Verifying Object Construction

How to use the builder pattern with the type safety of constructors

Martin Kellogg[a], Manli Ran[b], Manu Sridharan[b], Martin Schäf[c], Michael D. Ernst[a,c]

[a]University of Washington [b]University of California, Riverside [c]Amazon Web Services

Object construction APIs

```
public class UserIdentity {
    private final String name;          // required
    private final int id;               // required
    private final String nickname;      // optional
}
```

Object construction APIs

```
public class UserIdentity {
    private final String name;          // required
    private final int id;               // required
    private final String nickname;      // optional
}

public UserIdentity(String name, int id);
public UserIdentity(String name, int id,
                      String nickname);
```

Object construction APIs

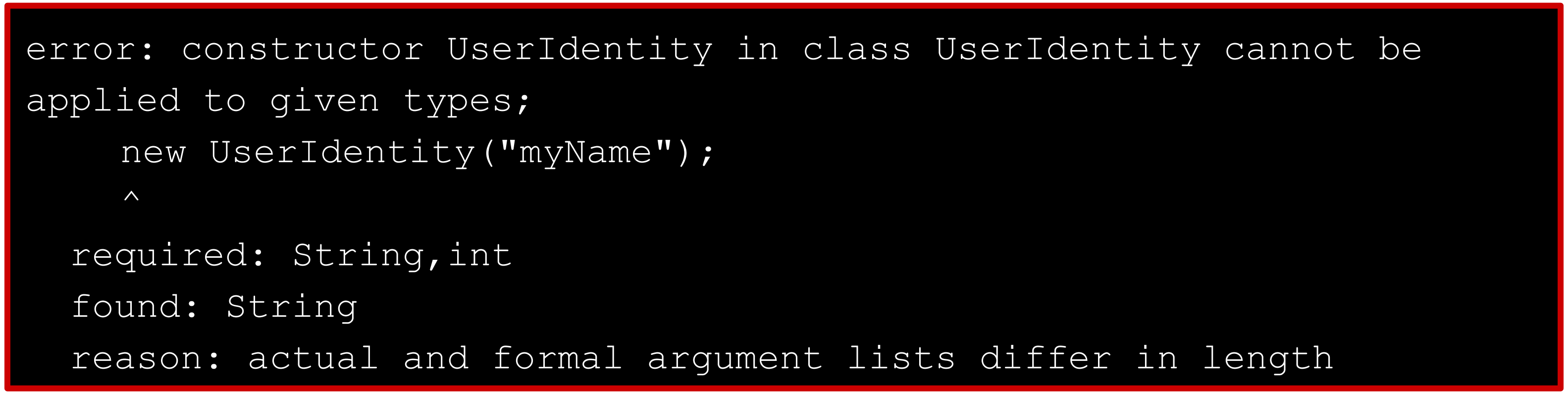
```
public UserIdentity(String name, int id);
public UserIdentity(String name, int id,
                            String nickname);

new UserIdentity("myName");
```

Object construction APIs

```
public UserIdentity(String name, int id);
public UserIdentity(String name, int id,
                                String nickname);


new UserIdentity(“myName”);
```



```
error: constructor UserIdentity in class UserIdentity cannot be
applied to given types;
    new UserIdentity("myName");
    ^
  required: String,int
  found: String
  reason: actual and formal argument lists differ in length
```

Pros and cons of constructors

+ compile-time verification that arguments are sensible

Pros and cons of constructors

+ compile-time verification that arguments are sensible

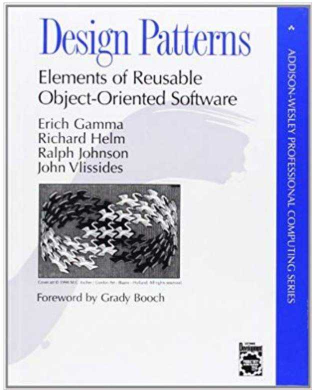
- user must define each by hand

- exponentially many in number of optional parameters

- arguments are positional (hard to read code)

The builder pattern

```
public class UserIdentity {
  public static UserIdentityBuilder builder();
  public class UserIdentityBuilder {
    public UserIdentityBuilder name();
    public UserIdentityBuilder id();
    public UserIdentityBuilder nickname();
    public UserIdentity build();
  }
  ...
}
```



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The builder pattern

```
UserIdentity identity = UserIdentity.builder()
                            .name(username)
                            .id(userId)
                            .build();
```

Pros and cons of the builder pattern

+ Flexible and easy to read
+ Frameworks implement automatically

The builder pattern

```
UserIdentity identity = UserIdentity.builder()
                                        .name(username)
                                        .id(userId)
                                        .build();
```

The builder pattern

```
UserIdentity identity = UserIdentity.builder()
                                   .name(username)
                                   .build();
```

Possible outcomes:

- Run-time error (bad!)
- Malformed object is used (worse!)

The builder pattern

```
UserIdentity identity = UserIdentity.builder()
                                     .name(username)
                                     .build();
```

Possible outcomes:

- Run-time error (bad!)
- Malformed object is used (worst!)

Pros and cons of the builder pattern

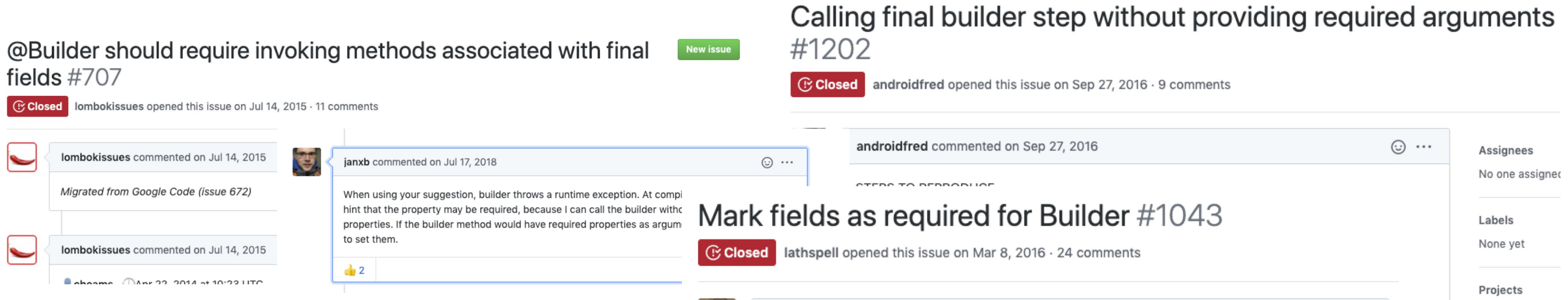
+ Flexible and easy to read
+ Frameworks implement automatically

- No guarantee that required arguments provided

Pros and cons of the builder pattern

+ Flexible and easy to read
+ Frameworks implement automatically

- No guarantee that required arguments provided


@Builder should require invoking methods associated with final fields #707

New issue

Closed lombokissues opened this issue on Jul 14, 2015 · 11 comments

lombokissues commented on Jul 14, 2015

Migrated from Google Code (issue 672)

lombokissues commented on Jul 14, 2015

janxb commented on Jul 17, 2018

When using your suggestion, builder throws a runtime exception. At compi
hint that the property may be required, because I can call the builder witho
properties. If the builder method would have required properties as argum
to set them.

👍 2

Calling final builder step without providing required arguments #1202

Closed androidfred opened this issue on Sep 27, 2016 · 9 comments

androidfred commented on Sep 27, 2016

Assignees

No one assigned

Labels

None yet

Projects

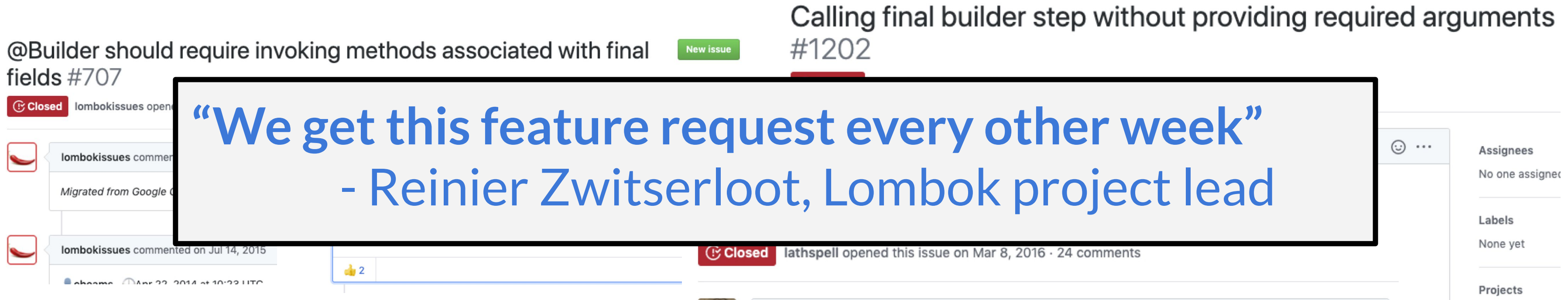
Mark fields as required for Builder #1043

Closed lathspell opened this issue on Mar 8, 2016 · 24 comments

Pros and cons of the builder pattern

+ Flexible and easy to read
+ Frameworks implement automatically

- No guarantee that required arguments provided


@Builder should require invoking methods associated with final fields #707

New issue

Closed lombokissues opene

lombokissues commen

Migrated from Google C

lombokissues commented on Jul 14, 2015

👍 2

Calling final builder step without providing required arguments #1202

Closed lathspell opened this issue on Mar 8, 2016 · 24 comments

Assignees

No one assignec

Labels

None yet

Projects

“We get this feature request every other week”

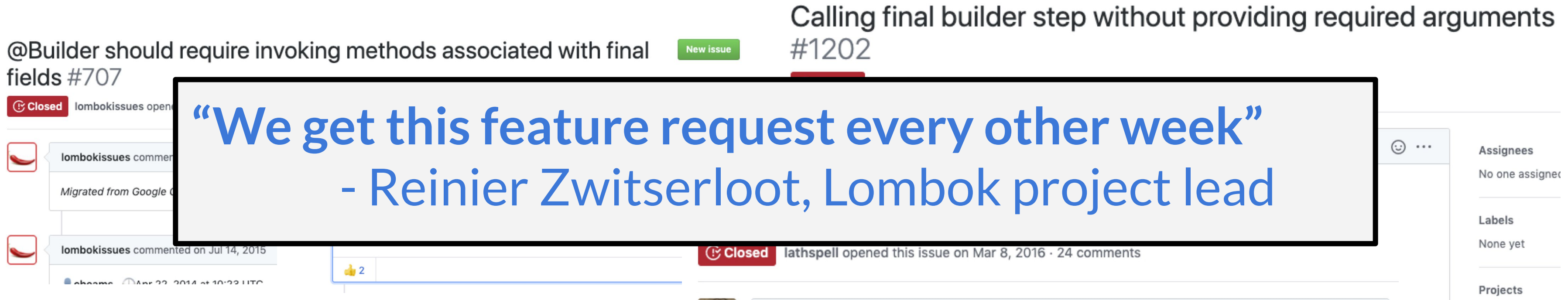
- Reinier Zwitserloot, Lombok project lead

Pros and cons of the builder pattern

+ Flexible and easy to read

Our approach:

- Provides **type safety** for uses of the builder pattern
- **Keeps advantages** of builder pattern vs. constructors



@Builder should require invoking methods associated with final fields #707

New issue

Closed lombokissues opene

lombokissues commen

Migrated from Google C

lombokissues commented on Jul 14, 2015

👍 2

Calling final builder step without providing required arguments #1202

Closed lathspell opened this issue on Mar 8, 2016 · 24 comments

Assignees

No one assigned

Labels

None yet

Projects

“We get this feature request every other week”

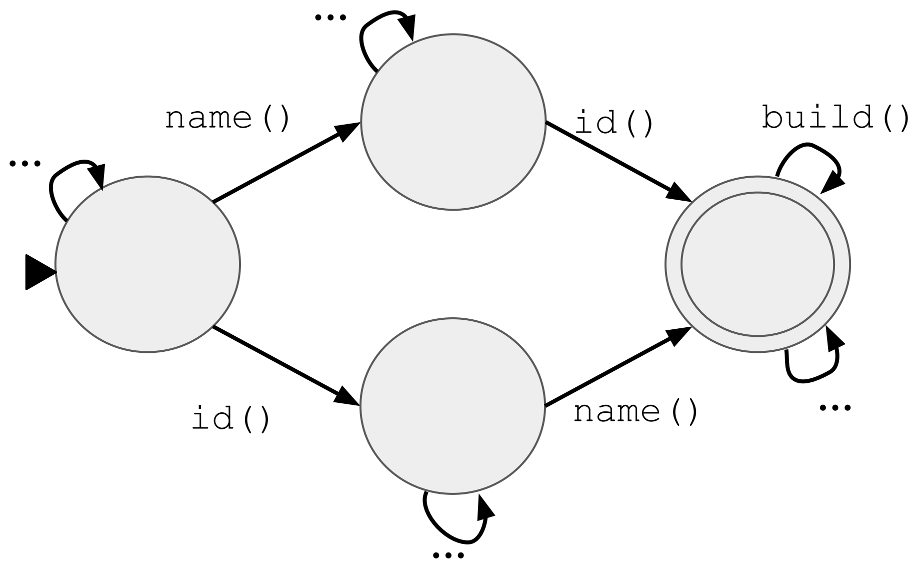
- Reinier Zwitserloot, Lombok project lead

Builder correctness as a typestate analysis

```
UserIdentity identity =
   UserIdentity.builder()
         .name(username)
         .id(userId)
         .build();
```

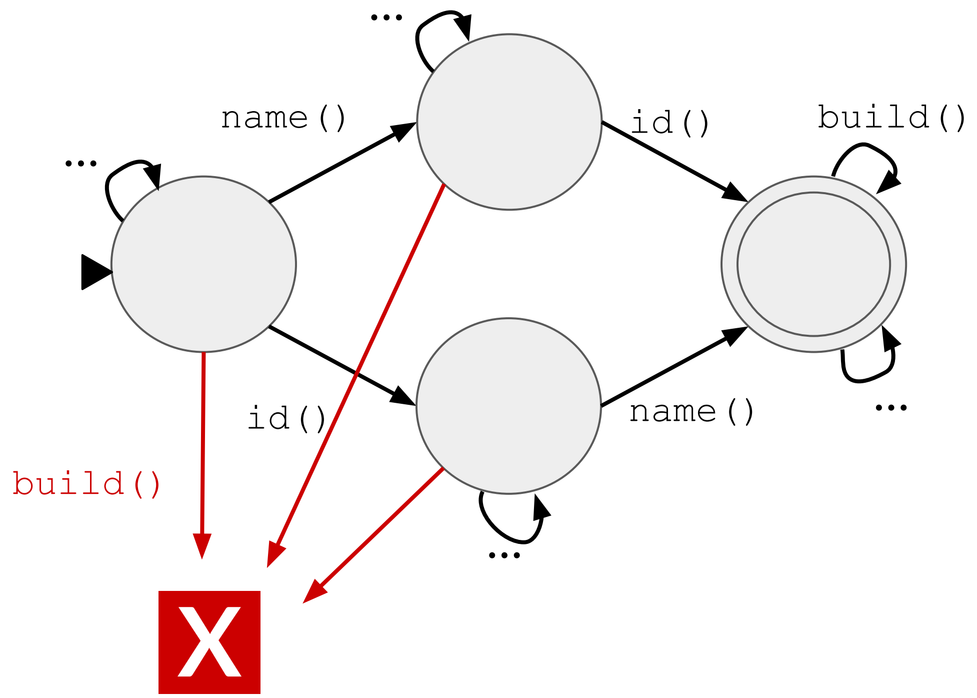
Builder correctness as a typestate analysis

```
UserIdentity identity =
   UserIdentity.builder()
         .name(username)
         .id(userId)
         .build();
```



Builder correctness as a typestate analysis

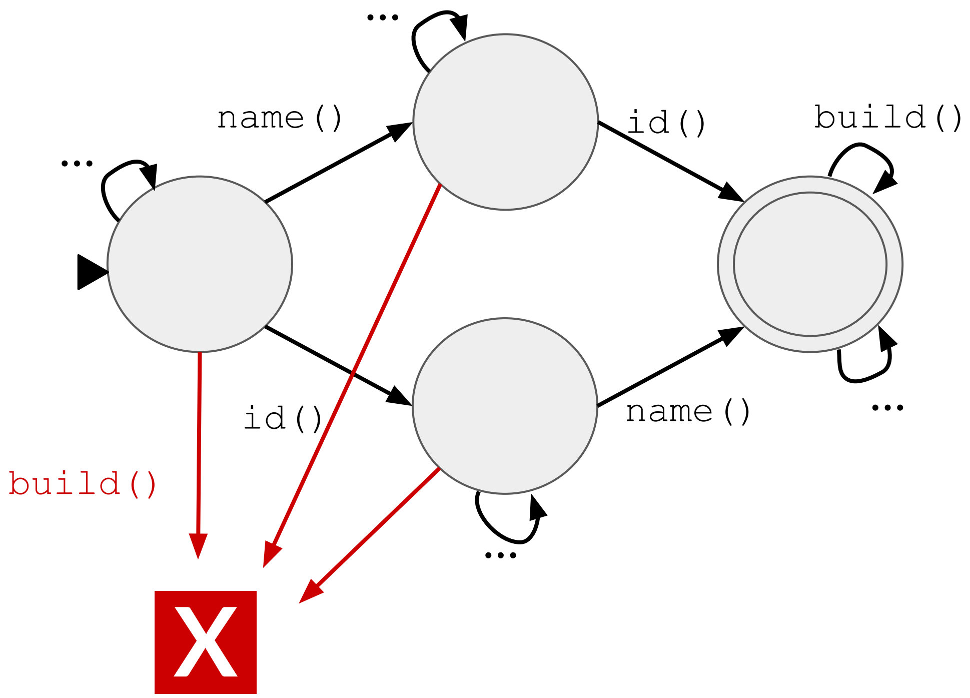
```
UserIdentity identity =
    UserIdentity.builder()
          .name(username)
          .id(userId)
          .build();
```



Builder correctness as a typestate analysis

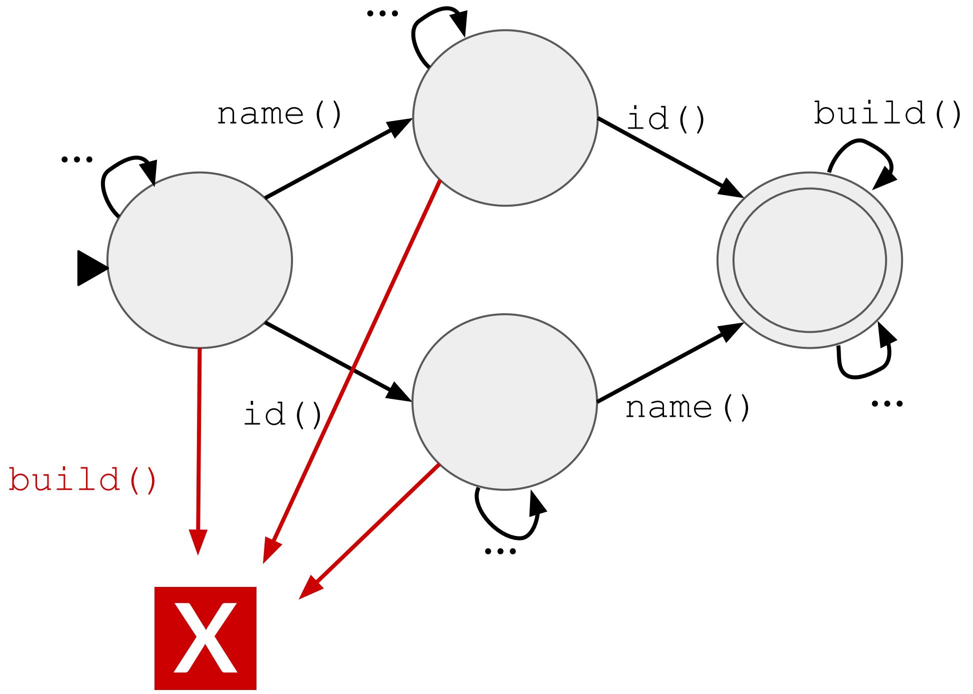
```
UserIdentity identity =
    UserIdentity.builder()
            .name(username)
            .id(userId)
            .build();
```

Problem:
Arbitrary typestate analysis is expensive: a whole-program alias analysis is required for soundness



Builder correctness as a typestate analysis

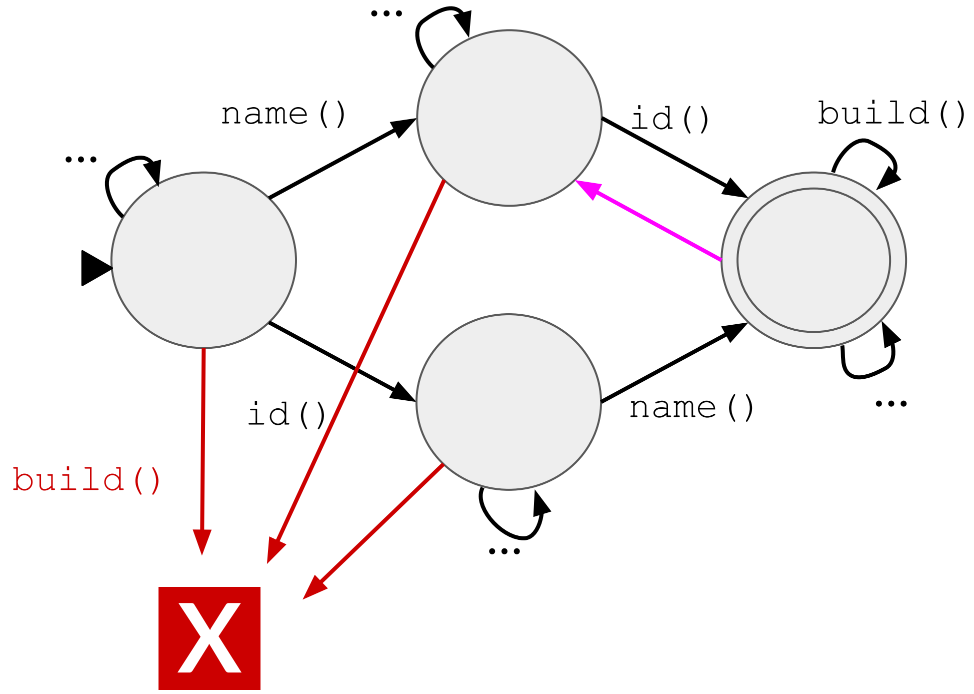
```
UserIdentity identity =
    UserIdentity.builder()
          .name(username)
          .id(userId)
          .build();
```



Key insight:
Transitions flow
in one direction!

Builder correctness as a typestate analysis

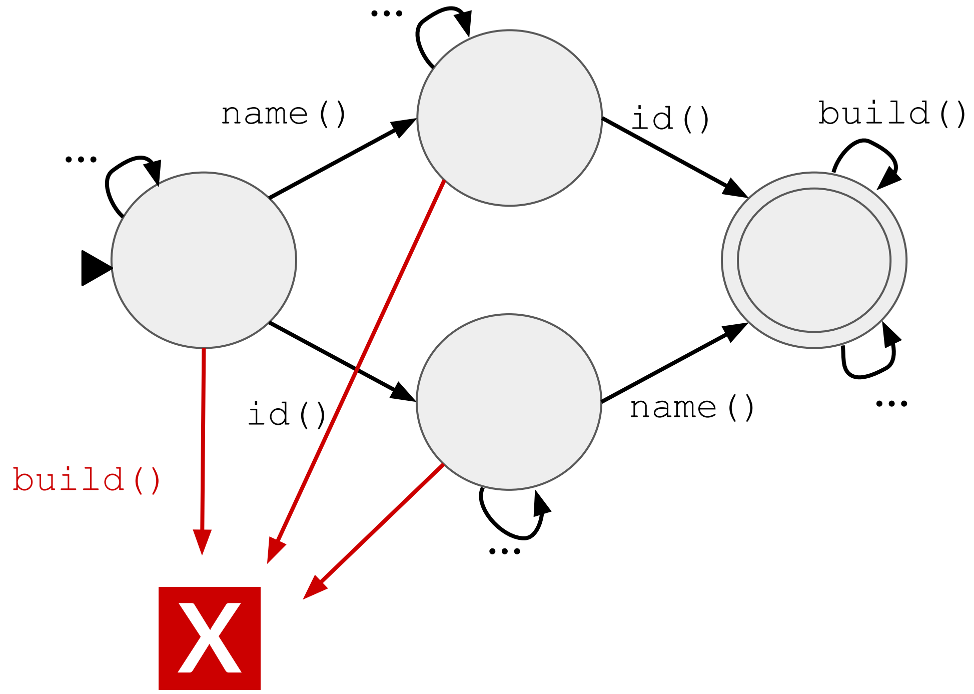
```
UserIdentity identity =
    UserIdentity.builder()
            .name(username)
            .id(userId)
            .build();
```



Key insight:
Transitions flow in one direction!

Builder correctness as a typestate analysis

```
UserIdentity identity =
    UserIdentity.builder()
          .name(username)
          .id(userId)
          .build();
```



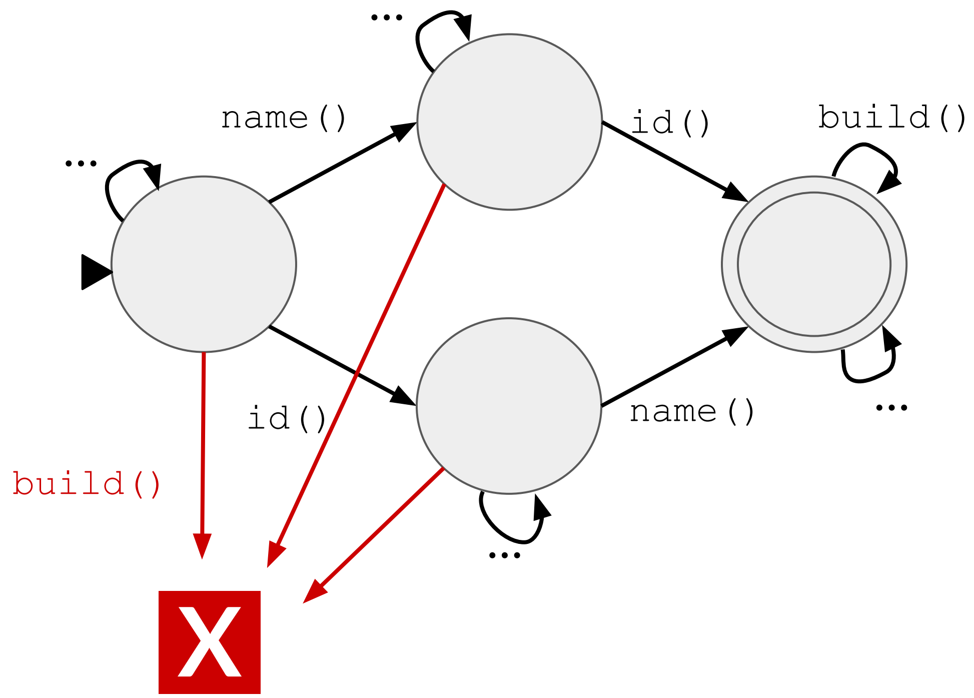
Key insight:
Transitions flow
in one direction!

accumulation
Builder correctness as a ~~typestate~~ analysis

```
UserIdentity identity =
    UserIdentity.builder()
          .name(username)
          .id(userId)
          .build();
```

“accumulation analysis”

Key insight:
Transitions flow in one direction!



Advantages of accumulation analysis

- always safe to under-approximate

Advantages of accumulation analysis

- always safe to under-approximate

  $\hookrightarrow$ does not require alias analysis for soundness

Advantages of accumulation analysis

- always safe to under-approximate
  - $\hookrightarrow$ does not require alias analysis for soundness
- can be implemented modularly (e.g., as a type system)

Advantages of a type system

- provides guarantees
- no alias analysis + modular ⇒ scalable
- type inference reduces need for annotations

`build()`'s specification

```
build(@CalledMethods({"name", "id"})
      UserIdentityBuilder this);
```

Results (1 of 3): security vulnerabilities

Lines of code	9.1M
Vulnerabilities found	16
False warnings	3
Annotations	34

Contributions

- **Static safety** of constructors with flexibility of **builders**
- ***Accumulation analysis***: special case of typestate
    - Does not require whole-program alias analysis

https://github.com/kelloggm/object-construction-checker

Accumulation doesn't need alias analysis

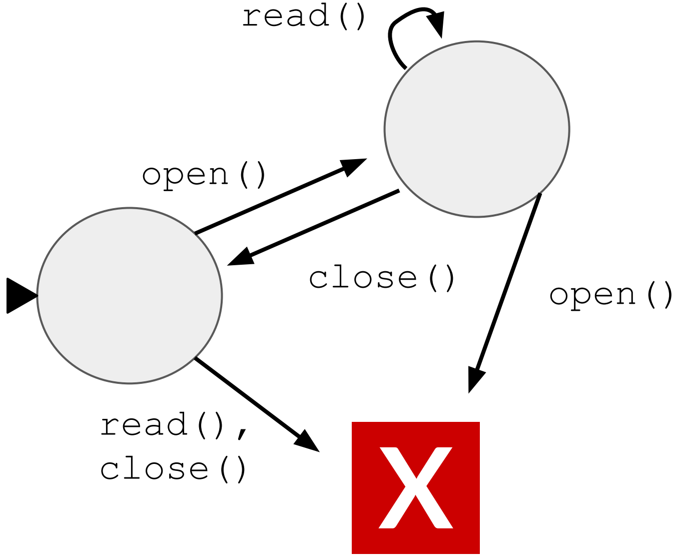
```
UserIdentityBuilder b = UserIdentity.builder();
b.name(username);
UserIdentityBuilder b2 = b;
b2.id(userId)
UserIdentity identity = b.build();
```

Accumulation doesn’t need alias analysis

```
UserIdentityBuilder b = UserIdentity.builder();
b.name(username);
UserIdentityBuilder b2 = b;
b2.id(userId)
UserIdentity identity = $\boxed{b.build();}$
```

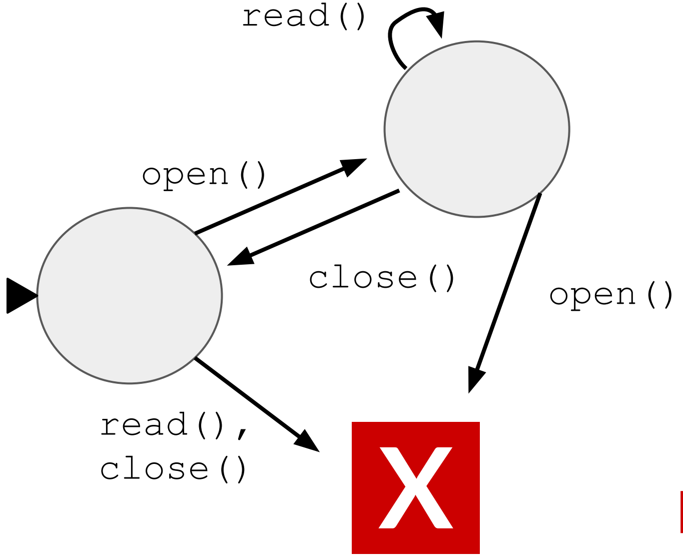
False positive here is worst-case scenario

Why typestate needs alias analysis



```
File f = …;
f.open();
File f2 = f;
f.close();
f2.read();
```

Why typestate needs alias analysis



```
File f = …;
f.open();
File f2 = f;
f.close();
f2.read();
```

No alias analysis leads to false negative

Example: Netflix/SimianArmy

```
public List<Image> describeImages(String... imageIds) {
    DescribeImagesRequest request =
            new DescribeImagesRequest();

    if (imageIds != null) {
        request.setImageIds(Arrays.asList(imageIds));
    }

    DescribeImagesResult result =
            ec2client.describeImages(request);

    return result.getImages();
}
```

The builder pattern

```
@Builder
public class UserIdentity {
    private final String name;          // required
    private final int id;               // required
    private final String nickname;      // optional
}
```

The builder pattern

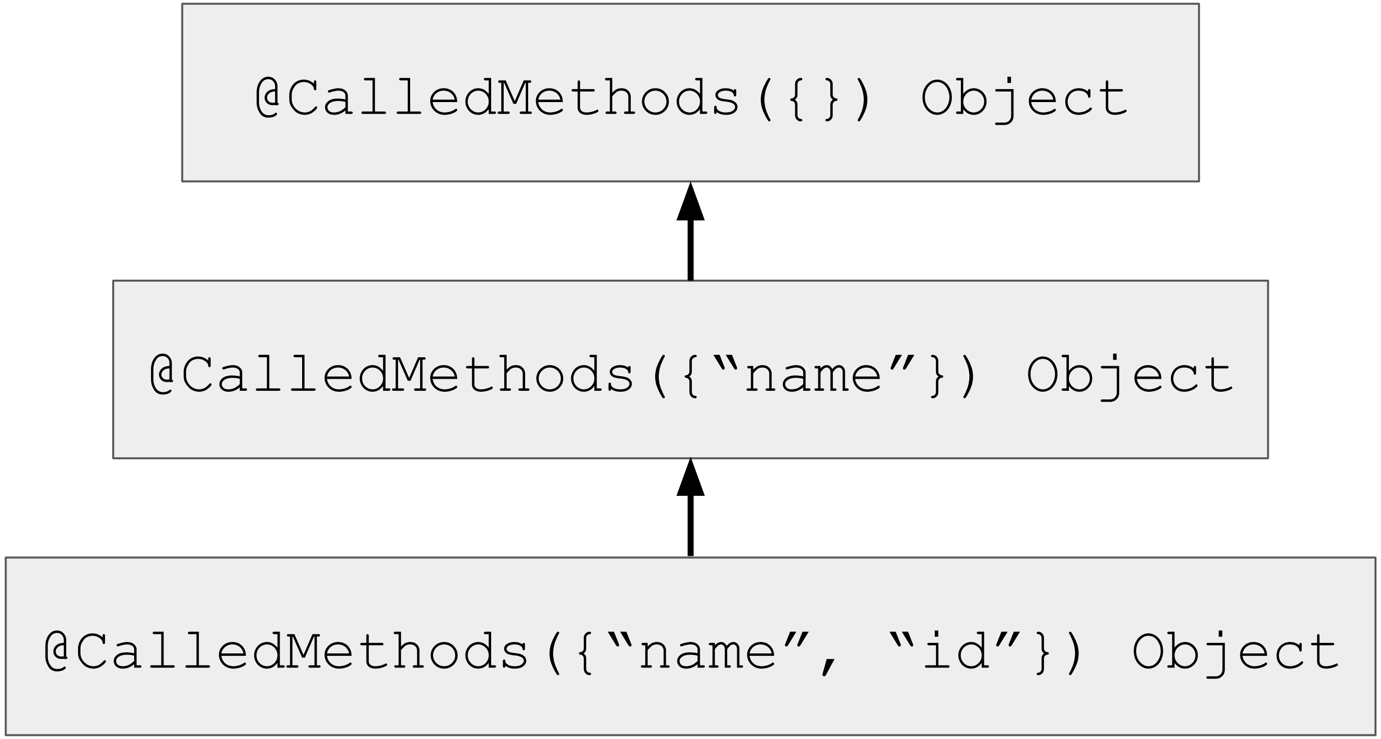
```
@Builder
public class UserIdentity {
    private final @NonNull String name;
    private final @NonNull int id;
    private final String nickname;    // optional
}
```

The builder pattern

```
@Builder
public class UserIdentity {
    private final @NonNull String name;
    private final @NonNull int id;
    private final String nickname;     // optional
}

UserIdentity identity = UserIdentity.builder()
                            .name(username)
                            .id(userId)
                            .build();
```

Type hierarchy


@CalledMethods({}) Object
@CalledMethods({“name”}) Object
@CalledMethods({“name”, “id”}) Object

What’s the type of `b`?

```
UserIdentityBuilder b = UserIdentity.builder();

b.name(username);

b.id(userId)

UserIdentity identity = b.build();
```

What's the type of `b`?

@CalledMethods({})

```
UserIdentityBuilder b = UserIdentity.builder();

b.name(username);

b.id(userId)

UserIdentity identity = b.build();
```

What’s the type of `b`?

```
@CalledMethods({})
```

```
UserIdentityBuilder b = UserIdentity.builder();
```

```
@CalledMethods({“name”})
```

```
b.name(username);

b.id(userId)

UserIdentity identity = b.build();
```

What’s the type of `b`?

```
                                  @CalledMethods({})

UserIdentityBuilder b = UserIdentity.builder();

                                  @CalledMethods({“name”})
b.name(username);

                                  @CalledMethods({“name”, “id”})
b.id(userId)

UserIdentity identity = b.build();
```

Fluent APIs and receiver aliasing

```
UserIdentity identity = UserIdentity.builder()
                                       .name(username)
                                       .id(userId)
                                       .build();
```

Fluent APIs and receiver aliasing

```
UserIdentity identity = UserIdentity.builder()
                                       .name(username)
                                       .id(userId)
                                       .build();
```

```
@CalledMethods({“id”})
```

Fluent APIs and receiver aliasing

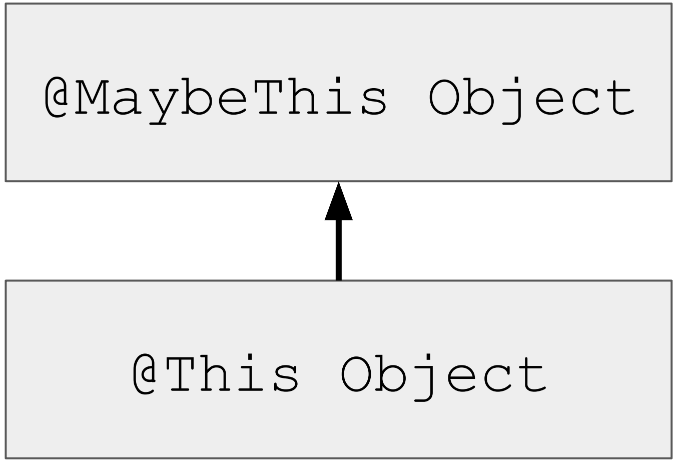
```
UserIdentity identity = UserIdentity.builder()
                                      .name(username)
                                      .id(userId)
                                      .build();
```

```
@CalledMethods({“id”})
```

How do we know that the **return type** of `id()` is the **same object** that `name()` was called on?

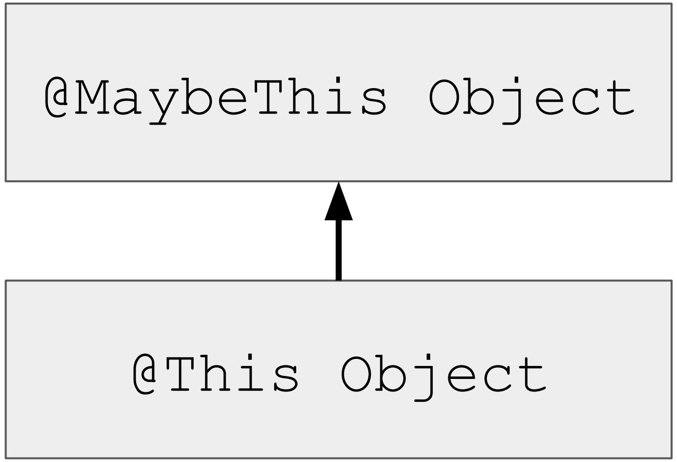
Returns receiver checking

A special case of aliasing, needed for **precision**!



Returns receiver checking

A special case of aliasing, needed for **precision**!



```
class UserIdentityBuilder {
  @This UserIdentityBuilder name();
  @This UserIdentityBuilder id();
}
```

Showing correct code is safe

```
UserIdentity identity = UserIdentity.builder()
                                        .name(username)
                                        .id(userId)
                                        .build();
```

Showing correct code is safe

```
UserIdentity identity = UserIdentity.builder()
                          .name(username)
                          .id(userId)
                          .build();
```

Accumulate more “called methods”

Results (2 of 3): Lombok user study

6 industrial developers with Java + Lombok experience

Task: add a new `@NonNull` field to a builder, and update all call sites

Results:

- 6/6 succeeded with our tool, only 3/6 without
- Those who succeeded at both 1.5x faster with our tool
- *"It was easier to have the tool report issues at compile time"*

Results (3 of 3): case studies

5 projects: 2 Lombok, 3 AutoValue (~200k sloc)

653 calls verified, 1 true positive (google/gapic-generator)

131 annotations, 14 false positives

"your static analysis tool sounds truly amazing!"
- gapic-generator engineer