Common Issues: AF, RI, Rep Exposure

CSE 331Section
Krysta Yousoufian
May 3, 2012

Representation Invariant

Common Misconceptions

Misconception #1

- RI for IntQueue1:

- What's wrong with this?
  - Describes how to derive abstract value
  - Belongs in AF
  - If violated, object still maps to an abstract value
- RI defines a broken object: if violated….
  - …object is outside domain of AF
  - …object can't be mapped to *any* abstract value

Rules of Thumb

1. RI should be verifiable in checkRep()
   - If you can't write code to verify it, maybe it belongs in AF
2. RI is w.r.t a snapshot of the object
   - Corollary of #1
   - Only considers state of the object at that moment
   - Description of changes over time more likely belong in the AF

- Only rules of thumb – there are exceptions

Misconception #2

- You have an instance field like:

  ```
  Map<String, String> stuff;
  ```

- Rep invariant:

  stuff != null

  ~~no entry in stuff maps to a null value~~

- My class makes sure nulls are never added, so I don't need to put that in the RI, right?

RI should always be true

- You're sure your class won't add nulls? Great!
- But bugs are good at hiding!
  - Sure you haven't missed anything?
- And code changes
  - Sure someone won't introduce a bug later?
- Everything in your RI, you *expect* to be true
  - So what's the point?
  - Safeguard against bugs via checkRep()
  - Provide documentation for future developers

Abstraction Function

Concrete to abstract

- AF describes how to derive abstract value from instance fields
- To write the AF, ask yourself:

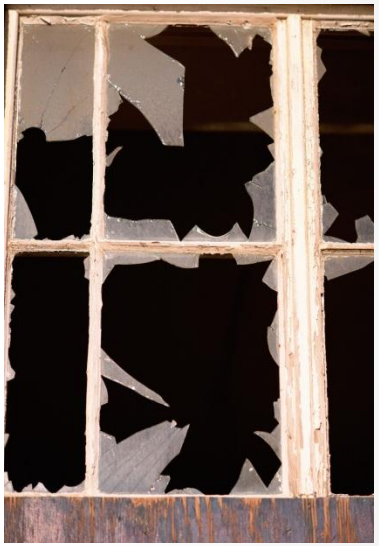
1. What does the client see?
   - What abstract fields, structure, or properties need to be defined?
2. For each item in #1: how is it stored in the concrete object?
   - If the client asked for it, what would you return?

- AF documents each item identified in #2

Representation Exposure

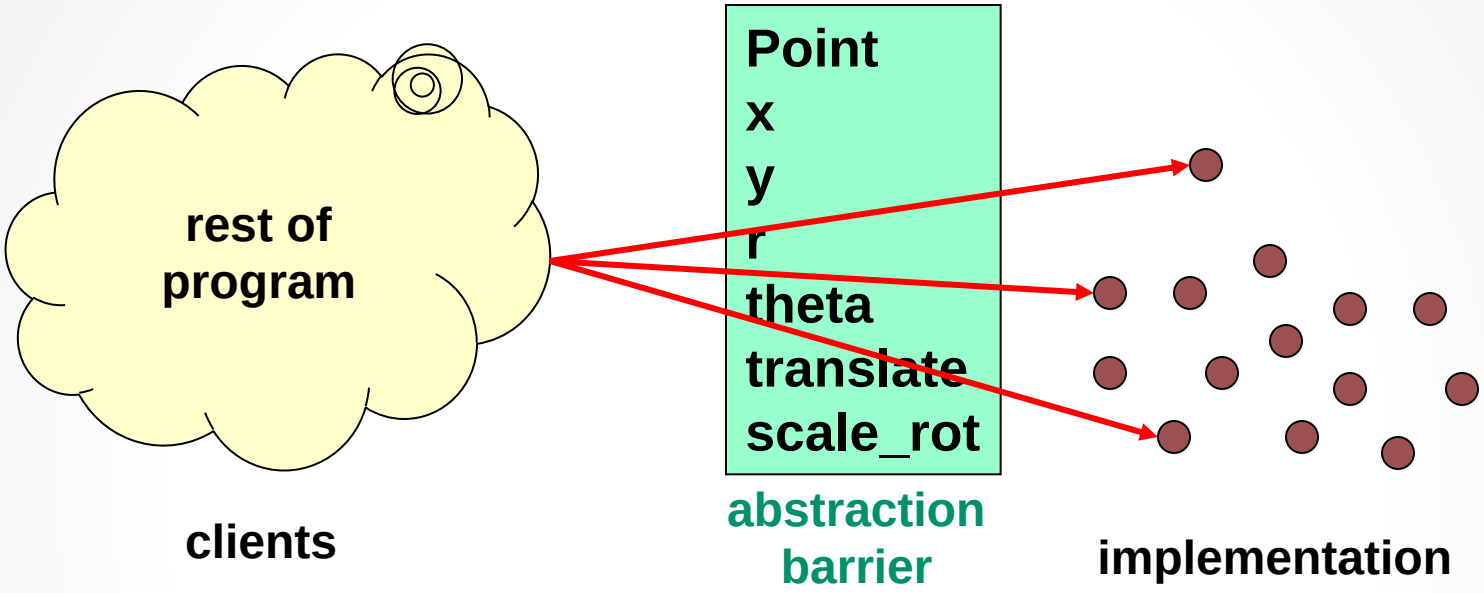
Representation Exposure

- Some object `x`, client that uses `x`
- Client gets a pointer to an object in `x`'s implementation
- Client can modify `x`'s representation directly – bypass public interface



It's like a broken window!

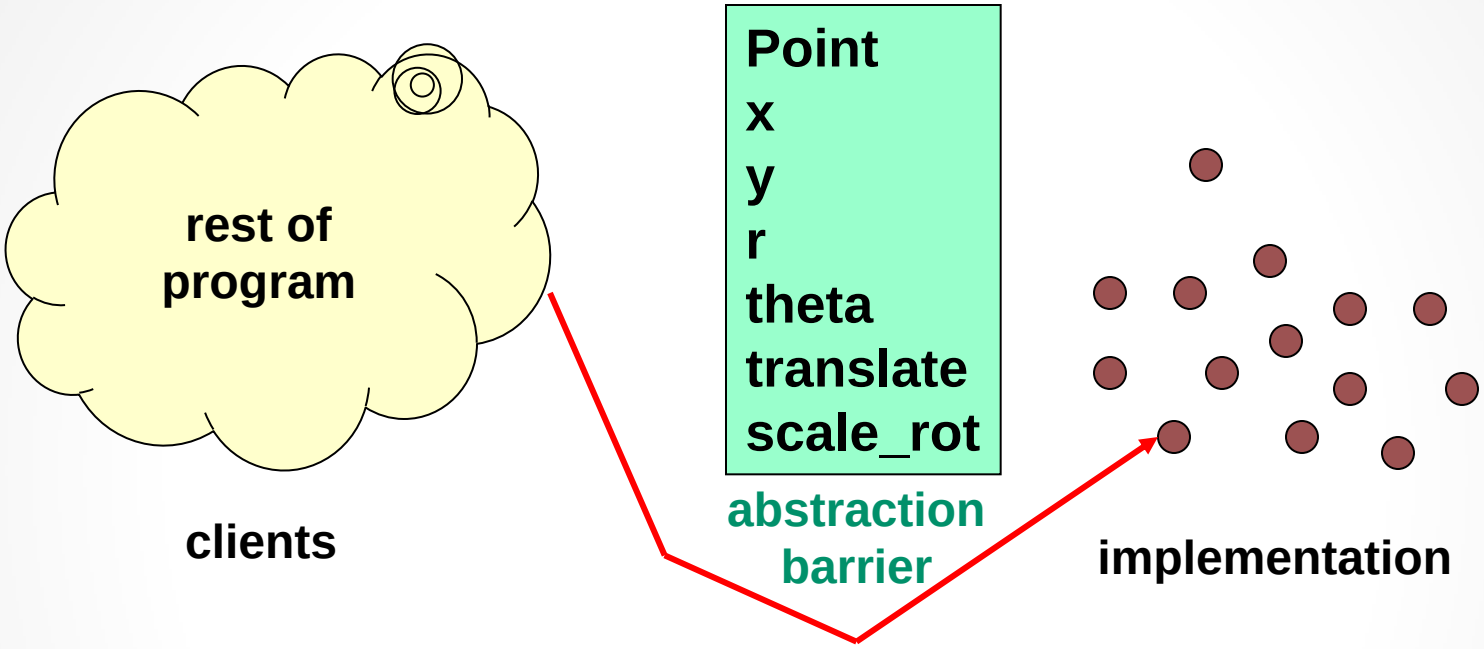
A correctly implemented class



The implementation is hidden

The only operations on objects of the type are those provided by the abstraction

With representation exposure



Client can bypass the abstraction barrier and modify the implementation directly

Representation Exposure

- Rep exposure happens…
  - When `x` returns an internal field
  - When `x` stores an object passed in by a client
- Rep exposure doesn't happen…
  - .. With primitives
  - With immutable objects
  - If `x` stores/returns *copies* of the object

Don't break your own window, yo.

Are references always bad?

- Not always
- Immutable objects – client can't do any harm
- Collections: client may want the original object back, not a copy
- Collections: copying every item can get expensive

Sharing is OK sometimes!