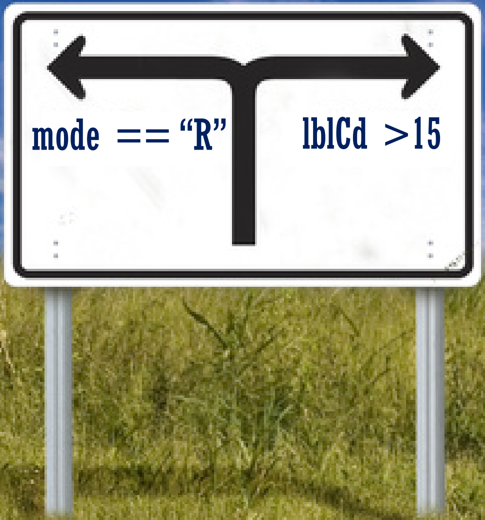
# Clean Code: Writing Code for Humans

Conditionals

Cory House
BitNative.com
Twitter: @housecor

pluralsight
hardcore developer training

## A fork in the road



**Understanding the original programmer's intent is the most difficult problem.**
**Fjelstad & Hamlen 1979**

1. Clear intent
2. Use the right tool
3. Bite-size logic
4. Sometimes code isn't the answer

## Compare Booleans Implicitly

**Dirty**

```
if (loggedIn == true)
{
    //do something nice.
}
```

**Clean**

```
if (loggedIn)
{
    //do something nice.
}
```

## Assign Booleans Implicitly

**Dirty**

```
bool goingToChipotleForLunch;

if (cashInWallet > 6.00)
{
    goingToChipotleForLunch = true;
} else {
    goingToChipotleForLunch = false;
}
```

**Clean**

```
bool goingToChipotleForLunch = cashInWallet > 6.00;
```

1. Fewer lines
2. No separate initialization
3. No repetition
4. Reads like speech

# Don't Be Anti-negative

In other words, use positive conditionals!

Dirty

```
if (!isNotLoggedIn)
```

Clean

```
if (loggedIn)
```

## Ternary is elegant

### Dirty

```
int registrationFee;

if (isSpeaker)
{
    registrationFee = 0;
}
else
{
    registrationFee = 50;
}
```

### Clean

```
int registrationFee = isSpeaker ? 0 : 50;
```

**Don't
Repeat
Yourself**

**You
Ain't
Gonna
Need
It.**

# Avoid being "Stringly" Typed

Dirty

```
if (employeeType == "manager")
```

Clean

```
if (employee.Type == EmployeeType.Manager)
```

1. Strongly typed =>No typos
2. Intellisense support
3. Documents states
4. Searchable

# Magic Numbers

**Which would you rather read?**
**Sally went to the #12 dealer to buy a #19 #515.**
**Sally went to the Ferrari dealer to buy a red Enzo.**

**Like magic, few can explain.**

**Dirty**

```
if (age > 21)
{
    //body here
}
```

**Clean**

```
const int legalDrinkingAge = 21;
if (age > legalDrinkingAge)
{
    //body here
}
```

**Dirty**

```
if (status == 2)
{
    //body here
}
```

**Clean**

```
if (status == Status.Active)
{
    //body here
}
```

# Complex Conditionals

```
if (car.Year > 1980
    && (car.Make == "Ford" || car.Make == "Chevrolet")
    && car.Odometer < 100000
    && car.Vin.StartsWith("V2") || car.Vin.StartsWith("IA3"))
{
    //do lots of things here.
}
```

1. **Intermediate variables**
2. **Encapsulate via function**

# Intermediate Variables

Dirty

```
if (employee.Age > 55
    && employee.YearsEmployed > 10
    && employee.IsRetired == true)
{
    //logic here
}
```

What question is this trying to answer?

Clean

```
bool eligibleForPension = employee.Age > MinRetirementAge
    && employee.YearsEmployed > MinPensionEmploymentYears
    && employee.IsRetired;
```

# Encapsulate Complex Conditionals

## Dirty

```
//Check for valid file extensions. Confirm admin or active
if (fileExtension == "mp4" ||
    fileExtension == "mpg" ||
    fileExtension == "avi")
    && (isAdmin || isActiveFile);
```

**Principle: Favor expressive code over comments**

## Clean

```
if (ValidFileRequest(fileExtension, isActiveFile, isAdmin))


private bool ValidFileRequest(string fileExtension, bool isActiveFile, bool isAdmin)
{
    return (fileExtension == "mp4" ||
        fileExtension == "mpg" ||
        fileExtension == "avi")
        && (isAdmin || isActiveFile);
}
    return validFileType && userIsAllowedToViewFile;
}
```

# Favor Polymorphism over Enums for Behavior

**Dirty**

```
public void LoginUser(User user)
{
    switch (user.Status)
    {
        case Status.Active:
            //logic for active users
            break;
        case Status.Inactive:
            //logic for inactive users
            break;
        case Status.Locked:
            //logic for locked users
            break;
    }
}
```

**Clean**

```
public void LoginUser(User user)
{
    user.Login();
}
```

```
public abstract class User
{
    public string FirstName;
    public string LastName;
    public Status Status;
    public int AccountBalance;

    public abstract void Login();
}
```

# Favor Polymorphism over Enums for Behavior

```
public class ActiveUser : User
{
    public override void Login()
    {
        //Active user logic here
    }
}

public class InactiveUser : User
{
    public override void Login()
    {
        //Inactive user logic here
    }
}

public class LockedUser : User
{
    public override void Login()
    {
        //Locked user logic here
    }
}
```

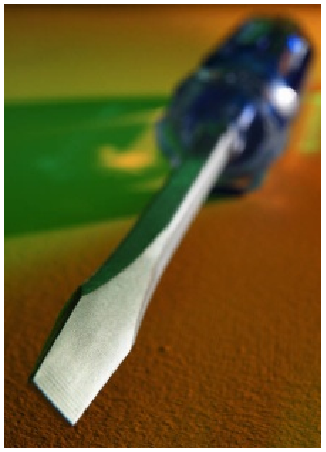
## Be declarative if possible

Dirty

```
List<User> matchingUsers = new List<User>();

foreach (var user in users)
{
    if (user.AccountBalance < minimumAccountBalance
        && user.Status == Status.Active)
    {
        matchingUsers.Add(user);
    }
}

return matchingUsers;
```



Clean

```
return users
    .Where(u => u.AccountBalance < minimumAccountBalance)
    .Where(u => u.Status == Status.Active);
```

C#: LINQ to objects
Java: Lambdaj
JavaScript: jLinq
Python: Pynq

Sometimes code isn’t the answer.

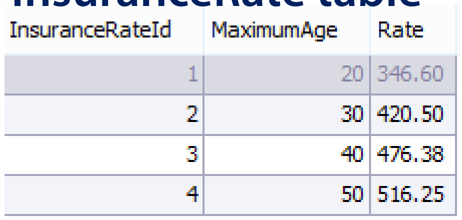
# Table Driven Methods

## Dirty

```
if (age < 20)
{
    return 345.60m;
}
else if (age < 30)
{
    return 419.50m;
}
else if (age < 40)
{
    return 476.38m;
}
else if (age < 50)
{
    return 516.25m;
}
```

## Clean

### InsuranceRate table



| InsuranceRateId | MaximumAge | Rate |
|---|---|---|
| 1 | 20 | 346.60 |
| 2 | 30 | 420.50 |
| 3 | 40 | 476.38 |
| 4 | 50 | 516.25 |

```
return Repository.GetInsuranceRate(age);
```

**Examples**
- Insurance rates
- Pricing structures
- Complex and dynamic business

## Table-driven methods

- Great for dynamic logic
- Avoids hard coding
- Write less code - Avoids complex data structures
- Easily changeable without a code change/app deployment

## Summary

- Strive for clear intent without leaning on comments
- Be strongly typed via constants and enums
- Be declarative rather than iterative when possible
- Consider leveraging the DB