Design Heuristics and Architectural Styles

(LL Chapter 9)

Michel Chaudron

Leiden Institute of Advanced Computer Science

Many slides based on Lethbridge and Laganiere

Agenda

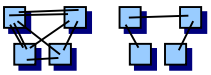
- Recap Design heuristics & guidelines
- Architectural Styles

- This afternoon: werkcollege use UML tools; location: PC zaal
- hand in assignments electronically
  chaudron@liacs.nl

Design Heuristics

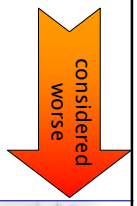
- Separation of Concerns
  - Information hiding
- Layering

- Modularity & Coupling



Types of Coupling

- Data coupling
  - data from one module is used in another
- Data type coupling
  - two modules use the same data type
- Control coupling
  - actions one module are controlled by another module (switch)
- Content coupling
  - a module refers to the internals of another module



Content coupling:

- Occurs when one component modifies data that is *internal* to another component
  - Reduce content coupling by *encapsulating* data
  - Information hiding
    - declare them `private`
    - and provide get and set methods

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);
  }
}
```

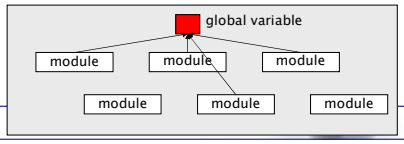
Information Hiding

- Usage of a module depends only on the information at the interface
- An interface should reveal as little as possible about the inner workings of the component
- An interface hides design decisions

D. L. Parnas, On the criteria to be used in decomposing systems into modules, *Communications of the ACM*, vol. 15, pp. 1053-1058, December 1972.

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



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
  - 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();
  }
}
```

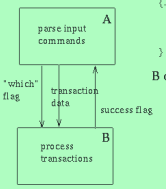
Caller needs to know:
Not drawCircle => draw Rectangle

Control Coupling Example

control coupling

Two modules are control coupled if they communicate using at least one "control flag"

Example:



A code:

```
{...
  call B(t_info, "update",ok)
  if not(ok) then ...
  ...
}
```

B code:

```
{receive parameters
    data, flag, outcome
  ...
  case (flag) of
    select: { ... }
    update: { ... }
    ...
    define: { ... }
    other: outcome := not ok
  end case
  outcome := ok
}
```

The behaviour of component B is controlled by component A through the parameter *flag*

Example from David Stotts
Dept. of Computer Science
University of North Carolina

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

class Employee
name: string address: string date-of-birth: date salary: number

```
public class Emailer
{
 public void sendEmail(Employee e, String message)
 {
    send(e.address, e.name, message)
 }
 ...
}
```


Example of stamp coupling

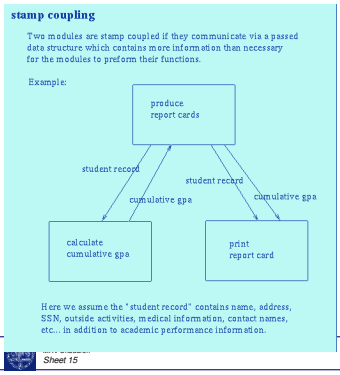
Using an interface to avoid stamp coupling

```
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)
 {...}
 ...
}
```


Stamp coupling Example



Example from David Stotts
Dept. of Computer Science
University of North Carolina

Data coupling

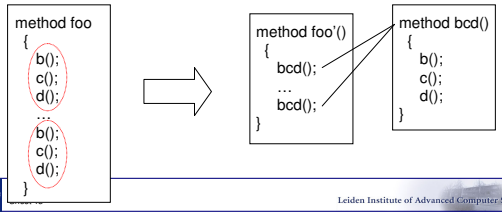
- Occurs whenever the types of method arguments are either primitive
  - The more arguments a method has, the higher the coupling
    - All methods that use the method must pass all the arguments
  - You should reduce coupling by not giving methods unnecessary arguments
  - There is a trade-off between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

Routine call coupling

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

Reduce Routine call coupling

- If you repetitively use the same sequence of 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 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.

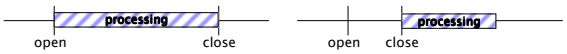
External coupling

- 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

Temporal Coupling

```
Program A
…
{
  …
  openfile(data)
  do_long_processing(data);
  closefile(data)
  …
}
```

```
Program B
…
{
  …
  openfile(data)
  closefile(data)
  do_long_processing(data);
  …
}
```



Temporal Coupling

A component **X** expects an input from component **Y** **every second**.

A component should handle all cases where attempts are made to use it inappropriately (be in intentionally or not).

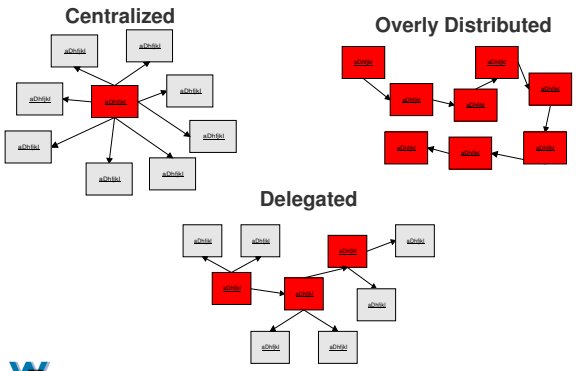
A RT-component should have a fall-back scenario:

If I don't receive an input, then I do 'plan B'.

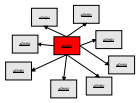
So that other components that depend on **X** will not also have to deal with this problem.

This is a way of 'fault containment' - prevent domino-effect.

Design of Control Styles



Characteristics of Centralized Control



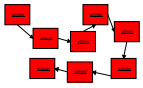
Centralized controllers can have extremely complex control logic

Controllers surrounded by simple information holders and service providers

These simple objects tend to have low-level, non-abstract interfaces

Drawback:
Changes can ripple among controlling and controlled objects

Characteristics of Overly Distributed Control

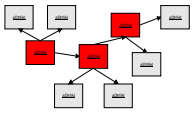


Long message chains to dig information out of information holders

Little or no value-added by those receiving a message and merely "delegating" request to next in chain

Drawback:
Hardwired dependencies between objects in call chain
May break encapsulation

Characteristics of Delegated Control



Coordinators know about fewer objects than dominating controllers

Higher level communications between objects

Benefits:
Changes typically localized and simpler
Easier to divide detailed design work

Interface Design

- An interface should reveal as little as possible about the inner workings of the component
- Users (callers) should depend only on the interface, not on the implementation

Recommended References:
- Effective Java: Programming Language Guide by Josh Bloch, Prentice Hall, 2001
  Check out video: http://www.infoq.com/presentations/effective-api-design
- Effective C++ by Scott Meyers, Addison-Wesley, 2005 (3rd ed).

Guidelines for Interface Design (1)

- *Completeness:*
  - include all functions
- *Essential/Minimal:*
  - omit needless features.
- *General:*
  - do not limit the applicability of an interface to its initial purpose as modules may be used in unexpected ways.
- *Consistency*
  - applies to many aspects of interface design such as naming conventions, parameter passing and exception handling.
- *Orthogonality:*
  - Keep independent features separately
  - Avoid offering the same service in multiple ways.
- *Open-ended:*
  - leave room for future expansion.
- *Opaqueness/Information-hiding:*
  - an interface should hide the details of the implementation.

Based on Hoffman [Hof90] based on o.a. Parnas.

Guidelines for Interface Design (2)

1. Keep interfaces *cohesive* and *small* (in that order)
2. Use *different interfaces* for users of the interface that play *different roles* with respect to the functionality
3. Don't combine *generic* and *specific* functionality in the same interface
4. Group *optional* functionality in *separate* interfaces
5. Avoid the introduction of *convenience functions*
6. Use *strongly typed* interfaces
7. Use *systematic naming* conventions

From Henk Jonkers c.s. Philips Research 2002

Guidelines for Naming Inventions

"…the relation of thought to word is not a thing but a process, a continual movement back and forth from thought to word and from word to thought. … Thought is not merely expressed in words; It comes into existence through them."

—Lev Vygotsky, *thought and language*

- **Fit a name into some naming scheme**
  - Java example: Calendar→ GregorianCalendar→JulianCalendar? ChineseCalendar?
- **Give service providers "worker" names**
  - Service providers are "workers", "doers", "movers" and "shakers "
  - Java example: StringTokenizer, ClassLoader, and Authenticator
- **Choose a name that suits a role**
  - Objects named "Manager" organize and pool collections of similar objects
  - AccountManager organizes Account objects

Guidelines for Naming Inventions

- **Choose names that don't limit behavior options**
  - Account or AccountRecord?
  - Record—information or facts set down in writing—an informational object
  - Account—sounds livelier—an object that makes informed decisions on the information it represents
- **Choose a name that suits a lifetime**
  - A ninety-year old named "Junior"?
  - ApplicationInitializer or ApplicationCoordinator?
- **Include facts most relevant to the users of a class**
  - MillisecondTimerAccurateWithinPlusOrMinusTwoMilleseconds or Timer?
- **Eliminate naming conflicts by adding description**
  - Rename a Properties candidate to TransactionHistoryProperties

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 Heuristics

Design defensively:
Do not trust that others will use your component as specified – each component should ensure its own integrity

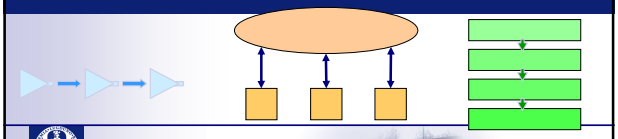
(from Lethbridge & Laganiere, p. 318)

A component should handle all cases where attempts are made to use it inappropriately:
- check whether all inputs are valid
- check preconditions

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

Architectural Styles



Leiden Institute of Advanced Computer Science

Theme/Objective of this lecture

The task of the architect is to come up with a good metaphor for the system
Alexander Ran (Nokia)

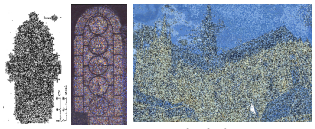
- Build vocabulary of architectural styles
  - a set of 'archetypes' that are often used
  - know their relative strengths and weaknesses
- Know when to apply or *not* to apply a particular style

CONTENTS

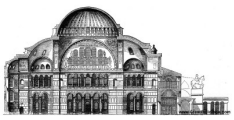
Architectural styles
- Client/Server
- Pipe and Filter style
- Blackboard style
- Publish Subscribe
- Peer-to-Peer

Architectural style

Nomenclature inspired by building architecture;
Buildings: Gothic, Byzantian, ...

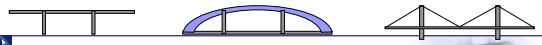


Cathedral Amiens



Hagia Sofia, Istanbul

bridges: suspension, arc, ... (check your Euro-notes)



http://nl.wikipedia.org/wiki/Architectural_style

Architectural style 1/2

An *architectural style* is defined by:
- A set of rules and constraints that prescribe
  - Which types of components, interfaces & connectors must/may be used in a system (vocabulary/metaphor) Possibly introducing domain-specific types
  - How components and connectors may be combined (structure)
  - How the system behaves (behaviour) The pattern of dependencies (control-flow and data-flow)
- A set of guidelines that support the application of the style (how to achieve certain system properties)

Architectural style

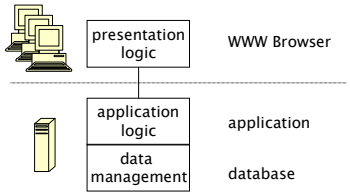
- Architectural styles are design paradigms for a set of design dimensions
  Some architectural styles emphasize different aspects such as: Subdivision of functionality, Topology or Interaction style
- Styles are open-ended; new styles will emerge
- Architectural styles are not disjoint
- An architecture can use several architectural styles

Client-Server Architectures

Nice source:
IT Architectures and Middleware:
Strategies for building Large Integrated Systems,
Chris Britton and Peter Bye, Addison Wesley, 2004

C/S Example: Thin Client

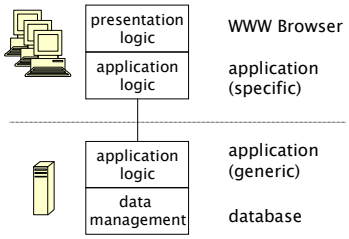
Thin Client C/S:
largest part of processing at the server-side



Network load: low
Config. Mngmnt: simple (only server)
Security: concentrated at server
Robustness: stateless clients => easy fault recovery

C/S Example: Thick Client

Thick Client:
significant processing
at the client-side



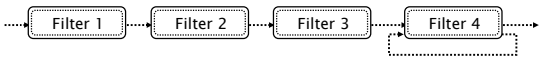
Network load: high
Config. Mngmnt: complex (both client & server)
Security: complex (both client & server)
Robustness: clients have state => complex fault recovery

C/S Benefits

Scalable
Interoperable

Pipe and Filter Style (1)

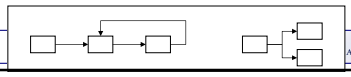
Concept: Series of filters / transformation
where each component is consumer and producer



Components: filters / transformations
possibly also: sources and sinks

Connectors: pipes;
interaction style: streaming of data

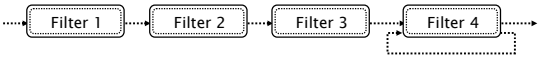
Topology: linear; possible variations:
feedback-loops, splitting pipes



Special types of filters

- **Pump (Producer)**
  Produces data and puts it to an output port that is connected to the input end of a pipe.
- **Sink (Consumer)**
  Gets data from the input port that is connected to the output end of a pipe and consumes the data.

Pipe and Filter Style (2)



Constraints about the way filters and pipes can be joined:

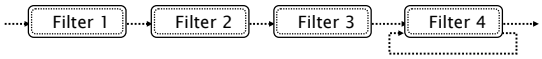
- Unidirectional flow
- Control flow derived from data flow

Behaviour Types:

a. **Batch sequential**
Run to completion per transformation

b. **Continuous**
Incremental transformation
variants: push, pull, asynchronous

Pipe and Filter Style (3)



Semantic Constraints

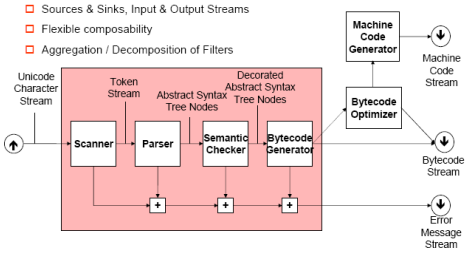
Filters are independent entities

- they do not share state
- they do not know their predecessor/successor

What are the dependencies between filters?
Compare this with Client Server?

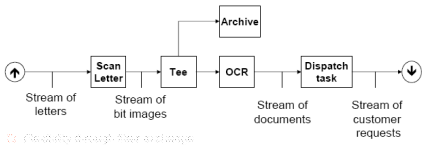
Example: P&F Compiler Architecture (1)

- Sources & Sinks, Input & Output Streams
- Flexible composability
- Aggregation / Decomposition of Filters



Example P&F Architecture

- No intermediate data structures necessary (but possible)
(Pipeline processing subsumes batch processing)



Pipe and Filter Style (4a)

Advantages:

- Simplicity:
  - no complex component interactions
  - easy to analyze (deadlock, throughput, …)
- Easy to maintain and to reuse
- Filters are easy to compose (also hierarchically?)
- Can be easily made parallel or distributed

Pipe and Filter Style (4b)

Disadvantages:

- Interactive applications are difficult to create
- Filter ordering can be difficult
- Performance:
  - Enforcement of lowest common data representation, ASCII stream, may lead to (un)parse overhead
  - If output can only be produced after all input is received,an infinite input buffer is required (e.g. sort filter)
- If bounded buffers are used, deadlocks may occur

Pipe and Filter Style (5) Quality Factors

Extendibility: extends easily with new filters

Flexibility: – functionality of filters can be easily redefined,
– control can be re-routed
(both at design-time, run-time is difficult)

Robustness: 'weakest link' is limitation

Security: –

Performance: allows straightforward parallelisation

Pipe and Filter Style (6) Application Context

Rules of thumb for choosing pipe-and-filter (o.a. from Shaw/Buschman):

- if a system can be described by a **regular interaction pattern** of a collection of processing units at the same level of abstraction; e.g. a series of incremental stages (horizontal composition of functionality);
- if the computation involves the **transformation of streams of data** (processes with limited fan-in/fan-out)

Hint: use a looped-pipe-and-filter if the system does continuous controlling of a physical system

Typical application domain: signal processing