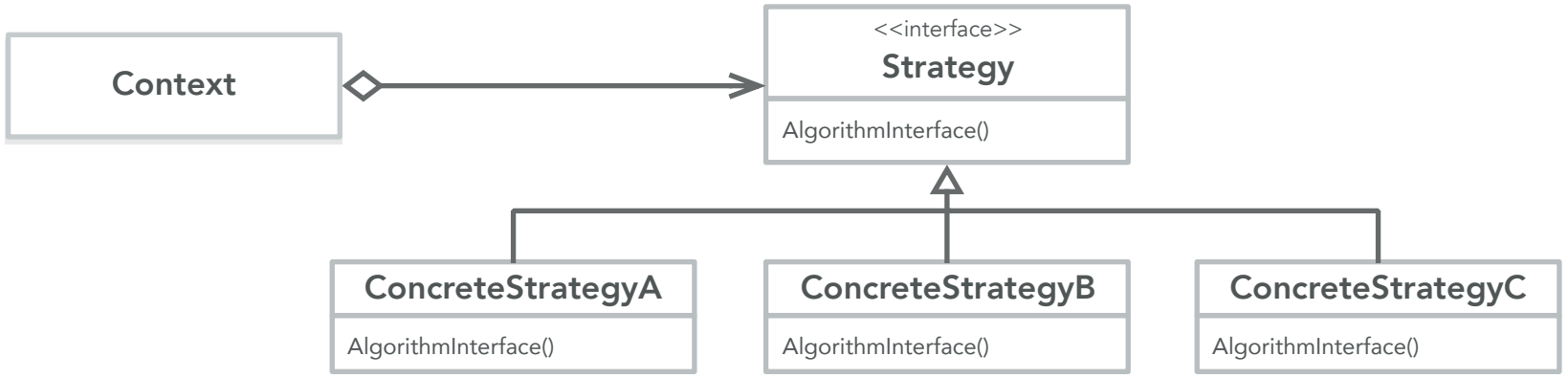
Strategy Pattern

- **Object Behavioral pattern**

- “Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.”*

* Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, 1995

Strategy Pattern

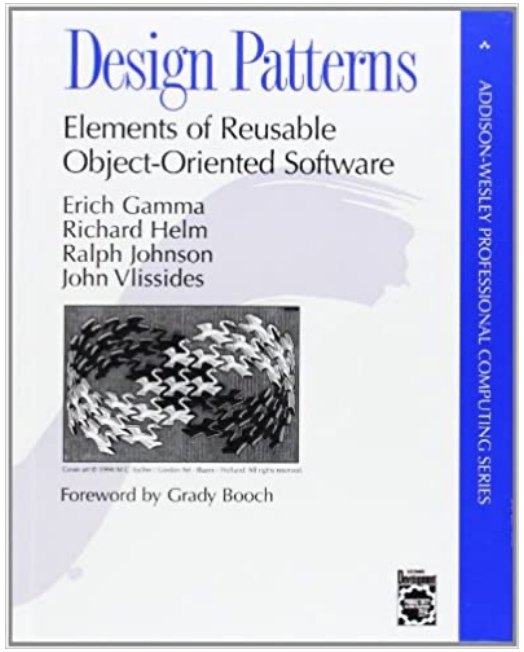


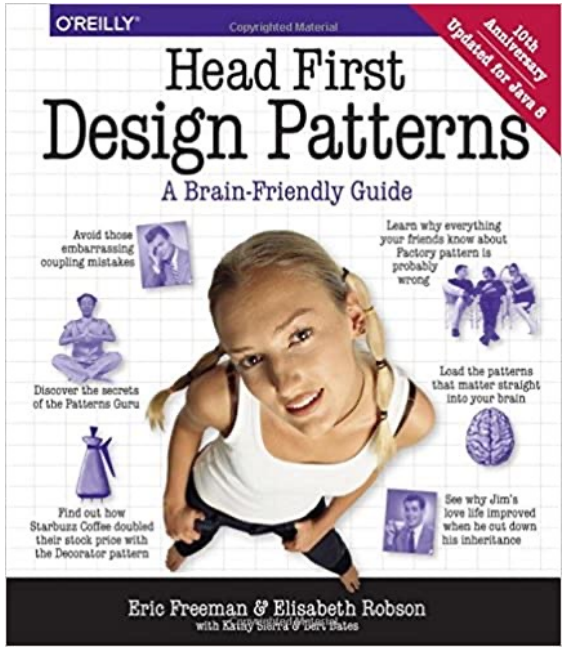
Strategy Pattern

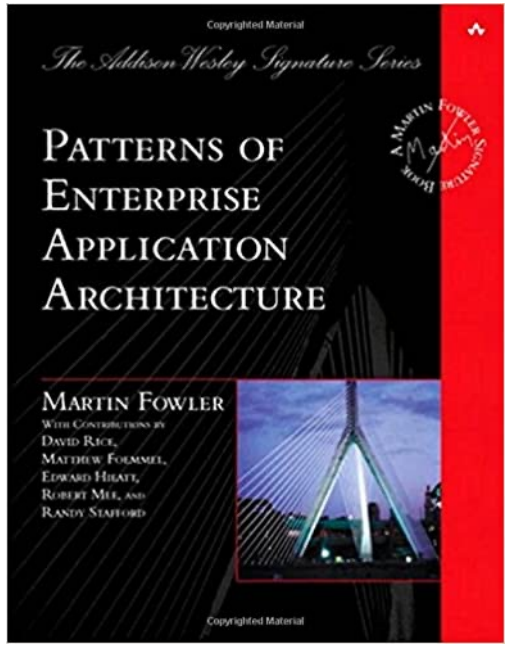
Benefits and liabilities

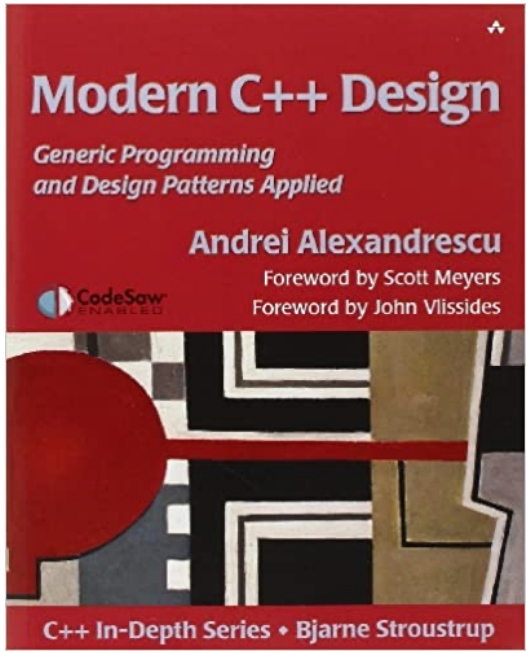
- Families of related algorithms
- Alternative to subclassing
- Eliminates conditional statements
- Communication overhead between context and strategy
- Increasing number of objects

Bibliography










Thanks!

https://forms.gle/Ld56WrTS3iqpynQg6

References

McCall,J.: Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager, Bd. 1-3. General Electric, November 1977.

Meyer, B.: Object-Oriented Software Construction, Prentice Hall PTR, 1988.

McConnell, S.: Code Complete, Second Edition. Microsoft Press, Redmond, WA, USA, 2004.

Dijkstra,E.W.:The humble programmer. Commun. ACM, 15(10):859–866, 1972.

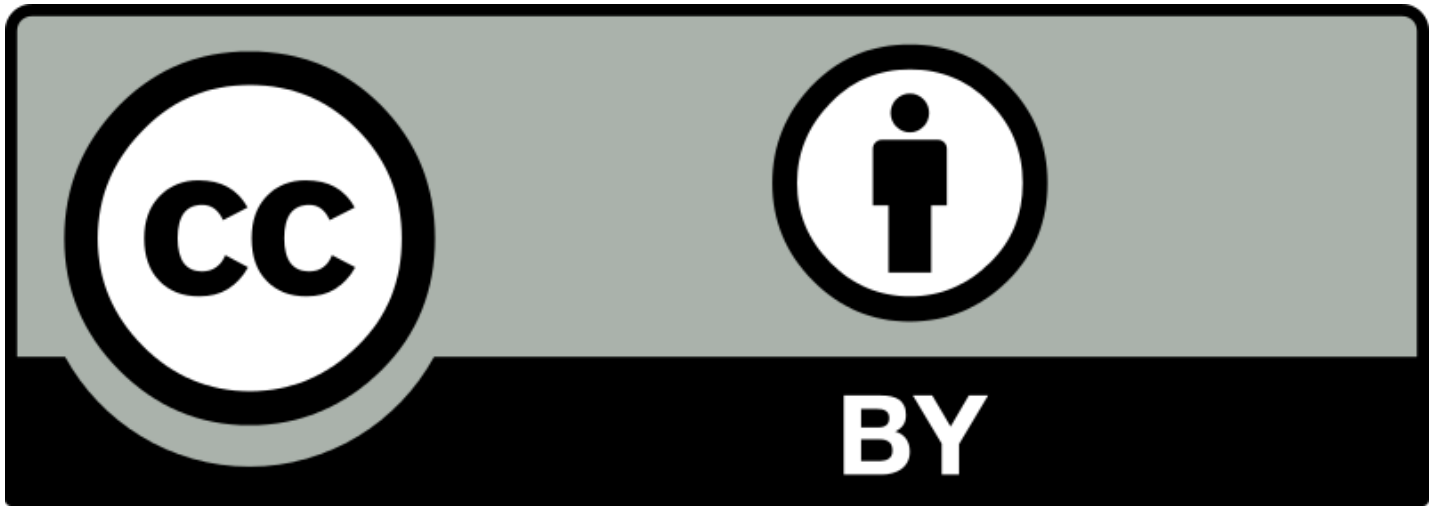
Booch, G., Maksimchuk, R.A., Engle, M.W., Young, B.J., Connallen, J. und Houston, K.A.: Object-oriented analysis and design with applications. Addison Wesley, 3. Aufl., 2007.

Yourdon, E. und Constantine, L. L.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Yourdon Press computing series. Prentice-Hall, Inc, Upper Saddle River, NJ, USA, 1979.

Ingalls,D.H.H.:Design Principles Behind Smalltalk. Byte,6(8):286–298,1981.

Brooks, Jr., F.P.: No Silver Bullet - Essence and Accidents of Software Engineering. Computer, 20(4):10–19, 1987.

Jack W. Reeves:, What is software design?. C++ Journal, 1992, http://www.developerdotstar.com/mag/articles/reeves_design.html


BY

The SURESOFT project is funded by the German Research Foundation (DFG) as part of the “e-Research Technologies” funding programme under grants: EG 404/1-1, JA 2329/7-1, KA 3171/12-1, KU 2333/17-1, LA 1403/12-1, LI 2970/1-1 and STU 530/6-1.