From designing to coding

- 1st step: *sensibly* split work among team members
  - Choose splits along “thin” interfaces
    - Probably not equal parts; split biggest parts again later
  - Formalize the interfaces – think of them as contracts
  - Write *least-coupled* parts first … most-coupled last
    - i.e., classes that don’t depend on any other classes

- Oh yeah, one more thing to think about: Reserve ample *time* for testing!
* .h files – consider them contracts

- **Write interfaces first** – share with the team
- Helps to formalizes much of the contract
  - Precisely defines available *services* (methods)
  - But pre- and post-conditions are not insured
    - These are communicated by documentation only
- **Implement classes and their client classes independently**
  - Can even compile clients (but cannot fully test)
Pre- and post-conditions

- **Pre-conditions** – what the client is responsible for
  - The “requires” clauses of the contract
    - Especially include any restrictions on calling arguments
    - Also any associations that should already exist

- **Post-conditions** – what will be accomplished by the operation *if* the pre-conditions are met
  - The “effects” and/or “modifies” contract clauses
  - Include objects created/destroyed, associations formed/broken, attribute values modified
  - Also remember to list side effects, and identify any exceptions that might be thrown
Converting designs into code

- Largely a direct translation of key artifacts
  - Class specs – variables and method definitions
  - Class and package diagrams – associations
    - Translate to instance variables and/or method arguments
  - Interaction and state-chart diagrams – method calls and sequences
- Still involves creativity, and probably change
  - Good ideas often arise during coding – okay, go for it
    - But also plan to revise design artifacts to match later
class SalesLineItem {
    public:
        SalesLineItem(ProductSpecification spec, int qty);
        Money getSubtotal();
        ...
    private:
        int quantity; // attribute
        ProductSpecification productSpec; // association
};

Code from class diagrams

<table>
<thead>
<tr>
<th>SalesLineItem</th>
<th>described by</th>
<th>ProductSpecification</th>
</tr>
</thead>
<tbody>
<tr>
<td>quantity : Integer</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>getSubtotal() : Money</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Implementing interactions – e.g., `enterItem(id, qty)`

```java
{  
    ProductSpecification spec = catalog.getSpecification(id);  
    sale.makeLineItem(spec, qty);  
}
```

```
1: spec := getSpecification(id)
```

```
enterItem(id, qty)   2: makeLineItem(spec, qty)
```

```
:Register
```

```
:Sale
```

```
:Product
Catalog
```
Least- to most-coupled order
Use your resources

- i.e., “don’t reinvent the wheel” if possible
  - Use STL and other library tools
  - See books like *Effective C++* (by Scott Meyers) for advice
- On a real project: consider 3rd party solutions, existing code, other quick fixes
  - Of course, we hope you mostly do yourself in CS 48
- Program in pairs – it works!
- And don’t wrestle with revision control problems
  - use a revision control system
Revision control problem

Two users read the same file
- Repository
  - Read to A
    - A
      - Harry
  - Read to A
    - A
      - Sally

They both begin to edit their copies
- Repository
  - A
    - Harry
    - Sally

Harry publishes his version first
- Repository
  - A
    - Write to A'
      - A'
        - Harry
      - A''
        - Sally

Sally accidentally overwrites Harry's version
- Repository
  - A''
    - Write to A''
      - A'
        - Harry
      - A''
        - Sally
Lock-Modify-Unock Solution

Harry "locks" file A, then copies it for editing
Repository

Harry writes his version, then releases his lock
Repository

While Harry edits, Sally's lock attempt fails
Repository

Now Sally can lock, read, and edit the latest version
Repository
Copy-Modify-Merge Solution: 1

Two users copy the same file:
- Repository
  - A
  - Harry
  - Read
  - A
  - Sally
  - Read

They both begin to edit their copies:
- Repository
  - A
  - Harry
  - A'
  - Sally
  - A'

Sally publishes her version first:
- Repository
  - A''
  - Write
  - A''
  - Harry
  - A'
  - Sally

Harry gets an "out-of-date" error:
- Repository
  - A''
  - Write
  - A''
  - Harry
  - A'
  - Sally

Continued next slide
Copy-Modify-Merge Solution: 2

- **Subversion** – an open source and widely used revision control
Testing – goal is to **find faults**

- Faults (a.k.a. bugs) cause systems to fail
  - e.g., a system crashes – the most obvious type of fault
  - e.g., a security system that allows unauthorized entry
  - e.g., a shot-down plane that continues on its flight path
- Can verify the presence of bugs, not their absence
- Testing and debugging are separate processes
  - Testing identifies faults
  - Debugging corrects/removes faults
    - But it can also introduce new ones, so retesting is required
When are faults introduced?

- During requirements analysis
  - Incorrect, missing, or unclear requirements
- During domain analysis and system design
  - Incorrect or unclear translation of problem
  - Incorrect or unclear design specification
- During system implementation
  - Misinterpretation of system design
  - Incorrect syntax or semantics
- Even during testing
  - New faults introduced when old ones corrected

Note how naïve to consider this the only source of bugs
Testing steps

- **Unit testing** – insure each part is correct
  - Each method of each class of each package should be tested independently
- **Integration testing** – insure parts work together
- **System testing**
  - Functional tests – a.k.a. use case testing
  - Performance tests – test system attribute requirements
  - Acceptance tests – client hands-on testing
  - Installation tests – final platform testing (on-site)
Unit and integration testing

- Test parts of the system before the whole
  - Units – test basic parts individually (each function, each class, each package)
  - Integration – test basic connections between parts (not whole system at once yet though)
- Requires special purpose test programs
  - i.e., “driver” programs and “stubs”
  - Or can use a framework like CppTest – patterned after ultra-popular Junit and Cunit
System testing phases

- **Use case testing**
  - Test pre- and post-conditions of system functions
  - Best if independent of the user interface
    - i.e., also requires special purpose testing code
- **Performance, acceptance, installation tests**
  - All involve the complete working system, GUI and all
- **If any changes to code** – rerun *all* tests
  - Term for this philosophy: *regression testing*
Tragic truth: testing takes time

- But it can save time and money in the long run
  - Get in the habit: “code a little, test a little, …”
- Inadequate testing costs lots of real world $$$ and maybe lives
- Fact: costs of testing/debugging increase as development progresses
  - Cheapest during requirements analysis (especially if an impossible requirement is uncovered)
  - Cheaper during unit than integration testing, …