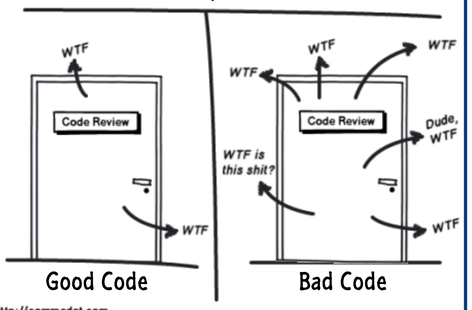


Clean Code

Boy Scout Rule

Always leave the code cleaner than you found it.

Use Descriptive and Intention-Revealing Names

- A name should tell you why it exists, what it does, and how it is used
- Don't be afraid to make a name long

Think Before You Write a Comment

- Can you use a descriptive method name or variable instead?
- Explain yourself in code
- Comments do not make up for bad code

Writing Clean Code Is an Iterative Process

Writing software is like any other kind of writing: the first draft might be clumsy and disorganized, so you wordsmith it and restructure it and refine it until it reads the way you want it to read.

So, **refactor often**!

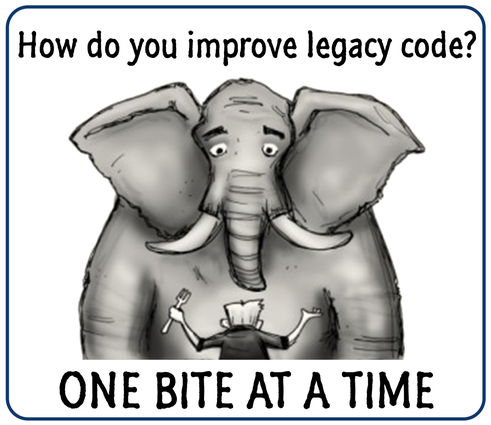
NNIT
Solutions Developer Academy
2013-2014

Methods

- Should do one thing and do it well
- Should only be at one level of abstraction
- Should have at most one level of indentation (no nested if's!)
- Should be small. No, even smaller! ≤ 5 lines of code
- Should ideally have no arguments, to help consumers of the method

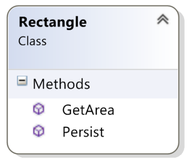
DRY

Don't **R**epeat **Y**ourself - duplication may be the root of all evil in software.
~~Don't Repeat Yourself~~
~~Don't Repeat Yourself~~ - no, seriously...

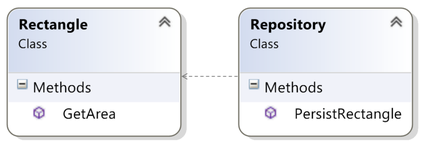


Single Responsibility Principle

- A class should have only one reason to change
- Separation of concerns - do one thing, and do it well



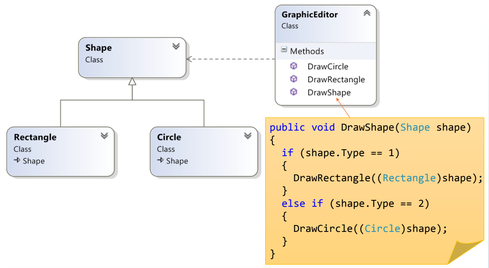
Rectangle contains two methods: `GetArea`, which returns the area of the rectangle, and `Persist`, which persists the instance to disk.
The Rectangle class does more than one thing.



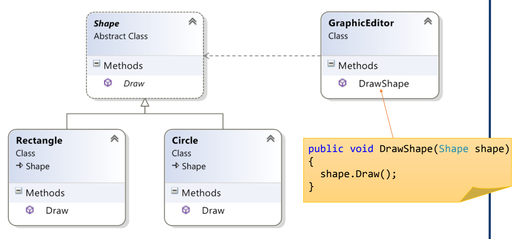
The Rectangle class only calculates the rectangle's area.
The Repository class only persists a rectangle to disk.
Each class has one single responsibility.

Open/Closed Principle

- A class should be open for extension, but closed for modification
- Favour composition over inheritance



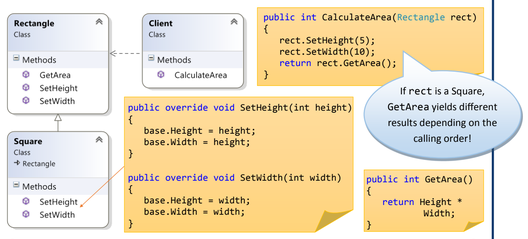
To add a new shape, the `DrawShape` method must be changed.
GraphicEditor is not closed for modification.



To add a new shape, the `DrawShape` method does not need to be changed.
GraphicEditor is open for extension, but closed for modification.

Liskov Substitution Principle

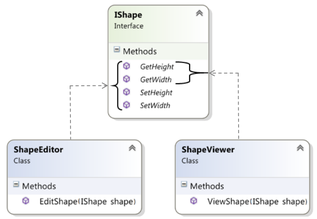
- Subtypes must be substitutable for their base types
- Subtypes should extend base types without replacing their behavior



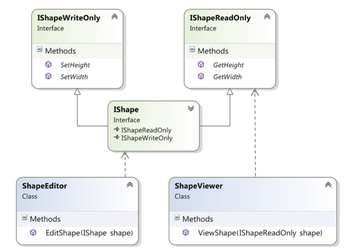
By inheriting from Rectangle, Square needs to implement both `SetHeight` and `SetWidth`, even though height = width for a square.
This changes the **behavior** of Rectangle's API for Square instances.
Square is not substitutable for Rectangle.

Interface Segregation Principle

- Clients should not be forced to depend on methods they do not use



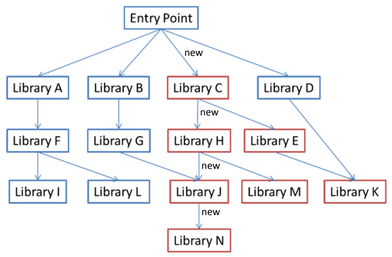
ShapeEditor and ShapeViewer use different parts of the same interface.
If the `SetWidth` method is changed, ShapeViewer is affected even though it does not use that method.



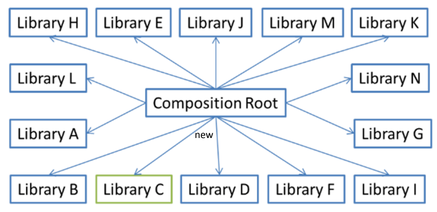
ShapeEditor and ShapeViewer use different interfaces.
If the `SetWidth` method is changed, ShapeViewer is not affected.

Dependency Inversion Principle

- High-level modules should not depend on low-level modules
- Both should depend on abstractions such as interfaces



Each library instantiates its own libraries.
High-level libraries thus depend on low-level libraries.



The composition root instantiates all libraries.
Libraries are not depending on each other at the instance level.

SQL Injection

```
var cmd = new SqlCommand(
"SELECT Id FROM Users WHERE Name = " + name, connection);
```

A hacker will try to inject "**'';DROP TABLE Users**" into the variable **name** and thereby delete the Users table.

```
var cmd = new SqlCommand(
"SELECT Id FROM Users WHERE Name = @Name", connection);
cmd.Parameters.Add(new SqlParameter("Name", name));
```

Data and syntax is separated by using parameters. The contents of the variable **name** is treated as a literal value, not as executable code.

Broken Authentication

- Storage of clear text passwords/weak hashing in files, databases, etc.
- Weak password policy
- OAuth vulnerabilities

- Use a strong cryptographic hash algorithm (PBKDF-2, Scrypt, Bcrypt)
- Authentication should be simple, centralized and standardized (put all policy checking in one file)
- Be sure SSL protects both credentials and session ID
- Don't include passwords in the URL/cookies
- Provide a password strength meter on account creation
- Never enable password recovery, always use password resets

Broken Session Management

- Session ID in URL
  http://www.mysite.com/Admin.aspx?SESSID=6aed50f09ab8c5e1b0341cba435047fa
- Credentials in URL
  http://www.mysite.com/Admin.aspx?user=admin&pass=123456
- Credentials in cookie
  SESSID=6aed50f09ab8c5e1b0341cba435047fa&username=admin&password=123

- Never use cookieless sessions
- In ASP.NET we are always vulnerable to replay attacks
  - Set SlidingExpiration="false" (it then uses absolute expiration)
  - Use TLS for authentication cookies (set FormsAuthentication.RequireSSL to true)
  - Create your own secure session handler
- Don't include the session ID anywhere else but in cookies

Password Hashing Done Right

```
// Get salt from application configuration
string appSalt = ConfigurationManager.AppSettings["AppSalt"];

// Generate a random salt
byte[] randomSalt = new byte[64];
RNGCryptoServiceProvider rng = new RNGCryptoServiceProvider();
rng.GetBytes(randomSalt);

// Combine the salts in a byte array
byte[] saltB = Encoding.UTF8.GetBytes(appSalt).Concat(randomSalt).ToArray();

// Convert the password to a byte array
byte[] passB = Encoding.UTF8.GetBytes(password);

// Hash the password using PBKDF2
byte[] hashedPassword;
using (Rfc2898DeriveBytes rfc2898 = new Rfc2898DeriveBytes(passB,
saltB, 4096))
{ hashedPassword = rfc2898.GetBytes(32); }
```

Command-Query Separation

Every method should either be a **command** that performs an action, or a **query** that returns data to the caller, but not both.

Commands have side effects, queries do not.

Commands:
```
void Save(Order order)
void Send(string message)
void Associate(IFoo foo, Bar bar)
```

Queries:
```
Order[] GetOrders(int userId)
IFoo Map(Bar bar)
T Create()
```