Programming Language Design

10/14 Language Support

Recall: Language construct

- The built-in elements given by a programming language to construct your programs
  - Rather than a piece of code

- Let you easily use a mechanism in the language
  - make code clear and shorter
  - easy to reuse and unplug

Recall: for construct

- We have to repeat the code
  - not only annoying but also error-prone
- for example, add values in arrays
  - with for

```
for(int i=0; i<5; i++)
  C[i] = A[i] + B[i];
```

  - without for

```
C[0] = A[0] + B[0];
C[1] = A[1] + B[1];
C[2] = A[2] + B[2];
C[3] = A[3] + B[3];
C[4] = A[4] + B[4];
```

another example: class construct

- Think about another example:
  without class construct?
  - *the example of for construct is too trivial*

- Implementing dynamic dispatch by ourselves!
  - *E.g. OO in C language*

class construct

- A construct for class in OOP
  - used to generate objects that share the same properties
  - used to hide OO mechanisms from you
- Without it?
  - manually generate every object
  - implement OO mechanism by ourselves

In other words, writing programs

- Should benefit from mechanisms supported in the language
  - by using the given language construct
- However, not all mechanisms are supported, of course
  - implement by yourself
  - refer to design patterns

Design pattern

- A general solution to a common problem in your software design
  - of course not just repeat code
- Explain the problem in detail
  - What you have and what you want to do
  - Define the roles in the problem
  - Describe how you should implement these roles

Design pattern (cont.)

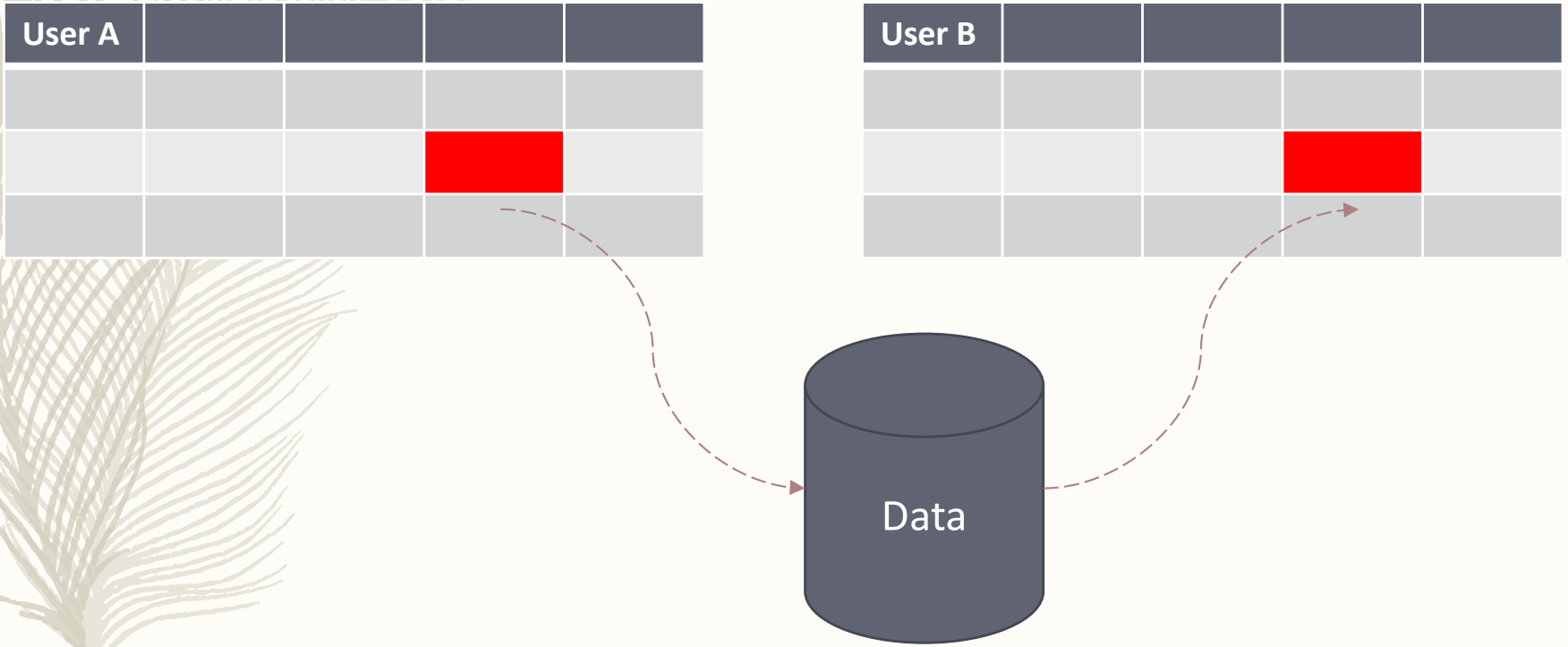
- Reuse as “pattern”
  - Follow it to implement again
  - Cannot be simply reused as libraries
- Usually due to the limitations in the language
  - Lack of dedicated constructs supporting it
  - Cannot simply resolve the problem with existing language support

An example

- Suppose we need a notification mechanism
  - The subject may publish some information
  - The observers can subscribe to it to get notification
  - E.g. a GUI frontend, a warning system, etc.

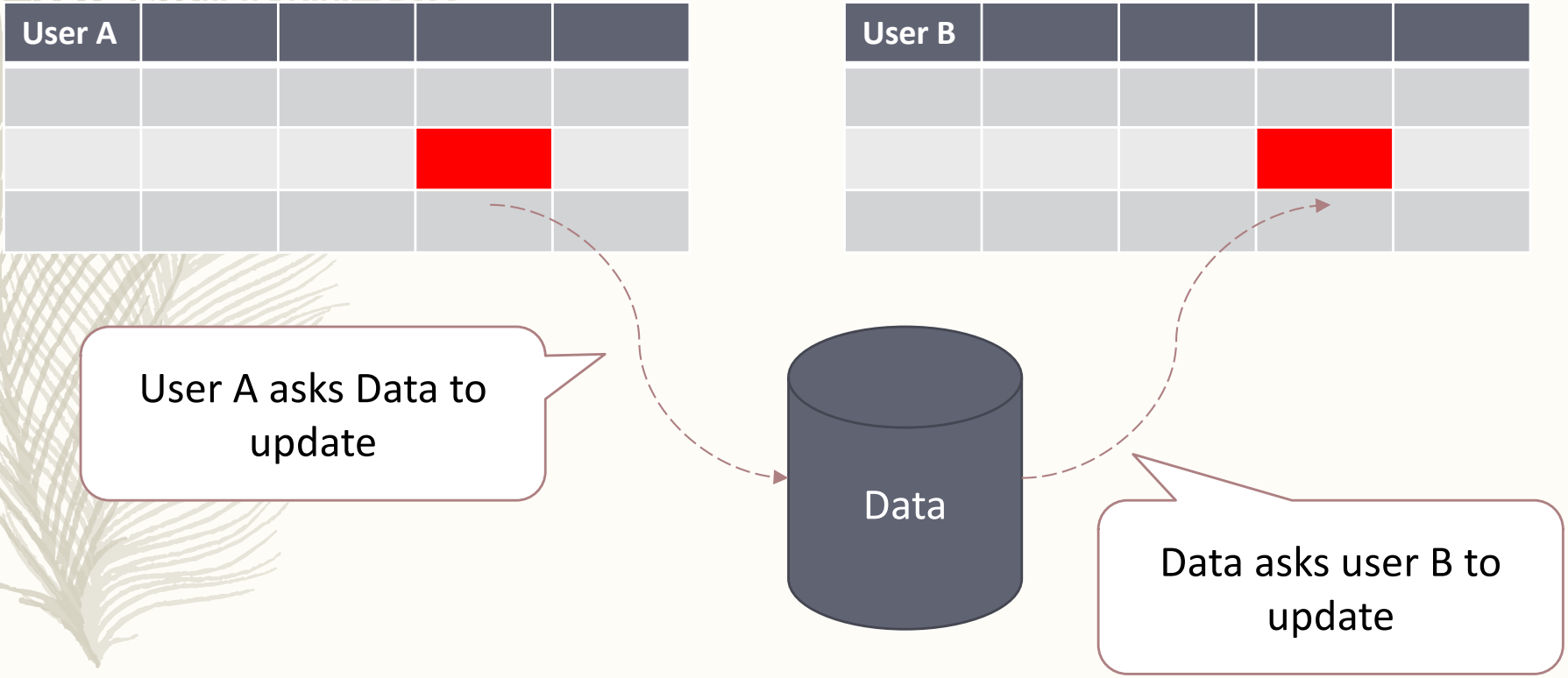
Data and its GUI frontends

- Data are stored in files or database
  - User A made a change
  - Notify of the change for User B



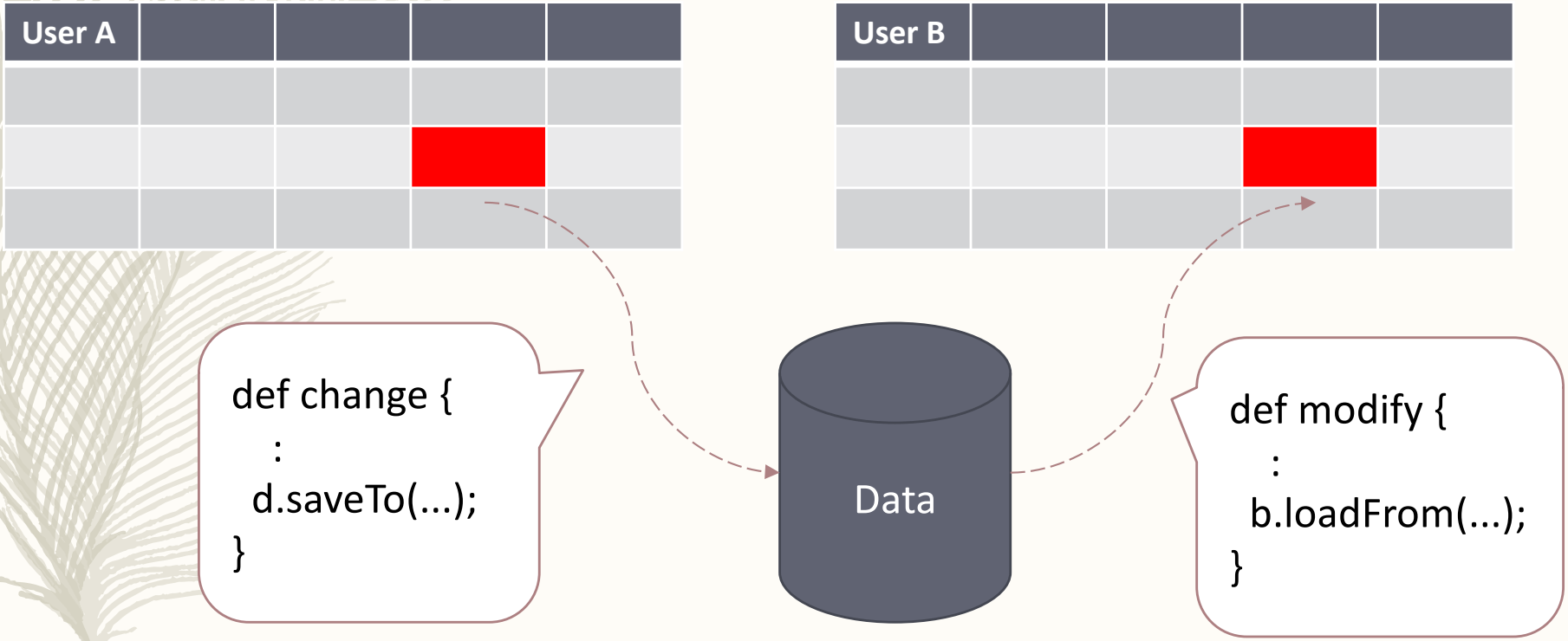
Use objects to describe

- User A, User B, Data are objects
- Operations for updating are methods inside these objects



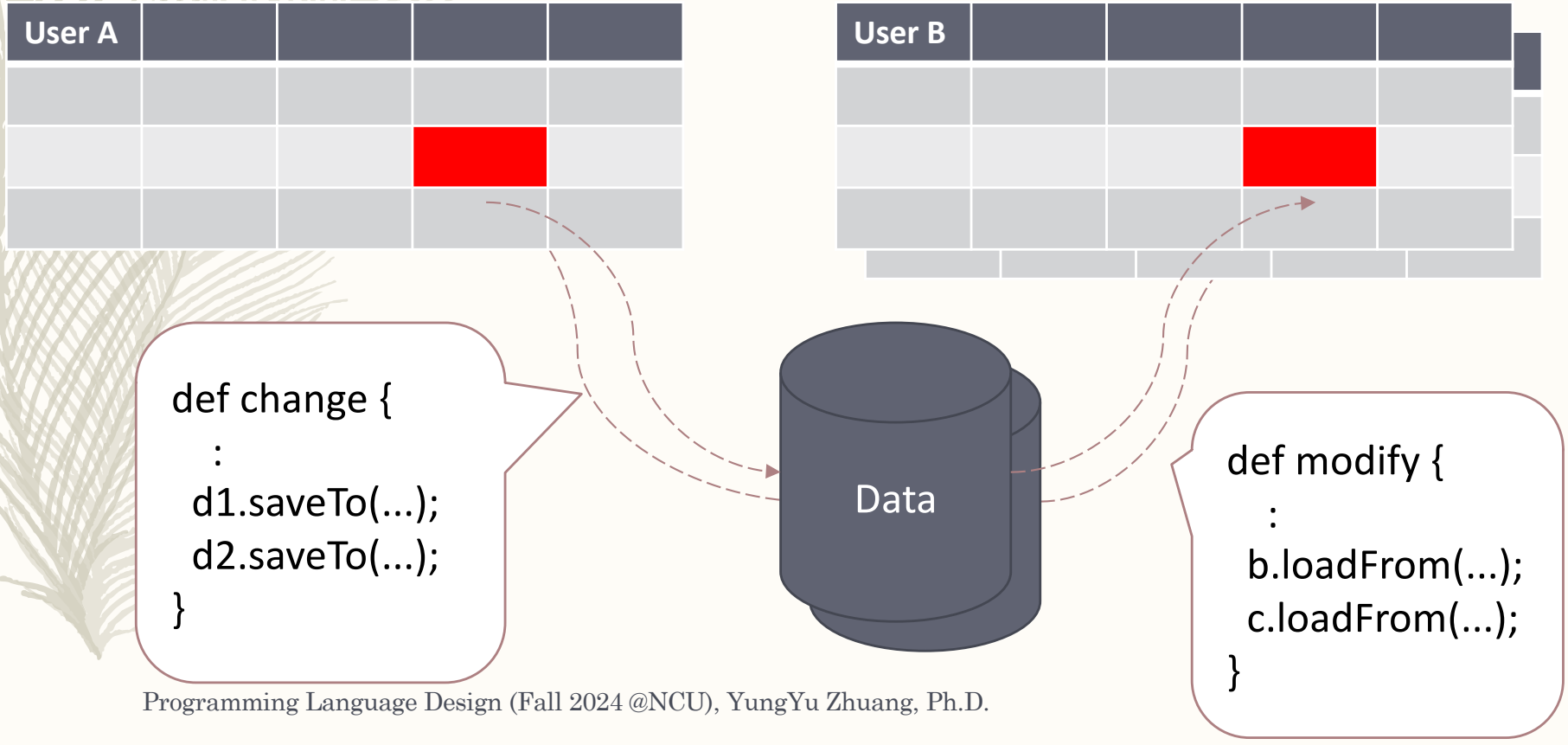
Hard-code??

- In User A, simply call a function in Data to update?
- In Data, simply call a function in User B to update?



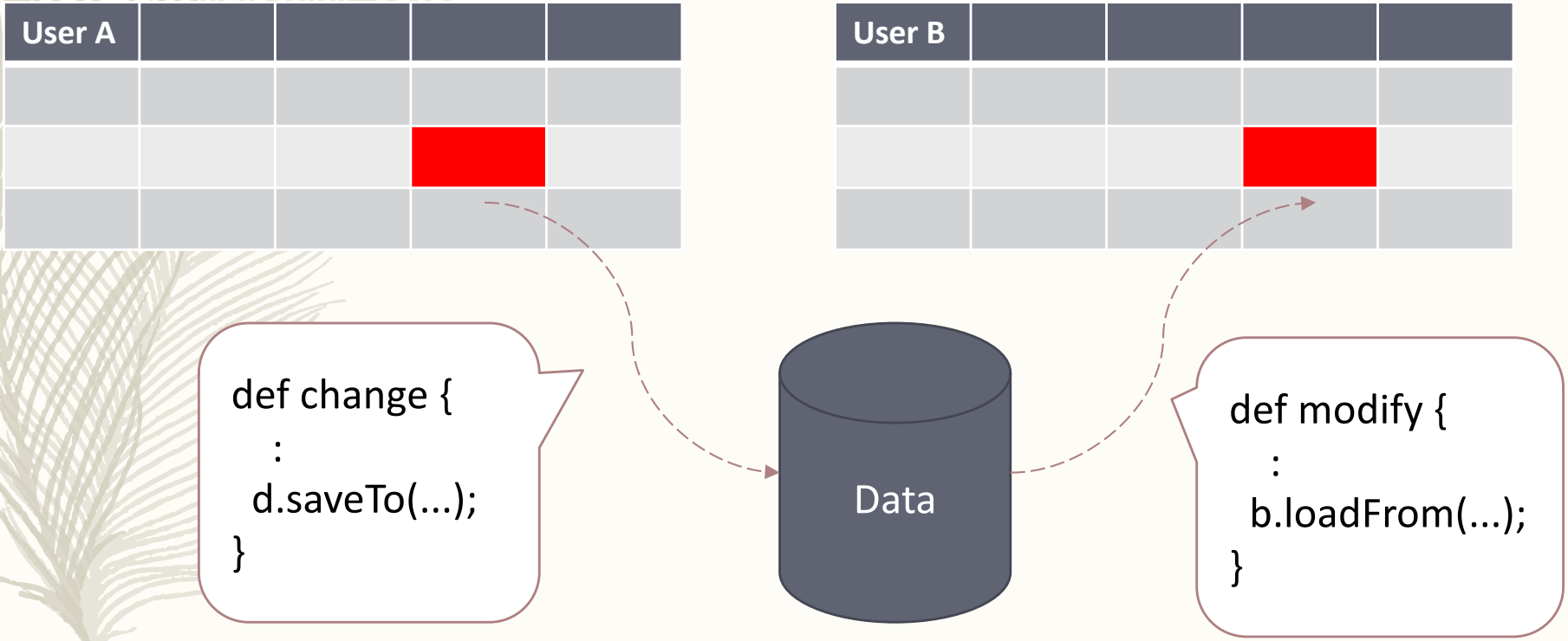
Drawbacks of hard-code

- for multiple data, we need to repeat "saveTo" several times
  - how to modify "saveTo" calls to "saveAs" smartly?



Drawbacks of hard-code (cont.)

- for each object, we have to write down its name
  - e.g. d and b
  - how to modify "d" to "d2" smartly?

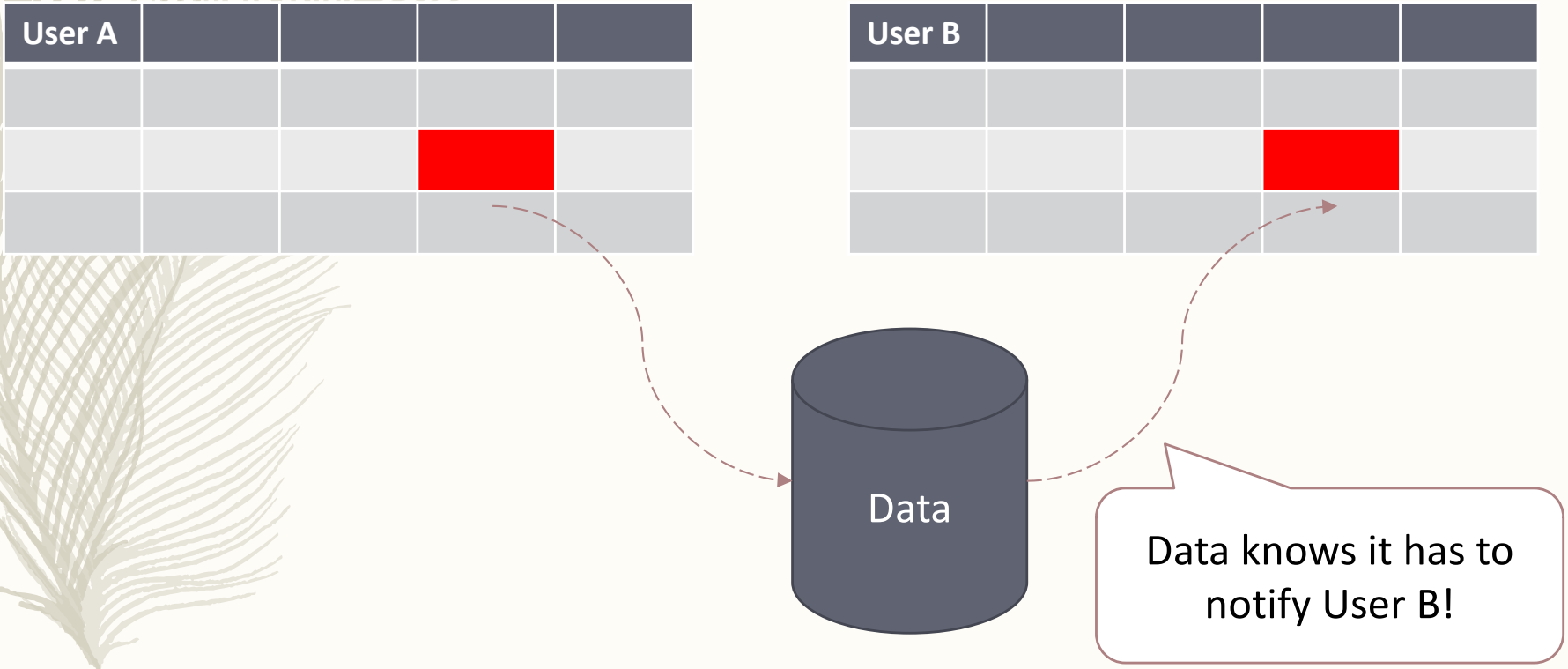


How to implement publish/subscribe?

- Of course you can implement with polling
  - Let observers ask the subject every 5 minutes
- Time delay: in the worst case observers might wait almost 5 minutes
- Waste of resource: unnecessary handling when nothing is published

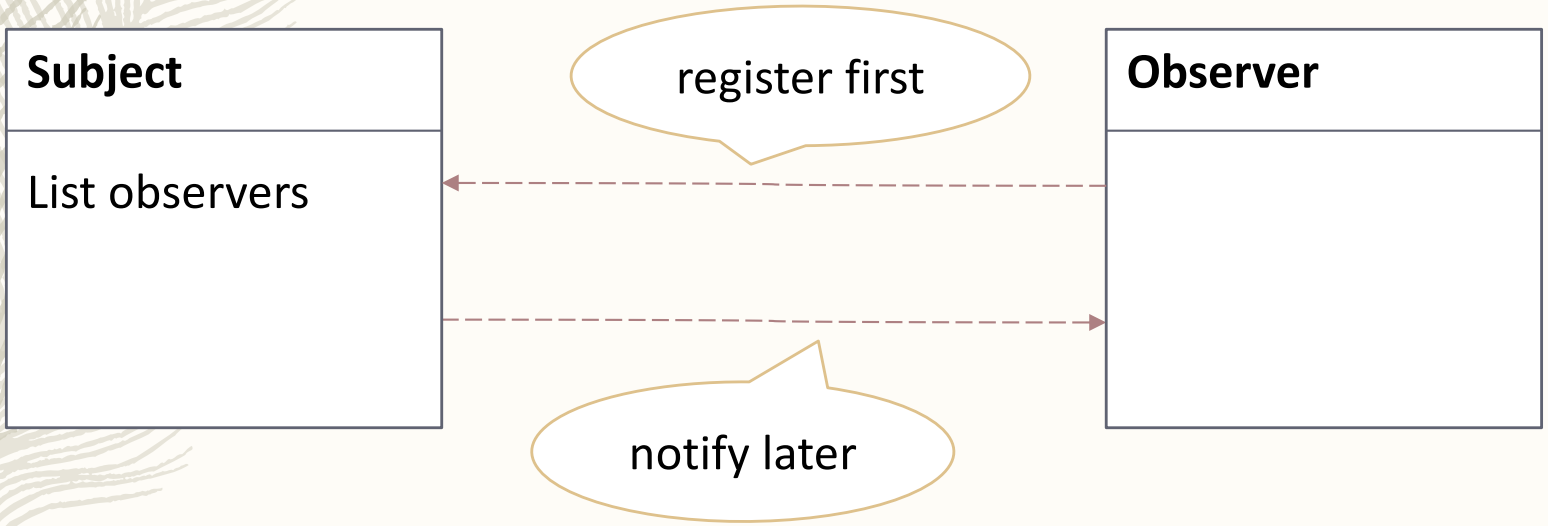
Maintain a list for notification

- If Data maintains a list for notification
  - Data knows who it has to notify
  - easily to add/remove User B into/from the list



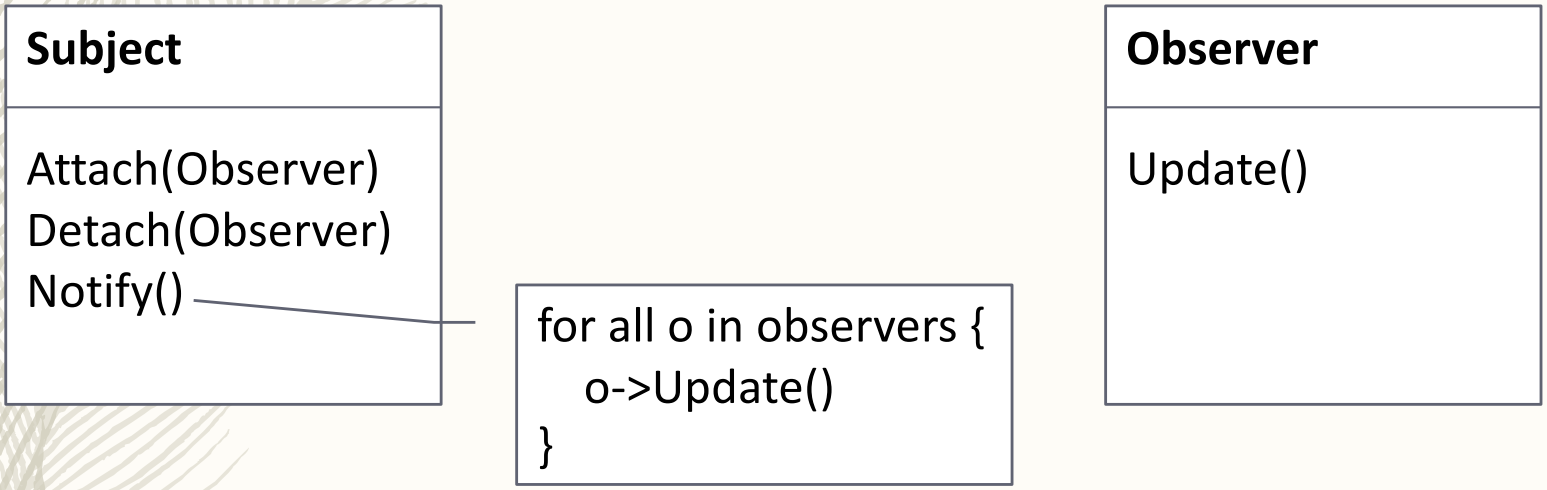
The Observer pattern

- Subject maintains an observer list
  - To remember which observer should be notified
- Observers can register themselves
  - To get notifications from subject



The Observer pattern (cont.)

- Subject maintains an observer list
- Observers can register themselves

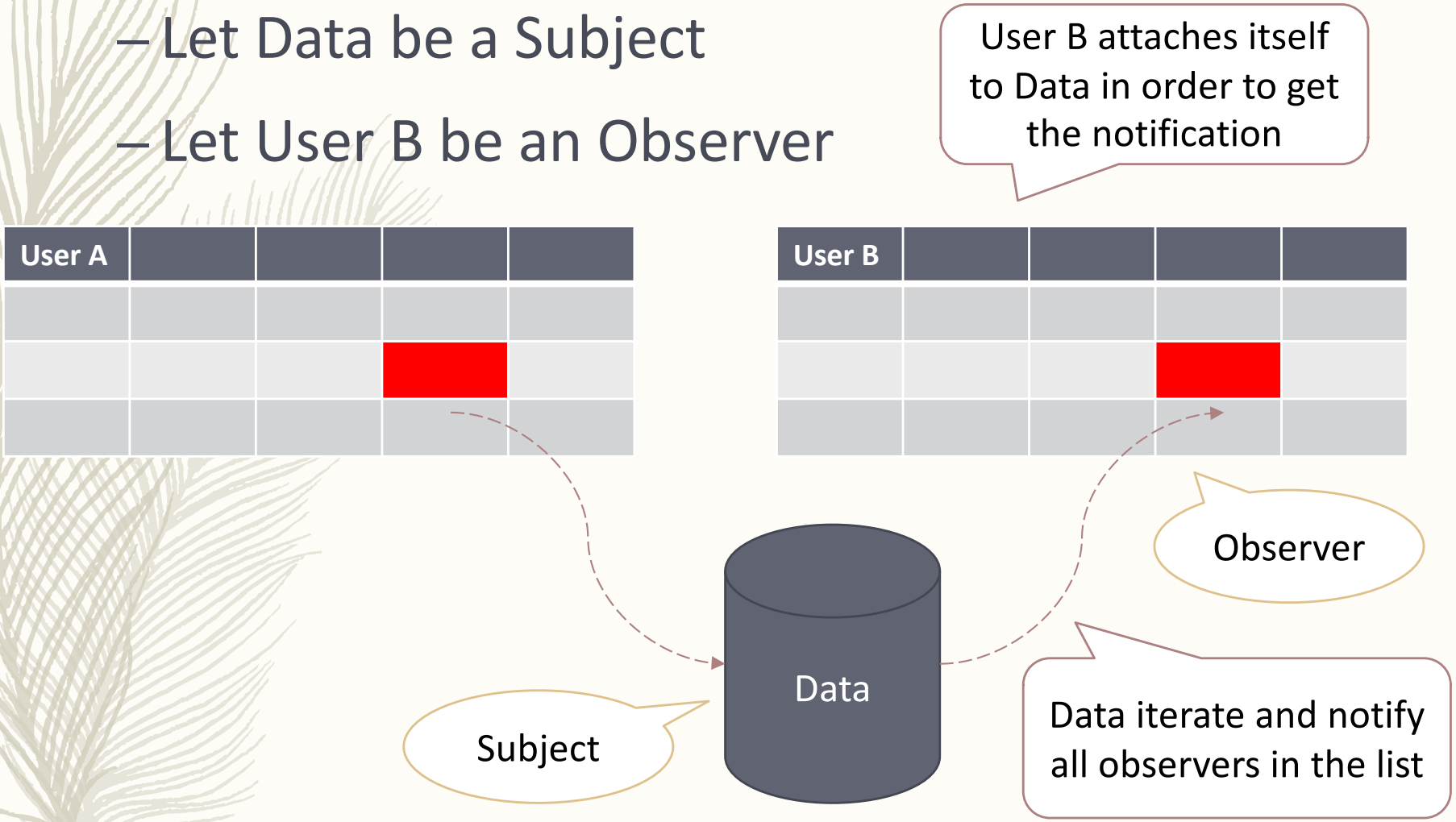


The Observer pattern (cont.)

- Describe the Observer pattern with classes
  - put the notification code in a different place
  - rather than hard-code the notification
- Easy to reuse the notification
- Easy to apply/remove the notification

Then we can...

- Let Data be a Subject
- Let User B be an Observer



- One of the most widely-used pattern in GoF design patterns

GoF design patterns

- The "GoF" design patterns book (1994)
  - The Gang of Four
- Help programmers to design and build better systems
  - Use with object-oriented programming (OOP)

→ Demonstrate how powerful OOP is??

Reference

- *E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Addison-Wesley, 1994.*

GoF design patterns (cont.)

- NO! it neither explains what OOP is nor shows how powerful OOP is
  - Indeed OOP is powerful
  - But not sufficient to resolve some design problems
  - How to overcome the things that cannot be resolved by using only dedicated OOP constructs
  - Such as class, method, interface, etc.

Reference

- *E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Addison-Wesley, 1994.*

GoF design patterns (cont.)

- Classify 23 patterns into three categories
  - Creational patterns
  - Structural patterns
    - *Decorator: we are going to see*
  - Behavioral patterns
    - *Observer: we already saw it*
    - *Visitor: we will see later*

Reference

- *E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Addison-Wesley, 1994.*

Another example

- We will see another example: the Decorator pattern
  - It is also frequently used
- Intent
  - attach additional responsibilities to an object dynamically
  - provide a flexible alternative to subclassing for extending functionality

Draw a Window

- Suppose now we are drawing windows on the screen by our selves
- We have a Window class/object, which
  - can draw its contents
  - to show some text and pictures in it

```
    :
w = new Window();
w.draw();
    :
```

Draw a Window (cont.)

– and give it to a client (e.g. “s” here)

– client will call its draw later

– w = new Window(...);

:

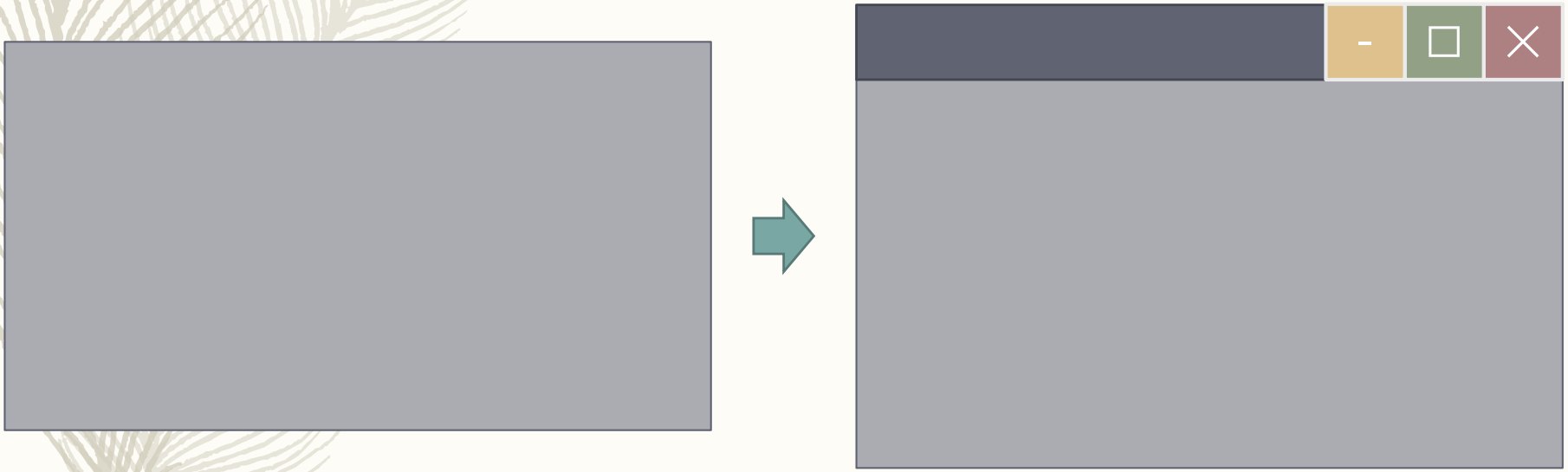
s.paint(w);

1. must create a Window explicitly
2. client must know Window

call w.draw() inside the paint method

How to add border for Window?

- Draw a border
  - allow to minimize, maximize, and close
  - allow to move and resize



Extending functionality

- Define a new kind of Window? (subclassing)
  - Then create a BorderWindow?
  - w = new BorderWindow(...);
    :
    s.paint(w);

now we must create a BorderWindow instead of Window

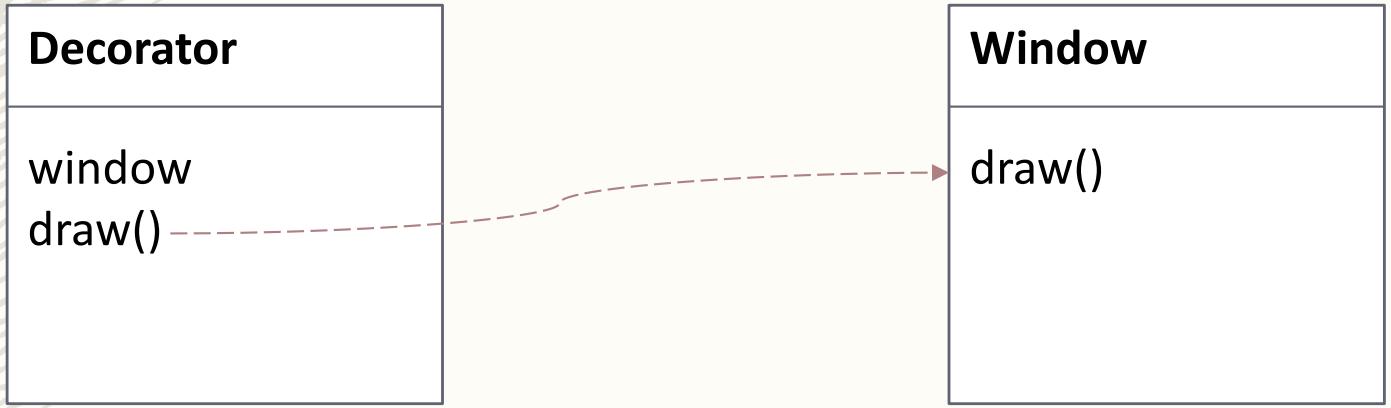
it's okay for clients since BorderWindow is also a Window

However...

- In some cases we want to avoid complicated subclassing
  - need an alternative
- want to add/remove the functionality dynamically
  - at runtime
  - apply it for individual objects

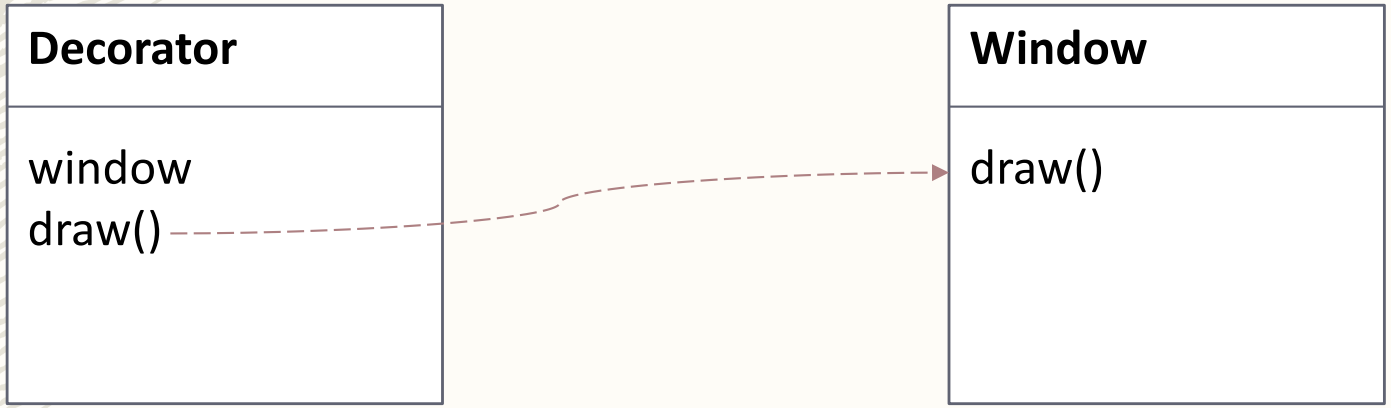
The Decorator pattern

- Define a Decorator
  - also a “kind” of Window (also has draw)
  - contains a Window object (window)
  - pass all calls to draw to its window



The Decorator pattern (cont.)

- Give the “decorated Window” to clients
  - clients think it is a Window
  - Windows can be decorated individually
  - and no need to re-create Window for removing border



Yet another example

- the Visitor pattern
  - It is widely used inside compilers
- The example will also bring up the next topic
  - Separation of concerns

Why we need AST in compilers?

- Why don't we do semantic analysis within parser?

- i.e. single phrase that does every thing

Reference

- *Andrew W. Appel. Modern compiler implementation in Java, 2nd edition, Cambridge. (the Tiger book)*

Recall: frontend of compilers

- Lexical analysis: recognize tokens in source code
- Syntactic analysis: use tokens to build a tree
- Semantic analysis: attach meaning to nodes in the tree

Reference

- *Andrew W. Appel. Modern compiler implementation in Java, 2nd edition, Cambridge. (the Tiger book)*

Why we need AST in compilers? (cont.)

- Why don't we do semantic analysis within parser?
  - i.e. single phrase that does every thing

- Separate issues of syntax from issues of semantics
  - Syntax: parsing
  - Semantics: type-checking

Reference

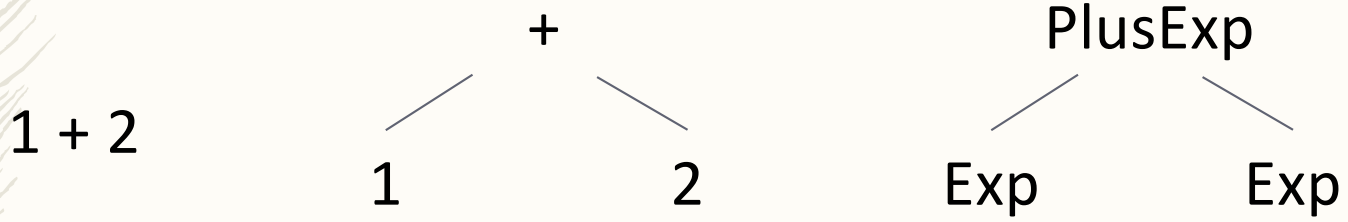
- *Andrew W. Appel. Modern compiler implementation in Java, 2nd edition, Cambridge. (the Tiger book)*

Why we need AST in compilers? (cont.)

- Let parser produce concrete syntax tree
- But it still depends too much on the grammar

  → Abstract Syntax Tree

- How to represent and manipulate AST as data structures?



Reference

- *Andrew W. Appel. Modern compiler implementation in Java, 2nd edition, Cambridge. (the Tiger book)*

Recall: Use classes to implement AST

```
public abstract class Exp {
  public abstract int eval();
}

public class PlusExp extends Exp {
  private Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() + e2.eval();
  }
}

public class MinusExp extends Exp {
  private Exp e1, e2;
  public MinusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() - e2.eval();
  }
}
```

- Here the inheritance in OOP is used
  - Both PlusExp and MinusExp inherit eval() from Exp
- Use the class Exp to represent expressions
  - PlusExp and MinusExp are kinds of Exp

Recall: Use classes to implement AST (cont.)

```
public abstract class Exp {
  public abstract int eval();
}

public class PlusExp extends Exp {
  private Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() + e2.eval();
  }
}

public class MinusExp extends Exp {
  private Exp e1, e2;
  public MinusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() - e2.eval();
  }
}
```

- We can write an interpreter by building AST and then interpret it
  - Every class of AST nodes contains “eval” method
  - Return the value of the represented expression

Recall: Use classes to implement AST (cont.)

```
public abstract class Exp {
  public abstract int eval();
}

public class PlusExp extends Exp {
  private Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() + e2.eval();
  }
}

public class MinusExp extends Exp {
  private Exp e1, e2;
  public MinusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() - e2.eval();
  }
}
```

- Looks good, but if we want to add another interpretation?
  - E.g. type check, optimization, etc.

Use classes to implement AST (cont.)

```
public abstract class Exp {
  public abstract int eval();
}

public class PlusExp extends Exp {
  private Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() + e2.eval();
  }
}

public class MinusExp extends Exp {
  private Exp e1, e2;
  public MinusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int eval() {
    return e1.eval() - e2.eval();
  }
}
```

- Add a new method like eval() for all classes!
- How to separate eval() from AST?
  - Separation of concerns

The Visitor pattern

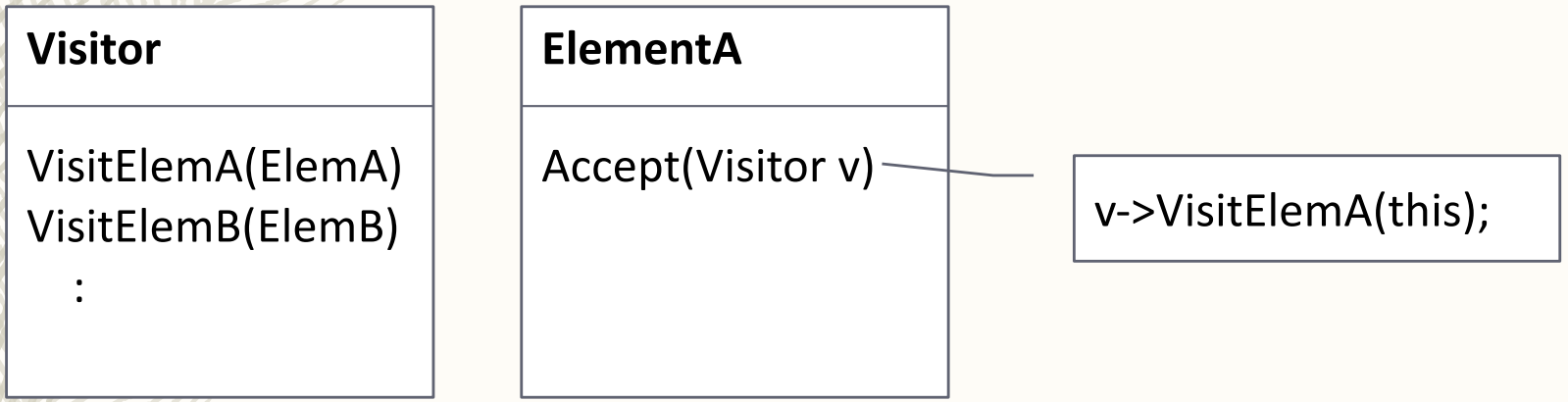
- There are three roles
  - elements are the set of objects (e.g. expressions)
  - visitor knows how to do operation (e.g. evaluation) for every element
  - client just want to perform the operation for elements

The Visitor pattern (cont.)

- Now operations for elements can be put together!
  - different operations can be defined in different visitors
- How it does?

The Visitor pattern (cont.)

- There are three roles
  - Every element can accept a visitor to do a specific operation (e.g. eval)
  - Every visitor define the operations for every element (e.g. PlusExp and MinusExp)
  - The client can choose a specific visitor to visit all elements



- The figure is simplified to avoid mentioning subtypes too much

Use the Visitor pattern to rewrite

```
public abstract class Exp {
  public abstract int accept(Visitor v);
}
public class PlusExp extends Exp {
  public Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int accept(Visitor v) {
    return v.visit(this);
  }
}

public interface Visitor {
  public int visit(PlusExp n);
  public int visit(MinusExp n);
}
public class Interpreter implements Visitor {
  public int visit(PlusExp n) {
    return n.e1.accept(this) + n.e2.accept(this);
  }
  public int visit(MinusExp n) {
    return n.e1.accept(this) - n.e2.accept(this);
  }
```

- All Exp classes can accept a Visitor
  - to do a certain operation
- The evaluation for all Exp classes are gathered in Interpreter

Use the Visitor pattern to rewrite (cont.)

```
public abstract class Exp {
  public abstract int accept(Visitor v);
}
public class PlusExp extends Exp {
  public Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int accept(Visitor v) {
    return v.visit(this);
  }
}

public interface Visitor {
  public int visit(PlusExp n);
  public int visit(MinusExp n);
}
public class Interpreter implements Visitor {
  public int visit(PlusExp n) {
    return n.e1.accept(this) + n.e2.accept(this);
  }
  public int visit(MinusExp n) {
    return n.e1.accept(this) - n.e2.accept(this);
  }
```

- Now the syntax is separated from interpretations
  - We can implement another Visitor for different operation
  - Without modifying Exp classes

Use the Visitor pattern to rewrite (cont.)

```
public abstract class Exp {
  public abstract int accept(Visitor v);
}
public class PlusExp extends Exp {
  public Exp e1, e2;
  public PlusExp(Exp a1, Exp a2) {
    e1 = a1; e2 = a2;
  }
  public int accept(Visitor v) {
    return v.visit(this);
  }
}

public interface Visitor {
  public int visit(PlusExp n);
  public int visit(MinusExp n);
}
public class Interpreter implements Visitor {
  public int visit(PlusExp n) {
    return n.e1.accept(this) + n.e2.accept(this);
  }
  public int visit(MinusExp n) {
    return n.e1.accept(this) - n.e2.accept(this);
  }
```

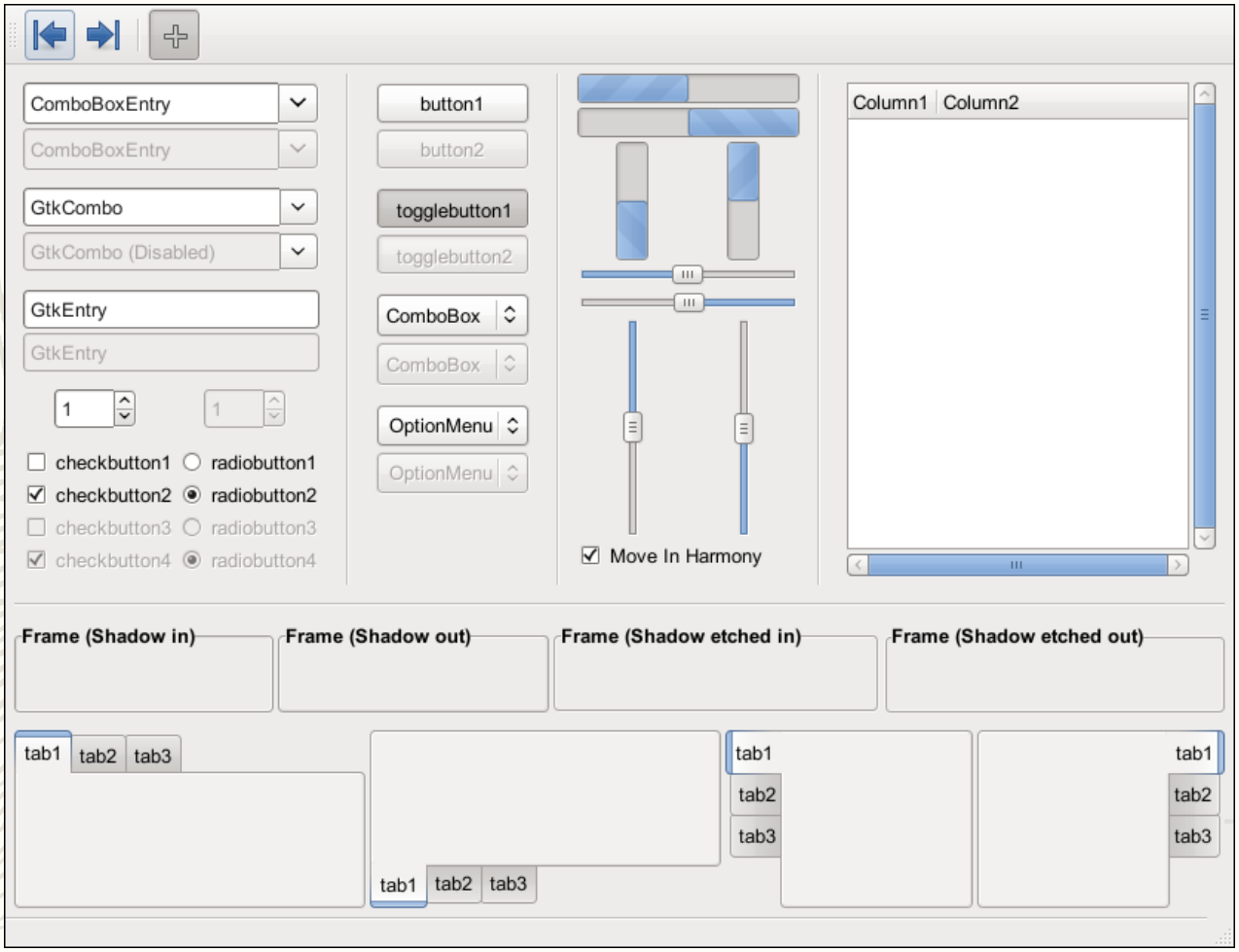
- Looks good, but if we need to add a new expression?
  - E.g. TimesExp, DivideExp, etc.
- Add a new visit method() for all visitors?!
  - How should we do?

OO style or Visitor style?

- OO style
  - Easy to add expressions
  - Hard to add interpretations
- Visitor style
  - Easy to add interpretations
  - Hard to add expressions
- How should we choose between?
  - In the case of compilers, the number of expressions is fixed → Visitor style

How about GUI widgets toolkit?

– For example, GTK+



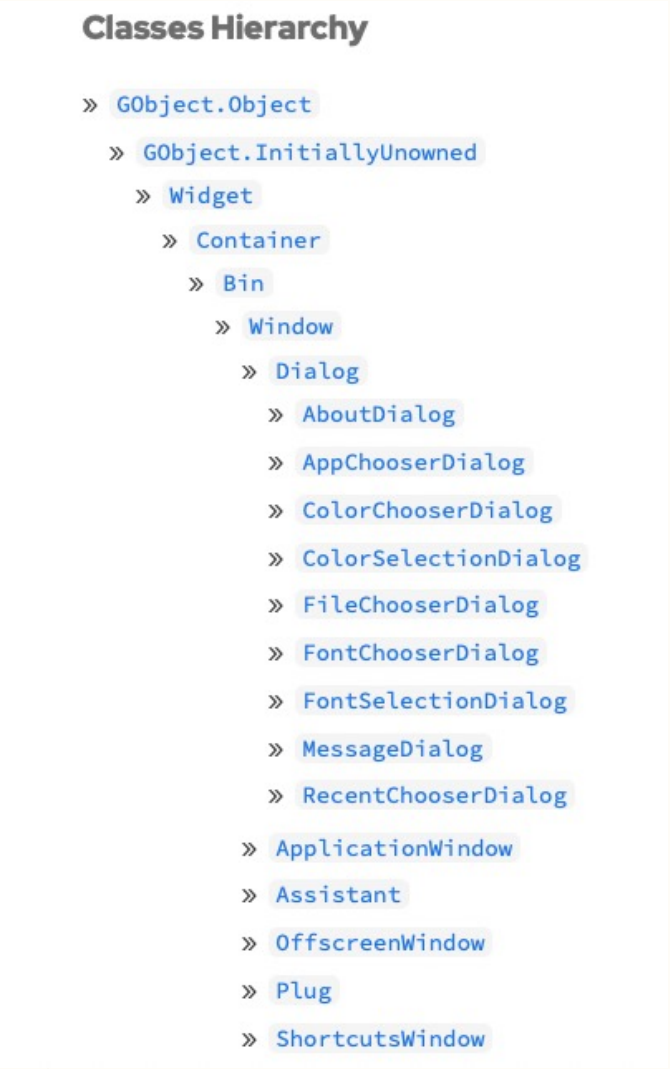
Reference

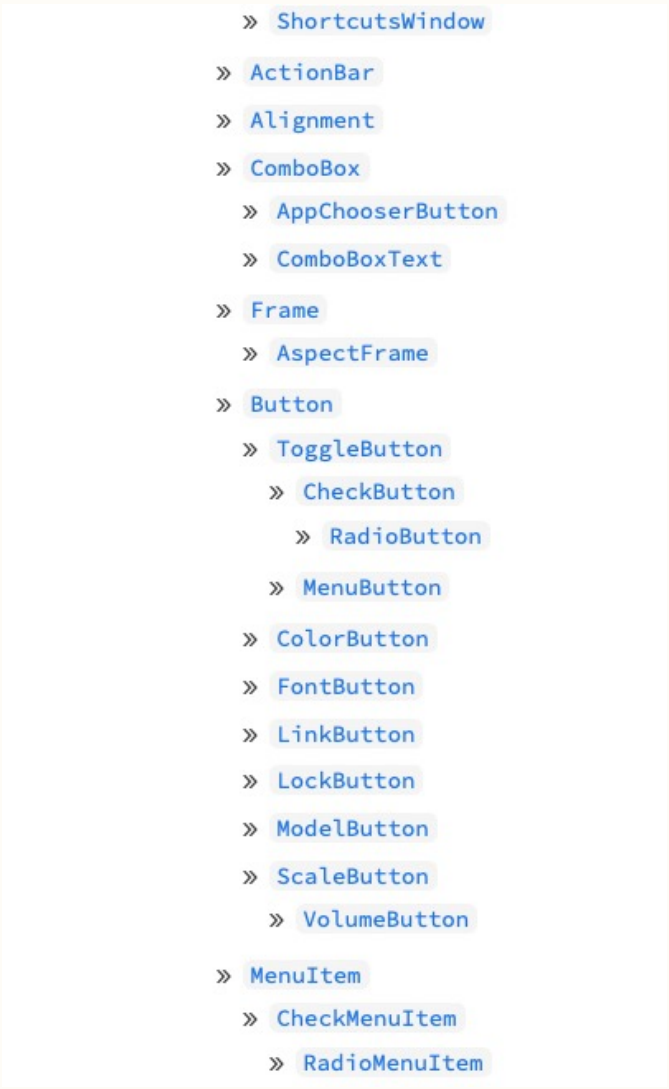
- https://docs.gtk.org/gtk4/visual_index.html

OO style is chosen for GUI widgets

- Since the number of interpretations (operations) is fixed
  - Draw and show
- But the number of expressions (widgets) will increase!
  - Library developers will create more and more widgets
  - Library users may extend them to create their own widgets

Recall: object hierarchy in GTK+





Reference

- https://docs.gtk.org/gtk3/classes_hierarchy.html

:

Separation of concerns

- "Focus one's attention upon some aspect"
  - "Does not mean ignoring the other aspects"
  - "It is just doing justice to the fact that from this aspect's point of view, the other is irrelevant"

--- Edsger W. Dijkstra

Reference

- *EWD 447: On the role of scientific thought, 1974.*
- *Edsger W. Dijkstra, Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982.*

By the way…something you must know

- When your are writing a paper/article…
  - Use the paragraphs you have published also needs citation
    - *Dijkstra explicitly quotes the paragraphs in his letter that he wrote to his friend*
  - Double submission is prohibited
    - *Even you are waiting for the notification*

Separation of concerns (cont.)

- Simplify the programs
  - Easier development
  - Better maintainability
  - Better reusability
- Similar in the real world?
  - Separation of duties
  - Separation of church and state??
    "Render unto Caesar the things that are Caesar's, and unto God the things that are God's"*

**from the Bible (Matt. 22:15-22; Mark 12:13-17; Luke 20:20-26)*

Some examples of SoC

- Model-View-Controller (MVC)
  - Model manages data, logic, and rules
  - View is responsible for the representations of information
  - Controller handles the input from user and give to Model
- HTML/CSS/JavaScript
  - HTML code contain the contents of web pages
  - CSS code defines the style
  - JavaScript code are used to interact with users

→ Easy to modify or replace one of them

Structured and modularized

- Structured programming
  - Use subroutine, block, loop, etc.
  - Instead of simple test and jump
- Modularize your code
  - Put the code for a functionality in one module
  - Avoid mixing the code for different things together

Reference

- *O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare. Structured Programming, 1972.*

An ideal program should be…

- Structured
  - Reuse code and make it readable
- Separation of concerns
  - Separate the code for different aspects
- Code modularity
  - One module for one functionality

but... how to?

- benefit from language constructs
  - use those **directly** given by the language
- benefit from design patterns
  - use those **indirectly** given by the language

→ Need the support of programming language!

Programming paradigms

"Unless you can support the paradigms I use when I program, or at least support my extending your language into one that does support my programming methods, I don't need your shiny new languages."

---R. W. Floyd

Reference

- *Robert W. Floyd. The Paradigms of Programming, Communications of the ACM, Aug. 1979.*

Programming paradigms?

- How we organize our code
  - to make the code structured
  - to separate the concerns in the code
  - to modularize the code
- Techniques that have been developed by smart guys
  - A good approach that you can refer and use

but... how to? (again)

- benefit from language constructs
  - use those **directly** given by the language

> language constructs such as class, method, function, lambda, etc.

- benefit from design patterns
  - use those **indirectly** given by the language

> use given language constructs to write a piece of code, i.e. design patterns

→ Need the support of programming language!

Who is he?

- Robert W. Floyd, Professor at Stanford University
  - 1978 ACM Turing Award
  - Helping to found the following important subfields of computer science
    - The theory of parsing
    - The semantics of programming languages
    - Automatic program verification
    - Automatic program synthesis
    - Analysis of algorithms

Reference

- *Robert W. Floyd. The Paradigms of Programming, Communications of the ACM, Aug. 1979.*

1978 ACM Turing Award Lecture

- Humorous sentences and convincing arguments
  - Nowadays it is still very convincing
  - Or, I should say, it is even more convincing
- I recommend you to read it by yourself
  - Here I pick up some paragraphs to share with you
- The lecture is revised and published on Communications of the ACM, August 1979, Volume 22 Number 8.

Reference

- *Robert W. Floyd. The Paradigms of Programming, Communications of the ACM, Aug. 1979.*

State of the art of programming reflects…

"I believe that the current state of the art of computer programming reflects

inadequacies in our stock of paradigms,

in our knowledge of existing paradigms,

in the way we teach programming paradigms,

and in the way our programming languages support,

or fail to support, the paradigms of their user communities."

Reference

- *Robert W. Floyd. The Paradigms of Programming, Communications of the ACM, Aug. 1979.*

Typically discourage others

"In computer science, one sees several such communities, each speaking its own language and using its own paradigms.

In fact, programming languages typically encourage use of some paradigms and discourage others."

Reference

- *Robert W. Floyd. The Paradigms of Programming, Communications of the ACM, Aug. 1979.*

Individual programmer requires

"If the advancement of the general art of programming requires the continuing invention and elaboration of paradigms,

advancement of the art of the individual programmer requires that he expand his repertory of paradigms."

Reference

- *Robert W. Floyd. The Paradigms of Programming, Communications of the ACM, Aug. 1979.*

In the words written on the wall…

In the words written on the wall of a Stanford University graduate student office,

"I would rather write programs to help me write programs than write programs."

Reference

- *Robert W. Floyd. The Paradigms of Programming, Communications of the ACM, Aug. 1979.*

Support and weakly support

"When a language makes a paradigm convenient, I will say the language supports the paradigm.

When a language makes a paradigm feasible, but not convenient, I will say the language weakly supports the paradigm."

Reference

- *Robert W. Floyd. The Paradigms of Programming, Communications of the ACM, Aug. 1979.*

Support the major paradigms

"I believe that the continued advance of programming as a craft requires development and dissemination of languages which support the major paradigms of their user's communities.

The design of a language should be preceded by enumeration of those paradigms, including a study of the deficiencies in programming caused by discouragement of unsupported paradigms."

Reference

- *Robert W. Floyd. The Paradigms of Programming, Communications of the ACM, Aug. 1979.*

To the serious programmer

"spend a part of your working day examining and refining your own methods.

Even though programmers are always struggling to meet some future or past dead-line, methodological abstraction is a wise long term investment."

Reference

- *Robert W. Floyd. The Paradigms of Programming, Communications of the ACM, Aug. 1979.*

To the designer of programming languages

"unless you can support the paradigms I use when I program, or at least support my extending your language into one that does support my programming methods, I don't need your shiny new languages;

like an old car or house, the old language has limitations I have learned to live with.
To persuade me of the merit of your language, you must show me how to construct programs in it."

Reference

- *Robert W. Floyd. The Paradigms of Programming, Communications of the ACM, Aug. 1979.*

Next lecture

10/21 **Practice 3**

10/28 no lecture (midterm exam week)

11/4 Functional Programming