“To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. ...

- Dijkstra, *A discipline of programming*, 1976 last chapter, *In retrospect*
… The other aspects have to wait their turn, because our heads are so small that we cannot deal with them simultaneously without getting confused. This is what I mean by ‘focussing one's attention upon a certain aspect’; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant for the current topic. ...
… Such separation, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of. ...

- Dijkstra, *A discipline of programming*, 1976
  last chapter, *In retrospect*
… I usually refer to it as ‘separation of concerns’, because one tries to deal with the difficulties, the obligations, the desires, and the constraints one by one. ...

- Dijkstra, *A discipline of programming*, 1976
  last chapter, *In retrospect*
… When this can be achieved successfully, we have more or less partitioned the reasoning that had to be done — and this partitioning may find its reflection in the resulting partitioning of the program into ‘modules’ — but I would like to point out that this partitioning of the reasoning to be done is only the result, and not the purpose. ...

- Dijkstra, *A discipline of programming*, 1976

  last chapter, *In retrospect*
… The purpose of thinking is to reduce the detailed reasoning needed to a doable amount, and a separation of concerns is the way we hope to achieve this reduction. ...

- Dijkstra, *A discipline of programming*, 1976
  last chapter, *In retrospect*
... The crucial choice is, of course, what aspects to study ‘in isolation’, how to disentangle the original amorphous knot of obligations, constraints and goals into a set of ‘concerns’ that admit a reasonably effective separation. ...

- Dijkstra, *A discipline of programming*, 1976
  last chapter, *In retrospect*
... To arrive at a successful separation of concerns for a new, difficult problem area will nearly always take a long time of hard work; it seems unrealistic to expect otherwise. ...

- Dijkstra, *A discipline of programming*, 1976
  last chapter, *In retrospect*
… The knowledge of the goal of ‘separation of concerns’ is a useful one: we are at least beginning to understand what we are aiming at.”

- Dijkstra, *A discipline of programming*, 1976
  last chapter, *In retrospect*
goal of this talk

- discuss the implementation of complex software systems
- focusing on issues of modularity
- how existing tools help achieve it
- propose a new tool to help improve modularity in some cases where existing tools are inadequate

slides, papers and system at www.parc.xerox.com/aop
format of this talk

• sharing context
• a problem and an idea
• our current instantiation of the idea
• implementation
• summary and hopes
sharing context
the engineering challenge

• extremely complex systems
• more than our mind can handle all at once
• must manage the complexity
problem decomposition

- break the problem into sub-problems
- address those relatively independently
solution construction & composition

- construct complete systems from the designs by
  - implementing the sub-parts, and
  - composing them to get the whole
design & implementation

- decomposition breaks big problems into smaller ones
- composition builds big solutions out of smaller ones
“clean separation of concerns”

we want:

– natural decomposition
– concerns to be localized
– explicit handling of design decisions
– in both design and implementation
achieving this requires...

- synergy among
  - problem structure and
  - design concepts and
  - language mechanisms

“natural design”

“the program looks like the design”
the “component”\textsuperscript{1} concept

- a modular unit of functionality
- fits many natural design concerns
- well-supported by existing programming technology
- a rich collection of
  - design principles, conventions and notations
  - programming mechanisms
object-orientation

“objects”
- used in design and implementation
- object-oriented design
- object-oriented programming
- many tools to bridge the gap

part i -- sharing context
procedural programming

“procedures”

– used in design and implementation
– some tools even bridge the gap
summary so far

good separation of concerns in both design and implementation

complex systems

→

design practices support decomposition into components

programming languages support implementation and composition of components
II

a problem and an idea
a distributed digital library
the component structure

• use objects
  – objects are a natural fit for this system

• so…
  – the design breaks down into component objects
  – implement using OOP
the class graph

```
<table>
<thead>
<tr>
<th>Book</th>
<th>holds</th>
<th>Library</th>
<th>cooperates</th>
</tr>
</thead>
<tbody>
<tr>
<td>title: string</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>author: string</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isbn: int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pdf: pdf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tiff: tiff</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Library</th>
<th></th>
<th>User</th>
<th>uses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Printer</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>User</th>
<th></th>
<th>Printer</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

part ii -- a problem and an idea  
Copyright 1997, 1998 Xerox Corporation
The code

Book

class Book {
    private String title;
    private String author;
    private String isbn;
    private PostScript ps;
    private User borrower;

    public Book(String t, String a, String i, PostScript p) {
        title = t;
        author = a;
        isbn = i;
        ps = p;
    }

    public User get_borrower() { return borrower; }
    public void set_borrower(User u) { borrower = u; }
    public PostScript get_ps() { return ps; }
}

User

class User {
    private String name;
    Library theLibrary;
    Printer thePrinter;

    public User(String n) { name = n; }

    public boolean getBook(String title) {
        Book aBook = theLibrary.getBook(this, title);
        thePrinter.print(this, aBook);
        return true;
    }
}

Library

class Library {
    Hashtable books;
    Library() {
        books = new Hashtable(100);
    }

    public Book getBook(User u, String title) {
        System.out.println("REQUEST TO GET BOOK " + title);
        if (books.containsKey(title)) {
            Book b = (Book)books.get(title);
            System.out.println("getBook: Found it:" + b);
            if (b != null) {
                if (b.get_borrower() == null) {
                    b.set_borrower(u);
                }
            }
        }
        return null;
    }
}

Printer

class Printer {
    String status = "Idle";
    Vector jobs;

    public boolean print(User u, Book b) {
        PostScript ps = get_ps(b);
        Job newJob = new Job(ps, u.get_name());
        return queue(newJob);
    }

    boolean queue(Job j) {
        //...
        return true;
    }
}

part ii -- a problem and an idea
all is well

• design is natural
• code looks like the design
• good separation of concerns
  – localized in the design
  – localized in the code
  – handled explicitly
a distributed digital library
minimizing network load

dataflow patterns
minimizing network load
controlling slot copying
class Book {
    private BookID id;
    private PostScript ps;
    private User user;

    public Book(String t, String a, String i, PostScript p) {
        id = new BookID(t, a, i);
        ps = p;
    }

    public User getUser() { return user; }
    public BookID get_bid() { return id; }
}

class BookID {
    private String title;
    private String author;
    private String isbn;

    public BookID(String t, String a, String i) {
        title = t;
        author = a;
        isbn = i;
    }

    public String get_title() { return title; }
}

class User {
    private UserID id;
    private String name;
    private Library library;
    private Printer printer;

    public User(String n) { id = new UserID(n); }

    public boolean getBook(String title) {
        BookID aBook = null;
        try {
            aBook = library.getBook(id, title);
            catch (RemoteException e) {}
            printer.print(id, aBook);
        } catch (RemoteException e) {} return true;
    }

    public BookID get_uid() { return id; }
}

class UserID {
    private String name;

    public UserID(String n) { name = n; }
}

interface LibraryInterface extends Remote {
    public BookID getBook(UserID u, String title) throws RemoteException;
    public PostScript getBookPS(BookID bid) throws RemoteException;
}

class Library extends UnicastRemoteObject implements LibraryInterface {
    private Hashtable books;
    public Library() throws RemoteException {
        books = new Hashtable(100);
    }

    public BookID getBook(UserID u, String title) throws RemoteException {
        System.out.println("REQUEST TO GET BOOK " + title);
        if (books.containsKey(title)) {
            Book b = (Book)books.get(title);
            if (b != null) {
                if (b.get_borrower() == null) {
                    b.set_borrower(u);
                    return b.get_bid();
                }
            }
        } return null;
    }

    public PostScript getBookPS(BookID bid) throws RemoteException {
        if (books.containsKey(bid.get_title())) {
            Book b = (Book)books.get(bid.get_title);
            if (b != null) {
                b.set_borrower(u);
                return b.get_bid();
            }
        } return null;
    }
}

interface PrinterInterface extends Remote {
    public boolean print (UserID u, BookID b) throws RemoteException;
}

class Printer extends UnicastRemoteObject implements PrinterInterface {
    private Library theLibrary;

    public Printer() throws RemoteException {
        String status = "Idle";
        Vector jobs;
        theLibrary = library;
    }

    public boolean print (User u, BookID b) throws RemoteException {
        PostScript ps = null;
        try {
            ps = theLibrary.getBookPS(b);
            catch (RemoteException e) {}}
        Job newJob = new Job (ps, u.get_name());
        queue(newJob);
        return true;
    }
}

class Job {
    private String name;
    private PostScript ps;

    public Job(String n, PostScript p) {
        name = n;
        ps = p;
    }
}

interface Remote {
    public Book getBook(String title) throws RemoteException;
    public PostScript getBookPS(String bid) throws RemoteException;
}

class RemoteImpl {
    public Book getBook(String title) throws RemoteException {
        //... return new Book();
    }

    public PostScript getBookPS(String bid) throws RemoteException {
        //... return new PostScript();
    }
}
why?

• why did so much code change?
• why wasn’t this concern well localized?
• why didn’t this “fit” the component structure?
because...

- we are working with “emergent entities”, and
- the component concept, and its associated implementation mechanisms, fundamentally don’t provide adequate support for working with emergent entities
emergent entities

library

library

book

printer

user
emergent entities

- emerge\(^1\) during program execution
  - from (possibly non-local) interactions of the components
- are not components
  - do not exist explicitly in the component model or code

\(^1\) emerge: to become manifest; to rise from or as if from an enveloping fluid; come out into view
emergent entities

- emerge\(^1\) during program execution
  - from (possibly non-local) interactions of the components
- are not components
  - do not exist explicitly in the component model or code

\(^1\) emerge: to become manifest; to rise from or as if from an enveloping fluid; come out into view
are tough to handle because...

- they are not explicit in the component model or code
- they have non-localized origins and interactions
- they cross-cut the component structure...
cross-cutting the components

- the sub-parts of the objects are not existing classes
- the desired dataflows are not existing message sends
but, but, but... the code can be remodularized to “fit” better...

<imagine your own alternative class diagram here>

violates separation of concerns leads to tangled code
claim

• remodularizing isn’t good enough!
  – it ruins the separation of concerns

• the functionality and the network optimization concern are fundamentally different

• would like different “carvings”\(^1\) of the system
  – in terms of component structure,
  – and in terms of emergent entities,
    • with support for the cross-cutting modularities

---

\(^1\) carve: to cut with care or precision, to cut into pieces or slices, to work as a sculptor or engraver
just try it

dataflow Book {Library to User}
{copy: title, author, isbn};

dataflow Book {Library to Printer}
{direct: pdf, tiff};
what it says

**dataflow** Book {Library to User}
{copy: title, author, isbn};

The dataflow of books, from library objects to user objects, should be implemented by copying the title, author and isbn slots only.
how it says it

dataflow Book {Library to User} {copy: title, author, isbn};

identifies emergent entity

controls its implementation

part ii -- a problem and an idea
cross-cutting

dataflow Book \{Library to User\}
\{copy: title, author, isbn\};
and...

**dataflow** Book \{Library to Printer\}
\{direct: pdf, tiff\};

part ii -- a problem and an idea
assume a...

• a “language processor” that
  – accepts two kinds of code as input;
  – produces “woven” output code, or
  – directly implements the computation
general claim

• remodularizing the component structure is not a satisfactory way of dealing with emergent entities

• want different carvings of the system:
  – keep the clean component structure
  – control emergent entities in “natural terms”
    • in terms of the emergent entity
    • with support for cross-cutting
emergent entities

• an entity that does not exist explicitly in the component model or code, but rather arises during execution
  – data flows
    • all the places this value goes...
  – control states
    • two methods running concurrently
    • one method blocked on another
    • all the callers of this function
    • history of calls up to this point (aka the stack)...

the “aspect” concept

• components are modular units of functionality

• aspects are modular units of control over emergent entities

• in the distributed digital library:
  – library component
  – book component
  – user component
  – printer component
  – lookup dataflow aspect
  – printing dataflow aspect
  – ...
  – ...
  – ...
“aspect languages”

• aspect languages connect to a component language, and provide:
  – a mechanism for referring to emergent entities
  – a mechanism for exercising some control over the implementation of the emergent entities
  – support for using cross-cutting modularities

\[\text{dataflow Book \{Library to Printer\}}\]
\[\{\text{direct: pdf, tiff}\};\]
summary so far

complex systems

\[ \downarrow \]

design practices support decomposition into components & aspects

\[ \downarrow \]

programming languages support implementation and composition of components & aspects

improved separation of concerns in both design and implementation
the AspectJ™ languages
AspectJ is...

- an extension to Java™
- targeted at distributed and/or concurrent applications
- several general-purpose aspect languages
  - remote data transfer aspect language
  - computation migration aspect language
  - coordination aspect language
- a weaver for those languages
a data transfer aspect language

- provides control over data transfers between execution spaces
  - transfer of arguments and/or return values
  - control over sub-fields, sub-sub-fields etc.

```
portal
```

```
execution space 1
```

```
execution space 2
```

```
o_t.m(o_p)
```

Copyright 1997, 1998 Xerox Corporation
referring to the emergent entity

```
portal Library { 
    Book find (String title){
        return:
            Book: {copy title, author, isbn;}
    }
}
```

```
execution space 1
```

```
execution space 2
```

Copyright 1997, 1998 Xerox Corporation
copy transfer mode

```java
portal Library {
    Book find (String title) {
        return:
            Book: {copy title, author, isbn;}
    }
}
```

execution space 1

execution space 2
gref transfer mode

```java
portal Library {
    Book find (String title) {
        return:
            Book: gref;
    }
}
```

**execution space 1**

**execution space 2**
direct transfer mode

```java
portal Printer {
    void print(Book book) {
        book: Book: {direct pages;}
    }
}
```

execution space 0
the aspect language cross-cuts OOP

when you send:

- **this kind of object**, as
- **this argument of**
- **this method**, send
- **this field**
- **this way**
aspect composition cross-cuts too

• these aspects compose along dataflows
• not along normal class/method composition

diagram:

```java
portal Library {
    Book find (String title) {
        return: Book: {copy title, author, isbn;}
    }
}

portal Printer {
    void print(Book book) {
        book: Book: {direct pages;}
    }
}
```
more on cross-cutting

```
portal Library {
    Book find(String title) {
        return: {copy title, author, isbn;
            Author bypass books;}
    }
}
```
what this is and isn’t

- weaver combines two kinds of code
- equivalent effect of complex tangled code
- equivalent elegance of original clean code
  – component code is unchanged
  – natural modularity of aspects
public class Shape {
    protected int x = 0;
    protected int y = 0;
    protected int w = 0;
    protected int h = 0;

    int getX() { return x; }
    int getY() { return y; }
    int getWidth() { return w; }
    int getHeight() { return h; }
    void adjustLocation() {
        x = longCalculation1();
        y = longCalculation2();
    }
    void adjustSize() {
        w = longCalculation3();
        h = longCalculation4();
    }
}
fits object-oriented modularity

**static coordinator** Shape {
  **selfex** adjustLocation,
  adjustSize;
  **mutex** {adjustLocation, getX};
  **mutex** {adjustLocation, getY};
  **mutex** {adjustSize, width};
  **mutex** {adjustSize, height};
}

• per-object
• per-class
cross-cuts object-oriented modularity

• per-object
• per-class
• multi-class
• any methods
status of AspectJ

• some preliminary user studies complete
  – results promising, not yet conclusive

• first public release to go on web-site shortly
  – free use (including in products)
  – weaver, documentation, example programs
  – coordination aspect language only

• next release early June
  – remote data transfer aspect language

• later releases
  – other aspect languages, operate directly on class files…
implementing aspect weavers
what aspect weavers do

- implement one or more aspect languages
- allow us to program in alternate modularity
  - in the modularity of the emergent entity
  - help with cross-cutting
- aspect weaver must “gather up the roots and contact points of emergent entities”
  - places spread around the OO program
  - this can appear difficult...
for example

```
coordinator Shape, Screen {
    mutex {adjustLocation, getX};
    mutex {adjustLocation, getY};
}
```

requires coordinated code in these places

```
public class Shape {
    protected int x = 0;
    protected int y = 0;
    protected int w = 0;
    protected int h = 0;

    int getX() { return x; }
    int getY() { return y; }
    int getWidth() { return w; }
    int getHeight() { return h; }
    void adjustLocation() {
        x = longCalculation1();
        y = longCalculation2();
    }
    ...
}
```
class Library {
    Hashtable books = new Hashtable(100);
}
public Book find(User u, String title) {
    frob();
    if (books.containsKey(title)) {
        frob();
        Book b = (Book) books.get(title);
        if (b != null) {
            frob();
            if (b.getBorrower() == null) {
                frob();
                b.setBorrower(u);
            }
            return b;
        }
    }
    return null;
}
domain transforms

- what is diffuse in one domain is local in another
- the Fourier transform moves between the two
  - it localizes what was non-local and vice-versa
reflection links two domains

• the object domain: localizes books and their functionality
• the meta domain: localizes “frob every method call”

```java
class Library {
    Hashtable books;
    Library() {
        books = new Hashtable(100);
    }
    public Book find(User u, String title) {
        if(books.containsKey(title)) {
            Book b = (Book)books.get(title);
            if (b != null) {
                if (b.get_borrower() == null)
                    b.set_borrower(u);
                return b;
            }
        }
        return null;
    }
}
```
aspect weavers

can require a variety of domain-transforms

• method calls (all, per-class, per-selector…), field accesses (…), methods (…);
• who else is running
• where will this value go next
• reflection
• unfolding
• CPS conversion
• partial evaluation
• abstract interpretation
conclusions
summary

- improved separation of concerns in both design and implementation
  - complex systems
  - design practices support decomposition into components & aspects
  - programming languages support implementation and composition of components & aspects

aspect-oriented design

aspect-oriented programming

www.parc.xerox.com/aop
an analogy
(what I hope aspects are like)

designing and building a simple bridge...
different kinds of picture

simple statics

more detailed statics

simple dynamics
a distributed digital library
different kinds of picture

- modeling of functionality
- modeling of control over emergent entities
different kinds of program

• programming with different carvings of the system
• allows clean separation of:
  – programming of functionality
  – programming of control over emergent entities
• AOP enables modular control over emergent entities
• using languages that support cross-cutting modularities
object & aspect programs

class Library {
    Hashtable books;
    Library() {
        books=new Hashtable(10);
    }
    ...
}

portal Printer {
    void print(Book book) {
        book: Book: {direct pages;}
    }
}

coordinator User, Library {
    mutex {checkOut, checkIn};
}

- AOP enables modular control over emergent entities
- using languages that support cross-cutting modularities