Overview - Object-Oriented Analysis and Design

• Design thinking

• Object Modeling Technique
  – Object-Oriented Analysis
  – Object-Oriented Design

• Three models
  – Object model
  – Dynamic model
  – Functional model

• Four phases
Design Goals

• Design transforms requirements into
  – an architecture diagram
    □ subsystems, modules and their relationships
  – a detailed design
    □ a specification of the abstract interface, data structures, and algorithms of each module

• Also develops
  – a review plan for ensuring the design meets the requirements
  – a test plan for ensuring the implementation meets the design
Object-Oriented Software Development

- **Object-Oriented Methodology**
  - development approach used to build complex systems using the concepts of object, class, polymorphism, and inheritance with a view towards reusability
  - encourages software engineers to think of the problem in terms of the application domain early and apply a consistent approach throughout the entire life-cycle

- **Object-Oriented Analysis and Design**
  - analysis models the “real-world” requirements, independent of the implementation environment
  - design applies object-oriented concepts to develop and communicate the architecture and details of how to meet requirements
Object Modeling Technique
Process via UML

• OMT [Rumbaugh et al., 1991] consists of
  – building three complementary models of the system
  – adding implementation details to the models
  – implementing the models

• OMT includes a set of
  – phases [processes]
  – diagramming techniques

• OMT has four phases
  – object-oriented analysis builds a real-world model
  – system design determines overall architecture of system
  – object design decides upon data structures and algorithms
  – implementation translates design into programming language
**OMT Stages and Models**

**Analysis**
- Model of real-world situation
- What?

**System Design**
- Overall architecture (sub-systems)

**Object Design**
- Refinement of Design
- Algorithms/data structures to implement each class

**Implementation**
- Translation of object classes and relationships to a particular object-oriented language
Introduction to Object-Oriented Analysis

Object-Oriented Analysis is the “requirements phase” of Object-Oriented Software Development
- think of it as an alternative semi-formal technique

Semi-formal technique
- class modeling
- dynamic modeling
- functional modeling

These steps focus on
- data
- actions
- their relationships

Reuses familiar tools
- E-R diagrams
- Finite State Machines
- Data flow diagrams

Steps and diagrams
- typically performed in parallel after initial class definition
- must be kept in synch
Object-Oriented Analysis

• Builds a “real-world” model from requirements
  – client interviews, domain knowledge, real-world experience collected in use cases and other simple notations

• OOA models address three aspects of the system (its objects)
  – class structure and relationships
  – sequencing of interactions and events
  – data transformations and computations
Models of Object-Oriented Analysis (cf UML)

- **Class Model**
  - static structure
  - what objects are in the system?
  - how are they related?
- **Dynamic Model**
  - behavioral aspects
  - what events occur in the system
  - when do they occur and in what order?
- **Functional Model**
  - data transformations
  - “what” does the system do
- **Data-Oriented**
- **Action-Oriented**
- **Both Data and Actions**
OO Analysis and Design: Steps

- Class Modeling
- Dynamic Modeling
- Functional Modeling
- Add Operations to the Class Model
- Iterate and refine the models
  - After the first iteration, steps may occur in parallel or out of order
  - All models must be kept in synch as changes are made
Class Modeling

- Identify objects and classes
- Prepare a data dictionary
- Identify associations between objects
- Identify class attributes and initial set of operations
- Organize object classes using inheritance
Classes, Attributes and Operations

- Attributes define the properties of the objects
  - every instance of the class has the same attributes
  - an attribute has a data type
  - the values of the attributes may differ among instances

- Operations define the behavior of the objects
  - action performed on or by an object
  - available for all instances of the class
  - need not be unique among classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Attributes</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ball</td>
<td>radius, weight</td>
<td>catch, throw</td>
</tr>
<tr>
<td>football</td>
<td>air pressure</td>
<td>pass, kick, hand-off</td>
</tr>
<tr>
<td>baseball</td>
<td>liveness</td>
<td>hit, pitch, tag</td>
</tr>
</tbody>
</table>
Object Model Notation (refresher)

Classes are represented as rectangles;
The class name is at the top, followed by attributes (instance variables) and methods (operations)
Depending on context some information can be hidden such as types or method arguments

Objects are represented as rounded rectangles;
The object’s name is its classname surrounded by parentheses
Instance variables can display the values that they have been assigned; pointer types will often point (not shown) to the object being referenced

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstanceVariable1</td>
</tr>
<tr>
<td>InstanceVariable2: type</td>
</tr>
<tr>
<td>Method1()</td>
</tr>
<tr>
<td>Method2(arguments) return type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(Class Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstanceVariable1 = value</td>
</tr>
<tr>
<td>InstanceVariable2: type</td>
</tr>
<tr>
<td>Method1()</td>
</tr>
<tr>
<td>Method2(arguments) return type</td>
</tr>
</tbody>
</table>
OMT Instantiation Notation

Class Name

| attribute_1: data_type_1 = default_1 |
| attribute_2: data_type_2 = default_2 |
| ... |
| attribute_m: data_type_m = default_m |

(Class Name)

attribute_1 = value_1
attribute_2 = value_2
... attribute_m = value_m

Class

Instance
Instantiation - Example

Person

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe Smith</td>
<td>age=39</td>
<td>weight=158</td>
</tr>
<tr>
<td>Mary Wilson</td>
<td>age=27</td>
<td>weight=121</td>
</tr>
</tbody>
</table>
Inheritance

• Classes with similar attributes and operations may be organized hierarchically

• Common attributes and operations are factored out and assigned to a broad superclass (generalization)
  – generalization is the “is-a” relationship
  – superclasses are ancestors, subclasses are descendants

• Classes iteratively refined into subclasses that inherit the attributes and operations of the superclass (specialization)

• Do you see any disadvantages to inheritance?
OMT Inheritance Notation

Generalization

Superclass

Class

Attributes

Operations

Ball

Radius, Weight

Throw, Catch

Subclasses

Football

air pressure

pass, kick, hand-off

Baseball

liveness

hit, pitch, tag

Basketball

air pressure, dimples

shoot, dribble, pass
Association and Links

• An association is a relation among two or more classes describing a group of links, with common structure and semantics

• A link is a relationship or connection between objects and is an instance of an association

• A link or association is inherently bi-directional
  – the name may imply a direction, but it can usually be inverted
  – the diagram is usually drawn to read the link or association from left to right or top to bottom

• A role is one end of an association
  – roles may have names
OMT Association Notation

Class, Association, and Roles

Person

Works For

equivalent

Employs

Company

Employer

Employee

Person

Company

Object and Link

(Person)

Johnson

Works For

(Company)

IBM
Association and Links

Class diagram

- **Country**
  - name
- **City**
  - name
  - has-capital

Instance diagram

- **(Country)**
  - Canada
  - has-capital
  - Ottawa
- **(Country)**
  - France
  - has-capital
  - Paris
- **(Country)**
  - Austria
  - has-capital
  - Vienna
Multiplicity of Associations

- Multiplicity is the number of instances of one class that may relate to a single instance of an associated class
  - 1-to-1
  - 1-to-many (0 or more)
  - 1-to-(zero-or-one) ‘optional’
  - 1-to-(one-or-more) ‘required’
  - 1-to-n

\[
\begin{array}{c}
\text{Required:} 1+ \\
\text{Optional:} \circ \\
\text{One:} \bullet \\
\text{Multiple:} n
\end{array}
\]
OMT Multiplicity Notation

Each course has at least one instructor and between 6 and 65 students
A student may take many courses
An instructor may teach many courses
Link attributes for associations

- **Person**
  - name
  - address

- **works-for**

- **Company**
  - name
  - salary
  - job title
Aggregation

- Aggregation is a special form of association that indicates a “part-of” relationship between a whole and its parts
- Useful when the parts do not have independent existence
  - A part is subordinate to the whole
- In an aggregation, properties and operations may be propagated from the whole to its parts
OMT Aggregation Notation

```
Window

TitleBar  ScrollBar  Border
```
Multilevel aggregation
An Example

FastData Inc. wants a subsystem to process office supply orders via the Web. The user will supply via a form their name, password, account number, and a list of supplies along with an indication of the quantities desired. The subsystem will validate the input, enter the order into a database, and generate a receipt with the order number, expected ship date, and the total cost of the order. If the validation step fails, the subsystem will generate an error message describing the cause of the failure.
Purpose of Example

• We will demonstrate the UML / OMT using this example
  – Class modeling will be done first
  – Dynamic and Functional modeling will occur next
  – Detailed design after that

• Things to remember
  – This example does not demonstrate how the technique is applied to ALL problems. Be sure to distinguish between the details of the example and the details of the technique!
Concise Problem Definition

• Define the problem concisely
  – Use only a single sentence

“FastData, Inc. employees may order office supplies via the Web and receive a receipt confirming the order”

• This is the first step towards identifying the classes of the subsystem
Informal Strategy

• Identify the constraints governing the system
  – Use only a single paragraph

“FastData, Inc. employees may order office supplies via the Internal Web and receive a receipt confirming the order. The order must include the user name, user password, account number, and the list of supplies. A receipt must be generated containing an order number, ship date, and total cost. If the order is valid, it must be entered into an order database. If the order is invalid, an error message must be generated.”

• We now have more information to be used in identifying classes for the subsystem
Formalize the Strategy

• Identify the nouns of the description, which serve as the basis for identifying the subsystem’s classes.
  – Look for out-of-domain nouns (and throw them out!)
  – Look for abstract nouns (use these for attributes)
  – The remaining nouns are good candidates!

“FastData, Inc. *employees* may order *office supplies* via the *Internal Web* and receive a *receipt* confirming the *order*. The *order* must include the *user name*, *user password*, *account number*, and the *list of supplies*. A *receipt* must be generated containing an *order number*, *ship date*, and *total cost*. If the *order* is valid, it must be entered into an *order database*. If the *order* is invalid, an *error message* must be generated.”
Nouns

• Out-of-Domain
  – Internal Web

• Abstract
  – user name
  – user password
  – account number
  – order number
  – ship date
  – total cost
  – list of supplies
  – office supplies -> item

• Good Candidates
  – employee
  – item (was office supplies)
  – receipt
  – order
  – order database
  – error message

• Notes
  We have decided not to worry about the Web in this design. Instead we focus on the inputs and outputs and defer the Web details until later.
Class Model

<table>
<thead>
<tr>
<th>employee</th>
<th>order</th>
<th>order DB</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>number</td>
<td></td>
</tr>
<tr>
<td>password</td>
<td>account</td>
<td></td>
</tr>
<tr>
<td></td>
<td>total cost</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>error message</th>
<th>receipt</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>explanation</td>
<td>order number</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>ship date</td>
<td>quantity</td>
</tr>
<tr>
<td></td>
<td>total cost</td>
<td>price</td>
</tr>
</tbody>
</table>
Since both receipts and error messages will be generated as output it might make sense to have them as subclasses of a more general class. We do not know enough yet to assign it attributes however.
Class Model, relationships

employee
name
password

order
number
account
total cost

order DB

error message
explanation

receipt
order number
ship date
total cost

item
name
quantity
price

1+
Overview - Object-Oriented Analysis and Design

• Three models
  – Object model
  – Dynamic model
  – Functional model

• Four phases
  – object-oriented analysis
  – system design
  – object design
  – Implementation

• Detailed Design

• Integration Testing
OMT Analysis and Design: Steps

• Class Modeling
• Dynamic Modeling
• Functional Modeling
• Add Operations to the Class Model
• Iterate and refine the models
  – After the first iteration, steps may occur in parallel or out of order
  – All models must be kept in synch as changes are made
Dynamic Modeling

• Prepare scenarios
• Identify events between objects
• Prepare an event trace for each scenario
• Build a state diagram
• Match events between objects to verify consistency
Dynamic Model Diagrams

• The dynamic model tracks behavior over time
  – described in terms of change in objects or event sequences between objects

• Event Trace Diagrams
  – show typical dialog or usage scenarios as well as exceptional and/or special cases

• State Diagrams
  – relates events, states, and state transitions
  – a scenario is a path through the state diagram
Events and Scenarios

• An event is [an ‘instantaneous’ change of state in the application domain] that triggers a change to an object’s object state (?)
  – events have attributes, which are the information transferred from one object to another

• A scenario is a specific sequence of events representing a path through a system’s state space

• Legitimate scenarios
  – common paths (e.g. frequently used functionality)
  – Error conditions and known exceptions

• An event trace extends the scenario to clarify events between objects
Event classes and attributes

- **Event Classes**
  - airplane departs (airline, flight number, city)
  - mouse button pushed (button, location)
  - phone receiver lifted
  - digit dialed (digit)

- **Events**
  - United Flight 23 departs from Rome
  - right mouse button pushed at (29, 30)
  - phone receiver lifted
  - digit dialed (2)
An example scenario

- Scenario for a phone call
  - caller lifts receiver
  - dial tone begins
  - caller dials digit (2)
  - caller dials digit (7)
  - caller dials digit (7)
  - specified phone rings
  - etc.
OMT Event Trace Notation

- objects are vertical lines
- events are horizontal lines
- arrows indicate sender and receiver
- time passes from top to bottom

```
Customer

select method of payment
select "credit"
insert card
slide card through reader
"select grade"
select "premium"
"pump on"
display unit cost, total cost, gallons dispensed
pump gas
update display with total cost, gallons dispensed

Pump


Credit Corp

verify account
return "approved"
charge total cost to account
```


Event Trace: example

**Caller**
- caller lifts receiver
- dial tone begins
- dials (2)
- dial tone ends
- dials (7)
- dials (7)
- dials (6)
- ringing tone

**Phone line**
- phone rings
- phones connected
- connection broken
- callee hangs up

**Callee**
- answers phone
- phones connected
- callee hangs up
- connection broken
States and Transitions

• A *state* is an interval between events (values of relevant variables to the problem…)
  – it may have an activity that can trigger starting, intermediate and ending events
  – defined in terms of a subset of object attributes and links

• A state *transition* is a change in an object’s attributes and links
  – it is the response of an object to an event
  – all transitions leaving a state must correspond to distinct events
OMT State Notation

- states represented as nodes: **rounded rectangles with state name**
  - initial state represented as solid circle
  - final state represented as bull’s eye
- transitions represented as edges between nodes and labeled with an *event name*

```
+----------------+       +----------------+
|                |       |                |
|   STATE-1      |       |   STATE-2      |
|                |       |                |
|                +----------------+       +----------------+
|                |       |                |
| Event-c        |       | Event-a        |
|                |       |                |
| STATE-3        |       |                |
|                |       |                |
|                +----------------+       +----------------+
|                |       |                |
| Event-b        |       | Event-e        |
|                |       |                |
| STATE-2        |       |                |
+----------------+       +----------------+```

result
OMT State Diagram - Example

Chess game

Start

White’s turn

Black’s turn

- Black moves
- White moves
- Checkmate
- Stalemate

Black wins

Draw

White wins

Checkmate

Stalemate
Guards, Activities and Actions

- **Guards** are boolean conditions on attribute values
  - transition can only happen when guard evaluates to “true”
  - automatic transitions occur as soon as an activity is complete (check guard!)
- **Activities** take time to complete
  - activities take place within a ‘state’
- **Actions** are relatively instantaneous
  - actions take place on a transition or within a state (entry, exit, event actions)
  - output can occur with an event

![Diagram showing state transitions and events with guards and actions]
Guards, Activities and Actions - Example

Vending machine model

Idle

Collecting money

- coins in (amount) / add to balance
- cancel / refund coins

- coins in (amount) / set balance

- [item empty]
- select (item)
- [change < 0]
- [change = 0]
- [change > 0]

- do: test item and compute change
- do: dispense item
- do: make change
OMT State Relationships

- States can be nested or concurrent
- Events can be split and merged
State Generalization: example

Transmission

Neutral

Reverse

push N

push R

push N

push F

Forward

First

Second

Third

upshift

downshift

upshift

downshift

stop
Returning to the FastData example

- Let's define a scenario for an office supply order processor: a successful order
  - Alternatively we could describe a scenario for an unsuccessful order

- Assumptions
  - We are not going to consider how the order form is transmitted to our system nor how our receipt is transmitted back
  - The employee object is responsible for validating the input to the system
A successful order

- input received (we don’t care how)
- create employee object
- pass input to employee
- validate name and password
- create order object
- validate account number
- for each item
  - create item
  - add item to order and validate item
- compute total cost
- add order to order DB and retrieve order number and ship date
- generate receipt
- return receipt (we don’t care how)
Event Trace

validate name/password

validated

create

validate account number

validated

create

add

validate item

validated

repeat
Event Trace, continued

<table>
<thead>
<tr>
<th>Employee</th>
<th>Order</th>
<th>Item</th>
<th>Receipt</th>
<th>Order DB</th>
<th>Employee DB</th>
<th>Product DB</th>
<th>Account DB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>compute cost</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>retrieve cost</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>add order</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>retrieve order number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>retrieve ship date</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>create</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(One Possible) State Transition Diagram

- **Idle**
  - Input received
  - Return receipt

- **Initialization**
  - Employee Created
  - Employee Validated
  - Order Created

- **Process Order**
  - Create Item
    - [remaining items > 0]
  - Process Order
  - Order Finished?

- **Finalize**
  - do: add order
  - create receipt

- **Finalize**
  - [remaining items = 0]
OMT Analysis and Design: Steps

- Class Modeling
- Dynamic Modeling
- Functional Modeling
- Add Operations to the Class Model
- Iterate and refine the models
  - After the first iteration, steps may occur in parallel or out of order
  - All models must be kept in synch as changes are made
Functional Modeling

• Identify input and output values
• Build data flow diagrams showing transformation and functional dependencies (expanding non-trivial processes)
• Describe functions (in some language)
• Identify constraints between objects (add to DM and FM)
OMT DFD Notation

- Processes transform data
- Actors are sources or sinks of data (= Active Objects)
- Data stores are persistent repositories of data, which may be accessed or updated (= Passive Objects)
- Data flows between processes, actors, and data stores
Data Value Notation

- Data may be a composed, decomposed, or duplicated

```
data-1
  composite
    data-1
    data-2
```

```
composite
  data-1
    data-2
```

```
data-1
  data-1
  data-1
```

```
data-1
  data-1
  data-1
```
Control Flow in the DFD
Hierarchical DFD

• High-level functionality iteratively refined into smaller functional units
  – each high-level process may be expanded into a separate DFD
  – top-level processes correspond to operations on complex objects, while lower-level processes are operations on basic objects
• Nesting depth is dependent on application
  – terminates with simple functions
  – each level must be coherent
• Hierarchical DFD corresponds to the following
  – context diagram shows boundaries of system
  – mid-level DFDs show context decomposition
  – primitive DFDs are simple functions that need not be expanded
Data Flow Diagram: Office Supply example

Web Server

input stream

Employee DB

verification

name/password

employee

Account DB

verification

account number

validate employee

validate order

response (receipt)

order
Data Flow Diagram: Office Supply example

- Process order
  - Verification
  - Item
  - Order

- Order DB
  - Order info
  - Order

- Validated order

- Finalize order

- Response (receipt)
OMT Analysis and Design: Steps

- Class Modeling
- Dynamic Modeling
- Functional Modeling
- Add Operations to the Class Model
- Iterate and refine the models
  - After the first iteration, steps may occur in parallel or out of order
  - All models must be kept in synch as changes are made
Add Operations to the Object Model

• From the Object Model:
  – Reading/writing object attributes (e.g., get_width, get_height of Rectangle)

• From Events, State Actions, and Activities in the Dynamic Model:
  – Each event sent to an object => operation
    (e.g., Vending machine: set_balance)
  – Actions/activities may be operations
    (e.g., Vending machine: do: test item and compute change)

• From Functions in the Functional Model:
  – Each function in the DFD corresponds to an operation
    (e.g., bank example: subtract withdrawal from Account)
Relation of the three models

- Things
  - *object model*
- Interactions
  - *dynamic model*
- Transformations
  - *functional model*
Relation of Dynamic Model to Class Model

• Dynamic model provides a second dimension - time - to objects and classes

• Dynamic model builds upon and is derived from object model
  – states in dynamic model represent sets of attribute and link values in object model
  – events in dynamic model yield operations in object model

• Relation between organization
  – inherent differences in objects are distinguished in object model as distinct classes
  – temporal differences in object attributes are distinguished in dynamic model as distinct states
Relation of Functional Model to Class and Dynamic Model

• Functional model describes the actions (what), the dynamic model describes the timing (when), and the class model describes what takes action (who)

• Functional model builds on / derived from class model
  – processes in the functional model correspond to operations on objects
  – The input streams of processes in the functional model identify objects that are related by function
  – data flows in the functional model correspond to objects or attribute values in the class model

• Functional model may capture actions not part of any scenario
OMT: Four phases

• Object-oriented analysis
  – builds a real-world model

• System design and Architecture
  – determines overall architecture of system

• Object design
  – decides upon data structures and algorithms

• Implementation
  – translates design into programming language
System Design and Architecture

• *Devises high-level strategy* for solving problem
  – Set trade-off priorities
• *Construct system architecture* by organizing into subsystems (system structuring)
  – Choose an approach for persistent data management (repository model)
  – Allocate components to processors and tasks (distribution model)
• Choose the implementation of control in software system (control modeling)
  – Identify concurrency inherent in the problem
  – Define access to global resources
• *Divide problem into implementable components* (modular decomposition)
Object Design

- Full definition of all the classes in the system
- Implementation alternatives evaluated and chosen
- Combine three models to obtain class operations
- Design algorithms to implement operations
- Optimize access paths to data
- Implement control for external interactions
- Adjust class structure to increase inheritance
- Design associations
- Determine object representation
- Package classes and associations into implementable modules
Detailed Design

• Detailed design is the process of completely specifying an architectural design such that module implementation can proceed (independently)

• Interface specifications
  – brief description of each module
  – attributes
    ☰ brief description and specify types
  – operations
    ☰ brief description
    ☰ list of parameters and parameter types
    ☰ return type (if applicable)
Detailed Design, continued

• Algorithm and data structure specification
  – the designer can give hints as to what algorithms or data structures might be most useful for a particular module
  – also, the client may have specified a particular algorithm or data structure that must be used
  – in addition, constraints in the requirements may require one approach over another
    ☹ for instance, implementing a data structure so that it uses the minimum amount of memory possible vs. keeping everything in memory for speed
Mapping design into code

• Most programming languages provide very similar sets of features
  – user-defined types
  – control structures
    ☒ if...then...else...
    ☒ while x do y
    ☒ for i = 1 to x
    ☒ etc
  – etc.

• This means that, in general, operations can be expressed in many different languages
Mapping design into code, continued

- Major differences between languages usually fall into these categories
  - compiled vs. interpreted
  - procedural vs. object-oriented
  - general purpose vs. application/domain specific
    - e.g. C++ vs. FileMaker Pro’s scripting language

- If a design takes advantage of, or depends on, one or more of these features then certain programming languages have to be excluded from implementation
Modularity Mechanisms

• One important feature of any programming language is how it can represent modules directly
  – C and C++ have separate header and body files
  – Java has package names and class files
  – Ada has a construct called a package with a specification and body (implementation)
  – etc.

• These features are important since it makes it easier to map the design into code and to trace a code module back to its design counterpart