Design Principle 1: Divide and conquer

Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things

- Separate people can work on each part.
- An individual software engineer can specialize.
- Each individual component is smaller, and therefore easier to understand.
- Parts can be replaced or changed without having to replace or extensively change other parts.

Design Principle 2: Increase cohesion where possible

A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things

- This makes the system as a whole easier to understand and change
- Type of cohesion:
  - —Functional, Layer, Communicational, Sequential, Procedural, Temporal, Utility

Communicational cohesion

All the *modules that access or manipulate certain data* are kept together (e.g. in the same class) - and everything else is kept out

- A class would have good communicational cohesion
  - —if all the system's facilities for storing and manipulating its data are contained in this class.
  - —if the class does not do anything other than manage its data.
- Main advantage: When you need to make changes to the data, you find all the code in one place

Utility cohesion

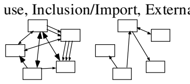
When *related utilities which cannot be logically placed in other cohesive units* are kept together

- A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable.
- For example, the `java.lang.Math` class.

Design Principle 3: Reduce coupling where possible

***Coupling* occurs when there are *interdependencies* between one module and another**

- When interdependencies exist, changes in one place will require changes somewhere else.
- A network of interdependencies makes it hard to see at a glance how some component works.
- Type of coupling:
  - —Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, External



Content coupling:

Occurs when one component *surreptitiously* modifies data that is *internal* to another component

- To reduce content coupling you should therefore *encapsulate* all instance variables
  - —declare them `private`
  - —and provide get and set methods
- A worse form of content coupling occurs when you directly modify an instance variable *of* an instance variable

Example of content coupling

```
public class Line
{
  private Point start, end;
  ...
  public Point getStart() { return start; }
  public Point getEnd()  { return end; }
}

public class Arch
{
  private Line baseline;
  ...
  void slant(int newY)
  {
    Point theEnd = baseline.getEnd();
    theEnd.setLocation(theEnd.getX(),newY);
  }
}
```


Common coupling

Occurs whenever you use a *global variable*

- All the components using the global variable become coupled to each other
- A weaker form of common coupling is when a variable can be accessed by a *subset* of the system's classes
  - —e.g. a Java package
- Can be acceptable for creating global variables that represent system-wide default values
- The Singleton design pattern provides encapsulated global access to an object

Intermezzo: introduction to Design Patterns

The recurring aspects of designs are called *design patterns*.

- A *pattern* is the outline of a reusable solution to a general problem encountered in a particular context
- Many of them have been systematically documented for all software developers to use
- A good pattern should
  - —Be as general as possible
  - —Contain a solution that has been proven to effectively solve the problem in the indicated context.

Studying patterns is an effective way to learn from the experience of others

Intermezzo: Design Pattern description

Context:
- The general situation in which the pattern applies

Problem:
- —A short sentence or two raising the main difficulty.

Forces:
- The issues or concerns to consider when solving the problem

Solution:
- The recommended way to solve the problem in the given context.
  - —'to balance the forces'

Antipatterns: (Optional)
- Solutions that are inferior or do not work in this context.

Related patterns: (Optional)
- Patterns that are similar to this pattern.

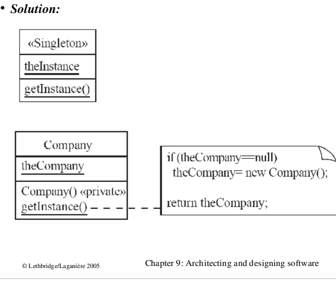
References:
- Who developed or inspired the pattern.

Intermezzo: the Singleton Design Pattern

- ***Context***:
  - —It is very common to find classes for which only one instance should exist (*singleton)*
- ***Problem***:
  - —How do you ensure that it is never possible to create more than one instance of a singleton class?
- ***Forces***:
  - —The use of a public constructor cannot guarantee that no more than one instance will be created.
  - —The singleton instance must also be accessible to all classes that require it

Intermezzo: Singleton Pattern (example)

- ***Solution:***



Control coupling

Occurs when one procedure calls another using *a 'flag' or 'command'* that explicitly controls what the second procedure does

- To make a change you have to change both the calling and called method
- The use of polymorphic operations is normally the best way to avoid control coupling
- One way to reduce the control coupling could be to have a *look-up table*
  - —commands are then mapped to a method that should be called when that command is issued

Example of control coupling

```
public routineX(String command)
{
   if (command.equals("drawCircle")
   {
      drawCircle();
   }
   else
   {
      drawRectangle();
   }
}
```


Stamp coupling:

Occurs whenever one of your application classes is declared as the *type* of a method argument

- Since one class now uses the other, changing the system becomes harder
  - —Reusing one class requires reusing the other
- Two ways to reduce stamp coupling,
  - —using an interface as the argument type
  - —passing simple variables

Example of stamp coupling

```
public class Emailer
{
  public void sendEmail(Employee e, String text)
  {...}
  ...
}
```

Using simple data types to avoid it:

```
public class Emailer
{
  public void sendEmail(String name, String email, String text)
  {...}
  ...
}
```


Example of stamp coupling

Using an interface to avoid it:

```
public interface Addressee
{
  public abstract String getName();
  public abstract String getEmail();
}

public class Employee implements Addressee {...}

public class Emailer
{
  public void sendEmail(Addressee e, String text)
  {...}
  ...
}
```


Data coupling

Occurs whenever the types of method arguments are either primitive or else simple library classes

- The more arguments a method has, the higher the coupling
  - —All methods that use the method must pass all the arguments
- You should reduce coupling by not giving methods unnecessary arguments
- There is a trade-off between data coupling and stamp coupling
  - —Increasing one often decreases the other

Routine call coupling

Occurs when one routine (or method in an object oriented system) calls another

- The routines are coupled because they depend on each other's behaviour
- Routine call coupling is always present in any system.

- If you repetitively use a sequence of two or more methods to compute something
  - —then you can reduce routine call coupling by writing a single routine that encapsulates the sequence.

Type use coupling

Occurs when a module uses a data type defined in another module

- It occurs any time a class declares an instance variable or a local variable as having another class for its type.
- The consequence of type use coupling is that if the type definition changes, then the users of the type may have to change
- Always declare the type of a variable to be the most general possible class or interface that contains the required operations

Inclusion or import coupling

Occurs when one component imports a package

- (as in Java)

or when one component includes another

- (as in C++).
- The including or importing component is now exposed to everything in the included or imported component.
- If the included/imported component changes something or adds something.
  - —This may raise a conflict with something in the includer, forcing the includer to change.
- An item in an imported component might have the same name as something you have already defined.

External coupling

When a module has a dependency on such things as the operating system, shared libraries or the hardware

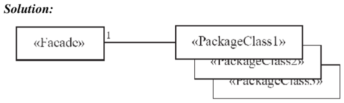
- It is best to reduce the number of places in the code where such dependencies exist.
- The Façade design pattern can reduce external coupling

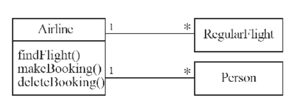
Intermezzo: The Façade Design Pattern

- ***Context***:
  - —Often, an application contains several complex packages.
  - —A programmer working with such packages has to manipulate many different classes
- ***Problem***:
  - —How do you simplify the view that programmers have of a complex package?
- ***Forces***:
  - —It is hard for a programmer to understand and use an entire subsystem
  - —If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.

Intermezzo: Façade DP (example)

- ***Solution:***





Design Principle 4: Keep the level of abstraction as high as possible

Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity

- A good abstraction is said to provide *information hiding*
- Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details

Abstraction and classes

Classes are data abstractions that contain procedural abstractions

- Abstraction is increased by defining all variables as private.
- The fewer public methods in a class, the better the abstraction
- Superclasses and interfaces increase the level of abstraction
- Attributes and associations are also data abstractions.
- Methods are procedural abstractions
  —Better abstractions are achieved by giving methods fewer parameters

Design Principle 5: Increase reusability where possible

Design the various aspects of your system so that they can be used again in other contexts

- Generalize your design as much as possible
- Follow the preceding three design principles
- Design your system to contain hooks
- Simplify your design as much as possible

Design Principle 6: Reuse existing designs and code where possible

Design with reuse is complementary to design for reusability

- Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
  —*Cloning* should not be seen as a form of reuse

Design Principle 7: Design for flexibility

Actively anticipate changes that a design may have to undergo in the future, and prepare for them

- Reduce coupling and increase cohesion
- Create abstractions
- Do not hard-code anything
- Leave all options open
  —Do not restrict the options of people who have to modify the system later
- Use reusable code and make code reusable

Design Principle 8: Anticipate obsolescence

Plan for changes in the technology or environment so the software will continue to run or can be easily changed

- Avoid using early releases of technology
- Avoid using software libraries that are specific to particular environments
- Avoid using undocumented features or little-used features of software libraries
- Avoid using software or special hardware from companies that are less likely to provide long-term support
- Use standard languages and technologies that are supported by multiple vendors

Design Principle 9: Design for Portability

Have the software run on as many platforms as possible

- Avoid the use of facilities that are specific to one particular environment
- E.g. a library only available in Microsoft Windows

Design Principle 10: Design for Testability

Take steps to make testing easier

- Design a program to automatically test the software
  - —Discussed more in Chapter 10
  - —Ensure that all the functionality of the code can by driven by an external program, bypassing a graphical user interface
- In Java, you can create a main() method in each class in order to exercise the other methods

Design Principle 11: Design defensively

Never trust how others will try to use a component you are designing

- Handle all cases where other code might attempt to use your component inappropriately
- Check that all of the inputs to your component are valid: the *preconditions*
  - —Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking

Design by contract

A technique that allows you to design defensively in an efficient and systematic way

- Key idea
  - —each method has an explicit *contract* with its callers
- The contract has a set of assertions that state:
  - —What *preconditions* the called method requires to be true when it starts executing
  - —What *postconditions* the called method agrees to ensure are true when it finishes executing
  - —What *invariants* the called method agrees will not change as it executes

9.3 Techniques for making good design decisions

Using priorities and objectives to decide among alternatives

- Step 1: List and describe the alternatives for the design decision.
- Step 2: List the advantages and disadvantages of each alternative with respect to your objectives and priorities.
- Step 3: Determine whether any of the alternatives prevents you from meeting one or more of the objectives.
- Step 4: Choose the alternative that helps you to best meet your objectives.
- Step 5: Adjust priorities for subsequent decision making.

Example priorities and objectives

Imagine a system has the following objectives, starting with top priority:

- **Security**: Encryption must not be breakable within 100 hours of computing time on a 400Mhz Intel processor, using known cryptanalysis techniques.
- **Maintainability**. No specific objective.
- **CPU efficiency**. Must respond to the user within one second when running on a 400MHz Intel processor.
- **Network bandwidth efficiency**: Must not require transmission of more than 8KB of data per transaction.
- **Memory efficiency**. Must not consume over 20MB of RAM.
- **Portability.** Must be able to run on Windows 98, NT 4 and ME as well as Linux

Example evaluation of alternatives

	Security	*Maintainability*	*Memory efficiency*	*CPU efficiency*	*Bandwidth efficiency*	*Portability*
Algorithm A	High	Medium	High	Medium; DNMO	Low	Low
Algorithm B	High	High	Low	Medium; DNMO	Medium	Low
Algorithm C	High	High	High	Low; DNMO	High	Low
Algorithm D	—	—	—	Medium; DNMO	DNMO	—
Algorithm E	DNMO	—	—	Low; DNMO	—	—

'DNMO' means *Does Not Meet the Objective*

Using cost-benefit analysis to choose among alternatives

- To estimate the *costs*, add up:
  - —The incremental cost of doing the *software engineering* work, including ongoing maintenance
  - —The incremental costs of any *development technology* required
  - —The incremental costs that *end-users and product support personnel* will experience
- To estimate the *benefits*, add up:
  - —The incremental software engineering time saved
  - —The incremental benefits measured in terms of either increased sales or else financial benefit to users