

# Code Readability

Munetoshi Ishikawa

Code readability session 1

# Introduction and Principles

# What readable code is

- **Obvious:** isVisible rather than flag
- **Simple:** isA && isB rather than !(isA || !isB) && isB
- **Isolated:** responsibility and dependency
- **Structured:** format, members, interactions, and abstraction layer

# Why we need readable code

# Why we need readable code

## Sustainable development

- To extend saturation curve of development

# Why we need readable code

## Sustainable development

- To extend saturation curve of development

## Reading code > Writing code

- Requesting code review from two or more engineers
- Complicated bug fix with a few lines

# Optimize for sustainable development

Focus on team productivity

# Optimize for sustainable development

Focus on team productivity

Your 5 minutes could save 1 hour for others

Add a comment, write test, refactor

# Optimize for sustainable development

Focus on team productivity

Your 5 minutes could save 1 hour for others

Add a comment, write test, refactor

We may need to update personnel rating criteria

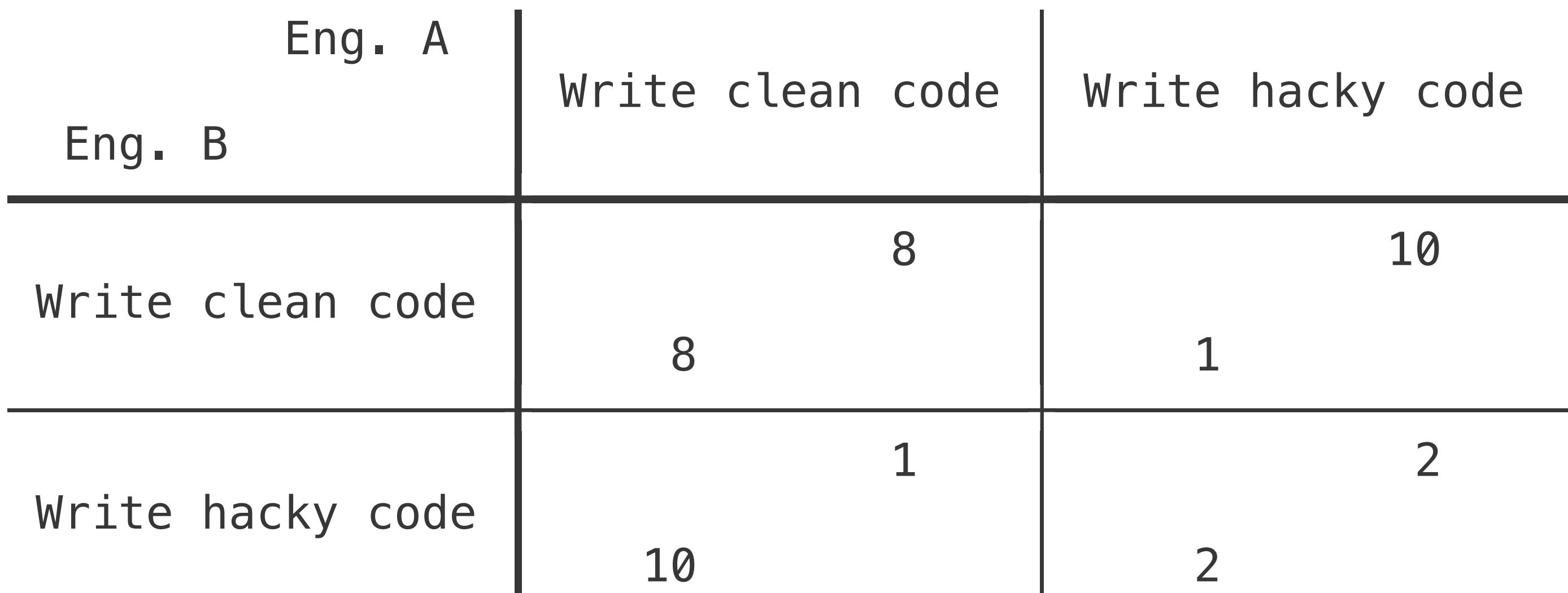
Don't focus only on short-term speed to implement

# Prisoner's dilemma

Team productivity may decrease if we focus on the personal

# Prisoner's dilemma

Team productivity may decrease if we focus on the personal



# Learn how to write readable code

Feature implementation itself is easy

Special training is not required

# Learn how to write readable code

Feature implementation itself is easy

Special training is not required

No learning, No readable code

- Training, lectures
- Peer code review
- Self-education

# Contents of this lecture

- Introduction and Principles
- Natural language: Naming, Comments
- Inner type structure: State, Procedure
- Inter type structure: Dependency (two sessions)
- Follow-up: Review

# Contents of this lecture

- Introduction and Principles
- Natural language: Naming, Comments
- Inner type structure: State, Procedure
- Inter type structure: Dependency (two sessions)
- Follow-up: Review

# Topics

- Introduction
- The boy scout rule
- YAGNI
- KISS
- Single responsibility principle
- Premature optimization is the root of all evil

# Topics

- Introduction
- The boy scout rule
- YAGNI
- KISS
- Single responsibility principle
- Premature optimization is the root of all evil

# The boy scout rule

Try to leave this world a little better than you found it...

— Robert Baden-Powell

# The boy scout rule

Try to leave this world a little better than you found it...

— Robert Baden-Powell

Introduced to software development by Robert C. Martin<sup>1</sup>

Clean code whenever you have touched it

---

<sup>1</sup> 97 Things Every Programmer Should Know: Collective Wisdom from the Experts, Kevlin Henney, 2010

# Dos for the boy scout rule

- Add: comments, tests
- Remove: unnecessary dependencies, members, and conditions
- Rename: types, functions, and values
- Break: huge types, huge functions, nests, and call sequences
- Structure: dependencies, abstraction layers, and type hierarchy

# Don'ts for the boy scout rule 1/2

## Don't add an element in a huge structure

- Sentence in a huge method
- Case in a huge conditional branch
- Member in a huge type
- Callback in a huge call sequence
- Inheritance in a huge hierarchy

# Don'ts for the boy scout rule 2/2

Don't add something without thought

# Don'ts for the boy scout rule 2/2

Don't add something without thought

Is the code location correct?

- Consider restructuring before adding a conditional branch

# Don'ts for the boy scout rule 2/2

Don't add something without thought

Is the code location correct?

- Consider restructuring before adding a conditional branch

Can't you simplify?

- Merge copied values or conditions
- Look around where you want to change

# Example of "don'ts" 1/2

**Question:** Is it fine to add a new type Z?

```
val viewType: ViewType = ... // An enum type
when (viewType) {
    A -> {
        view1.isVisible = true
        view2.text = "Case A"
    }
    B -> {
        view1.isVisible = false
        view2.text = "Case B"
    }
    ...
}
```

# Example of "don'ts" 2/2

Answer: No

## Example of "don'ts" 2/2

Answer: No

Don't add a new condition if there are too many branches

## Example of "don'ts" 2/2

**Answer: No**

Don't add a new condition if there are too many branches

**Solution:** Apply strategy pattern

# Example of "dos"

1: Extract parameters as constructor parameters  
(or computed properties)

```
enum class ViewType(val isVisible1Visible: Boolean, val view2Text: String)
```

# Example of "dos"

- 1: Extract parameters as constructor parameters  
(or computed properties)

```
enum class ViewType(val isVisible: Boolean, val view2Text: String)
```

- 2: Remove conditional branches with the parameters

```
view1.isVisible = viewType.isVisible  
view2.text = viewType.view2Text
```

# Example of "dos"

- 1: Extract parameters as constructor parameters  
(or computed properties)

```
enum class ViewType(val isVisible: Boolean, val view2Text: String)
```

- 2: Remove conditional branches with the parameters

```
view1.isVisible = viewType.isVisible  
view2.text = viewType.view2Text
```

- 3: Add a new type Z.

# Topics

- Introduction
- The boy scout rule
- YAGNI
- KISS
- Single responsibility principle
- Premature optimization is the root of all evil

# YAGNI

You Aren't Gonna Need It

= Implement it only when you need it

- 90% of features for the future are not used<sup>2</sup>
- Keep structure simple = ready for unexpected change<sup>2</sup>

---

<sup>2</sup> <http://www.extremeprogramming.org/rules/early.html>

# YAGNI

You Aren't Gonna Need It

= Implement it only when you need it

- 90% of features for the future are not used<sup>2</sup>
- Keep structure simple = ready for unexpected change<sup>2</sup>

Exception: Public library for external people

---

<sup>2</sup> <http://www.extremeprogramming.org/rules/early.html>

# YAGNI: Example

- Unused types, procedures and values
- Interface or abstract class with only one implementation
- Function parameter for a constant value
- Public global utility function only for one client code
- Code commented out

# Topics

- Introduction
- The boy scout rule
- YAGNI
- KISS
- Single responsibility principle
- Premature optimization is the root of all evil

# KISS

Keep It Simple Stupid

– Clarence Leonard "Kelly" Johnson

Choose simpler solution

- Limit and specify the usage of a library/framework/design
- Use the default implementation as far as possible

# KISS

Keep It Simple Stupid

– Clarence Leonard "Kelly" Johnson

Choose simpler solution

- Limit and specify the usage of a library/framework/design
- Use the default implementation as far as possible

Beautiful code is not always readable

# KISS: Good example

```
fun getActualDataSingle(): Single<List<Int>> = Single  
    .fromCallable(dataProvider::provide)  
    .subscribeOn(ioScheduler)
```

# KISS: Good example

```
fun getActualDataSingle(): Single<List<Int>> = Single  
    .fromCallable(dataProvider::provide)  
    .subscribeOn(ioScheduler)
```

```
fun getDummyDataSingle(): Single<List<Int>> =  
    Single.just(listOf(1, 10, 100))
```

# KISS: Bad example 1/2

```
fun getActualDataSingle(): Single<List<Int>> = Single  
    .fromCallable(dataProvider::provide)  
    .subscribeOn(ioScheduler)
```

```
fun getDummyDataSingle(): Single<List<Int>> = Single  
    .fromCallable { listOf(1, 10, 100) }  
    .subscribeOn(ioScheduler)
```

Makes the code complex for unnecessary consistency

# KISS: Bad example 2/2

```
fun getActualDataSingle(): Single<List<Int>> = Single  
    .fromCallable(dataProvider::provide)  
    .subscribeOn(ioScheduler)
```

```
fun getDummyDataSingle(): Single<List<Int>> = Observable  
    .range(1, 2)  
    .reduce(listOf(1)) { list, _ -> list + list.last() * 10 }  
    .subscribeOn(ioScheduler)
```

Uses Rx for unnecessary list creation  
(Rx was used only for changing the thread.)

# Topics

- Introduction
- The boy scout rule
- YAGNI
- KISS
- Single responsibility principle
- Premature optimization is the root of all evil

# Single method == small responsibility?

# Single method == small responsibility?

No, definitely!

```
class Alviss {  
    // May show a text, may break the device, may launch a rocket,  
    // may ...  
    fun doEverything(state: UniverseState)  
}
```

# Single responsibility principle

A class should have only one reason to change.

— Robert C. Martin

# Single responsibility principle

A class should have only one reason to change.

— Robert C. Martin

We should not mix up two unrelated features

# Single responsibility principle: Bad example

```
class LibraryBookRentalData(  
    val bookIds: MutableList<Int>,  
    val bookNames: MutableList<String>,  
) {  
    ...  
}
```

# Single responsibility principle: Bad example

```
class LibraryBookRentalData(  
    val bookIds: MutableList<Int>,  
    val bookNames: MutableList<String>,  
    val bookIdToRenterNameMap: MutableMap<Int, String>,  
) {  
    ...  
}
```

# Single responsibility principle: Bad example

```
class LibraryBookRentalData(  
    val bookIds: MutableList<Int>,  
    val bookNames: MutableList<String>,  
    val bookIdToRenterNameMap: MutableMap<Int, String>,  
    val bookIdToDueDateMap: MutableMap<Int, Date>, ...  
) {  
    ...  
}
```

# Single responsibility principle: Bad example

```
class LibraryBookRentalData(  
    val bookIds: MutableList<Int>,  
    val bookNames: MutableList<String>,  
    val bookIdToRenterNameMap: MutableMap<Int, String>,  
    val bookIdToDueDateMap: MutableMap<Int, Date>, ...  
) {  
    fun findRenterName(bookName: String): String?  
    fun findDueDate(bookName: String): Date?  
    ...  
}
```

# Single responsibility principle: What is wrong

# Single responsibility principle: What is wrong

BookRentalData has all data of book, user, and circulation record

# Single responsibility principle: What is wrong

BookRentalData has all data of book, user, and circulation record

**Split a model class for each entity**

# Single responsibility principle: Good example

```
data class BookData(val id: Int, val name: String, ...)  
data class UserData(val name: String, ...)
```

# Single responsibility principle: Good example

```
data class BookData(val id: Int, val name: String, ...)  
data class UserData(val name: String, ...)  
  
class CirculationRecord(  
    val onLoanBookEntries: MutableMap<BookData, Entry>  
) {  
    data class Entry(val renter: UserData, val dueDate: Date)}
```

# Keep responsibility small

## Split types

- Model for each entity
- Logic for each layer and component
- Utility for each target type

# How to confirm responsibility

List what the type does, and try to summarize it

# How to confirm responsibility

List what the type does, and try to summarize it

Split the type for each case as follows

- It's hard to make a summary
- The summary is too fat compared with the name

# Topics

- Introduction
- The boy scout rule
- YAGNI
- KISS
- Single responsibility principle
- Premature optimization is the root of all evil

# Premature optimization 1/2

We should forget about small efficiencies,  
say about 97% of the time:

**premature optimization is the root of all evil.**

– Structured Programming with go to Statements, Donald Knuth

# Premature optimization 2/2

Good: if the optimization **cleans** the code

Bad: if the optimization **makes** the code complex

# Example of good optimization 1/2

Before:

```
val data = arrayList.firstOrNull { data -> data.key == expectedKey }
```

# Example of good optimization 1/2

Before:

```
val data = arrayList.firstOrNull { data -> data.key == expectedKey }
```

After:

```
val data = hashMap.getOrNull(key)
```

# Example of good optimization 1/2

Before:

```
val data = arrayList.firstOrNull { data -> data.key == expectedKey }
```

After:

```
val data = hashMap.getOrDefault(key)
```

Simplify the code while reducing the calculation cost

# Examples of bad optimization

Don't optimize without **profiling** or **platform support**, either

- Mutable instance reusing
- Lazy initialization
- Cache
- Inline extraction
- Instance pool

# Drawbacks of optimization

- May obstruct simplification
  - Compiler is sometimes smarter than us
- May require overhead cost
  - Lazy initialization: (Synchronized) instance check
  - Cache: cache miss ratio \* cache access time

# Required actions before optimization

- Ask yourself  
"Do I really need it?" or "Is it easy enough to implement?"
- Profile  
Target (time, memory), number, rate

# Summary

Focus on sustainable development: Readability

# Summary

Focus on sustainable development: Readability

The boy scout rule: Clean code whenever you have touched it

# Summary

Focus on sustainable development: Readability

The boy scout rule: Clean code whenever you have touched it

**YAGNI:** Implement only when you need it

# Summary

Focus on sustainable development: Readability

The boy scout rule: Clean code whenever you have touched it

**YAGNI:** Implement only when you need it

**KISS:** Use simple method, beautiful != readable

# Summary

Focus on sustainable development: Readability

The boy scout rule: Clean code whenever you have touched it

**YAGNI:** Implement only when you need it

**KISS:** Use simple method, beautiful != readable

**Single responsibility principle:** Make the scope clear

# Summary

Focus on sustainable development: Readability

The boy scout rule: Clean code whenever you have touched it

**YAGNI:** Implement only when you need it

**KISS:** Use simple method, beautiful != readable

**Single responsibility principle:** Make the scope clear

**Premature optimization:** Profile or estimate before optimization

codeReadabilitySession2

N\_A\_M\_I\_N\_G

# Contents of this lecture

- Introduction and Principles
- Natural language: Naming, Comments
- Inner type structure: State, Procedure
- Inter type structure: Dependency (two sessions)
- Follow-up: Review

# What we name

- **Type**: class, interface, enum, struct, protocol, trait
- **Value**: property, field, parameter, local value
- **Procedure**: function, method, subroutine
- **Scope**: package, module, namespace
- **Resource**: file, directory, ID
- etc.

# What a good name is

- **Accurate**  
isVisible shouldn't be used for sound
- **Descriptive**  
width/height rather than w/h
- **Unambiguous**  
imageView/imageBitmap/imageUrl rather than image

# How to create a good name

- Use correct grammar
- Describe what rather than who/when
- Choose unambiguous words
- Avoid confusing abbreviations
- Add type/unit suffix
- Use positive affirmation

# Topics

- Use correct grammar
- Describe what rather than who/when
- Choose unambiguous words
- Avoid confusing abbreviations
- Add type/unit suffix
- Use positive affirmation

# Why grammar is important

Question 1: What is ListenerEventMessageClickViewText?

# Why grammar is important

Question 1: What is ListenerEventMessageClickViewText?

Answer?: A text of a click view (?) of listener event message (???)

# Why grammar is important

Question 1: What is ListenerEventMessageClickViewText?

Answer?: A text of a click view (?) of listener event message (???)

Question 2: What is MessageTextViewClickEventListener?

# Why grammar is important

Question 1: What is ListenerEventMessageClickViewText?

Answer?: A text of a click view (?) of listener event message (???)

Question 2: What is MessageTextViewClickEventListener?

Answer: A listener of click events on a message text view

# Use correct grammar

"Grammar" depends on the language and convention

Let's look at the case of Java/Kotlin

# Types of names 1/2

- **Noun:** Type, value (including property function)  
imageView, HashSet, indexOf
- **Imperative:** Procedure  
findMessage, compareTo

# Types of names 2/2

- **Adjective, participle:** Interface, state type/constant  
Iterable, PLAYING, CONNECTED
- **Interrogative, third-person verb:** Boolean value/function  
isTextVisible, contains, equalsTo, may/shouldShow
- **Adverb phrase with preposition:** Converter, callback  
toInt, fromMemberId, onNewIntent

# Grammar: Nouns

Place the essential words at the last  
 (= the word telling what it is)

# Grammar: Nouns

Place the essential words at the last  
 (= the word telling what it is)

: MessageEventHandler, buttonHeight

# Grammar: Nouns

Place the essential words at the last  
 (= the word telling what it is)

😊: MessageEventHandler, buttonHeight

☹️: xyzHeightForPortlait (don't use for a type name)

# Grammar: Nouns

Place the essential words at the last  
 (= the word telling what it is)

: MessageEventHandler, buttonHeight

: xyzHeightForPortlait (don't use for a type name)

: HandlerMessageEvent (if it's a handler), heightPortlait

# Grammar: Nouns

Place the essential words at the last  
 (= the word telling what it is)

: MessageEventHandler, buttonHeight

: xyzHeightForPortlait (don't use for a type name)

: HandlerMessageEvent (if it's a handler), heightPortlait

**Exception:** Property function with a preposition  
e.g., indexOf(value), maxValueIn(array)

# Grammar: Imperative

Place a verb at the first

# Grammar: Imperative

Place a verb at the first

"get X":  getX,  xGet

# Grammar: Imperative

Place a verb at the first

"get X":  getX,  xGet

"post Y":  postY,  yPost

# Grammar: Imperative

Place a verb at the first

"get X":  getX,  xGet

"post Y":  postY,  yPost

"map to Z":  mapToZ,  toZMap, zToMap

# How a name in a wrong order is created

```
class UserActionEvent
```

# How a name in a wrong order is created

```
class UserActionEvent  
  
class UserActionEventSwipe: UserActionEvent()  
class UserActionEventClick: UserActionEvent()
```

# How a name in a wrong order is created

```
class UserActionEvent
```

```
class UserActionEventSwipe: UserActionEvent()
```

```
class UserActionEventClickMessageText: UserActionEvent()
```

```
class UserActionEventClickProfileImage: UserActionEvent()
```

# How a name in a wrong order is created

```
class UserActionEvent
```

```
class UserActionEventSwipe: UserActionEvent()
```

```
class UserActionEventClickMessageText: UserActionEvent()
```

```
class UserActionEventClickProfileImage: UserActionEvent()
```

Don't focus on **comprehensiveness** and **consistency** too much

Imagine how it looks on the caller side

# Grammar: Summary

- Types for a name  
Noun, imperative, adjective, interrogative (+ third person), adverb
- Word order is important  
Imagine how the name looks on the caller side

# Topics

- Use the correct grammar
- Describe what rather than who/when
- Choose unambiguous words
- Avoid confusing abbreviations
- Add type/unit suffix
- Use positive affirmation

# Describe what rather than who/when

- A name must answer
  - 👉 **What** a type/value is
  - 👉 **What** a procedure does

# Describe what rather than who/when

- A name must answer
  - 👍 **What** a type/value is
  - 👍 **What** a procedure does
- A name should not mention to the caller
  - 👎 **Who** it calls/uses
  - 👎 **When/Where/Why/How** it's called/used

# Function name example: Declaration

Good 👍: Describe **what** this procedure does

```
class MessageRepository {  
    fun storeReceivedMessage(data: MessageData) {
```

# Function name example: Declaration

Good : Describe **what** this procedure does

```
class MessageRepository {  
    fun storeReceivedMessage(data: MessageData) {
```

Bad : Describe **when** this procedure should be called

```
class MessageRepository {  
    fun onMessageReceived(data: MessageData) {
```

# Function name example: Caller code

Good 👍: We can know what happens by the calling code

```
repository.storeReceivedMessage(messageData)  
handler.post { presenter.showNewReceivedMessage(messageData) }
```

# Function name example: Caller code

Good : We **can know what** happens by the calling code

```
repository.storeReceivedMessage(messageData)  
handler.post { presenter.showNewReceivedMessage(messageData) }
```

Bad : We **can't know what** happens

```
repository.onMessageReceived(messageData)  
handler.post { presenter.onMessageReceived(messageData) }
```

# Function name example: Another bad reason

The procedure responsibility is ambiguous

# Function name example: Another bad reason

The procedure responsibility is ambiguous

```
class MessageViewPresenter {  
    fun onMessageReceived(data: MessageData) {  
        // View presentation code for new message ...  
    }  
}
```

# Function name example: Another bad reason

The procedure responsibility is ambiguous

```
class MessageViewPresenter {  
    fun onMessageReceived(data: MessageData) {  
        // View presentation code for new message ...  
        launch(...) { repository.onMessageReceived(data) } // !!
```

# Function name example: Another bad reason

The procedure responsibility is ambiguous

```
class MessageViewPresenter {  
    fun onMessageReceived(data: MessageData) {  
        // View presentation code for new message ...  
        launch(...) { repository.onMessageReceived(data) } // !!
```

What happens with the following code?

```
repository.onMessageReceived(messageData)  
handler.post { presenter.onMessageReceived(messageData) }
```

# Parameter name example: Declaration

Good 👍: We can know what happens if it's true

```
fun showHistory(shouldShowDialogOnError: Boolean)
```

# Parameter name example: Declaration

Good : We can know what happens if it's true

```
fun showHistory(shouldShowDialogOnError: Boolean)
```

Bad : We can't know what happens if it's true

```
fun showHistory(isCalledFromMainActivity: Boolean)
```

# Parameter name example: Reason of "bad"

1: The name easily becomes obsolete

```
// ... from another activity requiring dialog  
showHistory(isCalledFromMainActivity = true)
```

# Parameter name example: Reason of "bad"

1: The name easily becomes obsolete

```
// ... from another activity requiring dialog  
showHistory(isCalledFromMainActivity = true)
```

2: The parameter will be used for other purposes

```
if (isCalledFromMainActivity) { setActivityResult(...)
```

# Parameter name example: Reason of "bad"

1: The name easily becomes obsolete

```
// ... from another activity requiring dialog  
showHistory(isCalledFromMainActivity = true)
```

2: The parameter will be used for other purposes

```
if (isCalledFromMainActivity) { setActivityResult(...)
```

Causes a bug if 1 and 2 happens at the same time

# Describe what: Exception

May need to name by how/when for an abstract callback interface

- e.g., onClicked / onSwiped / onDestroyed

# Describe what: Exception

May need to name by how/when for an abstract callback interface

- e.g., onClicked / onSwiped / onDestroyed
- Because "what" is not decided on the declaration

# Describe what: Exception

May need to name by how/when for an abstract callback interface

- e.g., onClicked / onSwiped / onDestroyed
- Because "what" is not decided on the declaration

We should name by "what" if possible even for a callback

```
abstract fun toggleSelectionState()  
...  
view.setOnClickListener { toggleSelectionState() }
```

# Describe what: Summary

Describe "what" it does/is

= Should not mention to the caller (who/when/where/why/how)

# Describe what: Summary

Describe "what" it does/is

= Should not mention to the caller (who/when/where/why/how)

- Show code responsibility clearly
- Make caller code readable

# Describe what: Summary

**Describe "what" it does/is**

= Should not mention to the caller (who/when/where/why/how)

- Show code responsibility clearly
- Make caller code readable

**Exception: Abstract callback interface**

# Topics

- Use the correct grammar
- Describe what rather than who/when
- Choose unambiguous words
- Avoid confusing abbreviations
- Add type/unit suffix
- Use positive affirmation

# Ambiguous words: Example 1/2

**Question:** What does initializationFlag represent?

- shouldInitialize
- isInitializing
- is/wasInitialized
- isInitializable
- isNotInitialized (!!)

# Ambiguous words: Example 1/2

**Question:** What does initializationFlag represent?

- shouldInitialize
- isInitializing
- is/wasInitialized
- isInitializable
- isNotInitialized (!!)

**Choose one from the above options (Except for the last one)**

# Ambiguous words: Example 2/2

**Question:** What does sizeLimit represent?

- max or min?
- height, width, byte, characters, or length?

# Ambiguous words: Example 2/2

**Question:** What does sizeLimit represent?

- max or min?
- height, width, byte, characters, or length?

Name maxHeight, for example.

# Rewording options to consider

- **flag**: is, was, should, can, may, will ...
- **check**: is, query, verify, measure, filter, notify, update ...
- **good, fine**: valid, completed, reliable, secure, satisfies ...
- **old**: previous, stored, expired, invalidated, deprecated ...
- **tmp, retval**: actual name

# Rewording options to consider

- **flag**: is, was, should, can, may, will ...
- **check**: is, query, verify, measure, filter, notify, update ...
- **good, fine**: valid, completed, reliable, secure, satisfies ...
- **old**: previous, stored, expired, invalidated, deprecated ...
- **tmp, retval**: actual name

Use dictionary and thesaurus

# Choose unambiguous words: Summary

- Avoid words like flag and check
- Use a dictionary and thesaurus

# Topics

- Use the correct grammar
- Describe what rather than who/when
- Choose unambiguous words
- **Avoid confusing abbreviations**
- Add type/unit suffix
- Use positive affirmation

# Abbreviations: Example 1/2

**Question:** What does im stand for?

# Abbreviations: Example 1/2

**Question:** What does im stand for?

input method, illegal message, instance manager ...

# Abbreviations: Example 1/2

**Question:** What does im stand for?

input method, illegal message, instance manager ...

**Don't use your own acronyms**

- Recognizing is easier than recalling<sup>3</sup>

---

<sup>3</sup> 100 Things: Every Designer Needs to Know About People, Susan Weinschenk, 2011

# Abbreviations: Example 2/2

**Question:** What does str stand for?

# Abbreviations: Example 2/2

**Question:** What does str stand for?

string, structure, stream, streak, street, sorted transaction record

# Abbreviations: Example 2/2

**Question:** What does str stand for?

string, structure, stream, streak, street, sorted transaction record

**Abbreviations may be acceptable if commonly used**

- Abbreviations like URL and TCP are totally fine
- str for string is acceptable especially for a limited scope

# Project-specific abbreviation

Some abbreviations are used project-wide

# Project-specific abbreviation

Some abbreviations are used project-wide

Make readable for a new team member

- Write documentation for type, value, or procedure
- Prepare glossary

# Avoid confusing abbreviations: Summary

- Don't use your own abbreviations
- Commonly used abbreviations are acceptable
- Prepare document or glossary for project-specific abbreviations

# Topics

- Use the correct grammar
- Describe what rather than who/when
- Choose unambiguous words
- Avoid confusing abbreviations
- Add type/unit suffix
- Use positive affirmation

# Add type/unit suffix

Add suffix as a supplement type/unit

- **timeout**: timeoutMillis, timeoutHours
- **width**: widthPoints, widthPx, widthInches
- **color**: colorInt, colorResId
- **i, j, k**: xxxIndex, row, col

# Add type/unit suffix: Aside

Consider creating a wrapper class of a unit

# Add type/unit suffix: Aside

Consider creating a wrapper class of a unit

```
class Inch(val value: Int)  
class Centimeter(val value: Int)
```

# Add type/unit suffix: Aside

Consider creating a wrapper class of a unit

```
class Inch(val value: Int)  
class Centimeter(val value: Int)  
  
fun setWidth(width: Inch) = ...  
setWidth(Centimeter(10)) // Compile error!
```

# Add type/unit suffix: Aside

Consider creating a wrapper class of a unit

```
class Inch(val value: Int)  
class Centimeter(val value: Int)
```

```
fun setWidth(width: Inch) = ...  
setWidth(Centimeter(10)) // Compile error!
```

"value class" (Scala) or "inline class" (Kotlin) may help you

# Topics

- Use the correct grammar
- Describe what rather than who/when
- Choose unambiguous words
- Avoid confusing abbreviations
- Add type/unit suffix
- **Use positive affirmation**

# Use positive affirmation

: Positive words, isEnabled

# Use positive affirmation

: Positive words, isEnabled

: Negative words, isDisabled

# Use positive affirmation

: Positive words, isEnabled

: Negative words, isDisabled

: Positive words with "not", "no", or "non", isNotEnabled

# Use positive affirmation

: Positive words, isEnabled

: Negative words, isDisabled

: Positive words with "not", "no", or "non", isNotEnabled

: Negative words with "not", "no", or "non", isNotDisabled

# Use positive affirmation

: Positive words, isEnabled

: Negative words, isDisabled

: Positive words with "not", "no", or "non", isNotEnabled

: Negative words with "not", "no", or "non", isNotDisabled

We can rename isNotDisabled → isEnabled,  
and isNotEnabled → isDisabled without any logic change

# Topics

- Use the correct grammar
- Describe what rather than who/when
- Choose unambiguous words
- Avoid confusing abbreviations
- Add type/unit suffix
- Use positive affirmation

# Topics

- Use the correct grammar
- Describe what rather than who/when
- Choose unambiguous words
- Avoid confusing abbreviations
- Add type/unit suffix
- Use positive affirmation
- Adhere to language/platform/project conventions

# Language/platform/project conventions

Adhere to convention/rule/manner of language/platform/project

# Language/platform/project conventions

Adhere to convention/rule/manner of language/platform/project

Some conventions use a get... method, satisfying the following requirements

- Returns a value
- Finishes immediately and O(1)
- Has no side-effect

# Summary

A name must be accurate, descriptive, and unambiguous

# Summary

A name must be accurate, descriptive, and unambiguous

Pay attention to

- describing "what" rather than "who/when"
- grammar and word choice

/\*\* Code readability session 3 \*/

# // Comments

# Contents of this lecture

- Introduction and Principles
- Natural language: Naming, Comments
- Inner type structure: State, Procedure
- Inter type structure: Dependency (two sessions)
- Follow-up: Review

# Why we should write comments

## Readability

- Convey intent: summarize, explain reasoning
- Prevent mistakes: note precautions
- Refactor: rename, simplify

# Why we should write comments

## Readability

- Convey intent: summarize, explain reasoning
- Prevent mistakes: note precautions
- Refactor: rename, simplify

We don't need a comment if the code is simple enough  
(depending on the convention)

# Why we should write comments

## Readability

- Convey intent: summarize, explain reasoning
- Prevent mistakes: note precautions
- Refactor: rename, simplify

We don't need a comment if the code is simple enough  
(depending on the convention)

# Refactoring with comments: Example

```
/**  
 * Adds a new pair of keyword and the definition to this dictionary,  
 * can be referenced by [getDefinition(String)].  
 *  
 * If the keyword is already registered, this registration fails.  
 * Then, this returns a boolean representing whether the registration  
 * succeeded.  
 */  
fun add(newData: Pair<String, String>): Boolean
```

# Refactoring with comments: Refactoring plan

Think about why we need a long comment

# Refactoring with comments: Refactoring plan

Think about why we need a long comment

for the parameter, error case, and return value

- Rename method
- Break parameters
- Simplify error cases
- Remove unnecessary return value

# Refactoring after comment: Refactoring result

```
/**  
 * Adds or overwrites a definition of a given [keyword].  
 * The registered definition is obtained by [getDefinition(String)].  
 */  
fun registerDefinition(keyword: String, definitionText: String)
```

# Types of comments

- Documentation: Formal comment for type/value/procedure...
- Inline comment: Informal comment in a code block
- To do: // TODO: , // FIXME:
- IDE/compiler support, code generation: // \$COVERAGE-IGNORE\$

# Topics

- Documentation
- Inline comment

# Topics

- Documentation
- Inline comment

# What we write documentation for

- **Type:** class, struct, interface, protocol, enum ...
- **Value:** field, property, constant, parameter ...
- **Procedure:** global function, method, extension ...
- **Scope:** package, module, namespace ...

# Format of documentation

Starts with...

- `/\*\*` for Kotlin, Java, Swift and Objective-C
- `///` for Swift and Objective-C
- `/\*!` for Objective-C

Refer to documentation for [KDoc](#), [JavaDoc](#), [Swift Markup](#) or [HeaderDoc](#) for more details

# Contents of documentation

Let's take a look at anti-patterns first

# Anti-patterns: Auto-generated comment

```
/**  
 * @param keyword  
 * @return  
 */  
fun getDescription(keyword: String): String
```

# Anti-patterns: Same to the name

```
/**  
 * Gets the description for a keyword.  
 */  
fun getDescription(keyword: String): String
```

# Anti-patterns: Translation of the code

```
/**  
 * Calls [doA] if `conditionA` is satisfied.  
 * Otherwise, calls [doB] and if ...  
 */  
fun getDescription(keyword: String): String {  
    if (conditionA) {  
        doA()  
    } else {  
        doB()  
        if (...) {...}  
    }  
}
```

# Anti-patterns: Referring to private members

```
/**  
 * Returns a string stored in a private map [dictionary].  
 */  
fun getDescription(keyword: String): String
```

# Anti-patterns: No summary

```
/**  
 * Throws an exception if the given `keyword` is empty.  
 */  
fun getDescription(keyword: String): String
```

# Anti-patterns: Mentioning to callers

```
/**  
 * ...  
 * This is called by class [UserProfilePresenter].  
 */  
fun getDescription(keyword: String): String
```

# Lessons from anti-patterns

Write documentation only when explanation is required

- Fine to skip if type/value/procedure name is enough

# Lessons from anti-patterns

Write documentation only when explanation is required

- Fine to skip if type/value/procedure name is enough

Summarize "what it is/does"

- Without mentioning to callers
- Without using implementation details

# Contents of documentation

# Contents of documentation

**Short summary (Mandatory)**

"What it is/does" in a phrase/sentence

# Contents of documentation

## Short summary (Mandatory)

"What it is/does" in a phrase/sentence

## Details (Optional)

Additional sentences to explain usages, limitations, and so on

# Contents of documentation

## Short summary (Mandatory)

"What it is/does" in a phrase/sentence

## Details (Optional)

Additional sentences to explain usages, limitations, and so on

# Short summary: How to write 1/2

Find the most important line/block/element of code

# Short summary: How to write 1/2

Find the most important line/block/element of code

```
if (!user.isValid) {  
    return  
}  
  
val rawProfileImage = getProfileImage(user.id, ...)  
val roundProfileImage = applyRoundFilter(rawProfileImage, ...)  
profileView.setImage(roundProfileImage)
```

# Short summary: How to write 1/2

Find the most important line/block/element of code

```
if (!user.isValid) {  
    return  
}  
  
val rawProfileImage = getProfileImage(user.id, ...)  
val roundProfileImage = applyRoundFilter(rawProfileImage, ...)  
profileView.setImage(roundProfileImage)
```

# Short summary: How to write 2/2

Complement information based on the most important point

# Short summary: How to write 2/2

Complement information based on the most important point

```
/**  
 * Shows ... a profile image  
 *  
 */  
...  
profileView.setImage(roundProfileImage)
```

# Short summary: How to write 2/2

Complement information based on the most important point

```
/**  
 * Shows a roundly cut profile image of a given [user].  
 * ...  
 */  
...  
profileView.setImage(roundProfileImage)
```

# Short summary: How to write 2/2

Complement information based on the most important point

```
/**  
 * Shows a roundly cut profile image of a given [user].  
 * ...  
 */  
...  
profileView.setImage(roundProfileImage)
```

The summary will also be a good hint for naming 😊

# Convention of short summary

Refer to standard library documents Examples: [Kotlin](#), [Java](#), [Swift](#), [Objective-C](#)

- **Type, Value:** Starts with a **noun phrase**  
e.g., "A generic ordered collection of elements."<sup>4</sup>
- **Procedure:** Starts with a **verb** with 3rd-person singular form  
e.g., "Adds a new element at the end of the array."<sup>5</sup>

---

<sup>4</sup> <https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/#list>

<sup>5</sup> <https://developer.apple.com/documentation/swift/array/3126937-append>

# Contents of documentation

## Short summary (Mandatory)

"What it is/does" in a phrase/sentence

## Details (Optional)

Additional sentences to explain usages, limitations, and so on

# Contents of "details" in documentation

- Specification and usage
- Function return value
- Limitation and error values/status
- Examples
- ...

# Contents of "details" in documentation

- Specification and usage
- Function return value
- Limitation and error values/status
- Examples

# Specification and usage

Comment on typical usage and expected behavior

# Specification and usage

Comment on typical usage and expected behavior

```
/**  
 * ...  
 * To update view components such as message text and sender name,  
 * give [MessageData] model object to [bindView].  
 */  
class MessageViewPresenter(messageView: View)
```

# Contents of "details" in documentation

- Specification and usage
- Function return value
- Limitation and error values/status
- Examples

# Function return value 1/2

**Question:** What does the following return value mean?

```
fun setSelectedState(isSelected: Boolean): Boolean
```

# Function return value 1/2

**Question:** What does the following return value mean?

```
fun setSelectedState(isSelected: Boolean): Boolean
```

**Possible answer:**

```
isToggled // true if the state is changed
```

# Function return value 1/2

**Question:** What does the following return value mean?

```
fun setSelectedState(isSelected: Boolean): Boolean
```

**Possible answer:**

isToggled // true if the state is changed

wasSelected // previous state before the function call

# Function return value 1/2

Question: What does the following return value mean?

```
fun setSelectedState(isSelected: Boolean): Boolean
```

Possible answer:

isToggled // true if the state is changed

wasSelected // previous state before the function call

isSuccessfullyUpdated // true if there is no error

# Function return value 1/2

Question: What does the following return value mean?

```
fun setSelectedState(isSelected: Boolean): Boolean
```

Possible answer:

isToggled // true if the state is changed

wasSelected // previous state before the function call

isSuccessfullyUpdated // true if there is no error

isSelected // pass through the given `isSelected` value

...

# Function return value 2/2

Comment on a return value if the function name is not enough

- has **side effects** for function call
- has **contract** on the return value

# Function return value 2/2

Comment on a return value if the function name is not enough

- has **side effects** for function call
- has **contract** on the return value

```
/**  
 * ...  
 * ... returns true if it was selected before this function call.  
 */  
fun setSelectedState(isSelected: Boolean): Boolean
```

# Function return value 2/2

Comment on a return value if the function name is not enough

- has **side effects** for function call
- has **contract** on the return value

```
/**  
 * ...  
 * The profile ID is non-negative value.  
 */  
fun getProfileId(): Int
```

# Contents of "details" in documentation

- Specification and usage
- Function return value
- Limitation and error values/status
- Examples

# Limitation and error values/status

Comment on the **expected precondition** and **error values/status**

- A required receiver state to call a function
- Restrictions on the arguments

# Limitation and error values/status

Comment on the **expected precondition** and **error values/status**

- A required receiver state to call a function
- Restrictions on the arguments

```
/**  
 * ...  
 * [prepare] must be called before calling [play] or [seek],  
 * or this throws [ResourceNotReadyException].  
 */  
class VideoPlayer(videoPath: String)
```

# Limitation and error values/status

Comment on the expected precondition and error values/status

- A required receiver state to call a function
- Restrictions on the arguments

```
/**  
 * ...  
 * Returns `null` if the given `position` is out of the array range.  
 */  
fun valueAt(position: Int): T?
```

# Limitations: Example

- Arguments and receiver state
- Instance lifecycle
- Caller thread
- Reentrancy, idempotency ...
- Calculation cost, memory size
- External environment (e.g., network, database)

# Contents of "details" in documentation

- Specification and usage
- Function return value
- Limitation and error values/status
- Examples

# Examples

Make code more understandable with examples

- Example code for usage
- Parameter or return value example

```
/**  
 * ...  
 * For example, this returns `arrayOf("a", "bc", "", "d")`  
 * for argument `"a, bc , ,d"`  
 */  
fun splitByComma(string: String): Array<String> {...}
```

# Documentation: Summary

# Documentation: Summary

**Target:** Type, value, procedure, and scope

# Documentation: Summary

**Target:** Type, value, procedure, and scope

**Contents:** Short summary (mandatory), details (optional)

# Documentation: Summary

**Target:** Type, value, procedure, and scope

**Contents:** Short summary (mandatory), details (optional)

**Cautions:**

- Write only when the name is insufficient as explanation
- Don't mention to the caller or the implementation detail

# Topics

- Documentation
- Inline comment

# What inline comment is: Kotlin case

# What inline comment is: Kotlin case

```
// Function explanation, which won't appear on KDoc
fun function(param /* Parameter explanation */: Param) {

    // Code block summary
    someMethod(value) // Reason of a statement
    newId = param.id + 123 /* Reason of constant value */ + ...
    ...
}
```

# What inline comment is: Kotlin case

```
// Function explanation, which won't appear on KDoc
fun function(param /* Parameter explanation */: Param) {

    // Code block summary
    someMethod(value) // Reason of a statement
    newId = param.id + 123 /* Reason of constant value */ + ...
    ...
}
```

- Comment with whatever helps readers
- Summary is not mandatory

# What requires inline comments

- Large code: Code blocks with comments
- Complex/unintuitive code: Summary, reason
- Workaround: Background, issue link

# What requires inline comments

- Large code: Code blocks with comments
- Complex/unintuitive code: Summary, reason
- Workaround: Background, issue link

# Inline comment for large code

Make code blocks with comments to summarize what they do

# Inline comment for large code

Make code blocks with comments to summarize what they do

```
...
val messageKey = ...
val messageData = messageCache[messageKey]
...

if (messageData != null || ...) { // <- When this satisfies?
    ... // <- Hard to overview code in a nest
}
```

# Inline comment for large code

Make code blocks with comments to summarize what they do

```
// Get message data cache if it's available
val messageKey = ...
val messageData = messageCache[messageKey]
...

// Load message data from DB if there's no cached data.
if (messageData != null || ...) {
    ...
}
```

# What requires inline comments

- Large code: Code blocks with comments
- Complex/unintuitive code: Summary, reason
- Workaround: Background, issue link

# Inline comment for unintuitive code

Explain summary/reason if it's hard to understand

```
//  
//  
wordReplacementData.reverse()  
    .forEach { (startIndex, endIndex, newText) ->  
        stringBuilder.replace(startIndex, endIndex, newText)  
    }
```

# Inline comment for unintuitive code

Explain summary/reason if it's hard to understand

```
//  
//  
wordReplacementData.reverse()  
    .forEach { (startIndex, endIndex, newText) ->  
        stringBuilder.replace(startIndex, endIndex, newText)  
    }
```

**Question:** Why do we need to call reverse()?

# Inline comment for unintuitive code

Explain summary/reason if it's hard to understand

```
// Replace texts in the reverse order because `replace()`  
// affects the following indices.  
wordReplacementData.reverse()  
    .foreach { (startIndex, endIndex, newText) ->  
        stringBuilder.replace(startIndex, endIndex, newText)  
    }
```

# Inline comment for unintuitive code

Explain summary/reason if it's hard to understand

```
// Replace texts in the reverse order because `replace()`  
// affects the following indices.  
wordReplacementData.reverse()  
    .forEach { (startIndex, endIndex, newText) ->  
        stringBuilder.replace(startIndex, endIndex, newText)  
    }
```

Prevent from removing reverse() by incorrect "refactoring"

# What requires inline comments

- Large code: Code blocks with comments
- Complex/unintuitive code: Summary, reason
- Workaround: Background, issue link

# Inline comment for workaround

Explain what a workaround does and explain the reason

```
// We restore the previous state here  
// because libraryFunction() may break the receiver state
```

# Inline comment for workaround

Explain what a workaround does and explain the reason

```
// We restore the previous state here  
// because libraryFunction() may break the receiver state
```

Add links to explain

```
// To avoid Device-X specific tinting bug (see, ISSUE-123456)
```

# Inline comments: Summary

Essentially the same with documentation

- Short summary is not mandatory

# Inline comments: Summary

Essentially the same with documentation

- Short summary is not mandatory

Explain for large, complex, unintuitive code

- Code block, summary, reason, and links

# Summary

- Comment with whatever readers require
- Refactor before/after writing comments
- Documentation: Short summary (mandatory) and details (optional)
- Inline comments: Explanation for large, complex, unintuitive code

```
const val CODE_READABILITY_SESSION_4
```

```
var State
```

# Contents of this lecture

- Introduction and Principles
- Natural language: Naming, Comments
- Inner type structure: State, Procedure
- Inter type structure: Dependency (two sessions)
- Follow-up: Review

# Object state and readability 1/2

It's hard to read overly stated code

# Object state and readability 1/2

It's hard to read overly stated code

**Question:** Reference transparent/immutable == readable?

# Object state and readability 1/2

It's hard to read overly stated code

Question: Reference transparent/immutable == readable?

No, sometimes it can be easier to read code with side-effect or mutable state.

# Object state and readability 2/2

Let's look at an example of width first search of a binary tree

```
class Node(value: Int, left: Node?, right: Node?)
```

# Object state and readability 2/2

Let's look at an example of width first search of a binary tree

```
class Node(value: Int, left: Node?, right: Node?)
```

**Case1:** With a mutable queue

**Case2:** With recursive call

# Case1: With a mutable queue

```
fun search(valueToFind: Int, root: Node): Node? {  
    val queue = ArrayDeque<Node>()  
    var node: Node? = root  
    while (node != null && node.value != valueToFind) {  
        node.left?.let(queue::add)  
        node.right?.let(queue::add)  
        node = queue.poll()  
    }  
    return node  
}
```

# Case2: With recursive call

```
fun search(valueToFind: Int, root: Node): Node? =  
    innerSearch(valueToFind, listOf(root))  
  
tailrec fun innerSearch(valueToFind: Int, queue: List<Node>): Node? {  
    val node = queue.getOrNull(0)  
    if (node == null || node.value == valueToFind) {  
        return node  
    }  
  
    val nextQueue = queue.subList(1, queue.size) +  
        (node.left?.let(::listOf) ?: emptyList()) +  
        (node.right?.let(::listOf) ?: emptyList())  
    return innerSearch(valueToFind, nextQueue)  
}
```

# Problems of the recursive call example

# Problems of the recursive call example

## More complicated

- Required a separate public interface from recursive function
- Complex queue calculation for the next iteration

# Problems of the recursive call example

## More complicated

- Required a separate public interface from recursive function
- Complex queue calculation for the next iteration

## No guarantee that `valueToFind` is constant

- Recursive call arguments can be changed for each call
- A nested function is required to make it constant

# What we should care about

# What we should care about

Fold/recursive call is just a way to make code readable/robust

- Must not be an objective
- Apply fold/recursive call only when it makes code readable

# What we should care about

Fold/recursive call is just a way to make code readable/robust

- Must not be an objective
- Apply fold/recursive call only when it makes code readable

Focus on actual program state rather than apparent immutability

- Mutability within a small scope may be fine
- Fold/recursive call may adversely affect actual state

# Topics

How to simplify states for readability

# Topics

## How to simplify states for readability

- "Orthogonal" relationship
- State design strategies

# Topics

## How to simplify states for readability

- "Orthogonal" relationship
- State design strategies

# "Orthogonal" relationship: Definition

# "Orthogonal" relationship: Definition

For two properties, they are "orthogonal" if one is modifiable regardless of the other's value

# "Orthogonal" relationship: Example

Orthogonal: authorName and layoutVisibility

Both properties can be updated regardless of the other value

# "Orthogonal" relationship: Example

**Orthogonal:** authorName and layoutVisibility

Both properties can be updated regardless of the other value

**Non-orthogonal:** authorId and authorName

Both properties depend on each other

authorName must be updated when authorId is also updated

# Why we avoid non-orthogonal relationships

Inconsistent state might occur

# Why we avoid non-orthogonal relationships

Inconsistent state might occur

```
var coinCount: Int  
var coinText: String
```

# Why we avoid non-orthogonal relationships

Inconsistent state might occur

```
var coinCount: Int  
var coinText: String
```

What does `coinCount == 10 && coinText == "0 coin"` represent?

# How to remove non-orthogonal relationships

- Replace with a property getter or function
- Encapsulate by algebraic data type

# How to remove non-orthogonal relationships

- Replace with a property getter or function
- Encapsulate by algebraic data type

# Replace by using getter/function 1/2

## Non-orthogonal properties

```
var coinCount: Int  
var coinText: String
```

# Replace by using getter/function 1/2

## Non-orthogonal properties

```
var coinCount: Int  
var coinText: String
```

Remove variable by means of a property getter

# Replace by using getter/function 2/2

Create coinText value by coinCount

```
var coinCount: Int  
var coinText: String
```

# Replace by using getter/function 2/2

Create coinText value by coinCount

```
var coinCount: Int  
val coinText: String  
    get() = resource.getQuantityString(..., coinCount)
```

# Replace by using getter/function 2/2

Create coinText value by coinCount

```
var coinCount: Int  
val coinText: String  
    get() = resource.getQuantityString(..., coinCount)
```

The modifiable property is only coinCount

# How to remove non-orthogonal relationships

- Replace with a property getter or function
- Encapsulate by algebraic data type

# Encapsulate by algebraic data type 1/2

Example:

```
var queryResultText: String?  
var queryErrorCode: Int?
```

One of queryResultText or queryErrorCode is non null exclusively

# Encapsulate by algebraic data type 1/2

Example:

```
var queryResultText: String?  
var queryErrorCode: Int?
```

One of queryResultText or queryErrorCode is non null exclusively

A variable cannot be computable from the other

# Encapsulate by algebraic data type 1/2

Example:

```
var queryResultText: String?  
var queryErrorCode: Int?
```

One of queryResultText or queryErrorCode is non null exclusively

A variable cannot be computable from the other

Use algebraic data type

# Algebraic data type

Data type to represent **direct sum**

# Algebraic data type

Data type to represent **direct sum**

Example of binary tree:

```
Node = Branch(Node, Node) | Leaf(Int) | Nil
```

# Algebraic data type

Data type to represent **direct sum**

Example of binary tree:

```
Node = Branch(Node, Node) | Leaf(Int) | Nil
```

Realized by:

Tagged union, variant, sealed class, associated value ...

# Encapsulate by algebraic data type 2/2

```
sealed class QueryResponse {  
    class Result(val resultText: String): QueryResponse()  
    class Error(val errorCode: Int): QueryResponse()  
}
```

# Encapsulate by algebraic data type 2/2

```
sealed class QueryResponse {  
    class Result(val resultText: String): QueryResponse()  
    class Error(val errorCode: Int): QueryResponse()  
}
```

QueryResponse **has** resultText or errorCode **exclusively**

# How to remove non-orthogonal relationships

- Replace with a property getter or function
- Encapsulate by algebraic data type

# How to remove non-orthogonal relationships

- Replace with a property getter or function
- Encapsulate by algebraic data type
  - Emulate algebraic data type
  - Use an enum for a special case

# How to remove non-orthogonal relationships

- Replace with a property getter or function
- Encapsulate by algebraic data type
  - Emulate algebraic data type
  - Use an enum for a special case

# Emulate algebraic data type 1/2

Some language does not have algebraic data type (e.g., Java)

# Emulate algebraic data type 1/2

Some language does not have algebraic data type (e.g., Java)

Emulate algebraic data type with a small class

# Emulate algebraic data type 2/2

```
class QueryResponse {  
    @Nullable private final String resultText;  
    @Nullable private final Integer errorCode;
```

# Emulate algebraic data type 2/2

```
class QueryResponse {  
    @Nullable private final String resultText;  
    @Nullable private final Integer errorCode;  
  
    private QueryResponse(...) { ... }
```

# Emulate algebraic data type 2/2

```
class QueryResponse {  
    @Nullable private final String resultText;  
    @Nullable private final Integer errorCode;  
  
    private QueryResponse(...) { ... }  
  
    @NonNull  
    static QueryResponse asResult(@NonNull String ...) { ... }  
    @NonNull  
    static QueryResponse asError(int errorCode) { ... }
```

# How to remove non-orthogonal relationships

- Replace with a property getter or function
- Encapsulate by algebraic data type
  - Emulate algebraic data type
  - Use an enum for a special case

# Use an enum for a special case

An enum may be enough to remove a non-orthogonal relationship

# Use an enum for a special case

An enum may be enough to remove a non-orthogonal relationship

```
// isResultViewShown && isErrorViewShown can't happen
var isResultViewShown: Boolean
var isErrorViewShown: Boolean
```

# Use an enum for a special case

An enum may be enough to remove a non-orthogonal relationship

```
// isResultViewShown && isErrorViewShown can't happen  
var isResultViewShown: Boolean  
var isErrorViewShown: Boolean
```

An enum can remove (true, true) case

```
enum class VisibleViewType { RESULT_VIEW, ERROR_VIEW, NOTHING }
```

# "Orthogonal" relationship: Summary

# "Orthogonal" relationship: Summary

## Orthogonal relationship

- Updatable independently
- Good to remove invalid state

# "Orthogonal" relationship: Summary

## Orthogonal relationship

- Updatable independently
- Good to remove invalid state

To remove non-orthogonal relationships, use

- property getter/function
- algebraic data type, enum, small class

# Topics

## How to simplify states for readability

- "Orthogonal" relationship
- State design strategies

# Types of object state transitions

# Types of object state transitions

- **Immutable**: e.g., constant value
- **Idempotent**: e.g., closable, lazy
- **Acyclic (except for self)**: e.g., resource stream
- **Cyclic**: e.g., reusable

# Types of object state transitions

- Immutable: e.g., constant value
- Idempotent: e.g., closable, lazy
- Acyclic (except for self): e.g., resource stream
- Cyclic: e.g., reusable

# Immutable object

All properties won't be changed

# Immutable object

All properties won't be changed

Immutable type example:

```
class Immutable(val value: Int)  
class AnotherImmutable(val immutable: Immutable)
```

# Immutable object

All properties won't be changed

Immutable type example:

```
class Immutable(val value: Int)  
class AnotherImmutable(val immutable: Immutable)
```

Mutable type example:

```
class Mutable(var value: Int)  
class AnotherMutable(var immutable: Immutable)  
class YetAnotherMutable(val mutable: Mutable)
```

# Immutable properties

Make properties immutable if possible

# Immutable properties

Make properties immutable if possible

```
class ItemListCategoryData(  
    // Use val if no need to update  
    var itemCategoryName: String,
```

# Immutable properties

Make properties immutable if possible

```
class ItemListCategoryData(  
    // Use val if no need to update  
    var itemCategoryName: String,  
  
    // Even if we need to update,  
    // val of MutableList or var of List is enough  
    var itemModelList: MutableList<ItemModel>  
)
```

# Types of object state transitions

- **Immutable**: e.g., constant value
- **Idempotent**: e.g., closable, lazy
- **Acyclic (except for self)**: e.g., resource stream
- **Cyclic**: e.g., reusable

# Idempotency

The result of multiple operations is the same as single operation

# Idempotency

The result of multiple operations is the same as single operation

```
val closable = Closable() // "OPEN" state  
closable.close() // "CLOSED" state  
closable.close() // Valid. Keep "CLOSED" state
```

# Idempotency

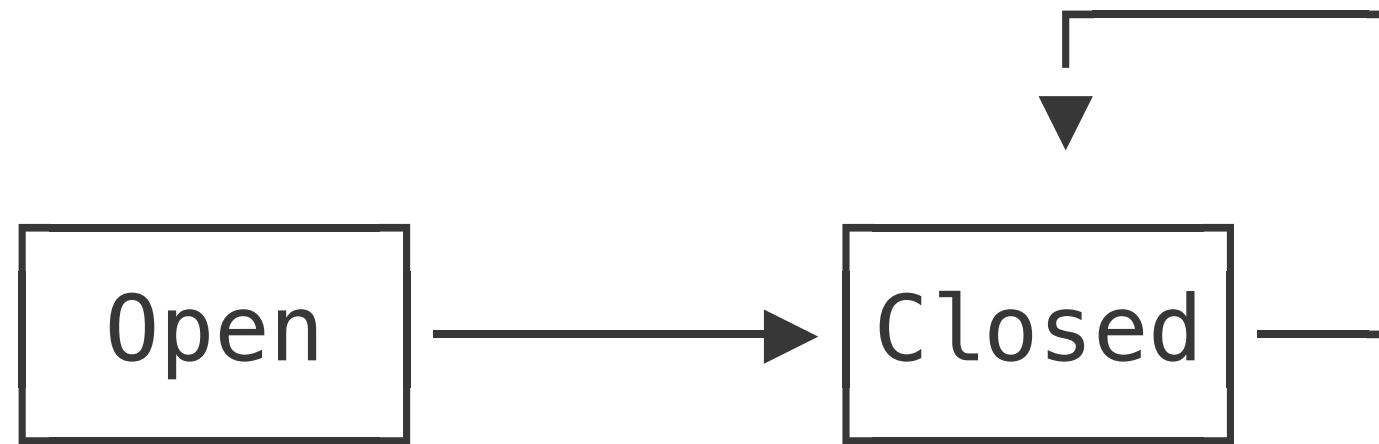
The result of multiple operations is the same as single operation

```
val closable = Closable() // "OPEN" state  
closable.close() // "CLOSED" state  
closable.close() // Valid. Keep "CLOSED" state
```

The side-effect may be hidden with a wrapper (e.g., Lazy)

# State graph of idempotent object

There are two states and one direction path



# Why idempotency is good

Encapsulate internal state to call a function

# Why idempotency is good

## Encapsulate internal state to call a function

```
// We may forget to check `isClosed`  
if (!nonIdempotentClosable.isClosed()) {  
    nonIdempotentClosable.close()  
}
```

# Why idempotency is good

## Encapsulate internal state to call a function

```
// We may forget to check `isClosed`  
if (!nonIdempotentClosable.isClosed()) {  
    nonIdempotentClosable.close()  
}  
  
// We can simply call `close` for an idempotent instance  
idempotentClosable.close()
```

# Non-idempotent case

It's NOT idempotent if the first state can remain after the operation

# Non-idempotent case

It's NOT idempotent if the first state can remain after the operation

```
fun get(): Result? {  
    if (latestResult == null) {  
        latestResult = queryToServer() // May return null  
    }  
    return latestResult
```

# Non-idempotent case

It's NOT idempotent if the first state can remain after the operation

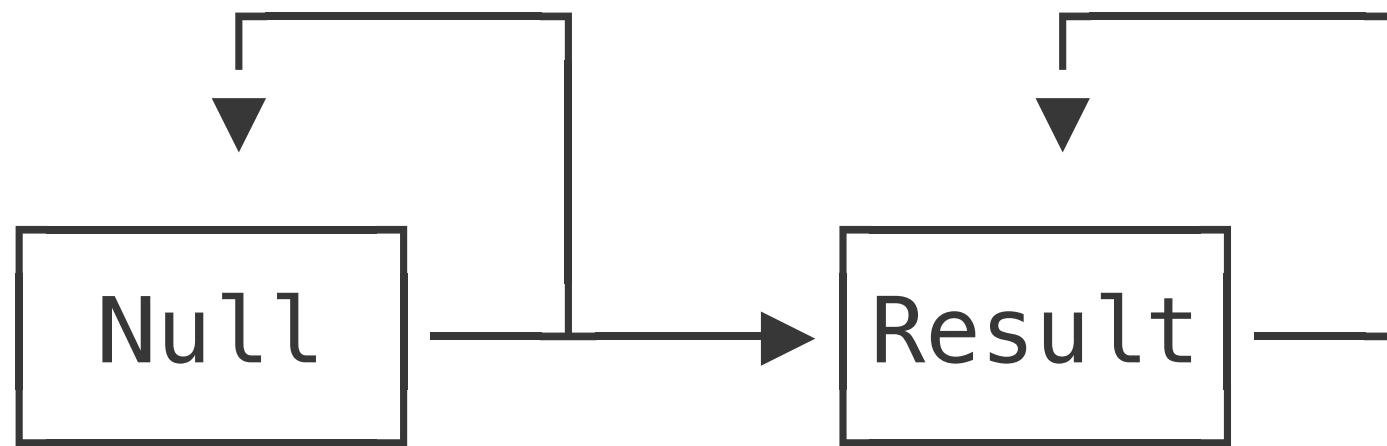
```
fun get(): Result? {  
    if (latestResult == null) {  
        latestResult = queryToServer() // May return null  
    }  
    return latestResult
```

The return values of the first call and the second can differ

```
get() != get() // This may be true!!
```

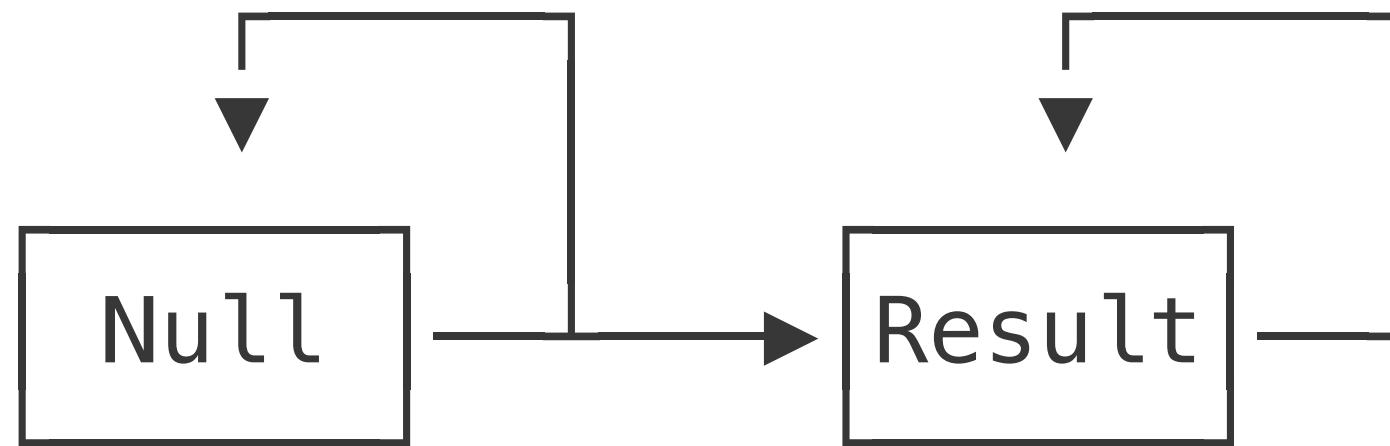
# Non-idempotent state transition graph

The first state also has a cycle to itself



# Non-idempotent state transition graph

The first state also has a cycle to itself



The side-effect should not be hidden

# Types of object state transitions

- Immutable: e.g., constant value
- Idempotent: e.g., closable, lazy
- Acyclic (except for self): e.g., resource stream
- Cyclic: e.g., reusable

# Acyclic state

Instance only used for a specific argument

- Another instance is created for another argument
- The performance is not good sometimes

# Acyclic state

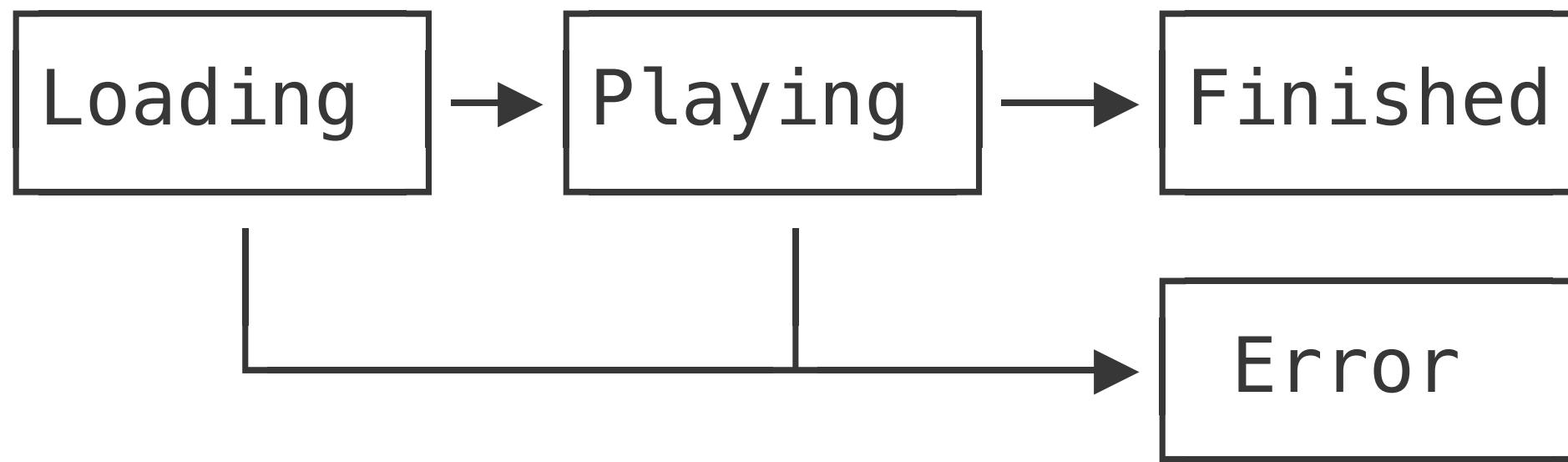
## Instance only used for a specific argument

- Another instance is created for another argument
- The performance is not good sometimes

```
// This class instance works only a given `videoPath`.  
class VideoPlayer(videoPath: String) {  
    enum class State { LOADING, PLAYING, FINISHED, ERROR }  
    ...  
}
```

# State graph of acyclic state

Partial order like state transition (if we ignore loop for itself)



# Why acyclic state is good 1/2

Let's compare with a type with cyclic states

# Why acyclic state is good 1/2

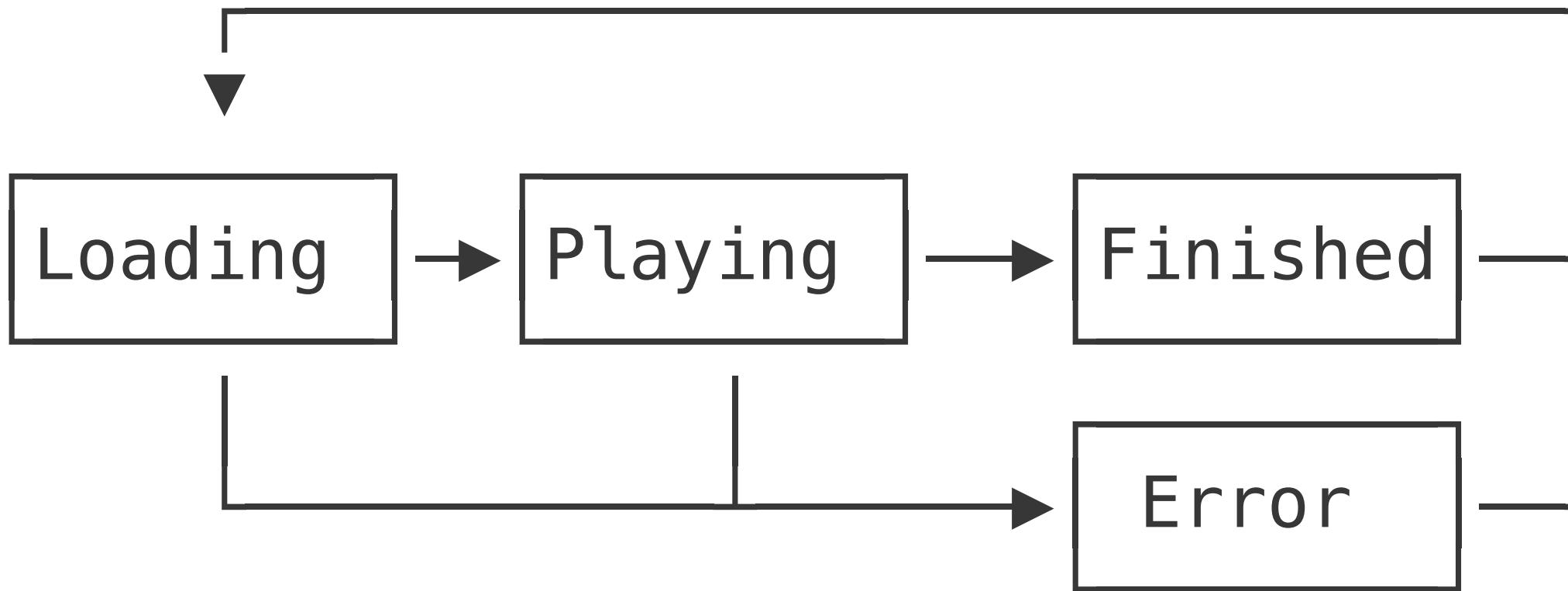
Let's compare with a type with cyclic states

Cyclic type example:

```
class VideoPlayer {  
    fun play(videoPath: String) { ... }
```

# Cyclic state transition graph

Loopback to "Loading" with a new video path



# Why acyclic state is good 2/2

Need to check the current loop argument if there's a cycle

# Why acyclic state is good 2/2

Need to check the current loop argument if there's a cycle

```
//  
// Abort playing video after 1000ms  
launch(...) {  
    delay(1000)  
  
    mediaPlayer.finish()
```

# Why acyclic state is good 2/2

Need to check the current loop argument if there's a cycle

```
val videoPath = videoPlayer.videoPath
// Abort playing video after 1000ms
launch(...) {
    delay(1000)

    // Need to check if a new video is not loaded
    if (videoPath == videoPlayer.videoPath) {
        videoPlayer.finish()
```

# Why acyclic state is good 2/2

Need to check the current loop argument if there's a cycle

```
val videoPath = videoPlayer.videoPath
// Abort playing video after 1000ms
launch(...) {
    delay(1000)

    // Need to check if a new video is not loaded
    if (videoPath == videoPlayer.videoPath) {
        videoPlayer.finish() // Not thread safe still
```

# Acyclic state VS reality

A cycle is often required to make a model simply

- May be overly complex to remove all cycles
- Example of VideoPlayer: PLAYING <-> PAUSED

# Acyclic state VS reality

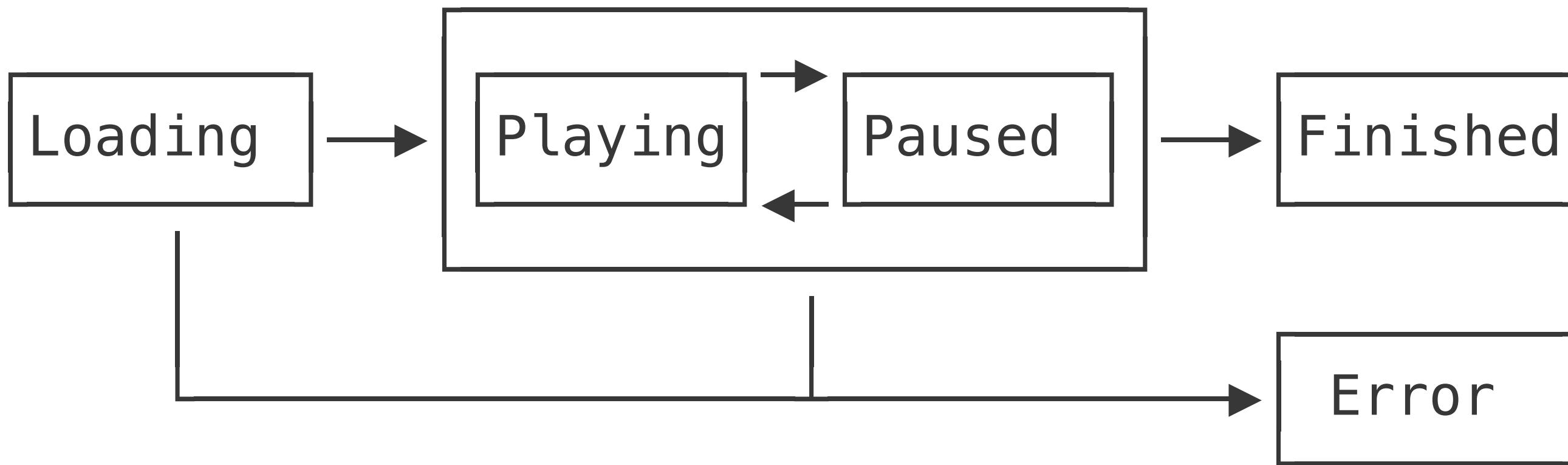
A cycle is often required to make a model simply

- May be overly complex to remove all cycles
- Example of VideoPlayer: PLAYING <-> PAUSED

Make the cycle as small as possible

# State cycle encapsulation

Put a cycle in an enclosing state = Don't create large cycle



# Types of object state transitions

- **Immutable**: e.g., constant value
- **Idempotent**: e.g., closable, lazy
- **Acyclic (except for self)**: e.g., resource stream
- **Cyclic**: e.g., reusable

# State design strategy: Summary

- Make properties immutable
- Idempotency is also good
- Remove/minimize state cycle

# Summary

Minimize state for readability and robustness

- Don't make it an objective

# Summary

Minimize state for readability and robustness

- Don't make it an objective

Remove/encapsulate non-orthogonal relationships

- Use a property getter/function, an algebraic data type, or enum

# Summary

Minimize state for readability and robustness

- Don't make it an objective

Remove/encapsulate non-orthogonal relationships

- Use a property getter/function, an algebraic data type, or enum

Care about state design strategy

- Immutability, idempotency and cycle

```
abstract fun codeReadabilitySession5()
```

```
fun Procedure()
```

# Contents of this lecture

- Introduction and Principles
- Natural language: Naming, Comments
- Inner type structure: State, Procedure
- Inter type structure: Dependency (two sessions)
- Follow-up: Review

# What "procedure" is

- main/subroutine
- function
- method
- computed property
- property getter/setter
- constructor/initialization block
- ...

# What a readable procedure is

# What a readable procedure is

## Predictable

- Consistent with the name
- Easy to write documentation
- Few error cases or limitations

# Topics

Make procedure responsibility/flow clear

# Topics

- Make responsibility clear
  - Split if it's hard to summarize
  - Split if it has both return value and side-effect
- Make flow clear
  - Perform definition-based programming
  - Focus on normal cases
  - Split by object, not condition

# Topics

- Make responsibility clear
  - Split if it's hard to summarize
  - Split if it has both return value and side-effect
- Make flow clear
  - Perform definition-based programming
  - Focus on normal cases
  - Split by object, not condition

# Split procedure if it's hard to summarize

Try to summarize what the procedure does

# Procedure summary: Example 1/2

```
messageView.text = messageData.contentText  
senderNameView.text = messageData.senderName  
timestampView.text = messageData.sentTimeText
```

# Procedure summary: Example 1/2

```
messageView.text = messageData.contentText  
senderNameView.text = messageData.senderName  
timestampView.text = messageData.sentTimeText
```

We can summarize this as:

"Bind/update message layout with a new message data"

# Procedure summary: Example 2/2

```
messageView.text = messageData.contentText  
doOnTransaction {  
    messageDatabase.insertNewMessage(messageData)  
}
```

# Procedure summary: Example 2/2

```
messageView.text = messageData.contentText  
doOnTransaction {  
    messageDatabase.insertNewMessage(messageData)  
}
```

How to summarize this

# Procedure summary: Example 2/2

```
messageView.text = messageData.contentText  
doOnTransaction {  
    messageDatabase.insertNewMessage(messageData)  
}
```

## How to summarize this

"Bind a new message **and save it**"

"Perform actions **on** a new message received"

# Procedure responsibility and summary

Hard to summarize = Typical bad signal

- Split the procedure into sub-procedures

# Procedure responsibility and summary

Hard to summarize = Typical bad signal

- Split the procedure into sub-procedures

```
fun bindMessageViewData(messageData: MessageData) { ...  
fun saveMessageDataToDatabase(messageData: MessageData) { ...
```

# Topics

- Make responsibility clear
  - Split if it's hard to summarize
  - Split if it has both return value and side-effect
- Make flow clear
  - Perform definition-based programming
  - Focus on normal cases
  - Split by object, not condition

# Command-query separation<sup>6</sup>

Asking a question should not change the answer<sup>7</sup>

---

<sup>6</sup> <https://martinfowler.com/bliki/CommandQuerySeparation.html>

<sup>7</sup> Eiffel: a language for software engineering, Meyer Bertrand, 2012

# Command-query separation<sup>6</sup>

Asking a question should not change the answer<sup>7</sup>

Split procedure by command and query

- **Command:** Procedure to modify receiver or parameters
- **Query:** Procedure to return without any modification

---

<sup>6</sup> <https://martinfowler.com/bliki/CommandQuerySeparation.html>

<sup>7</sup> Eiffel: a language for software engineering, Meyer Bertrand, 2012

# Command-query separation: Example 1/2

```
class IntList(vararg elements: Int) {  
    infix fun append(others: IntList): IntList = ...  
}
```

# Command-query separation: Example 1/2

```
class IntList(vararg elements: Int) {  
    infix fun append(others: IntList): IntList = ...  
}
```

```
val a = IntList(1, 2)  
val b = IntList(3, 4)  
val c = a append b
```

# Command-query separation: Example 1/2

```
class IntList(vararg elements: Int) {  
    infix fun append(others: IntList): IntList = ...  
}
```

```
val a = IntList(1, 2)  
val b = IntList(3, 4)  
val c = a append b
```

**Question:** What is the expected value of a, b, and c?

# Command-query separation: Example 2/2

```
val a = IntList(1, 2)
val b = IntList(3, 4)
val c = a append b
```

Expected: a={1, 2}, b={3, 4}, c={1, 2, 3, 4}

# Command-query separation: Example 2/2

```
val a = IntList(1, 2)
val b = IntList(3, 4)
val c = a append b
```

Expected: a={1, 2}, b={3, 4}, c={1, 2, 3, 4}

Result with bad code: a={1, 2, 3, 4}, b={3, 4}, c={1, 2, 3, 4}

# Command-query separation: Example 2/2

```
val a = IntList(1, 2)  
val b = IntList(3, 4)  
val c = a append b
```

Expected: a={1, 2}, b={3, 4}, c={1, 2, 3, 4}

Result with bad code: a={1, 2, 3, 4}, b={3, 4}, c={1, 2, 3, 4}

Function **append** must not modify a or b if it returns the result

# Command-query separation: Drawbacks 1/3

Command-query separation is not an objective

- May cause strong coupling (will explain at the next session)
- May make an unnecessary state

# Command-query separation: Drawbacks 2/3

Bad code example:

```
class UserDataStore {  
  
    // Command  
    fun saveUserData(userData: UserData) = ...  
  
}
```

# Command-query separation: Drawbacks 2/3

Bad code example:

```
class UserDataStore {  
    private var latestOperationResult: Result = NO_OPERATION  
    // Command  
    fun saveUserData(userData: UserData) = ...  
    // Query  
    val wasLatestOperationSuccessful: Boolean get() = ...  
}
```

# Command-query separation: Drawbacks 2/3

Bad code example:

```
class UserDataStore {  
    private var latestOperationResult: Result = NO_OPERATION  
    // Command  
    fun saveUserData(userData: UserData) = ...  
    // Query  
    val wasLatestOperationSuccessful: Boolean get() = ...  
}
```

State `latestOperationResult` may become a cause of a bug

# Command-query separation: Drawbacks 2/3

Good code example:

```
class UserDataRequester {  
    /**  
     * Saves a given user data to ...  
     * Then, returns true if the operation successful, otherwise false.  
     */  
    fun saveUserData(userData: UserData): Boolean = ...  
}
```

# Command-query separation: Drawbacks 2/3

Good code example:

```
class UserDataRequester {  
    /**  
     * Saves a given user data to ...  
     * Then, returns true if the operation successful, otherwise false.  
     */  
    fun saveUserData(userData: UserData): Boolean = ...  
}
```

No need to keep the latest result when returning it directly

# Responsibility on return value and side-effect

Don't modify when returning a "main" result

# Responsibility on return value and side-effect

Don't modify when returning a "main" result

= Acceptable to return a "sub" result with modification

# Responsibility on return value and side-effect

Don't modify when returning a "main" result

= Acceptable to return a "sub" result with modification

- Main result: conversion/calculation result, a new instance ...
- Sub result: error type, metadata of modification (stored size) ...  
(Documentation is mandatory)

# Responsibility on return value and side-effect

Don't modify when returning a "main" result

= Acceptable to return a "sub" result with modification

- Main result: conversion/calculation result, a new instance ...
- Sub result: error type, metadata of modification (stored size) ...  
(Documentation is mandatory)

Acceptable if the interface is a de facto standard

e.g., fun Queue<T>.poll(): T

# Make responsibility clear: Summary

# Make responsibility clear: Summary

Split a procedure if it's hard to summarize

- Try to write documentation anyway

# Make responsibility clear: Summary

Split a procedure if it's hard to summarize

- Try to write documentation anyway

Don't modify when returning a "main" result

- Fine to return "sub" result with modification
- Fine if the interface is a de facto standard

# Topics

- Make responsibility clear
  - Split if it's hard to summarize
  - Split if it has both return value and side-effect
- Make flow clear
  - Perform definition-based programming
  - Focus on normal cases
  - Split by object, not condition

# How to find if procedure flow is clear

# How to find if procedure flow is clear

Try to write short summary

# How to find if procedure flow is clear

Try to write short summary

- Hard to write when the flow is unclear

# Topics

- Make responsibility clear
  - Split if it's hard to summarize
  - Split if it has both return value and side-effect
- Make flow clear
  - Perform definition-based programming
  - Focus on normal cases
  - Split by object, not condition

# Definition-based programming

Prefer to define value/procedure with a name

# Definition-based programming

Prefer to define value/procedure with a name rather than:

- Nest (parameter, procedure, type, control flow)
- Anonymous procedure/object literal
- Chained call

# Definition-based programming: Bad example

# Definition-based programming: Bad example

```
fun startColorChangeAnimation(startColorInt: Int, endColorInt: Int) =  
    ColorAnimator(srcColorInt, destinationColorInt)  
        .also {  
            it.addUpdateListener {  
                if (it.colorValue == null) {  
                    return@addUpdateListener  
                }  
                // Apply the new color to views  
            }  
        }.start()
```

# What is wrong with the bad example

Hard to make summary of the procedure

# What is wrong with the bad example

Hard to make summary of the procedure

- Context and scope because of nested lambdas
- Which code is called directly, and which asynchronously
- Receiver caused by overly long call chain

# Definition-based programming: How to fix 1/3

Extract a lambda/parameter/receiver/call-chain  
as a named local value or a named private function

# Definition-based programming: How to fix 2/3

```
fun startColorChangeAnimation(startColorInt: Int, endColorInt: Int) =  
    ColorAnimator(srcColorInt, destinationColorInt)  
        .also {  
            it.addUpdateListener {  
                if (it.colorValue == null) {  
                    return@addUpdateListener  
                }  
                // Apply the new color to views  
            }  
        }.start()
```

# Definition-based programming: How to fix 2/3

```
fun startColorChangeAnimation(startColorInt: Int, endColorInt: Int) =  
    ColorAnimator(srcColorInt, destinationColorInt)  
        .also {  
            it.addUpdateListener {  
                if (it.colorValue == null) {  
                    return@addUpdateListener  
                }  
                // Apply the new color to views  
            }  
        }.start()
```

# Definition-based programming: How to fix 2/3

```
fun startColorChangeAnimation(startColorInt: Int, endColorInt: Int) =  
    ColorAnimator(srcColorInt, destinationColorInt)  
        .also {  
            it.addUpdateListener { applyColorToViews(it.colorValue) }  
        }.start()  
  
fun applyColorToViews(colorInt: Int?) {  
    if (colorInt == null) {  
        return  
    }  
    // Apply the new color to views  
}
```

# Definition-based programming: How to fix 3/3

```
fun startColorChangeAnimation(startColorInt: Int, endColorInt: Int) =  
    ColorAnimator(srcColorInt, destinationColorInt)  
        .also {  
            it.addUpdateListener { applyColorToViews(it.colorValue) }  
        }.start()  
  
fun applyColorToViews(colorInt: Int?) {  
    if (colorInt == null) {  
        return  
    }  
    // Apply the new color to views  
}
```

# Definition-based programming: How to fix 3/3

```
fun startColorChangeAnimation(startColorInt: Int, endColorInt: Int) {  
    val animator = ColorAnimator(srcColorInt, destinationColorInt)  
    animator.addUpdateListener { applyColorToViews(it.colorValue) }  
    animator.start()  
}  
  
fun applyColorToViews(colorInt: Int?) {  
    if (colorInt == null) {  
        return  
    }  
    // Apply the new color to views  
}
```

# Definition-based programming: How to fix 3/3

```
fun startColorChangeAnimation(startColorInt: Int, endColorInt: Int) {  
    val animator = ColorAnimator(srcColorInt, destinationColorInt)  
    animator.addUpdateListener { applyColorToViews(it.colorValue) }  
    animator.start()  
}  
  
fun applyColorToViews(colorInt: Int?) {  
    if (colorInt == null) {  
        return  
    }  
    // Apply the new color to views  
}
```

# Definition-based programming: Pitfall

Extracting may make the code less readable

# Definition-based programming: Pitfall

Extracting may make the code less readable

- Unnecessary state
- Unnecessary strong coupling (will explain at the next session)

# Pitfalls of extraction: Original code

```
val userNameTextView: View = ...  
val profileImageView: View = ...  
  
init {  
    // Complex userNameTextView initialization code  
    userNameTextView...  
  
    // Complex profileImageView initialization code  
    profileImageView...  
}
```

## Extract complex initialization code

# Pitfalls of extraction: Bad example

```
var userNameTextView: View? = null
var profileImageView: View? = null
init {
    initializeUserNameTextView()
    initializeProfileImageView()
}

private fun initializeUserNameTextView() {
    userNameTextView = ...
}

private fun initializeProfileImageView() {
    profileImageView = ...
}
```

# Pitfalls of extraction: What is wrong

- Unnecessary mutability and nullability
- Unsafe to call `initialize...` methods multiple times
- Function name `initialize...` is ambiguous

# Pitfalls of extraction: What is wrong

- Unnecessary mutability and nullability
- Unsafe to call initialize... methods multiple times
- Function name initialize... is ambiguous

Optimize the scope of the extracted methods

# Pitfalls of extraction: Good example

Extract view creation and initialization procedure

# Pitfalls of extraction: Good example

## Extract view creation and initialization procedure

```
val userNameTextView: View = createUserTextView()  
val profileImageView: View = createProfileImageView()
```

```
private fun createUserTextView(): TextView =  
    ...
```

```
private fun createProfileImageView(): ImageView =  
    ...
```

# Topics

- Make responsibility clear
  - Split if it's hard to summarize
  - Split if it has both return value and side-effect
- Make flow clear
  - Perform definition-based programming
  - Focus on normal cases
  - Split by object, not condition

# Normal cases and error cases

**Normal case:** Achieves the main purpose of the procedure

**Error case:** Quits the procedure without achieving the purpose

# Normal cases and error cases: Example

`String.toIntOrNull()`

# Normal cases and error cases: Example

`String.toIntOrNull()`

**Normal case:** Returns an integer value

e.g, "-1234", "0", "+001", "-2147483648"

# Normal cases and error cases: Example

`String.toIntOrNull()`

**Normal case:** Returns an integer value

e.g., "-1234", "0", "+001", "-2147483648"

**Error case:** Returns null

e.g., "--0", "text", "", "2147483648", "1.0"

# Procedure flow and normal/error cases

# Procedure flow and normal/error cases

Filter error cases to focus on the normal case

# Procedure flow and normal/error cases

Filter error cases to focus on the normal case

Apply early return

# Early return: Bad example

# Early return: Bad example

```
if (isNetworkAvailable()) {  
    val queryResult = queryToServer()  
    if (queryResult.isValid) {  
        // Do something with queryResult...  
    } else {  
        showInvalidResponseDialog()  
    }  
} else {  
    showNetworkUnavailableDialog()  
}
```

# Early return: What is wrong

Unclear

# Early return: What is wrong

## Unclear

- What the main case logic is
- The relation between error condition and handling logic

# Early return: Good example

# Early return: Good example

```
if (!isNetworkAvailable()) {  
    showNetworkUnavailableDialog()  
    return  
}  
  
val queryResult = queryToServer()  
if (!queryResult.isValid) {  
    showInvalidResponseDialog()  
    return  
}  
  
// Do something with queryResult...
```

# Benefits of early return

## Easy to find

- The main flow and purpose of a procedure
- Relation between error condition and handling logic

# Topics

- Make responsibility clear
  - Split if it's hard to summarize
  - Split if it has both return value and side-effect
- Make flow clear
  - Perform definition-based programming
  - Focus on normal cases
  - Split by object, not condition

# Two axes to break down procedure

Multiple conditions and target objects

Example:

- **Conditions:** Two account types (premium, free)
- **Objects:** Two views (background, icon)

# Two axes to break down procedure

Multiple conditions and target objects

Object	Condition	Premium account	Free account
Background color	Yellow		Gray
Account Icon	[Premium]		[Free]

# Two axes to break down procedure

Multiple conditions and target objects

Object	Condition	Premium account	Free account
Background color	Yellow		Gray
Account Icon	[Premium]		[Free]

Two ways to implement this matrix

# Bad example of "split by condition first"

```
fun bind ViewData(accountType: AccountType) {  
    when (accountType) {  
        PREMIUM -> updateViewsForPremium()  
        FREE -> updateViewsForFree()  
    }  
}
```

# Bad example of "split by condition first"

```
fun bind ViewData(accountType: AccountType) {  
    when (accountType) {  
        PREMIUM -> updateViewsForPremium()  
        FREE -> updateViewsForFree()  
    }  
}  
  
private fun updateViewsForPremium() {  
    backgroundView.color = PREMIUM_BACKGROUND_COLOR  
    accountTypeIcon.image = resources.getImage(PREMIUM_IMAGE_ID)  
}
```

# Split by object: What is wrong

- Can't summarize what the function does

```
/**  
 * Updates views depending whether the account type is free or premium.  
 */
```

# Split by object: What is wrong

- Can't summarize what the function does

```
/**  
 * Updates views depending whether the account type is free or premium.  
 */
```

- May cause bug because of no guarantee of completeness

# Split by object: What is wrong

- Can't summarize what the function does

```
/**  
 * Updates views depending whether the account type is free or premium.  
 */
```

- May cause bug because of no guarantee of completeness

Make supporting function or code block for each target object first

# Split by object: Good example 1/2

```
fun bind ViewData(accountType: AccountType) {  
    updateBackgroundColor(accountType)  
    updateAccountTypeImage(accountType)  
}
```

# Split by object: Good example 1/2

```
fun bind ViewData(accountType: AccountType) {  
    updateBackgroundViewColor(accountType)  
    updateAccountTypeImage(accountType)  
}  
  
private fun updateBackgroundViewColor(accountType: AccountType) {  
    backgroundView.color = when(accountType) {  
        PREMIUM -> PREMIUM_BACKGROUND_COLOR  
        FREE -> FREE_BACKGROUND_COLOR  
    }  
}
```

# Split by object: Good example

- Easy to write a short summary

```
/**  
 * Updates background color and icon according to a given account type.  
 */
```

# Split by object: Good example

- Easy to write a short summary

```
/**  
 * Updates background color and icon according to a given account type.  
 */
```

- Ensures that all the combinations are covered

# Split by object: Good example

- Easy to write a short summary

```
/**  
 * Updates background color and icon according to a given account type.  
 */
```

- Ensures that all the combinations are covered
- Can conduct further more refactoring
  - e.g., Make extracted functions reference transparent

# Split by object: More improvement

```
fun bind ViewData(accountType: AccountType) {  
    backgroundView.color = getBackgroundColorInt(accountType)  
    accountTypeIcon.image = getAccountTypeIconImage(accountType)  
}  
  
private fun getBackgroundColorInt(accountType: AccountType): Int =  
    when(accountType) {  
        PREMIUM -> PREMIUM_BACKGROUND_COLOR  
        FREE -> FREE_BACKGROUND_COLOR  
    }
```

# Early return VS split by object

Doesn't "early return" represent "split by condition"?

# Early return VS split by object

Doesn't "early return" represent "split by condition"?

Strategy 1:

- Handle an error case as if it's a normal case

# Early return VS split by object

Doesn't "early return" represent "split by condition"?

Strategy 1:

- Handle an error case as if it's a normal case

Strategy 2:

- Apply early return to error cases
- Apply "split by object"

# Procedure flow: Summary

# Procedure flow: Summary

Perform definition-based programming:

Extract nests/chains to a named value/function

# Procedure flow: Summary

Perform definition-based programming:

Extract nests/chains to a named value/function

Focus on normal cases:

Apply early return

# Procedure flow: Summary

Perform definition-based programming:

Extract nests/chains to a named value/function

Focus on normal cases:

Apply early return

Split by object, not condition:

Make a function or block for each target object

# Summary

Check whether it's easy to write a short summary

# Summary

Check whether it's easy to write a short summary

Procedure responsibility:

- Split if required
- Don't modify when returning a main result

# Summary

Check whether it's easy to write a short summary

Procedure responsibility:

- Split if required
- Don't modify when returning a main result

Procedure flow:

- Apply definition based programing, early return, split by object

Code → readability → session 6

Dependency ↑

# Contents of this lecture

- Introduction and Principles
- Natural language: Naming, Comments
- Inner type structure: State, Procedure
- Inter type structure: Dependency (two sessions)
- Follow-up: Review

# What "dependency" is

**Example:** Type "X" depends on type "Y"

- Type X has an instance of Y as a property
- A method of X takes Y as a parameter or returns Y
- X calls a method of Y
- X inherits Y

# What "dependency" is

**Example:** Type "X" depends on type "Y"

- Type X has an instance of Y as a property
- A method of X takes Y as a parameter or returns Y
- X calls a method of Y
- X inherits Y

"X" depends on "Y" if word "Y" appears in "X", roughly speaking

# What "dependency" is

Is the word "dependency" only for class?

# What "dependency" is

Is the word "dependency" only for class?

No

- Procedure, module, instance ...

# Topics

## First session:

- Coupling

## Second session:

- Direction
- Redundancy
- Explicitness

# Topics

First session:

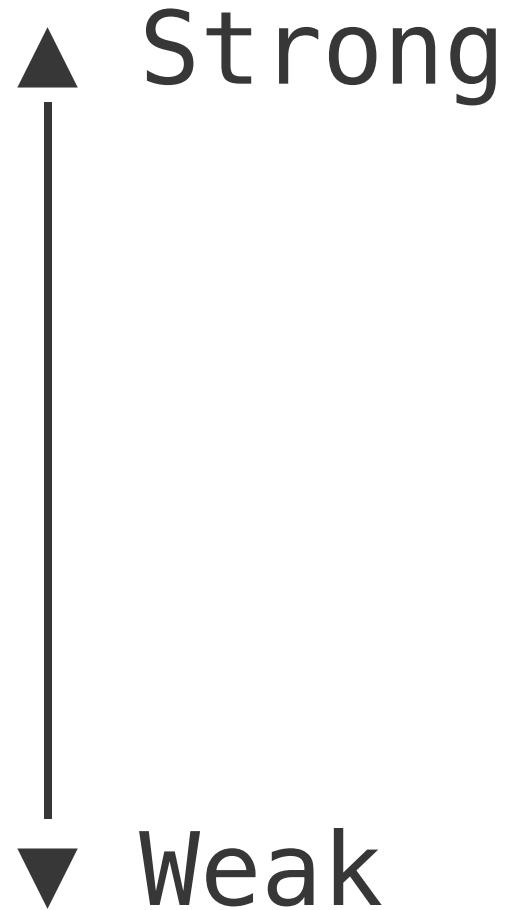
- Coupling

Second session:

- Direction
- Redundancy
- Explicitness

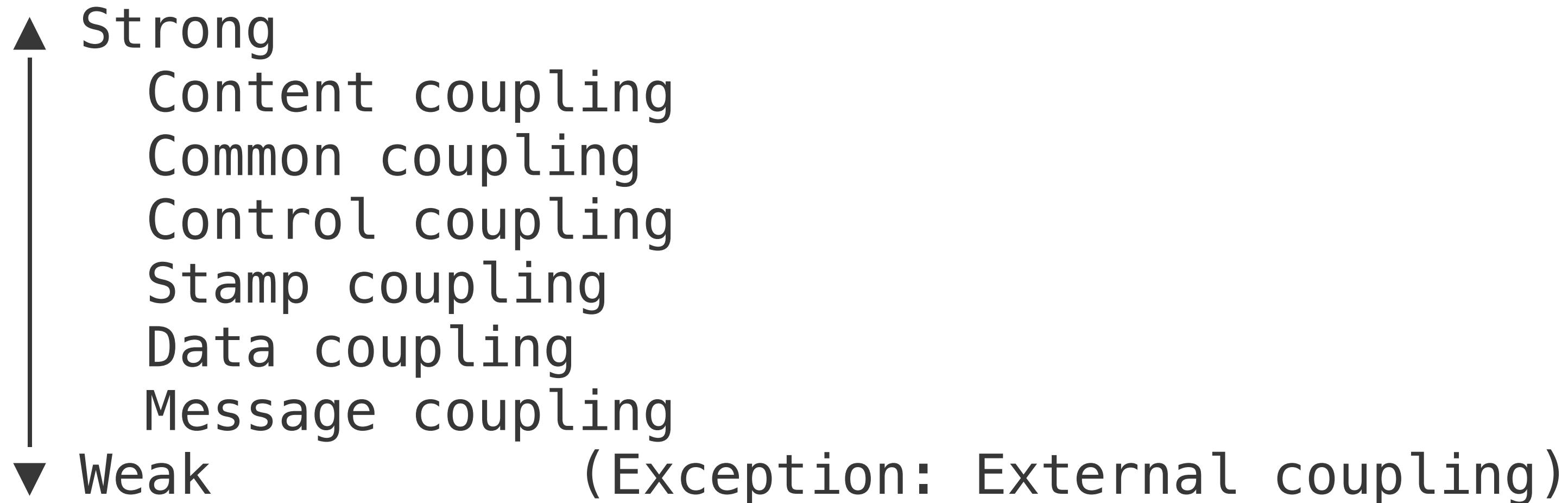
# Coupling

A property to represent how strong the dependency is



# Coupling

A property to represent how strong the dependency is



# Coupling

Bad/strong coupling: content, common, control

Fine/weak coupling: stamp, data, message

# Coupling

Bad/strong coupling: **content**, common, control

Fine/weak coupling: stamp, data, message

# Content coupling

Relies on the internal workings

# Content coupling

Relies on the internal workings

Example:

- Allows illegal usage
- Shares internal mutable state

# Content coupling

Relies on the internal workings

Example:

- Allows illegal usage
- Shares internal mutable state

# Allows illegal usage: Bad example

```
class Caller {  
    fun callCalculator() {  
  
        calculator.calculate()  
  
        ...  
    }  
}
```

# Allows illegal usage: Bad example

```
class Caller {  
    fun callCalculator() {  
        calculator.parameter = 42  
  
        calculator.calculate()  
  
        val result = calculator.result  
  
        ...  
    }  
}
```

# Allows illegal usage: Bad example

```
class Caller {  
    fun callCalculator() {  
        calculator.parameter = 42  
  
        calculator.prepare()  
        calculator.calculate()  
        calculator.tearDown()  
  
        val result = calculator.result  
        ...  
    }  
}
```

# Allows illegal usage: Bad points

# Allows illegal usage: Bad points

calculator has the following two requirements:

- Call prepare()/tearDown() before/after calling calculate()
- Pass/get a value by parameter/result property.

# Allows illegal usage: Bad points

calculator has the following two requirements:

- Call prepare()/tearDown() before/after calling calculate()
- Pass/get a value by parameter/result property.

Not robust to wrong usage and implementation change

# Allows illegal usage: How to fix

- Encapsulate stateful call sequence
- Pass/receive data as arguments or a return value

# Allows illegal usage: Fixed code example

```
class Caller {  
    fun callCalculator() {  
        val result = calculator.calculate(42)  
        ...  
    }  
}
```

# Allows illegal usage: Fixed code example

```
class Caller {  
    fun callCalculator() {  
        val result = calculator.calculate(42)  
        ...  
    }  
}
```

```
class Calculator {  
    fun calculate(type: Int): Int {  
        prepare()  
        ...  
        tearDown()  
        return ...  
    }  
}
```

# Allows illegal usage: What to check

# Allows illegal usage: What to check

Don't apply design concept/principal too much

- e.g., command/query separation

# Content coupling

Relies on the internal workings

Example:

- Allows illegal usage
- Shares internal mutable state

# Shares mutable state: Bad example

```
class UserListPresenter(val userList: List<UserData>) {  
    fun refreshViews() { /* ... */ }
```

# Shares mutable state: Bad example

```
class UserListPresenter(val userList: List<UserData>) {  
    fun refreshViews() { /* ... */ }  
  
class Caller {  
    val userList: List<UserData> = mutableListOf()  
    val presenter = UserListPresenter(userList)  
  
    ...  
}
```

# Shares mutable state: Bad example

```
class UserListPresenter(val userList: List<UserData>) {  
    fun refreshViews() { /* ... */ }  
  
class Caller {  
    val userList: List<UserData> = mutableListOf()  
    val presenter = UserListPresenter(userList)  
  
    ...  
    fun addUser(newUser: UserData) {  
        userList += newUser  
        presenter.refreshViews()  
    }  
}
```

# Shares mutable state: Bad points

Behavior of UserListPresenter depends on external mutable value

- Causes unexpected state update by other reference holders
- No limitation of modifying the list

# Shares mutable state: Bad points

Behavior of UserListPresenter depends on external mutable value

- Causes unexpected state update by other reference holders
- No limitation of modifying the list

Easy to break the state and hard to find the root cause

- A reference to the presenter is not required to update it

# Shares mutable state: How to fix

- Make values immutable, copy arguments, or apply copy-on-write
- Limit interfaces to update the state

# Shares mutable state: Fixed code example

```
class UserListPresenter {  
    val userList: MutableList<UserData> = mutableListOf()
```

# Shares mutable state: Fixed code example

```
class UserListPresenter {  
    val userList: MutableList<UserData> = mutableListOf()  
  
    fun addUsers(newUsers: List<UserData>) {  
        userList += newUsers  
        // Update views with `userList`  
    }  
}
```

# Shares mutable state: Fixed code example

```
class UserListPresenter {  
    val userList: MutableList<UserData> = mutableListOf()  
  
    fun addUsers(newUsers: List<UserData>) {  
        userList += newUsers  
        // Update views with `userList`  
    }  
}
```

No need to worry about whether newUsers is mutable or not  
(if addUsers is atomic)

# Shares mutable state: What to check

# Shares mutable state: What to check

Make a mutable value ownership clear

# Shares mutable state: What to check

Make a mutable value ownership clear

Options:

- Modify only by owner's interface
- Create manager class of mutable data

# Content coupling: Summary

Don't rely on internal working

- Remove illegal usage
- Make mutable data ownership clear

# Coupling

Bad/strong coupling: content, common, control

Fine/weak coupling: stamp, data, message

# Common coupling

Shares (mutable) global data

# Common coupling

Shares (mutable) global data

- Pass data by global variables
- Depend on a mutable singleton
- (Use files / databases / shared resources)

# Common coupling

Shares (mutable) global data

- Pass data by global variables
- Depend on a mutable singleton
- (Use files / databases / shared resources)

# Pass data by global variables: Bad example

```
var parameter: Int? = null  
var result: Data? = null
```

# Pass data by global variables: Bad example

```
var parameter: Int? = null  
var result: Data? = null  
  
class Calculator {  
    fun calculate() {  
        result = parameter + ...
```

# Pass data by global variables: Bad example

```
var parameter: Int? = null
var result: Data? = null

class Calculator {
    fun calculate() {
        result = parameter + ...

fun callCalculator() {
    parameter = 42
    calculator.calculate()
```

# Pass data by global variables: Bad points

No information/limitation of call relationships

- Breaks other simultaneous calls
- Tends to cause illegal state

# Pass data by global variables: How to fix

Pass as a function parameter or return value

# Pass data by global variables: Fixed code

```
class Calculator {  
    fun calculate(parameter: Int): Int =  
        parameter + ...
```

# Pass data by global variables: Fixed code

```
class Calculator {  
    fun calculate(parameter: Int): Int =  
        parameter + ...  
  
fun callCalculator() {  
    val result = calculator.calculate(42)
```

# Common coupling

Shares (mutable) global data

- Pass data by global variables
- Depend on a mutable singleton
- (Use files / databases / shared resources)

# Depend on a mutable singleton: Bad example

```
val USER_DATA_REPOSITORY = UserDataRepository()
```

# Depend on a mutable singleton: Bad example

```
val USER_DATA_REPOSITORY = UserDataRepository()

class UserListUseCase {
    suspend fun invoke(): List<User> = withContext(...) {
        val result = USER_DATA_REPOSITORY.query(...)
        ...
    }
}
```

# Mutable global variables: Bad points

## Singleton is not managed

- Unmanaged singleton lifecycle, scope, and references
- Not robust for specification change
- Bad testability

# Mutable global variables: How to fix

Pass an instance as a constructor parameter

(= Dependency injection)

The constructor caller can manage the instance

# Mutable global variables: Fixed code

```
class UserListUseCase(  
    val userDataRepository: UserDataRepository  
) {  
    suspend fun invoke(): List<User> = withContext(...) {  
        val result = userDataRepository.query(...)  
        ...  
    }  
}
```

# Common coupling: Note

Apply the same discussion to a smaller scope

- module, package, type ...

Example: Use a return value instead of instance field

# Common coupling in a class: Bad example

```
class Klass {  
    var result: Result? = null  
  
    fun firstFunction() {  
        secondFunction()  
        /* use `result` */  
        ...  
  
    }  
  
    fun secondFunction() {  
        result = /* calculate result */  
    }  
}
```

# Common coupling in a class: Fixed code

```
class Klass {  
  
    fun firstFunction() {  
        val result = secondFunction()  
        /* use `result` */  
        ...  
  
    }  
  
    fun secondFunction(): Int =  
        /* calculate a return value */
```

# Common coupling: Summary

No more singletons nor global mutable values

- Unmanageable, not robust, hard to test ...

# Common coupling: Summary

No more singletons nor global mutable values

- Unmanageable, not robust, hard to test ...

To remove common coupling:

- Pass value as arguments or a return value
- Use dependency injection

# Coupling

Bad/strong coupling: content, common, control

Fine/weak coupling: stamp, data, message

# Control coupling

Switches logic by passing flag "what to do"

- Has large conditional branch covering procedure
- Each branch body is unrelated to the other branches

# Control coupling example: Boolean flag

```
fun updateView(isError: Boolean) {  
    if (isError) {  
        resultView.isVisible = true  
        errorView.isVisible = false  
        iconView.image = CROSS_MARK_IMAGE  
    } else {  
        resultView.isVisible = false  
        errorView.isVisible = true  
        iconView.image = CHECK_MARK_IMAGE  
    }  
}
```

# Control coupling example: "To do" type

```
fun updateUserView(dataType: DataType) = when(dataType) {  
    UserName -> {  
        val userName = getUserName()  
        userNameView.text = userName  
    }  
    Birthday -> {  
        val birthdayMillis = ...  
        birthDayView.text = format(...)  
    }  
    ProfileImage -> ...  
}
```

# Control coupling: Bad points

## Unreadable

- Hard to summarize what it does
- Difficult to name and comment

# Control coupling: Bad points

## Unreadable

- Hard to summarize what it does
- Difficult to name and comment

## Not robust

- Requires the similar conditional branch to the callers
- Fragile against condition update

# Ideas for improvement

- Remove conditional branch
- Split by object (introduced last session)
- Apply strategy pattern

# Ideas for improvement

- Remove conditional branch
- Split by object (introduced last session)
- Apply strategy pattern

# Remove conditional branch

Split procedures for each argument

- Remove parameter from each new procedure

# Remove conditional branch

## Split procedures for each argument

- Remove parameter from each new procedure

Applicable if "what to do" value is different for each caller

- If not, causes another content coupling on the caller side

# Remove conditional branch: Example

```
fun updateUserView(dataType: DataType) = when(dataType) {  
    UserName -> {  
        val userName = getUserName()  
        userNameView.text = userName  
    }  
    Birthday -> {  
        val birthdayMillis = ...  
        birthDayView.text = format(...)  
    }  
    ProfileImage -> ...  
}
```

# Remove conditional branch: Example

```
fun updateUserNameView() {  
    val userName = getUserName()  
    userNameView.text = userName  
}  
  
fun updateBirthdayView() {  
    val birthdayMillis = ...  
    birthDayView.text = format(...)  
}  
  
fun updateProfileImage() { ... }
```

# Ideas for improvement

- Remove conditional branch
- Split by object (introduced last session)
- Apply strategy pattern

# Split by object

Create code block for each target object, not condition

- Break down content coupling into smaller scope

# Split by object

Create code block for each target object, not condition

- Break down content coupling into smaller scope

Applicable if target objects are common for each condition

- doSomethingIf... function may be used

# Split by object: Example

```
fun updateView(isError: Boolean) {  
    if (isError) {  
        resultView.isVisible = true  
        errorView.isVisible = false  
        iconView.image = CROSS_MARK_IMAGE  
    } else {  
        resultView.isVisible = false  
        errorView.isVisible = true  
        iconView.image = CHECK_MARK_IMAGE  
    }  
}
```

# Split by object: Example

```
fun updateView(isError: Boolean) {  
    resultView.isVisible = !isError  
    errorView.isVisible = isError  
    iconView.image = getIconImage(isError)  
}  
  
fun getIconImage(isError: Boolean): Image =  
    if (!isError) CHECK_MARK_IMAGE else CROSS_MARK_IMAGE
```

# Ideas for improvement

- Remove conditional branch
- Split by object (introduced last session)
- Apply strategy pattern

# Apply strategy pattern

Create types to decide "what to do"

- Branch with sub-typing, polymorphism, or parameter
- Implemented by an enum, an abstract class, or a callback object

# Apply strategy pattern

Create types to decide "what to do"

- Branch with sub-typing, polymorphism, or parameter
- Implemented by an enum, an abstract class, or a callback object

Type selection logic appears on the caller side

# Apply strategy pattern: Example

```
enum class Binder(val viewId: ViewId) {  
  
    fun updateView(holder: ViewHolder) =  
        holder.getView(viewId).let(::setContent)  
  
    abstract fun setContent(view: View)
```

# Apply strategy pattern: Example

```
enum class Binder(val viewId: ViewId) {  
    USER_NAME(USER_NAME_VIEW_ID) {  
        override fun setContent(...) = ...  
    }  
    BIRTHDAY(BIRTHDAY_VIEW_ID) {  
        override fun setContent(...) = ...  
    }  
    ...  
  
    fun updateView(holder: ViewHolder) =  
        holder.getView(viewId).let(::setContent)  
  
    abstract fun setContent(view: View)
```

# Control coupling: Summary

Don't create large conditional branch covering a whole procedure

# Control coupling: Summary

Don't create large conditional branch covering a whole procedure

To lease control coupling:

- Remove condition parameters
- Split by target instead of condition
- Use strategy pattern

# Coupling

Bad/strong coupling: content, common, control

Fine/weak coupling: stamp, data, message

# Stamp coupling

Passes a structure though some of the data that are not used

# Stamp coupling

Passes a structure though some of the data that are not used

Acceptable for some situations:

- To use sub-typing, structural typing, or strategy pattern
- To simplify parameter types

Make parameter type as small as possible

# Stamp coupling: Example

updateUserView(userData)

```
fun updateUserView(userData: UserData) {
```

# Stamp coupling: Example

updateUserView(userData)

```
fun updateUserView(userData: UserData) {  
    // Some property of `userData` are used  
    userNameView.text = userData.fullName  
    profileImage.setImageUrl(userData.profileImageUrl)  
  
    // `userData.mailAddress` and `phoneNumber` are not used  
}
```

# Coupling

Bad/strong coupling: content, common, control

Fine/weak coupling: stamp, **data**, message

# Data coupling

Passes a set of elemental data

- All the data must be used

```
fun updateUserView(fullName: String, profileImageUrl: Url) {  
    userNameView.text = fullName  
    profileImage.setImageUrl(profileImageUrl)  
}
```

```
updateUserView(userData.fullName, userData.profileImageUrl)
```

# Coupling

Bad/strong coupling: content, common, control

Fine/weak coupling: stamp, data, message

# Message coupling

Call a method without any data

- Hardly happens
- May cause content coupling in another layer

# Message coupling

## Call a method without any data

- Hardly happens
- May cause content coupling in another layer

```
fun notifyUserDataUpdated() {  
    userNameView.text = ...  
    profileImage.setImageUrl(...)  
}
```

notifyUserDataUpdated()

# Coupling: Summary

## Avoid strong coupling

- Use stamp/data coupling instead of content/common/control

# Coupling: Summary

## Avoid strong coupling

- Use stamp/data coupling instead of content/common/control

Content coupling: Relies on internal working

# Coupling: Summary

## Avoid strong coupling

- Use stamp/data coupling instead of content/common/control

Content coupling: Relies on internal working

Common coupling: Depends on global singleton/variables

# Coupling: Summary

## Avoid strong coupling

- Use stamp/data coupling instead of content/common/control

**Content coupling:** Relies on internal working

**Common coupling:** Depends on global singleton/variables

**Control coupling:** Has large conditional branch

Code ← readability ← session 7

Dependency ↑↑

# Contents of this lecture

- Introduction and Principles
- Natural language: Naming, Comments
- Inner type structure: State, Procedure
- Inter type structure: Dependency (two sessions)
- Follow-up: Review

# Topics

## First session:

- Coupling

## Second session:

- Direction
- Redundancy
- Explicitness

# Direction

Keep the dependency in one direction as far as possible

= It's good if there's no cycle

# Preferred dependency direction: Example

- Caller to callee
- Detail to abstraction
- Complex to simple
- Algorithm to data model
- Mutable to immutable
- Frequently updated layer to stable layer

# Preferred dependency direction

- Caller to callee
- Detail to abstraction
- Complex to simple

# Preferred dependency direction

- Caller to callee
- Detail to abstraction
- Complex to simple

# Bad code example

```
class MediaViewPresenter {  
    fun getVideoUri(): Uri = ...  
  
    fun playVideo() {  
        videoPlayerView.play(this)  
    }  
}
```

# Bad code example

```
class MediaViewPresenter {  
    fun getVideoUri(): Uri = ...  
  
    fun playVideo() {  
        videoPlayerView.play(this)  
        ...  
    }  
  
    class VideoPlayerView {  
        fun play(presenter: MediaViewPresenter) {  
            val uri = presenter.getVideoUri()  
        }  
    }  
}
```

# What is to be checked

## Call stack of the example

- MediaViewPresenter.playVideo()
- VideoPlayerView.play(...)
- MediaViewPresenter.getVideoUri()

# What is to be checked

## Call stack of the example

- MediaViewPresenter.playVideo()
- VideoPlayerView.play(...)
- MediaViewPresenter.getVideoUri()

A call stack of A → B → A is a typical bad signal

# How to remove reference to caller

## Remove unnecessary callback

- Pass values as parameters
- Extract small classes

# How to remove reference to caller

## Remove unnecessary callback

- Pass values as parameters
- Extract small classes

# Pass values as parameters

```
class MediaViewPresenter {  
    fun getVideoUri(): Uri = ...  
  
    fun playVideo() {  
        videoPlayerView.play(this)  
        ...  
    }  
  
    class VideoPlayerView {  
        fun play(presenter: MediaViewPresenter) {  
            val uri = presenter.getVideoUri()  
        }  
    }  
}
```

# Pass values as parameters

```
class MediaViewPresenter {  
    fun getVideoUri(): Uri = ...  
  
    fun playVideo() {  
        videoPlayerView.play(getVideoUri())  
    }  
    ...  
}
```

```
class VideoPlayerView {  
    fun play(videoUri: Uri) {  
        ...  
    }  
}
```

# How to remove reference to caller

## Remove unnecessary callback

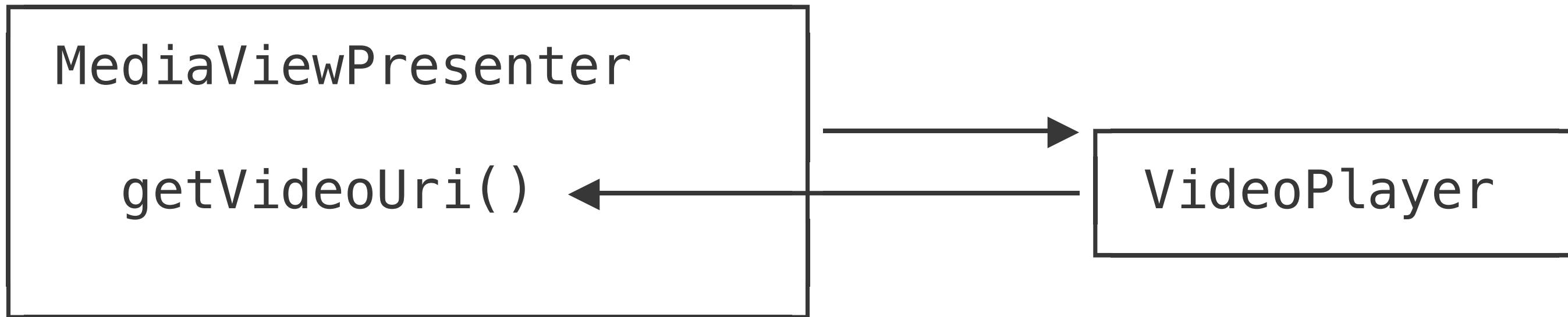
- Pass values as parameters
- Extract small classes

# Extract small classes

Can't pass value directly for asynchronous evaluation

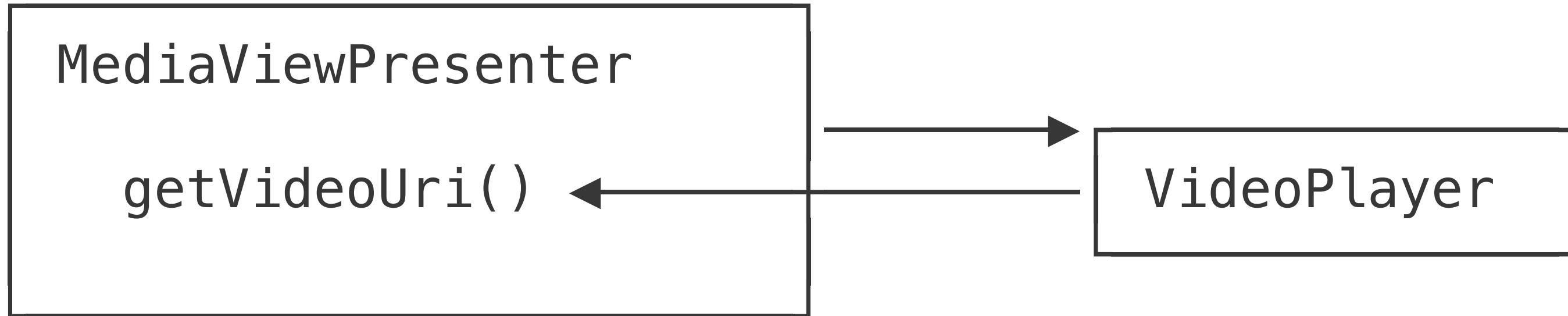
# Extract small classes

Can't pass value directly for asynchronous evaluation



# Extract small classes

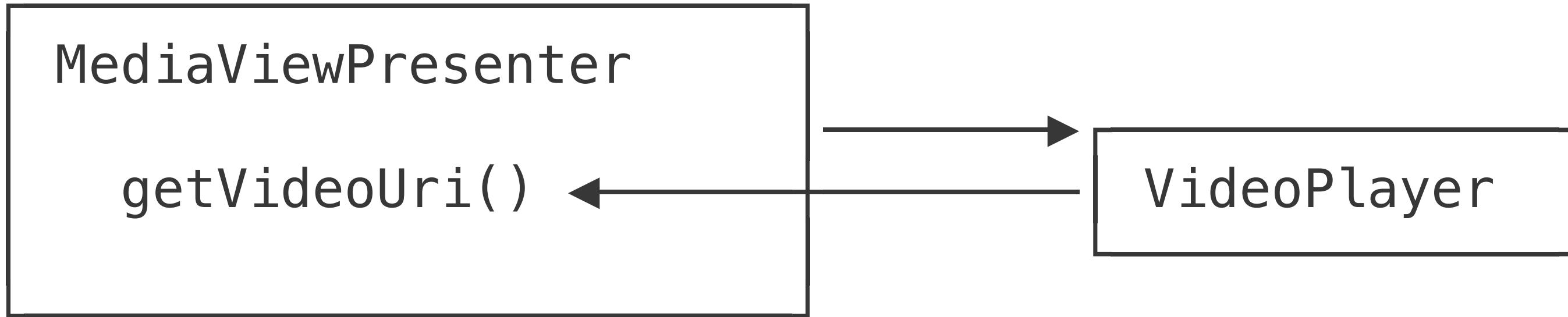
Can't pass value directly for asynchronous evaluation



Remove callback by extracting depended logic

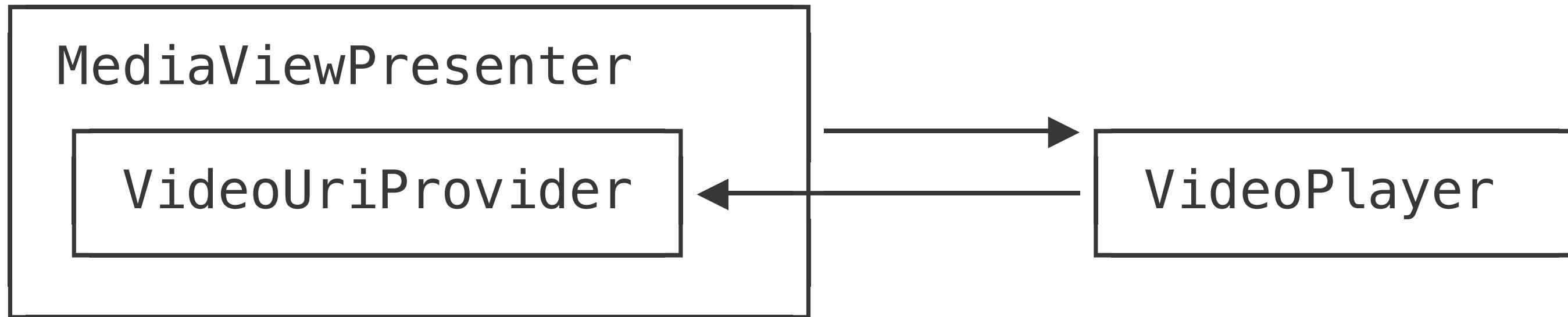
# Extract small classes: Step 1

Create a named class for the callback



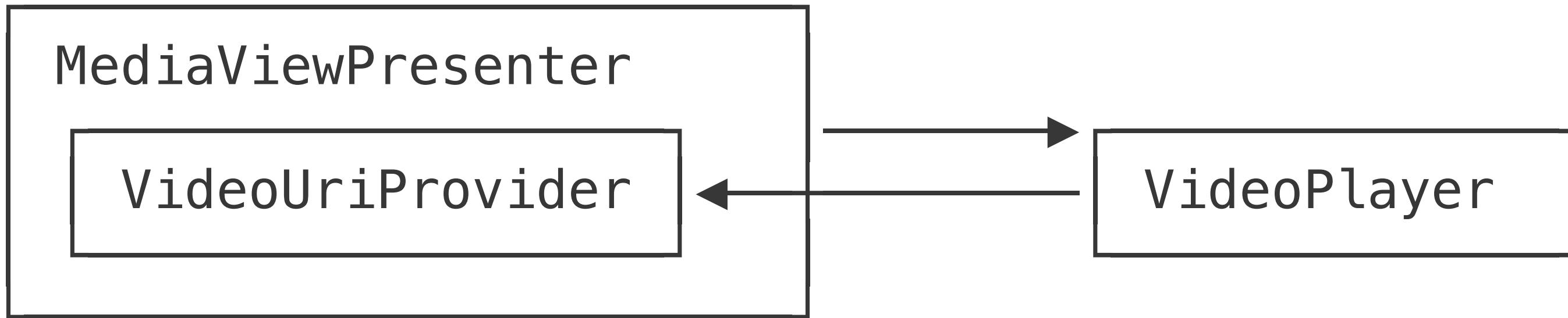
# Extract small classes: Step 1

Create a named class for the callback



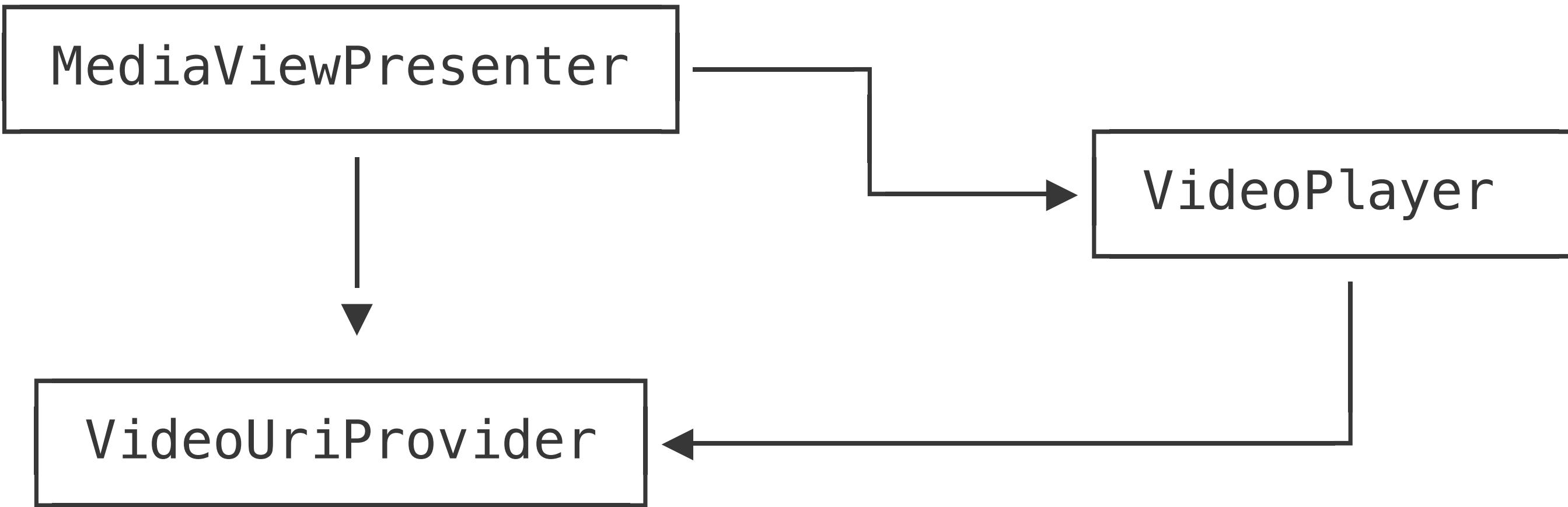
# Extract small classes: Step 2

Move the named class out of the outer class



# Extract small classes: Step 2

Move the named class out of the outer class



# How to remove reference to caller

## Remove unnecessary callback

- Pass values as parameters
- Extract small classes

# Dependency on caller: Exceptions

Dependency on a caller may be acceptable for the following:

# Dependency on caller: Exceptions

Dependency on a caller may be acceptable for the following:

- Threads
- Event listeners
- Strategy pattern like functions (e.g., forEach)

# Dependency on caller: Exceptions

Dependency on a caller may be acceptable for the following:

- Threads
- Event listeners
- Strategy pattern like functions (e.g., forEach)

Alternative options: Promise pattern, coroutine, reactive...

# Preferred dependency direction

- Caller to callee
- Detail to abstraction
- Complex to simple

# Dependency on detail

Don't depend on inheritances

= Don't check type of this or self

Exception: Sealed class

# Bad code example

```
open class IntList {  
    fun addElement(value: Int) {  
        if (this is ArrayList) {  
            ...  
        } else {  
            ...  
        }  
    }  
  
    class ArrayList: IntList() {  
        ...  
    }  
}
```

# What's wrong with the bad example

Brakes encapsulation and not robust

- For inheritance specification change
- For adding a new inheritance

# Fixed code example

## Remove downcast with overriding

```
open class IntList {  
    open fun addElement(value: Int) {  
        ...  
  
class ArrayList: IntList() {  
    override fun addElement(value: Int) {  
        ...
```

# Dependency direction

- Caller to callee
- Detail to abstraction
- Complex to simple

# Bad code example

```
class UserData(  
    val userId: UserId,  
    ...  
)  
  
class UserDataRequester {  
    fun obtainFromServer(): UserData {  
        ...  
    }  
}
```

# Bad code example

```
class UserData(  
    val userId: UserId,  
    ...  
    val requester: UserDataRequester  
)  
  
class UserDataRequester {  
    fun obtainFromServer(): UserData {  
        ...  
    }  
}
```

# What's wrong with the bad example

Unnecessary dependency may cause a bug

- May leak requester resource
- May make inconsistent state if there are multiple requesters

# What's wrong with the bad example

Unnecessary dependency may cause a bug

- May leak requester resource
- May make inconsistent state if there are multiple requesters

A simple type may be used widely and passed to other modules

# How to remove dependency on complex class

Just remove dependency of "simple to complex"

- Pass a complex class instance directly if required

```
class UserData(  
    val userId: UserId,  
    ...  
)
```

# Exceptions

It's sometimes necessary to depend on complex types

- Mediator pattern
- Adapter or facade
- Data binder

# Preferred dependency direction

- Caller to callee
- Detail to abstraction
- Complex to simple

# Direction: Summary

Maintain dependency in one direction

- From caller to callee
- From complex, detailed, large to simple, abstract, small

Exceptions

- Async call, sealed class, mediator pattern

# Topics

## First session:

- Coupling

## Second session:

- Direction
- Redundancy
- Explicitness

# Redundant dependency

## Cascaded dependency

- Nested & indirect dependency

## Redundant dependency set

- N:M dependency

# Redundant dependency

## Cascaded dependency

- Nested & indirect dependency

## Redundant dependency set

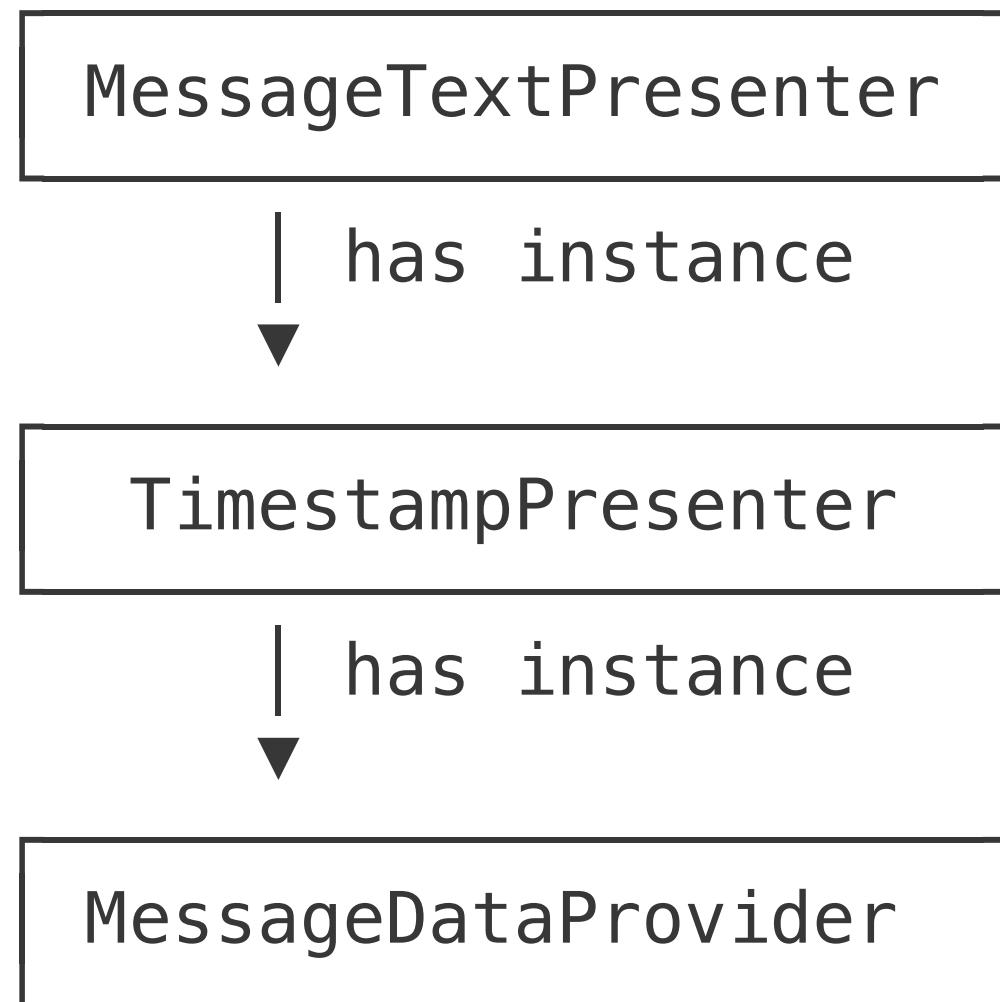
- N:M dependency

# Cascaded dependency

Avoid unnecessary dependency chain

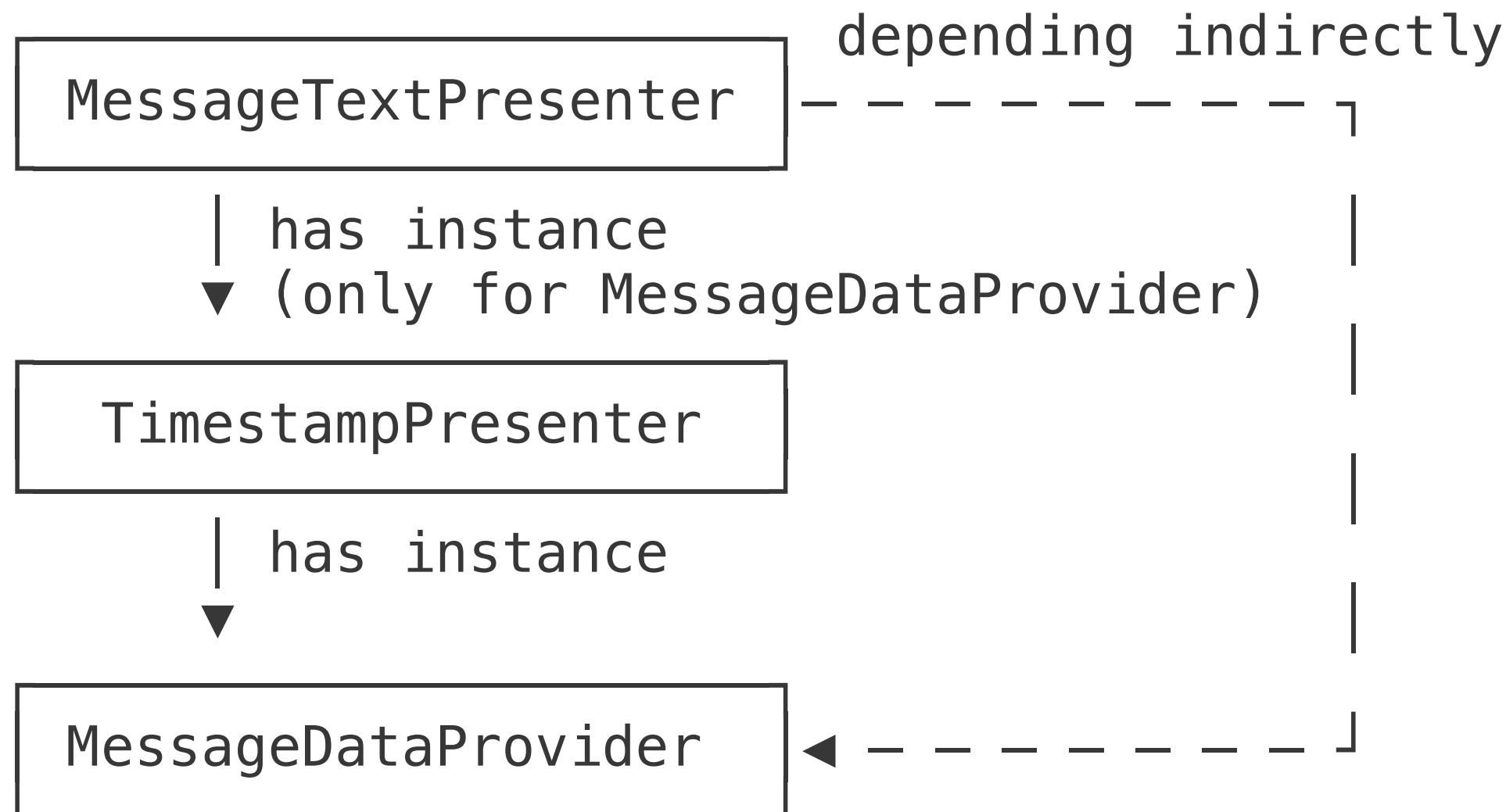
# Cascaded dependency

Avoid unnecessary dependency chain



# Cascaded dependency

Avoid unnecessary dependency chain



# Bad cascaded dependency example

```
class TimestampPresenter {  
    val dataProvider: MessageDataProvider = ...  
    fun invalidateViews() {  
        // Update timestamp by `dataProvider`  
        ...  
    }  
}
```

# Bad cascaded dependency example

```
class TimestampPresenter {  
    val dataProvider: MessageDataProvider = ...  
    fun invalidateViews() {  
        // Update timestamp by `dataProvider`  
        ...  
  
class MessageTextPresenter {  
    val timestampPresenter: TimestampPresenter = ...  
    fun invalidateViews() {  
        val messageData = timestampPresenter.dataProvider...  
        // Update message text view by `messageData`
```

# What's wrong with the bad example

## Unnecessary dependency

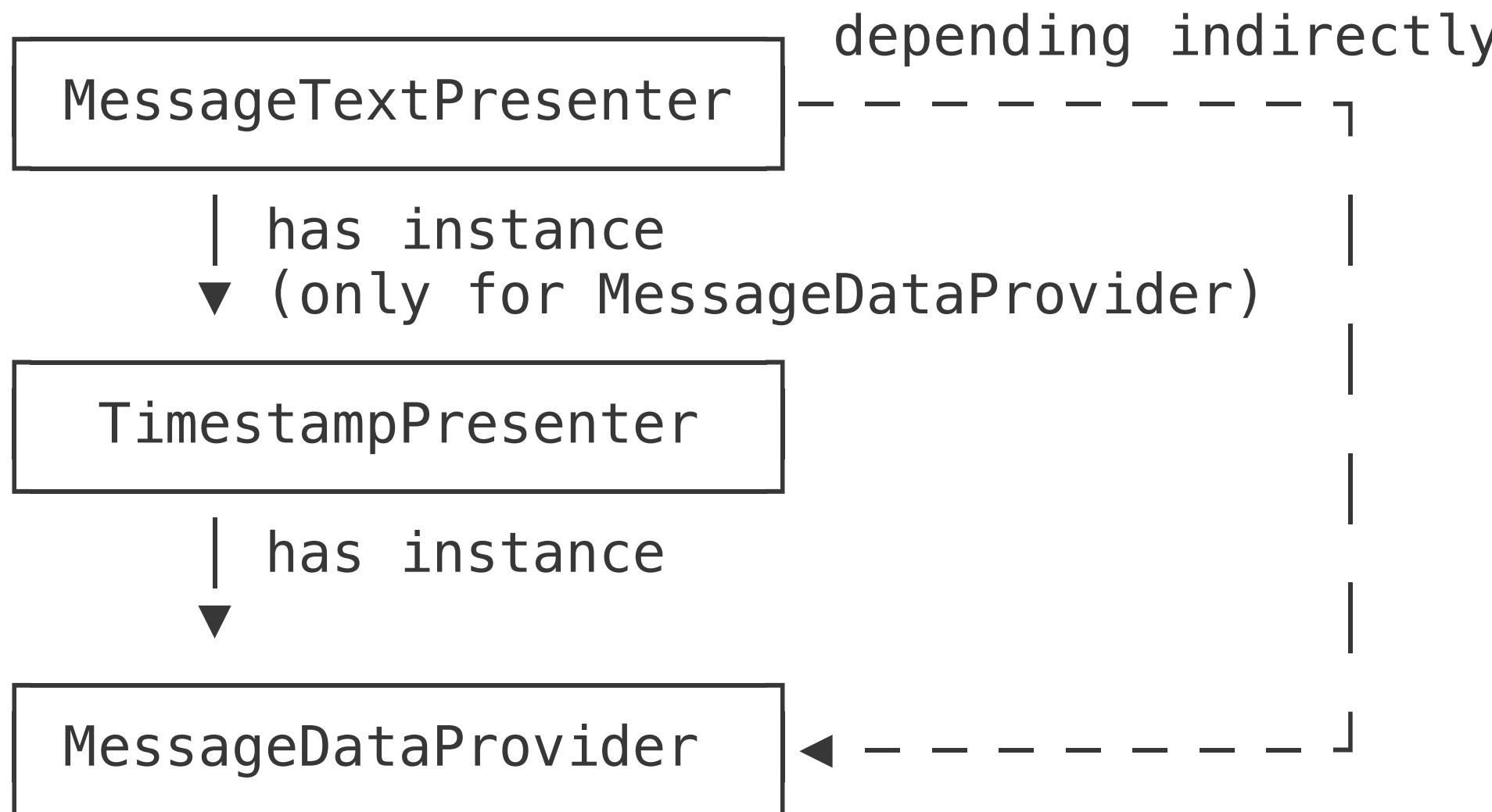
- MessageTextPresenter is unrelated with TimestampPresenter

## Indirect dependency

- MessageTextPresenter does not have MessageDataProvider directly

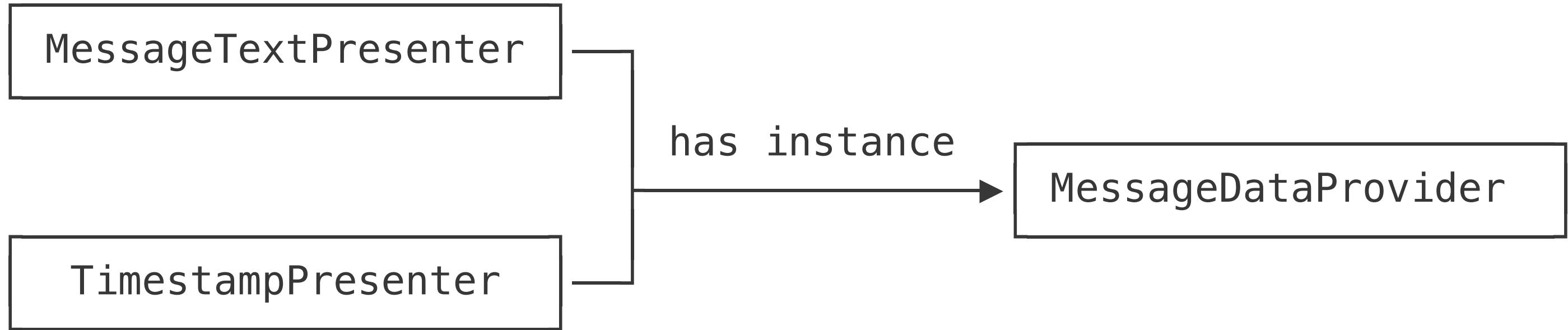
# How to fix cascaded dependency

Flatten nested dependencies by means of direct dependencies



# How to fix cascaded dependency

Flatten nested dependencies by means of direct dependencies



# Fixed example of cascaded dependency

```
class TimestampPresenter {  
    val dataProvider: MessageDataProvider = ...  
    fun invalidateViews() {  
        // Update timestamp by `dataProvider`  
        ...  
  
class MessageTextPresenter {  
    val dataProvider: MessageDataProvider = ...  
    fun invalidateViews() {  
        // Update message text view by `dataProvider`
```

# Redundant dependency

## Cascaded dependency

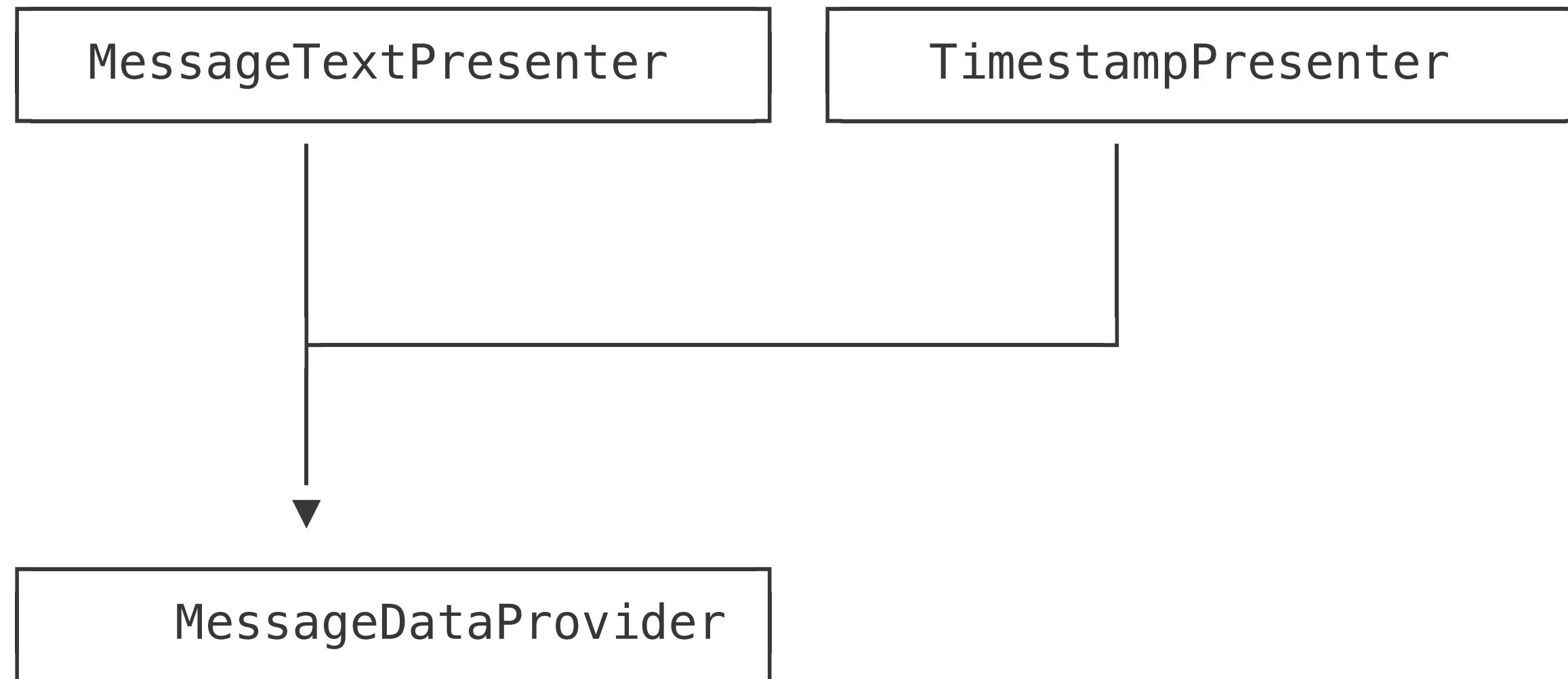
- Nested & indirect dependency

## Redundant dependency set

- N:M dependency

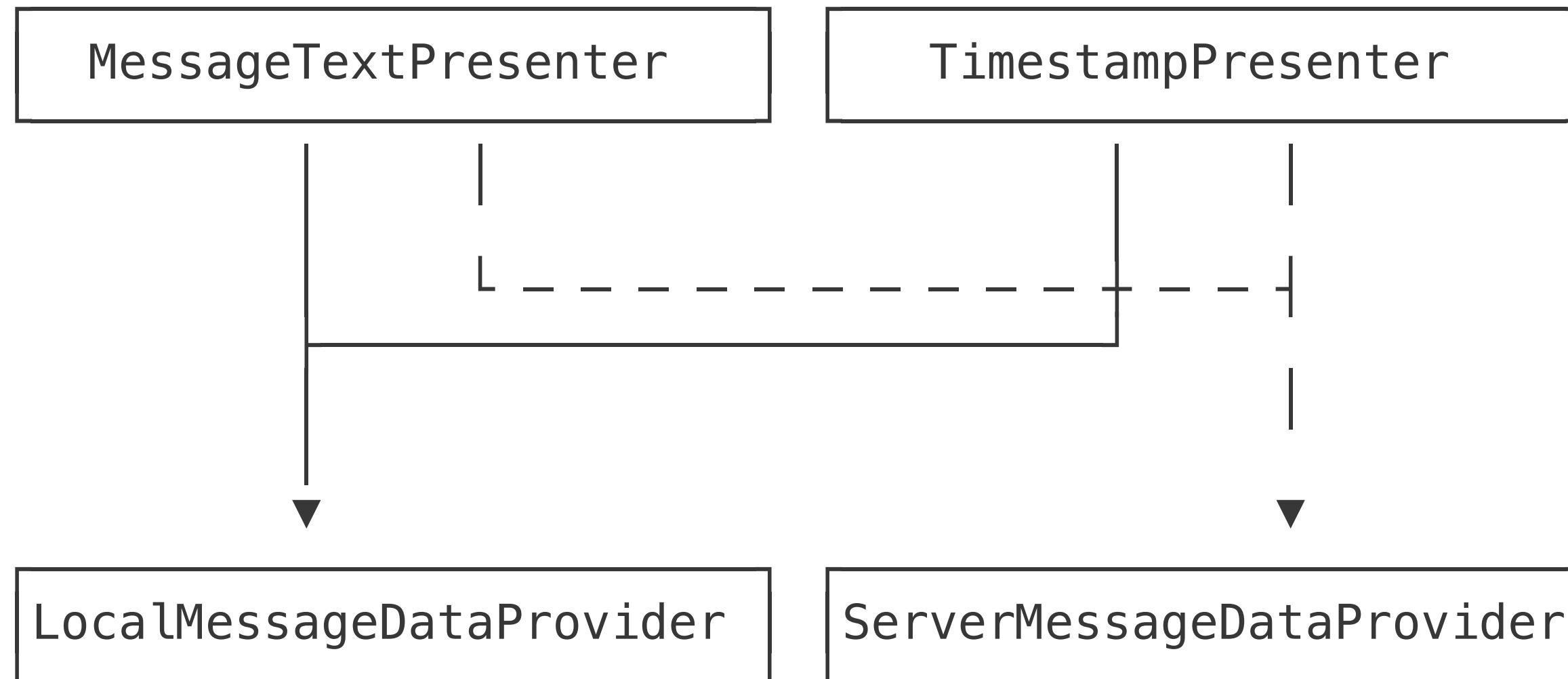
# Redundant dependency set

Avoid duplication of "depending class set"



# Redundant dependency set

Avoid duplication of "depending class set"



# What's wrong with redundant dependency set?

## Fragile for modification

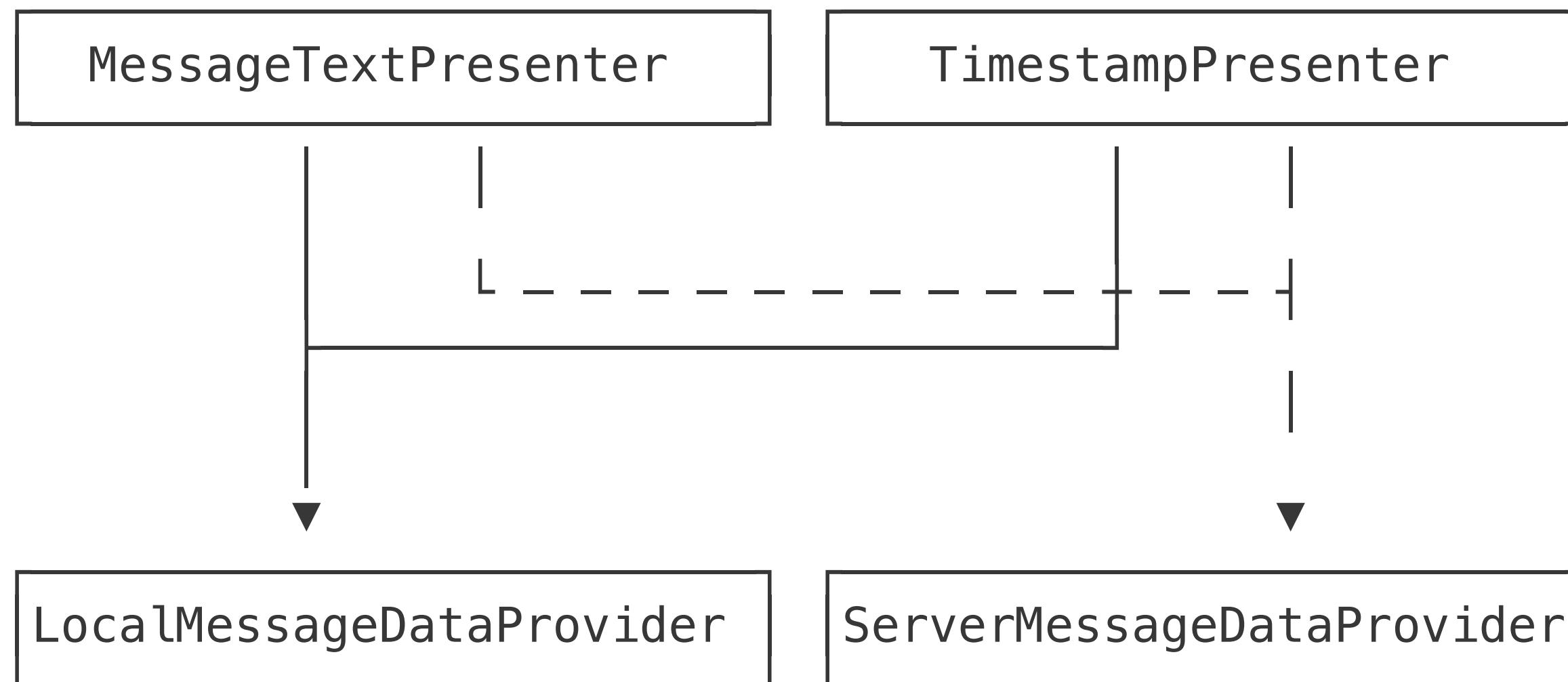
- When a new depending/depended type is added

## Duplicates switching logic

- Provider selection/fallback logic

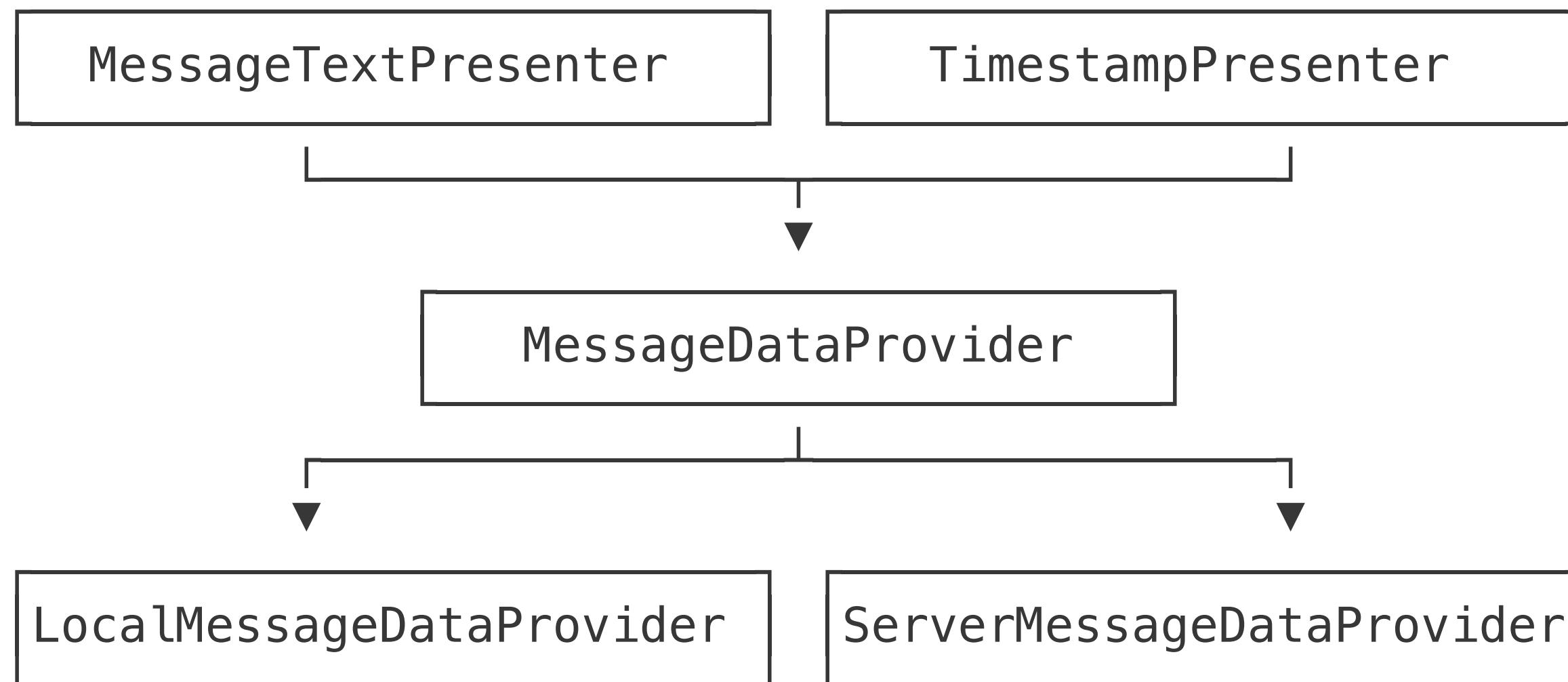
# How to remove redundant dependency set

Create an intermediate layer to consolidate dependency set



# How to remove redundant dependency set

Create an intermediate layer to consolidate dependency set



# Caution to remove redundant dependency

Don't create a layer if you don't need it now

- Keep "YAGNI" and "KISS" in mind

Don't provide access to a hidden class

- Never create `MessageDataProvider.serverMessageDataProvider`

# Redundancy: Summary

- Remove dependencies only for providing another dependency
- Add layer to remove dependency set duplication

# Topics

## First session:

- Coupling

## Second session:

- Direction
- Redundancy
- Explicitness

# Implicit dependency

A dependency does not appear on a class diagram

# Implicit dependency

A dependency does not appear on a class diagram

Avoid implicit dependencies, use explicit ones instead

# Implicit dependency example

- Implicit data domain for parameters
- Implicitly expected behavior of subtype

# Implicit dependency example

- Implicit data domain for parameters
- Implicitly expected behavior of subtype

# Bad code example

Question: What's wrong with the following function?

```
fun setViewBackgroundColor(colorString: String) {  
    val colorCode = when(colorString) {  
        "red" -> 0xFF0000  
        "green" -> 0x00FF00  
        else -> 0x000000  
    }  
    view.setBackgroundColor(colorCode)  
}
```

# Why implicit data domain is bad

Hard to find which value is accepted

- "blue", "RED", "FF0000" are invalid

# Why implicit data domain is bad

Hard to find which value is accepted

- "blue", "RED", "FF0000" are invalid

Color string conversion logic is tightly coupled with the function

- Easy to break if we add/remove a color

# How to remove implicit data domain

Create a data type representing the data domain

```
enum class BackgroundColor(val colorCode: Int) {  
    RED(0xFF0000),  
    GREEN(0x00FF00)  
}
```

# How to remove implicit data domain

Create a data type representing the data domain

```
enum class BackgroundColor(val colorCode: Int) {  
    RED(0xFF0000),  
    GREEN(0x00FF00)  
}  
  
fun setViewBackgroundColor(color: BackgroundColor) {  
    view.setBackgroundColor(color.colorCode)  
}
```

# Other options instead of defining a type

## Options to consider for an invalid value

- Just do nothing
- Return an error value
- Throw a runtime error (not recommended)

# Other options instead of defining a type

## Options to consider for an invalid value

- Just do nothing
- Return an error value
- Throw a runtime error (not recommended)

**Note 1:** Documentation is required for any option

# Other options instead of defining a type

## Options to consider for an invalid value

- Just do nothing
- Return an error value
- Throw a runtime error (not recommended)

**Note 1:** Documentation is required for any option

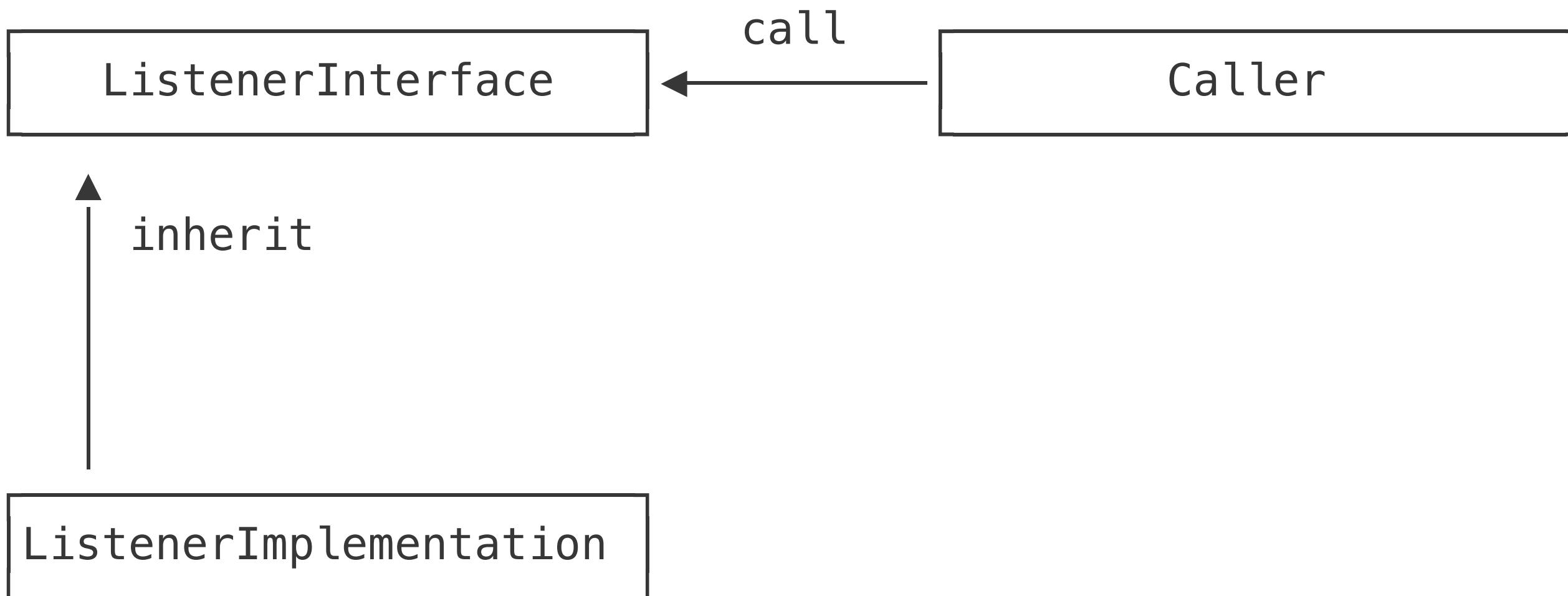
**Note 2:** An "error value" may also require another data domain type

# Implicit dependency example

- Implicit data domain for parameters
- Implicitly expected behavior of subtype

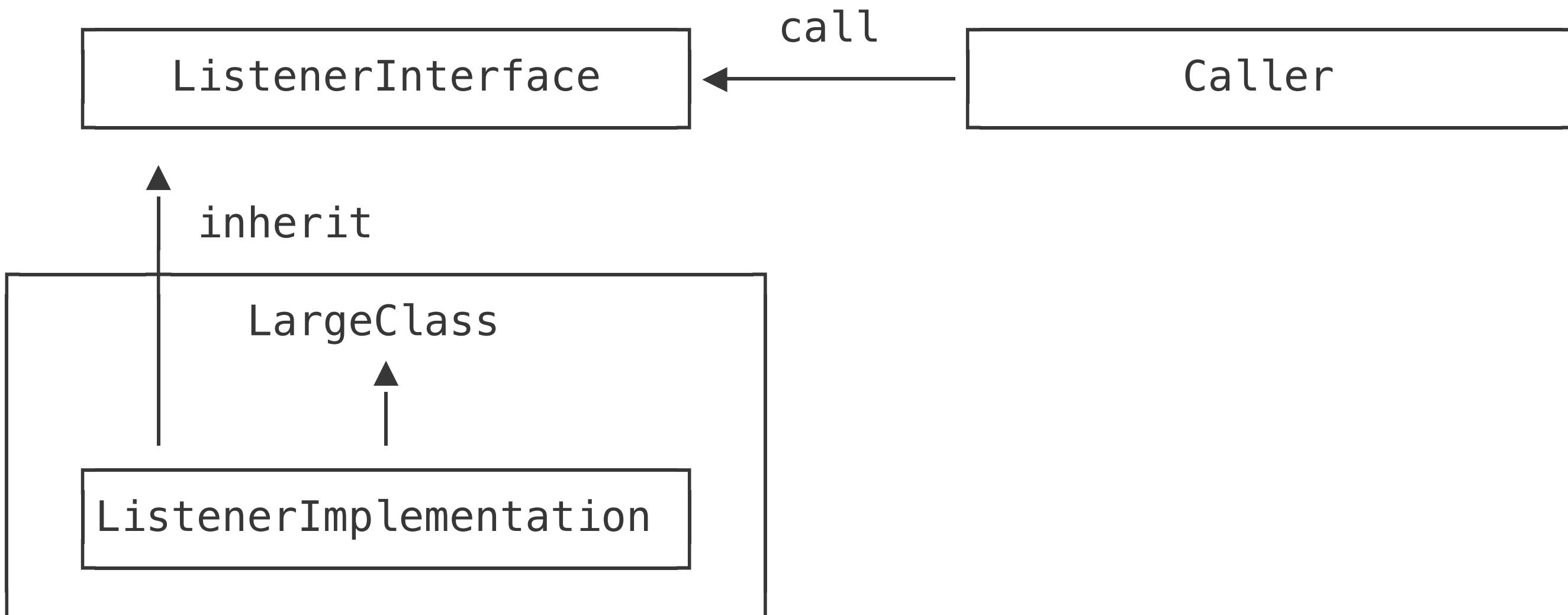
# Implicit dependency by subtype

A caller depends on the inherited type implicitly



# Implicit dependency by subtype

A caller depends on the inherited type implicitly



# Implicit dependency example by subtype

```
interface ListenerInterface { fun queryInt(): Int }

class LargeClass {
    inner class ListenerImplementation : ListenerInterface {
        ...
    }
}

class Caller(listener : ListenerInterface) {
    ...
}
```

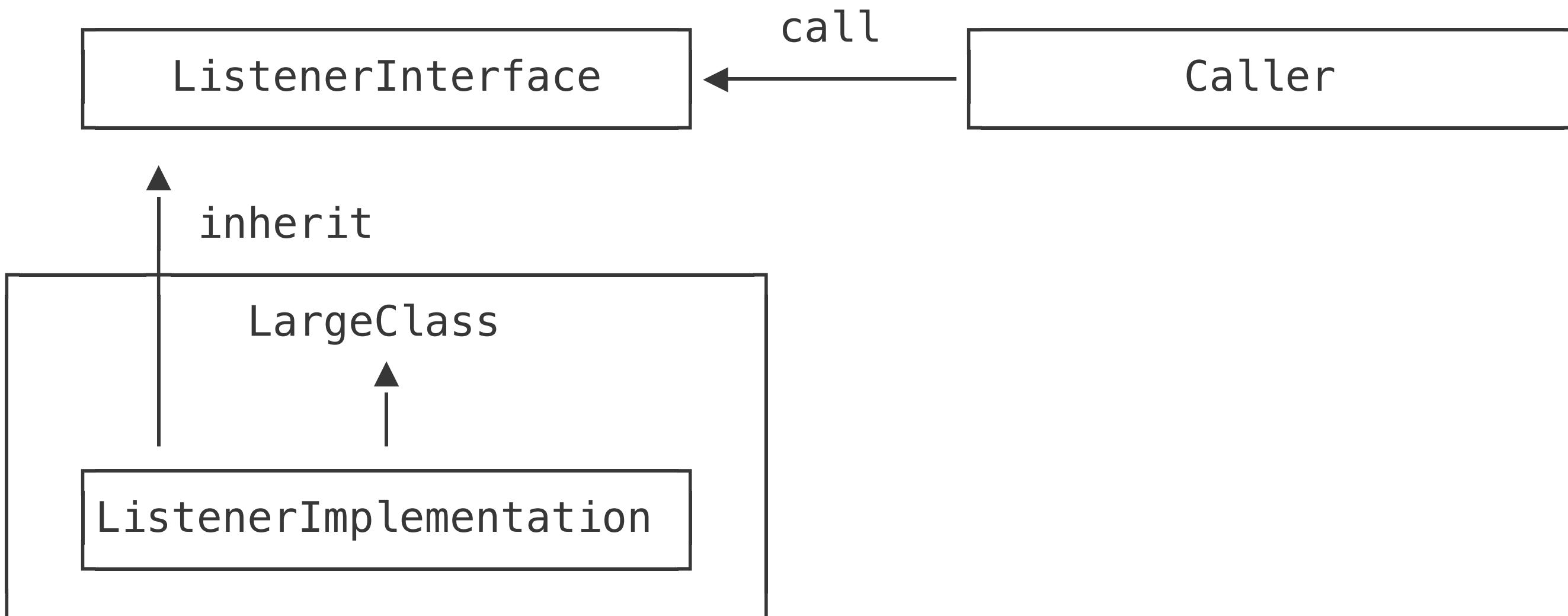
# What's wrong with subtype

Depends on result of the function call on LargeClass implicitly

- Does not appear on class diagram
- Hard to find Caller relying on LargeClass implementation detail

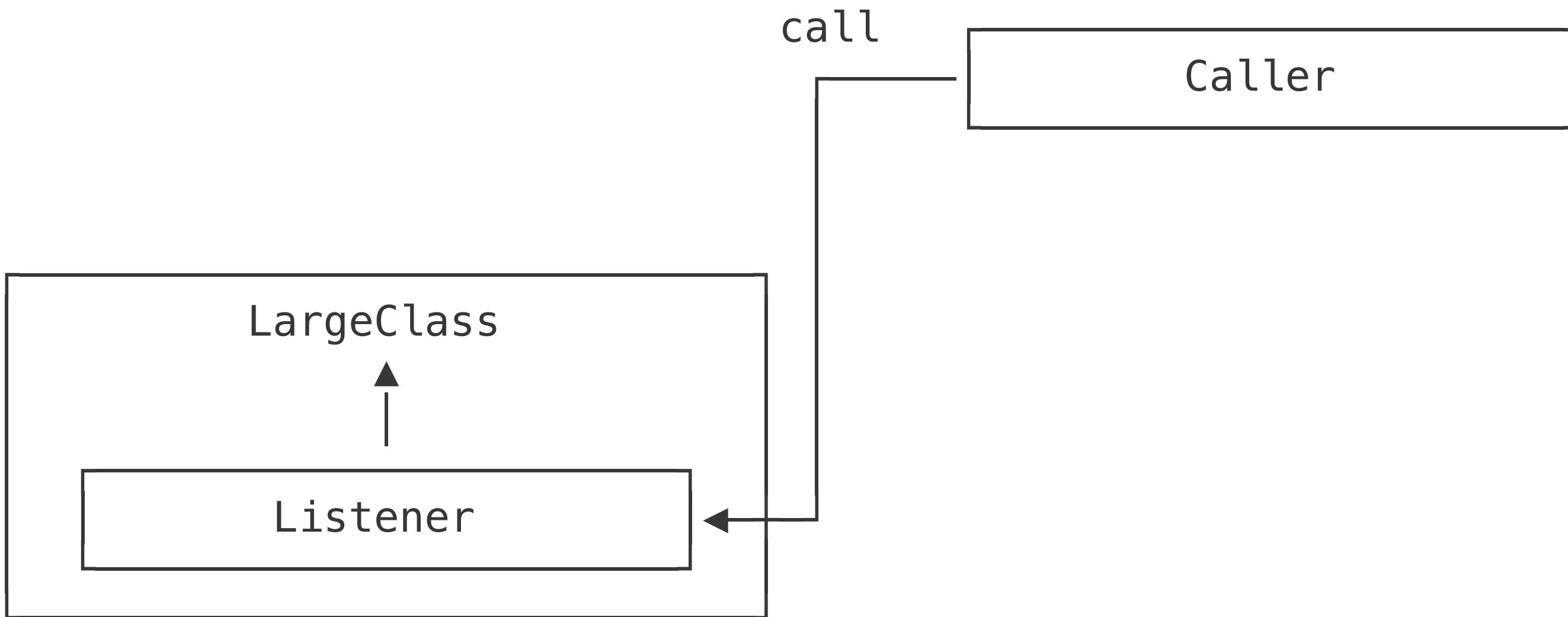
# How to remove implicit dependency

Remove the interface if there is only one implementation



# How to remove implicit dependency

Remove the interface if there is only one implementation



# Fixed code example

```
class LargeClass {  
    inner class Listener {  
        fun ...  
    }  
}  
  
class Caller(listener : Listener) {  
    ...  
}
```

# Explicitness: Summary

Make dependencies visible on a class diagram

- Define types to represent data domain
- Remove unnecessary interface

# Summary

- **Coupling:** Relax content/common/control coupling
- **Direction:** Remove cyclic dependency as far as possible
- **Redundancy:** Don't cascade or duplicate dependency set
- **Explicitness:** Make dependency visible on a class diagram

Code-Readability / Session-8

Review #1

# Contents of this lecture

- Introduction and Principles
- Natural language: Naming, Comments
- Inner type structure: State, Procedure
- Inter type structure: Dependency (two sessions)
- Follow-up: Review

# Why do we need code review?

# Why do we need code review?

## Most important: Readability

- Approve code change only when it is clean enough

# Why do we need code review?

## Most important: Readability

- Approve code change only when it is clean enough

## Nice to have: Logic correctness

- Edge cases, illegal state, race condition, exceptions...
- Author's responsibility

# Topics

## Authors:

- How to create PR
- How to address review comment

## Reviewers:

- Principles
- Comment contents

# Topics

## Authors:

- How to create PR
- How to address review comment

## Reviewers:

- Principles
- Comment contents

# How to create PR

- Tell your intention with commit structure
- Keep your PR small

# How to create PR

- Tell your intention with commit structure
- Keep your PR small

# Commit structure

- Squash unnecessary commits
- Don't put logical change and syntax change into a commit
- Think "cut of axis" for commits

# Commit structure

- Squash unnecessary commits
- Don't put logical change and syntax change into a commit
- Think "cut of axis" for commits

# Unnecessary commits

Remove or squash "reverting commit"

# Unnecessary commits

Remove or squash "reverting commit"

Bad example:

Commit 1: Add debug log

Commit 2: Implement a method of feature X

Commit 3: Remove debug log

# How to remove unnecessary commits

Use `git rebase -i` or similar command

# How to remove unnecessary commits

Use `git rebase -i` or similar command

Fixed example:

Commit 2: Implement a method of feature X

# Commit structure

- Squash unnecessary commits
- Don't put logical change and syntax change into a commit
- Think "cut of axis" for commits

# Logical and syntax change in a commit

## Typical anti-patterns

- IDE auto clean up + actual refactoring
- Extracting as a function + function body refactoring
- Renaming + non-trivial parameter type change

# Logical and syntax change in a commit

## Typical anti-patterns

- IDE auto clean up + actual refactoring
- Extracting as a function + function body refactoring
- Renaming + non-trivial parameter type change

If it's hard to read, consider splitting the commit

# Commit structure

- Squash unnecessary commits
- Don't put logical change and syntax change into a commit
- Think "cut of axis" for commits

# Example of splitting PR

```
class Deprecated {  
    fun functionA() { /* ... */ }  
    fun functionB() { /* ... */ }  
}
```

```
class New { /* ... */ }
```

# Example of splitting PR

```
class Deprecated {  
    fun functionA() { /* ... */ }  
    fun functionB() { /* ... */ }  
}
```

```
class New { /* ... */ }
```

**Objective:**

Remove class Deprecated with moving the functions into class New

# Axis options to consider

## Option A:

Commit 1: Define New.functionA/functionB

Commit 2: Replace receivers Deprecated with New

Commit 3: Remove class Deprecated

# Axis options to consider

## Option A:

Commit 1: Define New.functionA/functionB

Commit 2: Replace receivers Deprecated with New

Commit 3: Remove class Deprecated

## Option B:

Commit 1: Move Deprecated.functionA to New, and update the callers

Commit 2: Move Deprecated.functionB to New, and update the callers

Commit 3: Remove class Deprecated

# Better option for axis

# Better option for axis

Option B is better than A

# Better option for axis

Option B is better than A

Reason:

- Easy to understand the intention
- Verifiable that there is no logical change
- Traceable commit history

# How to create PR

- Tell your intention with commit structure
- Keep your PR small

# Keep your PR small

- Split your daily work into several PRs
- Create supporting PRs
- Specify the scope of a PR

# Keep your PR small

- Split your daily work into several PRs
- Create supporting PRs
- Specify the scope of a PR

# Split your daily work into several PRs

Don't create a large PR like "PR of the week"

- Hard to read for reviewers
- Makes review iteration long

Two ways to split PRs: Top-down, Bottom-up

# Top-down approach

Fill implementation detail later

# Top-down approach

Fill implementation detail later

1. Create skeleton classes to explain the structure
2. Explain future plan with TODO and PR comments

# Top-down approach

Fill implementation detail later

1. Create skeleton classes to explain the structure
2. Explain future plan with TODO and PR comments

```
class UserProfilePresenter(  
    val userProfileUseCase: UseCase  
    val profileRootView: View  
) {  
    fun showProfileImage() = TODO(...)  
    fun addUserTag() = TODO(...)
```

# Bottom-up approach

Create small parts first

# Bottom-up approach

Create small parts first

1. Implement simple types/procedure without any client code
2. Explain future plan of client code with PR comments

# Bottom-up approach

## Create small parts first

1. Implement simple types/procedure without any client code
2. Explain future plan of client code with PR comments

```
data class UserProfileData(val ..., val ...)
```

```
object UserNameStringUtils {  
    fun String.normalizeEmoji(): String = ...  
    fun isValidUserName(userName: String): Boolean = ...
```

# Keep your PR small

- Split your daily work into several PRs
- Create supporting PRs
- Specify the scope of a PR

# Create supporting PRs

**Question:** How can you make PRs with the following situation?

1. You made commits of class New
2. You found New has code duplication with Old
3. You need a large change to consolidate the logic (>100 LOC)
  - Because there are already a lot of callers of Old

# Worst way to create a PR

Commit1: Create a class "New"

Commit2: Extract duplicated code of "New" and "Old" # >100 LOC

**Problem:** The PR is large, and has two responsibilities

# Better way to create a PR

PR1:

Commit1: Create a class "New"

PR2 (including PR1):

Commit2: Extract duplicated code of "New" and "Old" # >100 LOC

**Problem:** There is temporary code duplication

- We may forget to create PR2

# Best way to create a PR 1/3

## Reorder, split, merge commits

### 1: Split Commit2 for New and Old

Commit1: Create a class "New"

Commit2\_New: Extract code of "New"

Commit2\_Old: Extract code of "Old"

# Best way to create a PR 2/3

## 2: Move Commit2\_Old to the first

Commit2\_Old: Extract code of "Old"

Commit1: Create a class "New"

Commit2\_New: Extract code of "New"

# Best way to create a PR 3/3

## 3: Squash commits Commit1 and Commit2\_New

Commit2\_Old: Extract code of "Old"

Commit1: Create a class "New"

## 4: Create PRs for Commit2\_Old and Commit1

# Keep your PR small

- Split your daily work into several PRs
- Create supporting PRs
- Specify the scope of a PR

# Conflict between author and reviewer

## Author's responsibility:

- Keep the PR small

# Conflict between author and reviewer

## Author's responsibility:

- Keep the PR small

## Reviewer's responsibility:

- Don't approve if there is room to improve

# Conflict between author and reviewer

Author's responsibility:

- Keep the PR small

Reviewer's responsibility:

- Don't approve if there is room to improve

These two responsibilities may conflict with each other

# Describe scope of PR

Write PR comment and TODO comment to explain:

- The main purpose
- Future plan
- Out of scope

# How to create PR: Summary

Make readable PR that is structured and small

- Make the objective of a PR/commits clear
- Squash, split, reorder commits

# Topics

## Authors:

- How to create PR
- How to address review comment

## Reviewers:

- Principles
- Comment contents

# Important point in addressing comments

# Important point in addressing comments

Don't just address comments

# Don't just address comments

- Think why reviewer asked/misunderstood
- Understand reviewer's suggestion
- Consider addressing a comment to other parts
- Don't make reviewers repeat

# Don't just address comments

- Think why reviewer asked/misunderstood
- Understand reviewer's suggestion
- Consider addressing a comment to other parts
- Don't make reviewers repeat

# Think why reviewer asked/misunderstood

Typical signal that the code is unreadable or wrong

# Think why reviewer asked/misunderstood

Typical signal that the code is unreadable or wrong

Example of comment "Why do you check null here?":

- Remove the null check if unnecessary
- Write inline comment to explain why
- Extract as a local value to explain (e.g., `isUserExpired`)

# Don't just address comments

- Think why reviewer asked/misunderstood
- Understand reviewer's suggestion
- Consider addressing a comment to other parts
- Don't make reviewers repeat

# Understand reviewer's suggestion

Don't just paste attached code in a review comment

- Think what is the key idea
- Consider renaming
- Confirm it's logically correct: exceptions, race conditions ...

# Don't just address comments

- Think why reviewer asked/misunderstood
- Understand reviewer's suggestion
- Consider addressing a comment to other parts
- Don't make reviewers repeat

# Consider addressing a comment to other parts

Find similar code and address it

# Consider addressing a comment to other parts

Find similar code and address it

Example of comment "Add `@Nullable` to this parameter":

- Apply to other parameters
- Apply to the return value
- Apply to other functions

# Don't just address comments

- Think why reviewer asked/misunderstood
- Understand reviewer's suggestion
- Consider addressing a comment to other parts
- **Don't make reviewers repeat**

# Don't make reviewers repeat

Check what you asked previously before sending a review request

- Coding style and conventions
- Natural language: word choice, grammar
- Language/platform idiom
- Conditional branch structure
- Testing

# Don't just address comment: Summary

Understand what is the intention of a review comment

- Find the key idea and reason of the comment
- Try to apply it to other parts
- Relieve reviewers' workload

# Topics

## Authors:

- How to create PR
- How to address review comment

## Reviewers:

- Principles
- Comment contents

# Principles for reviewers

Don't be too kind

# Principles for reviewers

Don't be too kind  
(but be kind a little)

# Principles for reviewers

- Don't neglect review requests
- Reject problematic PR
- Don't guess author's situation
- Consider other options of "answer"

# Principles for reviewers

- Don't neglect review requests
- Reject problematic PR
- Don't guess author's situation
- Consider other options of "answer"

# Don't neglect review requests

Define a reply time limit for the project

- e.g., 24 hours of working days

**Question:** What is the worst reaction to a review request?

# Reactions to review requests

GOOD:

NOT SO GOOD:

- Simply ignore

WORST:

# Reactions to review requests

GOOD:

NOT SO GOOD:

- Simply ignore

WORST:

- Reply "I'll review" without review

# Reactions to review requests

GOOD:

- Review soon
- Redirect to other reviewer
- "I cannot review", "I'll be late"...

NOT SO GOOD:

- Simply ignore

WORST:

- Reply "I'll review" without review

# When you cannot review

If you are busy, reply so

# When you cannot review

If you are busy, reply so

The author can:

- Wait for the PR review if it doesn't block other tasks
- Find other reviewers if it's an urgent PR

# Principles for reviewers

- Don't neglect review requests
- Reject problematic PR
- Don't guess author's situation
- Consider other options of "answer"

# Reject problematic PR

Don't spend too much time reviewing a problematic PR

# Reject problematic PR

Ask the author to re-create a PR without reading the detail

# Reject problematic PR

Ask the author to re-create a PR without reading the detail

Too large or hard to read:

- Ask to split

# Reject problematic PR

Ask the author to re-create a PR without reading the detail

Too large or hard to read:

- Ask to split

Totally wrong on purpose, assumption, or structure:

- Ask to make skeleton classes or have casual chat

# Principles for reviewers

- Don't neglect review requests
- Reject problematic PR
- **Don't guess author's situation**
- Consider other options of "answer"

# Don't guess author's situation

Reviewers should not care about deadline

- Let them explain if they want to merge soon
- Need to file issue tickets, and add TODO at least

# Don't guess author's situation

Reviewers should not care about deadline

- Let them explain if they want to merge soon
- Need to file issue tickets, and add TODO at least

Exception: Major/critical issues on a release branch or hot-fix

- Tests are still required
- Reviewers should keep cooler than authors

# Principles for reviewers

- Don't neglect review requests
- Reject problematic PR
- Don't guess author's situation
- Consider other options of "answer"

# Options of a review comment

The answer is not only showing answer

# Options of a review comment

The answer is not only showing answer

To save reviewer's time and encourage authors' growth:

- Ask questions to make the author come up with an idea
- Suggest options to make the author decide
- Leave a note to share knowledge

# Principles for reviewers: Summary

Don't be too kind, but help authors

- Reply if you can't review
- Fine to refuse problematic PR

# Principles for reviewers: Summary

Don't be too kind, but help authors

- Reply if you can't review
- Fine to refuse problematic PR

Think what is the best comment

- including time cost and author's growth

# Topics

## Authors:

- How to create PR
- How to address review comment

## Reviewers:

- Principles
- Comment contents

# What should be commented

- Styles, conventions, idioms ... (CI can review instead)
- Tests
- All in this presentation series
  - Principles, natural language, structure, dependency
- Any issues you found

# What should be commented

- Styles, conventions, idioms ... (CI can review instead)
- Tests
- All in this presentation series
  - Principles, natural language, structure, dependency
- Any issues you found

That's all!

# Code review case study

## Adding a parameter to function

# Code review case study

## Adding a parameter to function

```
fun showPhotoView(  
    photoData: PhotoData  
) {  
    if (!photoData.isValid) {  
  
        return  
    }  
  
    ...  
}
```

# Code review case study

## Adding a parameter to function

```
fun showPhotoView(  
    photoData: PhotoData,  
    isFromEditor: Boolean  
) {  
    if (!photoData.isValid) {  
        if (isFromEditor) { showDialog() }  
        return  
    }  
    ...  
}
```

# Code review case study

Naming: Describe what it is rather than how it is used

isFromEditor -> showsDialogOnError

# Code review case study

## Naming: Describe what it is rather than how it is used

```
fun showPhotoView(  
    photoData: PhotoData,  
    isFromEditor: Boolean  
) {  
    if (!photoData.isValid) {  
        if (isFromEditor) { showDialog() }  
        return  
    }  
    ...  
}
```

# Code review case study

## Naming: Describe what it is rather than how it is used

```
fun showPhotoView(  
    photoData: PhotoData,  
    showsDialogOnError: Boolean  
) {  
    if (!photoData.isValid) {  
        if (showsDialogOnError) { showDialog() }  
        return  
    }  
    ...  
}
```

# Code review case study

Dependency: Control coupling with a boolean flag

Extract showDialog call

# Code review case study

## Dependency: Control coupling with a boolean flag

### Extract showDialog call

- For editor:

```
val isViewShown = showPhotoView(...)  
if (!isViewShown) { showErrorDialog(...) }
```

- For other place:

```
showPhotoView(...)
```

# Code review case study

## Dependency: Control coupling with a boolean flag

```
fun showPhotoView(photoData: PhotoData): Boolean {  
    if (!photoData.isValid) {  
        return false  
    }  
}
```

...

# Code review case study

Comments:

Comment if a function has both side-effect and return value

Add comment to explain the return value

# Code review case study

## Comments:

Comment if a function has both side-effect and return value

```
/**  
 * Shows a photo at the center if the given photo data is . . . ,  
 * and returns true.  
 * Otherwise, this returns false without showing the view.  
 */  
  
fun showPhotoView(photoData: PhotoData): Boolean {  
    if (!photoData.isValid) {  
        return false  
    }  
}
```

# What we comment: Summary

Review the following items

- Styles, conventions, manners, idioms ...
- Principles, natural language, structure, dependency ...

# What we comment: Summary

Review the following items

- Styles, conventions, manners, idioms ...
- Principles, natural language, structure, dependency ...

Read over "code readability" presentation 😊

# Summary

Review is for readability

# Summary

Review is for readability

Author:

- Keep PR small and structured
- Understand review comments, don't just address

# Summary

Review is for readability

Author:

- Keep PR small and structured
- Understand review comments, don't just address

Reviewer:

- Don't be too kind, you can ask