SIEMENS Healthineers

Refactoring for Design Smells

Training material sourced from CT RDA

Outline

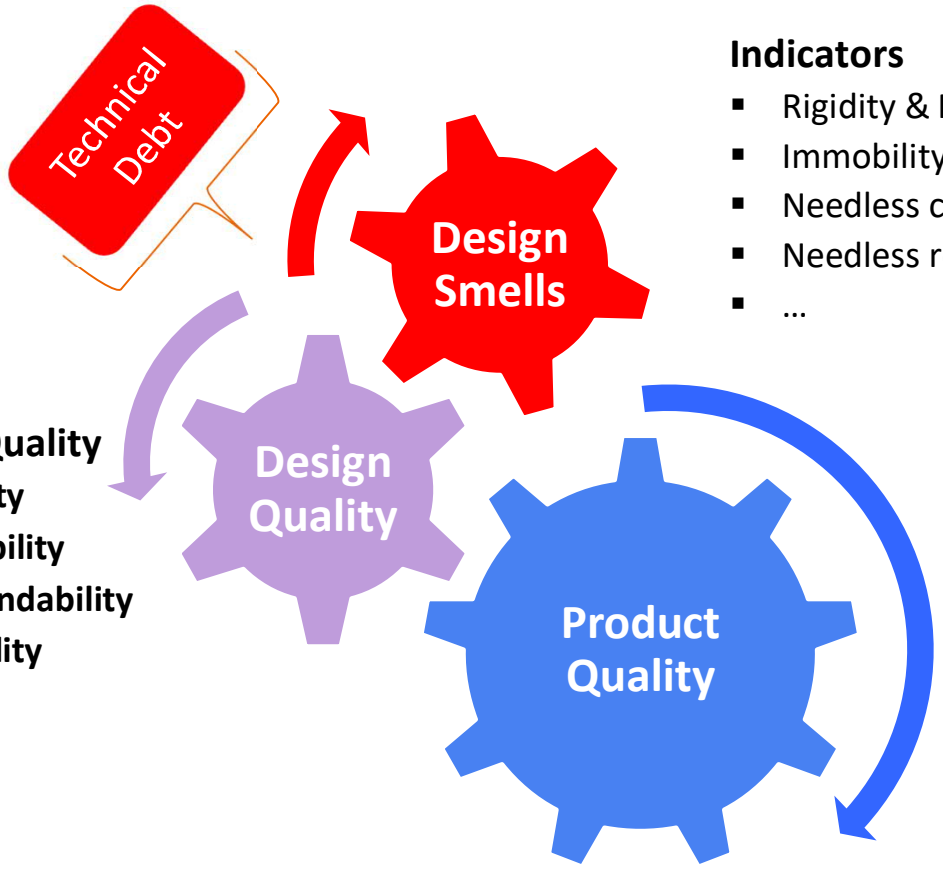
- Introduction
- Abstraction smells
- Encapsulation smells
- Modularization smells
- Hierarchy smells
- Refactoring design smells in practice

“Design smells” aka…

What is a smell?

“Design smells are certain structures in the design that indicate violation of fundamental design principles and negatively impact design quality.”

Why care about smells?



Indicators

- Rigidity & Fragility
- Immobility & Opacity
- Needless complexity
- Needless repetition
- …

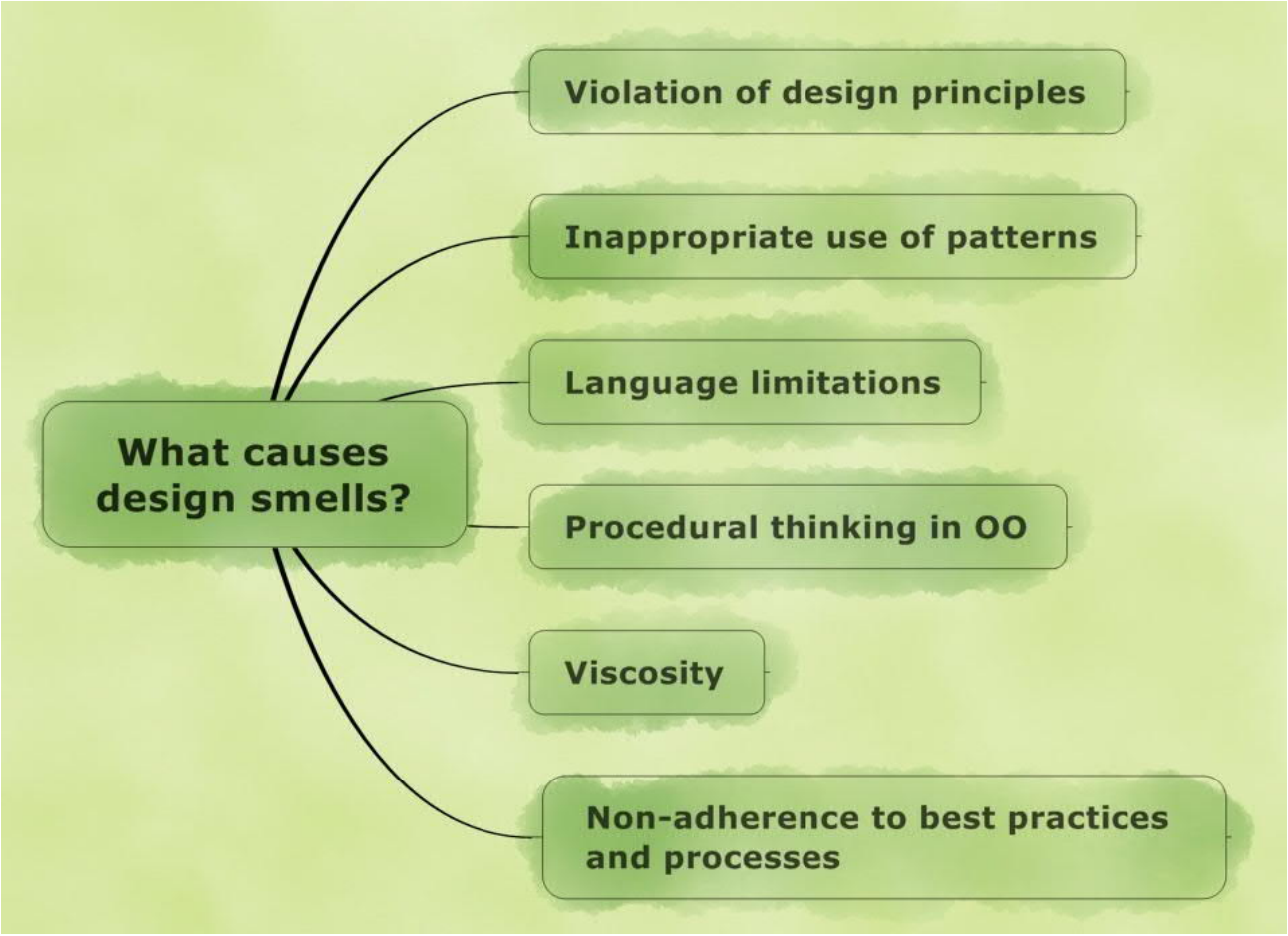
Impacted Quality

- **Reusability**
- **Changeability**
- **Understandability**
- **Extensibility**
- **…**

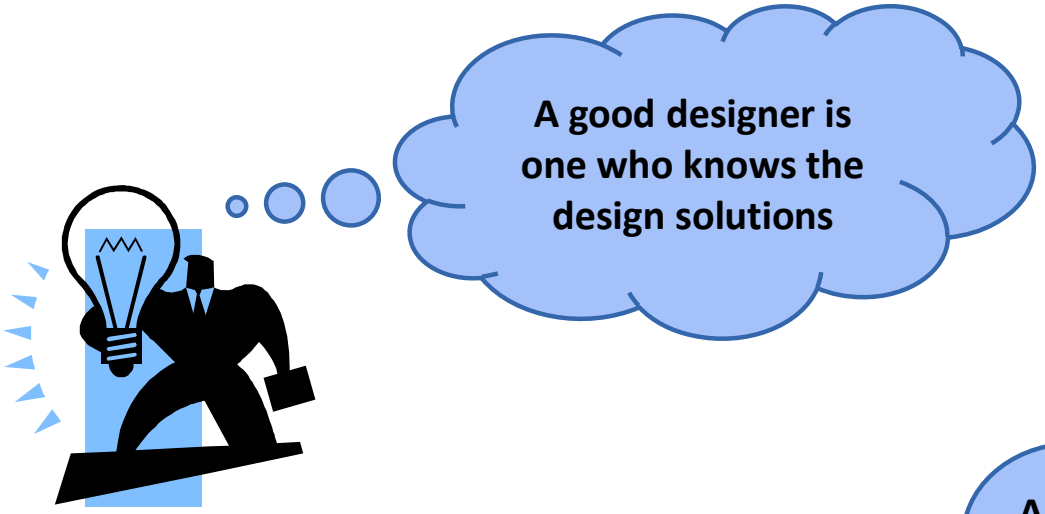
Impacted Quality

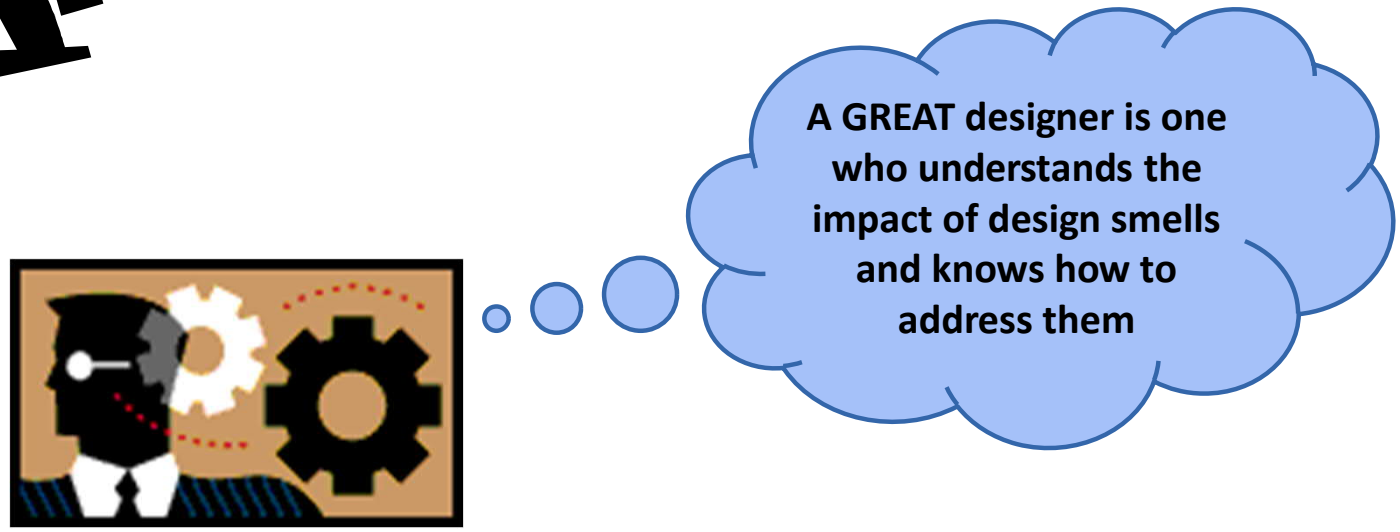
- *Maintainability:* Affected by changeability & extensibility
- *Reliability:* Impacted by poor understandability
- *Performance*
- …

What causes design smells?



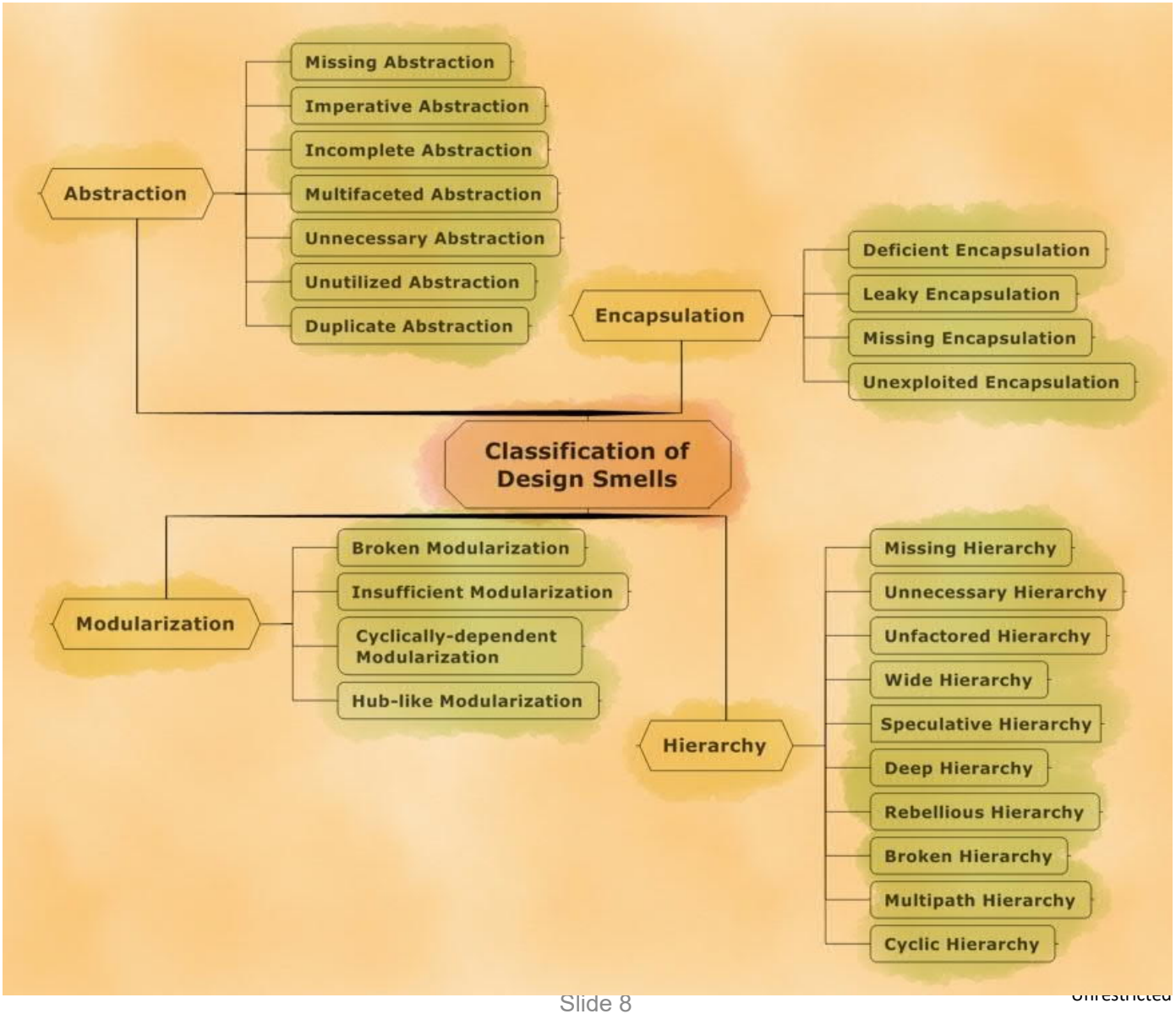
Why we focus on smells?





Quality attributes impacted by smells

Quality Attribute	Definition
Understandability	The ease with which the design fragment can be comprehended.
Changeability	The ease with which a design fragment can be modified (without causing ripple effects) when an existing functionality is changed.
Extensibility	The ease with which a design fragment can be enhanced or extended (without ripple effects) for supporting new functionality.
Reusability	The ease with which a design fragment can be used in a problem context other than the one for which the design fragment was originally developed.
Testability	The ease with which a design fragment supports the detection of defects within it via testing.
Reliability	The extent to which the design fragment supports the correct realization of the functionality and helps guard against the introduction of runtime problems.

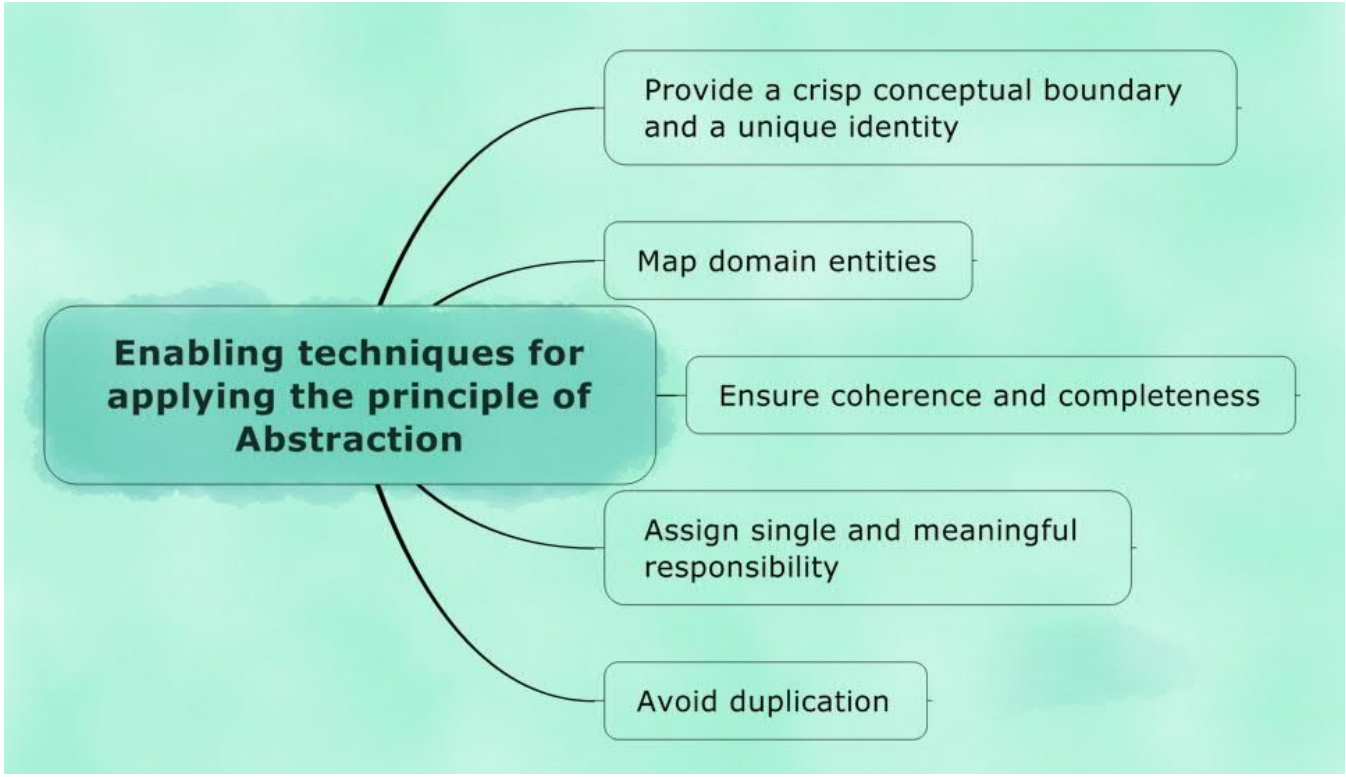


Outline

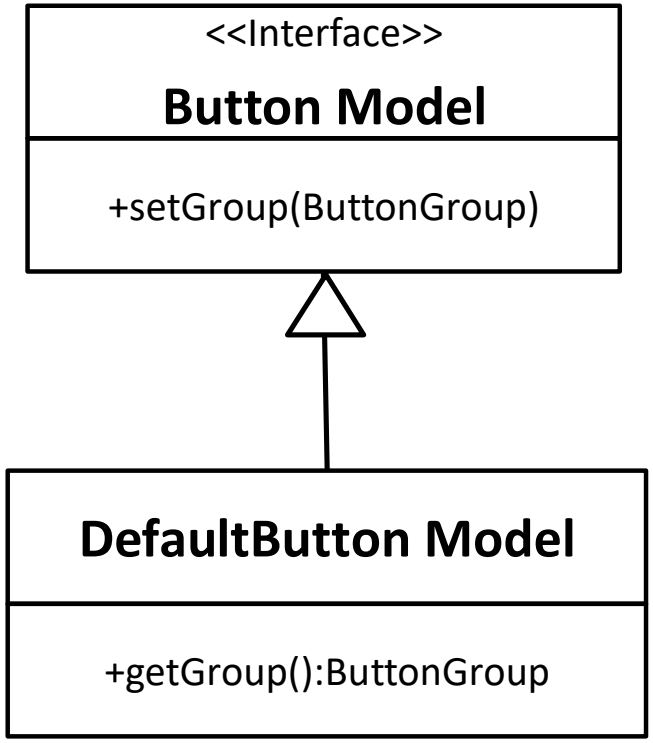
- Introduction
- Abstraction smells
- Encapsulation smells
- Modularization smells
- Hierarchy smells
- Refactoring design smells in practice

The principle of abstraction

Enabling techniques for abstraction



What's that smell?



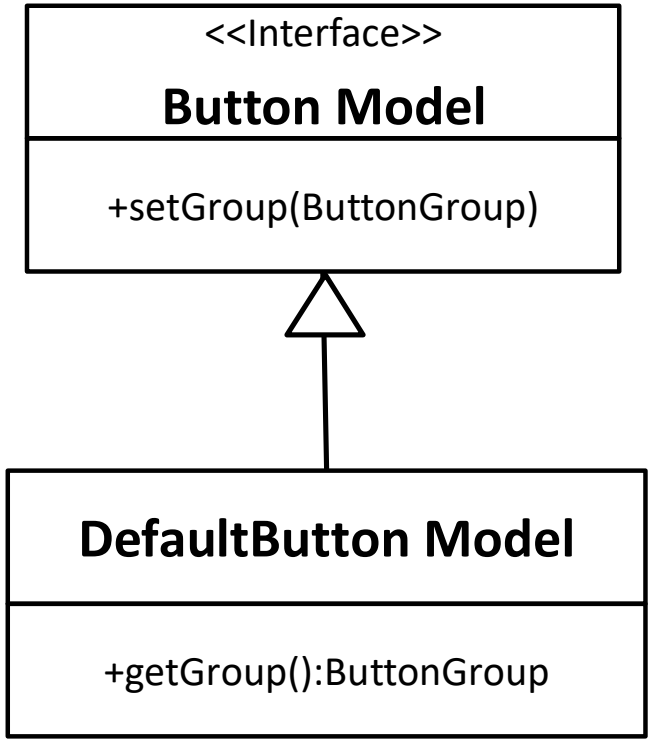
Incomplete Abstraction

This smell arises when an abstraction does not support complementary or interrelated methods completely

- Specifically, the public interface of the type is incomplete in that it does not support all related behavior needed by objects of its type

Imagine a car with an accelerator but without a brake!

Incomplete Abstraction – Example



- In this case, the ButtonModel interface supports only setGroup() but no corresponding getGroup() (which is provided in its derived class!)
- Hence, ButtonModel suffers from Incomplete Abstraction smell
- How to fix it?
  - "**Ensure coherence and completeness**"
- Provide all the *necessary and relevant* methods required for satisfying a responsibility completely in the class itself
  - In case of public APIs (as in this case), it is often "too late" to fix it!

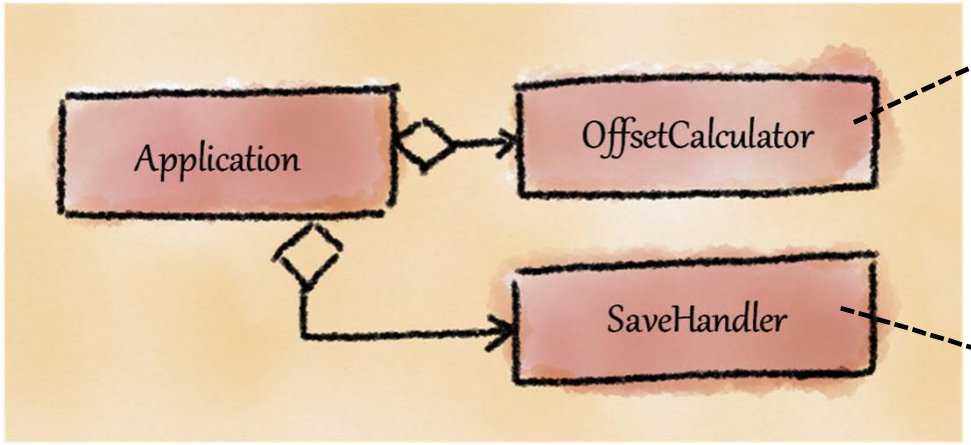
How to refactor & in future avoid this smell?

- For each abstraction (especially in public interface) look out for symmetrical methods or methods that go together
  - Look out for missing matching methods in symmetrical methods (see table)

min/max	open/close	create/destroy	get/set
read/write	print/scan	first/last	begin/end
start/stop	lock/unlock	show/hide	up/down
source/target	insert/delete	first/last	push/pull
enable/disable	acquire/release	left/right	on/off

Caution – Purposely done for e.g. read-only collection

What's that smell?



Contains only one method "calculateOffset"

Contains only one method "saveOffset"

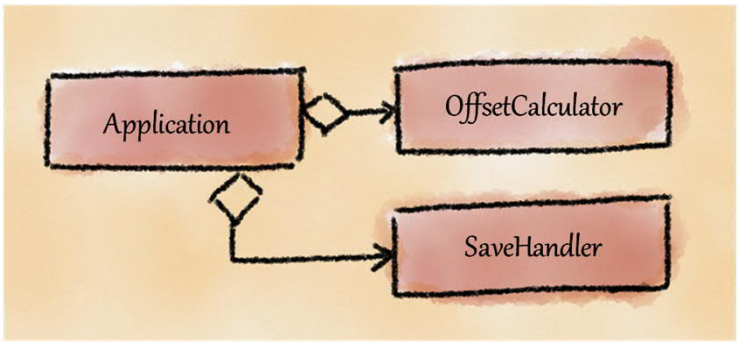
An image processing application, wherein, the Application class uses the `OffsetCalculator` to compute the offset between 2 images and saves it via `SaveHandler` class

Imperative Abstraction

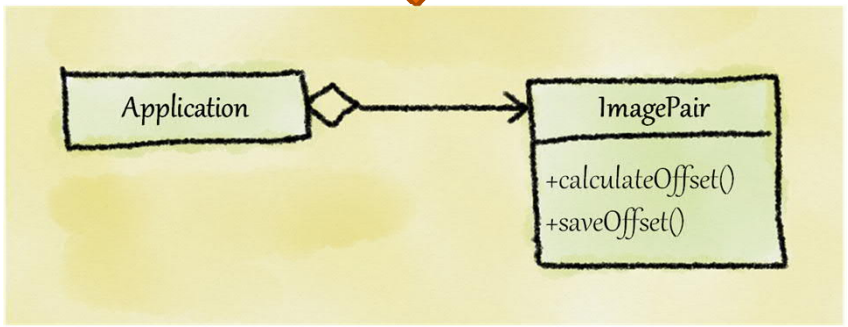
This smell arises when an operation is turned into a class.

This smell manifests as a class that has only one method defined within the class. At times, the class name itself may be identical to the one method defined within it.

Refactoring Imperative Abstraction smell

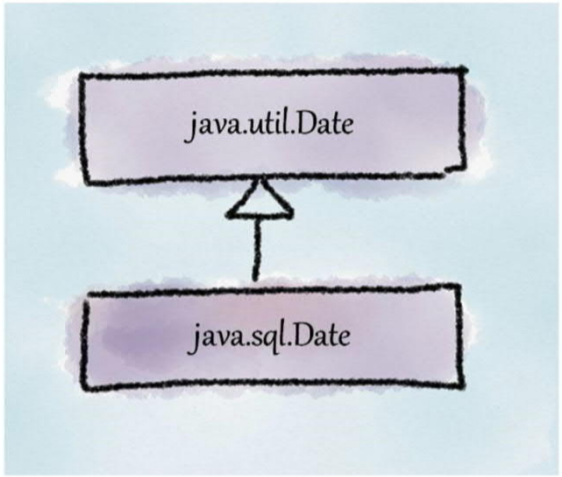


"Map domain entities"



Caution – Patterns e.g. Command and Strategy

What's that smell?



What's that smell?

Scanner

PublicScanner

org.eclipse.jdt.internal
.compiler.parser

org.eclipse.jdt.
core.compiler

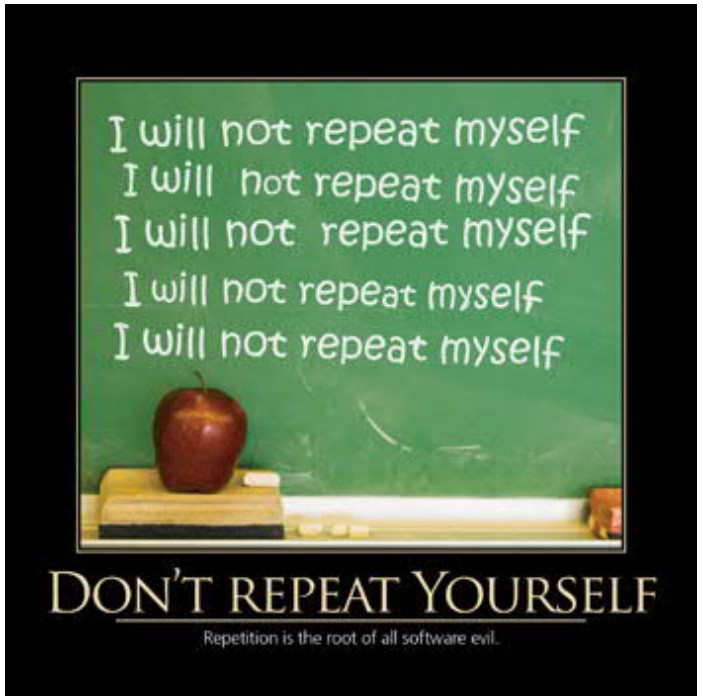
About 3500 lines of code (i.e. 99% of the code) are identical across these two classes!

Duplicate Abstraction

This smell arises when two or more abstractions have identical names or identical implementation or both.

Refactoring Duplicate Abstraction smell

- “Avoid duplication”
- Change name of one of the abstractions
- Factor out the commonality into a third abstraction which can be used by the original abstractions
- Remove a duplication from one of the abstractions and make it refer to the other abstraction for that particular functionality



What's that smell?

- ISBN is represented as a string
- At different parts in the code, there are checks for
  - Group number,
  - Publisher,
  - Title
  - …..
- And some operations are performed based on the information

Missing Abstraction

This smell arises when clumps of data or encoded strings are used instead of creating a class or an interface.

Refactoring for Missing Abstraction smell

"Provide crisp and conceptual boundaries and a unique identity"

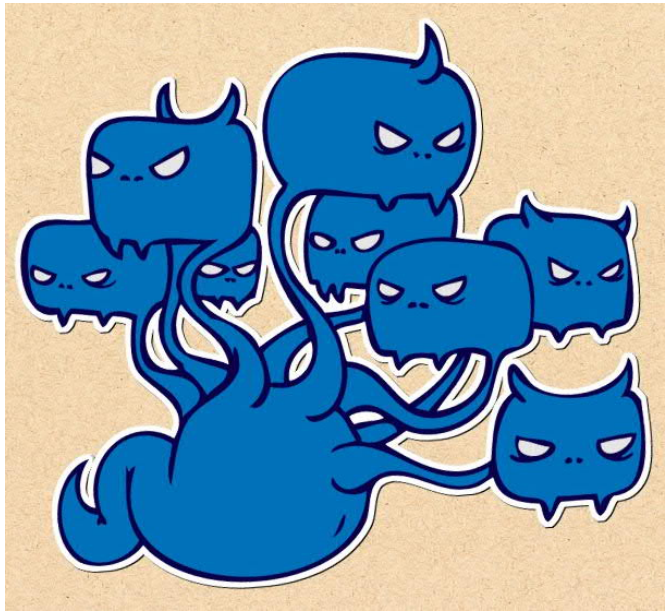
```
public class ISBN{
        private String isbnString;
        public int getGroupNumber();
        public  String getPublisherInformation();
        // other methods elided
}
```

Caution – Check if there is behavior associated with data

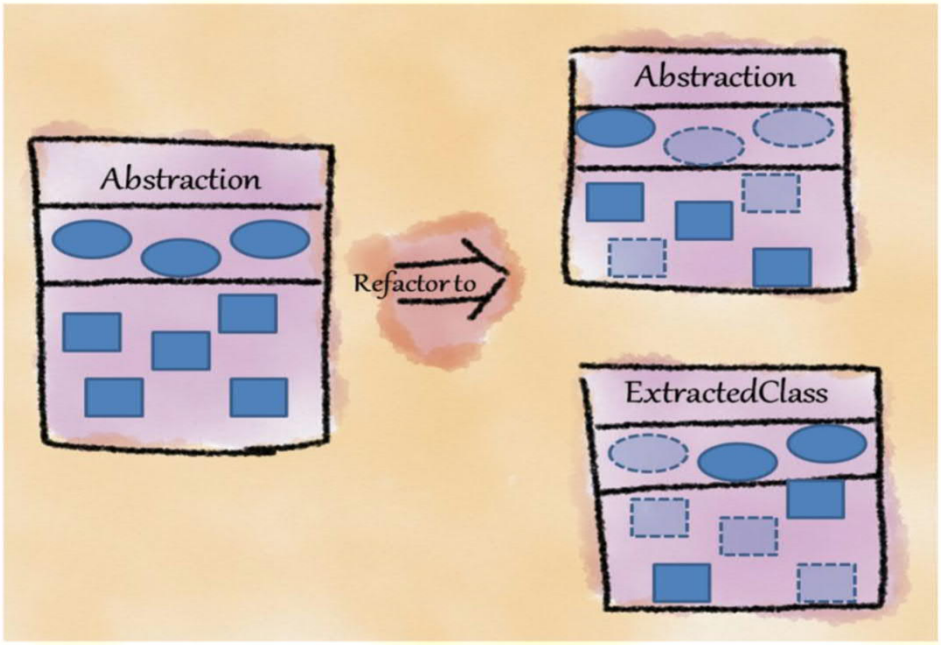
What's that smell?

Multifaceted Abstraction

This smell arises when an abstraction has more than one responsibility assigned to it.



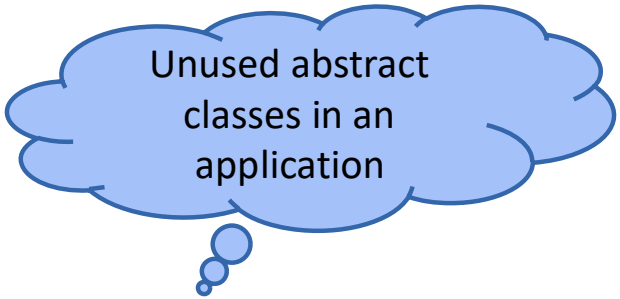
Refactoring for Multifaceted Abstraction smell

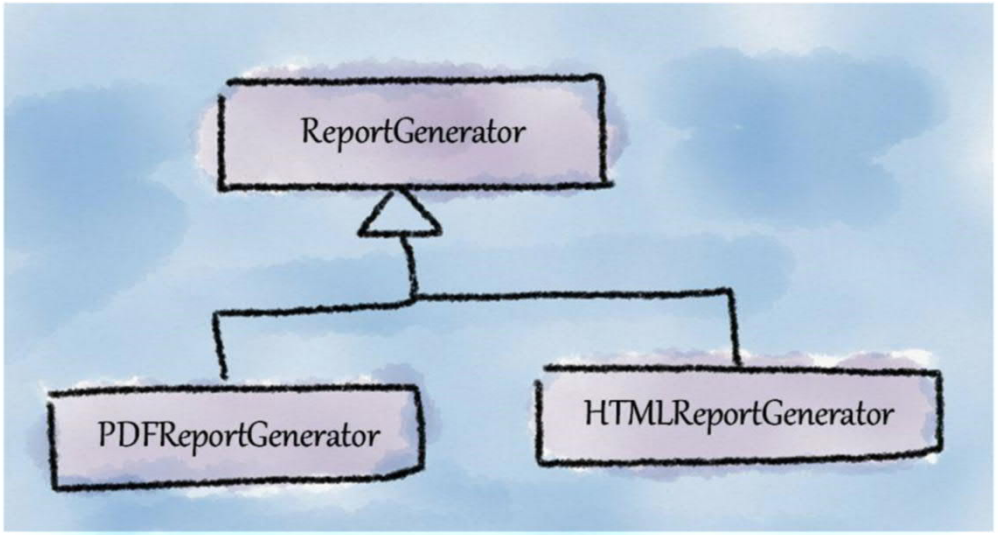


"Assign single and meaningful responsibility"

Caution – To reduce complexity of an already extremely-complex design

What's that smell?





Unutilized Abstraction

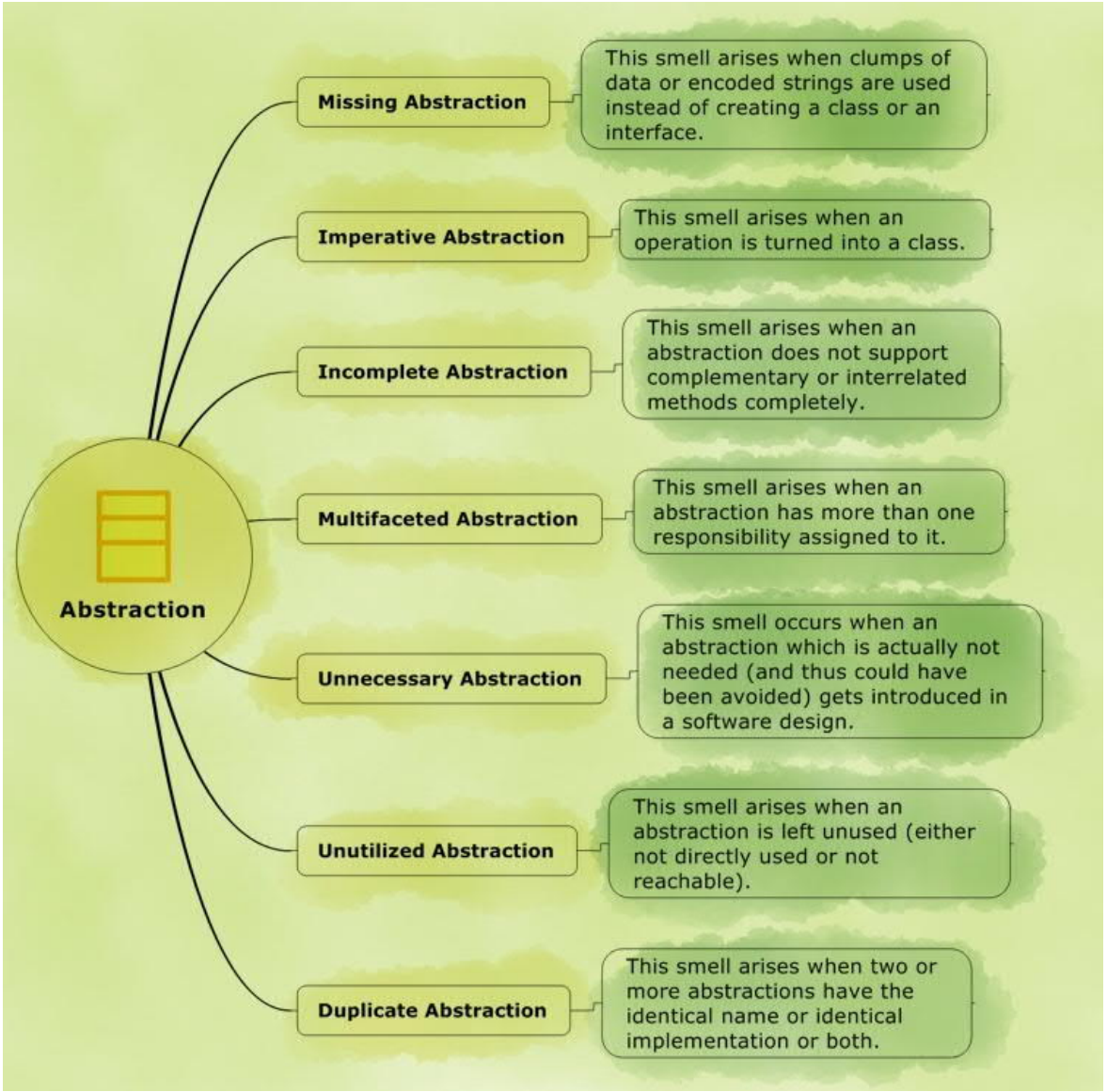
This smell arises when an abstraction is left unused (either not directly used or not reachable). Two forms:

- *Unreferenced abstractions* – Concrete classes that are not being used by anyone
- *Orphan abstractions* – Stand-alone interfaces/abstract classes that do not have any derived abstractions

Refactoring Unutilized Abstraction smell

- “Assign single and meaningful responsibility”
- Remove the “unutilized class or interface” especially if it has been created for an imagined need (i.e. speculatively)

Caution – Class libraries and frameworks usually provide extension points in the form of abstract classes or interfaces and may appear to be unused within the library or framework

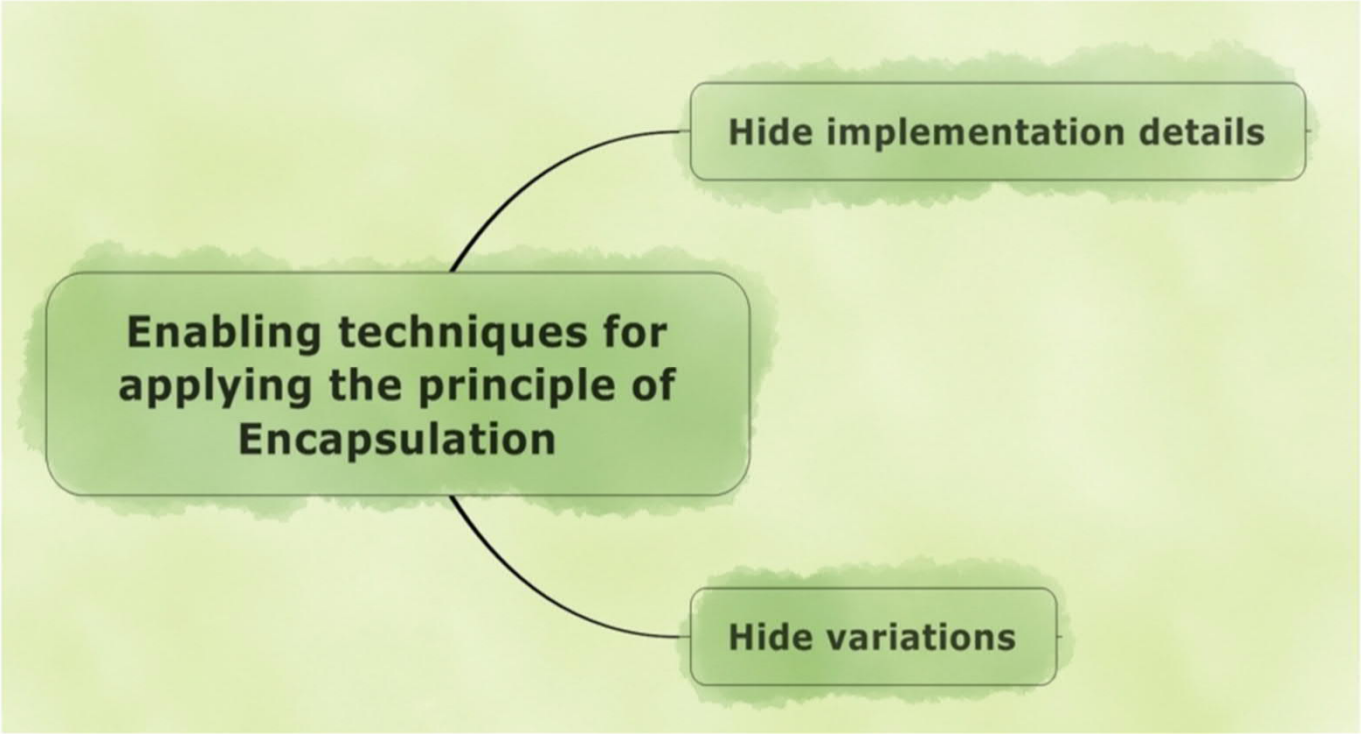


Outline

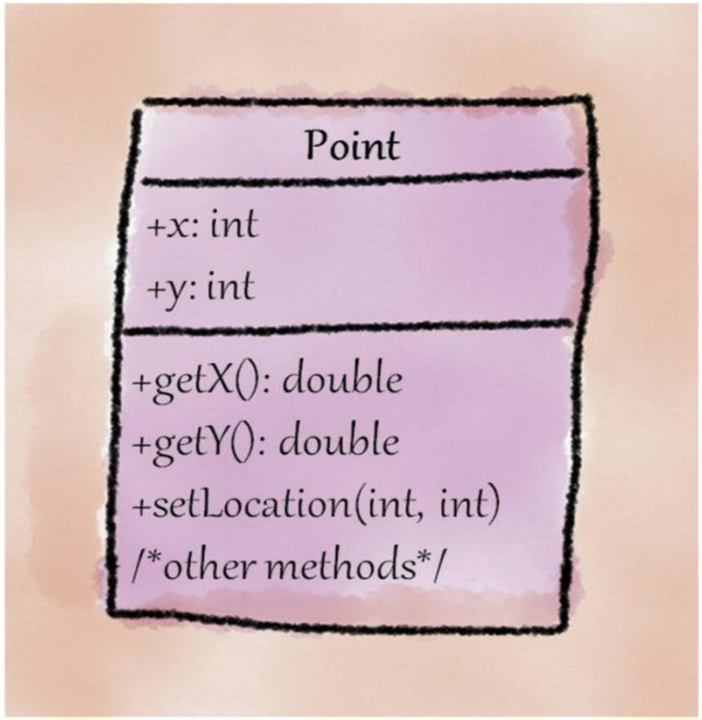
- Introduction
- Abstraction smells
- Encapsulation smells
- Modularization smells
- Hierarchy smells
- Refactoring design smells in practice

The principle of encapsulation

Enabling techniques for encapsulation



What's that smell?

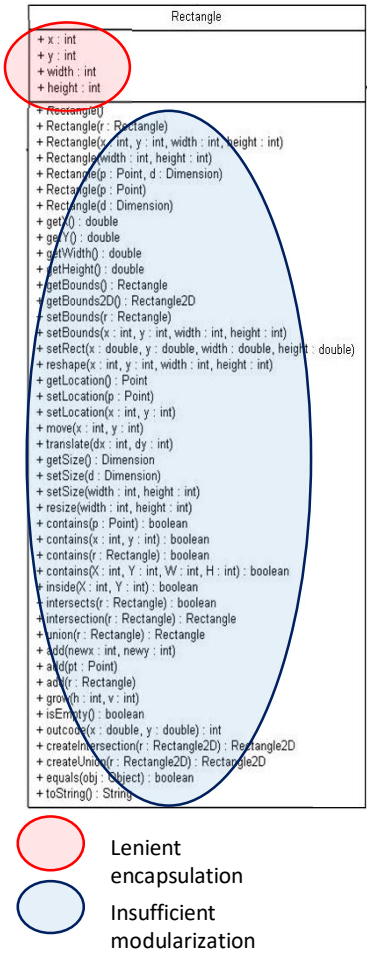


Deficient Encapsulation

This design smell occurs when the declared accessibility of one or more members of a class is more permissive than actually required.

(An abstraction should expose only the interface to the clients and hide the implementation details.)

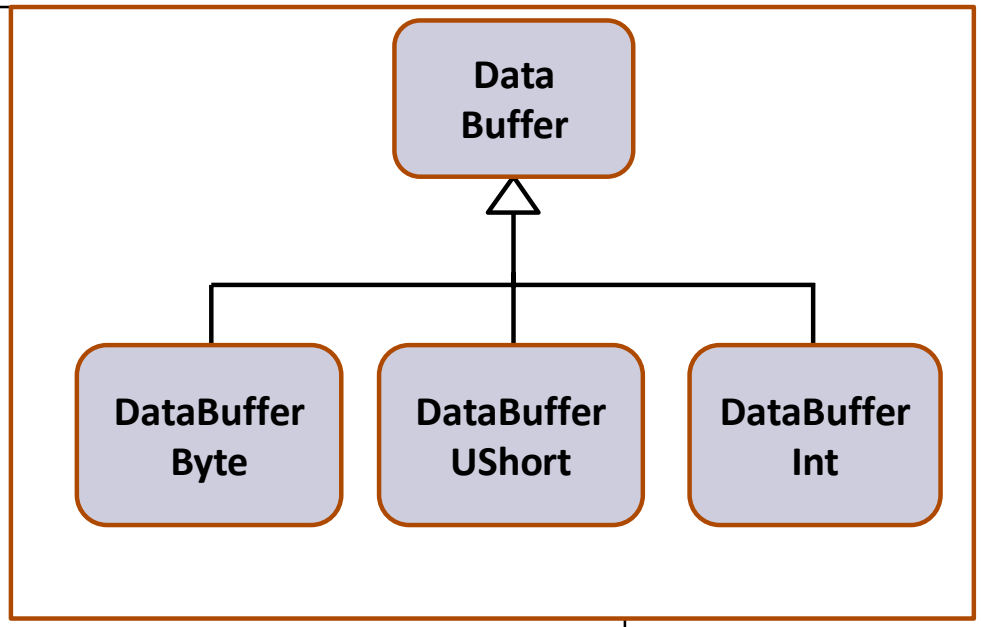
Deficient Encapsulation – Example & Refactoring



- The java.awt.Rectangle class has 4 public fields: x, y, width, and height *in addition to* public getter and setter methods for these fields!
  - Smell since the accessibility of its members is more permissive (i.e., are declared public) than actually required (i.e., they are required to be declared private)
- Refactoring suggestion: Make all data members private

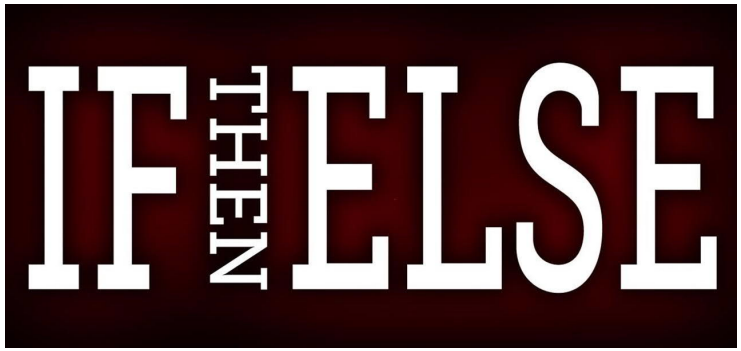
What's that smell?

```
switch (transferType) {
        case DataBuffer.TYPE_BYTE:
                byte bdata[] = (byte[])inData;
                pixel = bdata[o] & oxff;
                length = bdata.length;
                break;
        case DataBuffer.TYPE_USHORT:
                short sdata[] = (short[])inData;
                pixel = sdata[o] & oxffff;
                length = sdata.length;
                break;
        case DataBuffer.TYPE_INT:
                int idata[] = (int[])inData;
                pixel = idata[o];
                length = idata.length;
                break;
        default:
                throw new UnsupportedOperationException("This method has not been "+
                "implemented for transferType " + transferType);
}
```



Unexploited Encapsulation

This smell arises when client code uses explicit type checks (using chained if-else or switch statements that check for the type of the object) instead of exploiting the variation in types *already* encapsulated within a hierarchy.



Refactoring Unexploited Encapsulation smell

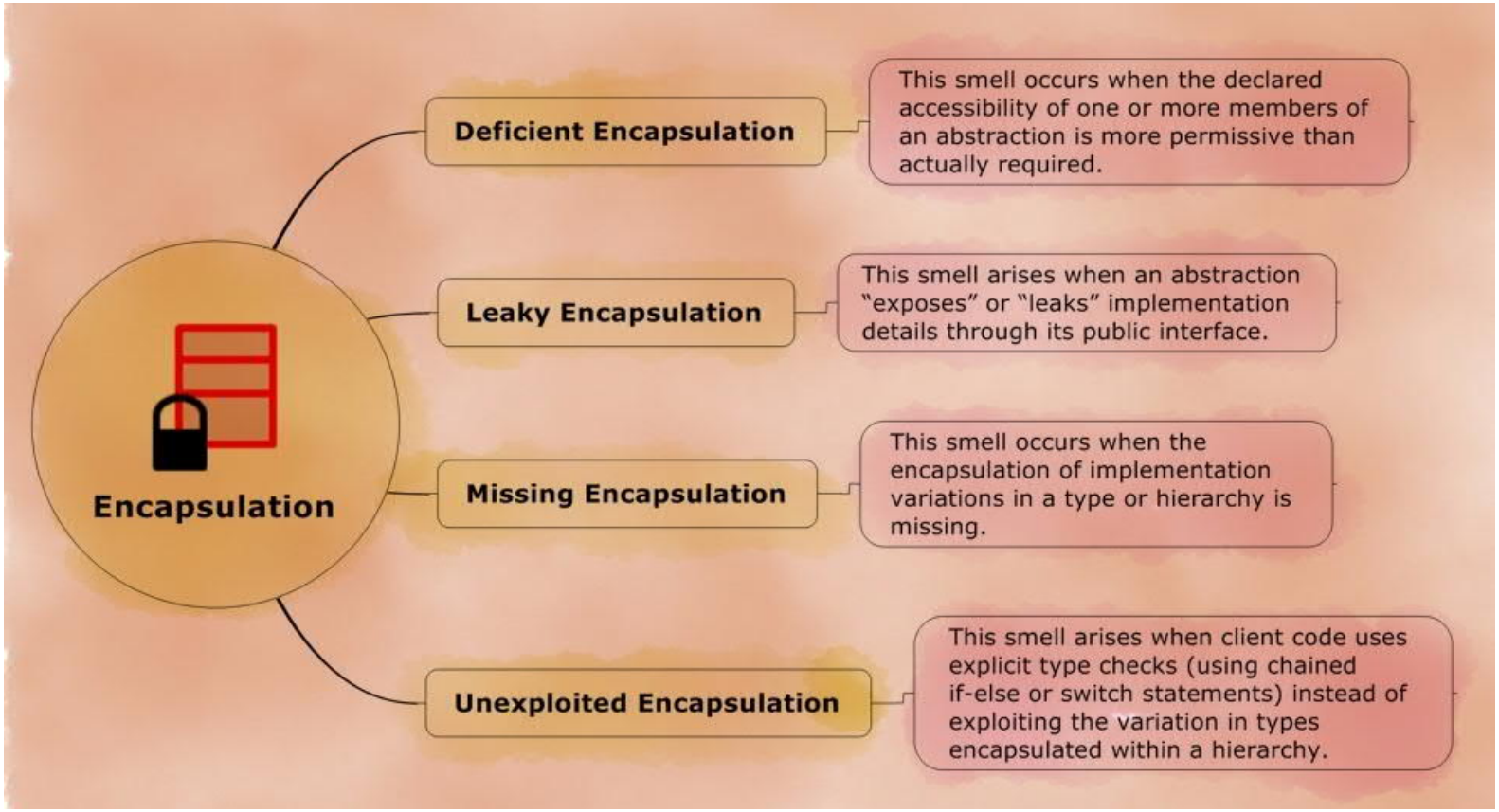
protected int transferType;

protected DataBuffer dataBuffer;

```
switch (transferType) {
        case DataBuffer.TYPE_BYTE:
                byte bdata[] = (byte[])inData;
                pixel = bdata[o] & oxff;
                length = bdata.length;
                break;
        case DataBuffer.TYPE_USHORT:
                short sdata[] = (short[])inData;
                pixel = sdata[o] & oxffff;
                length = sdata.length;
                break;
        case DataBuffer.TYPE_INT:
                int idata[] = (int[])inData;
                pixel = idata[o];
                length = idata.length;
                break;
        default:
                throw new UnsupportedOperationException("This
                method has not been "+ "implemented for
                transferType " + transferType);
}
```

pixel = dataBuffer.getPixel();

length = dataBuffer.getSize();

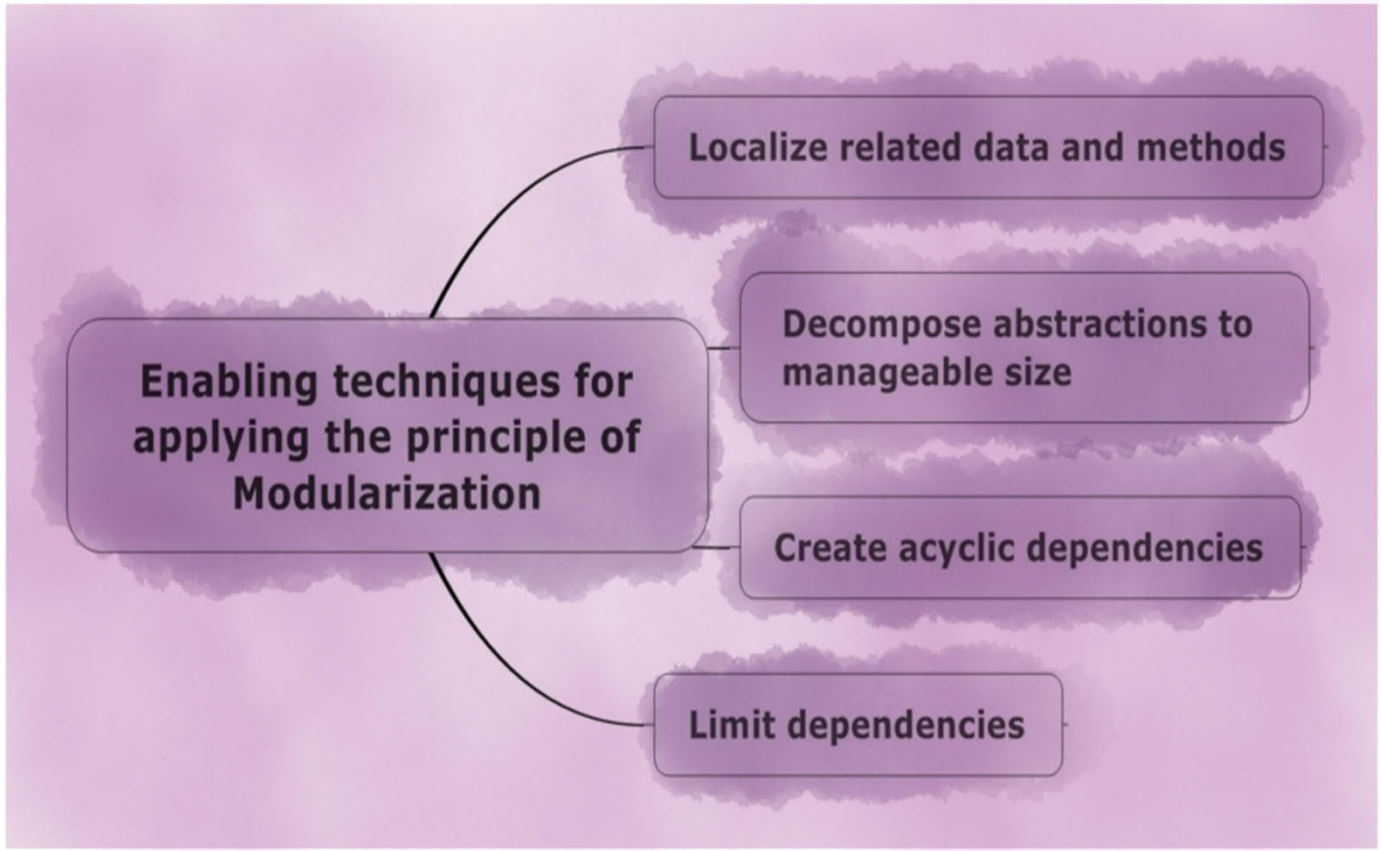


Outline

- Introduction
- Abstraction smells
- Encapsulation smells
- **Modularization smells**
- Hierarchy smells
- Refactoring design smells in practice

The principle of modularization

Enabling techniques for modularization

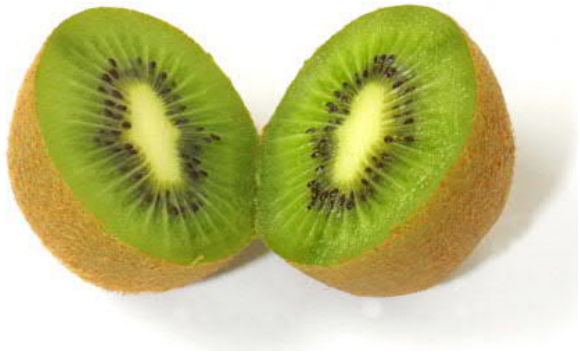


What's that smell?

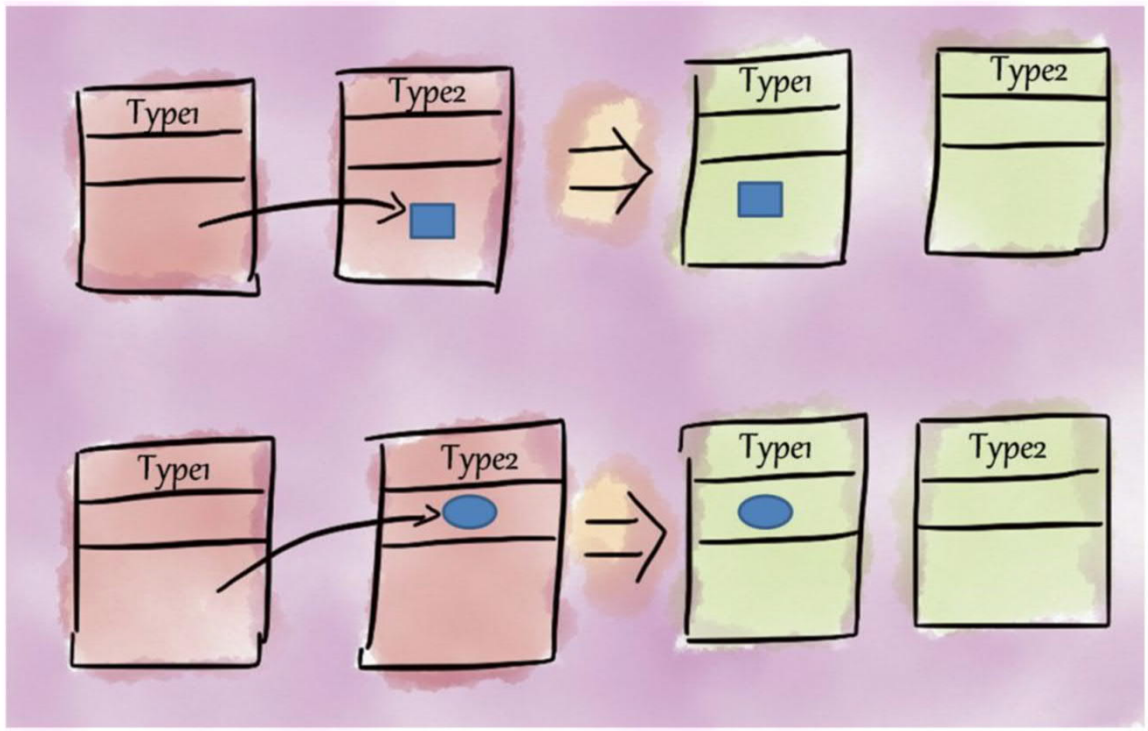
Broken modularization

This smell arises when members of an abstraction are broken and spread across multiple abstractions (when ideally they should have been localized into a single abstraction). Two forms of this smell:

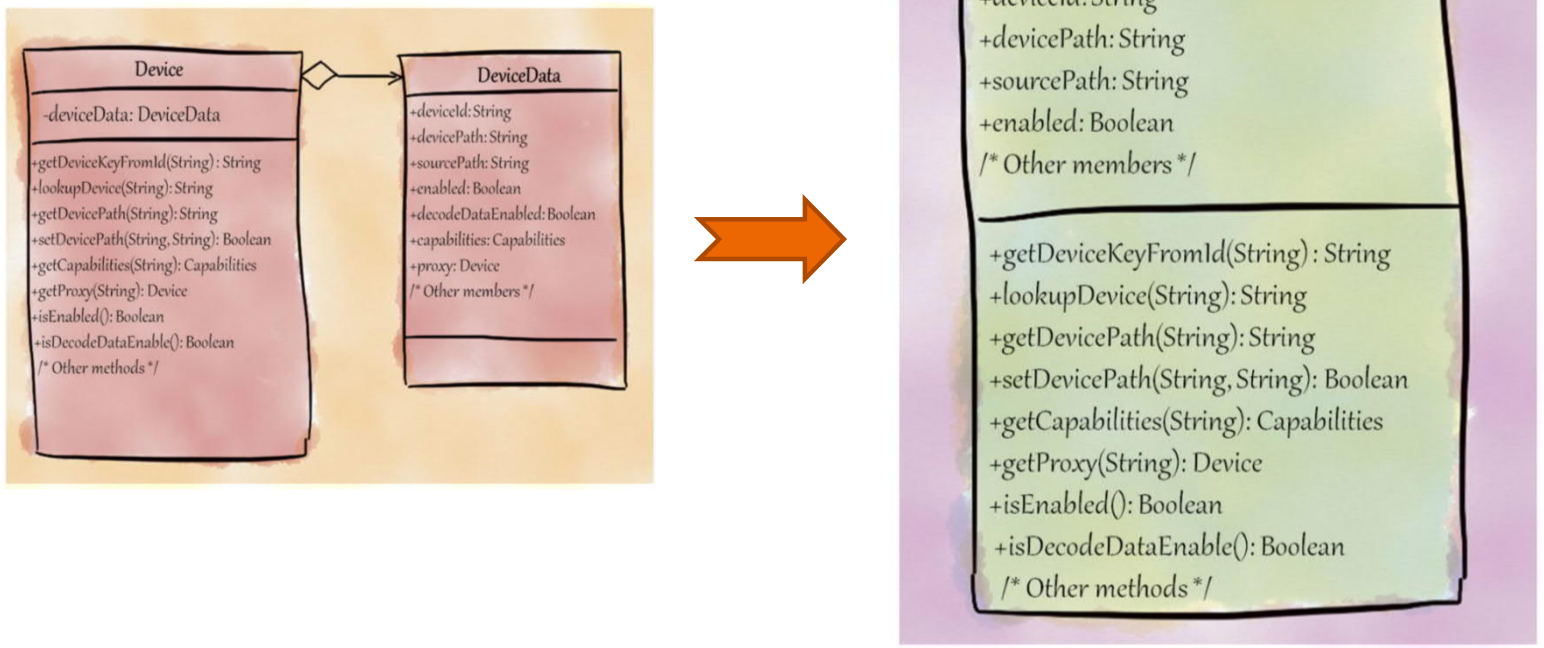
- Data and methods that ideally belong to an abstraction are split among two or more abstractions.
- Methods in a class that are interested in members of other classes.



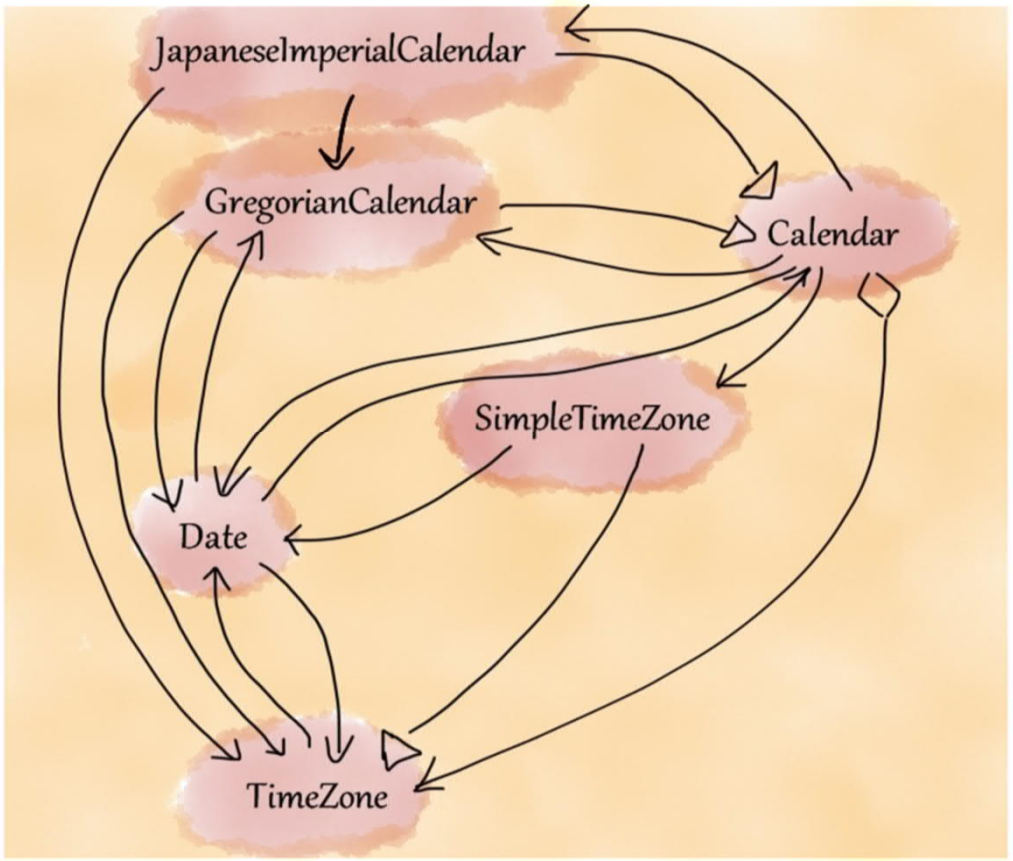
Suggested refactoring for broken modularization



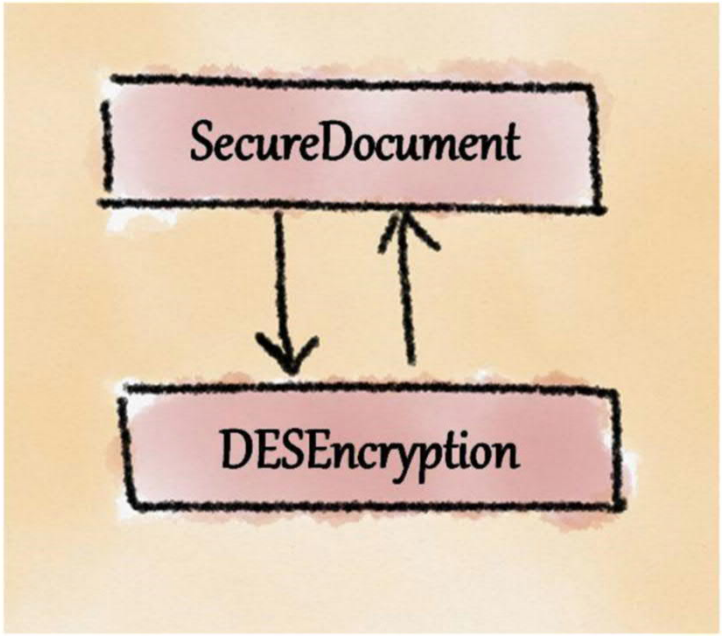
Refactoring for broken modularization smell



What's that smell?



What's that smell?

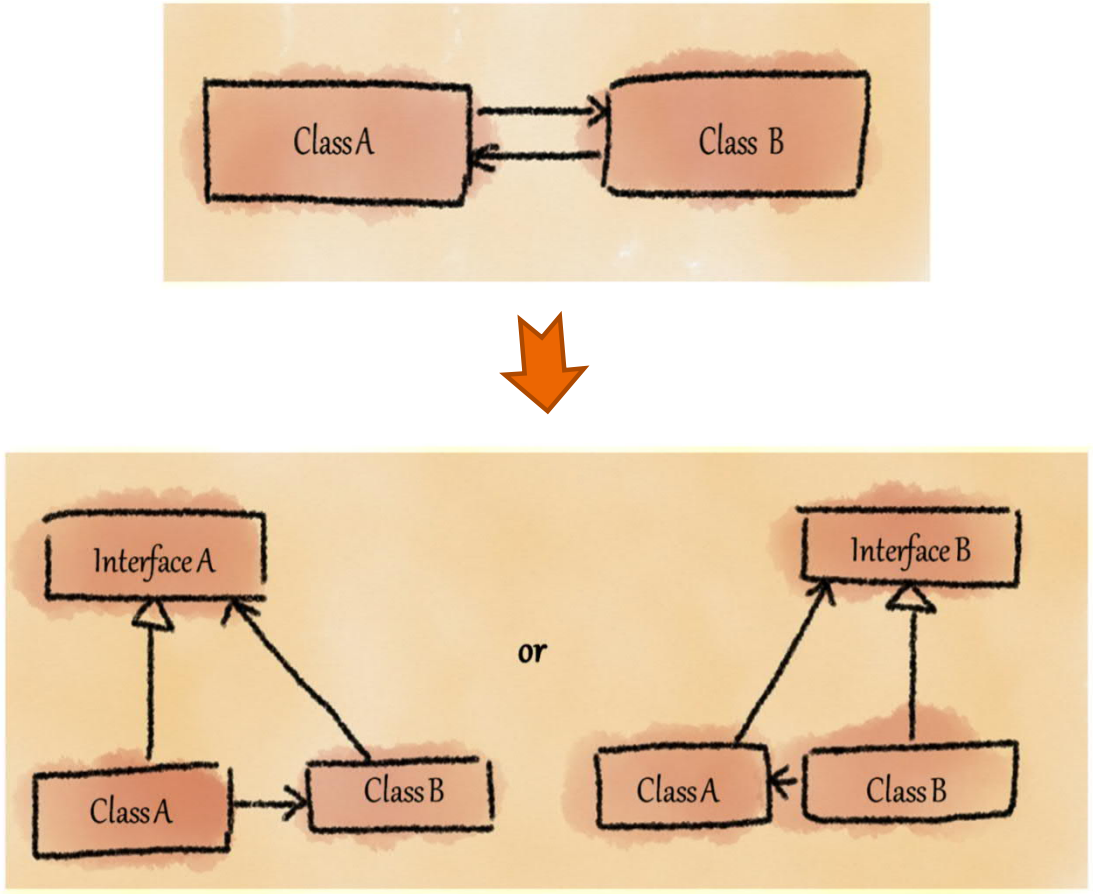


Cyclically-dependent modularization

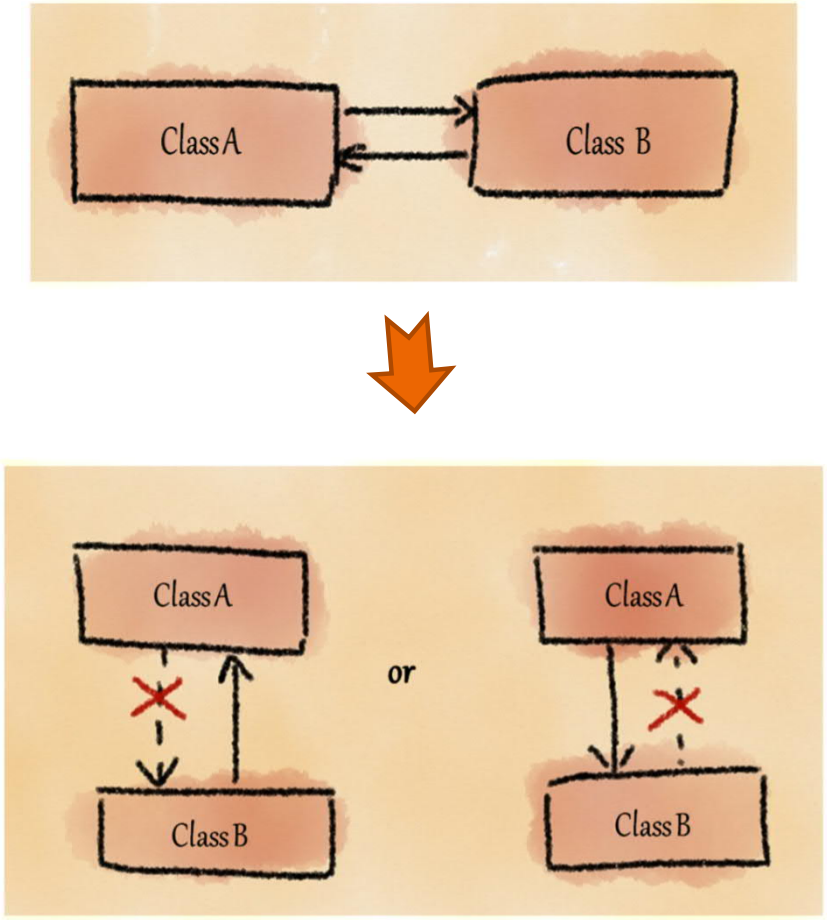
This smell arises when two or more class-level abstractions depend on each other directly or indirectly (creating a tight coupling among the abstractions).

(This smell is commonly known as “cyclic dependencies”)

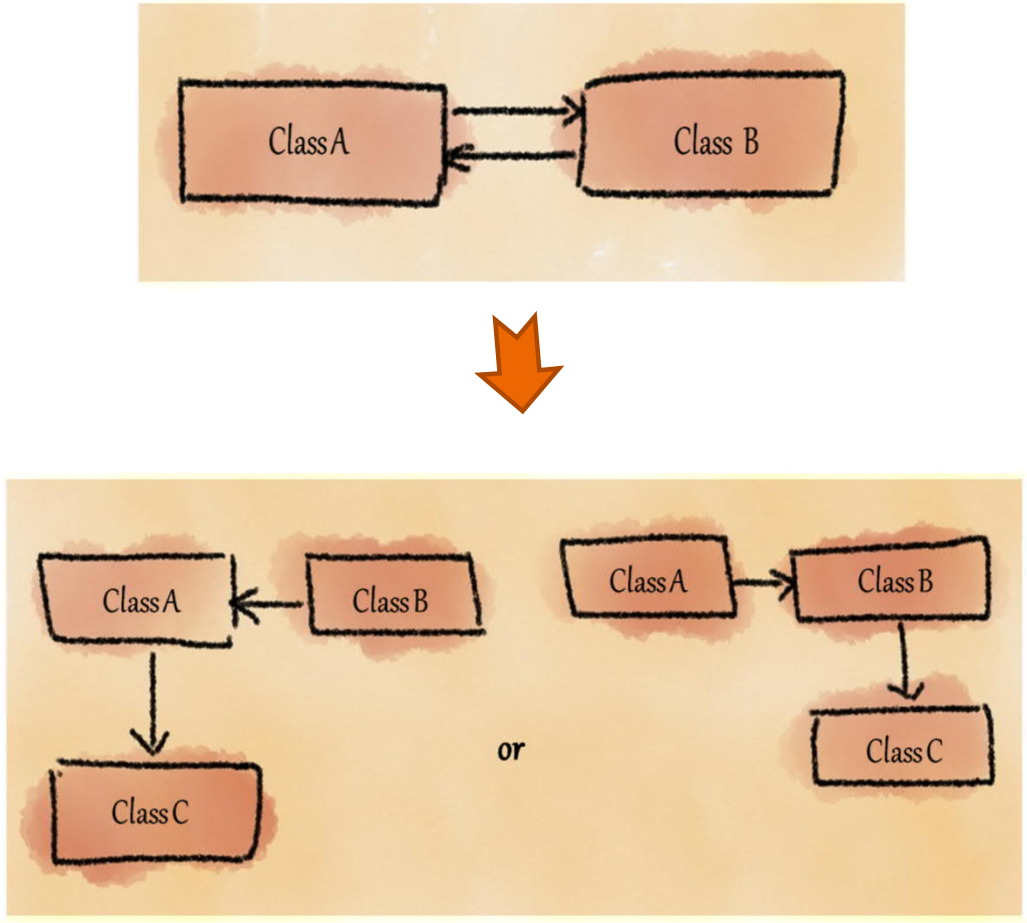
Refactoring cyclically-dependent modularization



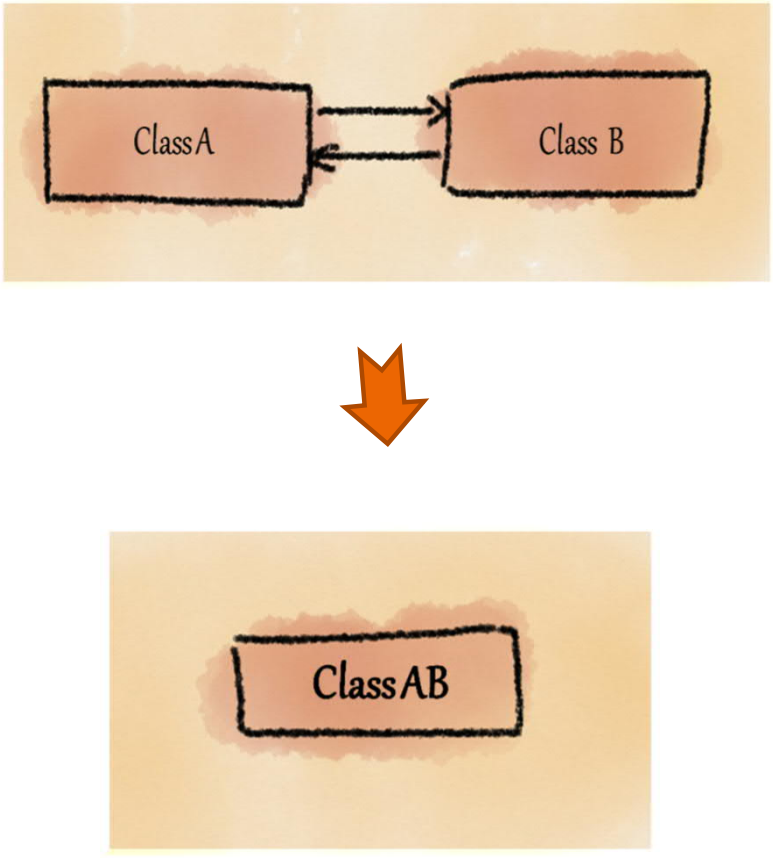
Refactoring cyclically-dependent modularization



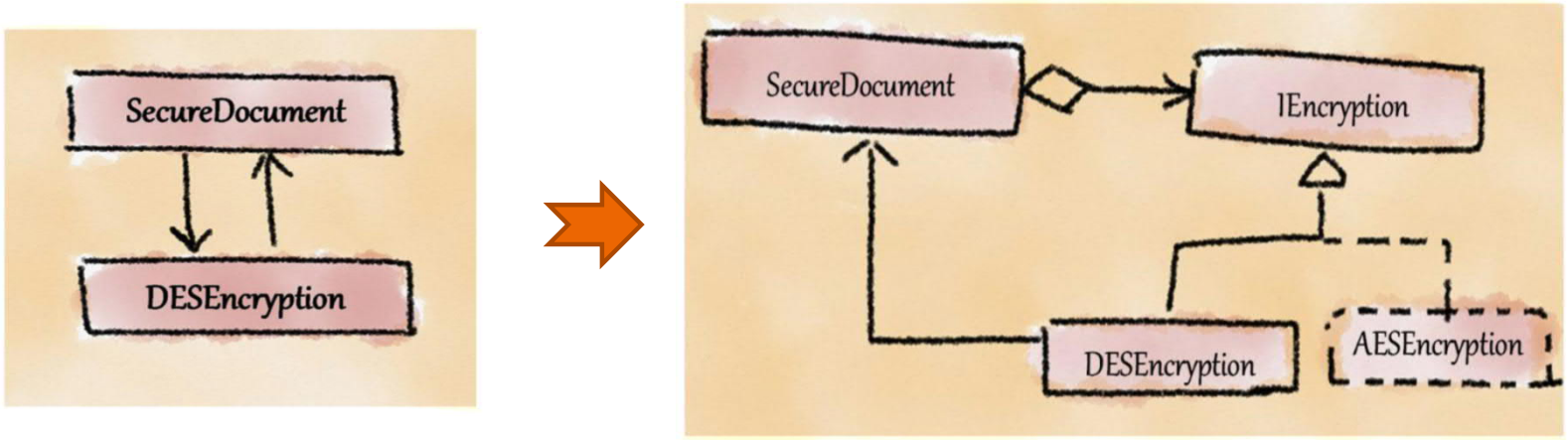
Refactoring cyclically-dependent modularization



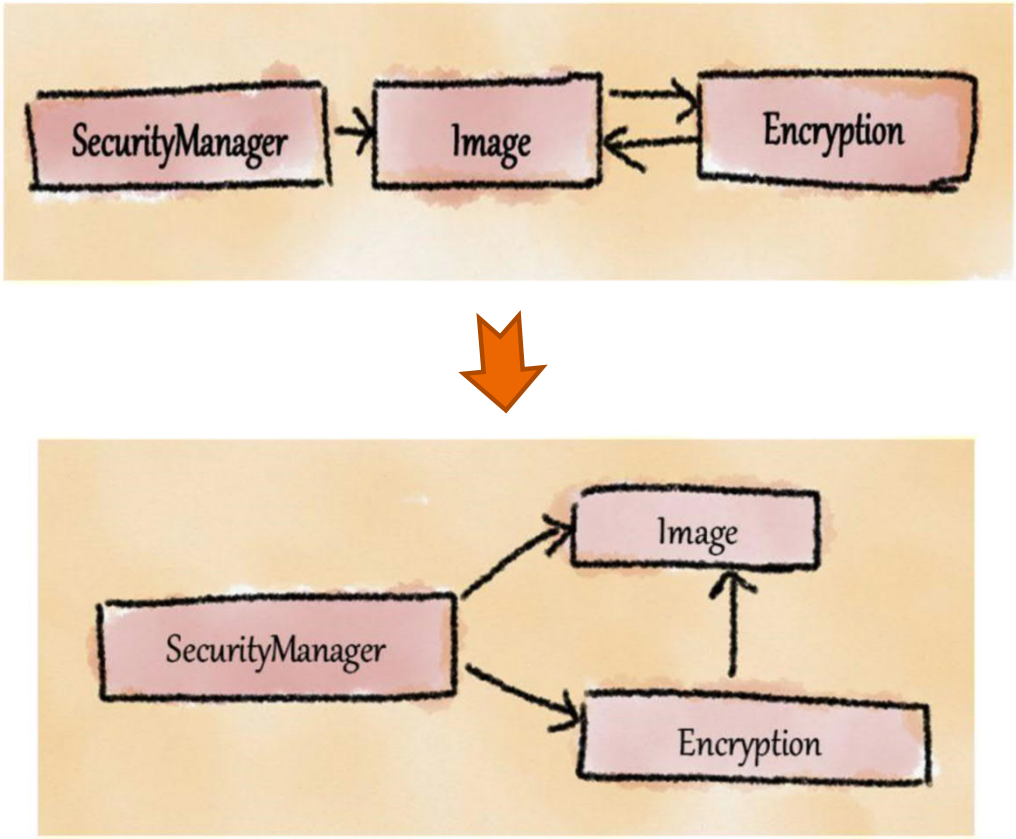
Refactoring cyclically-dependent modularization

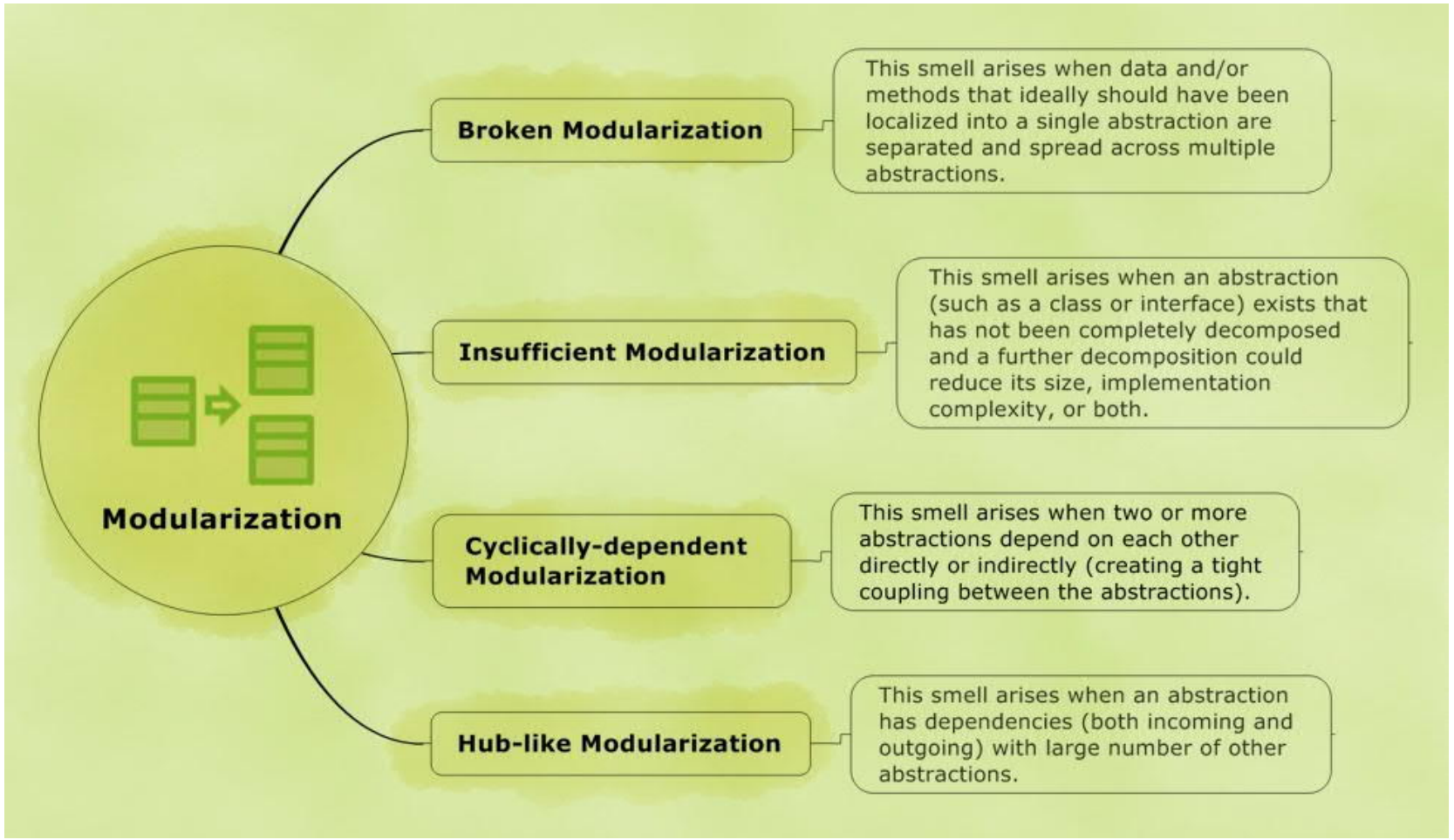


Refactoring cyclically-dependent modularization



Refactoring cyclically-dependent modularization

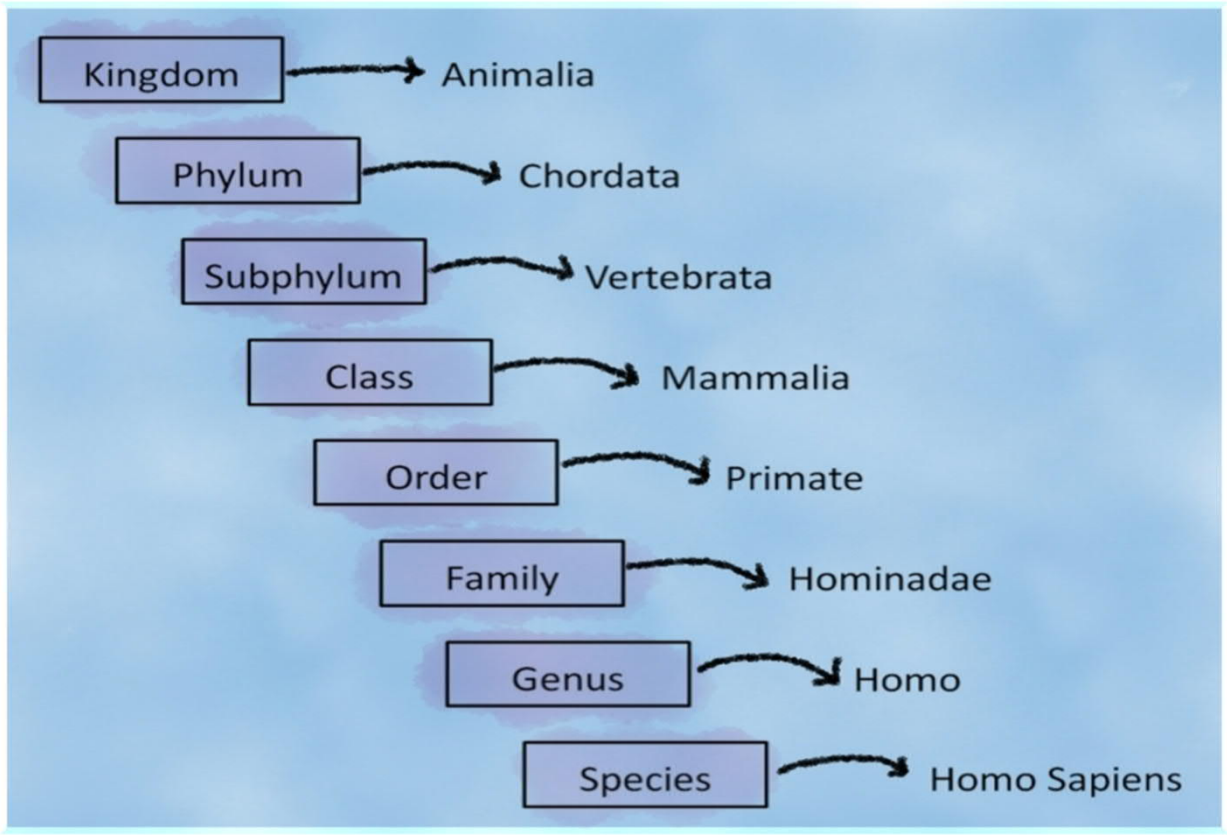




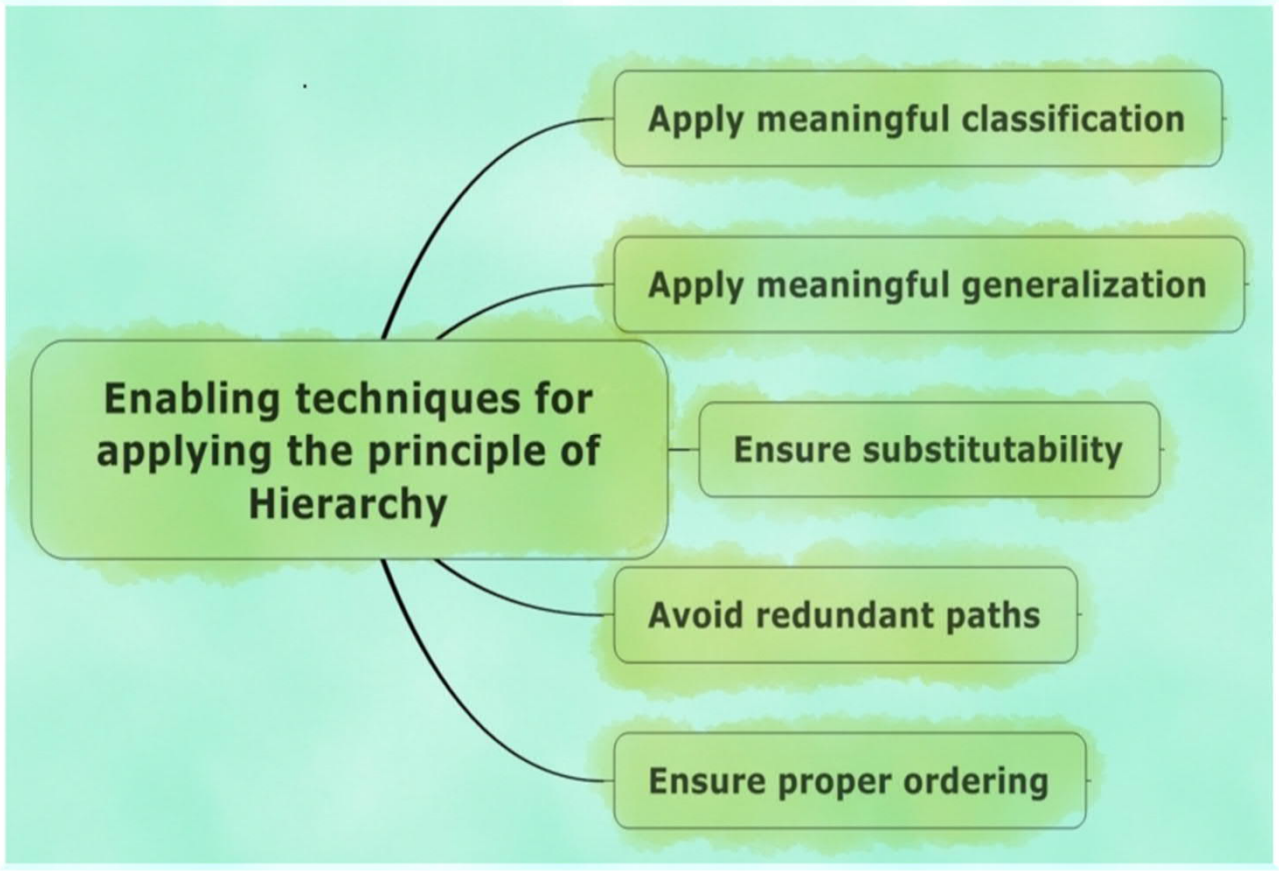
Outline

- Introduction
- Abstraction smells
- Encapsulation smells
- Modularization smells
- **Hierarchy smells**
- Refactoring design smells in practice

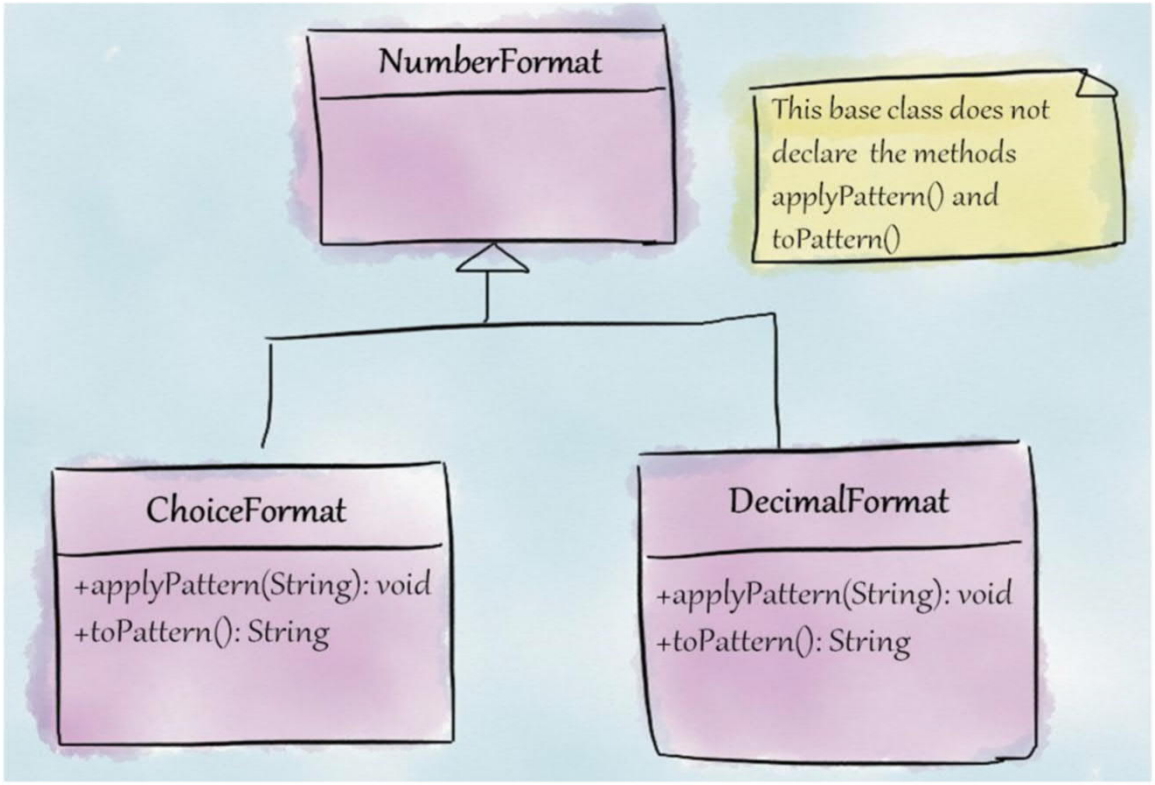
The principle of hierarchy



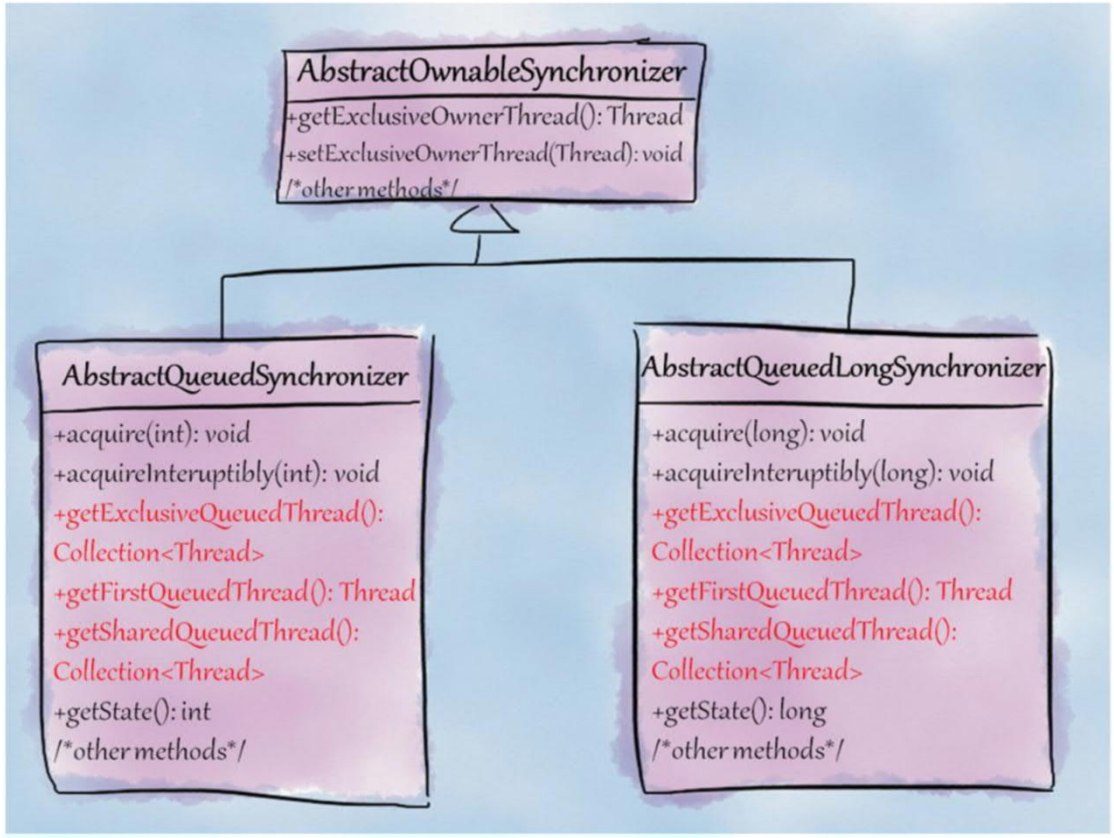
Enabling techniques for hierarchy



What's that smell?



What's that smell?

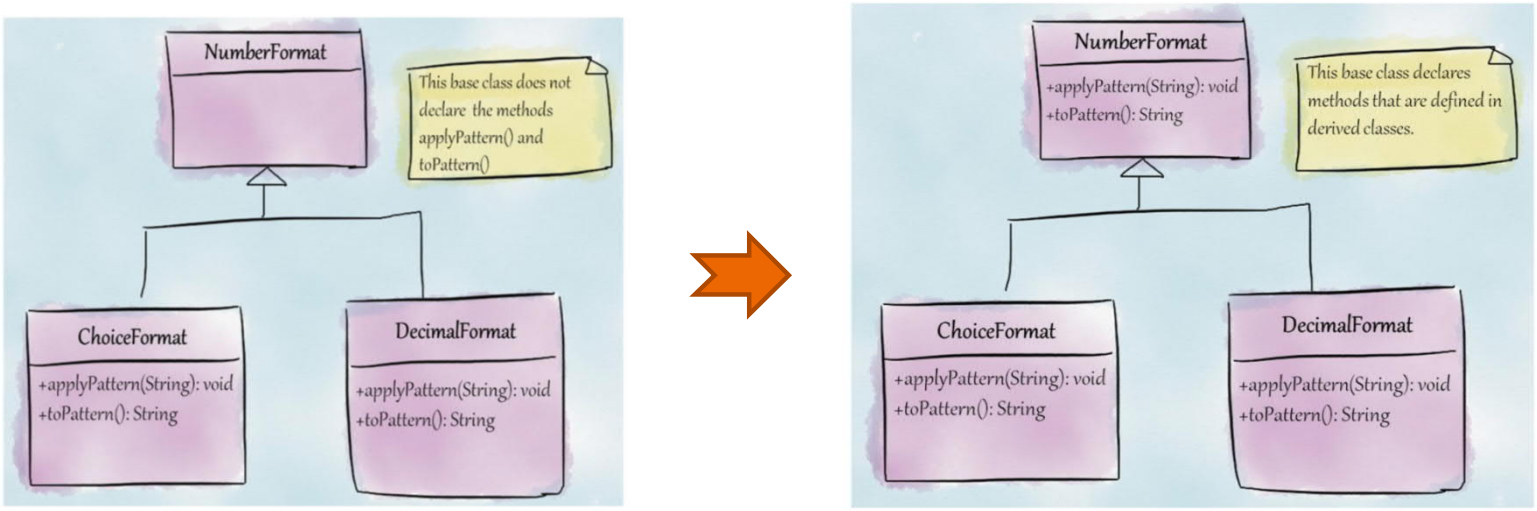


Unfactored hierarchy

This smell arises when there is unnecessary duplication among types in a hierarchy. Two forms of this smell:

- Duplication in sibling types
- Duplication in super and subtypes

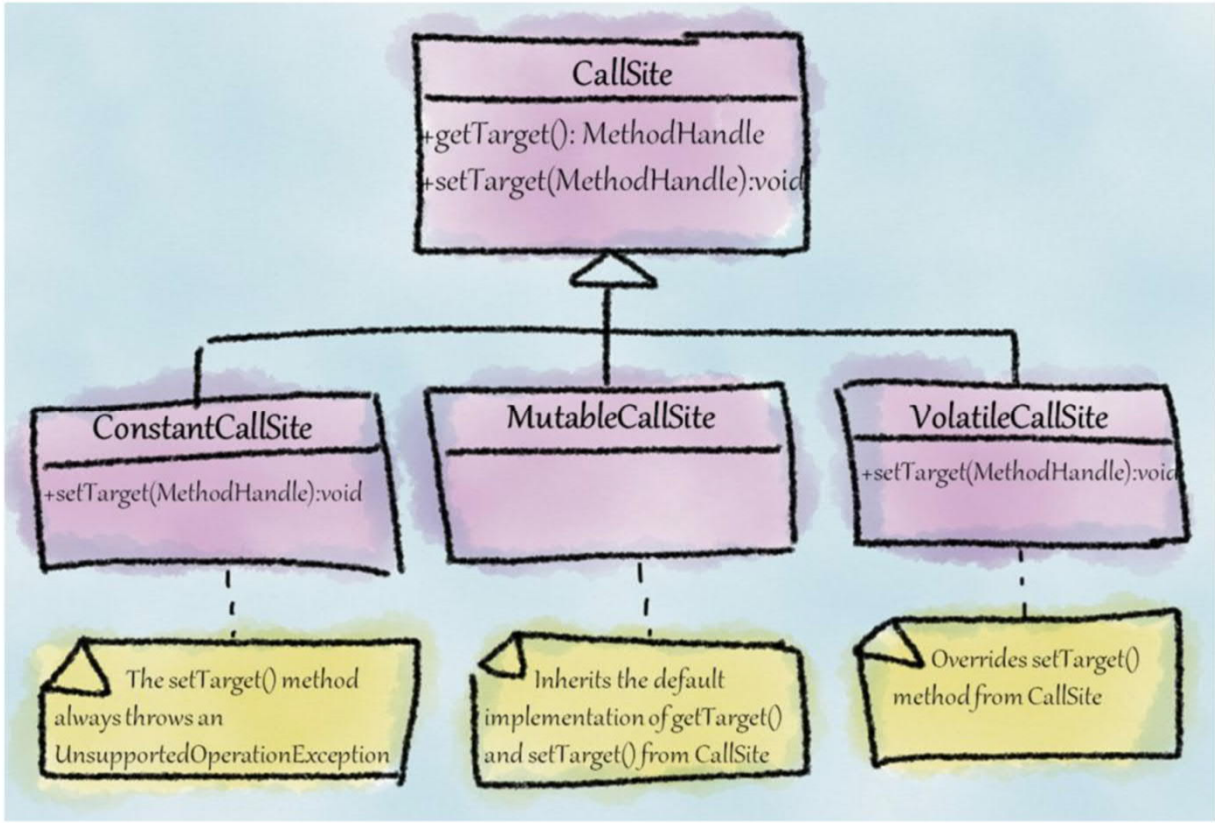
Refactoring for unfactored hierarchy



Caution:

- Inadequate language support to avoid duplication (e.g. lack of support for multiple inheritance)

What's that smell?

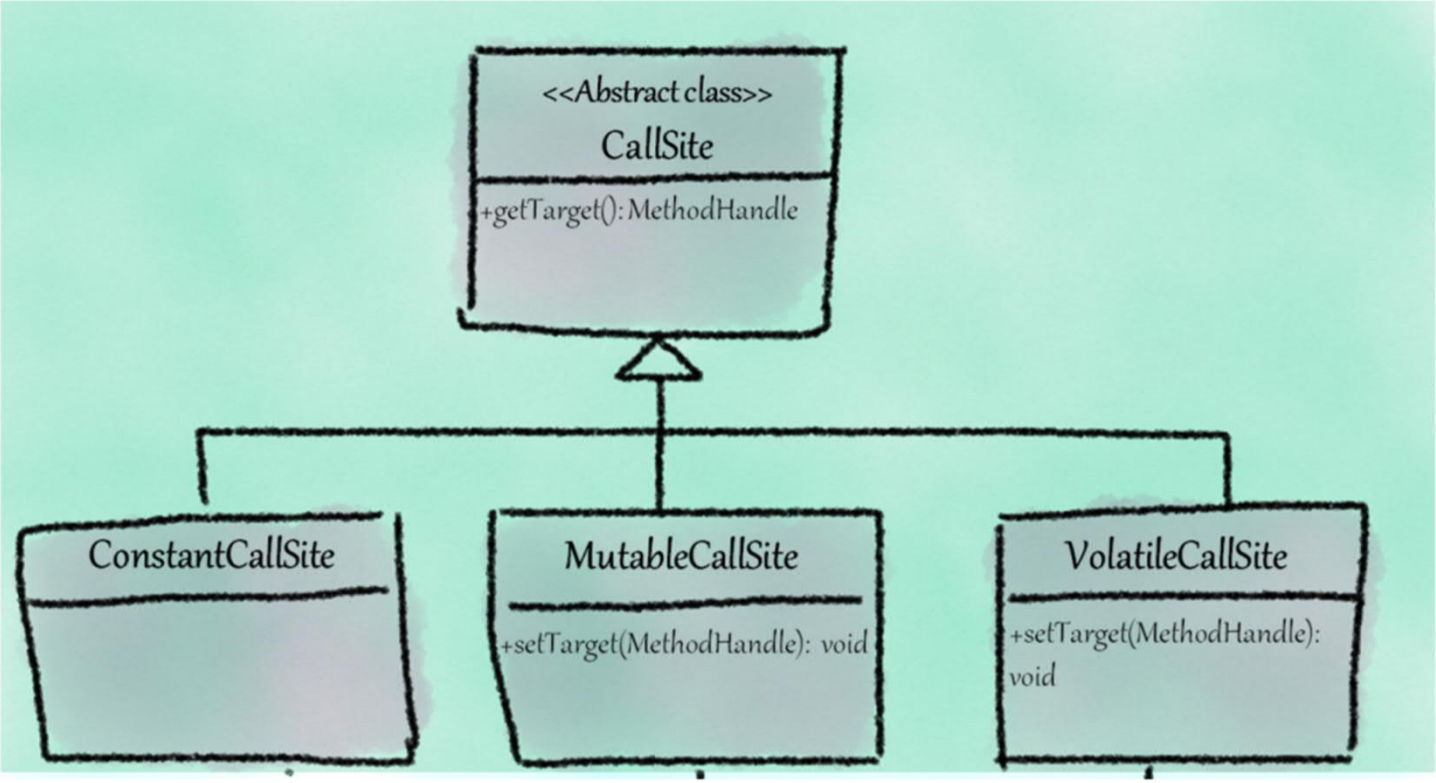


Rebellious hierarchy

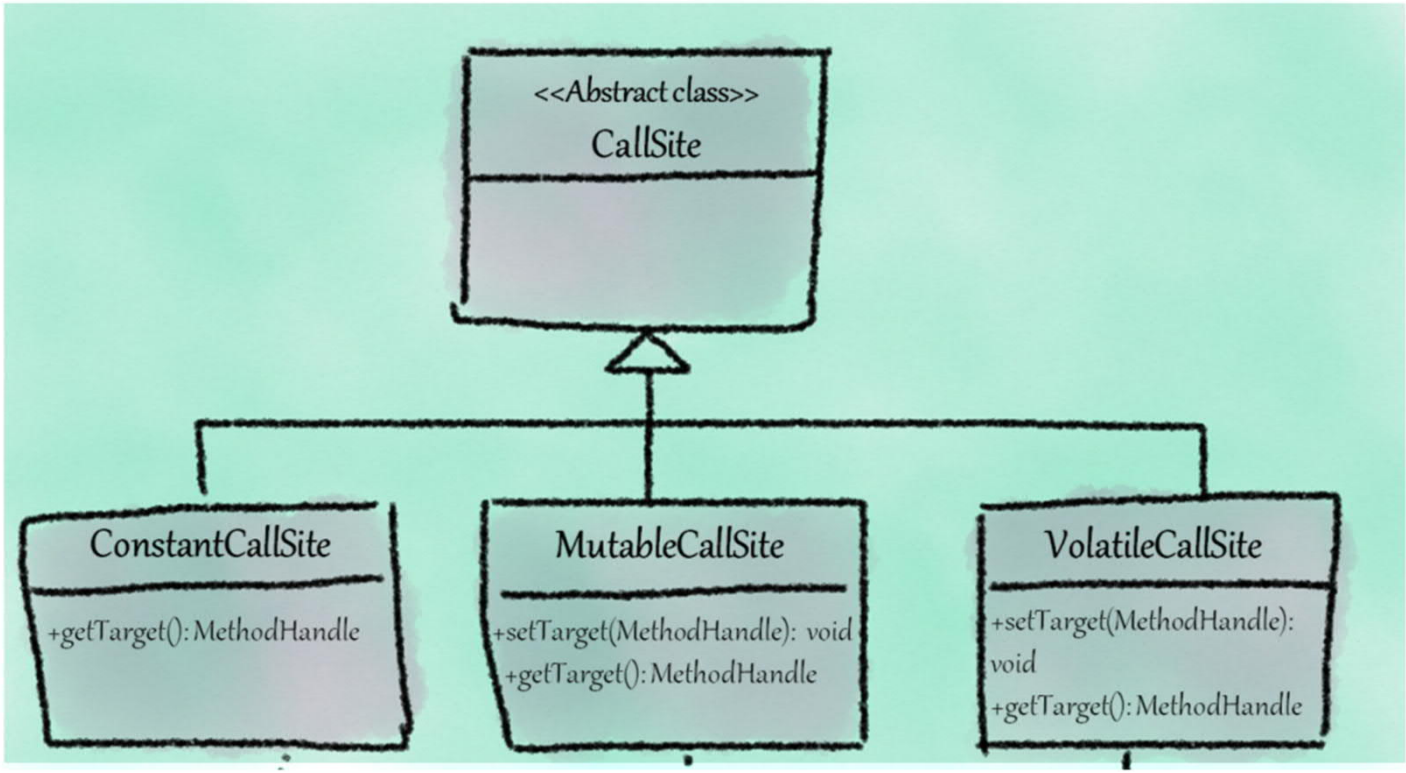
This smell arises when a subtype rejects the methods provided by its supertype(s):

- throw an exception,
- provide an empty (or NOP i.e., NO Operation) method
- provide a method definition that just prints “should not implement” message

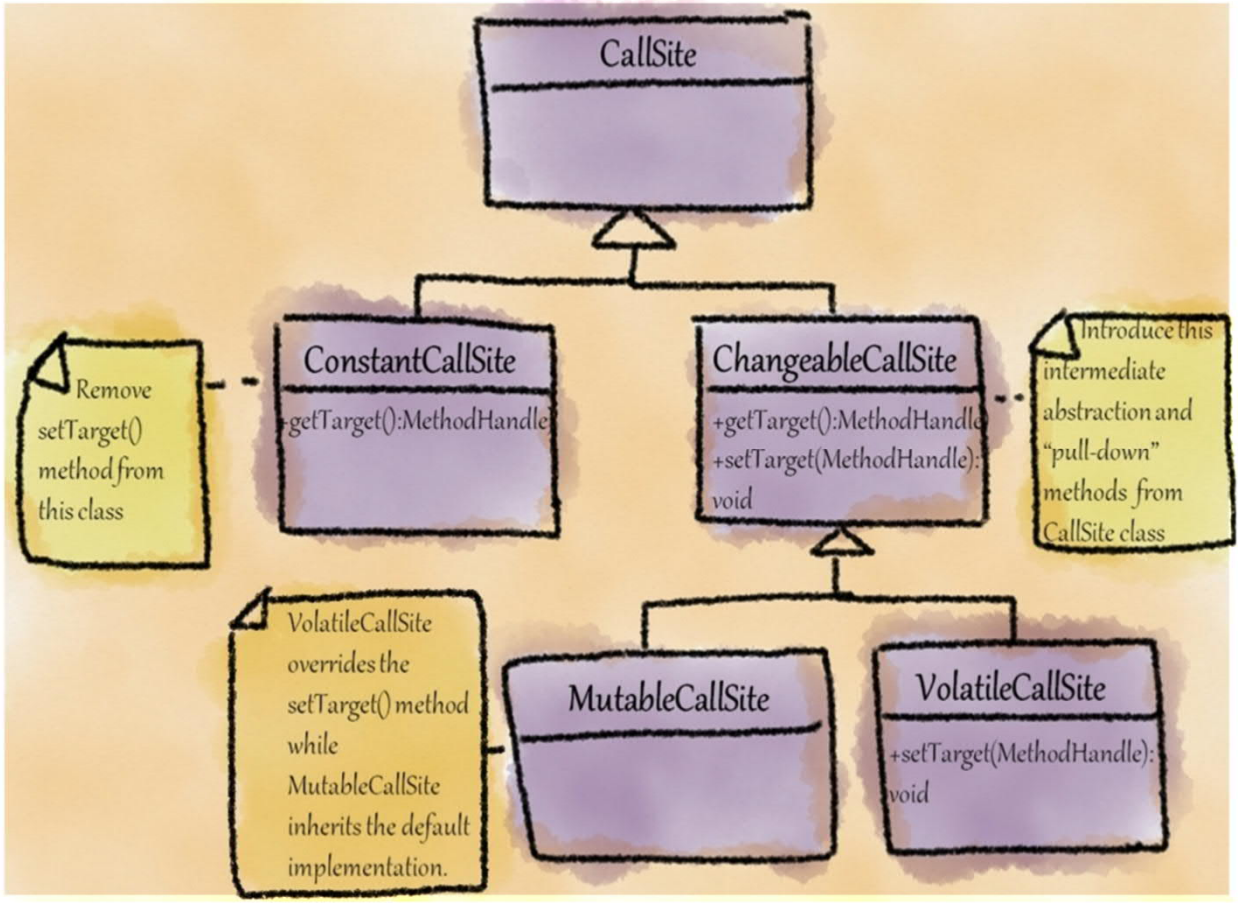
Suggested refactoring for rebellious hierarchy



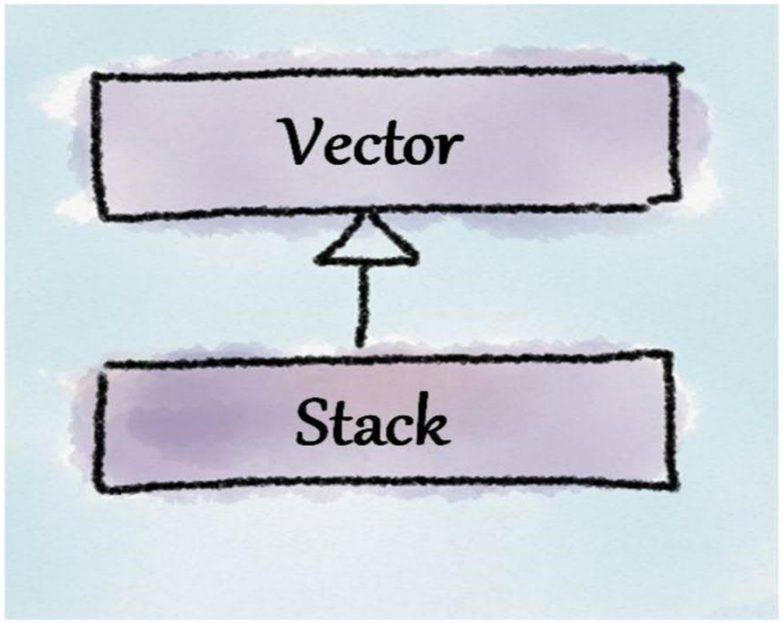
Suggested refactoring for rebellious hierarchy



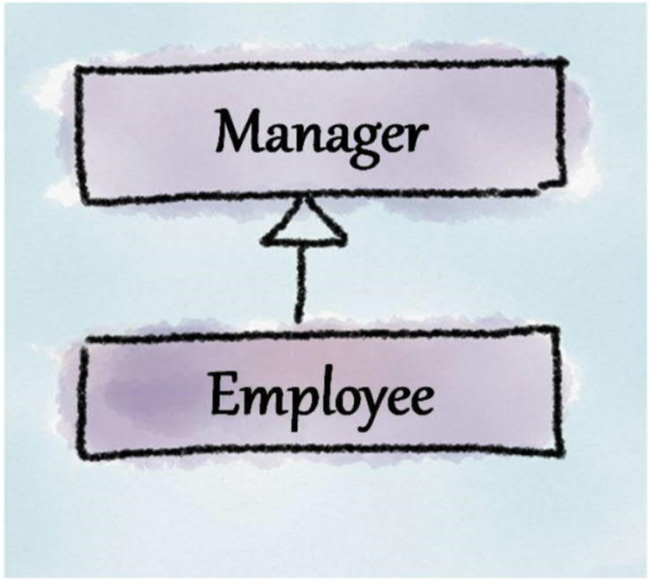
Suggested refactoring for rebellious hierarchy



What's that smell?



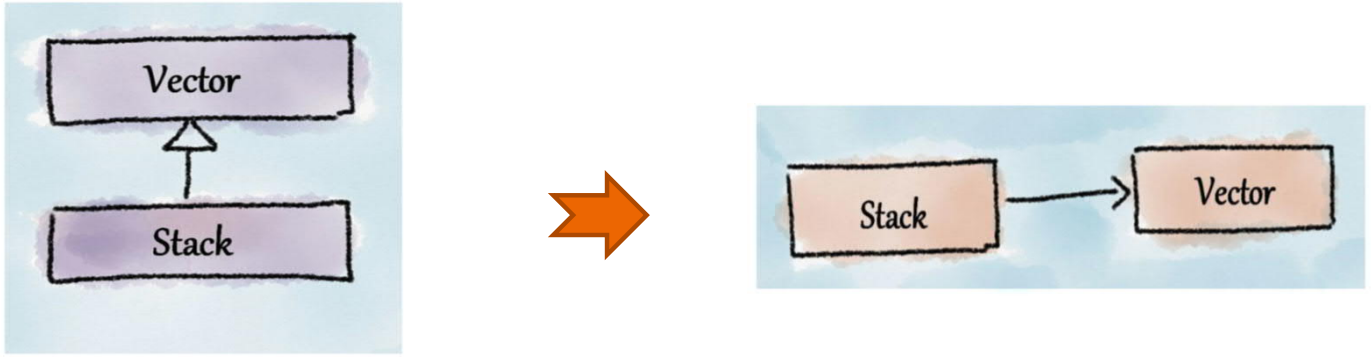
What's that smell?



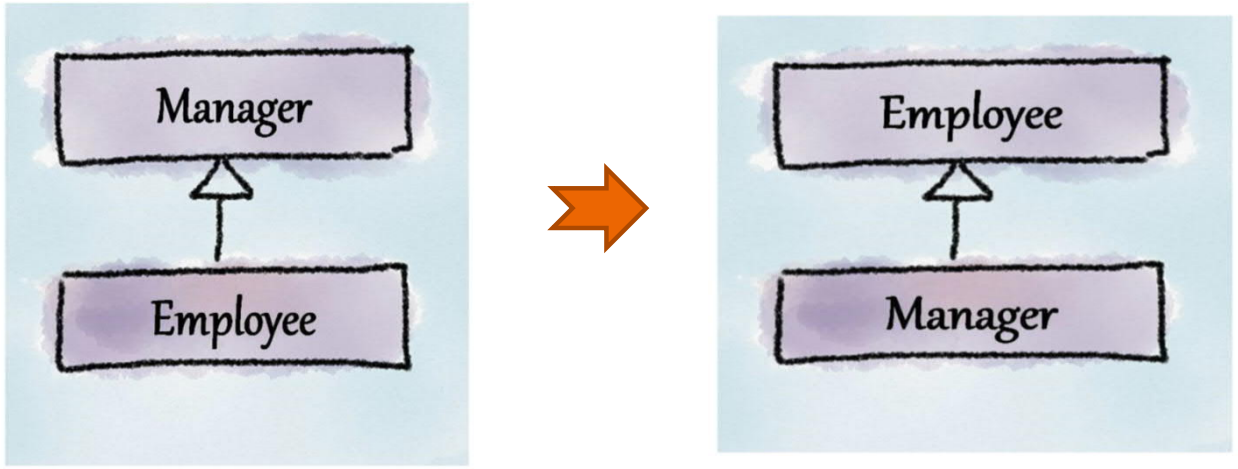
Broken hierarchy

This smell arises when a supertype and its subtype conceptually do not share an “IS-A” relationship resulting in broken substitutability.

Refactoring broken hierarchy



Refactoring broken hierarchy



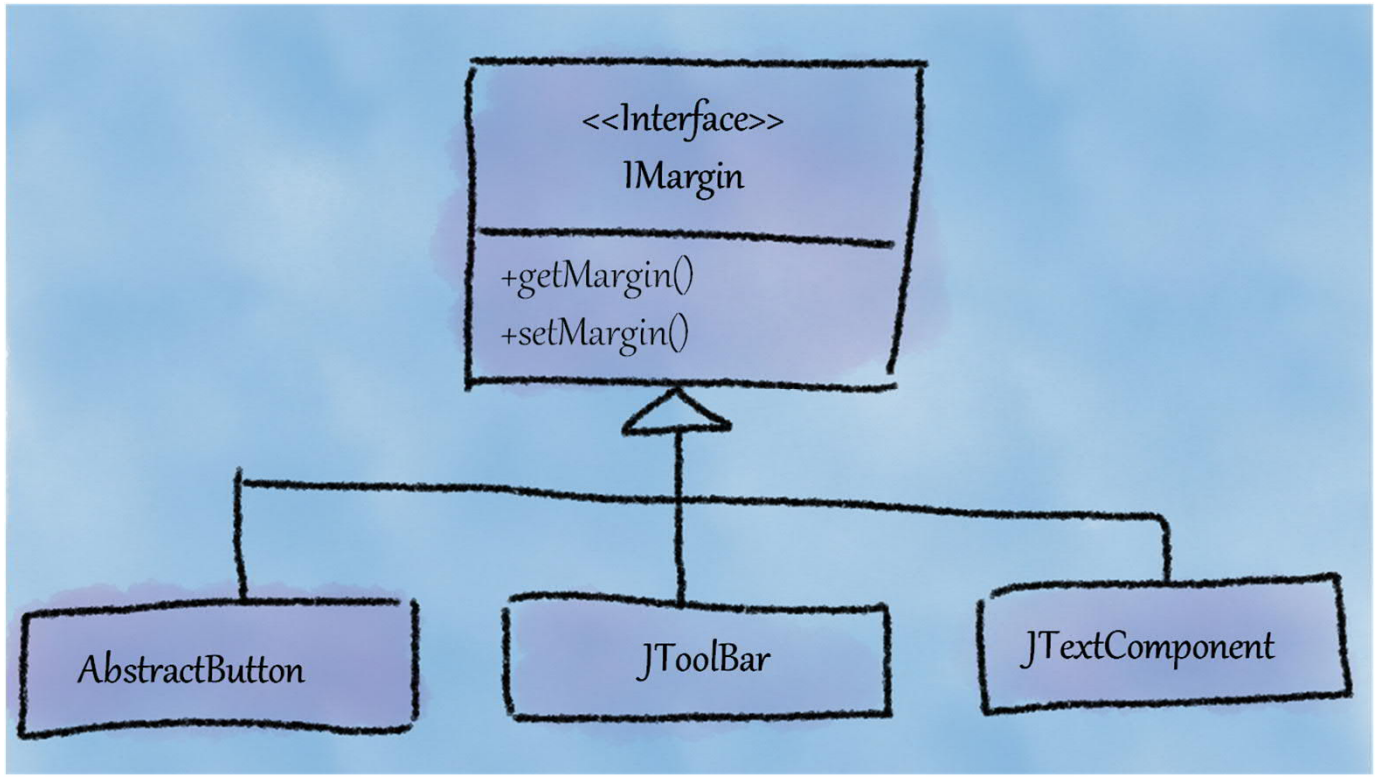
What's that smell?

```
public Insets getBorderInsets(Component c, Insets insets){
        Insets margin = null;
         // Ideally we'd have an interface defined for classes which
        // support margins (to avoid this hackery), but we've
        // decided against it for simplicity
        //
        if (c instanceof AbstractButton) {
                margin = ((AbstractButton)c).getMargin();
        } else if (c instanceof JToolBar) {
                margin = ((JToolBar)c).getMargin();
        } else if (c instanceof JTextComponent) {
                margin = ((JTextComponent)c).getMargin();
        }
        // rest of the code elided …
```

Missing hierarchy

This smell arises when an abstraction uses conditional logic to determine behavior where a hierarchy could have been formed to exploit variation in behavior.

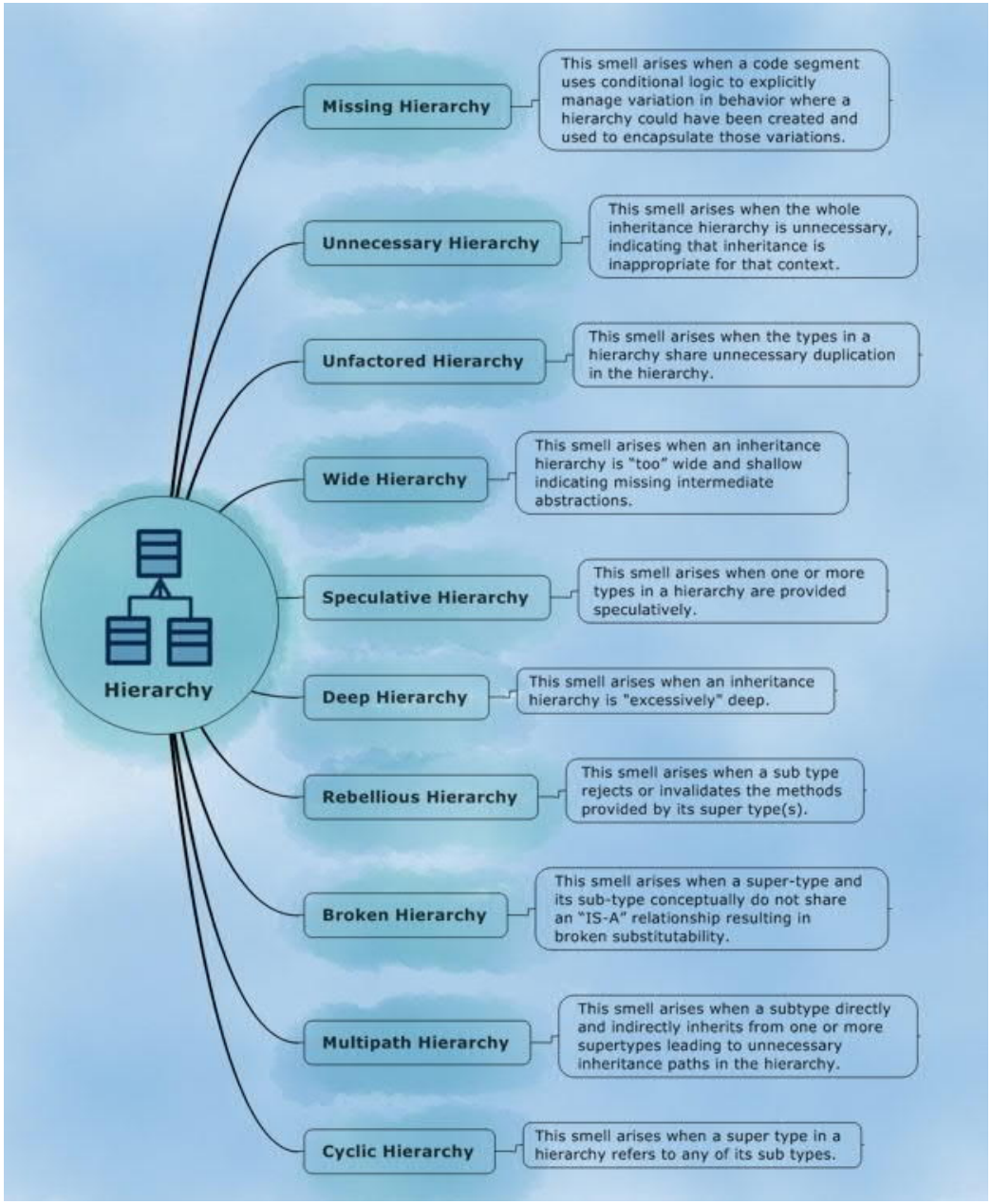
Refactoring for missing hierarchy



Refactoring for missing hierarchy

```
if (c instanceof AbstractButton) {
        margin = ((AbstractButton)c).getMargin();
} else if (c instanceof JToolBar) {
        margin = ((JToolBar)c).getMargin();
} else if (c instanceof JTextComponent) {
        margin = ((JTextComponent)c).getMargin();
}
```

```
margin = c.getMargin();
```



Key takeaways and conclusion

- Design quality is important for software (especially in case of large, complex, and/or reusable software)
- A great designer is one who understands the impact of design defects/smells and knows how to address them
- Design smells are candidates for refactoring
- Design smells can be viewed as violation of underlying fundamental design principles
- Some design smells can't be fixed
  - For such smells, the only way is to avoid introducing them in the first place!
- Context is important for smells
  - Given the liability of a smell, in certain contexts, a designer may make a *conscious decision* to live with that smell

Suggested Reading