Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 9:

Architecting and Designing Software

(Topic 8)

Design Decision

Question: What are some of the things you think, as a programmer, are important in order to say a piece of software is *well-designed*?

Question: We all have to make design decisions when creating software. Let's say that we decide to **not use a database.**

- Why might we make that decision?
- What do we have to think about when making that decision?

9.1 The Process of Design

Definition:

- *Design* is a problem-solving process whose objective is to find and describe a way:
  - To implement the system's *functional requirements*...
  - While respecting the constraints imposed by the *non-functional requirements*...
    - including the budget
  - And while adhering to general principles of *good quality*

Design as a series of decisions

A designer is faced with a series of *design issues*

- These are sub-problems of the overall design problem.
- Each issue normally has several alternative solutions:
  - design *options*.
- The designer makes a *design decision* to resolve each issue.
  - This process involves choosing the best option from among the alternatives.

Making decisions

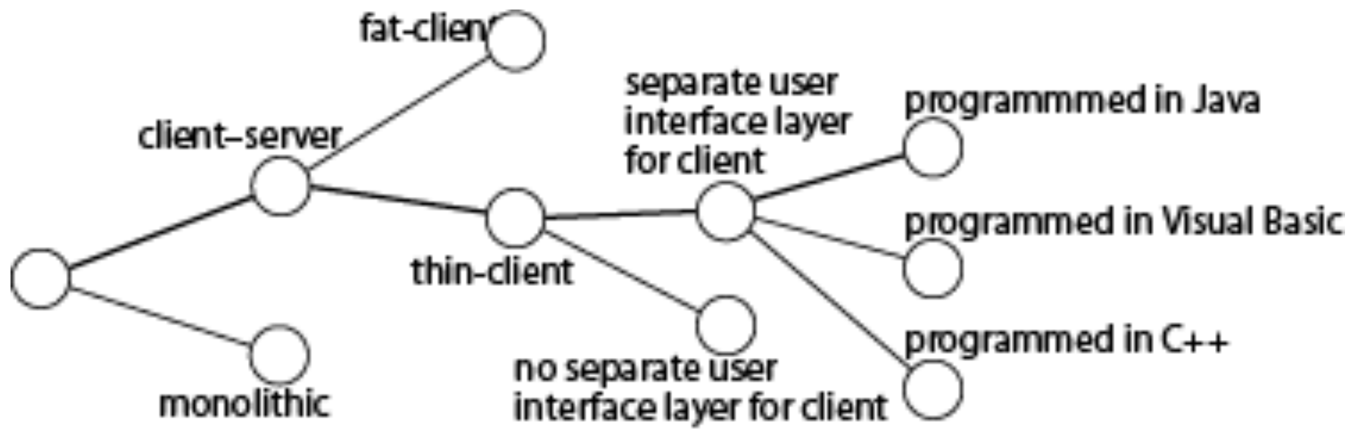
To make each design decision, the software engineer uses:

- Knowledge of
  - the requirements
  - the design as created so far
  - the technology available
  - software design principles and ‘best practices’
  - what has worked well in the past

Design space

The space of possible designs that could be achieved by choosing different sets of alternatives is often called the *design space*

- For example:


fat-client
client–server
separate user interface layer for client
programmed in Java
programmed in Visual Basic
thin-client
programmed in C++
no separate user interface layer for client
monolithic

Component

Any piece of software or hardware that has a clear role.

- A component can be isolated, allowing you to replace it with a different component that has equivalent functionality.
- Many components are designed to be reusable.
- Conversely, others perform special-purpose functions.

Module

A component that is defined at the programming language level

- For example, methods, classes and packages are modules in Java.

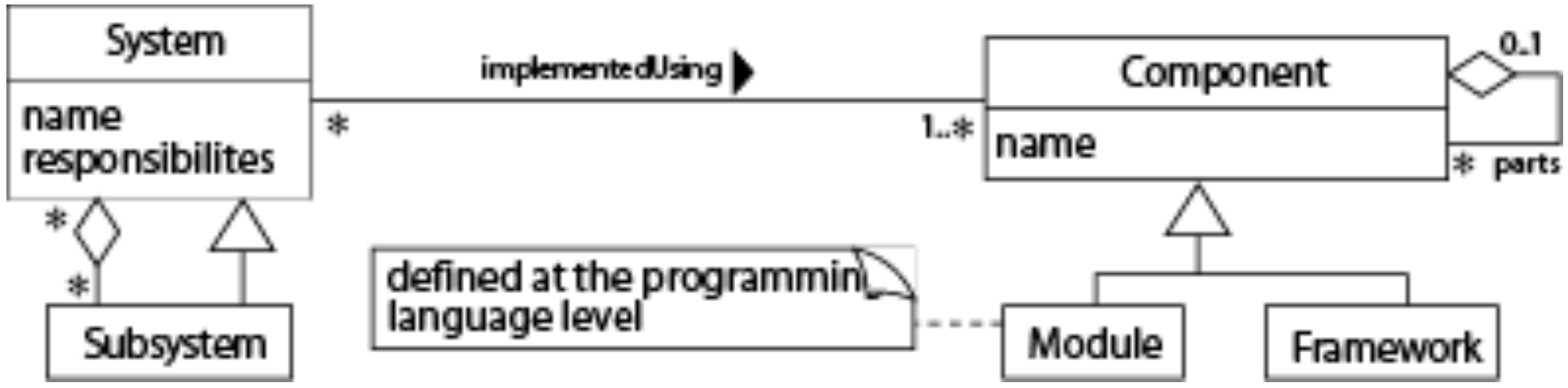
System

A logical entity, having a set of definable responsibilities or objectives, and consisting of hardware, software or both.

- A system can have a specification which is then implemented by a collection of components.
- A system continues to exist, even if its components are changed or replaced.
- The goal of requirements analysis is to determine the responsibilities of a system.

- **Subsystem**:
  - A system that is part of a larger system, and which has a definite interface

UML diagram of system parts



Top-down and bottom-up design

Top-down design

- First design the very high level structure of the system.
- Then gradually work down to detailed decisions about low-level constructs.
- Finally arrive at detailed decisions such as:
  - —the format of particular data items;
  - —the individual algorithms that will be used.

Top-down and bottom-up design

Bottom-up design

- Make decisions about reusable low-level utilities.
- Then decide how these will be put together to create high-level constructs.

A mix of top-down and bottom-up approaches are normally used:

- Top-down design is almost always needed to give the system a good structure.
- Bottom-up design is normally useful so that reusable components can be created.

Different aspects of design

- *Architecture design*:
  - The division into subsystems and components,
    - How these will be connected.
    - How they will interact.
    - Their interfaces.
- *Class design*:
  - The various features of classes.
- *User interface design*
- *Algorithm design*:
  - The design of computational mechanisms.
- *Protocol design*:
  - The design of communications protocol.

9.2 Principles Leading to Good Design

Overall *goals* of good design:

- Increasing profit by reducing cost and increasing revenue
- Ensuring that we actually conform with the requirements
- Accelerating development
- Increasing qualities such as
  - Usability
  - Efficiency
  - Reliability
  - Maintainability
  - Reusability

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.

Ways of dividing a software system

- A distributed system is divided up into clients and servers
- A system is divided up into subsystems
- A subsystem can be divided up into one or more packages
- A package is divided up into classes
- A class is divided up into methods

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

Functional cohesion

This is achieved when all the code that computes a particular result is kept together - and everything else is kept out

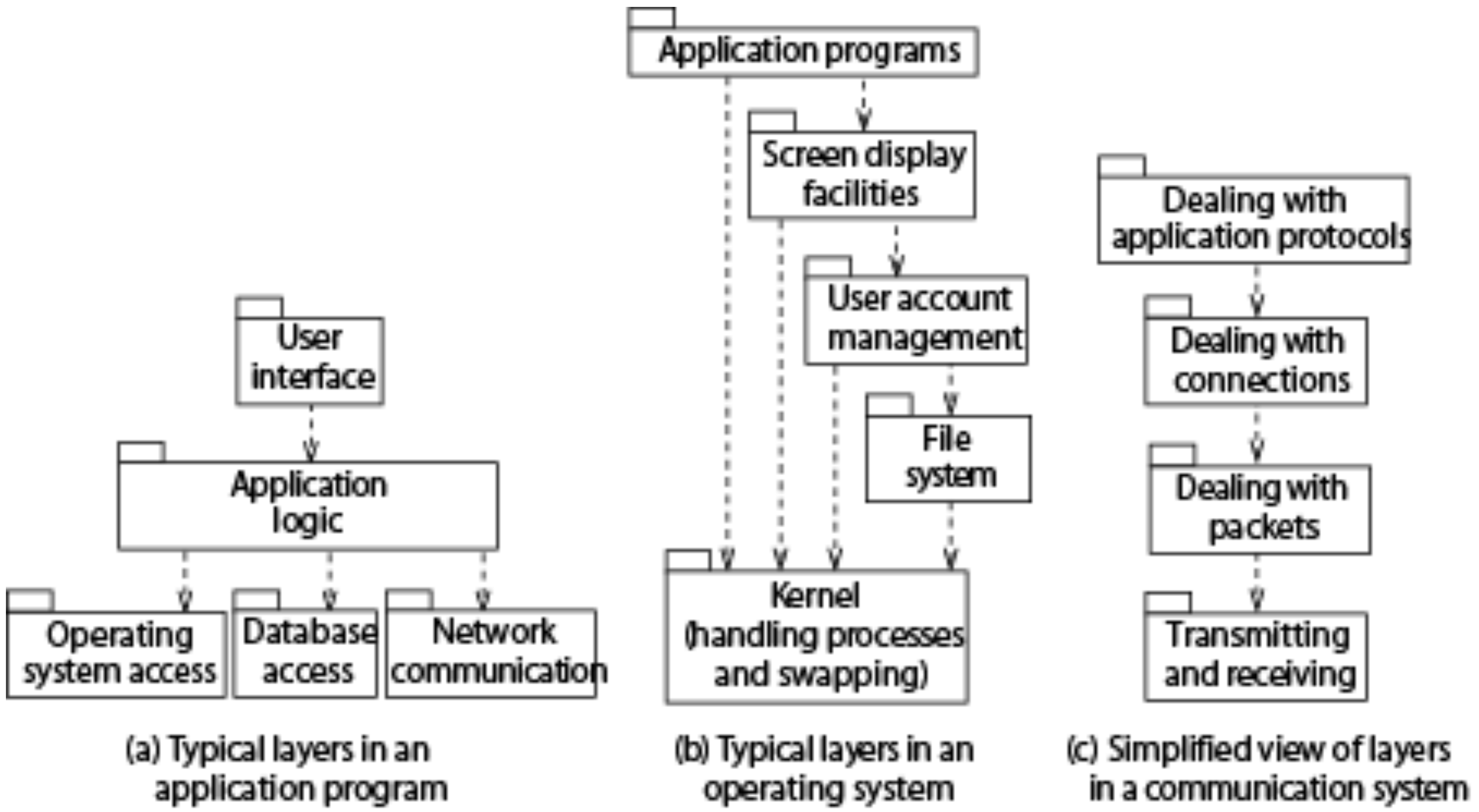
- i.e. when a module only performs a *single* computation, and returns a result, *without having side-effects*.
- Benefits to the system:
  - Easier to understand
  - More reusable
  - Easier to replace
- Modules that update a database, create a new file or interact with the user are not functionally cohesive

Layer cohesion

All the facilities for providing or accessing a set of related services are kept together, and everything else is kept out

- The layers should form a hierarchy
  - Higher layers can access services of lower layers,
  - Lower layers do not access higher layers
- The set of procedures through which a layer provides its services is the *application programming interface (API)*
- You can replace a layer without having any impact on the other layers
  - You just replicate the API

Example of the use of layers



(a) Typical layers in an application program

(b) Typical layers in an operating system

(c) Simplified view of layers in a communication system

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

Sequential cohesion

Procedures, in which one procedure provides input to the next, are kept together – and everything else is kept out

- You should achieve sequential cohesion, only once you have already achieved the preceding types of cohesion.

Procedural cohesion

Keep together several procedures that are used one after another

- Even if one does not necessarily provide input to the next.
- Weaker than sequential cohesion.

Temporal Cohesion

Operations that are performed during the same phase of the execution of the program are kept together, and everything else is kept out

- For example, placing together the code used during system start-up or initialization.
- Weaker than procedural cohesion.

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.

Question

Categorize the following aspects of design by the type of cohesion that they would exhibit if properly designed:

- **All information concerning an appointment booking (e.g. date, time, location) is kept inside a particular class**
- **A module is created to convert a bitmap image to a JPEG format**
- **A separate subsystem is created that runs every night to generate stats about the previous days sales**
- **A data processing operation involves receiving input from several sources, sorting it, summarizing information by input source, sorting according to the input source that generated the most data and then returning the results for the use of other subsystems. The code for these steps is all kept together.**

Another Question

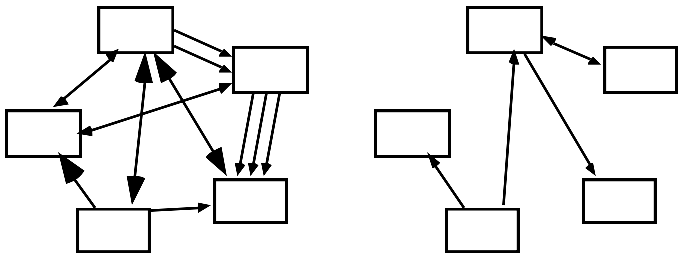
What is wrong with the following design from the perspective of cohesion and what could be done to improve it?

- **There are 2 subsystems in a university registration system that do the following:**
  - ***Subsystem A*** displays lists of courses to a student, accepts requests from the student to register in courses, ensures that the student has no schedule conflicts and is eligible to register in the courses, stores the data in the database and periodically backs up the database.
  - ***Subsystem B*** allows faculty members to input student grades, and allows administrators to assign courses to faculty members, add new courses, and change a students registration. It also prints the bills that are sent to students.

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 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

Common Coupling: Global Variable Communication

Functions communicating via global variables

- One sets variable, another refers to value

This is certainly possible in O-O languages

- One static object contains all "global data"
- This object's attributes are set and referred to

Disadvantage: consider debugging task: e.g.

- Global variable **count** has gotten some strange value
- Both **f** and **g** have been called
- Which changed it to the strange value?
- Must flip back and forth between **f** and **g** to find out
- Are there other functions which change **count**
- Have to keep whole program in mind when debugging

Avoiding Common Coupling

Don't declare variables as global

- Don't keep them in some public-accessible global object either

Pass them instead as parameters

Pass them only to those functions which really need them

This way:

- We know which functions change the data
- When there are problems with that data, we can ignore the functions which don't change it

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

Question: What OO concept should we be using to avoid control coupling like the example above?

Polymorphism FTW

```
public drawShape(Shape shape)
{
  shape.draw();
}

public abstract class Shape {
  public abstract void draw();
}

public class Circle extends Shape {
  public void draw() { … }
}

public class Rectangle extends Shape
{
  public void draw() { … }
}
```

Control Coupling: Another Example

```
void printReport (int which)  {
     switch (which) {
          case 1:  /* print disk space report */
          case 2:  /* print web usage report */
          …
     }
}
```

Person coding a function which calls `printReport` has to remember number, or go check `printReport` code

- Easier to remember function name than code number

To avoid:

- Break up functions doing two different things into individual ones with descriptive names

Control Coupling: Another Example

```
public class Report {
  public void printDiskSpaceReport();
  public void printWebUsageReport();
}
```

Polymorphism FTW

```
public abstract class Report {
  public abstract void print();
}

public class DiskSpaceReport extends Report {
  public void print() { … }
}

public class WebUsageReport extends Report {
  public void print() { … }
}
```

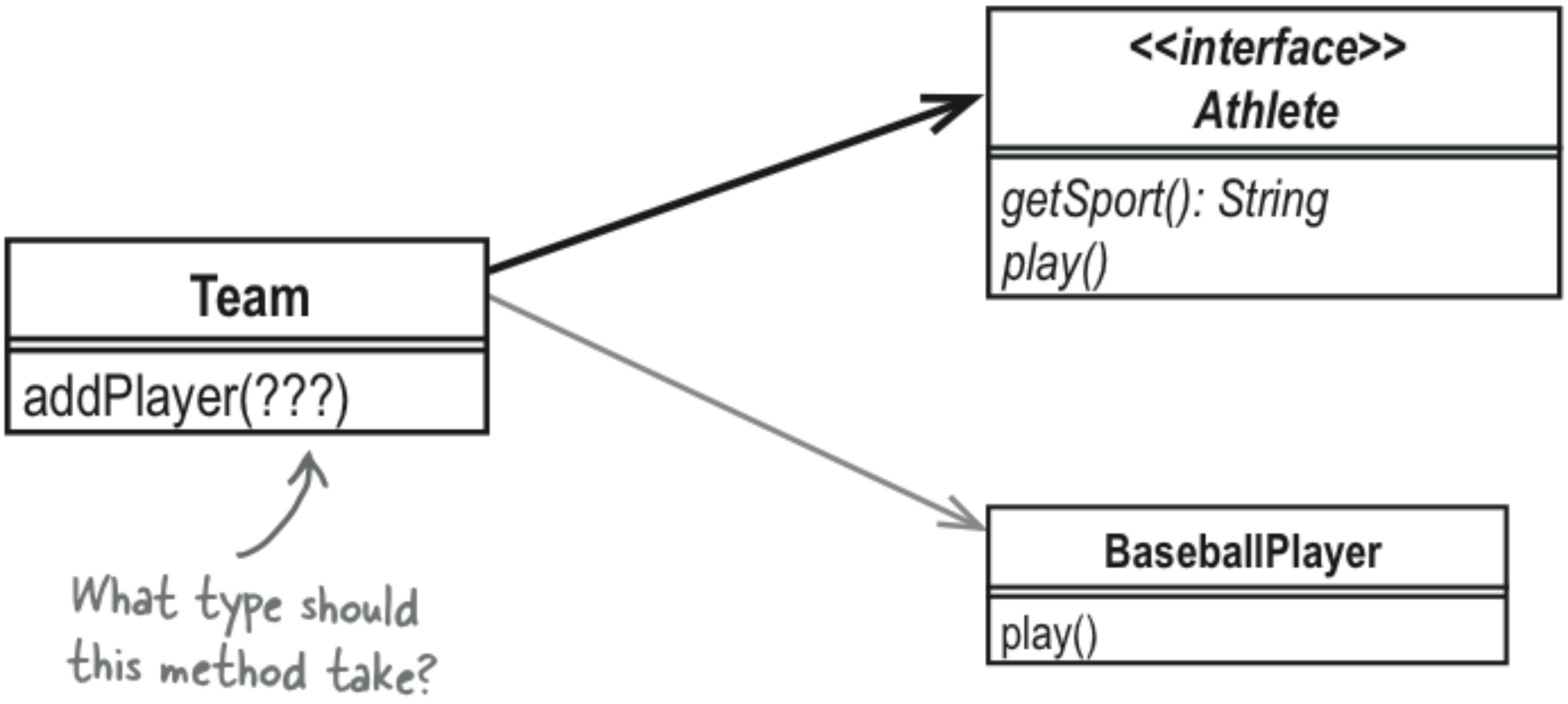
Stamp coupling

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

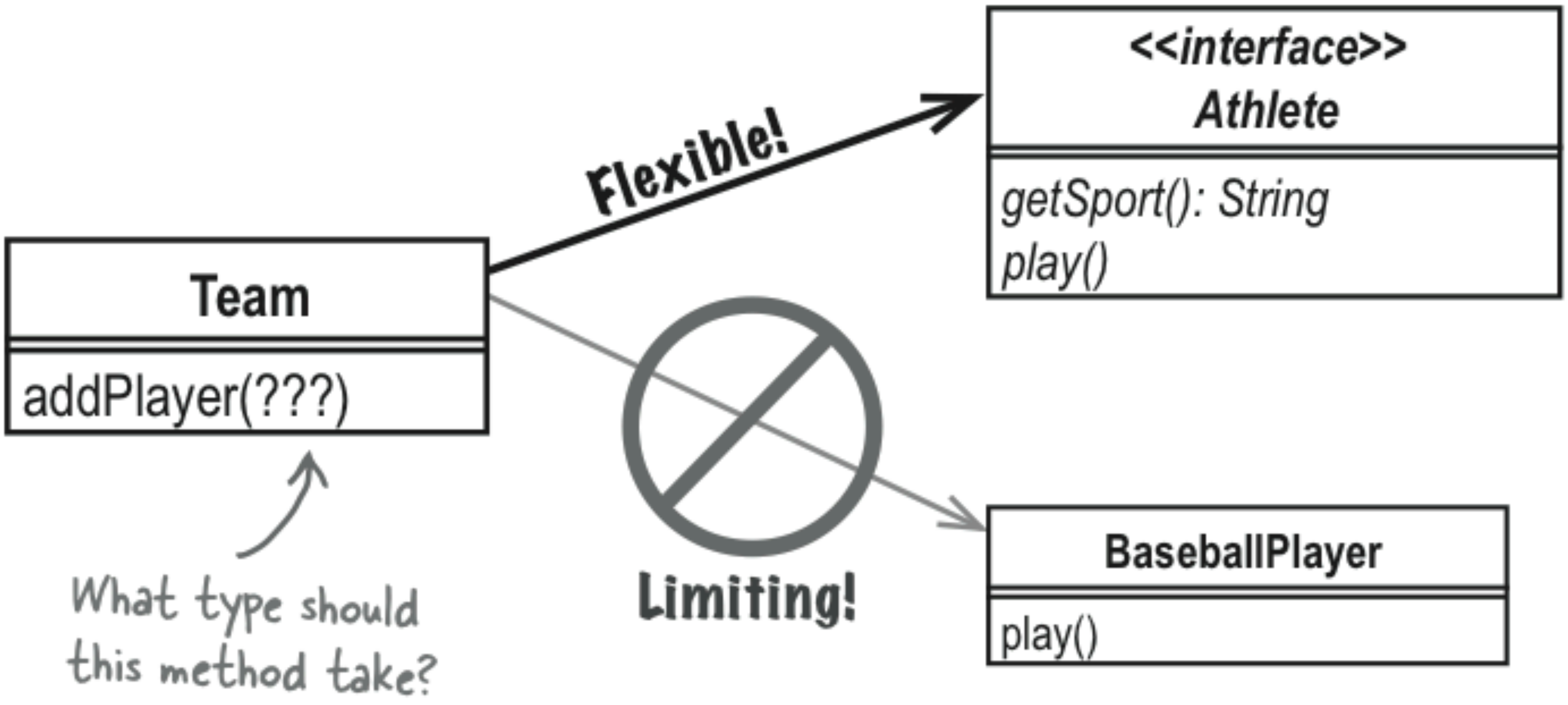
- Since one class now uses the other, changing the system becomes harder
  - Reusing one class requires reusing the other

- Two ways to reduce stamp coupling,
  - using an interface as the argument type
  - passing simple variables

Program to an Interface, Not an Implementation



Program to an Interface, Not an Implementation



Coupling Error → Stamp Coupling: Using Only a Piece

A function using only a small part of a parameter

Example: code for finding how much income tax to deduct from employee's salary:

```
int incomeTaxPayable(Person p) {
… /* code which refers to only p.salary */
}
```

- Function header leads us to believe that:
  - **incomeTaxPayable** uses all of **p**
  - We have to keep all of the **Person** fields in mind when debugging **incomeTaxPayable**

Question: How can we fix the above situation?

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 String getName();
  public 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.

Routine call coupling, what can we do?

Assume we have methods: aaa, bbb, ccc, and ddd, what could we do to help with the routine call coupling below?

```
private int aaa( …)
{
   …//more code
   ccc();
   ddd();
   ccc();
   …//more code
}
```

```
private int bbb( …)
{ ccc()
  ddd();
  ccc();
  …//more code
  ccc();
  ddd();
  ccc();
  …//more code
}
```

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 raises 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.

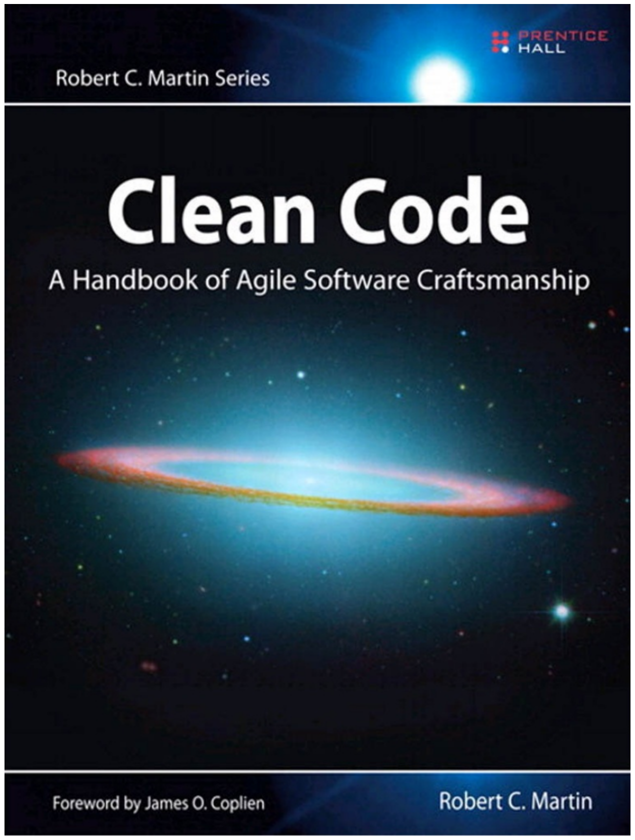
A Competing Opinion

J1: Avoid Long Import Lists by Using Wildcards

If you use two or more classes from a package, then import the whole package with

```
import package.*;
```

Long lists of imports are daunting to the reader. We don't want to clutter up the tops of our modules with 80 lines of imports. Rather we want the imports to be a concise statement about which packages we collaborate with.


PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

A Competing Opinion

Specific imports are hard dependencies, whereas wildcard imports are not. If you specifically import a class, then that class must exist. But if you import a package with a wildcard, no particular classes need to exist. The import statement simply adds the package to the search path when hunting for names. So no true dependency is created by such imports, and they therefore serve to keep our modules less coupled.

[…]

Wildcard imports can sometimes cause name conflicts and ambiguities. Two classes with the same name, but in different packages, will need to be specifically imported, or at least specifically qualified when used. This can be a nuisance but is rare enough that using wildcard imports is still generally better than specific imports.

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

Questions

Categorize the following aspects of a design by the types of coupling they exhibit:

•Class **CourseSection** has public class variables called **minClassSize** and **maxClassSize**. These are changed from time to time by the university administration. Many methods in classes Student and Registration access these variables.

•A user interface class imports a large number of Java classes, including those that draw graphics, those that create UI controls and a number of other utility classes.

•A system has a class called **Address.** This class has 4 public variables constituting different parts of an address. Several different classes, such as **Person** and **Airport** manipulate instances of this class, directly modifying the fields of address. Also, many methods declare one of their arguments to be an **Address**

Code Without Coupling Errors

Characteristics of code without coupling errors:

- Functions have low number of parameters
- Each parameter has an intuitive name and meaning
- Function uses all and only the data in the parameters

This is "data coupling": a good thing

We sometimes say that:

- Modules with coupling errors have ***high*** or ***tight*** coupling because they are very dependent on one another
- Modules without coupling errors have ***low*** or ***loose*** coupling because they are relatively independent

We therefore want *low* coupling between modules

Code Without Cohesion Errors

Characteristics of code without cohesion errors:

- Functions are relatively small
- Function names have clear intuitive meaning
- Functions actually do all and only what their names suggest

We sometimes say that:

- Modules with cohesion errors have *low* cohesion: they don't hang together very well
- Modules without cohesion errors have *high* cohesion: they hang together well

We want code with:

- Low coupling between functions / modules
- High cohesion within function / modules

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.

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

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