Application settings

From a monolithic to a modular architecture

S. Ducasse, L. Fabresse, G. Polito, and P. Tesone
Goals

- Think about **customizable** elements
- Think about modularity
- Study one real case: Preference in Squeak and Pharo
The case of Preferences

[Image of a settings browser window showing preferences settings for Appearance, Gradient, Standard fonts, and Desktop background image.}

- **Appearance**:
  - User interface theme: Pharo
  - Morphic: Available version: 2.4.9

- **Gradient**:
  - Other color: Blue
  - Direction: Horizontal
  - Origin: Center

- **Desktop background image**: Predefined styles: small, Medium, Large, Very large, Huge

- **Standard fonts**:
  - Default: Bitmap DejaVu Sans Regular 12
  - Code: Bitmap DejaVu Sans Regular 12
  - List: Bitmap DejaVu Sans Regular 12
  - Menu: Bitmap DejaVu Sans Regular 12

*Gradient*
If true, then more settings will be available in order to define the desktop background color gradient.
Challenges

- How to make sure that we can have
  - One application with only its preferences and its dependencies?
  - A modular definition of preferences?
- How do we make sure that
  - domain objects do not refer to preference objects and
  - still can offer preferences to the user?
Looking into the problem

Back in time in Squeak 3.8

- Preferences was a Facade (bad Design Pattern) managing preferences
- Preferences class was referenced 617 times
- Preferences was a huge dependency attractor
  - referring to many other subsystems (reading 3D files, RTF, PNG, Compiler....)
MenuMorph >> initialize
  super initialize.
  bounds := 0@0 corner: 40@10.
  self setDefaultParameters.
  self listDirection: #topToBottom.
  self hResizing: #shrinkWrap.
  self vResizing: #shrinkWrap.
  defaultTarget := nil.
  selectedItem := nil.
  stayUp := false.
  popUpOwner := nil.
  Preferences roundedMenuCorners ifTrue: [ self useRoundedCorners ]
UIButton >> label: aString font: aFontOrNil

| oldLabel m aFont |
(\(oldLabel := self findA: StringMorph\))
  ifNotNil: [ oldLabel delete ].
aFont := aFontOrNil ifNil: [ Preferences standardButtonFont ].
self extent: (m width + 6) @ (m height + 6).
m position: self center − (m extent // 2).
self addMorph: m.
m lock
Even core parts of the system

Class class >> templateForSubclassOf: priorClassName category: systemCategoryName

Preferences printAlternateSyntax
  ifTrue: [^ priorClassName asString, ' subclass (#NameOfSubclass)
  instanceVariableNames (""")
  classVariableNames ("""")
  poolDictionaries ("""")
  category (""', systemCategoryName asString, "")]
  ifFalse: [^ priorClassName asString, ' subclass: #NameOfSubclass
  instanceVariableNames: ""
  classVariableNames: ""
  poolDictionaries: ""
  category: "", systemCategoryName asString, ""]
InputSensor >> duplicateControlAndAltKeysChanged

(Preferences
valueOfFlag: #swapControlAndAltKeys
ifAbsent: [false]) ifTrue: [
  self inform: 'Resetting swapControlAndAltKeys preference'.
  (Preferences preferenceAt: #swapControlAndAltKeys) rawValue: false.
].
self installKeyDecodeTable.
The octopus AntiPattern :(

MenuMorph >> initialize
   super initialize.
   bounds := 0@0 corner: 40@10.
   self setDefaultParameters.
   self listDirection: #topToBottom.
   self hResizing: #shrinkWrap.
   self vResizing: #shrinkWrap.
   defaultTarget := nil.
   selectedItem := nil.
   stayUp := false.
   popUpOwner := nil.
   Preferences roundedMenuCorners ifTrue: [self useRoundedCorners]

BasicButton >> label: aString font: aFontOrNull
   | oldLabel m aFont |
   (oldLabel := self findA: StringMorph)
   ifNotNil: [oldLabel delete].
   aFont := aFontOrNull ifNil: [Preferences standardButtonFont].
   self extent: (m width + 6) @ (m height + 6).
   m position: self center - (m extent // 2).
   self addMorph: m.
   m lock
Referencing an attractor: monolithic system

MenuMorph >> initialize
  super initialize.
  bounds := 0@0 corner: 40@10.
  self setDefaultParameters.
  self listDirection: #topToBottom.
  self hResizing: #shrinkWrap.
  self vResizing: #shrinkWrap.
  defaultTarget := nil.
  selectedItem := nil.
  stayUp := false.
  popUpOwner := nil.

Preferences roundedMenuCorners ifTrue: [self useRoundedCorners]

BasicButton >> label: aString font: aFontOrNil
  | oldLabel m aFont |
  (oldLabel := self findA: StringMorph)
  ifNotNil: [oldLabel delete].
  aFont := aFontOrNil ifNil: [Preferences standardButtonFont].
  self extent: (m width + 6) @ (m height + 6).
  m position: self center - (m extent // 2).
  self addMorph: m.
  m lock
Analysis

- **Everybody** depends on Preferences
- Preferences is **not optional**
- Each time the Preferences class depends on a new item, all its **dependents are impacted**
- A clear **lost-lost**
- **Monolithic**
Facade and Singleton are against modularity

- A Facade should rarely be used
  - Propose a single entry point to a subsystem
  - Compiler is probably the only working example
- A Facade is often a disguised global variable!
- Singleton is most of the time not understood and correctly used (see Lectures on Singleton)
A new architecture

- A class **defines state / methods** that implement its customization points
- The class **declares** its settings via description
- The settings browser collects the **setting declaration** and builds a UI for the user
- The settings browser **configures** objects **using settings description**
Supporting Internal control flow

- Limiting external dependencies
- Reinforcing locality

MenuMorph >> initialize
  super initialize.
  bounds := 0@0 corner: 40@10.
  self setDefaultParameters.
  self listDirection: #topToBottom.
  self hResizing: #shrinkWrap.
  self vResizing: #shrinkWrap.
  defaultTarget := nil.
  selectedItem := nil.
  stayUp := false.
  popUpOwner := nil.
  shouldUseRoundCorners ifTrue: [self useRoundedCorners]

BasicButton >> label: aString font: aFontOrNil

  l oldLabel m aFont l
  (oldLabel := self findA: StringMorph)
  ifNotNil [oldLabel delete].
  aFont := aFontOrNil ifNil: [ self standardButtonFont].
  self extent: (m width + 6) @ (m height + 6).
  m position: self center - (m extent // 2).
  self addMorph: m.
  m lock
Sound obvious but so true

- An object should **be designed to be customizable** without referring to external global objects
- Think about encapsulation
- The state of customization should be internal to the object
In Action: A class implements its customization points

```
JobProgressBarMorph >> isInterruptible
  ^ self class isInterruptible

JobProgressBarMorph class >> isInterruptible
  ^ IsInterruptible ifNil: [ IsInterruptible := true ]

JobProgressBarMorph >> addInterruptionButton
  self isInterruptible ifFalse: [ ^ self ].
  self addMorphBack: (self iconNamed: #stop) asMorph
```

- IsInterruptible is a state local to JobProgressBarMorph
- JobProgressBarMorph uses its own internal state to configure itself
In Action: Settings declaration using a Builder

JobProgressBarMorph class >> interruptionSetting: aBuilder
<systemsettings>
(aBuilder setting: #isInterruptable)
  label: 'Make progress bar interruptable';
  default: true;
  description: 'When enabled, add a button to progress bars to interrupt the action when clicked.';
  parent: #progress;
  target: self;
  order: 1

- Using a builder as parameter we avoid direct references to Settings classes
- Can be optionally packaged in another package if needed
In Action: Settings Browser
A layered and modular architecture

**DOMAIN Setting Description**

JobProgressBarMorph class >> interruptionSetting: aBuilder

```systemsettings```
(aBuilder setting: #isInterruptable)
```
label: 'Make progress bar Interruptable';
default: true;
description: 'When enabled, add a button to progress bars to interrupt the action when clicked.';
parent: #progress;
target: self;
order: 1
```

uses its internal customisation hooks

declares its internal customisation hooks

Setting Browser

collect descriptions

Settings

<table>
<thead>
<tr>
<th>label</th>
<th>description</th>
</tr>
</thead>
</table>

SettingsCollector

| collect: |

uses its internal customisation hooks

declares its internal customisation hooks
Analysis

Layered

- the domain does not depend on the setting framework
- Settings do not depend on Browser

Modular

- The domain can be loaded alone
- We don’t have dependencies to unnecessary stuff
About customization

- An object should be **designed to be customisable**
- The logic flow should be **internal**
- The object logic should **not be tight to a preference object**
- The object customisation can be set from an external object (like the Setting browser)
Conclusion

- A good architecture should not promote global variable usage
- Avoid Singleton/Facade, these are anti-patterns
- Our theory is that Facade is only "useful" for Compiler :)
- Customization should first be internal