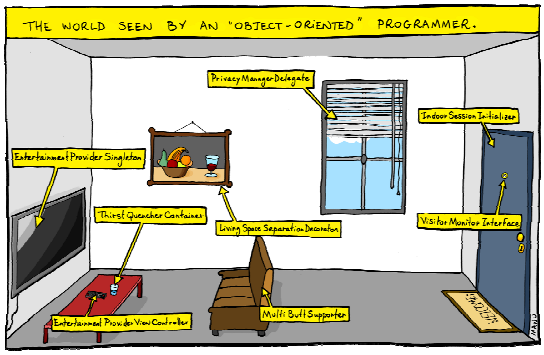


Clean Classes

www.cs.uoi.gr/~zarras/soft-devII.htm

from Clean Code by R. C. Martin, a.k.a "Uncle Bob"

Class Organization

Following the standard Java convention, a class should begin with a list of attributes.

Public static constants, if any, should come first. Then private static attributes, followed by private instance attributes.

Public functions/methods should follow the list of attributes. We like to put the private utilities called by a public function right after the public function itself. This follows the stepdown rule and helps the program read like a newspaper article.

Encapsulation

We like to keep our attributes and utility functions/methods private, but we're not fanatic about it.

Sometimes we need to make an attribute or utility function/methods protected.

There is seldom a good reason to have a public attribute !!!

Classes should be small!!

Listing 10-1
Too Many Responsibilities

```
public class SuperDashboard extends JFrame implements MetaDataUser
  public String getCustomizerLanguagePath()
  public void setSystemConfigPath(String systemConfigPath)
  public String getSystemConfigDocument()
  public void setSystemConfigDocument(String systemConfigDocument)
  public boolean getGuruState()
  public boolean getNoviceState()
  public boolean getOpenSourceState()
  public void showObject(MetaObject object)
  public void showProgress(String s)
  public boolean isMetadataDirty()
  public void setIsMetadataDirty(boolean isMetadataDirty)
  public Component getLastFocusedComponent()
  public void setLastFocused(Component lastFocused)
  public void setMouseSelectState(boolean isMouseSelected)
  public boolean isMouseSelected()
  public LanguageManager getLanguageManager()
  public Project getProject()
  public Project getFirstProject()
  public Project getLastProject()
  public String getNewProjectName()
  public void setComponentSize(Dimension dim)
  public String getCurrentDir()
  public void setCurrentDir(String newDir)
  public void updateStatus(int dotPos, int markPos)
  public Class[] getDataBaseClasses()
  public MetaDataFeeder getMetaDataFeeder()
  public void addProject(Project project)
  public boolean setCurrentProject(Project project)
  public boolean removeProject(Project project)
  public MetaProjectHeader getProgramMetaData()
  public void resetDashboard()
  public Project loadProject(String fileName, String projectName)
  public void setCanSaveMetaData(boolean canSave)
  public MetaObject getSelectedObject()
  public void deselectObjects()
  public void setProject(Project project)
  public void editorAction(String actionName, ActionEvent event)
  public void setMode(int mode)
  public FileManager getFileManager()
  public void setFileManager(FileManager fileManager)
  public ConfigManager getConfigManager()
  public void setConfigManager(ConfigManager configManager)
  public ClassLoader getClassLoader()
  public void setClassLoader(ClassLoader classLoader)
  public Properties getProps()
  public String getUserName()
  public String getBaseDir()
  public int getMajorVersionNumber()
  public int getMinorVersionNumber()
  public int getBuildNumber()
  public MetaObject pasting(
      MetaObject target, MetaObject pasted, MetaProject project)
  public void processMenuItems(MetaObject metaObject)
  public void processMenuItems(MetaObject metaObject)
  public void setAllowDragging(boolean allowDragging)
  public boolean allowDragging()
  public boolean isCustomizing()
  public void setTitle(String title)
  public IdeMenuBar getIdeMenuBar()
  public void showHelper(MetaObject metaObject, String propertyName)
  // ... many non-public methods follow ...
}
```

With functions we measured size by counting physical lines.

Listing 10-1 outlines a class, SuperDashboard, that exposes about 70 public methods.

Most developers would agree that it's a bit too super in size.

Some developers might refer to SuperDashboard as a "God class"

Classes should be small!!

Listing 10-2
Small Enough?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

But what if SuperDashboard contained only 5 methods ?

Five methods isn't too much, is it?

Classes should be small!!

Listing 10-2
Small Enough?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

With **classes** we use a different measure. We count ***responsibilities***.

In this case the class is NOT small, despite its small number of methods, SuperDashboard has too many *responsibilities*.

The name of a class should describe what responsibilities it fulfills.

In fact, naming is probably the first way of helping determine class size. If we cannot derive a concise name for a class, then it's likely too large. The more ambiguous the class name, the more likely it has too many responsibilities.

For example, class names including weasel words like Processor or Manager or Super often hint at unfortunate aggregation of responsibilities.

Classes should be small!!

Listing 10-2
Small Enough?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

We should also be able to write a brief description of the class in about 25 words, without using the words "if," "and," "or," or "but."

How would we describe the SuperDashboard?

"The SuperDashboard provides access to the GUI component that last held the focus, and it also allows us to track the version and build numbers."

The first "and" is a hint that SuperDashboard has too many responsibilities.

Classes should be small!!

Listing 10-2
Small Enough?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

Is there any well known principle that promotes small classes ???

The Single Responsibility Principle

Listing 10-2
Small Enough?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

This principle gives us both a definition of responsibility, and a guideline for class size.

Classes should have one responsibility—one reason to change !!

The Single Responsibility Principle

Listing 10-2
Small Enough?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

The seemingly small SuperDashboard class has two reasons to change.

First, it tracks version information that would seemingly need to be updated every time the software gets shipped.

Second, it manages Java Swing components (it is a derivative of JFrame, the Swing representation of a top-level GUI window).

Organizing for change

For most systems, change is continual.

Every change subjects us to the risk that the remainder of the system no longer works as intended. Effort is needed to analyze the impact of changes, compile, test, etc.

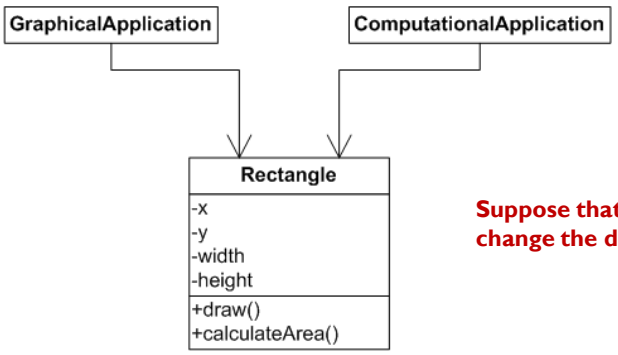
In a clean system we organize our classes so as to reduce this effort.

Big classes that violate SRP are significant impediments towards reducing this effort

➔ A change to a big class can affect many things inside the class

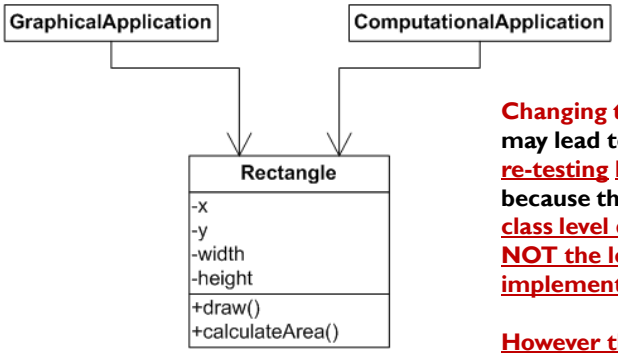
➔ A change to a big class may affect the clients and big classes have many clients

Organizing for change



Suppose that we want to change the drawing method

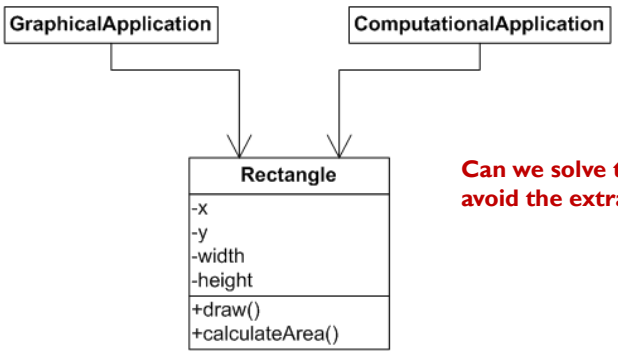
Organizing for change



Changing the drawing method may lead to re-compilation and re-testing both applications because the tools "see" the class level dependencies and NOT the low level implementation

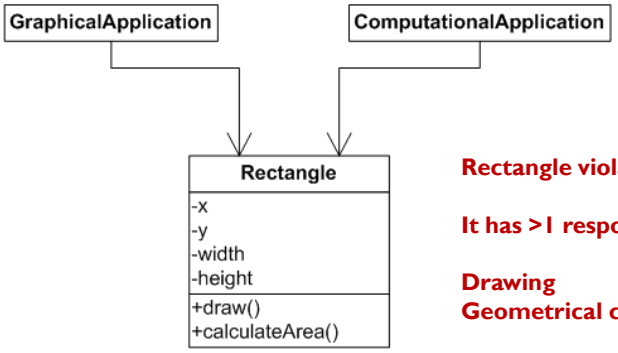
However this SHOULDN'T be the case because only the GraphicalApplication uses the drawing method !!!

Organizing for change



Can we solve the problem and avoid the extra effort??

Organizing for change

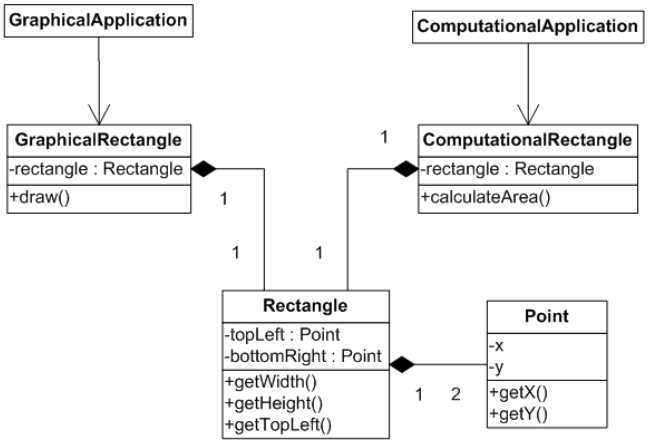


Rectangle violates SRP

It has >1 responsibities

Drawing
Geometrical calculations

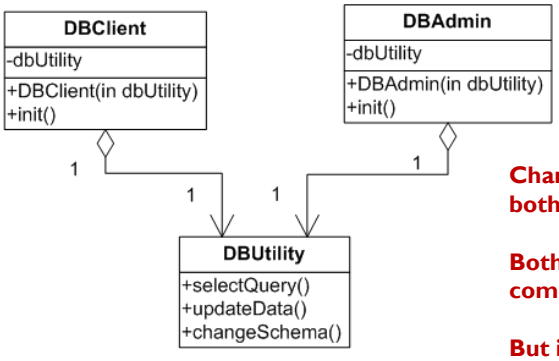
Organizing for change



Solution:
we break the responsibilities to different "smaller" classes…..

NOW changing the drawing method affects only the GraphicalApplication

Isolating from change

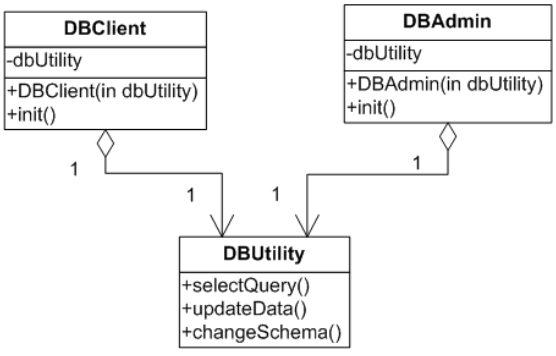


Changes to any method affect both DBAdmin and DBUser

Both should be re-tested, re-compiled, etc…

But it shouldn't because DBUser only uses selectData() while DBAdmin only uses updateData() and changeSchema()

Isolating from change



Maybe we should separate selectQuery and updateData, changeSchema to different classes ????

What if we DON'T have time to split DBUtility ?

Or what if it DOES NOT MAKE SENSE to split it because it is generally cohesive, providing a single responsibility (ACCESS TO Database) ?

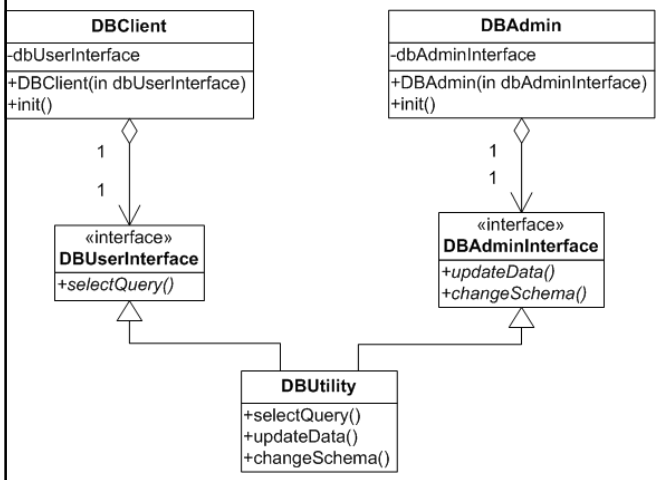
How can we organize better for change?

Isolating from change

The Interface Segregation Principle (ISP) says that our classes should not depend upon methods that they don't use !!

The idea then is to extract different interfaces that match the needs of the client classes…

Isolating from change



DBClient and DBAdmin are now isolated from changes to methods they do not use as they use different interfaces implemented by the same class

Isolating from change

Needs will change, therefore code will change.

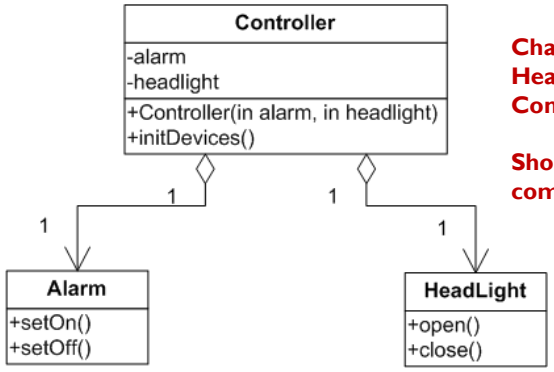
A client class depending upon concrete classes is at risk when those classes change.

We can introduce interfaces and abstract classes to isolate the impact of those changes.

The Dependency Inversion Principle (DIP) says that our classes should depend upon abstractions, not on concrete details !!

Also known as Abstract Coupling

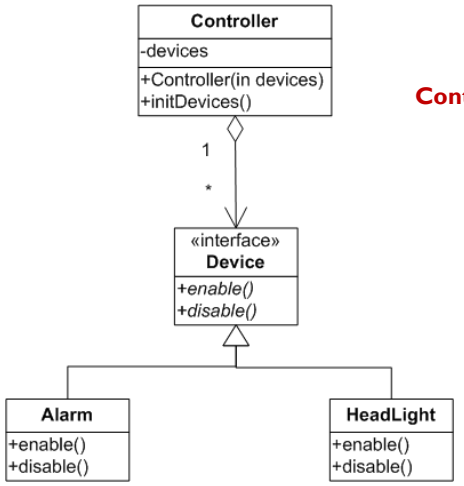
Isolating from change



Changes to Alarm or HeadLight affect Controller…..

Should be re-tested, re-compiled, etc

Isolating from change



Controller is isolated from change

Organize for extension

Needs change (DIP, ISP) but also **new needs may be added**, therefore code should be extended.

Extensions should be done without much effort.

Preferably, only by adding new classes and without having to understand and modify the internals of the software

Is there any well known principle that promotes this goal ??

Organize for extension

Needs change (DIP, ISP) but also **new needs may be added**, therefore code should be extended.

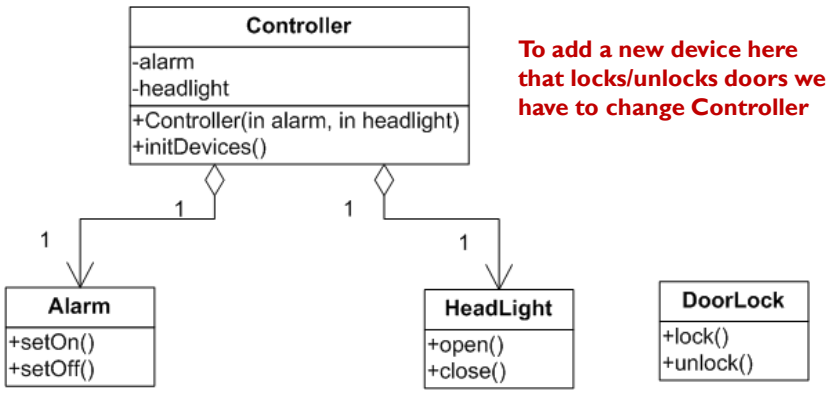
Extensions should be done without much effort.

Preferably, only by adding new classes and without having to understand and modify the internals of the software

Is there any well known principle that promotes this goal ??

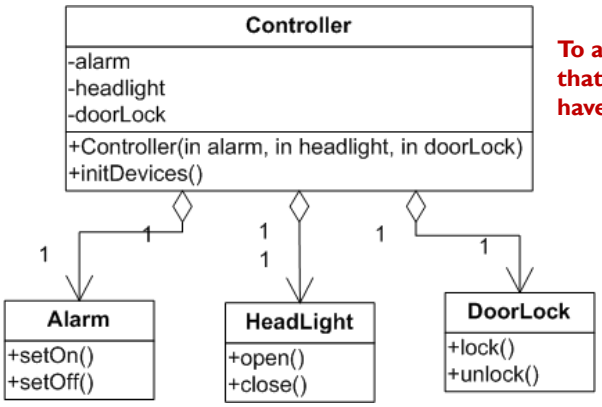
The Open Closed Principle (OCP) says that our software should be open for extensions and closed to modifications !!

Organize for extension



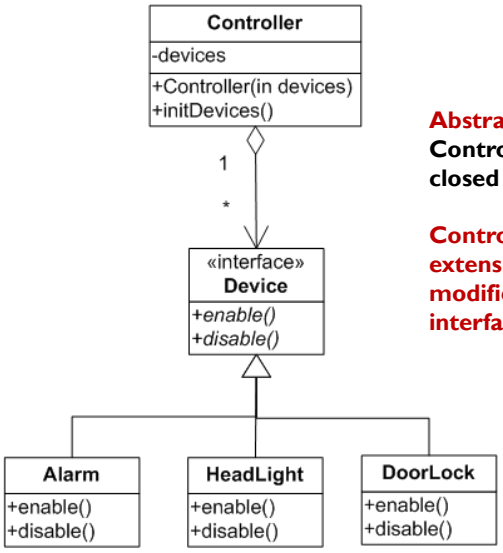
To add a new device here that locks/unlocks doors we have to change Controller

Organize for extension



To add a new device here that locks/unlocks doors we have to change Controller

Organize for extension



Abstract Coupling makes the Controller open for extension and closed to modification

Controller is now open for extension and closed to modification because it uses an interface instead of concrete classes