THE UNIVERSITY OF
MELBOURNE

SWEN30006
Software Modelling and Design

Introduction to Patterns

THE UNIVERSITY OF MELBOURNE

SWEN30006 Juggling by Design

Introduction to Patterns

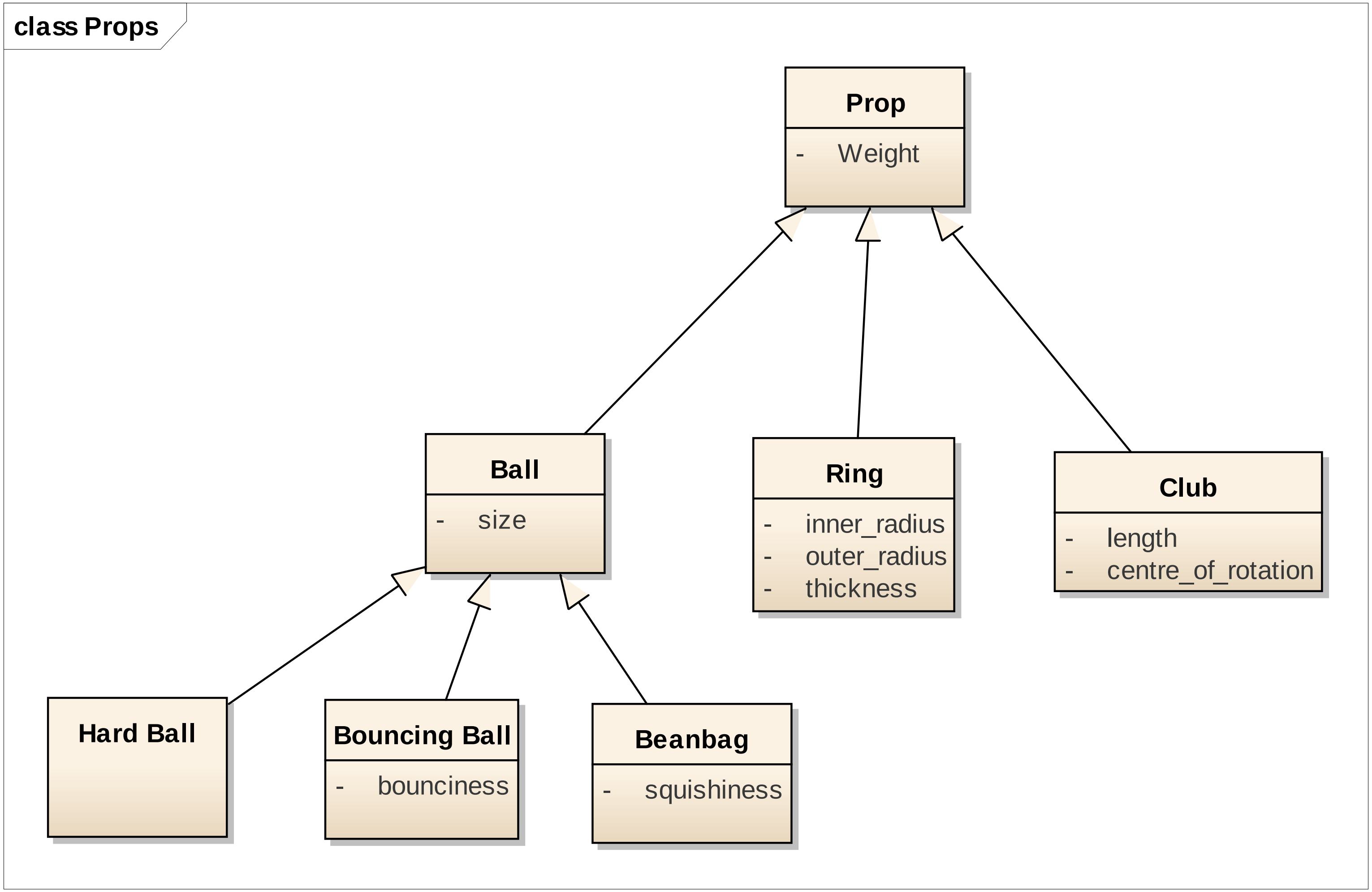
Teaching Staff

- *Coordinator/Lecturer*
  - Philip Dart (philip.dart@unimelb.edu.au)
    - University of Melbourne Student Union Juggling Teacher
    - Victorian Council of Adult Education Juggling Teacher
    - Member of Uniprocessors Juggling Troupe

Introduction

- Objects (revision)
- What are Patterns
- Examples of Patterns
- Advantages of Patterns

Objects (or Props)



Patterns

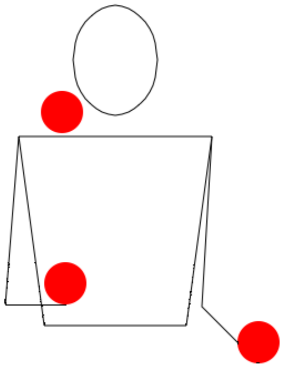
A ***pattern*** is a recurring successful application of expertise in a particular domain.

The term is usually restricted to application that is *independent* of particular technology or tools.

Patterns and Juggling

Juggling is:

- the manipulation (usually by throwing and catching)
- of props
- (more strictly) by less hands than props.

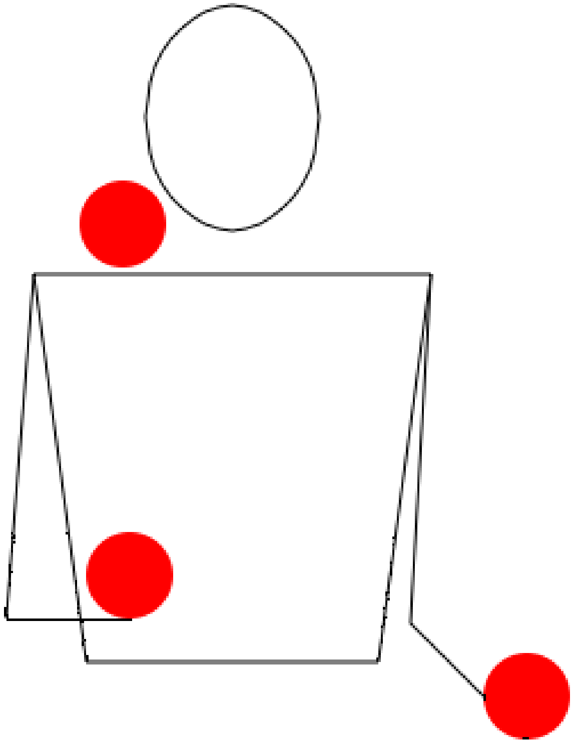


Patterns are:

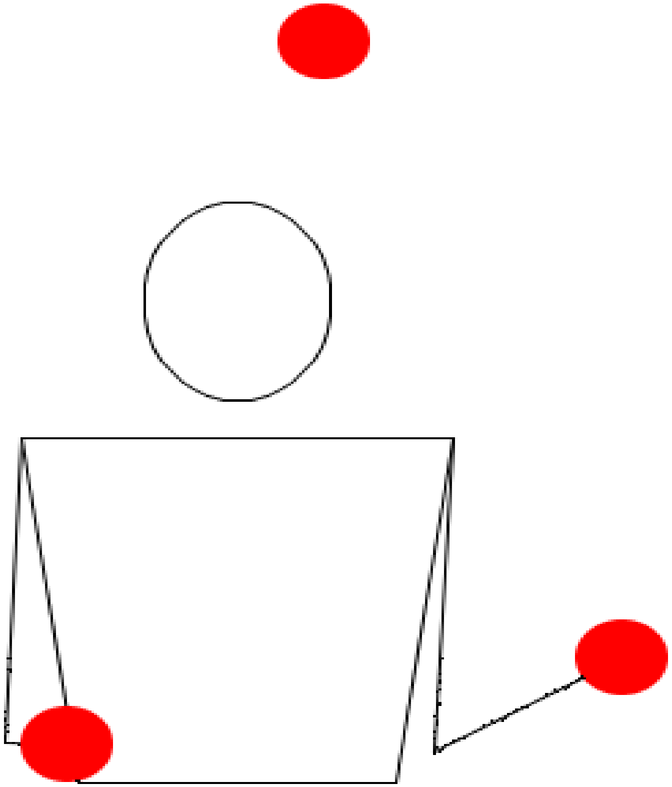
- a recurring application of juggling expertise
- independent of the particular props being juggled.
- Examples include: Cascade; Shower; Columns

See: Juggling (Wikipedia); Library of Juggling

Pattern Example: Cascade



Pattern Example: Shower

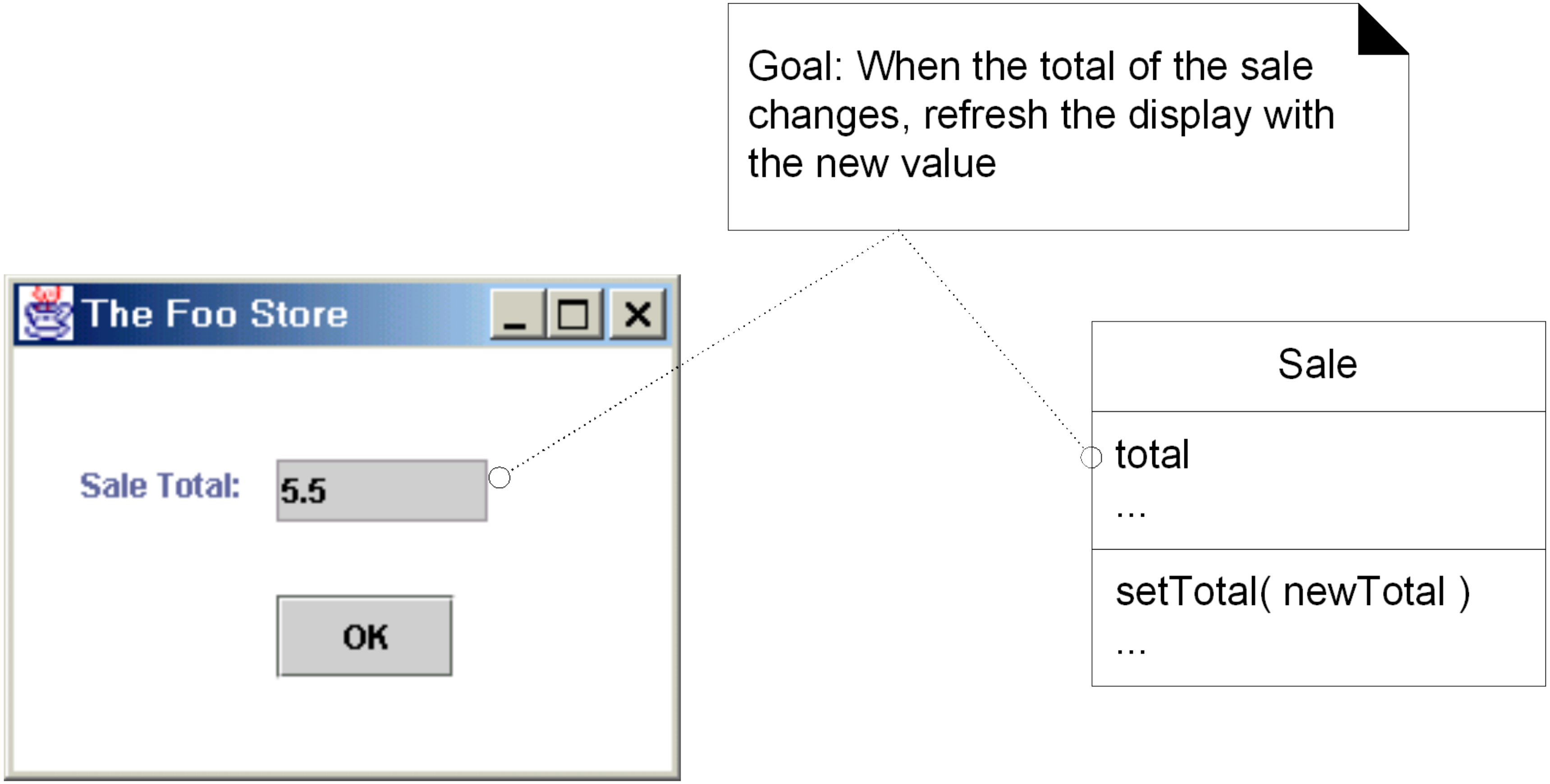


Advantages of patterns

"To become a master in a domain, one must study the successful results of other masters!"

- Capture expertise and make it accessible to non-experts in a standard form.
- Facilitate communication among practitioners by providing a common language.
- Make it easier to reuse successful applications of expertise.
- Facilitate generating modified applications.
- Improve understandability.
- Simplify documentation.

Observer Pattern (example)



Goal: When the total of the sale changes, refresh the display with the new value

Sale Total: 5.5

```
Sale
-------------------
total
...
-------------------
setTotal( newTotal )
...
```

“To handle this problem, let’s have the SalesFrame observing the Sale object”

Observer (Publish-Subscribe) Pattern

Problem

Different subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way, without the publisher needing to know much about the subscriber. What to do?

Solution: (advice)

Define a “subscriber” or “listener” interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

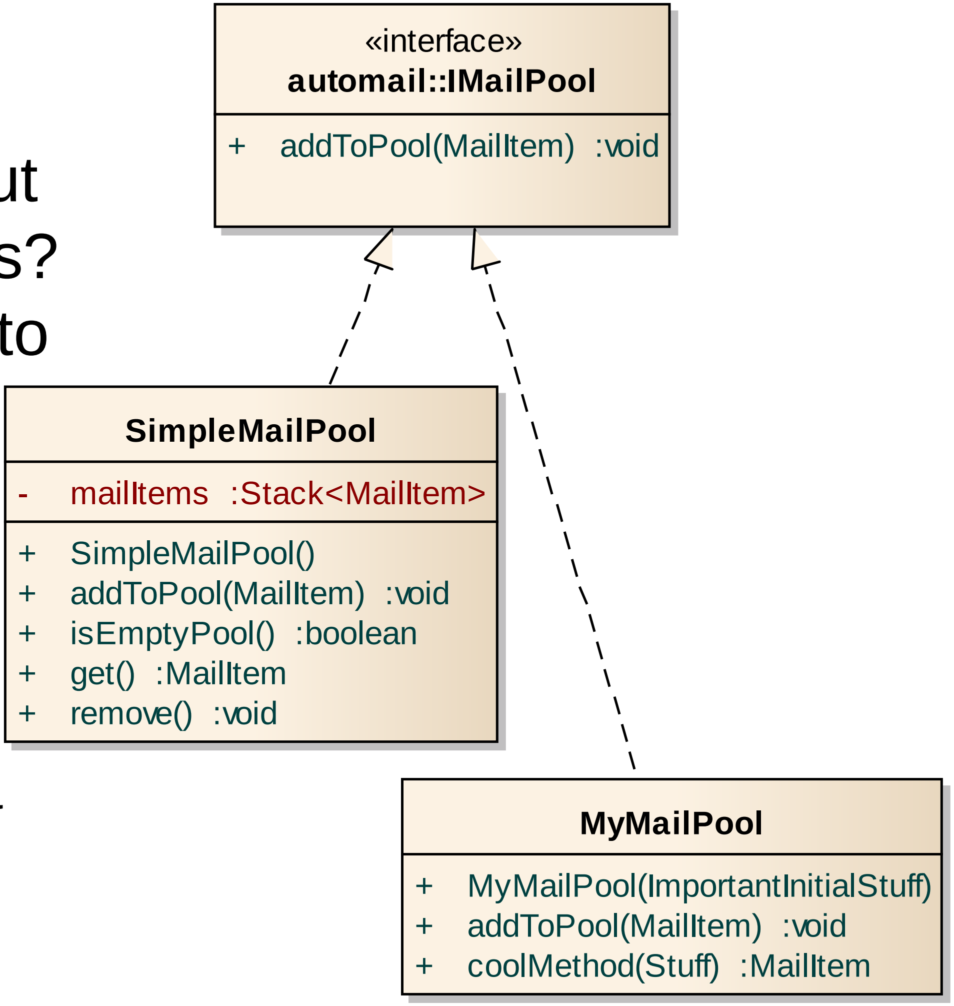
Strategy Pattern

Problem

How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies?

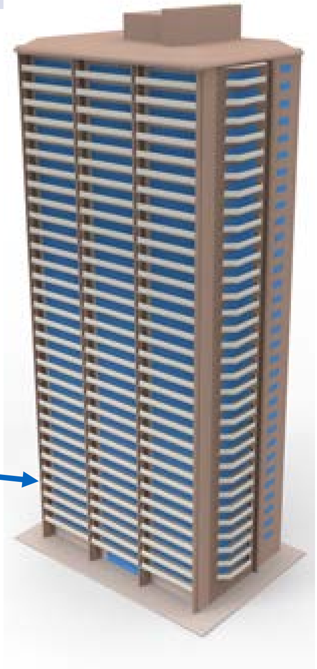
Solution: (advice)

Define each algorithm/policy/strategy in a separate class, with a common interface.


«interface»
automail::IMailPool
+ addToPool(MailItem) :void
SimpleMailPool
- mailItems :Stack<MailItem>
+ SimpleMailPool()
+ addToPool(MailItem) :void
+ isEmptyPool() :boolean
+ get() :MailItem
+ remove() :void
MyMailPool
+ MyMailPool(ImportantInitialStuff)
+ addToPool(MailItem) :void
+ coolMethod(Stuff) :MailItem

Project Part A: Mailbot Blues

- Mail Items arrive at arbitrary times at the Mail Room
- On arrival they are added to the MailPool



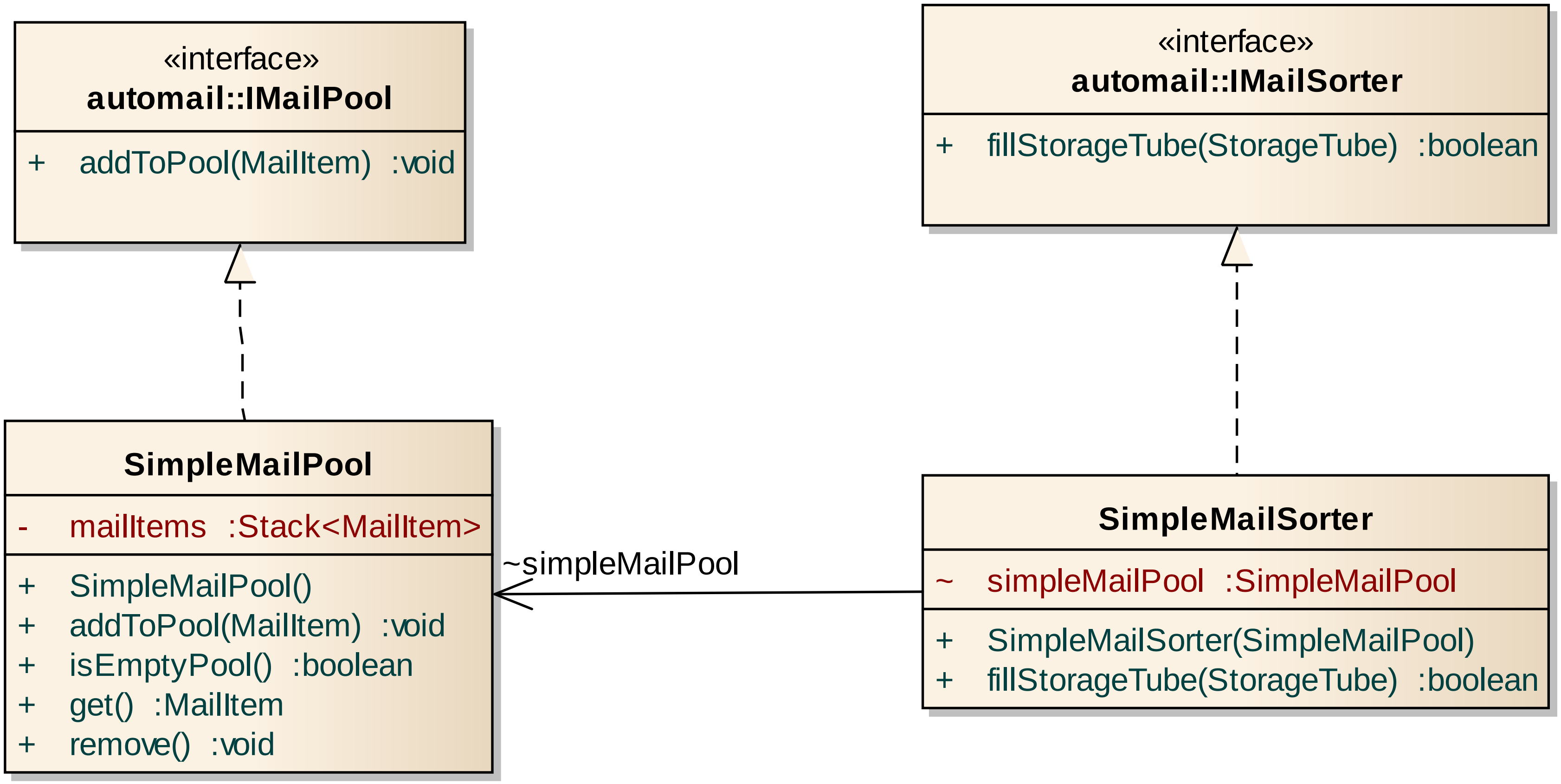
Mail Room (including MailPool)

- A Robot with a Storage Tube delivers mail
- It arrives with an empty tube at the Mail Room and requests the tube be filled
- It will only start delivery when told to do so
- It will *deliver* the tube items in FILO order

MailBot Blues: Factors

- How many floors in the building?
- Where is the Mail Room in the Building?
- What time do mail deliveries end?
- How much will fit in a Storage Tube?
- How big is a Mail Item?
- What is the priority of a Mail Item?
- What is the delivery cost of a Mail Item?
  - (time to deliver)^1.1 * (priority weight)

MailBot Blues: Design Framework



«interface»
automail::IMailPool

+ addToPool(MailItem) :void

«interface»
automail::IMailSorter

+ fillStorageTube(StorageTube) :boolean

SimpleMailPool

- mailItems :Stack<MailItem>

+ SimpleMailPool()
+ addToPool(MailItem) :void
+ isEmptyPool() :boolean
+ get() :MailItem
+ remove() :void

SimpleMailSorter

~ simpleMailPool :SimpleMailPool

+ SimpleMailSorter(SimpleMailPool)
+ fillStorageTube(StorageTube) :boolean

MailBot Blues: Advice

- Start early
- Read the instructions carefully
- Run the package
- Read the relevant parts of the code
- Get simple solution working
  - Test it with the build script
- Get a better solution working
  - Test it with the build script
  - Submit it