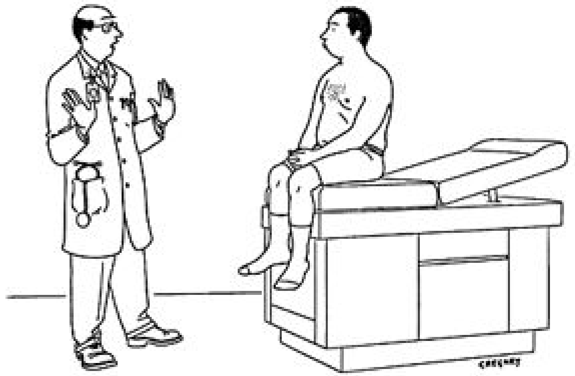
Information Hiding



"Whoa—<u>way</u> too much information!"

Too Much Information

```
- (void)dialogDidClose:(id)sender {
    if (gcontext(userInfo).energy >= appConstants(levelEnergyCost)) {
        [VC(GameViewController) restart];
    }
}
```

- Dependent on userInfo and appConstants
- Non-UI logic is mixed in with UI logic
- Code is duplicated in other places

Too Much Information

```
- (void)dialogDidClose:(id)sender {
   [VC(GameViewController) restartIfNeeded];
}
```

- Fewer dependencies
- Logic is pushed to controller
- Code is shared

Too Much Information

```
- (void)dialogDidClose:(id)sender {
    if ([VC(GameViewController) isRestartNeeded]) {
        [VC(GameViewController) restart];
    }
}
```

- Another solution, but exposes unnecessary details
  - Is `isRestartNeeded` used anywhere else?
  - What’s the advantage of exposing it?

Too Much Information

```
[self layoutIfNeeded];


if (self.needsLayout) {
    [self layoutSubviews];
}
```

- Apple often uses the first ‘if needed’ pattern
- Fewer details exposed that aren’t used elsewhere
- No logic needed, not even an if statement

Too Many Chains

```
function handleEvent(foo) {
    foo.bar().baz().qux();
}
```

- Tightly coupled to specific objects
- Any change to structure propagates
- Creates a brittle structure

Too Many Chains

```
function handleEvent(foo) {
    foo.qux();
}
```

- Hides the details of getting information
- A simpler interface (less to remember)
- A form of delegation, simplifying the API

Hiding Information

Self contained objects:

- Have private properties
- Don’t expose internal logic
- Provide interface and delegate methods

Hiding Information

Other examples:

```
if (![DolphinAppDelegate isHighEndDevice]) {
    [self drainCellCache];
}

[self playTapSound];  // Form of delegation
```

- We don’t care what is classified as a high end or low end device
- We don’t need to know who handles sound or how it’s implemented
- Higher level logic doesn’t need to know the implementation details

End

Default Implementations

```
@implementation Base

- (void)doSomething {
     // Can init things here

     [self doSomethingImpl]; // This method can be overridden
}

- (void)doSomethingImpl {
     // Default implementation
}

@end
```

Classes and methods should do one thing well

Small classes, short methods, and few parameters

Good method and variable naming goes a long way

Hide and push logic to controllers

Use private properties

Even if you don't unit test, writing in a style that supports it is good architecture

Summary methods and variables. A few singletons are ok. Settable singletons?

Code Smell	Alternative Pattern
● Feature Envy	Move Method, Extract Method
● Inapp. Intimacy	Move Method, Hide Delegate
● Message Chains	Hide Delegate, Extract Method
● Singleton	Service Locator, Dep Injection