Proactive Empirical Assessment of New Language Feature Adoption via Automated Refactoring: The Case of Java 8 Default Methods

Raffi T. Khatchadourian
CUNY Hunter College

Hidehiko Masuhara
Tokyo Institute of Technology

How does access to this work benefit you? Let us know!
Follow this and additional works at: https://academicworks.cuny.edu/hc_pubs
Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Recommended Citation

This Presentation is brought to you for free and open access by the Hunter College at CUNY Academic Works. It has been accepted for inclusion in Publications and Research by an authorized administrator of CUNY Academic Works. For more information, please contact AcademicWorks@cuny.edu.
Proactive Empirical Assessment of New Language Feature Adoption via Automated Refactoring: The Case of Java 8 Default Methods

Raffi Khatchadourian\textsuperscript{1,2} Hidehiko Masuhara\textsuperscript{3}
International Conference on the Art, Science, and Engineering of Programming April 2018, Nice, France

\textsuperscript{1}Computer Science, Hunter College, City University of New York, USA
\textsuperscript{2}Computer Science, The Graduate Center, City University of New York, USA
\textsuperscript{3}Mathematical and Computing Science, Tokyo Institute of Technology, Japan
Outline

Introduction

Background

Contributions

Methodology

Research Questions and Results

Conclusion
Introduction
New Programming Languages Features

- Programming languages change for a variety of reasons.
• Programming languages change for a variety of reasons.
• To benefit from new language features, developers must be willing to adopt them.
• An empirical study assessing the adoption of a new language feature: default methods.
• An empirical study assessing the adoption of a new language feature: default methods.
• Default methods are part of Java 8’s enhanced interfaces.
Background
Java 8 Default Methods

- Allow both method declarations and definitions.

interface Collection<E> {
    default void add(E elem) { // optional.
        throw new UnsupportedOperationException();}
}
Java 8 Default Methods

- Allow both method declarations and definitions.
- Implementers inherit the (default) implementation if none provided.

```java
interface Collection<E> {
    default void add(E elem) { // optional.
        throw new UnsupportedOperationException();}
}

class ImmutableList<E> implements Collection<E> {
}
```
Java 8 Default Methods

- Allow both method declarations and definitions.
- Implementers inherit the (default) implementation if none provided.
- Original motivation to facilitate interface evolution.

```java
interface Collection<E> {
    default void add(E elem) { // optional.
        throw new UnsupportedOperationException();}
}

class ImmutableList<E> implements Collection<E> {}
```
Java 8 Default Methods

- Allow **both** method declarations and **definitions**.
- Implementers inherit the **default** implementation if none provided.
- Original motivation to facilitate interface **evolution**.
- Can also be used as a replacement of the **skeletal implementation pattern** (Goetz 2011).

```java
interface Collection<E> {
    default void add(E elem) { // optional.
        throw new UnsupportedOperationException();}
}

class ImmutableList<E> implements Collection<E> {}

abstract class AbstractImmutableList<E> implements Collection<E> {
    @Override public void add(E elem) {
        throw new UnsupportedOperationException();}
}
Java 8 Default Methods

- Allow both method declarations and definitions.
- Implementers inherit the (default) implementation if none provided.
- Original motivation to facilitate interface evolution.
- Can also be used as a replacement of the skeletal implementation pattern (Goetz 2011).
  - Uses abstract class that interface implementers extend.

```java
interface Collection<E> {
    default void add(E elem) { // optional.
        throw new UnsupportedOperationException();}
}

class ImmutableList<E> implements Collection<E> {}

abstract class AbstractImmutableList<E> implements Collection<E> {
    @Override public void add(E elem) {
        throw new UnsupportedOperationException();}
}
```
Java 8 Default Methods

- Allow both method declarations and definitions.
- Implementers inherit the (default) implementation if none provided.
- Original motivation to facilitate interface evolution.
- Can also be used as a replacement of the skeletal implementation pattern (Goetz 2011).
  - Uses abstract class that interface implementers extend.
  - Makes interfaces easier to implement (Bloch 2008, Item 18).

```java
interface Collection<E> {
    default void add(E elem) { // optional.
        throw new UnsupportedOperationException();
    }
}

class ImmutableList<E> implements Collection<E> {}

abstract class AbstractImmutableList<E> implements Collection<E> {
    @Override public void add(E elem) {
        throw new UnsupportedOperationException();
    }
}
```
Contributions
Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.
Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.
- Pull requests (patches) issued that contained particular interface method implementations migrated to interfaces as default methods in a semantics-preserving fashion.
- Found that there are non-obvious trade-offs to using default methods.
- Detailed reactions of developers in adopting default methods in their projects.
- Extracted best practices of their uses.
- Situations where these new constructs work well and where trade-offs must be made.
Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.
- Pull requests (patches) issued that contained particular interface method implementations migrated to interfaces as default methods in a semantics-preserving fashion.
- Found that there are non-obvious trade-offs to using default methods.
Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.
- Pull requests (patches) issued that contained particular interface method implementations migrated to interfaces as default methods in a semantics-preserving fashion.
- Found that there are non-obvious trade-offs to using default methods.
- Detail reactions of developers in adopting default methods in their projects.
Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.
- Pull requests (patches) issued that contained particular interface method implementations migrated to interfaces as default methods in a semantics-preserving fashion.
- Found that there are non-obvious trade-offs to using default methods.
- Detail reactions of developers in adopting default methods in their projects.
- Extract best practices of their uses.
Our Study

- Performed empirical study on 19 real-world, open source Java projects hosted on GitHub.
- Pull requests (patches) issued that contained particular interface method implementations migrated to interfaces as default methods in a semantics-preserving fashion.
- Found that there are non-obvious trade-offs to using default methods.
- Detail reactions of developers in adopting default methods in their projects.
- Extract best practices of their uses.
- Situations where these new constructs work well and where trade-offs must be made.
A popular approach for assessing language features involves a *postmortem* analysis.
Traditional Approaches to Assessing New Languages Features

- A popular approach for assessing language features involves a *postmortem* analysis.
- *Past* data of source repositories are analyzed.
A popular approach for assessing language features involves a *postmortem* analysis.

Past data of source repositories are analyzed.

Surveys of previous coding activities are taken.
Traditional Approaches to Assessing New Languages Features

- A popular approach for assessing language features involves a *postmortem* analysis.
- *Past* data of source repositories are analyzed.
- Surveys of previous coding activities are taken.
- Developers must discover new language features and integrate them themselves before any analysis of the construct can be done.

Best practices and patterns that can normally be extracted from these studies are delayed. Developers may be unable to manually identify all opportunities where the new language construct can be utilized. Observing software histories may discover cases where new language features are adopted but may not easily identify those where they were rejected as these may not have been adequately documented.
A popular approach for assessing language features involves a *postmortem* analysis.

*Past* data of source repositories are analyzed

Surveys of previous coding activities are taken.

Developers must discover new language features and integrate them themselves before any analysis of the construct can be done.

Best practices and patterns that can normally be extracted from these studies are delayed.
A popular approach for assessing language features involves a *postmortem* analysis.

*Past* data of source repositories are analyzed.

Surveys of previous coding activities are taken.

Developers must discover new language features and integrate them themselves before any analysis of the construct can be done.

Best practices and patterns that can normally be extracted from these studies are delayed.

Developers may be unable to *manually* identify all opportunities where the new language construct can be utilized.
A popular approach for assessing language features involves a *postmortem* analysis.

Past data of source repositories are analyzed.

Surveys of previous coding activities are taken.

Developers must discover new language features and integrate them themselves before any analysis of the construct can be done.

Best practices and patterns that can normally be extracted from these studies are delayed.

Developers may be unable to *manually* identify *all* opportunities where the new language construct can be utilized.

Observing software histories may discover cases where new language features are *adopted* but may not easily identify those where they were *rejected* as these may not have been adequately documented.
Our Proactive Approach

• A novel technique for assessing new language constructs proactively.
Our Proactive Approach

- A novel technique for assessing new language constructs proactively.
- The pull request changes in our study consist of transformations performed via an automated refactoring tool.
Our Proactive Approach

- A novel technique for assessing new language constructs proactively.
- The pull request changes in our study consist of transformations performed via an automated refactoring tool.
- Developers are immediately introduced to the new construct via a semantically equivalent transformation that they can either accept or reject.
Our Proactive Approach

- A novel technique for assessing new language constructs proactively.
- The pull request changes in our study consist of transformations performed via an automated refactoring tool.
- Developers are immediately introduced to the new construct via a semantically equivalent transformation that they can either accept or reject.
- Their decisions can be studied early to assess the feature’s effectiveness, extracting best practices.
Methodology
Study Methodology

- The use of conservative, theoretically sound, and minimally invasive refactoring automation is key in minimizing human bias.

- We use the Migrate Skeletal Implementation to Interface refactoring tool (Khatchadourian and Masuhara 2017), based on type constraints (Palsberg and Schwartzbach 1994; Tip et al. 2011).

- Discover opportunities and semantics-preserving transformations for migrating methods possibly participating in the skeletal implementation pattern to interfaces as default methods.

- Assess the use of default methods in existing code.

- Substituting the skeletal implementation pattern is the only sensible use of default methods when not introducing new functionality.

- An acceptance of the refactoring is equivalent to acceptance of using default methods as a programming construct for existing code and vice-versa.
Study Methodology

- The use of conservative, theoretically sound, and minimally invasive refactoring automation is key in minimizing human bias.
- We use the MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE refactoring tool (Khatchadourian and Masuhara 2017), based on type constraints (Palsberg and Schwartzbach 1994; Tip et al. 2011).
Study Methodology

- The use of *conservative, theoretically sound, and minimally invasive* refactoring automation is key in minimizing human bias.
- We use the MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE refactoring tool (Khatchadourian and Masuhara 2017), based on type constraints (Palsberg and Schwartzbach 1994; Tip et al. 2011).
- Discover opportunities and semantics-preserving transformations for migrating methods possibly participating in the skeletal implementation pattern to interfaces as default methods.
Study Methodology

- The use of conservative, theoretically sound, and minimally invasive refactoring automation is key in minimizing human bias.
- We use the MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE refactoring tool (Khatchadourian and Masuhara 2017), based on type constraints (Palsberg and Schwartzbach 1994; Tip et al. 2011).
- Discover opportunities and semantics-preserving transformations for migrating methods possibly participating in the skeletal implementation pattern to interfaces as default methods.
- Assess the use of default methods in existing code.
Study Methodology

- The use of conservative, theoretically sound, and minimally invasive refactoring automation is key in minimizing human bias.
- We use the MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE refactoring tool (Khatchadourian and Masuhara 2017), based on type constraints (Palsberg and Schwartzbach 1994; Tip et al. 2011).
- Discover opportunities and semantics-preserving transformations for migrating methods possibly participating in the skeletal implementation pattern to interfaces as default methods.
- Assess the use of default methods in existing code.
- Substituting the skeletal implementation pattern is the only sensible use of default methods when not introducing new functionality.
Study Methodology

• The use of conservative, theoretically sound, and minimally invasive refactoring automation is key in minimizing human bias.

• We use the MIGRATE SKELETAL IMPLEMENTATION TO INTERFACE refactoring tool (Khatchadourian and Masuhara 2017), based on type constraints (Palsberg and Schwartzbach 1994; Tip et al. 2011).

• Discover opportunities and semantics-preserving transformations for migrating methods possibly participating in the skeletal implementation pattern to interfaces as default methods.

• Assess the use of default methods in existing code.

• Substituting the skeletal implementation pattern is the only sensible use of default methods when not introducing new functionality.

• An acceptance of the refactoring is equivalent to acceptance of using default methods as a programming construct for existing code and vice-versa.
Table 1: Pull requests. More info at http://cuny.is/interefact.
Question
In which situations do developers adopt default methods in their projects? What are the reasons?
Question
In which situations do developers adopt default methods in their projects? What are the reasons?

Answers

Interface Locality Default implementation was mostly in terms of both methods and constant fields declared either within the same interface or one up its hierarchy.
Default Method Adoption

Question
In which situations do developers adopt default methods in their projects? What are the reasons?

Answers

Interface Locality  Default implementation was mostly in terms of both methods and constant fields declared either within the same interface or one up its hierarchy.

Parameter Locality  No new dependencies introduced by the default method by referencing only parameters.
Default Method Adoption

**Question**
In which situations do developers adopt default methods in their projects? What are the reasons?

**Answers**

**Interface Locality** Default implementation was mostly in terms of both methods and constant fields declared either within the same interface or one up its hierarchy.

**Parameter Locality** No new dependencies introduced by the default method by referencing only parameters.

**Optional Methods** Default implementation threw `UnsupportedOperationException` (self-documenting).
Default Method Adoption

Question
In which situations do developers adopt default methods in their projects? What are the reasons?

Answers

**Interface Locality**  Default implementation was mostly in terms of both methods and constant fields declared either within the same interface or one up its hierarchy.

**Parameter Locality**  No new dependencies introduced by the default method by referencing only parameters.

**Optional Methods**  Default implementation threw `UnsupportedOperationException` (self-documenting).

**Static Methods as Instance Methods**  Allowed static methods to be called as instance methods via forwarding.
Question
Are there situations where developers do not favor default methods?
Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

JDK Versions • Needed to maintain compatibility with legacy clients (e.g., Android).
Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

JDK Versions

• Needed to maintain compatibility with legacy clients (e.g., Android).
• Developers must not only consider the language construct itself but also substantial reliance on platform backwards compatibility.
Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

JDK Versions

• Needed to maintain compatibility with legacy clients (e.g., Android).
• Developers must not only consider the language construct itself but also substantial reliance on platform backwards compatibility.
Question
Are there situations where developers do not favor default methods?

Answers

JDK Versions
- Needed to maintain compatibility with legacy clients (e.g., Android).
- Developers must not only consider the language construct itself but also substantial reliance on platform backwards compatibility.

Architecture
- Developers did not always want to introduce new external dependencies into interfaces as some default methods required.
Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

JDK Versions
- Needed to maintain compatibility with legacy clients (e.g., Android).
- Developers must not only consider the language construct itself but also substantial reliance on platform backwards compatibility.

Architecture
- Developers did not always want to introduce new external dependencies into interfaces as some default methods required.
- Projects separated their APIs (interfaces) and an implementation of that API into separate modules.
Default Method Rejection

Question
Are there situations where developers do not favor default methods?
Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

Clients
- Anxious about “inlining” skeletal implementations directly into interfaces, particular frameworks.
Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

**Clients**
- Anxious about “inlining” skeletal implementations directly into interfaces, particular frameworks.
- Desired **forcing** clients to implement interfaces directly despite providing skeletal implementations in a separate classes.
Default Method Rejection

**Question**
Are there situations where developers do not favor default methods?

**Answers**

**Clients**
- Anxious about “inlining” skeletal implementations directly into interfaces, particular frameworks.
- Desired **forcing** clients to implement interfaces directly despite providing skeletal implementations in a separate classes.
Question
Are there situations where developers do not favor default methods?

Answers

Clients
- Anxious about “inlining” skeletal implementations directly into interfaces, particular frameworks.
- Desired **forcing** clients to implement interfaces directly despite **providing** skeletal implementations in a separate classes.

Generality
- Skeletal implementations too **narrow** to be the “de facto.”
Default Method Rejection

Question
Are there situations where developers do not favor default methods?

Answers

Clients
• Anxious about “inlining” skeletal implementations directly into interfaces, particular frameworks.
• Desired **forcing** clients to implement interfaces directly **despite providing** skeletal implementations in a separate classes.

Generality
• Skeletal implementations too *narrow* to be the “de facto.”
• Pattern allows for multiple implementations per method, enhanced interfaces **do not**.
Question
Are there situations where developers do not favor default methods?

Answers

Clients
• Anxious about “inlining” skeletal implementations directly into interfaces, particular frameworks.
• Desired forcing clients to implement interfaces directly despite providing skeletal implementations in a separate classes.

Generality
• Skeletal implementations too narrow to be the “de facto.”
• Pattern allows for multiple implementations per method, enhanced interfaces do not.
• Skeletal implementations from tests were too specific.
Default Method Trade-offs

**Question**
What are the *trade-offs* of using default methods over the skeletal implementation pattern?
Default Method Trade-offs

Question
What are the trade-offs of using default methods over the skeletal implementation pattern?

Answers

**Control**
- Contrary to pattern, default methods are available to *all* interface implementers.
Default Method Trade-offs

Question
What are the trade-offs of using default methods over the skeletal implementation pattern?

Answers

Control
· Contrary to pattern, default methods are available to all interface implementers.
· Explicitly presents implementers with a skeletal implementation.
Default Method Trade-offs

Question
What are the trade-offs of using default methods over the skeletal implementation pattern?

Answers

Control

• Contrary to pattern, default methods are available to all interface implementers.
• Explicitly presents implementers with a skeletal implementation.
• Implementers may or may not choose to override with their own.
Default Method Trade-offs

Question
What are the trade-offs of using default methods over the skeletal implementation pattern?

Answers

Control
- Contrary to pattern, default methods are available to all interface implementers.
- Explicitly presents implementers with a skeletal implementation.
- Implementers may or may not choose to override with their own.
- May have a negative effect if not applicable to implementer but choose not to override.
External Factors

**Question**
Which external factors, if any, influence developer’s decisions in adopting default methods?
External Factors

Question
Which external factors, if any, influence developer’s decisions in adopting default methods?

Answers

Java 8 Projects that previously used (other) Java 8 features were more likely to accept.
External Factors

Question
Which external factors, if any, influence developer’s decisions in adopting default methods?

Answers

Java 8  Projects that *previously* used (other) Java 8 features were more likely to accept.

Size  Smaller change sets were **more** likely to be accepted.
External Factors

**Question**
Which external factors, if any, influence developer’s decisions in adopting default methods?

**Answers**

- **Java 8** Projects that *previously* used (other) Java 8 features were more likely to accept.

- **Size** Smaller change sets were *more* likely to be accepted.

- **Span** Change sets spanning *multiple files across module boundaries* were *less* likely.
Question
Which external factors, if any, influence developer’s decisions in adopting default methods?

Answers

Java 8  Projects that previously used (other) Java 8 features were more likely to accept.

Size  Smaller change sets were more likely to be accepted.

Span  Change sets spanning multiple files across module boundaries were less likely.

Abstractness  Implementations originating from abstract classes were more likely (more general).
Question
Are there best practices and/or patterns that can be extracted from these situations?

• Default methods should be simple.
  - Reduces likelihood of complex dependencies in interfaces.
  - Promote self-containment.
  - Enhancement to the interface documentation.

• What optional methods do when called if they are not implemented?
  - Take care in using default methods for new methods that interface implementers should override.
  - May inadvertently mask interface evolution if the developers’ intention is to break existing implementers.
Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Default methods should be simple.
Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

• Default methods should be simple.
  • Reduces likelihood of complex dependencies in interfaces.
Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Default methods should be *simple*.
  - Reduces likelihood of complex dependencies in interfaces.
  - Promote self-containment.
Best Practices for Default Methods

**Question**
Are there best practices and/or patterns that can be extracted from these situations?

**Answers**

- Default methods should be *simple*.
  - Reduces likelihood of complex dependencies in interfaces.
  - Promote self-containment.
  - Enhancement to the interface documentation.
Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Default methods should be *simple*.
  - Reduces likelihood of complex dependencies in interfaces.
  - Promote self-containment.
  - Enhancement to the interface documentation.
    - What optional methods do when called if they are not implemented?
Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

• Default methods should be simple.
  • Reduces likelihood of complex dependencies in interfaces.
  • Promote self-containment.
  • Enhancement to the interface documentation.
    • What optional methods do when called if they are not implemented?

• Take care in using default methods for new methods that interface implementers should override.
Best Practices for Default Methods

**Question**
Are there best practices and/or patterns that can be extracted from these situations?

**Answers**

- Default methods should be *simple*.
  - Reduces likelihood of complex dependencies in interfaces.
  - Promote self-containment.
  - Enhancement to the interface documentation.
    - What optional methods do when called if they are not implemented?
- Take care in using default methods for *new* methods that interface implementers should override.
  - May *inadvertently mask* interface evolution if the developers’ intention is to break *existing* implementers.
Best Practices for Default Methods

**Question**
Are there best practices and/or patterns that can be extracted from these situations?

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.
  - More self-contained.
  - Reduces external dependencies.
- Consider architectural implications.
  - Rethink separating interface declarations and interface implementations into separate modules.
- Default methods may contain references to implementation modules.
  - Typically not available to interface modules.
Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.
Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.
  - More self-contained.
Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.
  - More self-contained.
  - Reduces external dependencies.
Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

• Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  • Simplifies default method implementations.
  • More self-contained.
  • Reduces external dependencies.

• Consider architectural implications.
Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

• Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  • Simplifies default method implementations.
  • More self-contained.
  • Reduces external dependencies.

• Consider architectural implications.
  • Rethink separating interface declarations and interface implementations into separate modules.
Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  - Simplifies default method implementations.
  - More self-contained.
  - Reduces external dependencies.
- Consider architectural implications.
  - Rethink separating interface declarations and interface implementations into separate modules.
  - Default methods may contain references to implementation modules.
Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

• Write default methods in terms of (other) methods and constants of the same or closely related interfaces and/or their parameters.
  • Simplifies default method implementations.
  • More self-contained.
  • Reduces external dependencies.

• Consider architectural implications.
  • Rethink separating interface declarations and interface implementations into separate modules.
  • Default methods may contain references to implementation modules.
  • Typically not available to interface modules.
Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?
**Question**
Are there best practices and/or patterns that can be extracted from these situations?

**Answers**

- Call forwarding for *deprecated* interface methods.
Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Call forwarding for deprecated interface methods.
  - Forward to replacement API, if applicable.
Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Call forwarding for deprecated interface methods.
  - Forward to replacement API, if applicable.
  - Self-documenting.
Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

• Call forwarding for deprecated interface methods.
  • Forward to replacement API, if applicable.
  • Self-documenting.
  • Eliminates any confusion over deprecation between interface and skeletal implementation class.
Best Practices for Default Methods

**Question**
Are there best practices and/or patterns that can be extracted from these situations?

**Answers**

- Call forwarding for **deprecated** interface methods.
  - Forward to **replacement** API, if applicable.
  - Self-documenting.
  - Eliminates any confusion over deprecation between interface and skeletal implementation class.
- Choose **general** default implementations.
Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

• Call forwarding for deprecated interface methods.
  • Forward to replacement API, if applicable.
  • Self-documenting.
  •Eliminates any confusion over deprecation between interface and skeletal implementation class.

• Choose general default implementations.
  • General enough for all potential implementers.
Best Practices for Default Methods

Question
Are there best practices and/or patterns that can be extracted from these situations?

Answers

- Call forwarding for deprecated interface methods.
  - Forward to replacement API, if applicable.
  - Self-documenting.
  - Eliminates any confusion over deprecation between interface and skeletal implementation class.

- Choose general default implementations.
  - General enough for all potential implementers.
  - If too narrow, use skeletal implementation pattern instead.
Conclusion
Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.

Developers provide insight into their decisions. Facilitates reasons why new features are not adopted. May not be explicitly documented. Can possibly allude traditional postmortem approaches. Experienced project committers provide valuable feedback. Approach was applied to 19 open source projects to assess Java 8 default methods. Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth. Best practices extracted. Can benefit developers and language designers, especially those considering similar constructs for other languages. More info at http://cuny.is/interefact.
Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
- May not be explicitly documented.
- Can possibly allude traditional postmortem approaches.
- Experienced project committers provide valuable feedback.
- Approach was applied to 19 open source projects to assess Java 8 default methods.
- Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.
- Best practices extracted.
- Can benefit developers and language designers, especially those considering similar constructs for other languages.

More info at [http://cuny.is/interefact](http://cuny.is/interefact).
Summary

• Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
• New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
• Developers provide insight into their decisions.
• Facilitates reasons why new features are not adopted.
• May not be explicitly documented.
• Can possibly allude traditional postmortem approaches.
• Experienced project committers provide valuable feedback.
• Approach was applied to 19 open source projects to assess Java 8 default methods.
• Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.
• Best practices extracted.
• Can benefit developers and language designers, especially those considering similar constructs for other languages.
• More info at http://cuny.is/interefact.
Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
- May not be explicitly documented.
- Can possibly allude traditional postmortem approaches.
- Experienced project committers provide valuable feedback.
- Approach was applied to 19 open source projects to assess Java 8 default methods.
- Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.
- Best practices extracted.
- Can benefit developers and language designers, especially those considering similar constructs for other languages.

More info at [http://cuny.is/interefact](http://cuny.is/interefact).
Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
- May possibly allude traditional postmortem approaches.
- Experienced project committers provide valuable feedback.
- Approach was applied to 19 open source projects to assess Java 8 default methods.
- Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.
- Best practices extracted.
- Can benefit developers and language designers, especially those considering similar constructs for other languages.
- More info at http://cuny.is/interefact.
Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
  - Can possibly allude traditional postmortem approaches.
Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
  - Can possibly allude traditional postmortem approaches.
- Experienced project committers provide valuable feedback.
• Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
• New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
• Developers provide insight into their decisions.
• Facilitates reasons why new features are not adopted.
  • May not be explicitly documented.
  • Can possibly allude traditional postmortem approaches.
• Experienced project committers provide valuable feedback.
• Approach was applied to 19 open source projects to assess Java 8 default methods.
Summary

• Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
• New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
• Developers provide insight into their decisions.
• Facilitates reasons why new features are not adopted.
  • May not be explicitly documented.
  • Can possibly allude traditional postmortem approaches.
• Experienced project committers provide valuable feedback.
• Approach was applied to 19 open source projects to assess Java 8 default methods.
• Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.

More info at http://cuny.is/interefact.
Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
  - Can possibly allude traditional postmortem approaches.
- Experienced project committers provide valuable feedback.
- Approach was applied to 19 open source projects to assess Java 8 default methods.
- Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.
- Best practices extracted.
• Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
• New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
• Developers provide insight into their decisions.
• Facilitates reasons why new features are not adopted.
  • May not be explicitly documented.
  • Can possibly allude traditional *postmortem* approaches.
• *Experienced* project committers provide valuable feedback.
• Approach was applied to 19 open source projects to assess Java 8 default methods.
• Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.
• Best practices extracted.
• Can benefit developers and language designers, especially those considering similar constructs for other languages.
Summary

- Novel proactive approach, using automated refactoring, to empirically assess new programming language features early.
- New construct introduced to developers as refactorings that they decide whether to incorporate regardless of experience.
- Developers provide insight into their decisions.
- Facilitates reasons why new features are not adopted.
  - May not be explicitly documented.
  - Can possibly allude traditional postmortem approaches.
- Experienced project committers provide valuable feedback.
- Approach was applied to 19 open source projects to assess Java 8 default methods.
- Scenarios where and reasons why default method migrations were either accepted or rejected by developers were put forth.
- Best practices extracted.
- Can benefit developers and language designers, especially those considering similar constructs for other languages.
- More info at http://cuny.is/interefact.
For Further Reading

- Tip, Frank et al. (May 2011). “Refactoring Using Type Constraints”. In: *ACM Transactions on Programming Languages and Systems* 33.3, pp. 91–947. ISSN: 0164-0925. DOI: 10.1145/1961204.1961205.