Security Concern Refactoring: Increasing and Assessing the Security Level of Existing Software

Katsuhisa Maruyama
Department of Computer Science
Ritsumeikan University
maru@cs.ritsumei.ac.jp

Background

Software security is ever-increasingly becoming a serious issue since software comes to play an essential role in the real world and our lives much depend on software (Viega and McGraw, 2001)

Nevertheless,

- There are an enormous number of vulnerable software programs with unnecessary privileges, known flaws, or weak access control settings on resources
- Security characteristics of well-designed software programs will deteriorate due to recurring modifications
- Security vulnerabilities are easily inserted while on software modification

Approaches to Software Security

To remove risks posed by attacks which try to exploit security vulnerabilities:

- Secure software will be created from scratch
  - ✓ Hard to in advance design and implement secure software
- Existing software will be directly changed into secure one by partially modifying its design or code
  - ✓ Not perfectly remove security vulnerability from a large number of running programs in the world
  - ✓ Can decreases the likelihood of threats
- Design or code of existing software will be improved without assessing its security level
  - ✓ Not eliminate all kinds of interfusion of vulnerabilities resulting from improvement
  - ✓ Can know of decreasing of security level while improving

Security Concern Refactoring

Most modification of software affects (increases or decreases) its security characteristics

+

Refactoring is a special pattern of software modification, which changes the internal structure of existing code without changing its external behavior by applying a series of behavior-preservation transformations

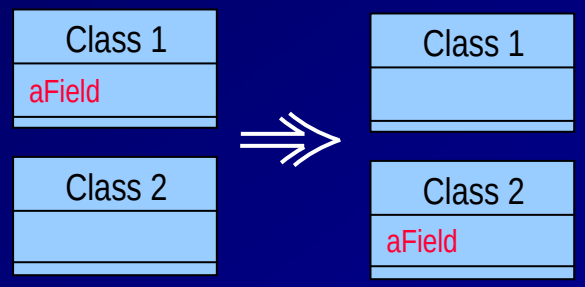
- Secure refactoring which increases the security level of existing code by its transformations
- Security-aware refactoring which assess the effects of the security level of existing code in its transformations

Remove and/or reveal inadvertent security flaws in programs that could be accidentally or intentionally exploited to damage assets

Refactoring Examples

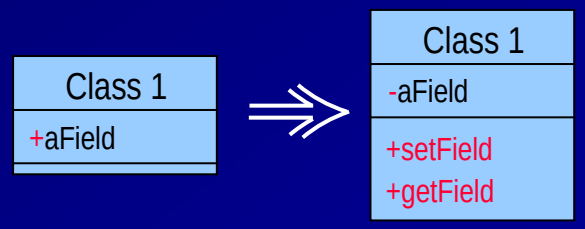
Move Filed

Moves a field in the class on which it is defined to another class that frequently uses the field



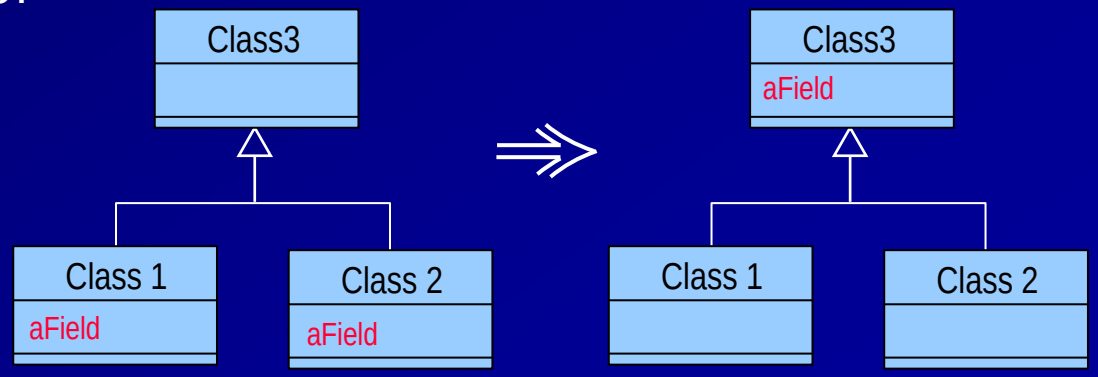
Encapsulate Field

Makes a public field private and provides its accessors



Pull Up Field

Moves the same field of sibling classes to their common superclass



Secure Refactoring

not traditional refactoring

insecure code

behavior-preservation transformations

more secure code

Directly removes vulnerable fragments from existing code

=

E.g., excessive disclosure of sensitive data
needless flexibility of code

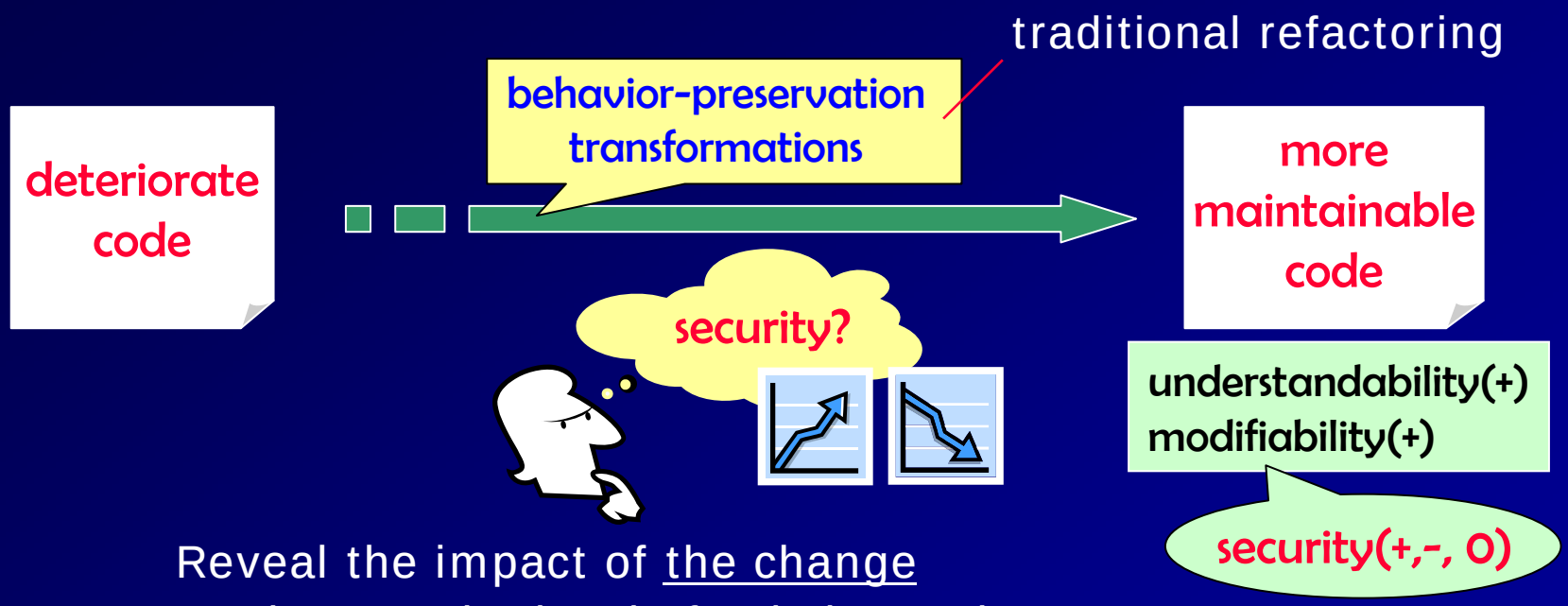
understandability(?)
modifiability(?)

security(+)

A change made to the internal structure of software
~~to make it easier to understand and cheaper to modify~~
without changing its observable behavior (Fowler, 1999)

A change made to the internal structure of software
to make it more secure without changing its observable behavior

Security-Aware Refactoring



Reveal the impact of <u>the change on the security level</u> of existing code

=

E.g., change of accessibility setting of sensitive data

A change made to the internal structure of software to make it easier to understand and cheaper to modify without both changing its observable behavior and missing unexpected changes of its security level.

Definitions (in this work)

Security = Protection of assets (e.g., sensitive information)

Classification of protection measures:

- Prevention
  - ✓ prevents assets from being damaged
- Detection
- Reaction

Deal with two of several security aspects:

- Confidentiality
  - ✓ Prevention of unauthorized disclosure of sensitive data
- Integrity
  - ✓ Prevention of unauthorized modification of sensitive data

The security level of software is measured based on the potential existence of vulnerable code (inadvertent code flaws) which might violate the confidentially and integrity

Java Security

- A Sandbox model which is responsible for protecting resources according to user-defined security policy
  - However, it cannot block violation by inadvertent code
- Four possible access levels specified by access modifiers
  - **`private`**: accessible from only inside the class defining the construct
  - **`(default)`**: accessible from any class in the same package as the class defining the construct
  - **`protected`**: accessible from the class defining the construct, classes within the same package as the defining class, or subclasses of the defining class
  - **`public`**: accessible from any class
- The final modifier so as that any program code cannot accidentally change the state of an element
  - A final variable can be set only once
  - A final field can be set only once by a constructor
  - A final method cannot be neither overridden nor hidden
  - No subclass can be derived from a final class

Secure Refactoring

Secure Refactoring Transformations

(1) Introduce Immutable Field

- ✓ Protects internal data stored in a field from modification
- ✓ The field must be declared final and its data is assigned only once in a constructor of the class defining the field

(2) Replace Reference with Copy

- ✓ Protects internal data stored in an immutable but mutable-typed field from modification
- ✓ Passes a copy of the original data instead of a reference to an instance storing the data

(3) Prohibit Overriding

- ✓ Removes excessive use of polymorphic calls by subclassing
- ✓ The method or class must be declared final so that it is not intended to be overridden or inherited

(4) Clear Sensitive Value Explicitly

- ✓ Prevents an adversary from stealing internal data stored in the memory
- ✓ The original data must be replaced with dummy as early as possible if the data will not be used in the future

Motivating Code: Prohibit Overriding (1/3)

Code before refactoring

```
// before refactoring (insecure)
class Member {
  public String getInfo(Password pw) {
    if (pw.check()) {
      return "Secret Info";
    } else {
      return "No Info";
    }
  }
}
class Password {
  private final String se
  private final String pas
  public Password(String
    passwd = pw;
  }
  public boolean check() {
    return passwd.equals(secretPasswd);
  }
}
```

```
// attacking code
class FakePassword extends Password {
  …
  public boolean check() {
    return true;
  }
}
```

```
Member member;
Password fake = new FakePassword("X");
System.out.println(member.getInfo(fake));
```

Security Smell: Prohibit Overriding (2/3)

Target class *C*

Instance *i* which is given by external code

```
class Member {
  public String getInfo(Password pw) {
    if (pw.check()) {
      …
```

Call to a method *m* of a class *Ci* for *i*

Security smell:
In a target class *C*, there is a call to a method *m* of a class *Ci* for an instance *i* which is given by external (possibly malicious) code, and any class can be derived from *Ci*

Situation in which the refactoring should be applied

Concise Procedure: Prohibit Overriding (3/3)

1. Find all subclasses of *Ci* and check to see if respective methods of the subclasses override *m*.
   ⇒ *If overriding methods exist, consider applying the* "**Replace Inheritance with Delegation** (352)" to replace them with delegation methods. *If some of them remain, cancel this refactoring.*
2. Add the keyword **final** to *m* in its declaration. If there is no subclass of *Ci*, make *Ci* final.

```
// after refactoring (more secure)
final class Password {
  private final String secretPasswd = "SECRET";
  private final String passwd;
  public Password(String pw) { … }
  public final boolean check() {
    return passwd.equals(secretPasswd);
  }
}
```

Class *Ci*

Method *m*

Code after refactoring

Security Smell: Clear Sensitive Value (1/4)

Target class *C*

Local variable (parameter) *v*

```
// before refactoring (insecure)
class Member {
  public void printMessage(int score) {
    if (score < 60) {
      print(score);
      printMessage();
    }
  }
}
```

Code before refactoring

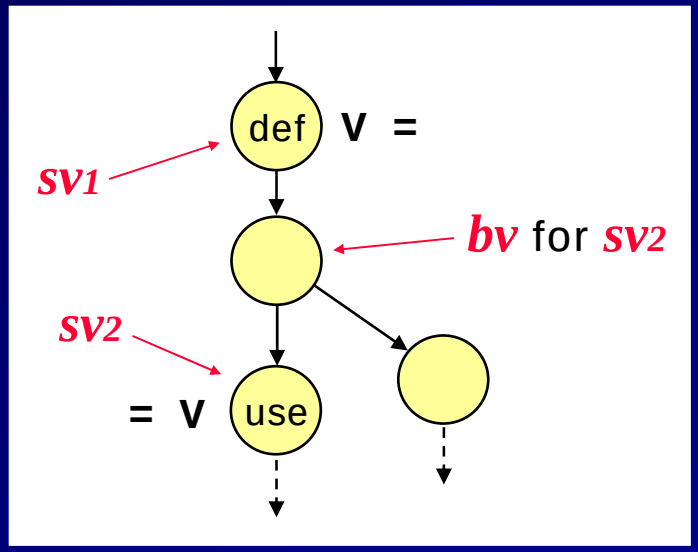
The value of score will not be used in the future but still remains in the memory

A stealer might be able to observe the value of score through memory

Security smell:
In a target class *C*, there is a method *m* which keeps the sensitive value of a local variable *v* until the execution of *m* is terminated

Concise Procedure: Clear Sensitive Value (2/4)

1. Check to see if the value of *v* is stored in an immutable instance.
   ⇒ *If an immutable instance is used, replace it with the use of a mutable instance.*
2. Find final-use statements for *v* by using control flow and data flow analysis.
3. Insert new statements that update each value of *v* with the dummy value.



sv: a statement either writing or reading the value of *v*

bv: a conditional statement a branch of which includes *sv* (*bv* dangles *sv*)

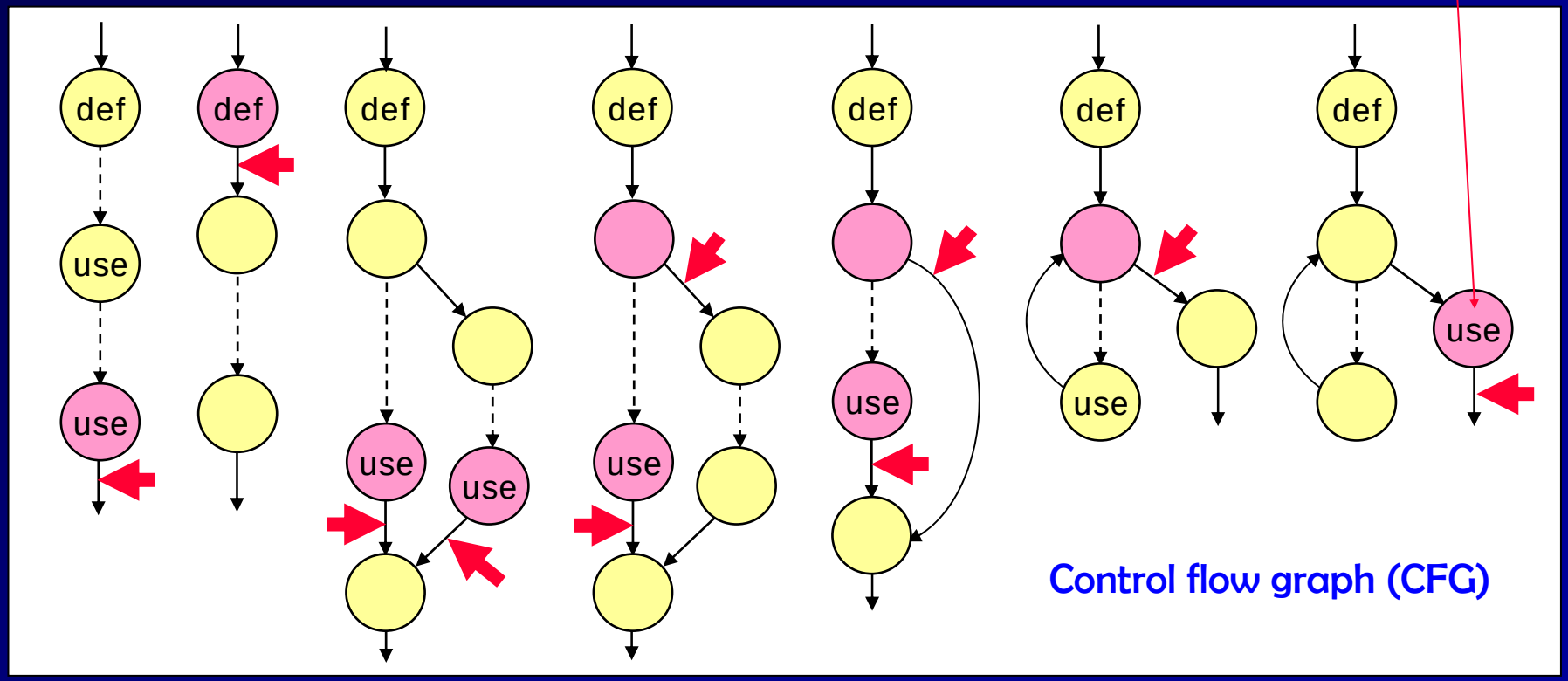
P(nv): a reachable path from a node which is one of nodes executed immediately after *nv* (= *sv* or *bv*) on control flow graph (CFG)

Final-Use Nodes: Clear Sensitive Value (3/4)

nv is defined as a final –use node for *v* if there is at least one P(nv) which does not include any nodes which correspond to statements using (writing or reading) the value of v.

Roughly, All reachable nodes from *nv* do not use *v*.

Final–use node



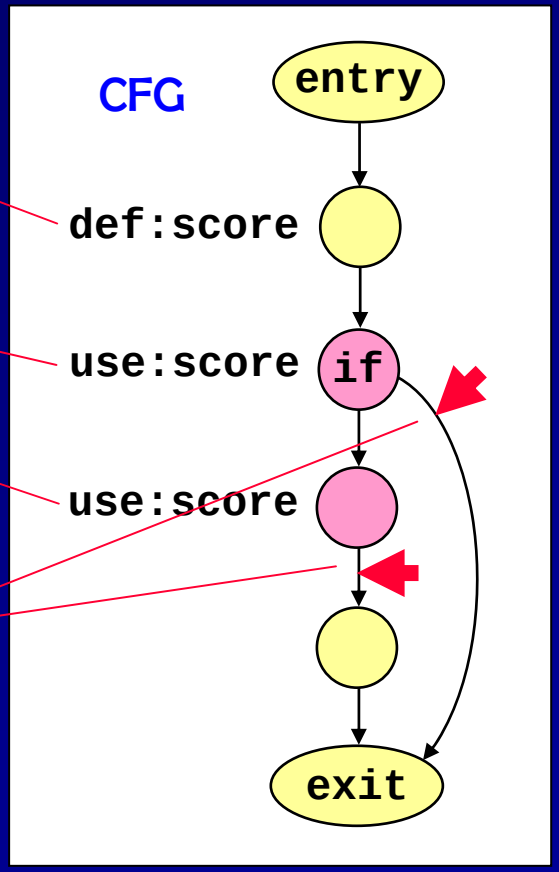
Final-Use Nodes: Clear Sensitive Value (4/4)

```
// before refactoring (insecure)
class Member {
  public void printMessage(int score) {
    if (score < 60) {
      print(score);
      printMessage();
    }
  }
```

```
// after refactoring (more secure)
class Member {
  public void printMessage(int score) {
    if (score < 60) {
      print(score);
      score = -1;
      printMessage();
    } else {
      score = -1;
    }
  }
}
```

Insert a statement clearing the value of score

The value of score cannot be observed



Secure-Aware Refactoring

Criteria for Security-Aware Refactoring

(1) Accessibility settings specified in code

- Lattice of access settings for classes, methods, and fields

(2) Location where internal data is stored or observed

- Lattice of residence for fields, local variables, parameters, and return values

(3) Possibility of Subclassing

- Can we create subclasses or overridden methods?

```
public class C {
  private int f;
  public int m(int p) {
    int v;
  }
}
class B extends C {
}
```

Class

Field

Return value

Parameter

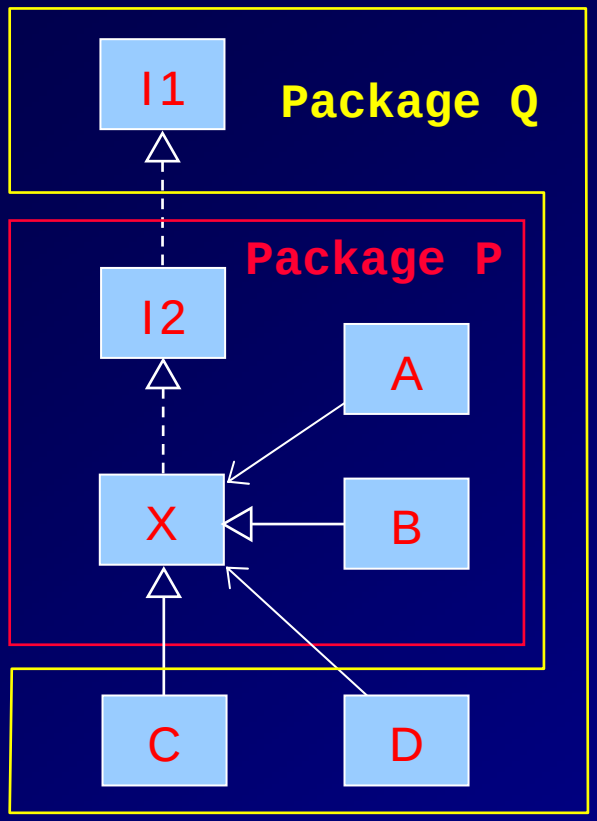
Local variable

Subclass

Access modifiers

Criterion of Access Level

Class diagram



X: target class
A, B, C, D: related class
I1, I2: interface

High

(*, private)	{ X }
↑	
(*, default)	{ X, A, B }
↑	
(default, protected)	{ X, A, B, (D) } (through I1)
↑	
(default, public)	{ X, A, B, (D) } (through I1/I2)
↑	
(public, protected)	{ X, A, B, C, (D) } (through I1)
↑	
(public, public)	{ X, A, B, C, D }

Low

Settings for a class

Settings for a field/method

Criterion of Locality

High

(local variable)

↑

(parameter)

↑

(return value)

↑

(field)

↑

(static return value)

↑

(static field)

Low

Local variable exists only inside the block

Parameter can be used in the called method and its overriding methods

Return value

Field

Return value of static method call

Static field

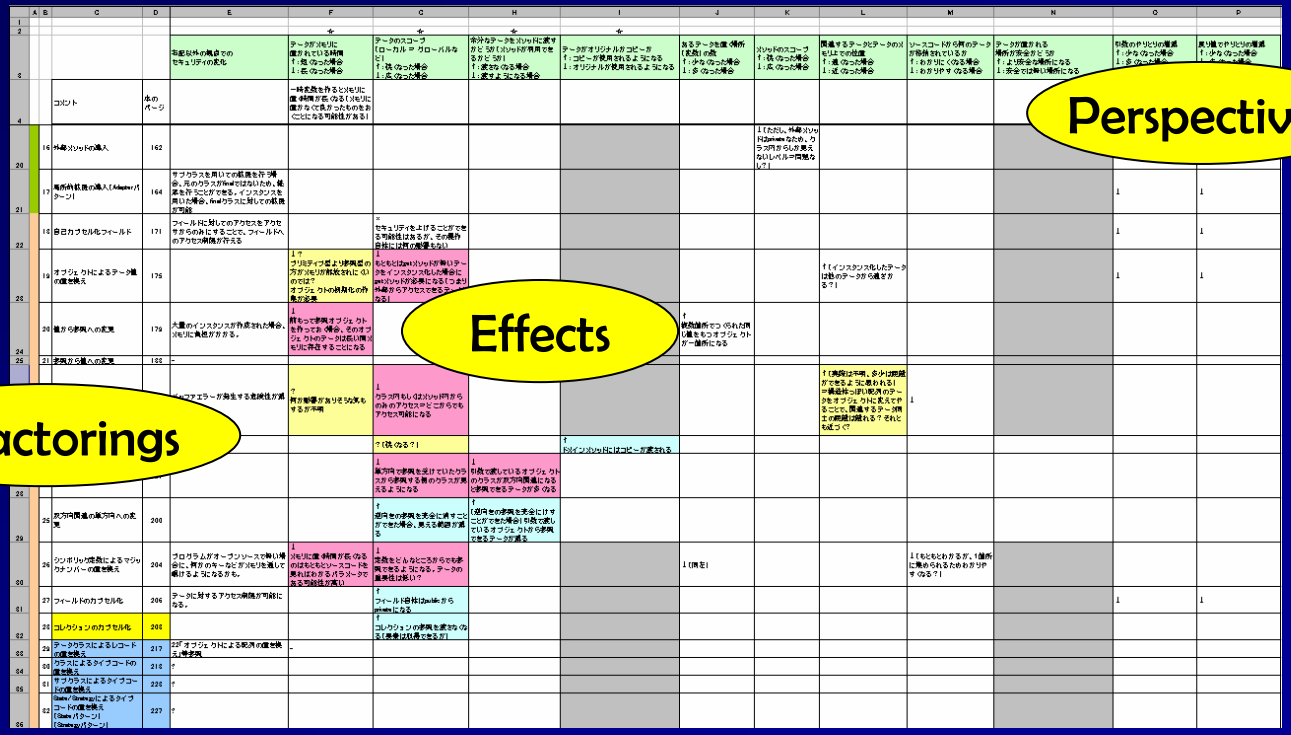
Accessible from code inside the class defining each member

Need to prepare suitable parameters to make method call a success
→ return value > field

Static members can be accessible by using class and member names
→ instance member > static one

Preliminary Study

- Used Fowler's catalog (Fowler, 1999)
- Examined transformations of all the 72 refactorings
- Operations of the transformations were classified into 7 primitive modifications:
  Create/Delete/Move/Extract/Inline/Encapsulate/Attribute change



						↓	↓	↓		↓						
					知識以外の観点での セキュリティの変化	データがメモリに 復元されている時間 ↑：長くなった場合 ↓：短くなった場合	データのスコープ （ローカル → グローバル） など！ ↑：長くなった場合 ↓：短くなった場合	依存カテゴリ（メソッドに変更 かどうかなどメソッドが利用できる かどうか） ↑：変わらなくなる場合 ↓：変わるようになる場合	データがオリジナルかコピーか ↑：コピーが使用されるようになる ↓：オリジナルが使用されるようになる	対象データを復元（解析 （復元）の点 ↑：少なくなった場合 ↓：多くなった場合	メソッドのスコープ ↑：長くなった場合 ↓：短くなった場合	関連するデータとデータの メモリ上での位置 ↑：遠くなった場合 ↓：近くなった場合	ソースコードから何のデータ が移動されているか ↑：わかりにくくなる場合 ↓：わかりやすくなる場合	データが置かれる 場所が安全かどうか ↑：より安全な場所になる ↓：安全では無い場所になる	引数のやりとりの簡潔 ↑：少なくなった場合 ↓：多くなった場合	戻り値でやりとりの簡潔 ↑：少なくなった場合 ↓：多くなった場合
		コメント	本のページ			一時変数を作るとメモリに復元時間が長くなる（メモリに復元するべき長さのものもあ ることになる可能性がある）										
20	16	外部メソッドの導入	162								↓（ただし、外部メソッドはpriveteなため、クラス内からしか使えないしレベル一問題なし？）					
21	17	場所依存機能の導入（Adapterパターン）	164		サブクラスを用いての機能を作る場合、元のクラスがfinalではないため、機能を作ることができる。インスタンスを用いた場合、finalクラスに対しての機能が可能										↓	↓
22	18	参照カプセル化フィールド	171		フィールドに対してのアクセスをアクセサからのみにすることで、フィールドへのアクセス制限が行える		＝ セキュリティをよけることができる可能性はあるが、その操作目的には既の配慮もない								↓	↓
23	19	オブジェクトによるデータ値の置き換え	175			↓？ プリミティブ型より参照型のほうがメモリが節約できたのでは？ オブジェクトの具体化の弊害が必要	↓ もともとはpriメソッドが無いデータをインスタンス化した場合にpubメソッドが必要になる（つまり外部からアクセスできるようになる）					↑（インスタンス化したデータは他のデータから遠ざかる？）			↓	↓
24	20	値から参照への変更	179		大量のインスタンスが作成された場合、メモリに負担がかかる。	↓ 新しいって参照オブジェクトを作ったら（場合、そのオブジェクトのデータは古い情報メモリに存在することになる				↑ 依存箇所でつくられた同じ値をもつオブジェクトが一箇所になる						
25	21	参照から値への変更	186													
26					ランタイムエラーが発生する危険性が高	？ 何が影響がありそうな気もするが不明	↓ クラス内もしくはメソッド内からのみのアクセスやどこからでもアクセス可能になる					↑（時間は不明、多少は問題がでるきょうに思われる！ ＝結局使うほう関係内のデータをオブジェクトに変えてやることで、関連するデータ間 エの問題は減れる？それとも広 ぐ？）	↓			
27							？（残（なる？！		↑ ドメインメソッドにはコピーが渡される							
28							↓ 最初に参照を受けていたクラスが参照する側のクラスが異なるようになる	↓ 参照で渡しているオブジェクトのクラスが変数関連になるため参照できるデータが多くなる								
29	25	双方向関連の単方向への変更	200				↑ 参照先の参照を安全に済ますことができた場合、参照を制限が減る	↑ 参照先の参照を安全にけすことができた場合！参照で渡しているオブジェクトが参照できるデータが減る								
30	26	クンポリッパ変数によるマジックナンバーの置き換え	204		プログラムがオープンソースで無い場合に、何かのキーなどがメモリを減して渡けるようになるかも。	↓ メモリに復元時間が長くなるのはもともとソースコードに埋められるパラメータである可能性が高い	↓ 変数をどんなところからでも参照できるようになる。データの重要性は低い？			↓（時間）			↓（もともとわかるが、1箇所に集められるためわかりやすくなる？！）			
31	27	フィールドのカプセル化	206		データに対するアクセス制限が可能になる。		↑ フィールド名称はpubleからprivetになる。								↓	↓
32	28	コレクションのカプセル化	208				↑ コレクションの参照を変わらない（と思意は仮定できる）？									
33	29	データクラスによるレコードの置き換え	217		22がオブジェクトによる処理の置き換え等参照											
34	30	クラスによるタイプコードの置き換え	218	↑												
35	31	サブクラスによるタイプコードの置き換え	220	↑												
36	32	State/Strategyによるタイプコードの置き換え（State パターン）（Strategy パターン）	227	↑												

Perspectives

Effects

Refactorings

Findings

Readable level indicates if external code can directly or indirectly internal data (Confidentiality)

Writable level indicates if external code can directly or indirectly modify Internal data (Integrity)

Refactorings

Operations

Impact

Table 1. The effects on the security level by the application of refactorings

	Refactoring (page number in Fowler's catalog)	Readable level	Writable level	Locality	Subclassing
4	Extract Method (110) Inline Method (117) Replace Temp with Query (120) Replace Method with Method Object (135)			↘ (p_extract, r_extract) ↗ (p_delete, r_delete) ↓ (r_extract) ↓ (f_extract)	
4	Move Method (142) Move Field (146) Extract Class (149) Inline Class (154)	↘ (m_move) ↘ (f_move) ↘ (c_extract) ↑ (c_inline)	↘ (m_move) ↘ (f_move) ↘ (c_extract) ↑ (c_inline)		
5	Encapsulate Field (206) Encapsulate Collection (208) Replace Type Code with Subclass (223) Replace Type Code with State/Strategy (227) Replace Subclass with Field (232)		↑ (m_delete)	↑ (f_encapsulate) ↑ (f_encapsulate)	↓ (c_create) ↓ (c_create) ↑ (c_delete)
3	Decompose Conditional (238) Replace Conditional with Polymorphism (255) Introduce Null Object (260)	↘ (c_create)	↘ (c_create)	↘ (p_extract, r_extract)	↓ (c_create) ↓ (c_create)
7	Add Parameter (275) Remove Parameter (277) Parameterize Method (283) Preserve Whole Object (288) Introduce Parameter Object (295) Remove Setting Method (300) Hide Method (303)	↑ (m_attr_change)	↑ (m_delete) ↑ (m_attr_change)	↓ (p_create) ↑ (p_delete) ↓ (p_create) ↓ (p_create) ↓ (p_create)	
7	Pull Up Field (320) Pull Up Method (322) Push Down Method (328) Push Down Field (329) Extract Subclass (330) Extract Superclass (336) Collapse Hierarchy (344)	↓ (f_move) ↘ (m_move) ↘ (m_move) ↑ (f_move) ↘ (m_move) ↘ (f_move, m_move) ↗ (f_move, m_move)	↓ (f_move) ↘ (m_move) ↘ (m_move) ↑ (f_move) ↘ (m_move) ↘ (f_move, m_move) ↗ (f_move, m_move)		↓ (c_create) ↓ (c_create) ↑ (c_delete)

↑ : surely increase, ↓ : surely decrease, ↗ : conditionally increase, ↘ : conditionally decrease
c_ope : operation for a class, m_ope : operation for a method, f_ope : operation for a field
p_ope : operation for a parameter, r_ope : operation for a return value, l_ope : operation for a local variable

total 30

Table 1. The effects on the security level by the application of refactorings

Refactoring (page number in Fowler's catalog)	Readable level	Writable level	Locality	Subclassing
Extract Method (110) Inline Method (117) Replace Temp with Query (120) Replace Method with Method Object (135)			↘ (p_extract, r_extract) ↗ (p_delete, r_delete) ↓ (r_extract) ↓ (f_extract)	
Move Method (142) Move Field (146) Extract Class (149) Inline Class (154)	↘ (m_move) ↘ (f_move) ↘ (c_extract) ↑ (c_inline)	↘ (m_move) ↘ (f_move) ↘ (c_extract) ↑ (c_inline)		
Re				e) e) e)
Re				e) e)
Preserve Whole Object (288) Introduce Parameter Object (295) Remove Setting Method (300) Hide Method (303)	↑ (m_attr_change)	↑ (m_delete) ↑ (m_attr_change)	↓ (p_create)	
Pull Up Field (320)	↓ (f_move)	↓ (f_move)		
Pull Up Method (322) Push Down Method (328) Push Down Field (329) Extract Subclass (330) Extract Superclass (336) Collapse Hierarchy (344)	↘ (m_move) ↘ (m_move) ↑ (f_move) ↘ (m_move) ↘ (f_move, m_move) ↗ (f_move, m_move)	↘ (m_move) ↘ (m_move) ↑ (f_move) ↘ (m_move) ↘ (f_move, m_move) ↗ (f_move, m_move)		 ↓ (c_create) ↓ (c_create) ↑ (c_delete)

↑ : surely increase, ↓ : surely decrease, ↗ : conditionally increase, ↘ : conditionally decrease
c_ope : operation for a class, m_ope : operation for a method, f_ope : operation for a field
p_ope : operation for a parameter, r_ope : operation for a return value, l_ope : operation for a local variable

Pull Up Field: moves the same field appearing in sibling classes to their common superclass

Access levels of the moved field: **private** → **protected**

(The moved field might be used by the subclasses)

- ✓ Readable level: decreased
- ✓ Writable level: decreased
- ✓ Locality: no change

Table 1. The effects on the security level by the application of refactorings

Refactoring (page number in Fowler's catalog)	Readable level	Writable level	Locality	Subclassing
Extract Method (110) Inline Method (117) Replace Temp with Query (120) Replace Method with Method Object (135)			↘ (p_extract, r_extract) ↗ (p_delete, r_delete) ↓ (r_extract) ↓ (f_extract)	
Move Method (142)	↘ (m_move)	↘ (m_move)		
Re				te) te) te)
Re				te) te)
Parameterize Method (285)			↓ (p_create)	
Preserve Whole Object (288)			↓ (p_create)	
Introduce Parameter Object (295)			↓ (p_create)	
Remove Setting Method (300)		↑ (m_delete)		
Hide Method (303)	↑ (m_attr_change)	↑ (m_attr_change)		
Pull Up Field (320)	↓ (f_move)	↓ (f_move)		
Pull Up Method (322)	↘ (m_move)	↘ (m_move)		
Push Down Method (328)	↘ (m_move)	↘ (m_move)		
Push Down Field (329)	↑ (f_move)	↑ (f_move)		
Extract Subclass (330)	↘ (m_move)	↘ (m_move)		↓ (c_create)
Extract Superclass (336)	↘ (f_move, m_move)	↘ (f_move, m_move)		↓ (c_create)
Collapse Hierarchy (344)	↗ (f_move, m_move)	↗ (f_move, m_move)		↑ (c_delete)

↑ : surely increase, ↓ : surely decrease, ↗ : conditionally increase, ↘ : conditionally decrease
c_ope : operation for a class, m_ope : operation for a method, f_ope : operation for a field
p_ope : operation for a parameter, r_ope : operation for a return value, l_ope : operation for a local variable

Remove Setting Method: deletes all methods which can modify a specified field

Access levels of the deleted method: **public** → **(non-existence)**

(No code can modify the value of the field related to the deleted methods)

- ✓ Readable level: no change
- ✓ Writable level: increased
- ✓ Locality: no change

Table 1. The effects on the security level by the application of refactorings

Refactoring (page number in Fowler's catalog)	Readable level	Writable level	Locality	Subclassing
Extract Method (110)			↘ (p_extract, r_extract)	
Inline Method (117)			↗ (p_delete, r_delete)	
Replace Temp with Query (120)			↓ (r_extract)	
Replace Method with Method Object (135)			↓ (f_extract)	
Re				
Re				
Preserve Whole Object (288)			↓ (p_create)	
Introduce Parameter Object (295)			↓ (p_create)	
Remove Setting Method (300)		↑ (m_delete)		
Hide Method (303)	↑ (m_attr_change)	↑ (m_attr_change)		
Pull Up Field (320)	↓ (f_move)	↓ (f_move)		
Pull Up Method (322)	↘ (m_move)	↘ (m_move)		
Push Down Method (328)	↘ (m_move)	↘ (m_move)		
Push Down Field (329)	↑ (f_move)	↑ (f_move)		
Extract Subclass (330)	↘ (m_move)	↘ (m_move)		↓ (c_create)
Extract Superclass (336)	↘ (f_move, m_move)	↘ (f_move, m_move)		↓ (c_create)
Collapse Hierarchy (344)	↗ (f_move, m_move)	↗ (f_move, m_move)		↑ (c_delete)

↑ : surely increase, ↓ : surely decrease, ↗ : conditionally increase, ↘ : conditionally decrease
c_ope : operation for a class, m_ope : operation for a method, f_ope : operation for a field
p_ope : operation for a parameter, r_ope : operation for a return value, l_ope : operation for a local variable

Extract Method: extracts collective code fragments from an existing method and creates a new method including the extracted fragments

Locality of the variable: **local variable** → **parameter**

(A new parameter might be extracted)

Locality of the variable: **local variable** → **return value**

(A new return value might extracted)

Readable level: no change

✓ Writable level: no change

✓ Locality: conditionally decreased

Table 1. The effects on the security level by the application of refactorings

Refactoring (page number in Fowler's catalog)	Readable level	Writable level	Locality	Subclassing
Extract Method (110)			↘ (p_extract, r_extract)	
Inline Method (117)			↗ (p_delete, r_delete)	
Replace Temp with Query (120)			↓ (r_extract)	
Replace Method with Method Object (135)			↓ (f_extract)	
Move Method (142)	↘ (m_move)	↘ (m_move)		
Move Field (146)	↘ (f_move)	↘ (f_move)		
Extract Class (149)	↘ (c_extract)	↘ (c_extract)		
Inline Class (154)	↑ (c_inline)	↑ (c_inline)		
Re				e) e) e)
Re				e) e)
Remove Setting Method (300)		↑ (m_delete)		
Hide Method (303)	↑ (m_attr_change)	↑ (m_attr_change)		
Pull Up Field (320)	↓ (f_move)	↓ (f_move)		
Pull Up Method (322)	↘ (m_move)	↘ (m_move)		
Push Down Method (328)	↘ (m_move)	↘ (m_move)		
Push Down Field (329)	↑ (f_move)	↑ (f_move)		
Extract Subclass (330)	↘ (m_move)	↘ (m_move)		↓ (c_create)
Extract Superclass (336)	↘ (f_move, m_move)	↘ (f_move, m_move)		↓ (c_create)
Collapse Hierarchy (344)	↗ (f_move, m_move)	↗ (f_move, m_move)		↑ (c_delete)

↑ : surely increase, ↓ : surely decrease, ↗ : conditionally increase, ↘ : conditionally decrease
c_ope : operation for a class, m_ope : operation for a method, f_ope : operation for a field
p_ope : operation for a parameter, r_ope : operation for a return value, l_ope : operation for a local variable

Move Method: moves a specified method in the class defining it to another class that frequently uses the method

Access levels of the moved method: **default/protected** → **public**

(The moved method might be used by the source class existing in the package not including the target class)

- ✓ Readable level: conditionally decreased
- ✓ Writable level: conditionally decreased
- ✓ Locality: no change

Conclusions

- Secure refactoring directly increases the security levels of existing code
  - ✓ Easily obtain more secure programs with the same behavior as before
- Security-Aware Refactoring reveals the impact of changes of existing code on security
  - ✓ Avoid unknowingly distributing dangerous code

Remaining issues:

- Complete the implementation of the procedure of each secure refactoring
- Formalize secure refactorings so that they never alter the behavior of existing code
- Explore many or more complex criteria for measuring the effects of the application of refactorings
  - ✓ Tried to introduce the concept of information flow

Thank you for your kind attention!

Katsuhisa Maruyama
Department of Computer Science
Ritsumeikan University
maru@cs.ritsumei.ac.jp

Any question and comment?

Preliminary Experiment & Findings

Table 1. The effects on the security level by the application of refactorings

Refactoring (page number in Fowler's catalog)	Readable level	Writable level	Locality	Subclassing
Extract Method (110)			↘ (p_extract, r_extract)	
Inline Method (117)			↗ (p_delete, r_delete)	
Replace Temp with Query (120)			↓ (r_extract)	
Replace Method with Method Object (135)			↓ (f_extract)	
Move Method (142)	↘ (m_move)	↘ (m_move)		
Move Field (146)	↘ (f_move)	↘ (f_move)		
Extract Class (149)	↘ (c_extract)	↘ (c_extract)		
Inline Class (154)	↑ (c_inline)	↑ (c_inline)		
Encapsulate Field (206)			↑ (f_encapsulate)	
Encapsulate Collection (208)		↑ (m_delete)	↑ (f_encapsulate)	
Replace Type Code with Subclass (223)				↓ (c_create)
Replace Type Code with State/Strategy (227)				↓ (c_create)
Replace Subclass with Field (232)				↑ (c_delete)
Decompose Conditional (238)			↘ (p_extract, r_extract)	
Replace Conditional with Polymorphism (255)	↘ (c_create)	↘ (c_create)		↓ (c_create)
Introduce Null Object (260)				↓ (c_create)
Add Parameter (275)			↓ (p_create)	
Remove Parameter (277)			↑ (p_delete)	
Parameterize Method (283)			↓ (p_create)	
Preserve Whole Object (288)			↓ (p_create)	
Introduce Parameter Object (295)			↓ (p_create)	
Remove Setting Method (300)		↑ (m_delete)		
Hide Method (303)	↑ (m_attr_change)	↑ (m_attr_change)		
Pull Up Field (320)	↓ (f_move)	↓ (f_move)		
Pull Up Method (322)	↘ (m_move)	↘ (m_move)		
Push Down Method (328)	↘ (m_move)	↘ (m_move)		
Push Down Field (329)	↑ (f_move)	↑ (f_move)		
Extract Subclass (330)	↘ (m_move)	↘ (m_move)		↓ (c_create)
Extract Superclass (336)	↘ (f_move, m_move)	↘ (f_move, m_move)		↓ (c_create)
Collapse Hierarchy (344)	↗ (f_move, m_move)	↗ (f_move, m_move)		↑ (c_delete)

↑ : surely increase, ↓ : surely decrease, ↗ : conditionally increase, ↘ : conditionally decrease
c_ope : operation for a class, m_ope : operation for a method, f_ope : operation for a field
p_ope : operation for a parameter, r_ope : operation for a return value, l_ope : operation for a local variable