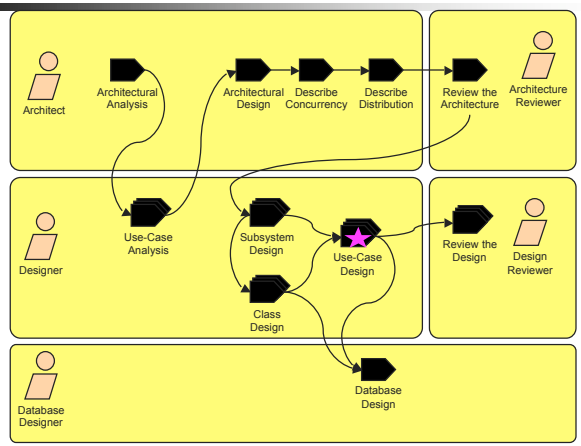
20 - Design & Implementation

- Readings:
- XP
  - Extreme Programming Explained, Kent Beck Addison Wesley 1999
  - Refactoring: Improving the Design of Existing Code, Martin Fowler, Addison Wesley 1999
  - http://www.extremeprogramming.org
  - http://www.xp2001.org
- AOP
  - Aspect-Oriented Programming with AspectJ™ Erik Hilsdale, Gregor Kiczales (with Bill Griswold, Jim Hugunin, Wes Isberg, Mik Kersten),

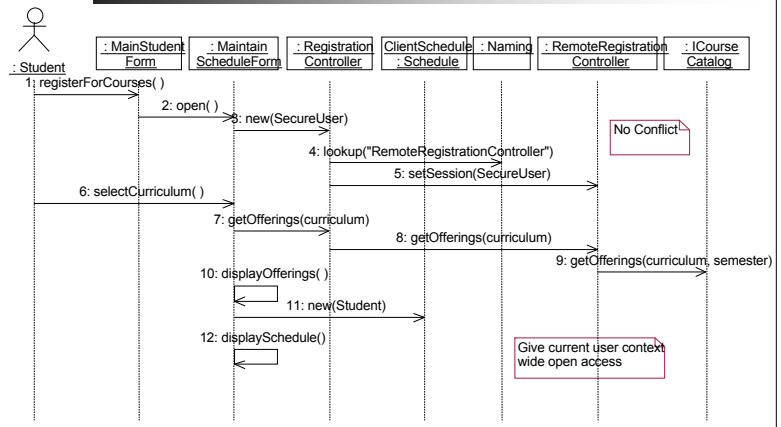
So Where Are We?



Strategy - where are we?

- made an initial attempt at defining the architecture
- defined the major elements of our system
  - the subsystems, their interfaces, the design classes, the processes and threads
  - relationships & how these elements map into the hardware on which the system will run.
- Now, concentrate on
  - making sure that there is consistency from beginning to end of use case implementation, i.e., that nothing has been missed (i.e., this is where we make sure that what we have done in the previous design activities is consistent with regards to the use case implementation).
- we do some Use Case Design before Subsystem Design
- Subsystem Design, Class Design and Use Case Design activities are tightly bound and tend to alternate between one another.

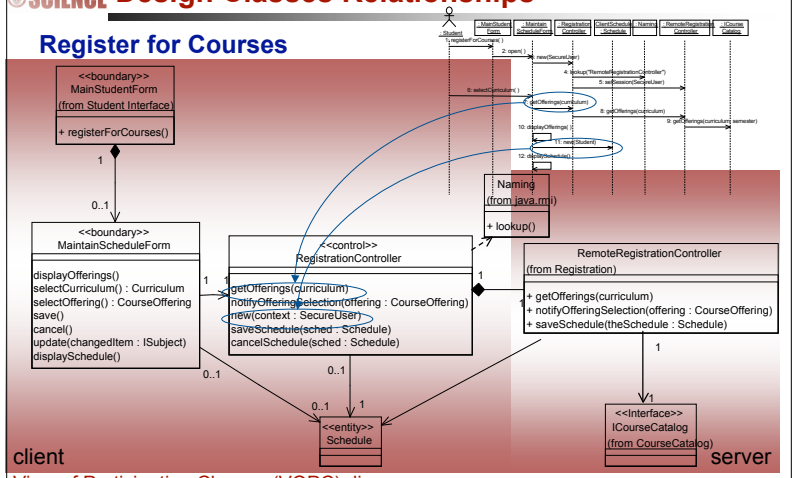
Design Element Interactions (Register For Courses - Set-Up)



COMPUTER SCIENCE

Design Classes Relationships

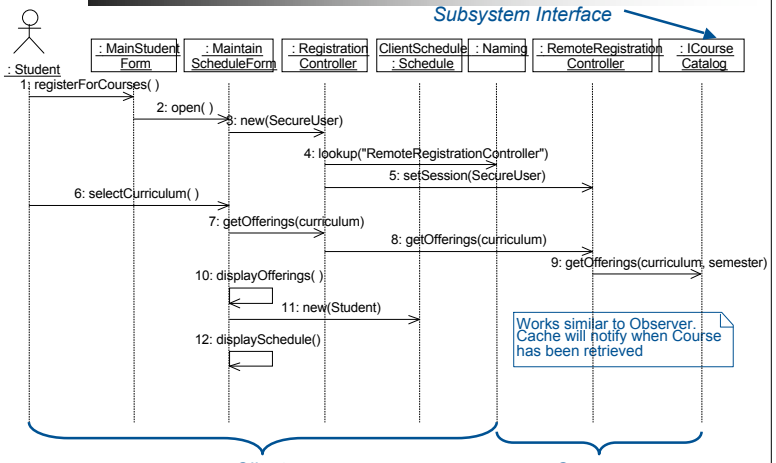
Register for Courses



View of Participating Classes (VOPC) diagram.

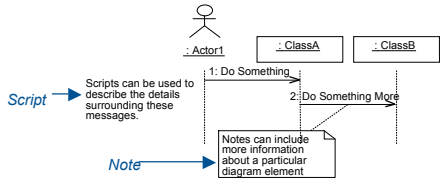
COMPUTER SCIENCE

Deployment

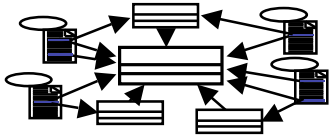


More steps

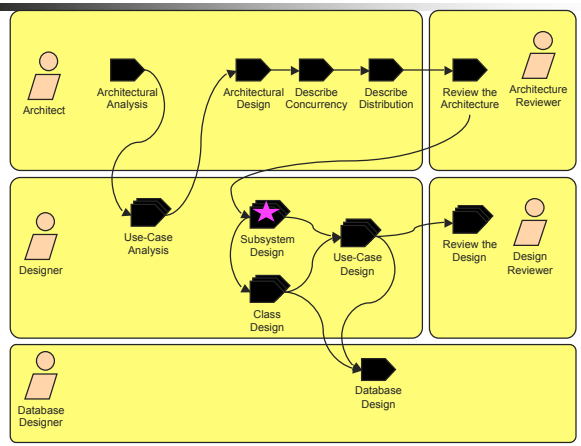
- Annotate the sequence diagrams



- Unify classes & subsystems
  - merge similar model elements
  - use inheritance to abstract model elements



So Where Are We?



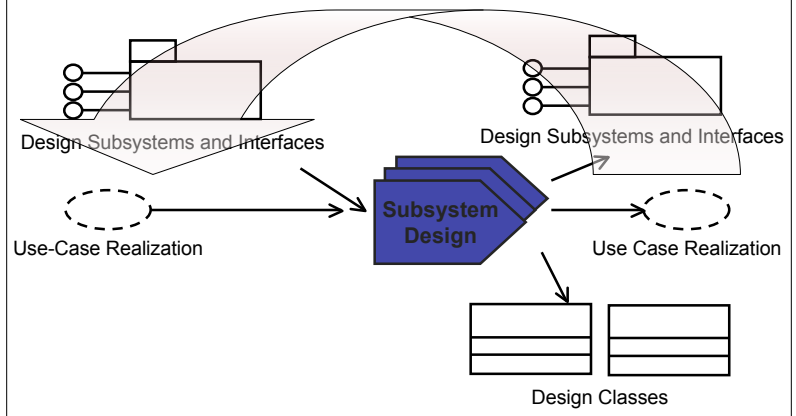
Strategy -- where are we?

- have defined subsystems, their interfaces, and their dependencies as “containers” of complex behavior that, for simplicity, we treat as a ‘black box’
- made an initial cut at some design classes, which have been allocated to subsystems
- need to flesh-out the details of the internal interactions
  - what classes exist in the subsystem to support?
  - how do they collaborate to support, the responsibilities documented in the subsystem interfaces?
  - In Subsystem Design, we look at the responsibilities of the subsystems in detail, defining and refining the classes that are needed to implement those responsibilities, refining subsystem dependencies, as needed. The internal interactions are expressed as collaborations of classes and possibly other components or subsystems

Strategy

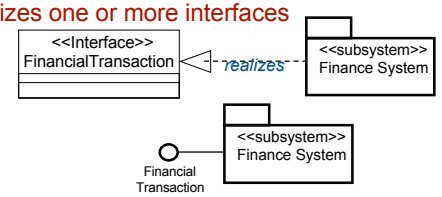
- need to do some Use Case Design before Subsystem Design
- after Analysis and Architectural Design
  - usually only have sketchy notions of responsibilities of classes and subsystems
  - details need to get worked out in Use Case Design, before one is really ready to design the classes and subsystems
- Reminder: there is frequent iteration between Use Case Design, Subsystem Design and Class Design.

Subsystem Design Overview



Interface and Subsystem

- What is an interface?
  - a model element which defines a set of behaviors (a set of operations) offered by a classifier model element (specifically, a class, subsystem or component)
- What is a subsystem?
  - Contains other model elements and has behavior
  - Realizes one or more interfaces

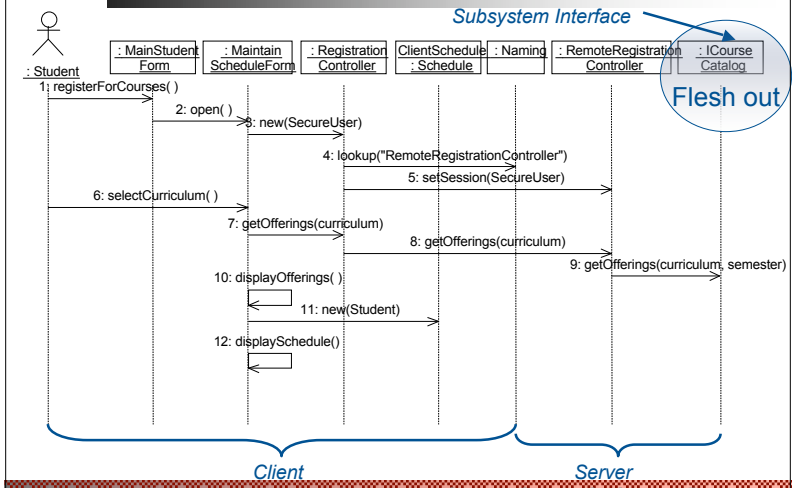


Recall: packages provide no behavior; they are simply containers of things which provide behavior

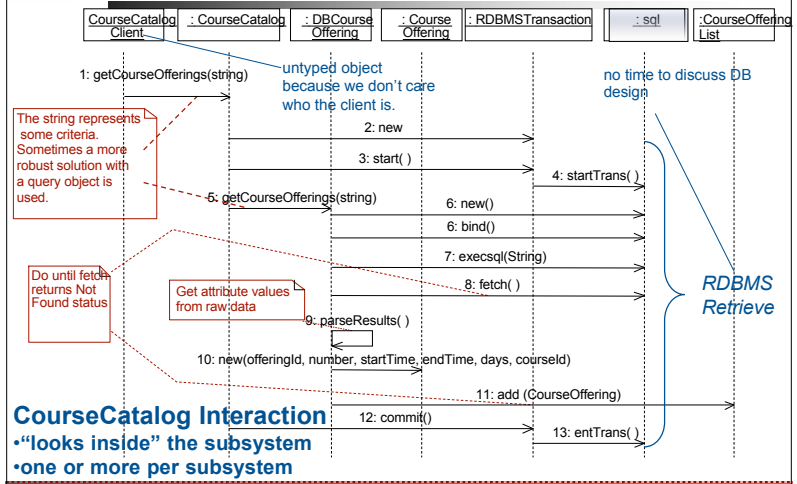
Distribute Subsystem Responsibilities

- Identify or reuse existing classes and/or subsystems
- Allocate subsystem responsibilities to classes and/or subsystems
- Incorporate the applicable mechanisms (e.g., persistence, distribution, etc.)
- Document collaborations with “interface realization” diagrams
  - 1 or more sequence diagrams per interface operation
- Revisit Architectural Design
  - Adjust subsystem boundaries and/or dependencies, as needed

Subsystem design



Local Subsystem Interaction

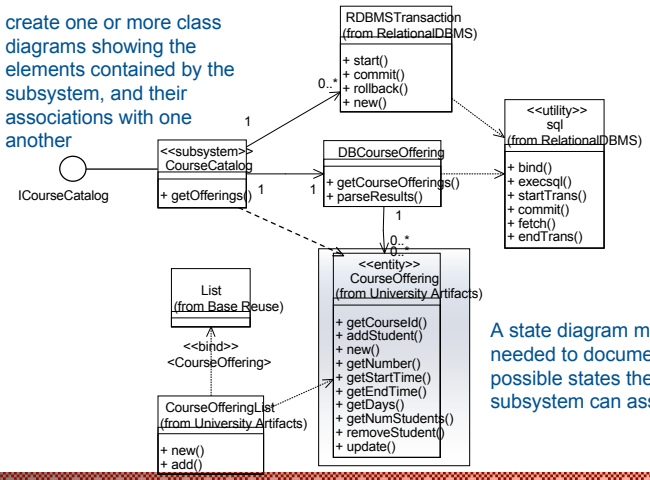


CourseCatalog Interaction

- "looks inside" the subsystem
- one or more per subsystem

Document Subsystem Elements

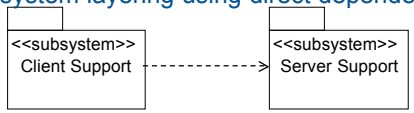
create one or more class diagrams showing the elements contained by the subsystem, and their associations with one another



A state diagram may be needed to document the possible states the subsystem can assume

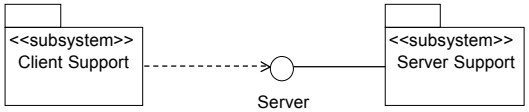
Describe Subsystem Dependencies

- Subsystem layering using direct dependency

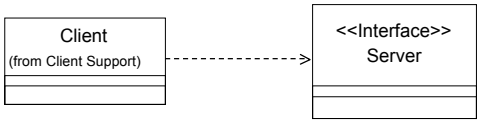


Not recommended

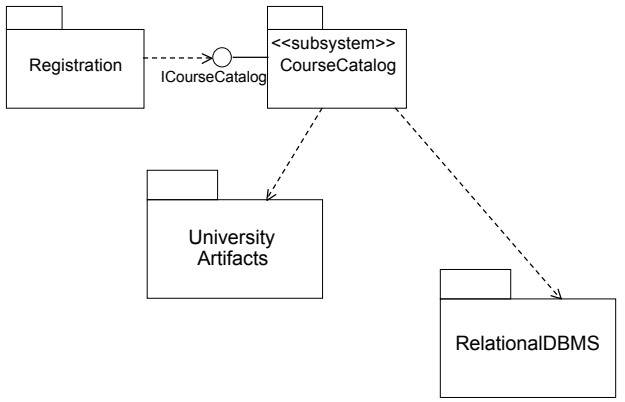
- Subsystem layering using interface dependency



More flexible



Describe Subsystem Dependencies



COMPUTER SCIENCE

Implementation

COMPUTER SCIENCE

XP Practices

- The Planning Game.
- Small Releases.
- Metaphor.
- Simple Design.
- Testing.
- Refactoring.
- Pair Programming.
- Collective Ownership.
- Continuous Integration.
- 40-Hours a Week.
- On-Site Customer.
- Coding Standards.

XP Practices

- The Planning Game
  - customers and developers cooperate to produce the maximum business value as rapidly as possible.
  - planning game happens at various scales, but the basic rules are pretty much the same:
    - customer comes up with a list of desired features for the system written out as a *User Story*, which gives the feature a name, and describes, broadly, what is required.
    - developer estimates how much effort each story will take, and how much effort the team can produce in a given time interval (an *iteration*).
    - customer decides which stories to implement in what order, as well as when and how often to produce a production releases of the system.
- Small Releases
  - start with the smallest useful feature set
  - release early and often, adding a few features each time
  - each iteration ends in a release

XP Practices

- System Metaphor
  - each project has an organizing metaphor, which provides an easy to remember naming convention.
  - the names should be derived from the vocabulary of the problem and solution domains
- Simple Design
  - always use the simplest possible design that gets the job done.
  - the requirements will change tomorrow, so only do what's needed to meet today's requirements.
  - uses the fewest number of classes and methods

Continuous Testing

- before programmers add a feature, they write a test for it. when the suite runs, the job is done.
  - Unit Testing
    - unit tests are automated tests written by the developers to test functionality as they write it.
    - each unit test typically tests only a single class, or a small cluster of classes.
    - unit tests are typically written using a unit testing framework (e.g., junit, parasoft).
  - Acceptance Testing
    - acceptance tests (functional tests) are specified by the customer to test that the overall system is functioning as specified. they typically test the entire system, or some large part.
    - when all the acceptance tests pass for a given user story, that story is considered complete.
    - at the very least, an acceptance test could consist of a script of user interface actions and expected results that a human can run.
    - ideally acceptance tests should be automated, either using a unit testing framework, or a separate acceptance testing framework.

XP Practices

- Refactoring
  - purpose is to improve the design of the code for greater comprehension, preparation for added features, ease of maintenance, etc. without changing behavior
  - refactorings include extracting methods, moving methods in an inheritance hierarchy, etc.
  - unit tests allows this to occur without danger

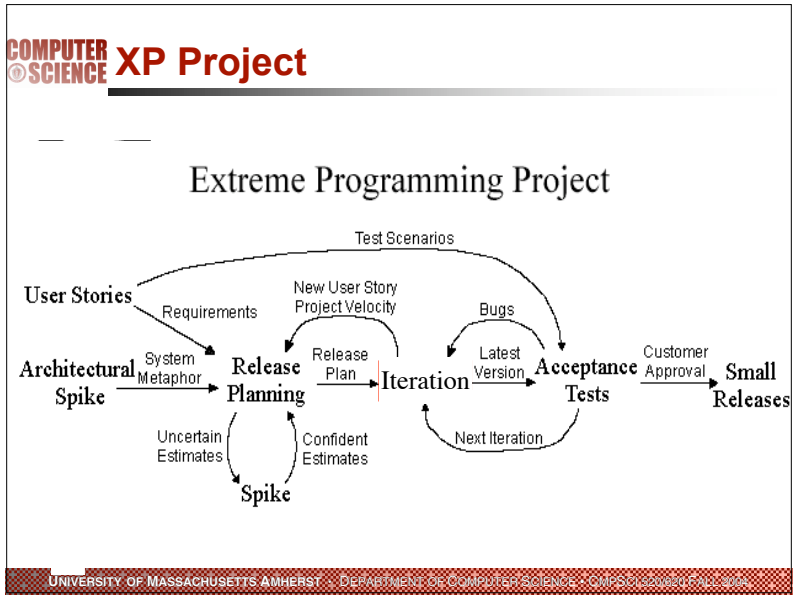
Refactoring

- Why?
  - cost of software change becomes higher when done late in process.
  - good design must anticipate future change. Makes design too complex.
  - design should evolve.
- Strategy
  - disciplined technique for restructuring an existing body of code.
  - alter structure, behavior unchanged.
  - series of small behavior preserving transformations.
  - each refactoring is small, less likely to go wrong.
  - system kept working after each step.
- Fowler's catalog of common refactorings
  - many refactorings can be automated
  - some tools help the refactoring process..

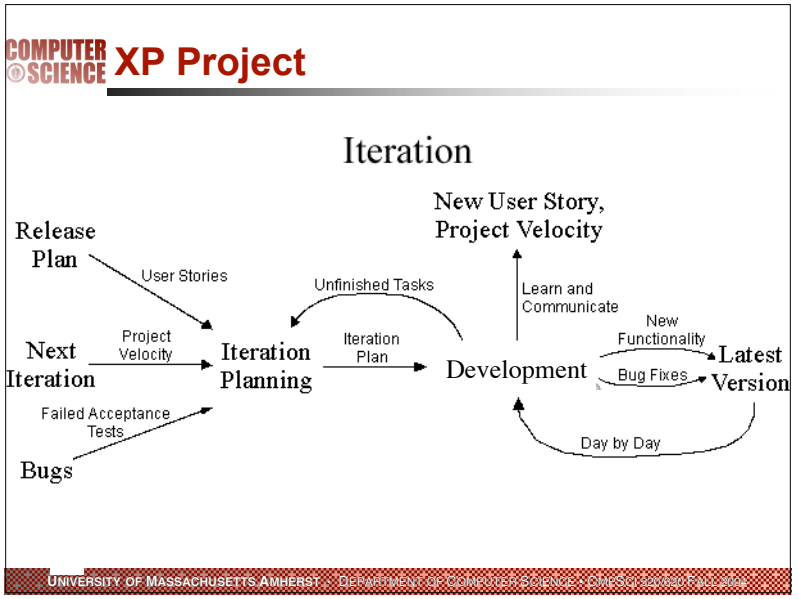
XP Practices

- Pair Programming
  - all production code is written by two programmers sitting at one machine; essentially, all code is reviewed as it is written.
    - Helm – at keyboard and mouse doing implementation
    - Tactician – thinking about the implications and possible problems

XP Project



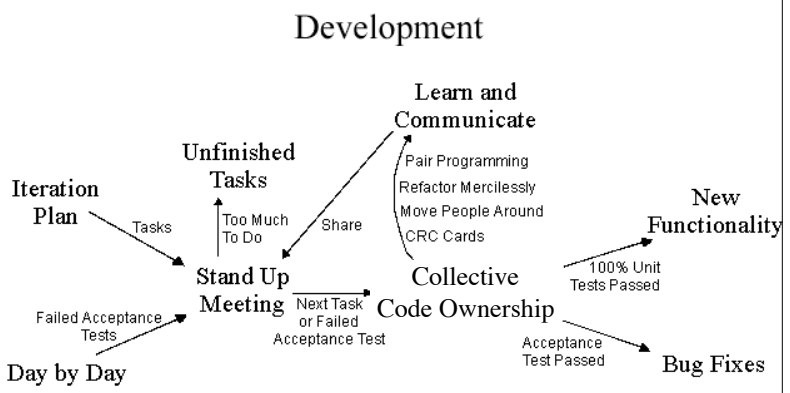
XP Project



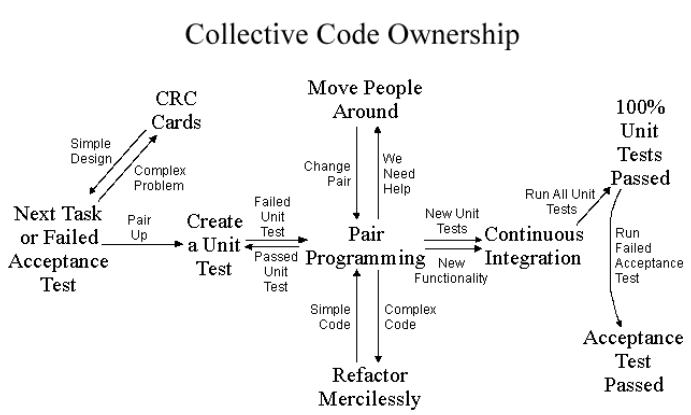
XP Practices

- Collective Code Ownership
  - no single person "owns" a module
  - any developer is expect to be able to work on any part of the code base at any time
  - improvement of existing code can happen at anytime by any pair
- Continuous Integration
  - all changes are integrated into the code base at least daily
  - the tests have to run 100% both before and after integration.
- 40-Hour Work Week
  - programmers go home on time. in crunch mode, up to one week of overtime is allowed.
  - multiple consecutive weeks of overtime are treated as a sign that something is very wrong with the process.
- On-Site Customer
  - development team has continuous access to a real live customer, that is, someone who will actually be using the system.
  - for commercial software with lots of customers, a customer proxy (usually the product manager) is used instead

XP Project



XP Project



Extreme Programming

- an example of an agile process
- short-term emphasis
  - frequent releases, no pre-design, task lists, metaphors
- customer oriented
  - stories, use-cases, on-site customers, feature negotiation
- contributions
  - pair-programming, test-first
- applicability
  - new, high-risk, small-to-medium projects

Design by Contract

- Originated by Bertram Meyer (Eiffel)
  - Incorporated in others methods (e.g., JML) and tools (e.g. Parasoft)
- methodology for evolving code together with its specification.
  - classes define their responsibility precisely.
  - class invariants, method preconditions and postconditions.
  - compiler instruments code to monitor.

Preconditions and postconditions

Class to client: *"If you promise to call r with pre satisfied, then I promise to deliver a final state in which post is satisfied …"*

- A method's precondition
  - says what must be true to call it, I.e., what must hold upon entry to method
  - binds the client.
- A method's normal postcondition
  - method guarantees it will hold upon exit
  - what is true when it returns normally (i.e., without throwing an exception).
- A method's exceptional postcondition
  - what is true when a method throws an exception.

```
//@ signals (IllegalArgumentException e) x < 0;
```

- Class Invariants
  - Global properties of a class.
  - Must be preserved by all exported routines.

DBC

- The role of a contract
  - may monitor at run time - debugging tool.
    - eiffel - by the compiler.
    - other languages - specific tools.
  - conceptual tool for correctness and robustness.
  - design aid.
  - aid to understanding
  - documentation
- advantages
  - clear responsibility for checking.
  - run time violation shows a bug:
    - Precondition violation -- bug in client.
    - Postcondition violation -- bug in supplier.
  - simplify code
    - method need not check precondition.
    - If precondition is not satisfied, do anything

Contracts and inheritance

Methodological implications of contracts on inheritance:

- Invariants and postconditions may only be strengthened;
- Preconditions may only be weakened.
- Eiffel enforces this principle..

Aspect-oriented programming

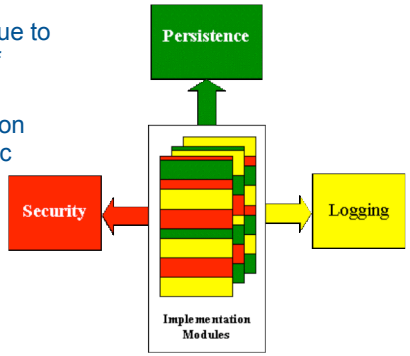
- AOP compliments O-O programming, but doesn't replace it
- the problem
  - some programming tasks cannot be neatly encapsulated in objects, but must be scattered throughout the code
  - examples:
    - logging (tracking program behavior to a file)
    - profiling (determining where a program spends its time)
    - tracing (determining what methods are called when)
    - session tracking, session expiration
    - special security management
  - the result is **crosscutting** code--the necessary code "cuts across" many different classes and methods

Some examples

```
public class SomeBusinessClass extends OtherBusinessClass
 {
       // Core data members

       // Override methods in the base class
       public void
 performSomeOperation(OperationInformation info)
       {

              .

              // ==== Perform the core operation ====

       }

 }
```

Breaking modularity

- Got the picture ?
- Non-modularization due to client-server nature of OOP
- Current popular solution EJB, servlets, dynamic proxies



modularity

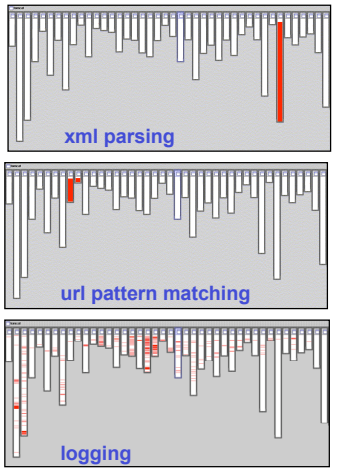
intuitive definition

a concern is implemented[1] in a modular way if the code for the concern is:

- localized and
- has a clear interface with the rest of the system

[1] coded, designed, modeled …

code from org.apache.tomcat



COMPUTER SCIENCE

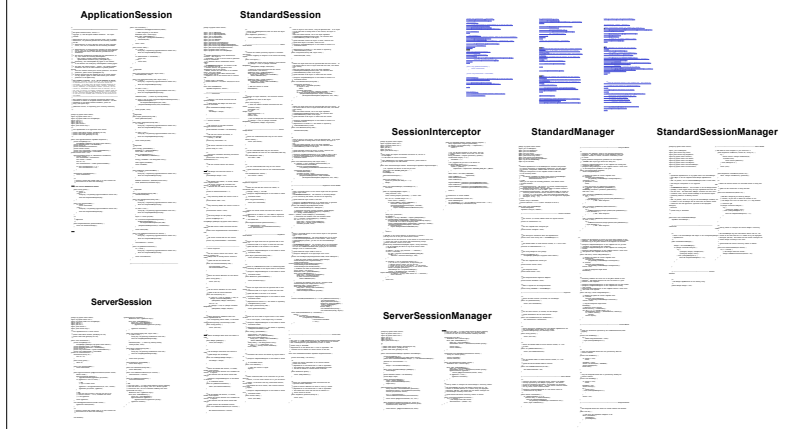
session expiration is not modularized…

COMPUTER SCIENCE

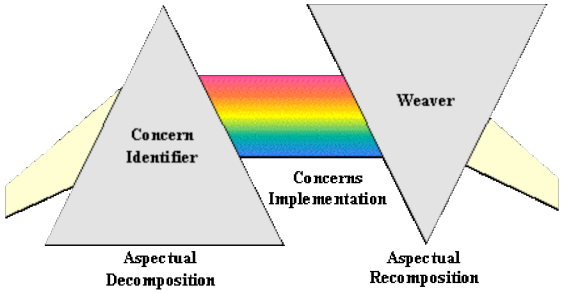
Non-modularization

- Symptoms
  - Code tangling
  - Code scattering
    - Duplicated code blocks
    - Complementary code blocks
- Consequences
  - Redundant code
    - Same fragment of code in many places
  - Difficult to reason about
    - Non-explicit structure
    - The big picture of the tangling isn't clear
  - Difficult to change
    - Have to find all the code involved...
    - ...and be sure to change it consistently
    - ...and be sure not to break it by accident
  - Inefficient when crosscuting code is not needed

If we just could…



How does AOP do it ?



AOP development stages

AspectJ™

- AspectJ is a small, well-integrated extension to Java
  - Based on the 1997 PhD thesis by Christina Lopes, *A Language Framework for Distributed Programming*
- AspectJ modularizes crosscutting concerns
  - That is, code for one *aspect* of the program (such as tracing) is collected together in one place
- The AspectJ compiler is free and open source
- AspectJ works with JBuilder, Forté, Eclipse, probably others

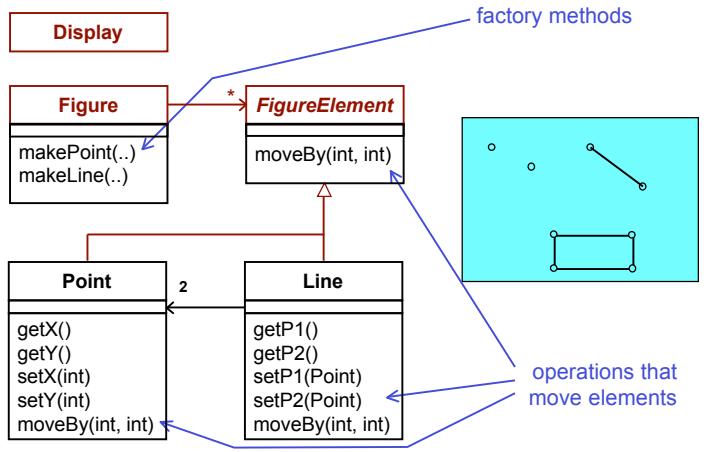
What is a Concern ?

- concern is a particular goal, concept, or area of interest
- concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts

Terminology

- A **join point** is a well-defined point in the program flow
- A **pointcut** is a group of join points
- **Advice** is code that is executed at a pointcut
- **Introduction** modifies the members of a class and the relationships between classes
- An **aspect** is a module for handling crosscutting concerns
  - Aspects are defined in terms of pointcuts, advice, and introduction
  - Aspects are reusable and inheritable

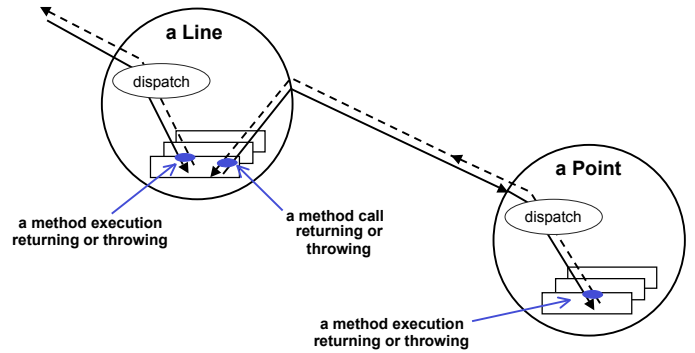
a simple figure editor



join points

imagine `l.moveBy(2, 2)`

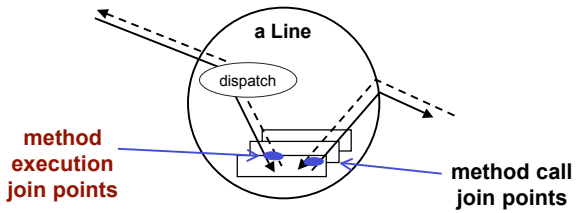
key points in dynamic call graph



Join points

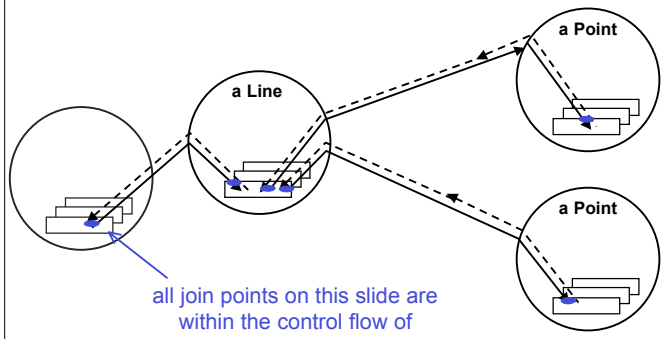
- A join point is a well-defined point in the program flow
  - We want to execute some code ("advice") each time a join point is reached
  - We do *not* want to clutter up the code with explicit indicators saying *"This is a join point"*
  - AspectJ provides a syntax for indicating these join points "from outside" the actual code
- A join point is a point in the program flow "where something happens"
  - Examples:
    - When a method is called
    - When an exception is thrown
    - When a variable is accessed

join point terminology



- several kinds of join points
  - method & constructor call
  - method & constructor execution
  - field get & set
  - exception handler execution
  - static & dynamic initialization

join point terminology



Pointcuts

- Pointcut definitions consist of a left-hand side and a right-hand side, separated by a colon
  - The left-hand side consists of the pointcut name and the pointcut parameters (i.e. the data available when the events happen)
  - The right-hand side consists of the pointcut itself
- Example pointcut:

```
pointcut setter(): call(void setX(int));
```

name

no parameters

Example pointcut designators

- When a particular method body executes:
  - execution(void Point.setX(int))
- When a method is called:
  - call(void Point.setX(int))
- When an exception handler executes:
  - handler(ArrayOutOfBoundsException)
- When the object currently executing (i.e. this) is of type SomeType:
  - this(SomeType)
- When the target object is of type SomeType
  - target(SomeType)
- When the executing code belongs to class MyClass
  - within(MyClass)
- When the join point is in the control flow of a call to a Test's no-argument main method
  - cflow(call(void Test.main()))

Pointcut designator wildcards

- It is possible to use wildcards to declare pointcuts:
  - execution(* *(..))
    - Chooses the execution of any method regardless of return or parameter types
  - call(* set(..))
    - Chooses the call to any method named set regardless of return or parameter type
    - In case of overloading there may be more than one such set method; this pointcut picks out calls to all of them

pointcut composition

- a pointcut is a kind of predicate on join points that:
  - can match or not match any given join point and
  - optionally, can pull out some of the values at that join point
- pointcuts compose like predicates, using &&, || and !

a "void Line.setP1(Point)" call

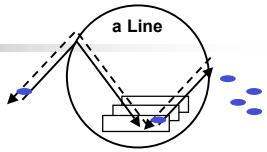
or

```
call(void Line.setP1(Point)) ||
call(void Line.setP2(Point));
```

a "void Line.setP2(Point)" call

whenever a Line receives a "void setP1(Point)" or "void setP2(Point)" method call

after advice



action to take after computation under join points

```
aspect DisplayUpdating {

  pointcut move(FigureElement fe):
    target(fe) &&
    (call(void FigureElement.moveBy(int, int)) ||
     call(void Line.setP1(Point))              ||
     call(void Line.setP2(Point))              ||
     call(void Point.setX(int))                ||
     call(void Point.setY(int)));

  after(FigureElement fe) returning: move(fe) {
    Display.update(fe);
  }
}
```


Figure Editor with AspectJ

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Display.update();
  }
  void setP2(Point p2) {
    this.p2 = p2;
    Display.update();
  }
}
```

```
class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    Display.update();
  }
  void setY(int y) {
    this.y = y;
    Display.update();
  }
}
```

```
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
  }
  void setP2(Point p2) {
    this.p2 = p2;
  }
}
```

```
class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
  }
  void setY(int y) {
    this.y = y;
  }
}
```

```
aspect DisplayUpdating {

  pointcut move():
    call(void FigureElement.moveBy(int, int)) ||
    call(void Line.setP1(Point))              ||
    call(void Line.setP2(Point))              ||
    call(void Point.setX(int))                ||
    call(void Point.setY(int));

  after() returning: move() {
    Display.update();
  }
}
```

- clear display updating module
  - all changes in single aspect
  - evolution is modular
- no locus of "display updating"
  - evolution is cumbersome
  - changes in all classes
  - have to track & change all callers

AspectJ advice

- Before advice runs as a join point is reached, before the program proceeds with the join point
- After advice on a particular join point runs after the program proceeds with that join point
  - after returning advice is executed after a method returns normally
  - after throwing advice is executed after a method returns by throwing an exception
  - after advice is executed after a method returns, regardless of whether it returns normally or by throwing an exception
- Around advice on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point

Concluding remarks

- Aspect-oriented programming (AOP) is a new paradigm--a new way to think about programming
- AOP is somewhat similar to event handling, where the "events" are defined outside the code itself
- AspectJ is not itself a complete programming language, but an adjunct to Java
- AspectJ does not add new capabilities to what Java can do, but adds new ways of modularizing the code
- AspectJ is free, open source software
- Like all new technologies, AOP may--or may not--catch on in a big way

Myths and realities of AOP

- The program flow in an AOP-based system is hard to follow True
- AOP doesn't solve any new problems True
- AOP promotes sloppy design False
- AOP is nice, but a nice abstract OOP interface is all you need False
- AOP compiler simply patches the core implementation False
- AOP breaks the encapsulation True, but ..
- AOP will replace OOP False

some AOP languages

		means of … join points	
JPM	join points	identifying	specifying semantics at
AspectJ dynamic JPM	points in execution call, get, set…	signatures w/ patterns, static & dynamic props of JPs	advice declarative & imperative composition of code
static JPM	class members	signatures	add members
Composition Filters	message sends & receptions	signature & property based object queries	wrappers declarative (filters) imperative (~ advice)
Hyper/J	members	signatures w/ patterns, whole class ops	add, compose (and remove) members
Demeter traversals	when traversal reaches object or edge	class & edge names	define visit method

COMPUTER SCIENCE

Other “hot” technology

- Generative programming
- Meta programming
- Reflective programming
- Compositional filtering
- Adaptive programming
- Subject oriented programming
- Intentional programming