Matriona: Class Nesting with Parameterization in Squeak/Smalltalk

Matthias Springer    Fabio Niephaus
Robert Hirschfeld    Hidehiko Masuhara

Hasso Plattner Institute, University of Potsdam
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology

March 16, 2016
Overview

Introduction

Requirements

Mechanism

Examples

Conclusion
Vision for Matriona

Matriona should be a module system . . .

- **for Squeak/Smalltalk**
  - Easy to implement (metaprogramming, reflection)
  - Needs a module system

- **for long-living systems** (c.f. highly available systems)
  Cannot turn off (restart) system to install new software

- **for a programming environment** that hosts a variety of applications
  Single OS process, multiple applications in the same object space (image)

- **that makes it easy to experiment** (*exploratory programming*)
  *Try out new stuff* and see what happens (Live programming, inspector, . . .)

- **that promotes modularity**
  (composability, decomposability, understandability)

---

1B. Meyer: Object-Oriented Software Construction
Vision for Matriona

Matriona should be a module system . . .

• . . . for Squeak/Smalltalk
  − Easy to implement (metaprogramming, reflection)
  − Needs a module system

• . . . for long-living systems (c.f. highly available systems)
  Cannot turn off (restart) system to install new software

• . . . for a programming environment that hosts a variety of applications
  Single OS process, multiple applications in the same object space (image)

• . . . that makes it easy to experiment (exploratory programming)
  Try out new stuff and see what happens (Live programming, inspector, . . .)

• . . . that promotes modularity\(^1\)
  (composability, decomposability, understandability)

\(^1\)B. Meyer: Object-Oriented Software Construction
Vision for Matriona

Matriona should be a module system . . .

• . . . for Squeak/Smalltalk
  – Easy to implement (metaprogramming, reflection)
  – Needs a module system

• . . . for long-living systems (c.f. highly available systems)
  Cannot turn off (restart) system to install new software

• . . . for a programming environment that hosts a variety of applications
  Single OS process, multiple applications in the same object space (image)

• . . . that makes it easy to experiment (exploratory programming)
  Try out new stuff and see what happens (Live programming, inspector, . . .)

• . . . that promotes modularity¹
  (composability, decomposability, understandability)

¹B. Meyer: Object-Oriented Software Construction
Vision for Matriona

Matriona should be a module system . . .

• . . . for Squeak/Smalltalk
  – Easy to implement (metaprogramming, reflection)
  – Needs a module system

• . . . for long-living systems (c.f. highly available systems)
  Cannot turn off (restart) system to install new software

• . . . for a programming environment that hosts a variety of applications
  Think of the programming language as an operating system

• . . . that makes it easy to experiment (exploratory programming)
  Try out new stuff and see what happens (Live programming, inspector, . . .)

• . . . that promotes modularity¹
  (composability, decomposability, understandability)

¹B. Meyer: Object-Oriented Software Construction
Vision for Matriona

Matriona should be a module system . . .

• . . . for Squeak/Smalltalk
  − Easy to implement (metaprogramming, reflection)
  − Needs a module system

• . . . for long-living systems (c.f. highly available systems)
  Cannot turn off (restart) system to install new software

• . . . for a programming environment that hosts a variety of applications
  “An operating system is a collection of things that don’t fit into a language. There shouldn’t be one.” (Dan Ingalls)

• . . . that makes it easy to experiment (exploratory programming)
  Try out new stuff and see what happens (Live programming, inspector, . . .)

• . . . that promotes modularity\(^1\)
  (composability, decomposability, understandability)

\(^1\) B. Meyer: Object-Oriented Software Construction
Vision for Matriona

Matriona should be a module system . . .

• . . . for Squeak/Smalltalk
  − Easy to implement (metaprogramming, reflection)
  − Needs a module system

• . . . for long-living systems (c.f. highly available systems)
  Cannot turn off (restart) system to install new software

• . . . for a programming environment that hosts a variety of applications
  “An operating system is a collection of things that don’t fit into a language. There shouldn’t be one.” (Dan Ingalls)

• . . . that makes it easy to experiment (exploratory programming)
  Try out new stuff and see what happens (Live programming, inspector, . . .)

• . . . that promotes modularity\(^1\)
  (composability, decomposability, understandability)

---

\(^1\)B. Meyer: Object-Oriented Software Construction
Vision for Matriona

Matriona should be a module system . . .

- . . . for Squeak/Smalltalk
  - Easy to implement (metaprogramming, reflection)
  - Needs a module system

- . . . for long-living systems (c.f. highly available systems)
  Cannot turn off (restart) system to install new software

- . . . for a programming environment that hosts a variety of applications
  “An operating system is a collection of things that don’t fit into a language. There shouldn’t be one.” (Dan Ingalls)

- . . . that makes it easy to experiment (exploratory programming)
  “A system [...] to serve the creative spirit” (Dan Ingalls)

- . . . that promotes modularity\(^1\) (composability, decomposability, understandability)

---

\(^1\) B. Meyer: Object-Oriented Software Construction
Vision for Matriona

Matriona should be a module system . . .

- . . . for Squeak/Smalltalk
  - Easy to implement (metaprogramming, reflection)
  - Needs a module system

- . . . for long-living systems (c.f. highly available systems)
  Cannot turn off (restart) system to install new software

- . . . for a programming environment that hosts a variety of applications
  “An operating system is a collection of things that don’t fit into a language. There shouldn’t be one.” (Dan Ingalls)

- . . . that makes it easy to experiment (exploratory programming)
  Try out new stuff and see what happens (Live programming, inspector, . . .)

- . . . that promotes modularity
  (composability, decomposability, understandability)

---

1B. Meyer: Object-Oriented Software Construction
Matriona
Running Example: Space Cleanup

- All game objects are subclasses of Morph
- Game is built using Morph composition
- Classes: Item, Player, ...
Running Example: Space Cleanup

- All game objects are subclasses of Morph
- Game is built using Morph composition
- Classes: Item, Player, ..., Tile
Running Example: Space Cleanup

- All game objects are subclasses of Morph
- Game is built using Morph composition
- Classes: Item, Player, ..., Tile, Level
Running Example: Space Cleanup

• All game objects are subclasses of Morph
• Game is built using Morph composition
• Classes: Item, Player, ..., Tile, Level

```smalltalk
class Level extends Morphic.Morph {
    int stepTime() {
        return 1000;
    }
}
```
Overview

Introduction

Requirements

Mechanism

Examples

Conclusion
Module Versioning

• **Goal**: Run a variety of applications, composability

• **Dependency Conflicts**: Multiple applications require the same dependency in different versions

**Application A** → **Library C v1.4**

provides class Foo

**Application B** → **Library C v1.6**

provides class Foo

• **Problem**: Naming conflicts between versions
Module Versioning

- **Goal:** Run a variety of applications, composability
- **Dependency Conflicts:** Multiple modules require the same dependency in different versions

```
Application A  Library C  ➔  Library E v1.4
            ➔  Library D  ➔  Library E v1.6
```

- **Problem:** Naming conflicts between versions
Module Versioning

- **Goal:** Run a variety of applications, composability, long-living system
- **Dependency Conflicts:** Multiple modules require the same dependency in different versions

Application A ➔ Library C ➔ Library E v1.4
Library D ➔ Library E v1.6

- **Application Upgrade:** Install both versions, then perform upgrade (possibly live upgrade)
- **Problem:** Naming conflicts between versions
Module Inheritance

• **Goal:** Exploratory programming, decomposability
• **Task:** Add unforeseen variation points. Design variants of Space Cleanup, where . . .
  - the speed of the game can be adjusted (overwrite Level»stepTime)
  - items can deal damage (add methods to all items)
Module Inheritance

• **Goal:** Exploratory programming, decomposability

• **Task:** Add unforeseen variation points. Design variants of Space Cleanup, where . . .
  - the speed of the game can be adjusted (overwrite Level»stepTime)
  - items can deal damage (add methods to all items)

• **Design Constraints:** Apply changes to the original application automatically, leave the original application intact
External Configuration

- **Goal:** Exploratory programming, composability
- **Task:** Design a variant of Space Cleanup, where a UI framework implementation is passed as an argument

![Diagram](image)

- **Problem:**
  - UI elements are subclasses of Morphic.Morph
  - Dependency cannot simply be passed as argument to constructor/factory method, because class hierarchy depends on it
Overview

Introduction

Requirements

Mechanism

Examples

Conclusion
Mechanism

- Classes can have variables, methods, and **nested classes**
- Nested classes are . . .
  - . . . class-side members
  - . . . accessed using message sends
  - . . . can have parameters (accessed using message sends to class object)
- Top-level class is called *module*
Name Lookup Example (1/4)

Smalltalk

Collection

Algorithm

Sorting

sort(…)

Array

sort(…)

void sort(…) {
    result = scope.Algorithm.Sorting.sort(…)
    ...
}

Start lookup in self class, then enclosing classes

scope.Algorithm should resolve to St.Collection.Algorithm
Name Lookup Example (2/4)

Smalltalk

Collection
  Algorithm
    Sorting
      sort(...) 
  Array
    sort(...) 

SpaceCleanup
  Algorithm
    Graph
    ResizingArray
      extends Smalltalk.Collection.Array

void sort(...) {
    result = scope.Algorithm.Sorting.sort(...)
    ...
}

St.Scu.ResizingArray.sort: scope.Algorithm should resolve to St.Collection.Algorithm
Name Lookup Example 3/4

Smalltalk
  ┌───────────┐
  │ QuickCollection │ Collection
  │   ┌───────────┐  └───────────┐
  │   │ Algorithm   │ Algorithm
  │   │   ┌───────────┐  └───────────┐
  │   │   └──Sorting │    └──Sorting
  │   │             │                └──sort(...)
  │   │             │    sort(...)       Array
  │   │             │                └──sort(...)       Array
  │   └──Array      │

St.QC.Array.sort: scope.Algorithm should resolve to St.Collection.Algorithm
Name Lookup Example 4/4

- scope.Algorithm is late bound and can refer to classes, methods, parameters
- Name lookup mechanism determines which Algorithm to choose
Inherited Class Copies

Smalltalk

QuickCollection  
Algorithm  
   Sorting  
      sort(...)  
|          |  
Array  

Collection  
Algorithm  
   Sorting  
      sort(...)  
|          |  
Array  

- meta programming (allInstances)
- class-side state
- possibly different parameters

• super(St.QC.Algorithm) is an inherited class copy of St.C.Algorithm
• Notation: St.QC.Algorithm[St.C.Algorithm]
High-level Idea

- **Idea:** Generalize method lookup to class nesting hierarchies
- **Standard Method Lookup:**
  \( \text{sub}(C) \) can override methods defined in \( C \)
- **Nesting-aware Name Lookup:**
  - \( \text{sub}(C) \) can override names defined in \( C \)
  - \( \text{sub}(\text{enclosing}(C)) \) can override names defined in \( \text{enclosing}(C) \)
  - \( \text{sub}(\text{enclosing}(\text{enclosing}(C))) \) can override names defined in \( \text{enclosing}(\text{enclosing}(C)) \)
  - ...
Relative Name Lookup (1/2)

- **Lexical Class Nesting Hierarchy**: static hierarchy of enclosing classes
- **Run-time Class Nesting Hierarchy**: dynamic hierarchy of enclosing classes, taking into account run-time (polymorphic) type of receiver

\[ L = (\text{St.C.Array}, \text{St.C}, \text{St}) \]
\[ R_1 = (\text{St.QC.Array}\,[\text{St.C.Array}], \text{St.QC}, \text{St}) \]
\[ R_2 = (\text{St.Scu.ResizingArray}, \text{St.Scu}, \text{St}) \]
Traverse both lexical ($L$) and run-time class nesting hierarchy ($R$) in parallel ($R$ takes precedence), as long as one of the following is true, where $l \in L$ and $r \in R$.

- $r = l$
- $r$ is a subclass of $l$, i.e., $r \triangleright l$
- $r$ is an inherited class copy of $l$, i.e., $r \rightsquigarrow l$
- $r$ is a subclass of an inherited class copy of $l$, i.e., $r \triangleright \rightsquigarrow l$
Example: Relative Name Lookup (1/2)

Smalltalk

Collection
  Algorithm
    Sorting
      sort(...)

Array
  sort(...)

SpaceCleanup
  Algorithm
    Graph

ResizingArray
  extends Smalltalk.Collection.Array
  void sort(...) {
    result = scope.Algorithm.Sorting.sort(...)
    ...
  }

St.Scu.ResizingArray $\triangleright$ St.C.Array ($\rightarrow$ R, L)

Lookup fails in both R and then L
Example: Relative Name Lookup (1/2)

Smalltalk

Collection

Algorithm

Sorting

Array

sort(...)

SpaceCleanup

Algorithm

Graph

ResizingArray

extends Smalltalk.Collection.Array

L

R

St.Scu \{=, \triangleright, \sim, \sim\triangleright\} St.C

Lookup succeeds in L

void sort(...) {
    result = scope.Algorithm.Sorting.sort(...)
    ...
}
Example: Relative Name Lookup (2/2)

Smalltalk

QuickCollection

Algorithm

Sorting

Array

Collection

Algorithm

Sorting

Array

void sort(...) {
  result = scope.Algorithm.Sorting.sort(...)
  ...
}

St.QC.Array ↦ St.C.Array  (→ R, L)
Lookup fails in both R and then L
Example: Relative Name Lookup (2/2)

Smalltalk

QuickCollection
  extends Smalltalk.Collection
  extends super.Algorithm
  extends super.Sorting
  sort(...)

Collection
  extends Smalltalk.Collection
  extends super.Algorithm
  extends super.Sorting
  sort(...)

Array
  sort(...)

void sort(...) {
  result = scope.Algorithm.Sorting.sort(...)
  ...
}

St.QC ▷ St.C  (→ R, L)
Lookup succeeds in R
Class Nesting Details

- Lookup mechanism is similar to Java, differs from Newspeak (lookup in self class, then superclasses, then enclosing class and superclasses, etc.)
- Nested classes are virtual and can be overridden
- Lookup mechanism looks up methods and nested classes (and parameters)
- extends supports arbitrary expressions
- Overwritten and original nested classes do not have to be in a subclass/subtype relationship (c.f. Jx, gbeta)
Class Parameterization (1/2)

- Must provide argument to obtain concrete class object
- Different class object for every *instantiation* (c.f. C++ templates)
- Access parameter value via message send to class object
- Same name lookup mechanism
- Name lookup precedence (→ shadowing)
  1. Method in $r$
  2. Parameter in $r$
  3. Class in $r$
  4. Method in $l$
  5. Parameter in $l$
  6. Class in $l$
Class Paramterization (2/2)

Smalltalk

QuickCollection
  extends Smalltalk.Collection
  Algorithm
    extends super.Algorithm
      Sorting
        extends super.Sorting
          sort(...)
Overview

Introduction

Requirements

Mechanism

Examples

Conclusion
Module Versioning

Smalltalk
  SpaceCleanup
    v1
      extends Smalltalk.Matriona.Version
  v1
    extends Smalltalk.Matriona.Version
      Morphic
        return Smalltalk.Morphic.v3.v1;
    Level
      extends scope.Morphic.Morph
        ...
    v2
      extends Smalltalk.Matriona.Version
        Morphic
          return Smalltalk.Morphic.v4
          .>=<=(2, 4);
      ...

• Convenience methods: <, <=, >, >=, >>, <=>>, <=>>, latest
• Name lookup finds classes, parameters, and methods
• Morphic is an import
External Configuration (1/2)

- Decouple implementation from dependencies
- Morphic parameter should implement Morphic interface
class Smalltalk {
    class SpaceCleanup<Morphic implements Smalltalk.Morphic.Interface> {
        class Tile extends scope.Morphic.Morph {
            class Item extends scope.Morphic.Morph { /* ... */ }
            class Player extends scope.Item { /* ... */ }
            class Monster extends scope.Item { /* ... */ }
        }

        static void run() { /* ... */ }
    }
}

Smalltalk.SpaceCleanup<Smalltalk.NativeRendering>.run();
Module Inheritance

- **Task:** Design variants of Space Cleanup, where ...
  - the speed of the game can be adjusted (overwrite `Level>>stepTime`)
  - items can deal damage (add methods to all items)

- **Design Constraints:** Apply changes to the original application automatically, leave the original application intact
Module Inheritance: Speedy Space Cleanup

class Smalltalk {
    class SpaceCleanup {
        Level currentLevel;
        class Level { /* ... */ }
    }
}

class SpeedySpaceCleanup extends scope.SpaceCleanup {
    @Override class Level extends super.Level {
        int stepTime;
        @Override int stepTime() { return stepTime; }
    }
    void setSpeed(int stepTime) {
        currentLevel.stepTime = stepTime;
    }
}
}
Module Inheritance: Damage Space Cleanup (1/3)

- **Smalltalk**
  - SpaceCleanup
    - Tile
    - Item
      - extends Smalltalk
      - .Morphic.Morph
    - Player
      - extends scope.Item
    - Monster
      - extends scope.Item

- **Damage** functionality should be implemented in items.
- Need to define subclasses of Item and Monster.
- Monster\(_{dmg}\) should inherit from both Monster and Item\(_{dmg}\).
Module Inheritance: Damage Space Cleanup (2/3)

Smalltalk

SpaceCleanup

Tile

Item

Player

Monster

DamageSpaceCleanup

extends Smalltalk.SpaceCleanup

tile

extends super.Tile

damage

extends Smalltalk.SpaceCleanup

extends super.Tile

damage

extends super.Item

damage

extends scope.Item

damage

extends scope.Item

damage

extends scope.Item

damage

^ { #player -> 0.25. 
         #slime -> 0.33 }

^ { }...

...
Module Inheritance: Damage Space Cleanup (3/3)

Resulting superclass hierarchy of Monster_dmg:

2. St.DScu.Tile.Item
4. St.Morphic.Morph
Generalization: More than 2 Hierarchies

- Effectively implements multiple inheritance
- Related work: Mixin layers
Overview

Introduction

Requirements

Mechanism

Examples

Conclusion
Conclusion

• **Vision for Matriona**: support long-living systems, multiple applications in one execution environment, exploratory programming, modularity (composability, decomposability, understandability)

• **Techniques**: Module versioning, module inheritance, external configuration

• **First steps**: A module system that . . .
  - hosts modules in various versions
    (→ composability, long-living systems)
  - makes it easy to design module variants
    (→ exploratory programming, decomposability)

• **Next steps**:
  - Migration of running applications (state/object migration)
  - Class extensions (backward compatibility)\(^1\)
  - Method/class visibility (modular protection)

\(^1\)LASSY workshop: Hierarchical Layer-based Class Extensions in Squeak/Smalltalk