A Study of the Fragile Base Class Problem

Leonid Mikhajlov
Turku Centre for Computer Science / Åbo Akademi

Based on joint work with
Emil Sekerinski, McMaster University

A Subject of the Study

- The Syntactic Fragile Base Class Problem
  - the necessity to recompile extension and client classes when base classes are changed
- The Semantic Fragile Base Class Problem
  - object-oriented systems employing *code inheritance* as the implementation reuse mechanism along with *self-recursion* are vulnerable to this problem
  - in a large closed system analyzing the effect of certain base class revisions on the entire system is possible in theory but can be infeasible in practice
  - in an open system the fragile base class problem requires consideration during design

An Example of the Fragile Base Class Problem

$$
\begin{array}{l}
Bag \;=\; \textbf{class} \\
\quad b : bag\ of\ char \\
\quad init \mathrel{\widehat{=}} b := [\![\,]\!] \\
\quad add(\textbf{val}\ x : char) \mathrel{\widehat{=}} b := b \cup [\![x]\!] \\
\quad addAll(\textbf{val}\ bs : bag\ of\ char) \mathrel{\widehat{=}} \\
\qquad \textbf{while}\ bs \neq [\![\,]\!]\ \textbf{do} \\
\qquad\quad \textbf{begin var}\ y \mid y \in bs \cdot \\
\qquad\qquad self.add(y); \\
\qquad\qquad bs := bs - [\![y]\!] \\
\qquad\quad \textbf{end} \\
\qquad \textbf{od} \\
\quad cardinality(\textbf{res}\ r : int) \mathrel{\widehat{=}} r := |b| \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
CountingBag \;=\; \textbf{class inherits}\ Bag \\
\quad n : int \\
\quad init \mathrel{\widehat{=}} n := 0; super.init \\
\quad add(\textbf{val}\ x : char) \mathrel{\widehat{=}} n := n + 1; super.add(x) \\
\quad cardinality(\textbf{res}\ r : int) \mathrel{\widehat{=}} r := n \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
Bag' \;=\; \textbf{class} \\
\quad b : bag\ of\ char \\
\quad init \mathrel{\widehat{=}} b := [\![\,]\!] \\
\quad add(\textbf{val}\ x : char) \mathrel{\widehat{=}} b := b \cup [\![x]\!] \\
\quad addAll(\textbf{val}\ bs : bag\ of\ char) \mathrel{\widehat{=}} b := b \cup bs \\
\quad cardinality(\textbf{res}\ r : int) \mathrel{\widehat{=}} r := |b| \\
\textbf{end}
\end{array}
$$

Inheritance, Substitutability, and the Essence of the Problem

Inheritance via Modifiers, Class Substitutability

An extension class E results from the application of a modifier M to a base class C, i.e. $E = M\ \mathbf{mod}\ C$.

A class $C\ is\ refined\ by$ a class C', if the externally observable behavior of objects generated by C' is the externally observable behavior of objects generated by C or an improvement of it.

The Essence of the Fragile Base Class Problem

The *flexibility property*

if $C\ is\ refined\ by\ C'$ and $C\ is\ refined\ by\ (M\ \mathbf{mod}\ C)$
then $C\ is\ refined\ by\ (M\ \mathbf{mod}\ C')$

does not hold in general, i.e.

$$C \sqsubseteq C' \ \wedge\ C \sqsubseteq (M\ \mathbf{mod}\ C) \ \not\Rightarrow\ C \sqsubseteq (M\ \mathbf{mod}\ C')$$

Unanticipated Mutual Recursion

$$
\begin{array}{ll}
C \;=\; \textbf{class} & M \;=\; \textbf{modifier} \\
\quad x : int := 0 & \\
\quad m \mathrel{\widehat{=}} x := x + 1 & \\
\quad n \mathrel{\widehat{=}} x := x + 1 & \quad n \mathrel{\widehat{=}} self.m \\
\textbf{end} & \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
C' \;=\; \textbf{class} \\
\quad x : int := 0 \\
\quad m \mathrel{\widehat{=}} self.n \\
\quad n \mathrel{\widehat{=}} x := x + 1 \\
\textbf{end}
\end{array}
$$

"No cycles" requirement: *A base class revision and a modifier should not jointly introduce new cyclic method dependencies*

Unjustified Assumptions in Revision Class

$$
\begin{array}{l}
C \;=\; \textbf{class} \\
\quad m(\textbf{val}\ x : \mathit{real}, \textbf{res}\ r : \mathit{real}) \mathrel{\widehat{=}} \\
\qquad \textbf{pre}\ x \geq 0\ \textbf{post}\ r'^2 = x \\
\quad n(\textbf{val}\ x : \mathit{real}, \textbf{res}\ r : \mathit{real}) \mathrel{\widehat{=}} \\
\qquad \textbf{pre}\ x \geq 0\ \textbf{post}\ r'^4 = x \\
\textbf{end}
\end{array}
\qquad
\begin{array}{l}
M \;=\; \textbf{modifier} \\
\quad m(\textbf{val}\ x : \mathit{real}, \textbf{res}\ r : \mathit{real}) \mathrel{\widehat{=}} \\
\qquad r := -\sqrt{x} \\
\\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
C' \;=\; \textbf{class} \\
\quad m(\textbf{val}\ x : \mathit{real}, \textbf{res}\ r : \mathit{real}) \mathrel{\widehat{=}} \\
\qquad r := \sqrt{x} \\
\\
\quad n(\textbf{val}\ x : \mathit{real}, \textbf{res}\ r : \mathit{real}) \mathrel{\widehat{=}} \\
\qquad \mathit{self}.m(x, r); \mathit{self}.m(r, r) \\
\textbf{end}
\end{array}
$$

"No revision self-calling assumptions" requirement: *Revision class methods should not make any additional assumptions about the behavior of the other methods of itself. Only the behavior described in the base class may be taken into consideration.*

Unjustified Assumptions in Modifier

$$
\begin{array}{l}
C \;=\; \textbf{class} \\
\quad l(\textbf{val}\ v : int) \mathrel{\widehat{=}} \{v \geq 5\} \\
\quad m(\textbf{val}\ v : int) \mathrel{\widehat{=}} self.l(v) \\
\quad n(\textbf{val}\ v : int) \mathrel{\widehat{=}} \textbf{skip} \\
\textbf{end}
\end{array}
\qquad
\begin{array}{l}
M \;=\; \textbf{modifier} \\
\quad l(\textbf{val}\ v : int) \mathrel{\widehat{=}} \textbf{skip} \\
\quad n(\textbf{val}\ v : int) \mathrel{\widehat{=}} self.m(v) \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
C' \;=\; \textbf{class} \\
\quad l(\textbf{val}\ v : int) \mathrel{\widehat{=}} \{v \geq 5\} \\
\quad m(\textbf{val}\ v : int) \mathrel{\widehat{=}} \{v \geq 5\} \\
\quad n(\textbf{val}\ v : int) \mathrel{\widehat{=}} \textbf{skip} \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
(M\ \textbf{mod}\ C) \;=\; \textbf{class} \\
\quad l(\textbf{val}\ v : int) \mathrel{\widehat{=}} \textbf{skip} \\
\quad m(\textbf{val}\ v : int) \mathrel{\widehat{=}} self.l(v) \\
\quad n(\textbf{val}\ v : int) \mathrel{\widehat{=}} self.l(v) \\
\textbf{end}
\end{array}
\qquad
\begin{array}{l}
(M\ \textbf{mod}\ C') \;=\; \textbf{class} \\
\quad l(\textbf{val}\ v : int) \mathrel{\widehat{=}} \textbf{skip} \\
\quad m(\textbf{val}\ v : int) \mathrel{\widehat{=}} \{v \geq 5\} \\
\quad n(\textbf{val}\ v : int) \mathrel{\widehat{=}} \{v \geq 5\} \\
\textbf{end}
\end{array}
$$

"No base class down-calling assumptions" requirement: *Modifier methods should disregard the fact that base class self-calls can get redirected to the modifier itself. In this case bodies of the corresponding methods in the base class should be considered instead, as if there were no dynamic binding.*

Direct Access to the Base Class State

$$
\begin{array}{ll}
C = \textbf{class} & M = \textbf{modifier} \\
\quad x : int := 0 & \\
\quad m \mathrel{\widehat{=}} x := x + 1 & \\
\quad n \mathrel{\widehat{=}} x := x + 2 & \quad n \mathrel{\widehat{=}} x := x + 2 \\
\textbf{end} & \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
C' = \textbf{class} \\
\quad x : int := 0; y : int := 0 \\
\quad m \mathrel{\widehat{=}} y := y + 1; x := y \\
\quad n \mathrel{\widehat{=}} y := y + 2; x := y \\
\textbf{end}
\end{array}
$$

"No direct access to the base class state" requirement: *An extension class should not access the state of its base class directly, but only through calling base class methods.*

Refinement Calculus

- extended language of Dijkstra's guarded commands:

$$\begin{aligned} S ::= \quad & \mathbf{skip} \mid \{p\} \mid [p] \mid x := e \mid \{x := x' \cdot b\} \\ & \mid [x := x' \cdot b] \mid S_1; S_2 \mid \mathbf{if}\ g\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} \\ & \mid \mathbf{while}\ g\ \mathbf{do}\ S\ \mathbf{od} \mid \mathbf{begin\ var}\ x \cdot S\ \mathbf{end} \end{aligned}$$

  every statement is identified with its weakest precondition predicate transformer

- correctness: S is correct with respect to specification (p, q) iff $p \subseteq S\ q$.

- refinement: improving a program while preserving correctness. $S \sqsubseteq T$ holds iff T satisfies any specification satisfied by S.

- data refinement: refinement between programs operating on different state spaces. $S \sqsubseteq_R T$ holds iff T refines S w.r.t. abstraction relation R coercing state space of T to that of S.

Formalization of Classes

Classes are templates for creation of objects.

A method of a class with n methods is defined by:

$$m_i = \lambda self.c_i, \text{ where } self = (x_1, ..., x_n)$$

Method parameters are modeled by global variables that methods of a class and its clients can access.

A class C is declared by:

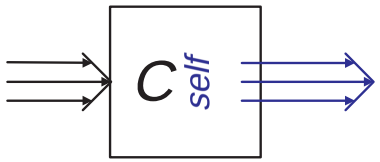
$$C \;=\; \textbf{class}\ c := c_0,\ m_1 \;\widehat{=}\; c_1, ..., m_n \;\widehat{=}\; c_n\ \textbf{end}$$

and is defined by:

$$C \;=\; (c_0, cm), \text{where } cm = \lambda self \cdot (c_1, ..., c_n),$$

and where c_0 is an initial value of the internal class state.

Graphical representation of a class C:



Object Creation

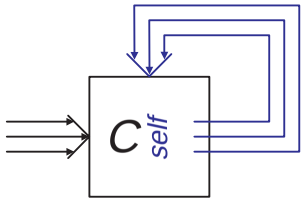
Objects have all self-calls resolved with methods of the same class.

Statement tuples form a complete lattice with the refinement ordering and $cm = \lambda self \cdot (c_1, ..., c_n)$ is monotonic in its $self$ argument, therefore there exists a unique least fixed point of the function cm.

The operation $create$ returns an object of class C:

$$create\ C \ \widehat{=}\ (c_0, \mu\ cm)$$

Graphical representation of creating an instance of class C:



Modifier = extension class - base class.

A modifier is declared as follows:

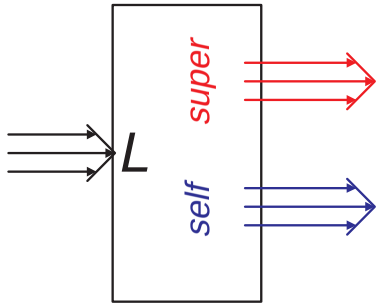
$$L = \textbf{modifier } m_1 \mathrel{\widehat{=}} l_1, ..., m_n \mathrel{\widehat{=}} l_n \textbf{ end}$$

and modeled by:

$$L = \lambda self \cdot \lambda super \cdot (l_1, ..., l_n), \text{ where}$$

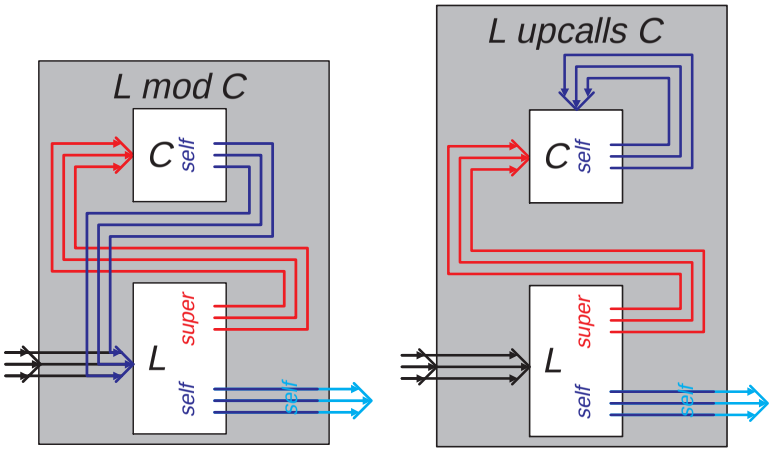
- $self$ is an abbreviation for $(x_1, ..., x_n)$
- $super$ is an abbreviation for $(y_1, ..., y_n)$
- $(l_1, ..., l_n)$ are the bodies of the overriding methods
- L is monotonic in both arguments

Graphical representation of a modifier L:



Applying Modifiers to Classes

Graphical representation of applying a modifier to a base class:



The operators **mod** and **upcalls** create an extension class from the base class C and the modifier L:

$$L \ \mathbf{mod} \ C \quad \widehat{=} \quad (\ c_0,\ lcm = \lambda self \cdot (\ lm \ \overbrace{self}^{self} \ \overbrace{(cm \ \ self)}^{super}\ )\ )$$

$$L \ \mathbf{upcalls} \ C \quad \widehat{=} \quad (\ c_0,\ lcm = \lambda self \cdot (\ lm \ \overbrace{self}^{self} \ \overbrace{(\mu \ \ cm)}^{super}\ )\ )$$

Refinement on Classes

Abstract Data Type Refinement

An abstract data type T is defined by (t_0, tp), where t_0 is an initial value of the internal state and tp is a tuple of procedures modifying the internal state of T.

The ADT $T = (t_0, tp)$ is data refined by an ADT $T' = (t'_0, tp')$ via a relation R, coercing the state space of T' to that of T, if the initialization establishes R and all procedures preserve R,

$$T \sqsubseteq_R T'$$

Class Refinement

The class $C' = (c'_0, cm')$ refines the class $C = (c_0, cm)$ if there exists a relation R, coercing their corresponding internal states, such that the objects these classes generate are in the data refinement relation with respect to R:

$$C \sqsubseteq C' \;\widehat{=}\; \exists R \cdot (\mathit{create}\ C) \sqsubseteq_R (\mathit{create}\ C')$$

Flexibility Theorem

Flexibility property: $C \sqsubseteq D \ \wedge \ C \sqsubseteq (M \ \mathbf{mod} \ C) \ \not\Rightarrow \ C \sqsubseteq (M \ \mathbf{mod} \ D)$

Flexibility Theorem: *Let* $C = (c_0, cm)$, *with* $cm = \lambda self \cdot (c_1, ..., c_n)$, $D = (d_0, dm)$ *with* $dm = \lambda self \cdot (d_1, ..., d_n)$ be classes, and $L = \lambda self \cdot \lambda super \cdot (l_1, ..., l_n)$ *a modifier, and* R*a state coercing relation. Then the following holds*:

$$\mu \ C \sqsubseteq_R (d_0, \ dm \ ((\mu \ cm)\downarrow(R \times Id))) \ \wedge \ \mu \ C \sqsubseteq \mu \ (L \ \mathbf{upcalls} \ C) \ \Rightarrow$$
$$\mu \ C \sqsubseteq_R (L \ \mathbf{mod} \ D)$$

- the "no cycles" requirement is handled by total reordering of the methods
- the "no revision self-calling assumptions" requirement is formalized as the first conjunct in the flexibility theorem
- the "no base class down-calling assumptions" requirement is formalized as the second conjunct
- the "no direct access to the base class state" requirement is addressed by our formalization of modifiers

Conclusions and Future Work

Conclusions

- The application of formal methods to analyzing the fragile base class problem allowed to reveal the problems that were overlooked by other researchers. This study provides precise explanations to the problems recognized by many practitioners.
- The analysis of the fragile base class problem has revealed that the inheritance is not monotonic in its base class argument.
- Our formalization of classes and inheritance allows for reasoning about the class behavior in presence of dynamic binding and self-recursion. Our definition of class refinement is based on the well-established theory of data refinement.

Future Work

- Generalizing the results by relaxing the confinements on the problem and weakening the restrictions we have imposed on inheritance.
- Studying the effects of disciplining inheritance on component and object-oriented languages and systems.
- Comparing benefits of disciplined inheritance and other safe code reuse techniques.
- Developing a methodology for applying disciplined inheritance in practice.