Writing Easy-To-Change Code

Your second-most important goal as a developer

Session 112

Ken Kocienda
Principal Engineer, iOS Apps and Frameworks

This is a talk about writing code

Easy to… read

… learn

… understand

… maintain

… change

Easy to change software

Your second-most important goal

What is your most important goal?

Ship products!

This is how we think at Apple

Over 30 iOS releases since 2007

Releases are complicated

Improving existing features

Too few people

Testing

Legal

Too many people

New hardware

Marketing

App Store submission

Work with other companies

Tight schedules

Competition

New OS features

New app features

Bug fixes

Changing priorities

Help you make change easier

You always change your software

What Kinds of Change?

- Bug fixes
- Adding new features
- Enhancing existing features
- Changing code someone else wrote
- Changing code you wrote six months ago

General conventions

Mac and iOS conventions

Topics

Things to think about

Topics

- Style
- Stories
- Laziness
- Hygiene
- Notifications
- Optimization
- Dependencies
- Mixing
- Expectations
- Wrap up

Style

More than skin deep

Coding conventions

Coding Conventions

- Brace style for `if-else`
- Parenthesis Style
- Leading underscores
- Code indenting
- CapitalizationStyle (i.e. `capitalization_style`)

Local consistency is important

The beginnings of style

Style goes deeper

"People think that I can teach them style. What stuff it all is! Have something to say, and say it as clearly as you can. That is the only secret of style."

Matthew Arnold

Clarity

Clear writing is easier to understand

Clear code is easier to change

Elements of a clear coding style?

Good names

Common idioms

Good names

Common idioms

Good Names Are Descriptive

```
NSString *searchString = [self _searchString];
BOOL searchStringIsNotEmpty = [searchString length] != 0;

if (searchStringIsNotEmpty) {
    [self _findBanner]->findString(searchString,
        shouldBeep ? BeepOnFailure : DoNotBeepOnFailure);
}
```

Good Names Are Descriptive

```
NSString *searchString = [self _searchString];
BOOL searchStringIsNotEmpty = [searchString length] != 0;

if (searchStringIsNotEmpty) {
    [self _findBanner]->findString(searchString,
        shouldBeep ? YES : NO);
}
```

Descriptive Names

You can go overboard

```
@interface YesYouCanMakeNamesForClassesWhichAreTooLong
{
    id _aReallyVerboseNameJustToBePerfectlyClear;
}
@end
```

Bad Names? Boolean Arguments

Hard to know what they mean

```
[magnifier stopMagnifying:NO];
```

Bad Names? Boolean Arguments

Hard to know what they mean

```
- (void)stopMagnifying:(BOOL)animated;
```

Bad Names? Boolean Arguments

Hard to know what they mean

```
- (void)stopMagnifyingAnimated:(BOOL)animated;
```

Good names are descriptive

Good names

Common idioms

- ✔ Good names
- Common idioms

Workhorse Lines of Code

Hard to know what they mean

```
[_rightView setAlpha:![[_temporary text] length] ? 1.0 : 0.0];
```

Count square brackets?

Workhorse Lines of Code

Hard to know what they mean

```
[_rightView setAlpha:![[_temporary text] length] ? 1.0 : 0.0];
```

Rewrite Workhorse Lines of Code

Be clear!

```
BOOL textIsEmpty = [_temporary.text length] == 0;
float alpha = textIsEmpty ? 1.0 : 0.0;
[_rightView setAlpha:alpha];
```

Read and understand quickly

Design patterns

Design Patterns

Very common patterns

- Singleton
- Observer
- Prototype
- Chain of responsibility
- Command

Design Patterns

Patterns used in Apple frameworks

- MVC
- Target-action
- Delegation
- Autorelease
- View controller

Idioms communicate at a high level

Shared vocabulary

- ✓ Good names
- ✓ Common idioms

Style

More than skin deep

Stories

Now I understand

Bug. Why?

Did not anticipate

Did not understand

Debug

“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?”

Brian Kernighan

Step 1: Debugger

What are you really looking for?

Think

Step 1: ~~Debugger~~

How could this bug happen?

“The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.”

Brian Kernighan

Debugging is understanding

Debugging is not jiggling code

`-performSelector:withObject:afterDelay:`

Rarely right!

```
[self foo];
[self bar];
```

Why?

Each bug fix should tell a story

Investigate. Eureka!

Tell someone before you code the fix

Tell the story during code review

Write the story into your bug tracker

Anticipate more

Understand better

Stories

Now I understand

Laziness

Wake me when it is over

Lazy initialization

It is good

It is not magic

Singleton Objects

Lazy initialization is common

```
FooController *controller = [FooController sharedInstance];
```

Singleton Objects

Lazy initialization is common

```
@implementation FooController

+ (FooController *)sharedInstance
{
    static dispatch_once_t once;
    static FooController *instance;
    dispatch_once(&once, ^{
        instance = [[FooController alloc] init];
    });
    return instance;
}

@end
```

Singleton Objects

Lazy initialization is common

```
@implementation FooController (Continued)

- (id)init
{
     ...
    BarController *barController = [BarController
      sharedInstance];
    ...
}

@end
```

Singleton Objects

Lazy initialization is common

```
@implementation BarController

- (id)init
{
    ...
    FooController *fooController = [FooController
      sharedInstance];
    ...
}

@end
```

Init storm

Several problems

Long pause

Order of initialization

Singleton Objects

How many do you have?

```
@implementation FooController

+ (FooController *)sharedInstance
{
    static dispatch_once_t once;
    static FooController *instance;
    dispatch_once(&once, ^{
        instance = [[FooController alloc] init];
    });
    return instance;
}

@end
```

Singleton Objects

How many do you have?

```
@implementation FooController

+ (FooController *)sharedInstance
{
    static FooController *instance;
    if (!instance)
       instance = [[FooController alloc] init];
    return instance;
}

@end
```

Multiple instantiation of *singleton*

Mess

Think through lazy initialization

No silver bullets

Lightweight alloc at program start

Better singleton decomposition

Alternative accessor patterns

Alternative Accessor Patterns

Create or not?

```
@interface FooController

+ (FooController *)sharedInstance; // will create
+ (FooController *)activeInstance; // won’t create
+ (FooController *)sharedInstance
     createIfNeeded:(BOOL)createIfNeeded;

@end
```

Laziness

Wake me when it is over

Hygiene

You make the mess… you clean it up!

Good hygiene takes effort

“The best writing is rewriting.”

E.B. White

Do not throw away code

Conflict?

Changes are part of a process

Your top priority should be to ship

Do not rewrite… refactor

Refactoring

Keep functionality, but change form

What about cruft?

Cruft is not…

…code you do not understand

…code you did not write

…code you do not like

What is genuine cruft?

What Is Genuine Cruft?

- Dead code
- Comments which no longer apply
- There is no number three

Use compiler for dead code checks

Delete or check old comments

Accumulated knowledge

Size of change is important

Small: clean up as you go

Medium: need coordination

Large: need real planning

Beware of regressions

Test!

Hygiene

You make the mess… you clean it up!

Notifications

Open the window and holler!

`goto`

Notifications are a glorified `goto`

You do not even say where to go!

You can go to more than one place!

Frustrate code inspection

You can not see what code will run

Non-deterministic behavior

Callbacks are unordered

Notifications can complicate change

Not all bad

Notifications promote loose coupling

Model/View/Controller (MVC)

CoreData

`will/did`

You should know about endpoints

Think twice about other uses

Code can be too loosely coupled

Consider protocols or delegates

```
goto NextTopic;
```

Optimization

The 3% solution

“We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.”

Donald Knuth

…97% of the time…

100% - 97% = 3%

Which 3% to worry about?

Things Which Can Be Slow

- Memory allocation
- View creation
- Drawing
- Questionable algorithms
- Questionable data structures
- I/O
- Blocking on information
- Unnecessary work
- New work you just added

Optimize when you have measured

Use Instruments

Optimize when you understand

Optimize code with clearest role

Optimize *slowest* and *oldest* 3%

Keep new code easiest to change

Trades are OK!

Never make the program slower

Change, test, measure, optimize

Optimization

The 3% solution

Dependencies

“Don’t call us… we’ll call you”

Implications of change

Limit collateral damage

Inheritance trees

Call graphs

Inheritance trees

Call graphs

Shallow is better

Avoid layers of overridden methods

Use delegation

Delegation

- Customize by calling another object
- Keeps conceptual overhead small
- Vary customization at runtime as needed

✔ Inheritance trees

Call graphs

Smaller is better

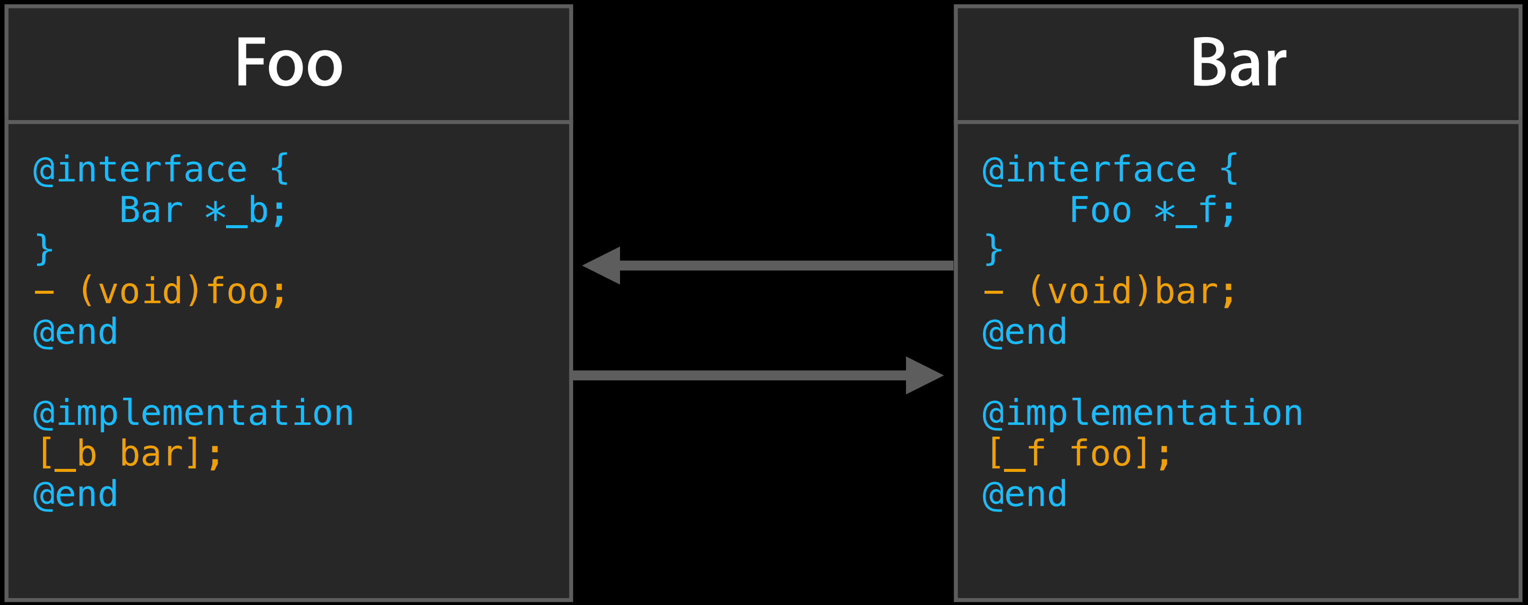
Limit includes

You get faster compile times

Strive for unidirectional calling

Bidirectional Call Graph

We are all friends here

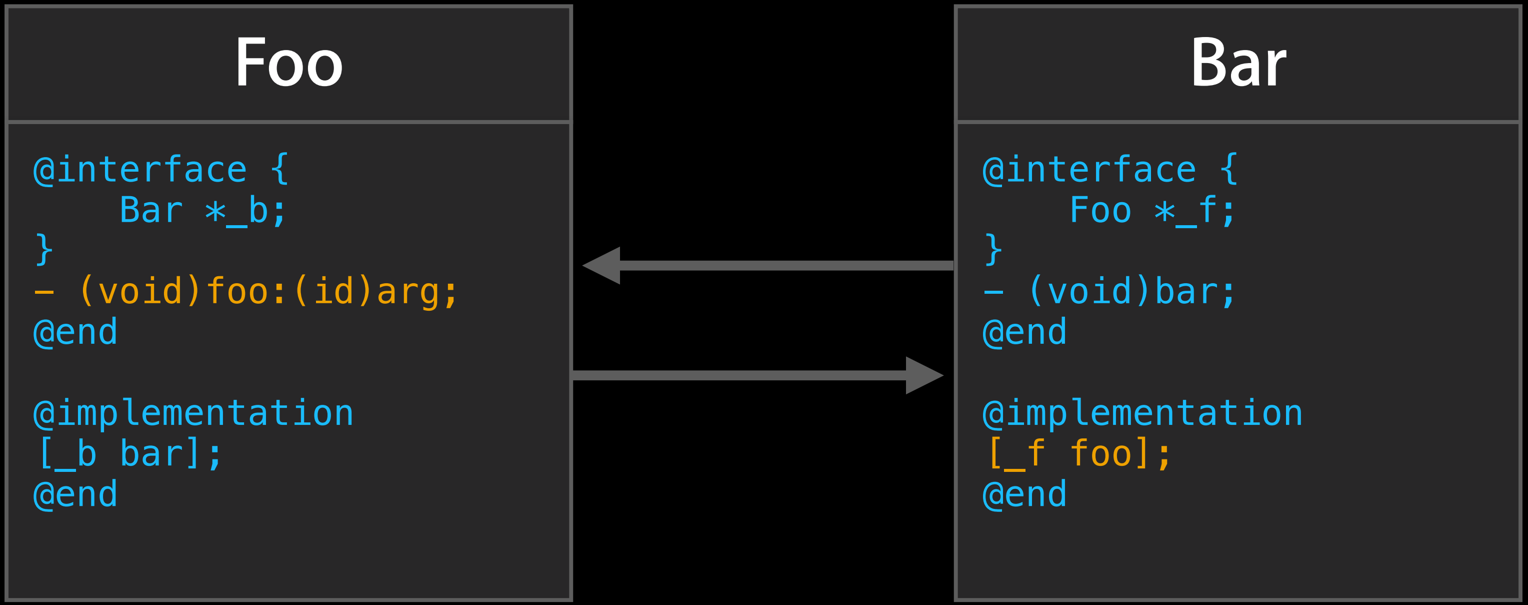


```
@interface {
    Bar *_b;
}
- (void)foo;
@end

@implementation
[_b bar];
@end
```

```
@interface {
    Foo *_f;
}
- (void)bar;
@end

@implementation
[_f foo];
@end
```

Bidirectional Call Graph

We are all friends here

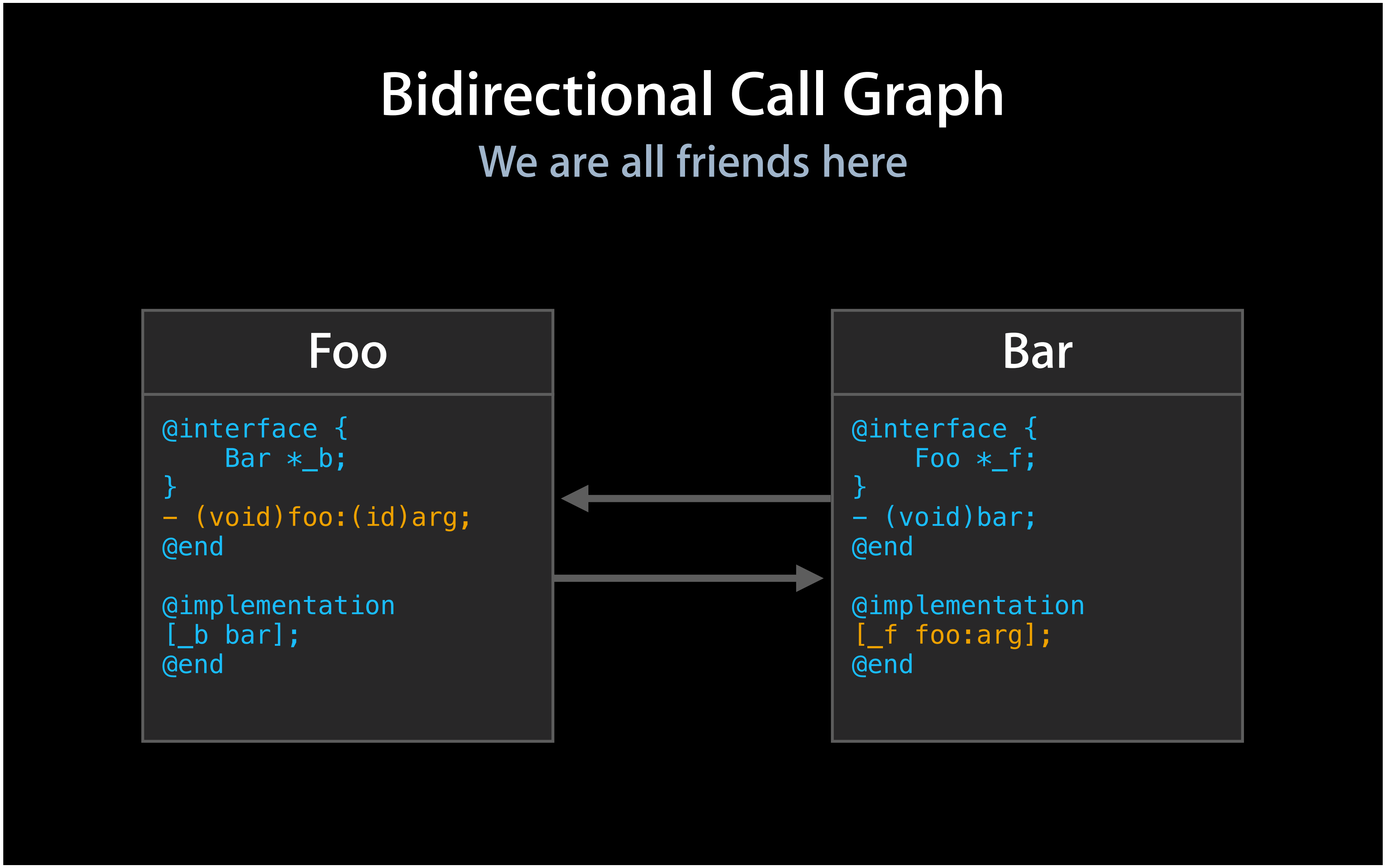


Foo

```
@interface {
    Bar *_b;
}
- (void)foo:(id)arg;
@end

@implementation
[_b bar];
@end
```

Bar

```
@interface {
    Foo *_f;
}
- (void)bar;
@end

@implementation
[_f foo];
@end
```

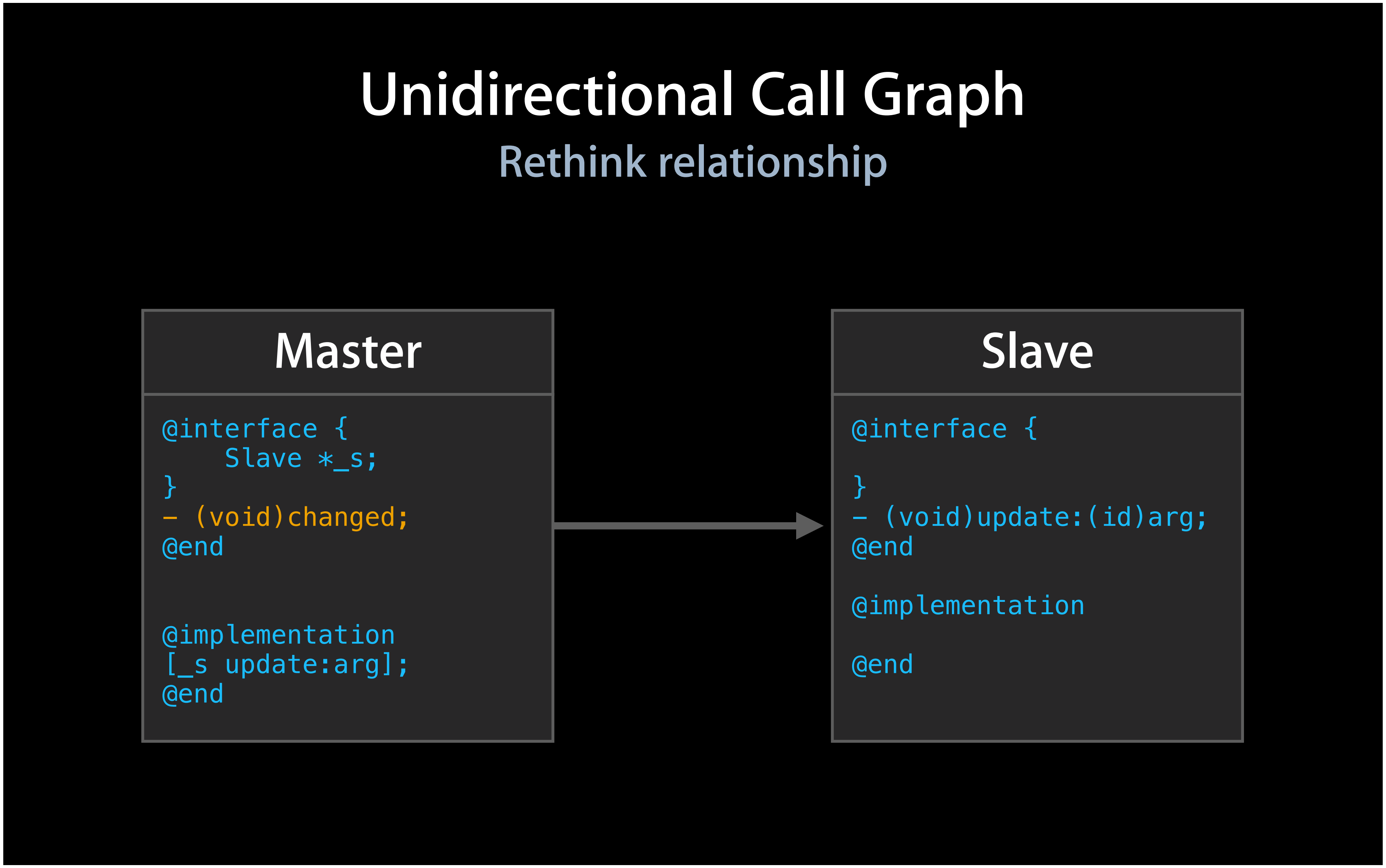


Bidirectional Call Graph

We are all friends here

```
@interface {
    Bar *_b;
}
- (void)foo:(id)arg;
@end

@implementation
[_b bar];
@end
```

```
@interface {
    Foo *_f;
}
- (void)bar;
@end

@implementation
[_f foo:arg];
@end
```



Unidirectional Call Graph

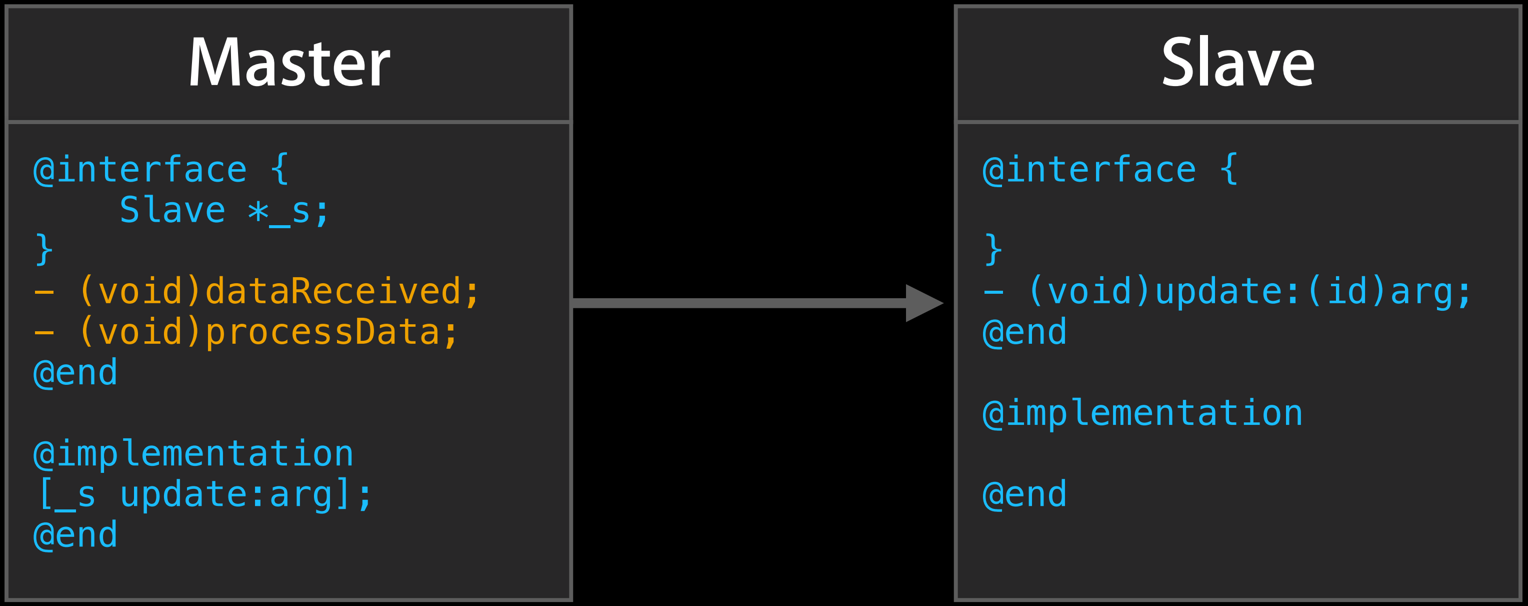
Rethink relationship

```
@interface {
    Slave *_s;
}
- (void)changed;
@end


@implementation
[_s update:arg];
@end
```

```
@interface {

}
- (void)update:(id)arg;
@end

@implementation

@end
```

Unidirectional Call Graph

Rethink relationship



```
@interface {
    Slave *_s;
}
- (void)dataReceived;
- (void)processData;
@end

@implementation
[_s update:arg];
@end
```

```
@interface {

}
- (void)update:(id)arg;
@end

@implementation

@end
```

- ✔ Inheritance trees
- ✔ Call graphs

Dependencies

“Don’t call us… we’ll call you”

Mixing

Purity of essence (OPE)

Model/View/Controller (MVC)

Do not mix model and view changes

Do not mix different things

Computation and I/O

Algorithms and data sources

UI and a specific screen resolution

UX and an interface paradigm

Conflicted About Animation Arguments

Is this mixing too much?

```
- (void)setEditing:(BOOL)editing animated:(BOOL)animated;
```

Hard-code animations?

Multitasking gestures

We change how the system works

App launch without animating… hard

Do not mix different things

Mixing

Purity of essence (OPE)

Expectations

How do I work this thing?

Bugs are often disappointments

I expected A, you did B

“Be conservative in what you send;
be liberal in what you accept.”

Jon Postel

Hard to use wrong

Method arguments

Assertions and early returns

Assertions

This will never work

```
// UIActionSheet.m

- (void)showInView:(UIView *)view
{
    NSParameterAssert(view != nil);

    ...
}
```

Early Returns

The method will not run right now

```
- (void)beginWork
{
    if (AlreadyBusy())
        return;
    ...
}
```

What about ivars?

Global variables are bad, right?

Scope is too broad

ivar scope also can be too broad

Rules of Thumb for ivars

- As few as possible
- Simple life-cycles
- Avoid tight relationships
- Avoid letting non-setter methods change ivars

Hard to manage ivar state?

Use a state machine

UIGestureRecognizers

Multitasking gestures

State Machines

How do they help?

- States help to think things through
- States help to limit possibilities
- States help to make assertions
- Need to add a feature?
  - Add a state
  - Handle the transitions

Hard to use wrong

Expectations

How do I work this thing?

Wrap Up

Ten things to think about

Easy To Change Code

Ten things to think about

1. Write clear code
2. Bug fixes should tell a story
3. Keep control of lazy initialization
4. Refactor instead of rewriting
5. Use notifications for the right things

Easy To Change Code, (Cont.)

Ten things to think about

6. Keep new code easy to change
7. Optimize slowest and oldest code
8. Limit dependencies
9. Do not mix different things
10. Make code that is hard to use wrong

WWDC2011