Introduction to Software Technology
3. Software Design

Klaus Ostermann

Goal of Software Design

- For each desired program behavior there are infinitely many programs that have this behavior
  - What are the differences between the variants?
  - Which variant should we choose?
- Since we usually have to synthesize rather than choose the solution…
  - How can we design a variant that has the desired properties?

Example

- Sorting with configurable order, variant A

```
void sort(int[] list, String order) {
    …
   boolean mustswap;
   if (order.equals(“up”)) {
      mustswap = list[i] < list[j];
   } else if (order.equals(“down”)) {
      mustswap = list[i] > list[j];
   }
   …
}
```

Example

- Sorting with configurable order, variant B

```
void sort(int[] list, Comparator cmp) {
   …
  boolean mustswap;
  mustswap = cmp.compare(list[i], list[j]);
  …
}
interface Comparator {
  boolean compare(int i, int j);
}
class UpComparator implements Comparator {
  boolean compare(int I, int j) { return i<j; }}

class DownComparator implements Comparator {
  boolean compare(int I, int j) { return i>j; }}
```

(by the way, this design is called “strategy pattern”)

Quality of a Software Design

- How can we measure the internal quality of a software design?
  - Extensibility, Maintainability, Understandability, Readability, …
  - Robustness to change
  - Low Coupling & High Cohesion
  - Reusability
  - All these qualities are typically summarized by the term **modularity**
- …as opposed to external quality
  - Correctness: Valid implementation of requirements
  - Ease of Use
  - Resource consumption
  - Legal issues, political issues, …

Modularity

Modularity

- A software construction method is modular if it helps designers to produce software systems made of autonomous elements connected by a coherent, simple structure

- In the following we'll elaborate on that:
  - Five criteria
  - Five Rules
  - Five Principles

Five Criteria: Modular Decomposability

A software construction method satisfies Modular Decomposability if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them.

Five Criteria: Modular Decomposability

- Modular Decomposability implies: Division of Labor possible!
- Example: Top-Down Design
- Counter-Example: Production of a global initialization module

Five Criteria: Modular Composability

A method satisfies Modular Composability if it favors the products of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed.

Five Criteria: Modular Composability

- Is dual to modular decomposability
- Is directly connected with reusability
  - Old “dream” of programming: programming as construction box activity
- Example 1: Libraries have been reused successfully in countless domains
- Example 2: Unix Shell Commands
- Counter-Example: Preprocessors

Five Criteria: Modular Understandability

A method favors Modular Understandability if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others.

Five Criteria: Modular Understandability

- Important for maintenance
- Applies to all software artifacts, not just code
- Counter-example: Sequential dependencies between modules

Five Criteria: Modular Continuity

A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in the problem specification will trigger a change of just one module, or a small number of modules.

Five Criteria: Modular Continuity

- Example 1: Symbolic constants (as opposed to magic numbers)
- Example 2: Hiding data representation behind an interface
- Counter-Example: Program designs depending on fragile details of hardware or compiler

Five Criteria: Modular Protection

A method satisfied Modular Protection if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules.

Five Criteria: Modular Protection

- Motivation: Big software will always contain bugs etc., failures unavoidable
- Example: Defensive Programming
- Counter-Example: An erroneous null pointer in one module leads to an error in a different module

Five Rules

- Five Rules will follow which we must observe to ensure high-quality design

Five Rules: Direct Mapping

The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modeling the problem domain.

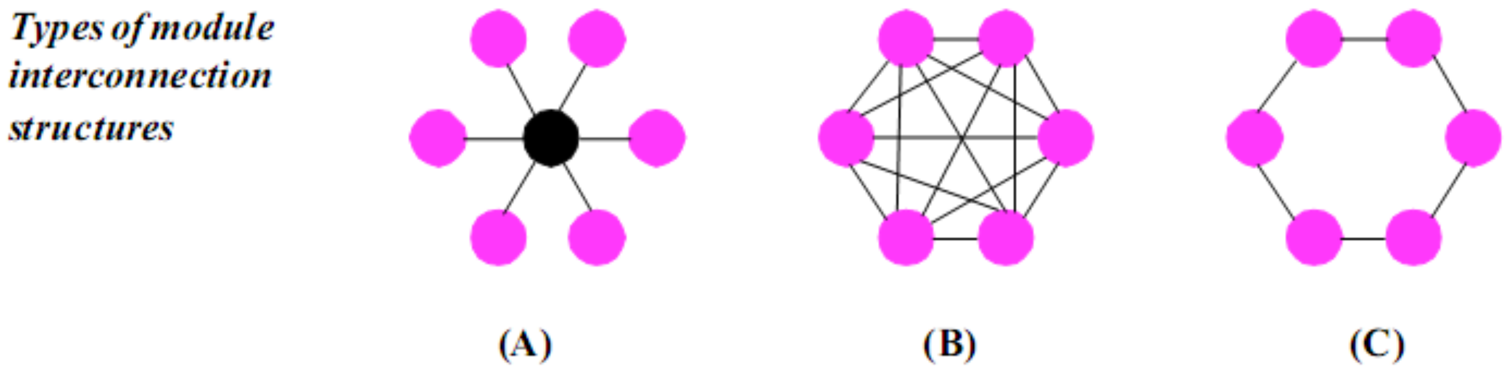
Five Rules: Direct Mapping

- Follows from continuity and decomposability
- A.k.a. “low representational gap”[C. Larman]

Five Rules: Few Interfaces

If two modules communicate, they should exchange as little information as possible

Five Rules: Few Interfaces



- Want topology with few connections
- Follows from continuity and protection; otherwise changes/errors would propagate more

Five Rules: Small Interfaces

If two modules communicate, they should exchange as little information as possible

Five Rules: Small Interfaces

- Follows from continuity and protection, required for composability
- Counter-Example: Big Interfaces ☺

Five Rules: Explicit Interfaces

Whenever two modules A and B communicate, this must be obvious from the interface of A or B or both.

Five Rules: Explicit Interfaces

- Counter-Example 1: Global Variables
- Counter-Example 2: Aliasing – mutation of shared heap structures

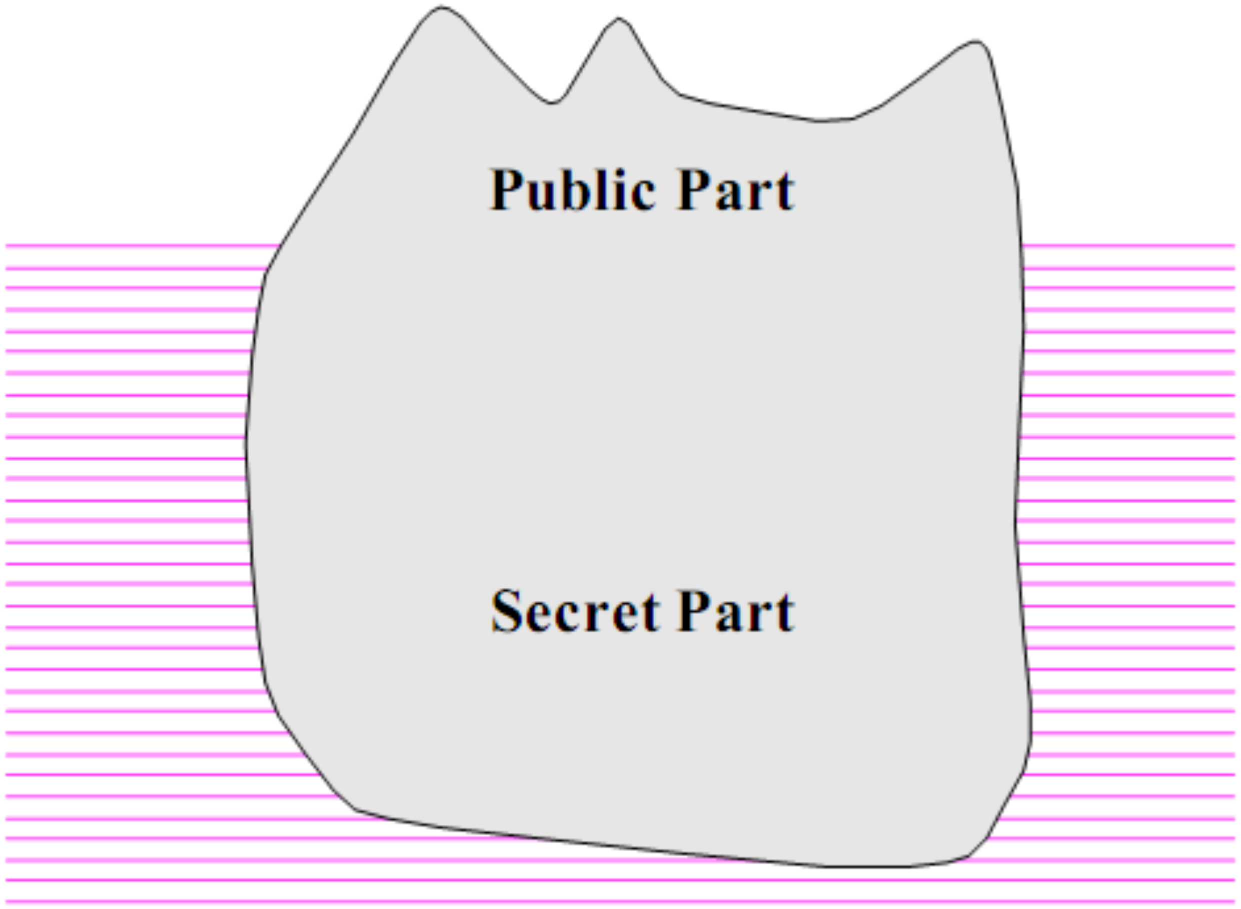
Intermezzo: Law of Demeter (LoD)

- LoD: Each module should have only limited knowledge about other units: only units "closely" related to the current unit
- In particular: Don’t talk to strangers!
- For instance, no a.getB().getC().foo()
- Motivated by continuity

Five Rules: Information Hiding

The designer of every module must select a subset of the module’s properties as the official information about the module, to be made available to authors of client modules.

Five Rules: Information Hiding



Five Rules: Information Hiding

- Reynolds’ parable about complex numbers…
- Implied by continuity
- The iceberg analogy is slightly misleading, since an interface also *abstracts* over the implementation

Five Principles

- From the preceding rules, and indirectly from the criteria, five principles of software construction follow:
  - The Linguistic Modular Units principle.
  - The Self-Documentation principle.
  - The Uniform Access principle.
  - The Open-Closed principle.
  - The Single Choice principle.

Five Principles: Linguistic Modular Units

Modules must correspond to syntactic units in the language used.

Five Principles: Linguistic Modular Units

- Excludes methods that suggest a certain module concept and a language that does not offer the corresponding modular construct
- Implied by continuity and direct mapping
  - Both require direct correspondence between specification, design, and implementation modules
- Implied by decomposability and composability
  - The implementation of every task must result in a well-delimited syntactic unit

Five Principles: Self-Documentation Principle

The designer of a module should strive to make all information about the module part of the module itself.

Five Principles: Self-Documentation Principle

- Precludes keeping information about the module in a separate document
- Justification:
  - Modular understandability principle
  - Continuity, hard to keep separate documents “in sync”
  - In general: Changeability
- Traditional “heavy-weight” SE processes have a different point of view on this
- See also material of previous lecture about literate programming and good documentation in general

Five Principles: Uniform Access

All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation

Five Principles: Uniform Access

- Justification: Continuity criterion, special case of information hiding
- Example: The balance of an account may be stored as data, or it may be computed from the list of transactions
  - This is a time/space tradeoff
  - Difference should not be visible for a client
- Some languages support this principle directly
  - Ruby, Eiffel, Python(*), Smalltalk(*)
  - A design convention in other languages
    - “getter/setter” methods in Java

Five Principles: Open-Closed Principle

Modules should be both open and closed.

Five Principles: Open-Closed Principle

- A module is said to be open if it is still available for extension.
  - For example, it should be possible to expand its set of operations or add fields to its data structures.
- A module is said to be closed if it is available for use by other modules.
  - Well-defined stable interface
  - Can be compiled, stored, …
- Motivation: Openness for future extensions, closedness for composition

Five Principles: Open-Closed Principle

- Example: Classes in OO languages are open through inheritance yet can be used through constructor calls
- Counter-Example: Packages in Java
- What happens if modules are not open:
  - “Monkey patching” in Javascript

```
eval("getBrowser().removeTab ="+
  getBrowser().removeTab.toString().replace(
    'this.addTab("about:blank");',
    'if (SpeedDial.loadInLastTab) {this.addTab('
  +'"chrome://speeddial/content/speeddial.xul"'
  +')} else { this.addTab("about:blank")}'
));
```

(you don’t need to understand this example in detail)

Five Principles: Single Choice

Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.

Five Principles: Single Choice

- Special case of the DRY principle (Don’t repeat yourself):
  - Every piece of knowledge must have a single, unambiguous, authoritative representation within a system
- Typical examples of violations of this principle:
  - Multiple If/then/else or case statements with identical conditions
  - Pattern Matching in functional programming
- Is a consequence of open-closed principle
- Is a form of information hiding

Five Principles: Single Choice

- Avoided in OO languages by using subtyping and late binding
  - Cf. sorting example in the beginning of this presentation
  - However, OO violates the principle itself in that the list of methods in an interface is closed and replicated in all implementations
- Simple principle but quite hard to realize
  - Cf. solutions to the so-called ‘expression problem’

Discussion

- Examine the modular structures of any programming language which you know
- Assess how they support the criteria and principles presented in this lecture

Reusability

Reusability

- Old dream of “mass-produced software components” by McIlroy at NATO conference
  - Idea: Mimic engineering
  - However, cf. pitfalls of confusing product and plan, see first lecture
- Other (equally problematic) common analogy: Software as LEGO
- Although these visions are both problematic, reuse is a major concern in software design

Goals of Reuse

- Timeliness: Having less software to develop means that we can build it faster
- Decreased maintenance effort: If someone else is responsible for the software he or she is also responsible for its evolution
  - And he does so once for all clients: Fix once, profit many times
- Redundancy is a source of inconsistency. Reuse can avoid redundancy.
- Reliability
  - If software is reused many times more effort is put into software quality
- Efficiency
  - By the same arguments as reliability
- Consistency
- Investment

What should we reuse?

- Reuse of personnel
  - E.g. avoid loss of know-how by transferring software engineers from project to project
- Reuse of designs and specifications
  - Notion of design/specification as software product independent of implementation is dubious, cf. self-documentation principle
  - The implementation *is* the (detailed) design
  - With continuity, direct mapping etc., the distinction between reusing modules and reusing designs tends to fade away
    - High-Level Design (e.g. class diagram) is like a table of contents of the detailed design (implementation)

What should we reuse?

- Design Patterns
  - = architectural ideas applicable across a broad range of application domains
  - An important form of architecture reuse which we will study in detail later on
- Source Code
  - We want to reuse something that “runs”, since eventually we want to run programs (“Bubbles don’t crash!”)

Reuse of abstracted modules

- If we want to reuse source code, in what form should we reuse it?
- Reusing it in the form of the actual source text is problematic
  - It removes information hiding: Users may rely on implementation details only visible by studying the source text
  - Developers of software distributed in source text may be tempted to violate modularity rules
- What we want to reuse are abstracted modules
  - Software described through a well-defined interface that hides implementation details
  - The software may still be distributed in source text; the difference is the primary source of information about it

Reuse and Patterns

- A reusable piece of code abstracts over certain patterns (patterns in the general sense, not design patterns)
- Conversely, a program contains opportunities for reuse if it contains patterns
  - Can be formalized as: The Kolmogorov complexity of the program is smaller than the program
- Let’s look at some typical patterns of redundancy (patterns of patterns ☺) and ways to abstract over them

Reuse: Constants

- Sometimes the same number/string/… shows up many times in a program (“magic number”)
  - A source of inconsistency, if the number/string may change
    - E.g., the value of Pi is unlikely to change ☺
  - No documentation of the intention of the number/string/…
- Reuse by defining a constant
  - Can reuse the constant in all places where it is needed
  - Can change in one place
  - Can give meaningful name to constant

Reuse: Common Subexpressions

- Code may contain the same subexpressions in many places, e.g., **`5*89+3`** or **`x+y/2`**
  - Must keep track of scoping rules if variables are involved
- Can abstract over pattern by using a local variable
- If the subexpression performs side-effects (I/O, mutation, …) can abstract over pattern using procedure/method
- This may or may not yield more efficient code
  - Depending on how smart the compiler is
- More importantly, it will typically improve the code
  - Gives the idea behind the common subexpression a name
  - Less redundancy, …

Reuse: Almost common subexpressions

- Code contains similar, but not identical subexpressions
  - Differ only in the value of “first-class” expressions
- Can abstract over pattern using procedural abstraction (or, methods, functions, lambdas, …)
- E.g. average(x,y) = x+y/2 instead of computing averages in place
- Gives abstraction a name, avoids redundancy, …

Reuse: Almost common subexpressions

- What if the expressions differ in the subroutines they call?
  - Nothing changes in a language with first-class subroutines; can abstract over these calls and turn them into parameters
    - E.g. functional languages
  - In OO languages, this problem has given rise to design patterns such as “strategy” and “template method” which we will discuss in detail later on

Reuse: Almost common subexpressions

- What if the expressions differ in the types they use?
  - E.g. quicksort algorithm on integers vs. quicksort on strings
- Can use generic types!
  - List<T> sort<T>(List<T> in, Comparator<T> cmp) {…}

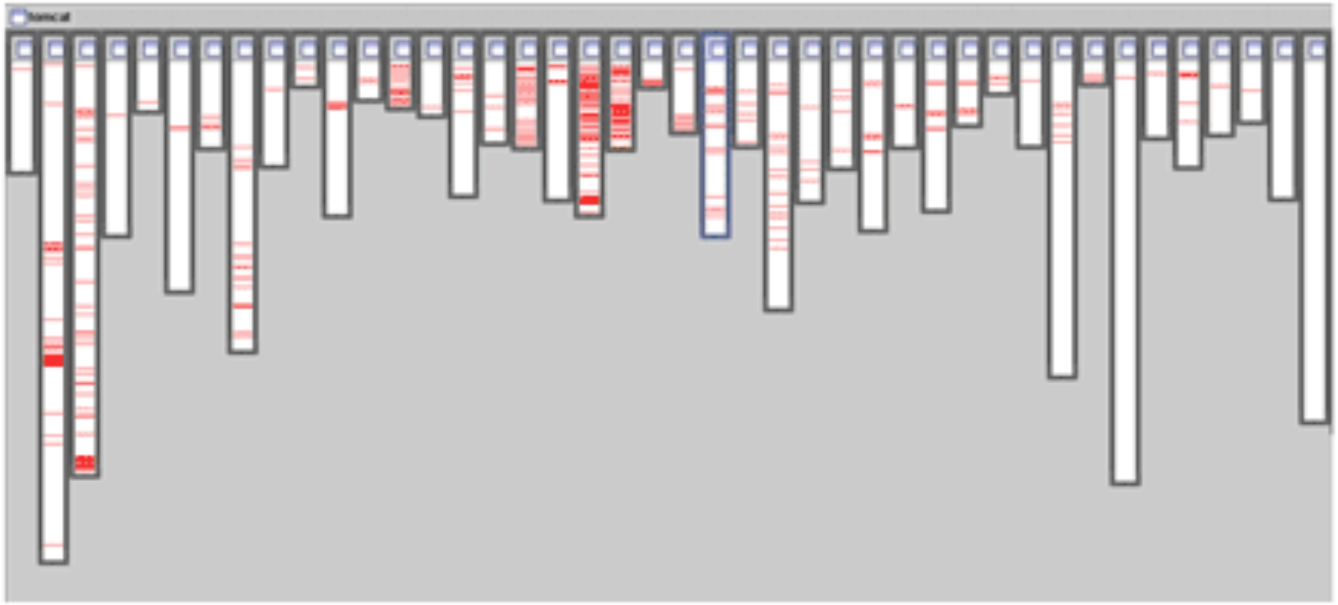
Reuse: Similar class definitions

- Can factor out commonalities in common superclass
- Late binding and possibility to extend fields/methods allow one to specialize in powerful ways
  - However, subtyping limits the ways how a subclass can be different from a superclass
    - “Liskov Substitution Principle” → SE Design Lecture
- However, different points of view on when to use inheritance
  - “Scandinavian” style of inheritance: Mechanism for conceptual specialization, driven by modeling
  - “American” style of inheritance: Reuse and patch code

Some patterns are hard to abstract over, though

problems like…

logging is not modularized



- **where is logging in org.apache.tomcat**
  - red shows lines of code that handle logging
  - not in just one place
  - not even in a small number of places

aspectj.org

Reuse and Efficiency

- Abstract designs can be less efficient (in terms of space/ time behavior) than their redundant expansions
  - "All design is just adding more levels of indirection"
  - A conflict between reuse and efficiency?
- Most of the time not a real concern
  - Compiler techniques remove much abstraction overhead
    - Inlining, devirtualization, tail-call optimization, partial evaluation, staging, …
  - Cost of method/function calling rarely the main bottleneck
  - Only a fraction of the code is performance-critical anyway
- Can be a concern if very inefficient abstraction techniques are used
  - E.g., reflection

Literature

- Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, 1997 [Chapter 3, 4]