DESIGN SMELLS

TEAM MEMBERS

20PW12 - Jeevickha K A

20PW13 - Jersha Heartly X

20PW14 - Karthickpranav S N

20PW15 - Karthik K M

20PW16 - Karthik Manian S

DESIGN SMELLS

- Hierarical Smells
  - Missing hierarchy
  - Unnecessary hierarchy
  - Unfactored hierarchy
  - Wide hierarchy
  - Speculative hierarchy
  - Deep hierarchy
  - Rebellious hierarchy
  - Broken hierarchy
  - Multipath hierarchy
  - Cyclic hierarchy

- Encapsulation Smells
  - Deficient encapsulation
  - Leaky encapsulation
  - Missing encapsulation
  - Unexploited encapsulation

- Modularization Smells
  - Broken modularization
  - Insufficient modularization
  - Cyclically-dependant modularization
  - Hub-like modularization

- Abstraction Smells
  - Missing abstraction
  - Imperative abstraction
  - Incomplete abstraction
  - Multifaceted abstraction
  - Unnecessary abstraction
  - Unutilized abstraction
  - Duplicate abstraction

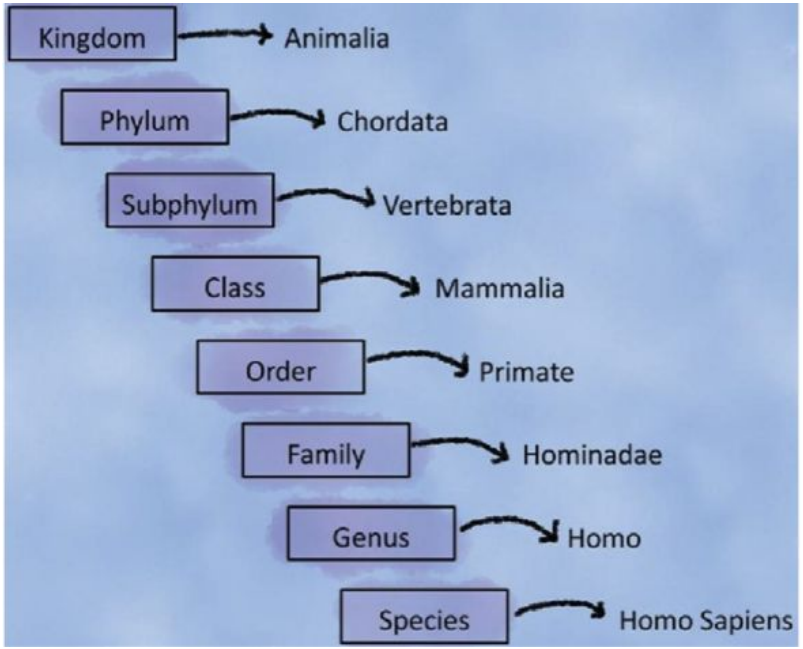
REFERENCE BOOK

HIERARCHICAL SMELLS

Taxonomy

Our planet houses more than 8.7 million species. How can we deal with this large-scale biodiversity?

Taxonomy is the science of *identifying, describing, classifying, and naming* living organisms based on their shared characteristics and evolutionary relationships.



So ?

- So, a hierarchical organization offers a powerful tool for dealing with complex diversity in real-world entities.
- It helps order the entities, deal with similarities and differences, and think in terms of generalizations and specializations.
- Modern software is extremely complex and requires a way to understand, reason about, and analyze the relationships between a large number of real-world entities.
- Object-oriented languages support two kinds of hierarchies: **type hierarchy** (is-a relationship) and **object hierarchy** (part-of relationship). For now, we concern ourselves with type hierarchies and use the term hierarchy synonymously with type hierarchy.

Enabling techniques - Principle of Hierarchy in software

1. Apply meaningful classification.
2. Apply meaningful generalization.
3. Ensure substitutability.
4. Avoid redundant paths.
5. Ensure proper ordering.

The smells described in this chapter map to violation of one or more of these enabling technique(s).

Design Smells and violated Enabling Techniques

Missing Hierarchy Unnecessary Hierarchy	Apply meaningful classification.
Unfactored Hierarchy Wide Hierarchy Speculative Hierarchy Deep Hierarchy	Apply meaningful generalization.
Rebellious Hierarchy Broken Hierarchy	Ensure substitutability.
Multipath Hierarchy	Avoid redundant paths.
Cyclic Hierarchy	Ensure proper ordering.

Missing Hierarchy

This smell arises when a code segment uses conditional logic (typically in conjunction with “tagged types”) to explicitly manage variation in behavior where a hierarchy could have been created and used to encapsulate those variations.

Rationale

Switch-based-on-type codes is one of the most well-known design smells. The commonality among those types is also left unexploited. In addition, the switch (or chained if-else) statements will be repeated in multiple places violating “Don’t Repeat Yourself” (DRY) principle and causing difficulties in maintenance.

Using conditional logic to handle variation in behavior clearly indicates that the enabling technique *“apply meaningful classification”* was not employed and as a result **a hierarchy is “missing”** in the design. Hence, we name this smell Missing Hierarchy.

Potential Causes

- **Misguided simplistic design** – Inexperienced designers often (wrongly) believe that *Switch-based-on-type-codes* results in a “straight-forward” or “simple” design.
- **Procedural approach to design** – In C-like procedural languages, the main way to handle variation in behavior is to use encoded types and conditional logic.
- **Overlooking inheritance as a design technique** – Cases where developers were unaware of the benefits that inheritance can bring to design when used correctly and end up relying on explicit conditional logic for handling variation instead of creating and using a hierarchy.

Example

```
public enum RequestType {
   GET, POST, PUT, DELETE;
}

public class RequestHandler {
   public void handleRequest(RequestType type, HttpServletRequest request, HttpServletResponse response) {
       switch (type) {
           case GET:
               // handle GET request
               break;
           case POST:
               // handle POST request
               break;
           case PUT:
               // handle PUT request
               break;
           case DELETE:
               // handle DELETE request
               break;
       }
   }
}
```

Suggested refactoring

Consider the following refactoring for this smell depending on the context:

- If two or more implementations within the condition checks have common method calls, relevant interfaces can be introduced to *abstract that common protocol*.
- If the code has conditional statements that can be transformed into classes, "extract hierarchy" refactoring can be applied to create a hierarchy of classes wherein each class represents a case in the condition check. This hierarchy can now be exploited via runtime polymorphism.

Refactored with hierarchy:

```
public abstract class RequestHandler {
   public abstract void handleRequest(HttpServletRequest request, HttpServletResponse response);
}

public class GetRequestHandler extends RequestHandler {
   public void handleRequest(HttpServletRequest request, HttpServletResponse response) {
       // handle GET request
   }
}

public class PostRequestHandler extends RequestHandler {
   public void handleRequest(HttpServletRequest request, HttpServletResponse response) {
       // handle POST request
   }
}

// class for handling PUT
// class for handling DELETE

// in the code that uses RequestHandler
RequestType type = getRequestType(); // or some other way to determine request type
RequestHandler handler = createHandlerForRequestType(type); // use a factory method to create the appropriate
handler
handler.handleRequest(request, response);
```

Impacted Quality Attributes

- Understandability
- Changeability and Extensibility
- Reusability
- Testability
- Reliability

Aliases

- Tag class
- Missing inheritance
- Collapsed type hierarchy
- Embedded features

Practical Considerations

- **Interacting with the external world**

```
private void computeFieldOffsets() throws InvalidClassException
{
    primDataSize = 0;
    numObjFields = 0;
    int firstObjIndex = -1;
    for (int i = 0; i < fields.length; i++) {
        ObjectStreamField f = fields[i];
        switch (f.getTypeCode()) {
            case 'Z':
            case 'B':
                f.setOffset(primDataSize++);
                break;
            case 'C':
            case 'S':
                f.setOffset(primDataSize);
                primDataSize += 2;
                break;
            // rest of the code elided to save space ...
            default:
                break;
        }
    }
}
```

It is difficult to avoid conditional logic when an application interacts with entities from the external world, as in the following cases:

- Reading a configuration file to alter the runtime configuration of an application.
- Instantiating objects using a factory based on the application requirement.

Unnecessary Hierarchy

This smell arises when the whole inheritance hierarchy is unnecessary, indicating that inheritance has been applied needlessly for the particular design context.

Rationale

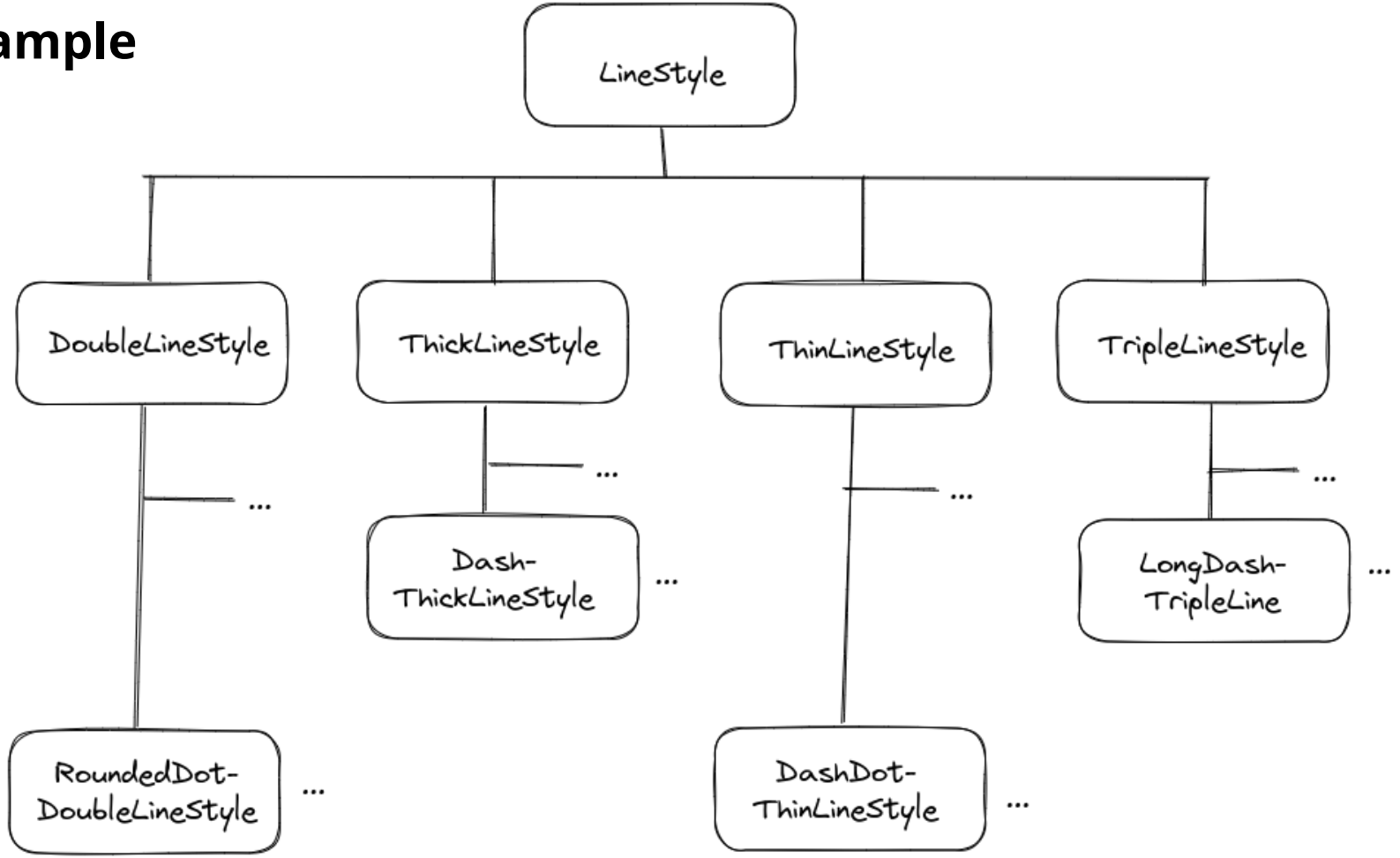
To effectively apply the principle of hierarchy, it's important to apply the enabling technique of *"apply meaningful classification"*. This means focusing on capturing the **commonalities and variations in behavior, rather than data.**

When this technique is violated, it can result in an "unnecessary" inheritance hierarchy that needlessly complicates the design. This is known as the Unnecessary Hierarchy smell.

Potential Causes

- **Subclassing instead of instantiating** – Assume that these different kinds of roses differ only in their color and not in their behavior. Hence, it will be inappropriate to create a Rose supertype with subtypes such as RedRose, PinkRose, and WhiteRose. Instead of creating subclasses, a better design would be to create objects of type Rose with an attribute named color that can take values such as red, pink, and white.
- **Taxonomy mania** – Some designers tend to overuse inheritance and sometimes force-fit it as a design solution in unwarranted contexts. Meyer calls this “taxonomy mania”. Such overuse of inheritance leads to unnecessary hierarchies in design.

Example

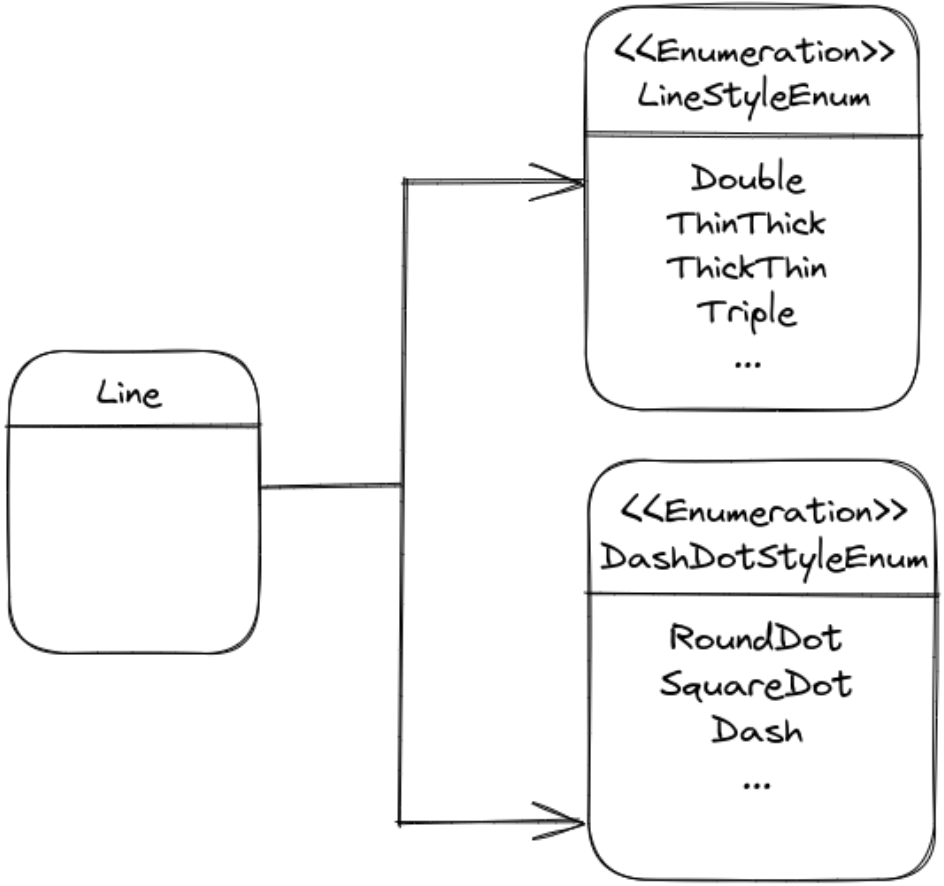

LineStyle
DoubleLineStyle
ThickLineStyle
ThinLineStyle
TripleLineStyle
...
...
...
...
Dash-
ThickLineStyle
...
LongDash-
TripleLine
...
RoundedDot-
DoubleLineStyle
...
DashDot-
ThinLineStyle
...

Suggested refactoring

Consider these refactoring strategies depending on the context:

- When instances of a type are instead modeled as subtypes of that type, remove the hierarchy and transform the subtypes into instances of that type.
- If an inheritance hierarchy is created instead of using a suitable language feature (such as an enumeration), then remove the Unnecessary Hierarchy and use that language feature instead.

Refactored hierarchy:



Impacted Quality Attributes

- Understandability
- Extensibility
- Testability

Aliases

- Taxomania
- Object classes

Practical Considerations

- None

Unfactored Hierarchy

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

- Duplication in sibling types
- Duplication in super and subtypes:
    - Unfactored interface
    - Unfactored implementation
    - Unfactored interface and implementation

Rationale

One of the motivations behind the usage of inheritance is to avoid redundancy by applying generalization, i.e., by factoring out the commonalities from types to a supertype in the inheritance hierarchy. This has two main advantages:

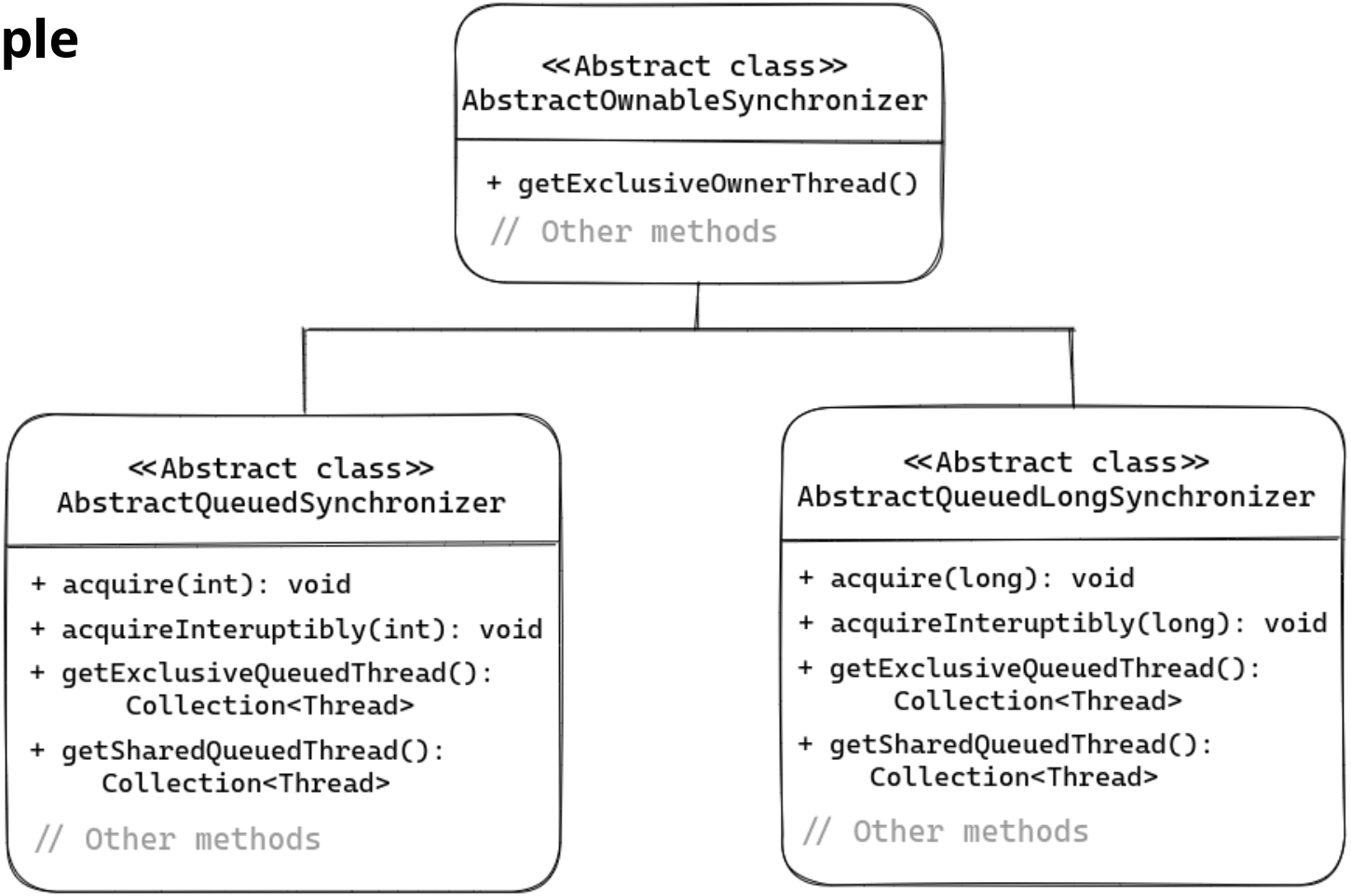
- Elevating the common interface in the hierarchy allows clients to depend on the common interface rather than concrete implementation details, enabling subtypes to change without affecting clients.
- Elevating the implementation to a supertype removes duplication across types in the hierarchy, avoiding unnecessary duplication.

This smell violates the Don't Repeat Yourself (DRY) principle and the enabling technique "*apply meaningful generalization.*" Since the **commonalities in the types are not factored out**, we name this smell Unfactored Hierarchy.

Potential Causes

- **Copying subtypes** – Copying code from a subtype to create a new subtype with necessary changes can introduce duplication and errors, indicating a lack of focus on design quality.
- **Improper handling of outliers** – In a Bird hierarchy, adding a new non-flying class such as Penguin can lead to a Rebellious Hierarchy smell. One solution is to remove the fly() method from the Bird class and duplicate it in all flying subclasses except Penguin, but this introduces Unnecessary Duplication smell. A better solution is to introduce intermediate classes, such as FlyingBird and NonFlyingBird, and have only the FlyingBird subclass support the fly() method.

Example


<<Abstract class>>
AbstractOwnableSynchronizer
+ getExclusiveOwnerThread()
// Other methods
<<Abstract class>>
AbstractQueuedSynchronizer
+ acquire(int): void
+ acquireInteruptibly(int): void
+ getExclusiveQueuedThread(): Collection<Thread>
+ getSharedQueuedThread(): Collection<Thread>
// Other methods
<<Abstract class>>
AbstractQueuedLongSynchronizer
+ acquire(long): void
+ acquireInteruptibly(long): void
+ getExclusiveQueuedThread(): Collection<Thread>
+ getSharedQueuedThread(): Collection<Thread>
// Other methods

The JavaDoc comment for AbstractQueuedLongSynchronizer reads:

"A version of AbstractQueuedSynchronizer in which synchronization state is maintained as a long. *This class has exactly the same structure, properties, and methods as AbstractQueuedSynchronizer with the exception that all state-related parameters and results are defined as long rather than int.* This class may be useful when creating synchronizers such as multilevel locks and barriers that require 64 bits of state."

In fact, the code comment inside AbstractQueuedLongSynchronizer reads:

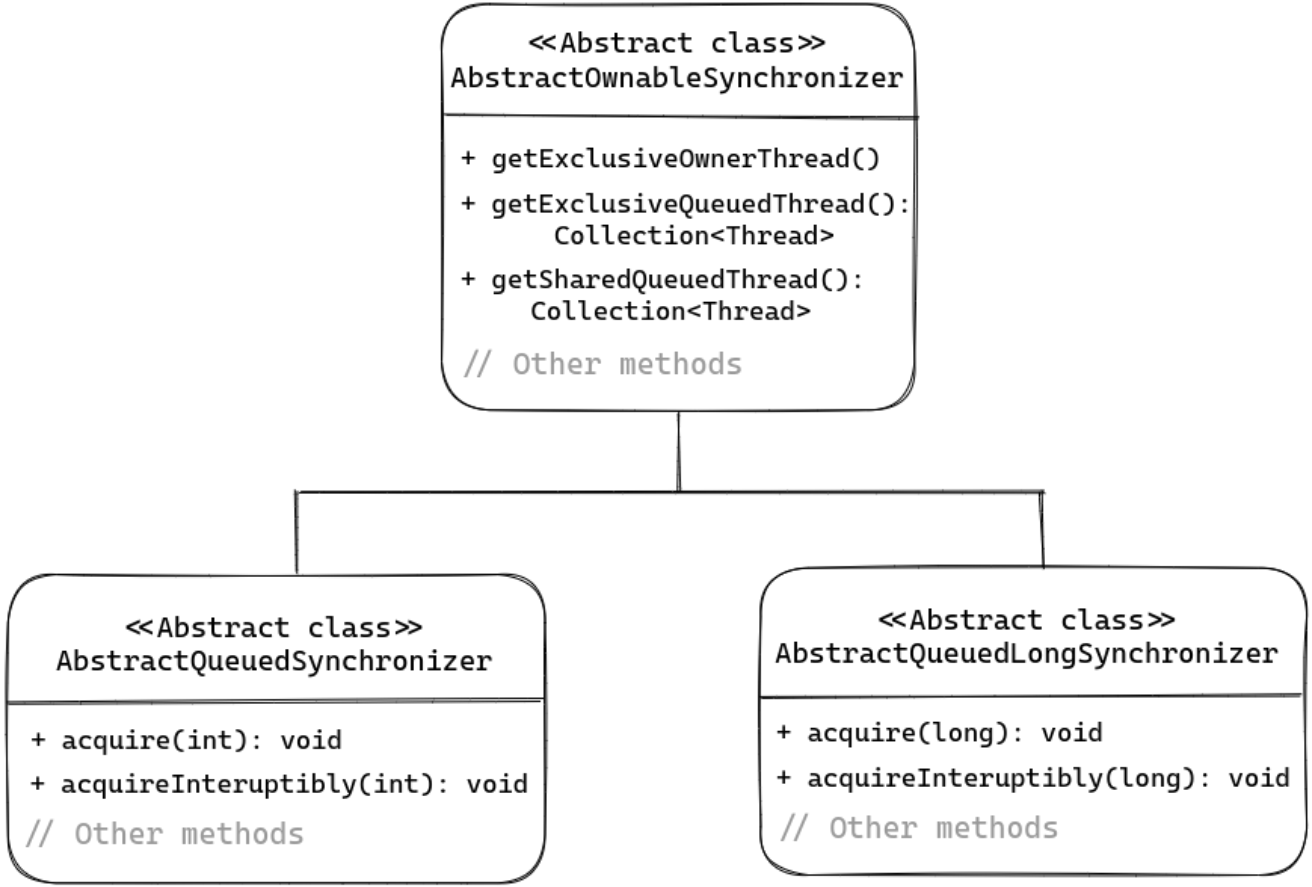
```
/*
To keep sources in sync, the remainder of this source file is exactly cloned from
AbstractQueuedSynchronizer, replacing class name and changing ints related with sync state
to longs. Please keep it that way.
*/
```

Suggested refactoring

The high-level refactoring approach to address Unfactored Hierarchy is to apply the principle of “factoring”:

- If the method signatures, method definitions, or fields are same across sibling types, then “pull-up” the relevant method interfaces, method definitions, or fields to the supertype.
- If methods defined across sibling types are similar but differ in some ways, apply relevant techniques to factor out the commonality to the supertype.

Refactored hierarchy:



Impacted Quality Attributes

- Understandability
- Changeability and Reliability
- Extensibility
- Reusability
- Testability

Aliases

- Orphan sibling method/attribute
- Incomplete inheritance
- Repeated functionality
- Redundant variable declarations
- Significant sibling duplication

Practical Considerations

- **Inadequate language support to avoid duplication**

  Unfactored Hierarchy can be addressed by factoring out duplicate implementation to the superclass, but this can introduce other smells. Factoring out implementation to a new superclass may not be feasible in languages like Java and C# due to lack of multiple class inheritance. Lack of support for generics in a language may also result in unavoidable duplication.

Wide Hierarchy

This smell arises when an inheritance hierarchy is “**too wide”** indicating that **intermediate types may be missing**.

It indicates the violation of the enabling technique “**apply meaningful generalization**”. It can result in the following problems:

• Missing intermediate types may force to directly refer to the subtypes.

• There may be unnecessary duplication within types (since commonality cannot be properly abstracted due to lack of intermediate types).

Rationale

- For effective design of inheritance hierarchies, we need to “**maintain moderation in size**”.
- If there are a large number of subtypes at the same level, it becomes harder to understand and use the hierarchy. Since the hierarchy is “too wide,” this smell is named **Wide Hierarchy**.
- We consider a hierarchy wide if any type in the hierarchy has more than 9 immediate subtypes.
- Both unfactored and wide hierarchies look similar but the difference is that Wide Hierarchy arises when there are “too” many immediate subtypes for a supertype, whereas Unfactored Hierarchy arises when there is duplication in the hierarchy. In most cases, both of them occur together.

Potential Causes

- Ignoring generalization by not checking whether relevant intermediate supertypes would be needed in the hierarchy to better accommodate those subtypes.
- Lack of periodic refactoring as the application evolves.

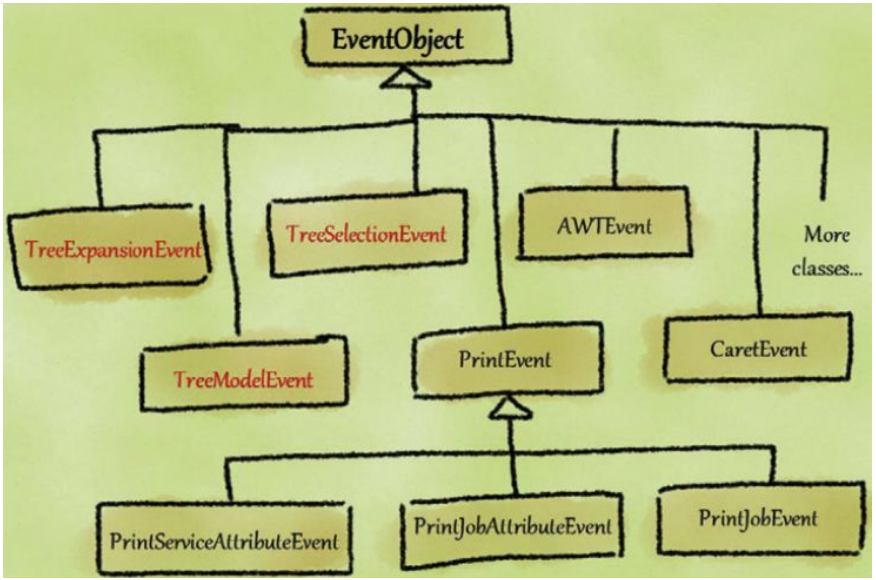
Impacted quality attributes

1. Understandability
2. Changeability
3. Extensibility
4. Reusability

Aliases

1. Wide inheritance hierarchy
2. Missing levels of abstraction
3. Coarse hierarchies
4. Getting away from abstraction

Suggested Refactoring

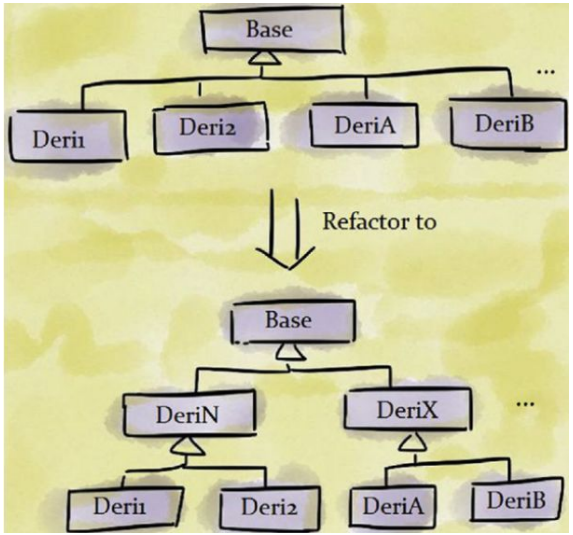


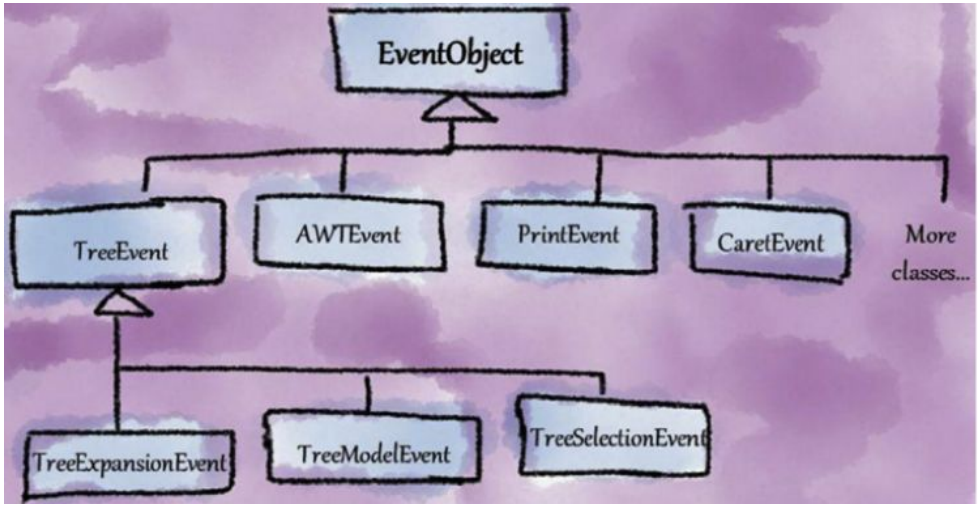
In case of Wide Hierarchy, apply "**extract superclass**" refactoring and introduce intermediate abstractions.

In this example, the classes TreeExpansionEvent, TreeModelEvent and TreeSelectionEvent could be made as subtypes of a common supertype TreeEvent (when we introduce TreeEvent class, it is at the same abstraction level as PrintEvent and AWTEvent, which indicates it is a good refactoring suggestion).

Examples

- The **java.util.EventObject** class is the superclass of **36 immediate subclasses**.
- The class **java.util.ListResourceBundle** has **443 classes** within JDK.





Speculative Hierarchy

This smell arises when one or more types in a hierarchy are provided speculatively (i.e., based on imagined needs rather than real requirements).

Rationale

- By the enabling technique “**apply meaningful generalization**”, generalization should be performed for exploiting commonalities and it should also take into account the planned future needs of the product.
- However, when this is not followed and generalization is applied purely based on **imagined requirements**, the resulting hierarchy may have supertypes that aren’t needed at all. The importance of not adding anything speculatively is also supported by the **YAGNI** (“You Ain’t Gonna Need It”) principle.

Potential Causes

Future-proofing

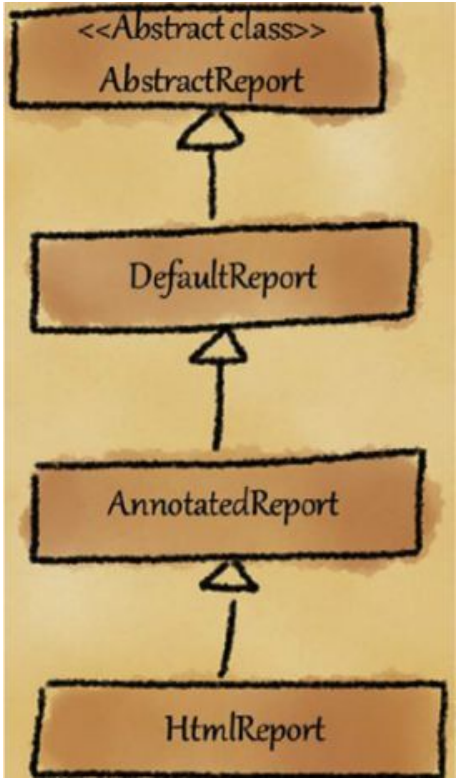
This design smell often arises due to attempts to make the design of the system "future-proof," i.e., a design that can accommodate all changes one could imagine will happen in future.

Over-engineering

When designers over-engineer a design and provide generalized types that aren't actually needed to address the real requirements, it can lead to a Speculative Hierarchy smell.

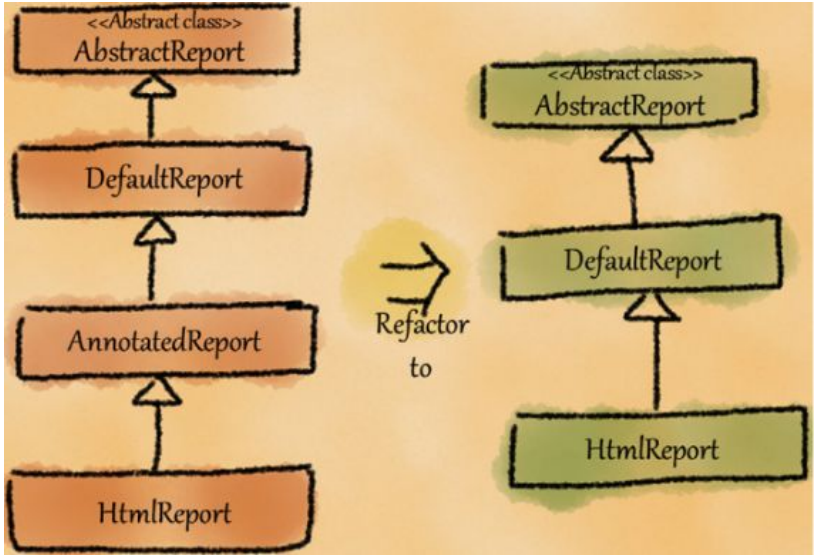
Example

- This hierarchy was part of a code analyzer tool that statically analyzed the source code and reported potential defects in the source code.
- One of the features of this tool was its support for generating HTML reports. It was also planned that this tool would support reports in other formats (such as .doc, .rtf, and .pdf) in the future.
- During the process of design, the designer speculated that the reports may need the "**annotation feature**" to be supported in the future even though it was not planned. For this reason, he introduced an intermediate type in the hierarchy as a placeholder to support annotation feature.



Suggested refactoring

- When one or more supertypes are created speculatively, apply "**collapse hierarchy**" refactoring to remove the concerned supertype(s).
- Since the current need for the tool is the ability to generate HTML reports, the AnnotatedReport type is based on a speculated need and it could be removed from the hierarchy.
- It should be pointed out that it is acceptable to have the AbstractReport and DefaultReport in the hierarchy because the support for different types of reports is part of the product roadmap.



Impacted quality attributes

1. Understandability
2. Testability

Aliases

1. Extra sub-class
2. Speculative general types
3. Speculative generality
4. List-like inheritance hierarchies

Deep Hierarchy

This smell arises when an inheritance hierarchy is “excessively” deep.

Rationale

When we apply generalization excessively, hierarchy becomes excessively deep and significantly increases the difficulty in predicting the behavior of code in a leaf type since the type inherits a relatively large number of methods from its supertypes. Following problems may arise

• A method from a supertype close to the root may be overridden many times in the hierarchy; hence, in the leaf types, the semantics of the inherited method may not be clear.

• A field defined in a supertype may be hidden in one of the methods in the subtypes (e.g., a local variable name is provided in a derived method, which happens to be same as a field name inherited from the supertype).

Rationale

- Both these cases can confuse a developer and makes the design more complex and leads to poor understandability and a greater possibility of design or implementation errors. Hence, it is important to avoid hierarchies that are excessively deep.

- As a rule-of-thumb if the inheritance hierarchy is deeper than **six levels**, it can be considered deep.

- Often, a hierarchy that is deep may have one or more intermediate supertypes that have been added because of imagined future needs. Thus, it is common to see a hierarchy that has a Deep Hierarchy also suffer from a Speculative Hierarchy smell.

Potential Causes

Excessive focus on reuse

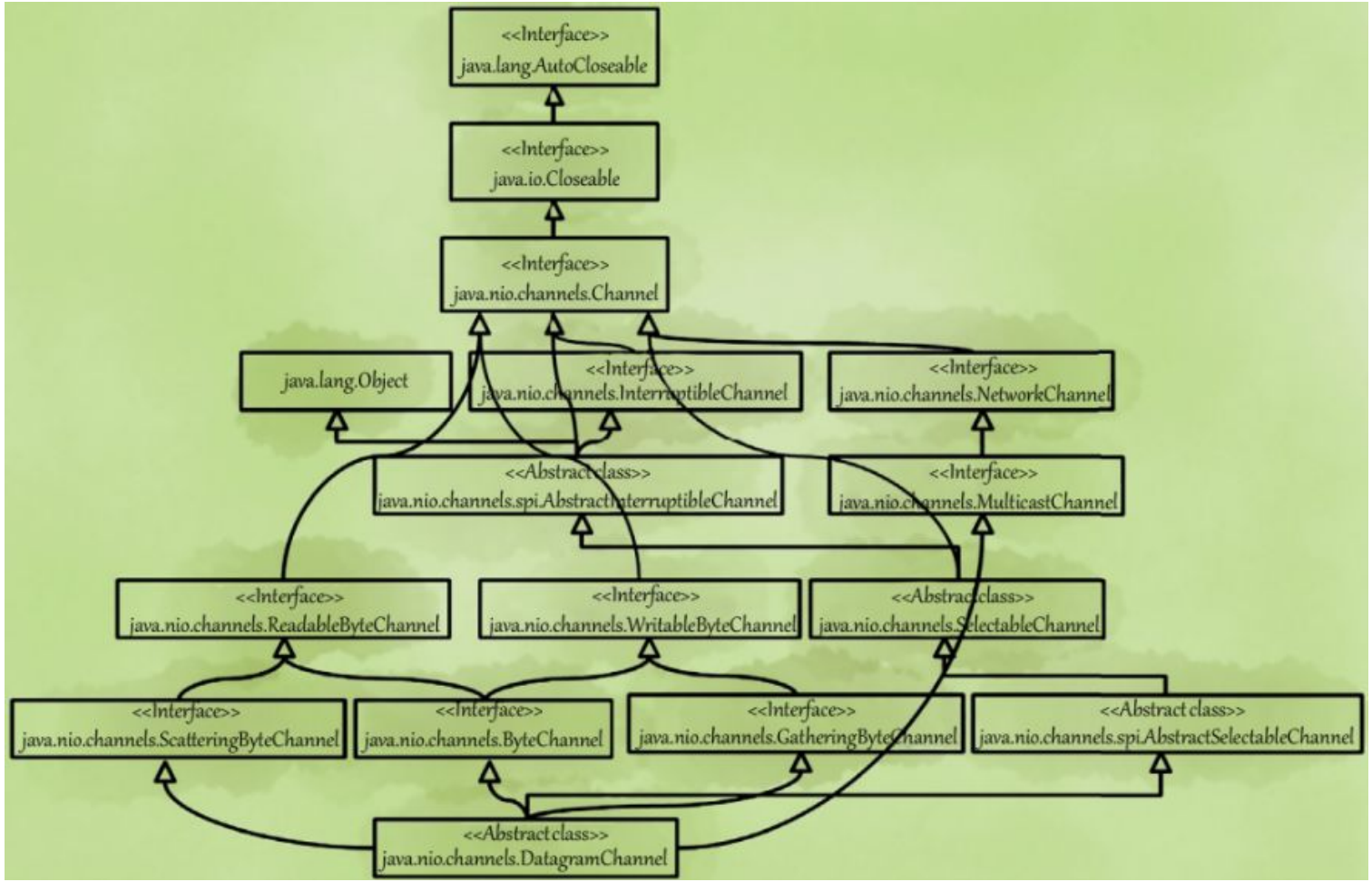
Sometimes, developers may indulge in needless generalization to create intermediate types that can be reused. Such excessive focus on reuse at the cost of understandability and usability can lead to the occurrence of a Deep Hierarchy smell.

Speculative generalization

When designers speculatively add supertypes in a hierarchy for imagined future needs, the hierarchy tends to become deep.

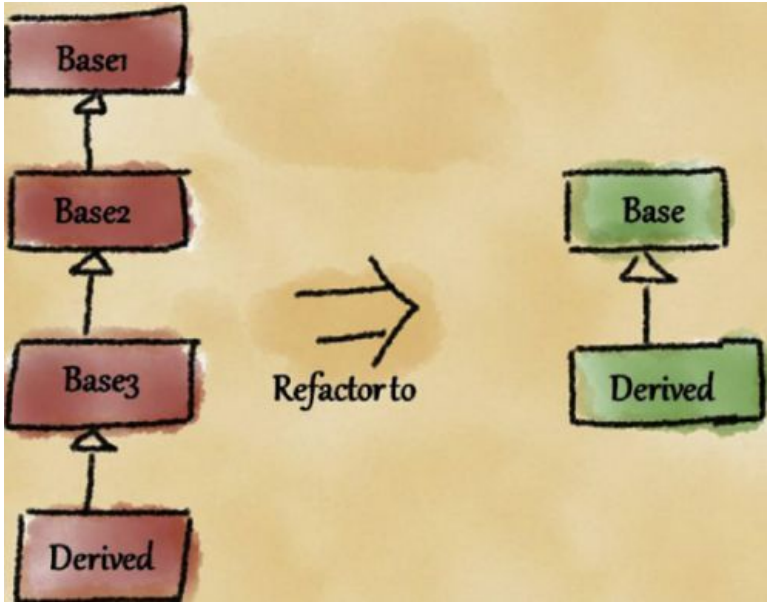
Example

Consider the inheritance hierarchy of **java.nio.channels.DatagramChannel** class which abstracts “a selectable channel for datagram-oriented sockets”. The depth of interface inheritance tree of this hierarchy is equal to **seven** and the depth of class inheritance is equal to **four**. Clearly this hierarchy is deep and difficult to understand, and suffers from Deep Hierarchy smell.


<<Interface>>
java.lang.AutoCloseable
<<Interface>>
java.io.Closeable
<<Interface>>
java.nio.channels.Channel
java.lang.Object
<<Interface>>
java.nio.channels.InterruptibleChannel
<<Interface>>
java.nio.channels.NetworkChannel
<<Abstract class>>
java.nio.channels.spi.AbstractInterruptibleChannel
<<Interface>>
java.nio.channels.MulticastChannel
<<Interface>>
java.nio.channels.ReadableByteChannel
<<Interface>>
java.nio.channels.WritableByteChannel
<<Abstract class>>
java.nio.channels.SelectableChannel
<<Interface>>
java.nio.channels.ScatteringByteChannel
<<Interface>>
java.nio.channels.ByteChannel
<<Interface>>
java.nio.channels.GatheringByteChannel
<<Abstract class>>
java.nio.channels.spi.AbstractSelectableChannel
<<Abstract class>>
java.nio.channels.DatagramChannel

Suggested Refactoring

If the hierarchy that is deep has one or more intermediate abstractions introduced unnecessarily or speculatively, apply “**Collapse Hierarchy**” refactoring.



Suggested Refactoring

- Now consider the interface java.nio.channels.InterruptibleChannel interface. It is a “**marker interface**” i.e., the interface essentially serves as a tag that can be checked at runtime (using instance of check) to test whether the implementing class supports a certain functionality or protocol.

- However, with the introduction of annotations feature in Java 1.5, there is no need for marker interfaces. Hence, ideally this interface could be safely replaced with an **annotation**. By performing such refactorings, the depth of the inheritance hierarchy given in this example can be reduced.

Impacted quality attributes

1. Understandability
2. Changeability
3. Extensibility
4. Testability
5. Reliability

Aliases

1. Distorted hierarchy

Rebellious Hierarchy

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

In this smell, a supertype and its subtypes conceptually share an IS-A relationship, but some methods defined in subtypes violate this relationship. For example, for a method defined by a supertype, its overridden method in the subtype could:

• throw an exception rejecting any calls to the method

• provide an empty (or NOP i.e., NO Operation) method

• provide a method definition that just prints “should not implement” message

• return an error value to the caller indicating that the method is unsupported

Rationale

- A subtype can mutate behavior it inherits from its supertype. The mutation is harmless when the behavior is enhanced. However, when the overriding method restricts, or cancels the behavior of the supertype method, the mutation is harmful.

- Though the supertype and subtypes may conceptually share an IS-A relationship, the methods do not. In such a case, when a supertype reference points to one of its subtype objects, it may result in defects.

- Thus, the subtype objects cannot be safely substituted in place of a supertype reference, thereby violating **Liskov's Substitution Principle** (LSP) and the enabling technique "**ensure substitutability**".

- Since the subtype "rebels" against the behavior promised by the supertype, this smell is named Rebellious Hierarchy.

Potential Causes

Creating a hierarchy rooted in an existing concrete class

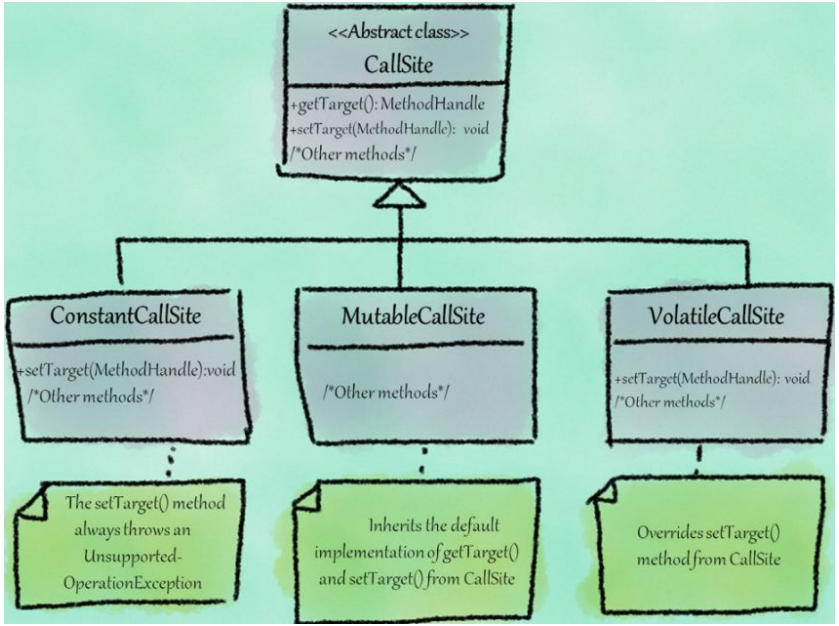
In real-world projects, as the design evolves, existing concrete classes often become the root for new hierarchies. In this context, it is important to perform periodic refactoring of the hierarchy to ensure that the types toward the root are generalized abstractions. In the absence of such refactoring, subtypes may be forced to reject irrelevant methods they inherit from their supertypes.

Creating “swiss-knife” types

When designers try to create a supertype that provides everything, specialized sub-types may be forced to reject some of the methods they inherit from the supertype.

Examples

- The abstract class CallSite is the base type for three concrete classes: Constant-CallSite, MutableCallSite and VolatileCallSite.
- CallSite class defines two methods with default implementation that are of interest to us: getTarget() and setTarget().
- A ConstantCallSite is a CallSite whose target is permanent, and can never be changed. However, its sibling classes MutableCallSite and VolatileCallSite have targets that are not permanent, i.e., their targets can be changed.
- For this reason, ConstantCallSite rejects setTarget(); any attempt to call setTarget() on a ConstantCallSite object will result in an **UnsupportedOperationException**.



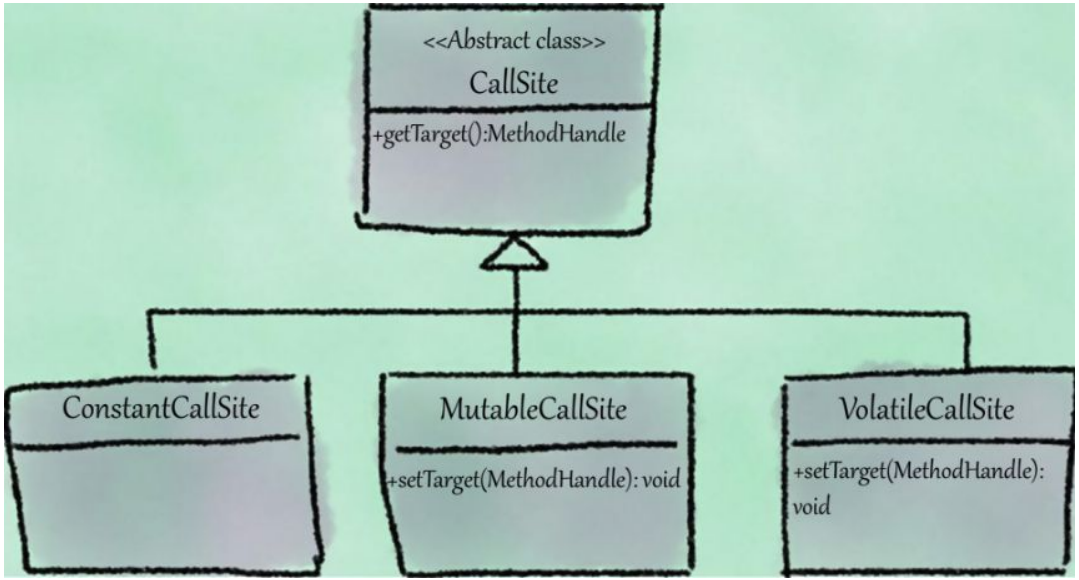
Suggested Refactoring

- If the contentious methods in the supertype that are being “refused” in some subtypes are relevant only to some subtypes, apply “**move method**” refactoring and lower those methods from the supertype to the relevant subtypes.

- If the contentious methods in the supertype are being “refused” in all of its subtypes, then apply “**remove method**” refactoring and remove the contentious methods from the supertype.

- In some cases, introducing a new intermediate type to capture the needs of a subset of subtypes may help address this smell.

By moving method to lower hierarchy, rebellious smell is avoided.



Impacted quality attributes

1. Understandability
2. Changeability
3. Extensibility
4. Reusability
5. Reliability

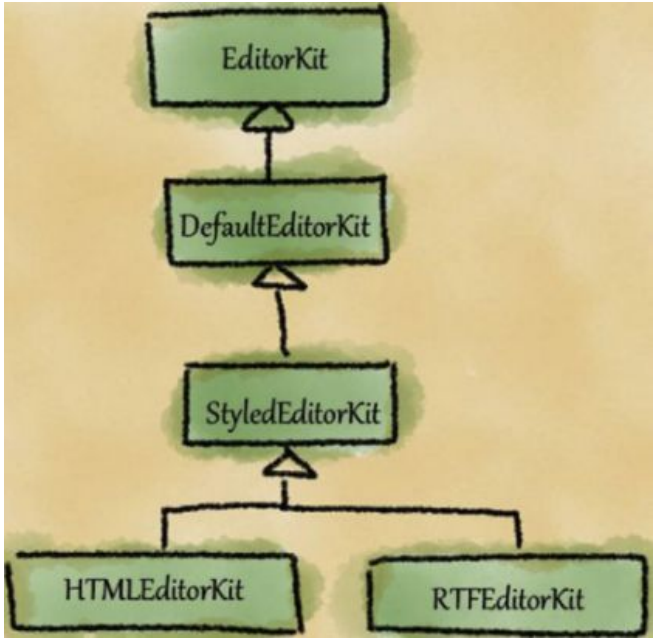
Aliases

1. Refused bequest
2. Refused parent bequest
3. Naughty children
4. Premature interface abstraction

Practical Considerations

Yet-to-be implemented functionality

Consider the class javax.swing.text.rtf.RTFEditorKit that provides support for RTF (Rich Text Format). This class and the HTMLEditorKit class both extend the StyledEditorKit. In this hierarchy, the write() method in RTFEditorKit rejects the inherited write method by throwing an IOException. It is clear that the designers have chosen to reject the write() method because it is a work-in-progress or yet-to-be-implemented functionality.



Broken Hierarchy

This smell arises when a supertype and its subtype conceptually do not share an “IS-A” relationship resulting in broken substitutability. There are three forms of this smell:

• Form 1—The methods in the supertype are still applicable or relevant in the subtype (though the supertype and subtype do not share an IS-A relationship).

• Form 2—The interface (i.e., public methods) of the subtypes includes (by inheritance) the supertype methods that are not relevant or acceptable for the subtypes (note that there is no IS-A relationship); however, the subtypes do not reject those irrelevant or unacceptable methods that are inherited from the supertype.

• Form 3—The subtype implementation explicitly rejects irrelevant or unacceptable methods that are inherited from the supertype.

Rationale

This smell violates “principle of substitutability” and the enabling technique “ensure substitutability.”

Potential Causes

Inheritance for implementation reuse

The subtype inherits from the supertype for reusing the supertype’s functionality without sharing a true IS-A relationship with the supertype.

Incorrect application of inheritance

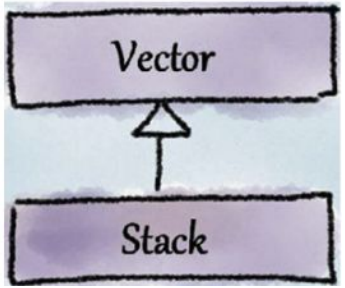
Sometimes, an inexperienced designer may mistakenly invert the inheritance relationship between types.

Examples and Suggested refactoring

Consider the relationship between java.util.Stack and java.util.Vector in JDK7

Stack - LIFO and Vector - class implements a growable array of objects

A Stack is definitely not a Vector, and hence this inheritance relationship is an example of Broken Hierarchy. Because of this design mistake, it is possible to insert or remove elements from the middle of a Stack (since Vector allows such methods and Stack inherits them), which is obviously not desirable!



Impacted quality attributes

1. Understandability
2. Changeability
3. Extensibility
4. Reusability
5. Reliability

Aliases

1. Inappropriate use of inheritance
2. Containment by inheritance
3. Mistaken aggregates
4. Misapplying IS A
5. Improper inheritance
6. Inverse abstraction hierarchies
7. Subclasses do not redefine methods
8. Subclass inheriting inappropriate operations from superclass

Multipath Hierarchy

This smell arises when a subtype inherits both directly as well as indirectly from a supertype leading to unnecessary inheritance paths in the hierarchy.

Rationale

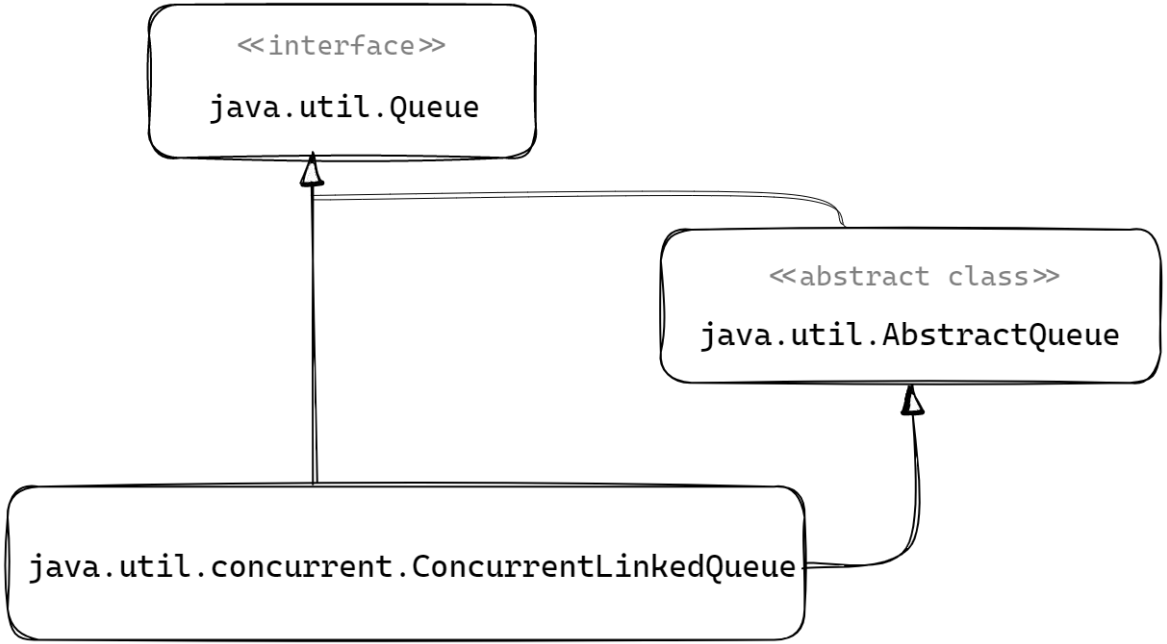
Multipath Hierarchy is a design issue in which an inheritance hierarchy includes unnecessary paths, violating the *"avoid redundant paths"* principle.

This can make it harder for designers to comprehend and understand the relationship between types, and may even lead to runtime problems. In this smell, a subtype inherits a supertype from **one or more redundant paths**, cluttering the hierarchy and making it less clear.

Potential Causes

- **Overlooking existing inheritance paths** – The primary cause of the Multipath Hierarchy smell is overlooking existing inheritance paths and needlessly inheriting a supertype again. This mistake is more likely in deep inheritance hierarchies, where the number of inheritance paths increases and designers can miss redundant paths, inadvertently creating unnecessary ones.

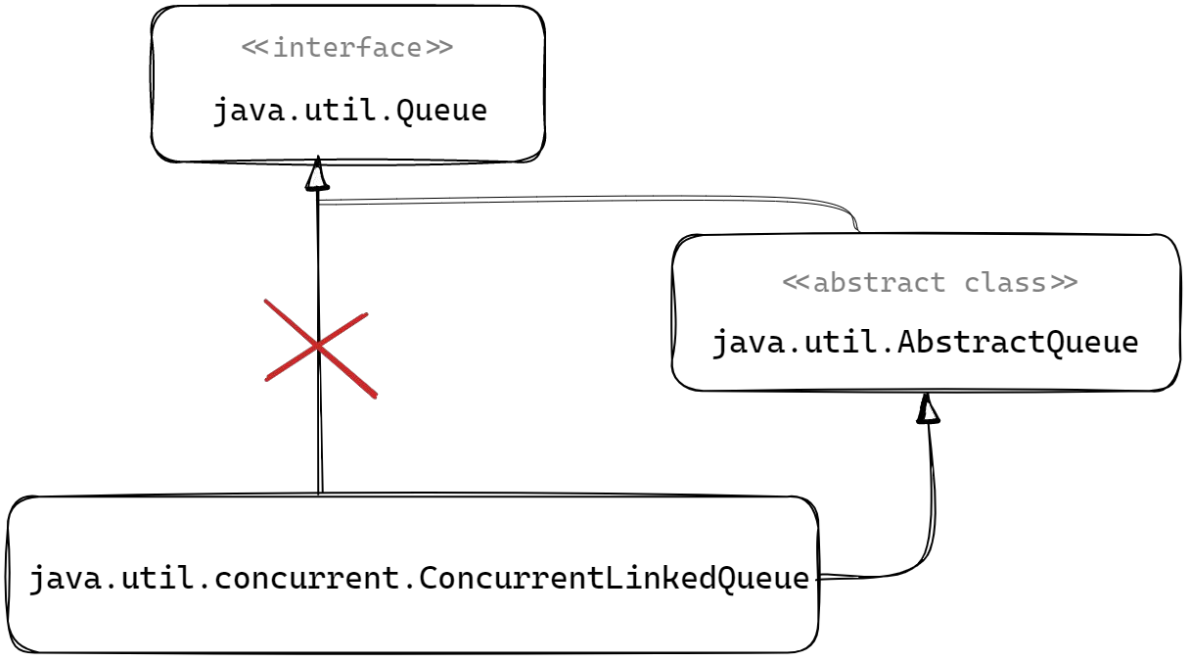
Example


<<interface>>
java.util.Queue
<<abstract class>>
java.util.AbstractQueue
java.util.concurrent.ConcurrentLinkedQueue

Suggested refactoring

- To address the Multipath Hierarchy smell, the refactoring involves removing unnecessary paths in the hierarchy.
- For example, in the ConcurrentLinkedQueue hierarchy, extending AbstractQueue already implements Queue, making it redundant for ConcurrentLinkedQueue to explicitly implement Queue.
- Therefore, the inheritance path for java.util.Queue can be safely removed.

Refactored hierarchy:



Impacted Quality Attributes

- Understandability
- Reliability
- Testability

Aliases

- Degenerate inheritance
- Repeated inheritance

Practical Considerations

- Introducing a redundant inheritance path for convenience

Cyclic Hierarchy

This smell arises when a supertype in a hierarchy depends on any of its subtypes.This dependency could be in the following forms:

- a supertype contains an object of one of its subtypes
- a supertype refers to the type name of one of its subtypes
- a supertype accesses data members, or calls methods defined in one of its subtypes.

Rationale

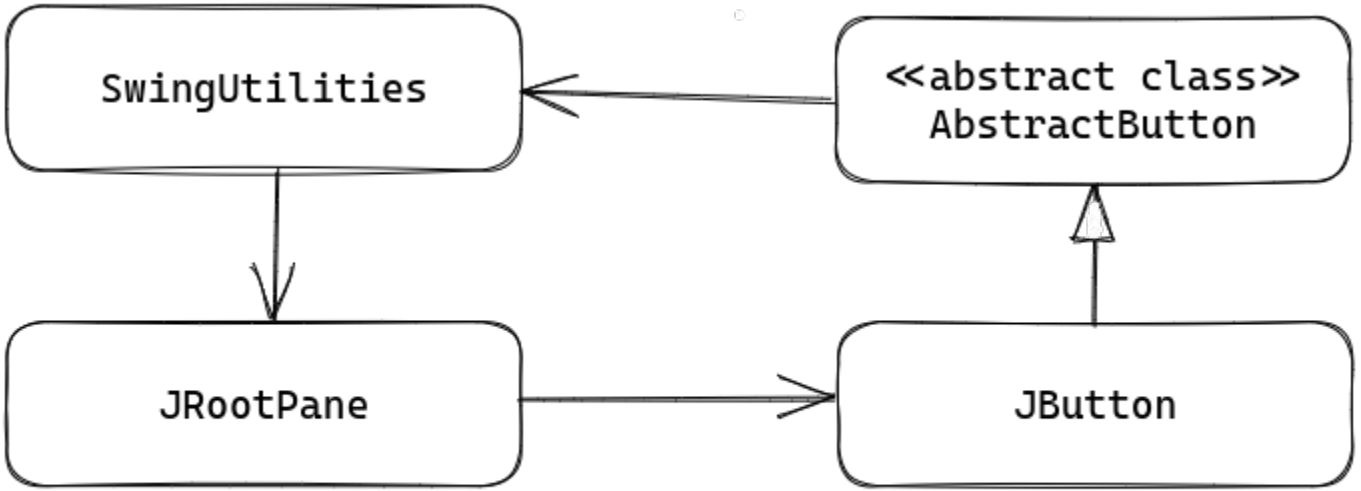
A hierarchical organization simplifies complex relationships between types, but it's crucial to express them logically and consistently to reap its benefits. A supertype should depend on its subtypes, not the other way around, violating the Acyclic Dependencies Principle and the *"ensure proper ordering"* technique.

A **reference from a supertype to any of its subtypes creates a "cycle" in the hierarchy**, leading to the Cyclic Hierarchy smell. When a supertype's constructor uses one of its subtypes, it can access uninitialized parts of the subtype, causing subtle bugs.

Potential Causes

- **Improper assignment of responsibilities** – The most common cause of Cyclic Hierarchy is improper assignment of responsibilities across types. For instance, if a responsibility that ideally should have belonged to a subtype is assigned to the supertype, the supertype will have a dependency on the subtype leading to a Cyclic Hierarchy.
- **Hard to visualize indirect dependencies** – In complex software systems, designers usually find it difficult to mentally visualize dependency relationships between types. As a result, designers may inadvertently end up creating a long chain of indirect dependencies from supertypes to subtypes.

Example

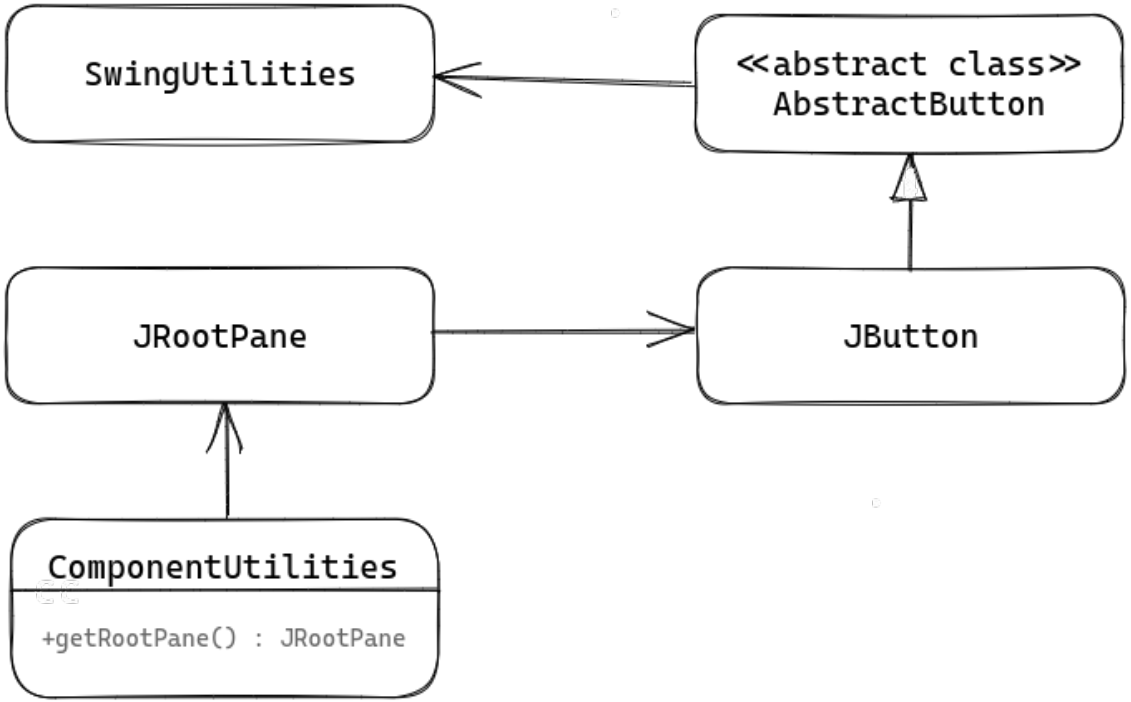

SwingUtilities
<<abstract class>>
AbstractButton
JRootPane
JButton

Suggested refactoring

Consider the following approaches for refactoring Cyclic Hierarchy:

- If the reference from the supertype to its subtype(s) is unnecessary, remove that reference.
- If there is an extensive coupling between a supertype and its immediate subtype, consider merging them together.
- Depending on the context, consider applying refactoring such as “move method” or “extract class” to help break the cyclic dependency from the supertype to its subtype(s).
- Consider applying State or Strategy patterns if the supertype requires the services of one of its subtypes.

Refactored hierarchy:



Impacted Quality Attributes

- Understandability
- Changeability, Extensibility, and Reliability
- Reusability
- Testability

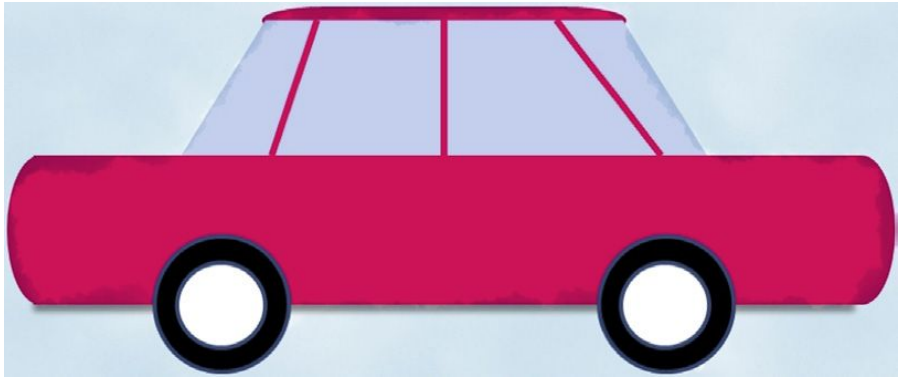
Aliases

- Knows of derived
- Curious superclasses
- Inheritance/reference cycles
- Descendant reference
- Superclass uses subclass during initialization
- Inheritance loops

Practical Considerations

- **Stable subtypes** – The main problem with the supertype having knowledge of one of its subtypes is that changes to subtypes can potentially affect the supertype. Hence, this design smell might be acceptable if it is known that the subtypes are not going to change in the future.

ENCAPSULATION SMELLS



- Hide implementation details.
- Hide variations.

What happens when we violate Encapsulation?

1. Deficient Encapsulation
2. Leaky Encapsulation
3. Missing Encapsulation
4. Unexploited Encapsulation

DEFICIENT ENCAPSULATION

This smell occurs when the declared accessibility of one or more members of an abstraction is more permissive than actually required. Deficient encapsulation occurs when an object's internal state can be accessed and modified directly by other parts of the program, violating the principle of encapsulation in object-oriented programming.

POTENTIAL CAUSES

- ***Easier testability***

  Often developers make private methods of an abstraction public to make them testable.

- ***Procedural thinking in object oriented context***

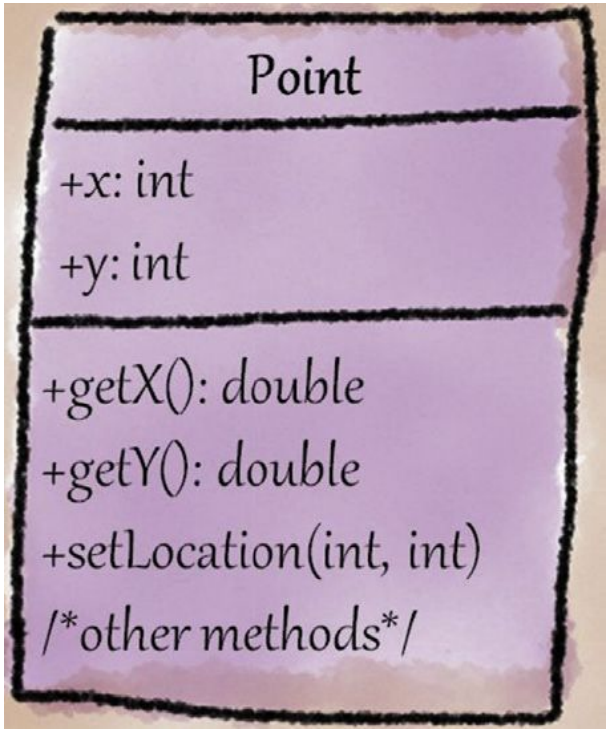
  Expose data as global variables when the data needs to be used by multiple abstractions.

- ***Quick-fix solutions***

  Delivery pressures often force developers to take shortcuts

Example

java.awt.Point



SUGGESTED REFACTORING

- Make the data members in classes Point private.
- Provide suitable accessor and mutator methods for the fields.

ALIASES

- Hideable public attributes/methods
- Unencapsulated class
- Class with unparameterized methods

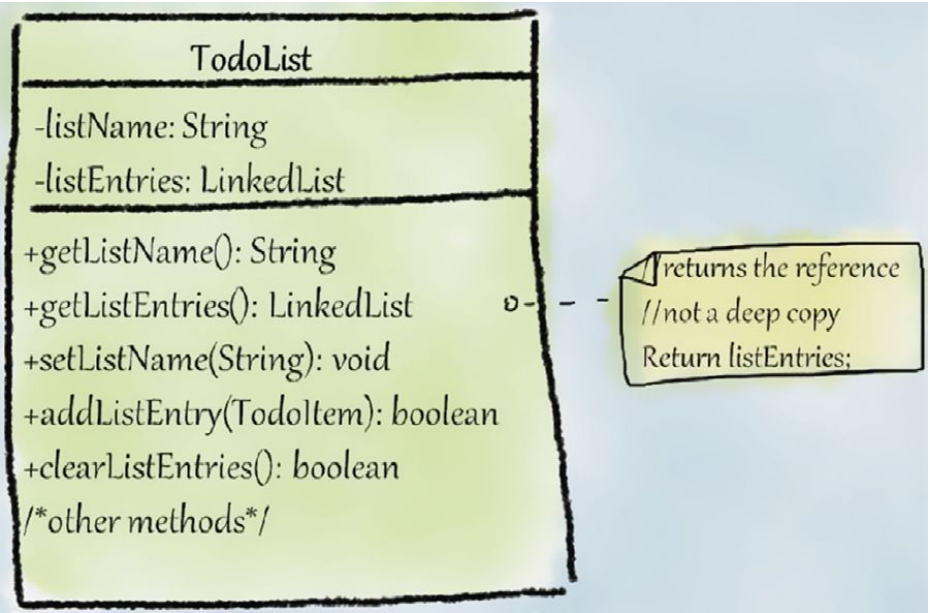
LEAKY ENCAPSULATION

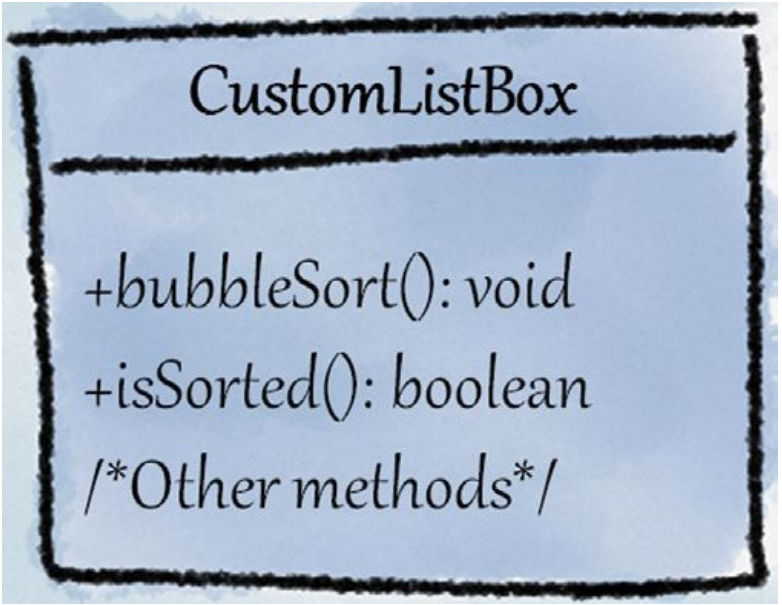
Leaky encapsulation occurs when an object's internal state can be accessed or modified indirectly, such as through public methods or properties that expose too much information about the object's implementation. This violates the principle of encapsulation in object-oriented programming and can lead to code that is difficult to maintain and modify.

POTENTIAL CAUSES

- *Lack of awareness of what should be “hidden”*
- *Viscosity*
- *Use of fine-grained interface*

Example


TodoList
-listName: String
-listEntries: LinkedList
+getListName(): String
+getListEntries(): LinkedList
+setListName(String): void
+addListEntry(TodoItem): boolean
+clearListEntries(): boolean
/*other methods*/
//returns the reference
//not a deep copy
Return listEntries;


CustomListBox
+bubbleSort(): void
+isSorted(): boolean
/*Other methods*/

SUGGESTED REFACTORING

- Change the return type of the *getListEntries()* in such a way that it does not reveal the underlying implementation.

- Since List interface is the base type of classes such as ArrayList and LinkedList, one refactoring would be to use List as the return type of *getListEntries()* method.

- The suggested refactoring for the **CustomListBox** example is to replace the method name *bubbleSort()* with *sort().*

MISSING ENCAPSULATION

- This smell occurs when implementation variations are not encapsulated within an abstraction or hierarchy.
- A client is tightly coupled to the different variations of a service it needs. Thus, the client is impacted when a new variation must be supported or an existing variation must be changed.
- Whenever there is an attempt to support a new variation in a hierarchy, there is an unnecessary “explosion of classes,” which increases the complexity of the design.

Open Closed Principle!

POTENTIAL CAUSES

- Lack of awareness of changing concerns

  Inexperienced designers are often unaware of principles such as OCP and can end up creating designs that are not flexible enough to adapt to changing requirements.

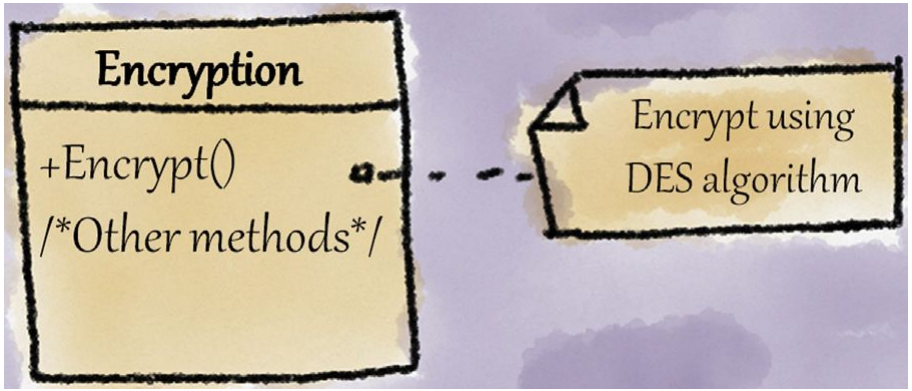
- Lack of refactoring

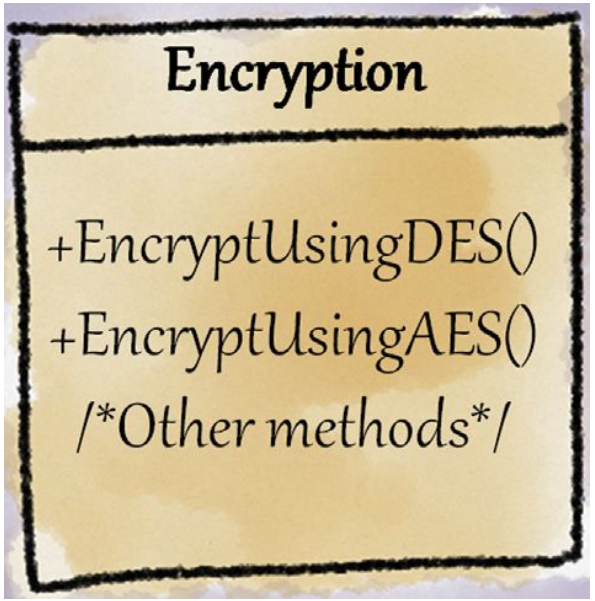
  As existing requirements change or new requirements emerge, the design needs to evolve holistically to accommodate the change.

- Naive design decisions

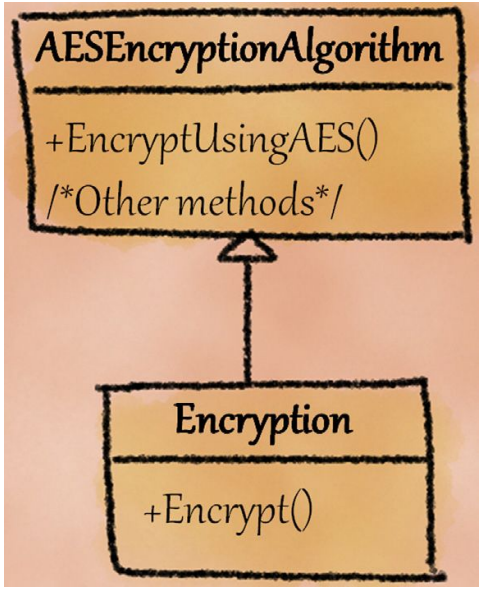
  When designers naively use simplistic approaches such as creating a class for every combination of variations, it can result in unnecessarily complex designs.

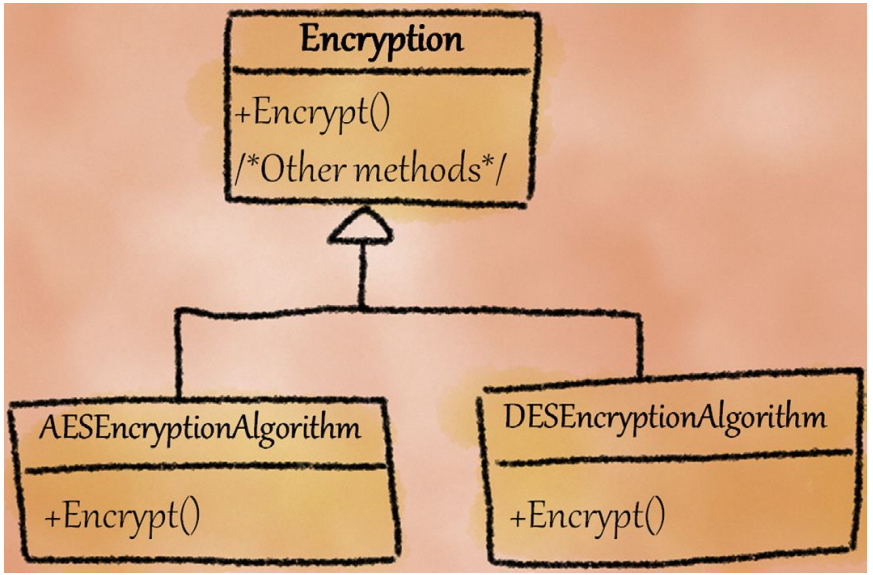
EXAMPLE


Encryption
+Encrypt()
/*Other methods*/
Encrypt using
DES algorithm


Encryption
+EncryptUsingDES()
+EncryptUsingAES()
/*Other methods*/

SUGGESTED REFACTORING





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.

POTENTIAL CAUSES

- Procedural thinking in object-oriented language
- Lack of application of object-oriented principles

EXAMPLE

```
Animal animal = // create an animal
if(animal instanceof Dog){
   // make a dog sound
}
else if(animal instanceof Cat){
   // make a cat sound
}
```

SUGGESTED REFACTORING

```
Animal animal = // create an animal

String sound = animal.makeSound();
```

Use polymorphism!

MODULARIZATION SMELLS

PRINCIPLE

The principle of modularization advocates the creation of cohesive and loosely coupled abstractions through techniques such as localization and decomposition.

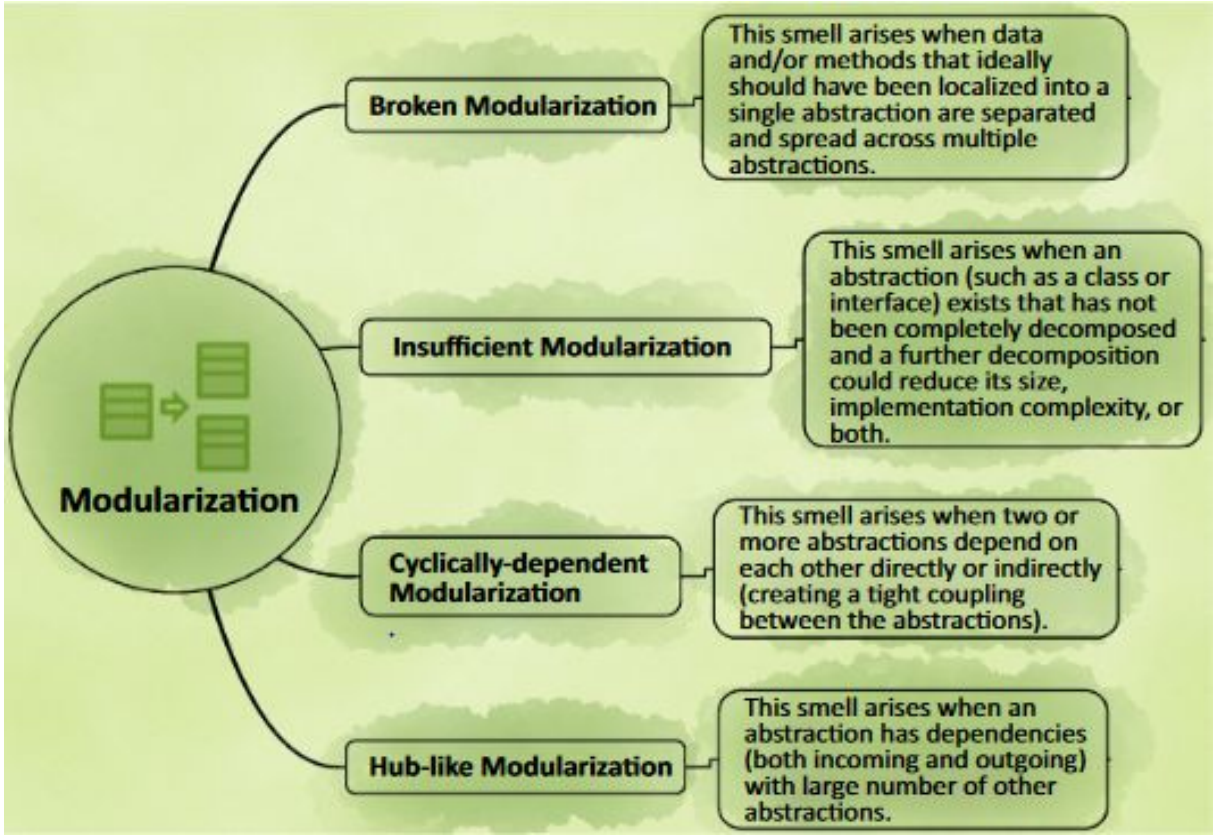
TECHNIQUES

- **Localize related data and methods -** Each abstraction should be cohesive in nature
- **Decompose abstractions to manageable size -** Break large abstractions into smaller ones that are moderate in size
- **Create acyclic dependencies -** Abstractions should not be cyclically-dependent on each other.
- **Limit dependencies -** Create abstractions with low fan-in and fan-out

Fan-in and Fan-out

- Fan-in refers to the number of abstractions that are dependent on a given abstraction.
- Thus, a change in an abstraction with high fan-in may result in changes to a large number of its clients.
- Fan-out refers to the number of abstractions on which a given abstraction depends.
- A large fan-out indicates that any change in any of these abstractions may impact the given abstraction.
- Thus, to prevent a ripple effect due to potential changes, it is important to reduce the number of dependencies between abstractions in the design.

Smells that violate the principle of modularization



Design Smells and Corresponding Violated Enabling Technique

Design Smells	Violated Enabling Technique
Broken modularization (5.1)	*Localize related data and methods*
Insufficient modularization (5.2)	*Decompose abstractions to manageable size*
Cyclically-dependent modularization (5.3)	*Create acyclic dependencies*
Hub-like modularization (5.4)	*Limit dependencies*

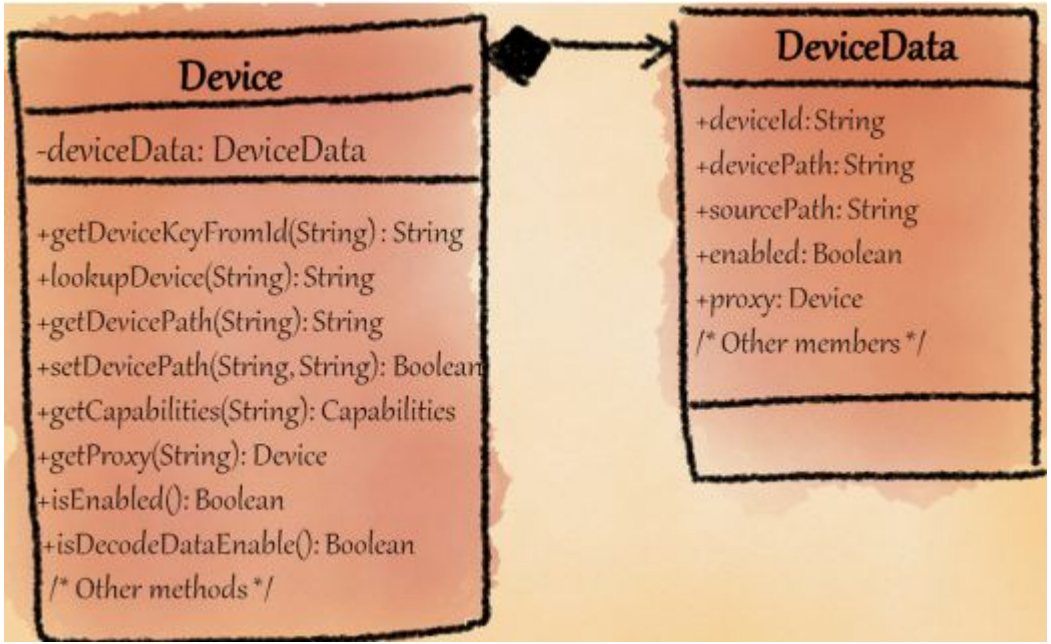
BROKEN MODULARIZATION

This smell arises when data and/or methods that ideally should have been localized into a single abstraction are separated and spread across multiple abstractions.

This smell commonly manifests as:

- Classes that are used as a holder for data but have no methods operating on the data within that class.
- Methods in a class that are more interested in members of other classes.

EXAMPLE


Device
-deviceData: DeviceData
+getDeviceKeyFromId(String) : String
+lookupDevice(String): String
+getDevicePath(String): String
+setDevicePath(String, String): Boolean
+getCapabilities(String): Capabilities
+getProxy(String): Device
+isEnabled(): Boolean
+isDecodeDataEnable(): Boolean
/* Other methods */
DeviceData
+deviceId: String
+devicePath: String
+sourcePath: String
+enabled: Boolean
+proxy: Device
/* Other members */

Rationale:

- An abstraction suffering from this smell violates this enabling technique when the abstraction has only data members and the methods operating on the data are provided in some other abstraction(s).
- When members that should ideally belong in a single abstraction are spread across multiple abstractions, the result is a tight coupling among the abstractions.
- Hence, this smell violates the principle of modularization.
- Since related members are “broken” and are made part of different abstractions, this smell is named Broken Modularization.

POTENTIAL CAUSES

- **Procedural thinking in object-oriented languages:**

Procedural languages provide language features such as “structs” (in C) and “record” (in Pascal) that hold data members together. Functions process the common data stored in structs/records. When developers from procedural language backgrounds such as C or Pascal move to an object-oriented language, they tend to separate data from functions operating on it, thereby resulting in this smell.

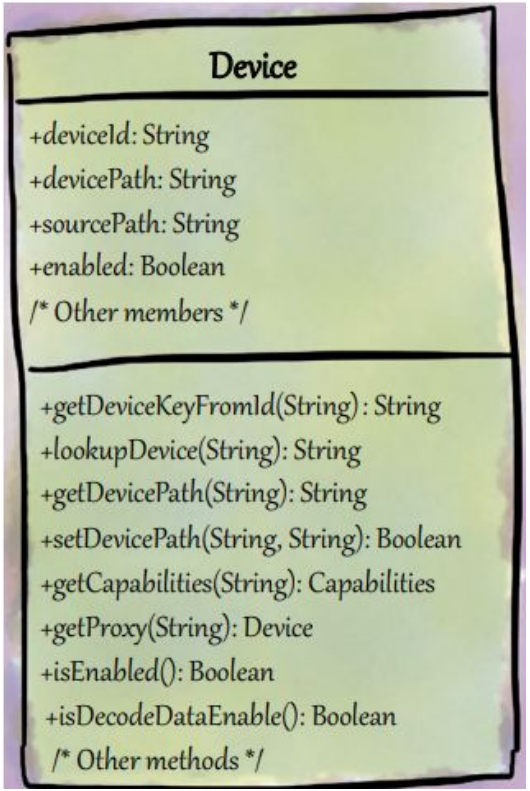
- # Lack of knowledge of existing design

Large real-world projects have a complex design and have a codebase that is spread over numerous packages. In such projects, developers usually work on a small part of the system and may not be aware of the other parts of the design. Due to this, while implementing a new feature, developers may be unaware of the most appropriate classes where data/methods should be placed; this may lead to members being misplaced in the wrong classes

Refactoring

- If a method is used more by another class (Target class) than by the class in which it is defined (Source class), apply “move method” refactoring and move that method from Source class to the Target class.
- If a field is used more by another class (Target class) than the class in which it is defined (Source class), apply “move field” refactoring and move that field from Source class to the Target class.

Solution for the example



*Note
Grouping all seemingly related methods into a single abstraction to address Broken Modularization can lead to a Multifaceted Abstraction or Insufficient Modularization smell. Hence, due caution needs to be exercised while refactoring this smell.

IMPACTED QUALITY ATTRIBUTES

//Refer pg:100&101

- Understandability
- Changeability and Extensibility
- Reusability and Testability
- Reliability

ALIASES

- Class passively stores data
- Data class
- Data records
- Record (class)
- Data container
- Misplaced operations
- Feature envy
- Misplaced control

PRACTICAL CONSIDERATIONS

Auto-generated code - The code generated from auto-code generators (from higher level models) often consists of a number of data classes. In the context of auto-generated code, it is not a recommended practice to directly change the generated code, since the models will be out-of-sync with the code. Hence, it may be acceptable to live with such data classes in generated code.

INSUFFICIENT MODULARIZATION

This smell arises when an abstraction exists that has not been completely decomposed, and a further decomposition could reduce its size, implementation complexity, or both.

There are two forms of this smell:

- **Bloated interface**: An abstraction has a large number of members in its public interface. Bloated interfaces can lead to code that is harder to maintain, harder to test, and more error-prone.
- **Bloated implementation**: An abstraction has a large number of methods in its implementation or has one or more methods with excessive implementation complexity.

The smell could also appear as a combination of these two forms, i.e., “bloated interface and implementation.”

Rationale

- Modularization concerns the logical partitioning of a software design so that the design becomes easy to understand and maintain.
- One of the key enabling techniques for effective modularization is to “decompose abstractions to manageable size.”
- When an abstraction has complex methods or a large number of methods, it violates this enabling technique and the principle of modularization.
- Since the abstraction is inadequately decomposed, we name this smell Insufficient Modularization.

Potential Causes

- **Providing centralized control**

A typical cause of bloated implementation is centralizing control and assigning a large amount of work to a single abstraction or a method in an abstraction.

- **Creating large classes for use by multiple clients**

Often, software developers create a large abstraction so that it can be used by multiple clients. However, creating a single large abstraction that fits the needs of multiple clients leads to many problems, such as reduced changeability and extensibility of the design

- **Grouping all related functionality together**

Often, inexperienced developers tend to group together and provide all related functionality in a single class or interface without understanding how the Single Responsibility Principle (SRP) should be properly applied. This results in bloated interfaces or classes.

Example

An example of Insufficient Modularization with bloated implementation is java. net.SocketPermission in JDK 1.7. The problem with the SocketPermission class is that the sum of Cyclomatic complexities of its methods is 193! In particular, one of its private methods getMask() has a Cyclomatic complexity of 81 due to its complex conditional checks with nested loops, conditionals, etc.

Suggested Refactoring

- **Refactoring for bloated interface**
  - If a subset of the public interface consists of closely related (cohesive) members, then extract that subset to a separate abstraction.
  - If a class consists of two or more subsets of members and each set of members is cohesive, split these subsets into separate classes.
- **Refactoring for bloated implementation**
  - If the method logic is complex, introduce private helper methods that help simplify the code in that method.
  - In case an abstraction has both Multifaceted Abstraction smell as well as Insufficient Modularization smell, encapsulate each responsibility within separate (new or existing) abstractions.

Solution for the example

Create helper methods for the complex methods such as getMask() in java.net. SocketPermission. For example, the private getMask() method has code for skipping whitespace in the passed string argument, which could be extracted to a new helper method. Interestingly, getMask() method also has duplicate blocks of code for checking characters in four known strings: CONNECT, RESOLVE, LISTEN, and ACCEPT. One possible refactoring is to create a helper method named matchForKnownStrings() that checks for the characters in these four strings. Now, the code within the getMask() method can be simplified by making four calls to matchForKnownStrings() instead of duplicating code four times for making these checks.

IMPACTED QUALITY ATTRIBUTES

//Refer Pg 106&107

- Understandability
- Changeability and Extensibility
- Testability
- Reliability

ALIASES

- God class
- Fat interface
- Blob class
- Classes with complex control flow
- Too much responsibility
- Local breakable (class)

PRACTICAL CONSIDERATIONS

- **Key classes**

Key classes abstract most important concepts in a system and tend to be large, complex, and coupled with many other classes in the system. While it is difficult to avoid such classes in real-world systems, it is still important to consider how they could be decomposed so that they become easier to maintain.

- **Auto-generated code**

Often, classes that are created as part of automatically generated code have complex methods. For example, tools for generating parsers usually create complex code with extensive conditional logic. In practice, it is difficult to change the generator tool or manually refactor such complex code.

CYCLICALLY-DEPENDENT MODULARIZATION

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

Rationale

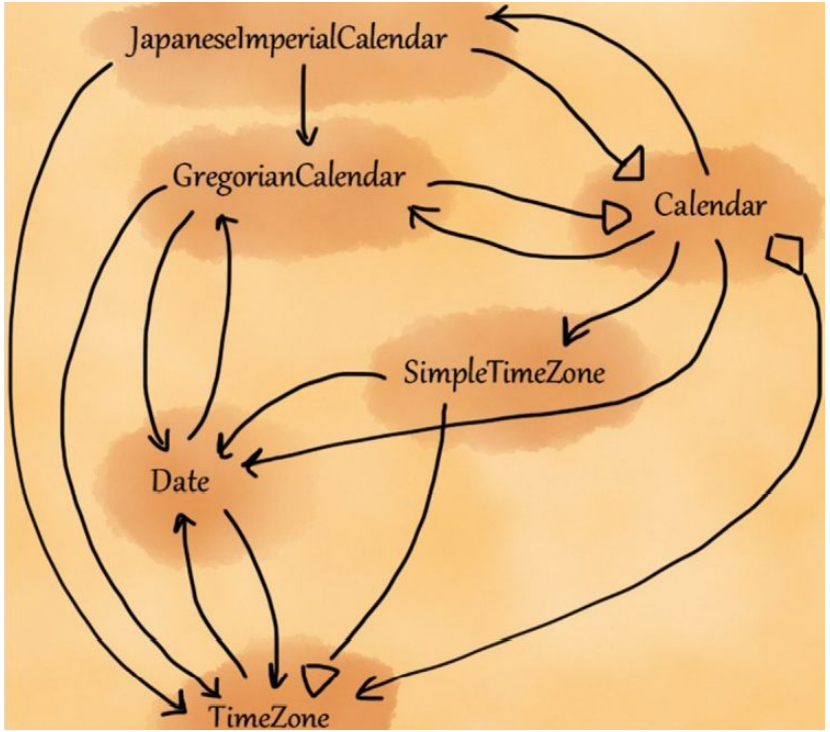
- Cyclic dependencies between abstractions violate the Acyclic Dependencies Principle (ADP) and Ordering Principle(Open-closed principle).
- In case of cyclic dependencies, changes in one class (say A) may lead to changes in other classes in the cycle (say B). However, because of the cyclic nature, changes in B can have ripple effects on the class where the change originated (i.e., A).
- Large and indirect cyclic dependencies are usually difficult to detect in complex software systems and are a common source of subtle bugs.

Special form of cyclic dependency

- Inheritance hierarchy.
- A subtype has a dependency on its supertype because of the inheritance relationship.
- However, when the supertype also depends on the subtype (for instance, by having an explicit reference to the subtype), it results in a cyclic dependency.

//Cyclic Hierarchy smell

The Tangled



//Cycles between six abstractions in java.util package.

POTENTIAL CAUSES

- **Improper responsibility realization**

Often, when some of the members of an abstraction are wrongly misplaced in another abstraction, the members may refer to each other, resulting in a cyclic dependency between the abstractions.

- **Passing a self reference**

A method invocation from one abstraction to another often involves data transfer. If instead of explicitly passing only the required data, the abstraction passes its own reference (for instance, via “this”) to the method of another abstraction, a cyclic dependency is created.

Potential Causes

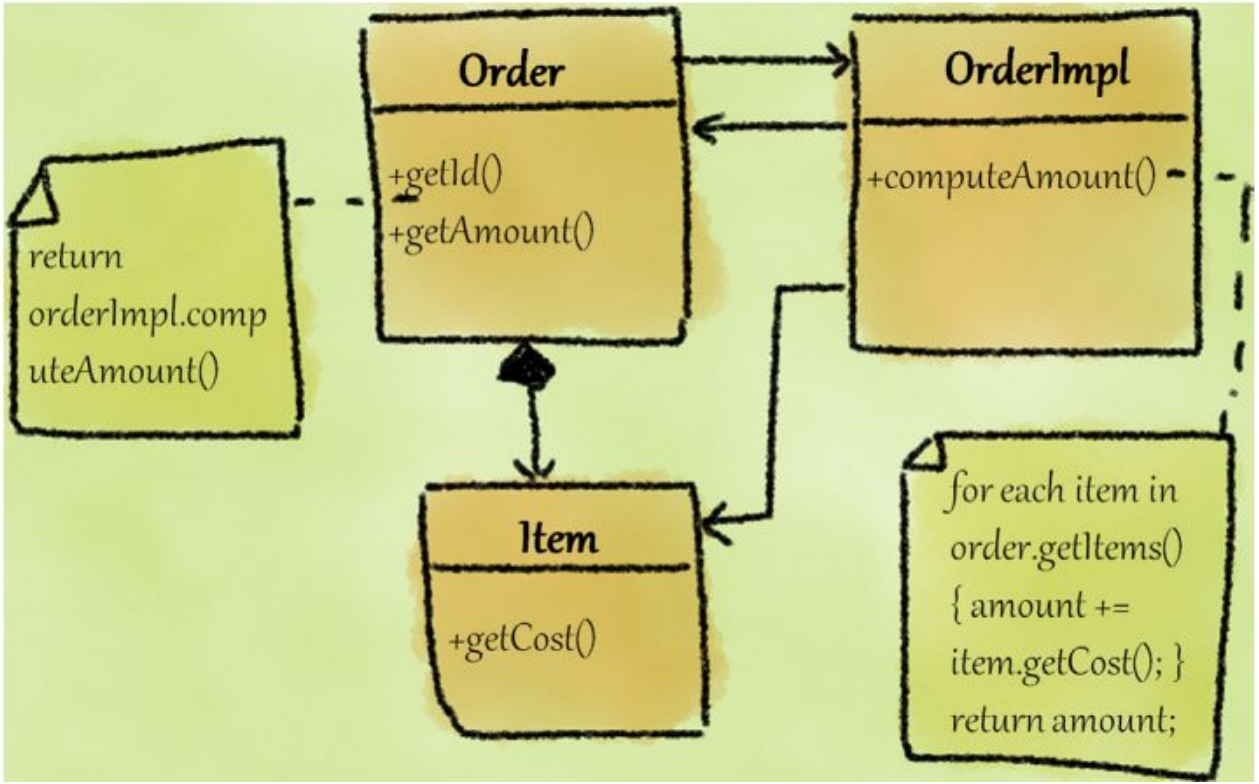
- **Implementing call-back functionality**

Circular dependencies are often unnecessarily introduced between two classes while implementing call-back functionality. This is because inexperienced developers may not be familiar with good solutions such as design patterns which can help break the dependency cycle between the concerned classes.

- **Hard-to-visualize indirect dependencies**

In complex software systems, designers usually find it difficult to mentally visualize dependency relationships between abstractions. As a result, designers may inadvertently end up creating cyclic dependencies between abstractions

Example


Order
+getId()
+getAmount()
OrderImpl
+computeAmount()
return orderImpl.computeAmount()
Item
+getCost()
for each item in order.getItems() { amount += item.getCost(); }
return amount;

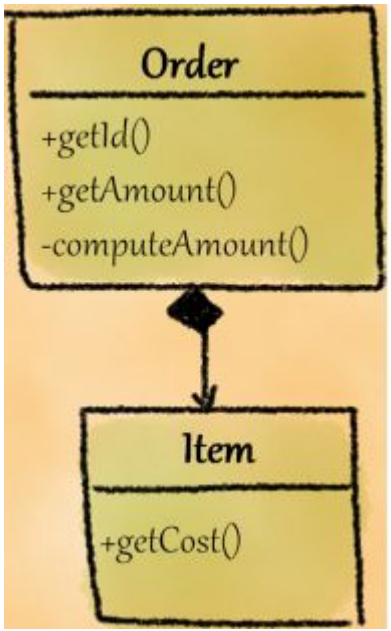
Example description

Consider an order-processing module in an e-commerce application. In this application, assume that you have two classes named Order and OrderImpl that provide support for order processing . The Order class maintains information about an order and the associated information about the ordered items. The method getAmount() in Order class uses computeAmount() method of OrderImpl class. In turn, the computeAmount() method extracts all the items associated with the Order object and computes the sum of costs of all the items that are a part of the order. In effect, classes Order and OrderImpl depend on each other, hence the design has the Cyclically-dependent Modularization smell.

SUGGESTED REFACTORING

- **Option 1**: Introduce an interface for one of the abstractions involved in the cycle.
- **Option 2**: In case one of the dependencies is unnecessary and can be safely removed, then remove that dependency. For instance, apply “move method” (and “move field”) refactoring to move the code that introduces cyclic dependency to one of the participating abstractions.
- **Option 3**: Move the code that introduces cyclic dependency to an altogether different abstraction.
- **Option 4**: In case the abstractions involved in the cycle represent a semantically single object, merge the abstractions into a single abstraction.

Solution for the Example



OrderImpl class could be merged into the Order class

IMPACTED QUALITY ATTRIBUTES

- Understandability
- Changeability and Extensibility
- Reusability
- Testability
- Reliability

ALIASES

- Dependency cycles
- Cyclic dependencies
- Cycles
- Bidirectional relation
- Cyclic class relationships

PRACTICAL CONSIDERATIONS

- **Unit cycles between conceptually related abstractions**

Cycles of size one (i.e., cycles that consist of exactly two abstractions) are known as unit cycles. Often, unit cycles are formed between conceptually related pairs.

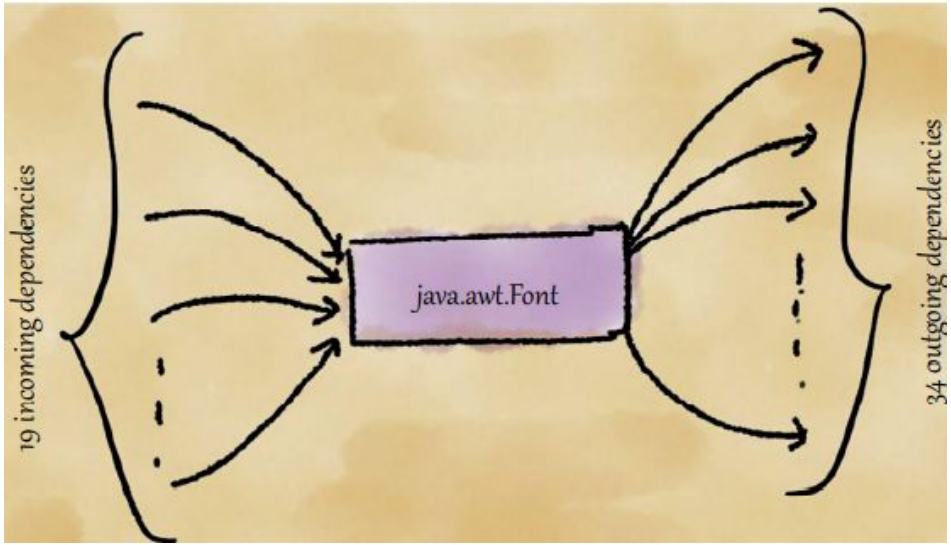
For instance, consider the classes Matcher and Pattern (both part of java.util.regex package) that are cyclically-dependent on each other.

These two classes are almost always understood, used, or reused together. In real-world projects, it is common to find such unit cycles.

However, since the unit cycle is small, it is easier to understand, analyze, and implement changes within the two abstractions that are part of the cycle.

HUB-LIKE MODULARIZATION

This smell arises when an abstraction has dependencies (both incoming and outgoing) with a large number of other abstractions.



Rationale

- “High cohesion and low coupling” is the basis for effective modularization. Meyer’s “few interfaces” rule for modularity says that “every module should communicate with as few others as possible.”
- For effective modularization, we must follow the enabling technique “limit dependencies.”
- When an abstraction has large number of incoming and outgoing dependencies, the principle of modularization is violated.
- When an abstraction has a large number of incoming dependencies and outgoing dependencies, the dependency structure looks like a hub, hence we name this smell Hub-like Modularization.

POTENTIAL CAUSES

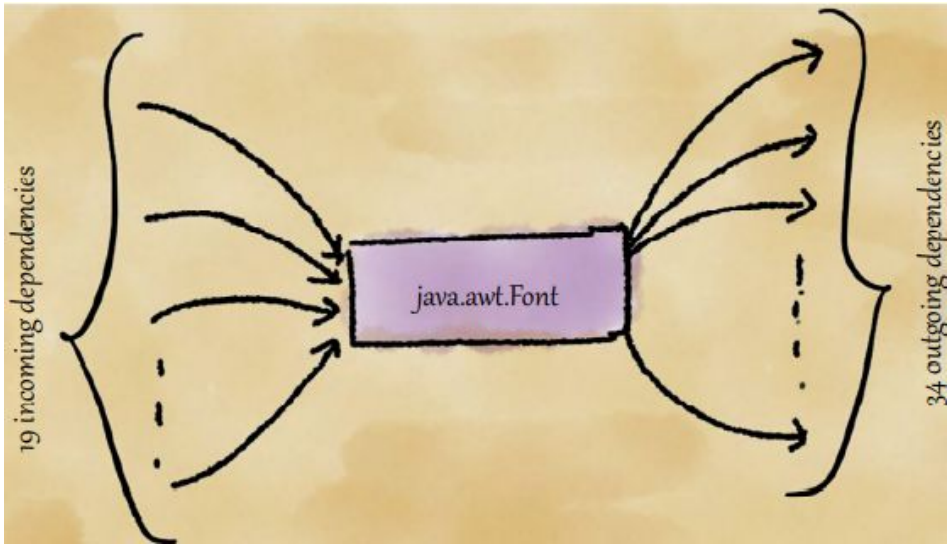
- **Improper responsibility assignment (to a hub class)**

When an abstraction is overloaded with too many responsibilities, it tends to become a hub class with a large number of incoming as well as outgoing dependencies. In other words, most classes in the design will talk to this hub class, and this hub class also communicates with most other classes.

- **“Dump-all” utility classes**

  Classes that provide supporting functionality often grow very large and get coupled with a large number of other classes.

Example


19 incoming dependencies
java.awt.Font
34 outgoing dependencies

Example Description

The java.awt.Font class represents fonts and supports functionality to render text as a sequence of glyphs on Graphics and Component objects. This class has 19 incoming dependencies and 34 outgoing dependencies.

Note that these do not include dependencies from method implementations within this class.

SUGGESTED REFACTORING

- If the hub class has multiple responsibilities, indicating improper responsibility assignment, the refactoring suggestion is to split up the responsibilities across multiple new/old abstractions so that the number of incoming and outgoing dependencies is reduced.
- If the dependencies are caused due to misplaced members in the hub class, the refactoring suggestion is to assign those members to appropriate abstractions.
- Sometimes the Chain of Responsibility pattern can be used to reduce the number of incoming dependencies on a hub class.

Refactoring for Example

The java.awt.Font class has some members that could be assigned to more suitable classes.

```
public GlyphVector createGlyphVector(FontRenderContext frc.
CharacterIterator ci) {
  return (GlyphVector)new StandardGlyphVector(this, ci, frc);
}
```

Although, conceptually, glyphs are related to fonts, glyph-related functionality need not belong to Font class, and could be moved to GlyphVector class

IMPACTED QUALITY ATTRIBUTES

//Refer pg no: 122

- Understandability
- Changeability, Extensibility, and Reliability
- Reusability and Testability

ALIASES

- Bottlenecks
- Local hubs
- Man-in-the-middle

PRACTICAL CONSIDERATIONS

Core abstractions

If you analyze large object oriented applications, you will find that there are a few “core abstractions” that play a central role in the design. From our experience, we find that it is usually difficult to keep the number of incoming as well as outgoing dependencies low for such core abstractions.

For instance, consider the class java.lang.Class in JDK. This class represents all classes and interfaces in a Java application, and hence is a core abstraction in JDK. Since the java.lang.Class has more than 1000 incoming dependencies and 40 outgoing dependencies, it has Hub-like Modularization smell. However, limiting the number of incoming and outgoing dependencies of such core abstractions is difficult

ABSTRACTION SMELLS

What happens when we violate Abstraction?

Design smells:

1. Missing Abstraction
2. Imperative Abstraction
3. Incomplete Abstraction
4. Multifaceted Abstraction
5. Unnecessary Abstraction
6. Unutilized Abstraction
7. Duplicate Abstraction

MISSING ABSTRACTION

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

Rationale

An enabling technique for applying the principle of abstraction is to create entities with crisp conceptual boundaries and a unique identity. Since the abstraction is not explicitly identified and rather represented as raw data using primitive types or encoded strings, the principle of abstraction is clearly violated; hence, we name this smell Missing Abstraction.

Potential Causes

- Inadequate design analysis
  - this often occurs when software is developed under tight deadlines or resource constraints.
- Lack of refactoring
  - As requirements change, software evolves and entities that were earlier represented using strings or primitive types may need to be refactored into classes or interfaces. When the existing clumps of data or encoded strings are retained as they are without refactoring them, it can lead to a Missing Abstraction smell.
- Misguided focus on minor performance gains

Example

From 1.0 version of Java, the stack trace was printed to the standard error stream as a string with `printStackTrace()` method.

Client programs that needed programmatic access to the stack trace elements had to write code to process the stack trace to get, for example, the line numbers or find if the bug is a duplicate of another already-filed bug. Due to this dependence of client programs on the format of the string, the designers of JDK were forced to retain the string encoding format in future versions of JDK.

Suggested Refactoring

```
public final class StackTraceElement {

public String getFileName();

public int getLineNumber();

public String getClassName();

public String getMethodName();

public boolean isNativeMethod();

}
```

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. For instance, if you see class with name Read that contains only one method named read() with no data members, then the Read class has Imperative Abstraction smell.

Rationale

The founding principle of object-orientation is to capture real world objects and represent them as abstractions.

By following the enabling technique map domain entities, objects recognized in the problem domain need to be represented in the solution domain, too. Furthermore, each class representing an abstraction should encapsulate data and the associated methods. Defining functions or procedures explicitly as classes (when the data is located somewhere else) is a glorified form of structured programming rather than object-oriented programming.

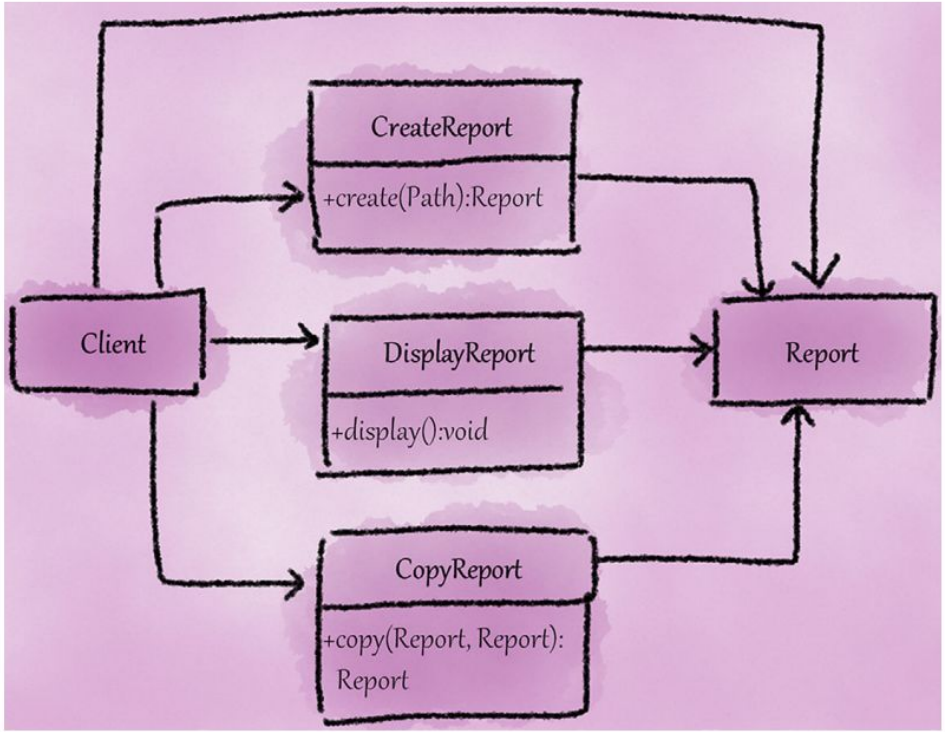
One-operation classes cannot be representative of an “abstraction,” especially when the associated data is placed somewhere else. Clearly, this is a violation of the principle of abstraction. Since these classes are ‘doing’ things instead of ‘being’ things, this smell is named Imperative Abstraction

POTENTIAL CAUSES

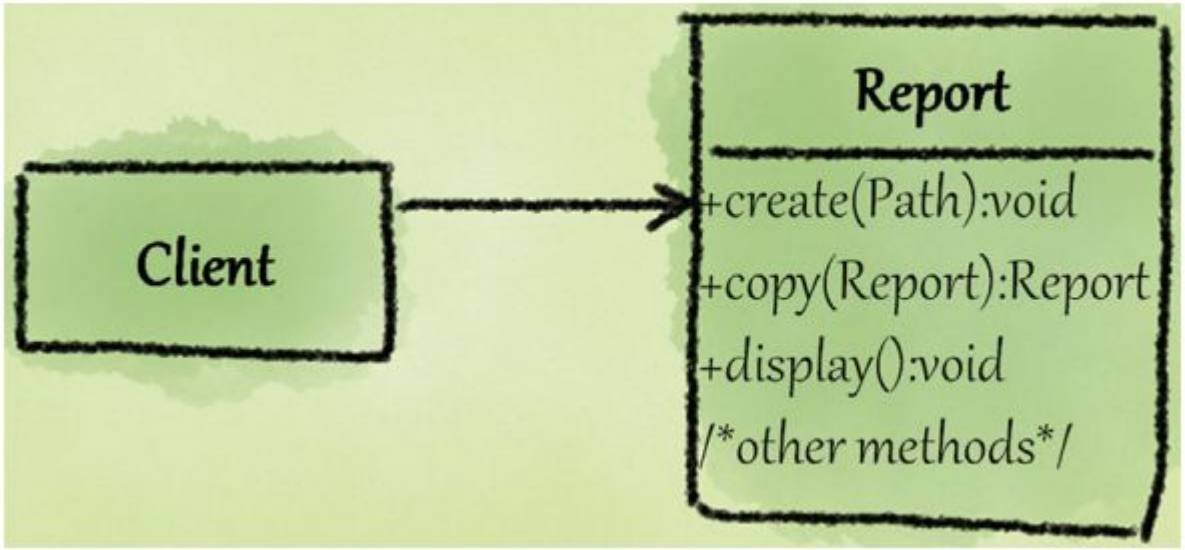
Procedural thinking

Procedural thinking can be a potential cause for the imperative abstraction smell because it often leads to code that is difficult to maintain and understand. Procedural thinking focuses on the steps required to perform a task, rather than on the data and operations that are required to achieve the desired outcome. This can result in code that is difficult to read and maintain, as it can be hard to understand the purpose and context of each step.

Example


CreateReport
+create(Path):Report
Client
DisplayReport
+display():void
Report
CopyReport
+copy(Report, Report):
Report

Suggested Refactoring



INCOMPLETE ABSTRACTION

This smell arises when an abstraction does not support complementary or interrelated methods completely. For instance, the public interface of an abstraction may provide an initialize() method to allocate resources. However, a dispose() method that will allow clean-up of the resources before it is deleted or re-collected may be missing in the abstraction. In such a case, the abstraction suffers from Incomplete Abstraction smell because complementary or interrelated methods are not provided completely in its public interface.

Rationale

one of the interrelated methods is implemented somewhere else, cohesion is compromised and the Single Responsibility Principle is violated. Since complementary or interrelated methods are not provided completely in the abstraction, this smell is named Incomplete Abstraction

Potential Causes

- Missing overall perspective
- Lack of separation of concerns:
  - Classes that try to do too much or that have too many responsibilities can lead to incomplete abstractions. This can make it more difficult to maintain the code and extend it in the future.
- Overuse of primitive data types:
  - Using primitive data types instead of more abstract data types can also lead to incomplete abstractions. Primitive data types may be limited in their ability to express complex ideas or to represent complex data structures.

Example

example of incomplete abstraction in the ButtonModel class is the **isSelected()** method. This method is used to determine whether a button is currently selected or not. However, this method only provides information about the selected state, and does not provide a complete abstraction of all possible button states..

Suggested Refactoring

To address these issues, a more complete abstraction of button state could include methods such as **isDisabled()**, **isReleased()**, and **isHovered()**, which would provide information about all possible button states. This would make the ButtonModel class a more complete abstraction of button state, and make it easier for developers to work with button controls in GUI applications.

MULTIFACETED ABSTRACTION

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

Rationale

An important enabling technique to effectively apply the principle of abstraction is to assign single and meaningful responsibility for each abstraction. In particular, the Single Responsibility Principle says that an abstraction should have a single well-defined responsibility and that responsibility should be entirely encapsulated within that abstraction. An abstraction that is suffering from Multifaceted Abstraction has more than one responsibility assigned to it, and hence violates the principle of abstraction.

POTENTIAL CAUSES

General-purpose abstractions

Evolution without periodic refactoring

The burden of processes

Mixing up concerns

Example

java.util.Calendar class as an example of a class having multiple responsibilities [64]. A class abstracting real-world calendar functionality is expected to support dates, but the java.util.Calendar class supports time related functionality as well, and hence this class suffers from Multifaceted Abstraction smell.

SUGGESTED REFACTORING

For the Calendar class, a possible refactoring is to extract time-related functionality from the Calendar class into a new Time class and move the relevant methods and fields into the newly extracted class. Java 8 has introduced new classes supporting date and time (and other classes such as clocks, duration, etc.) in a package named java.time so that future clients can use this new package instead.

UNNECESSARY ABSTRACTION

This smell occurs when an abstraction that is actually not needed (and thus could have been avoided) gets introduced in a software design.

Rationale

A key enabling technique to apply the principle of abstraction is to assign single and meaningful responsibility to entities. However, when abstractions are created unnecessarily or for mere convenience, they have trivial or no responsibility assigned to them, and hence violate the principle of abstraction. Since the abstraction is needlessly introduced in the design, this smell is named Unnecessary Abstraction.

Potential Causes

- Procedural thinking in object-oriented languages
- Using inappropriate language features for convenience
  - Using “constant interfaces” instead of enums, for example, is convenient for programmers. By implementing constant interfaces in classes, programmers don’t have to explicitly use the type name to access the members and can directly access them in the classes. It results in unnecessary interfaces or classes that just serve as placeholders for constants without any behavior associated with the classes.
- Over-engineering
  - Sometimes, this smell can occur when a design is over-engineered. Consider the example of a customer ID associated with a Customer object in a financial application. It may be overkill to create a class named CustomerID because the CustomerID object would merely serve as holder of data and will not have any non-trivial or meaningful behavior associated with it. A better design choice in this case would be to use a string to store the customer ID within a Customer object.

Example

Consider the case of an e-commerce application that has two classes: namely, BestSellerBook and Book. Whenever the client wants to create a best-seller book, it creates an instance of a BestSellerBook. Internally, BestSellerBook delegates all the method calls to the Book class and does nothing else. Clearly, the BestSellerBook abstraction is unnecessary since its behavior is exactly the same as the Book abstraction.

Refactoring : Simply remove it 👍

Unutilized Abstraction

This smell arises when an abstraction is left unused (either not directly used or not reachable). This smell manifests in 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

Rationale

One of the enabling techniques for applying the principle of abstraction is to assign a single and meaningful responsibility to an entity. When an abstraction is left unused in design, it does not serve a meaningful purpose in design, and hence violates the principle of abstraction.

Design should serve real needs and not imagined or speculative needs. Unrealized abstract classes and interfaces indicate unnecessary or speculative generalization, and hence are undesirable. This smell violates the principle YAGNI (You Aren’t Gonna Need It), which recommends not adding functionality until deemed necessary . Since the abstraction is left unutilized in the design, this smell is named Unutilized Abstraction.

POTENTIAL CAUSES

Changing requirements

Leftover garbage during maintenance

Fear of breaking code

Speculative design

Example and Refactoring

```
public interface Shape {
    public void draw();
    public void resize();
    public void move();
}
public class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing a rectangle");
    }
}
```

Refactoring : Simply remove it 👍

DUPLICATE ABSTRACTION

The Duplicate Abstraction smell occurs when two or more classes or interfaces have the same abstraction level and contain similar or identical methods or properties. This can lead to duplication of code and maintenance issues, as changes made to one class may need to be replicated across other similar classes.

POTENTIAL CAUSES

Copy-paste programming

Ad hoc maintenance

Lack of communication

Classes declared non-extensible

Example

There are three different classes with the name Timer in JDK. The description of these classes as per JDK documentation is as follows:

• javax.swing.Timer is meant for executing objects of type ActionEvent at specified intervals.

• java.util.Timer is meant for scheduling a thread to execute in the future as a background thread.

• javax.management.timer.Timer is meant for sending out an alarm to wake up the listeners who have registered to get timer notifications.

Since all three timers have similar descriptions (they execute something in future after a given time interval), choosing the right Timer class can be challenging to inexperienced programmers.

Suggested Refactoring

For the Timer class example, since the main concern is identical name, the refactoring suggestion is to change their names so that they are unique. javax.swing.Timer is meant for executing objects of type ActionEvent, so it can be renamed as ExecutionTimer or EventTimer. Similarly, the Timer implementation from java.util can be renamed to AlarmTimer. The java.util.Timer can be retained as it is.

THANK YOU