Problem Statement
A manufacturer of custom bicycles wants a system that allows its customers to place and track orders via the web, its sales staff to manage the orders, its stock manager to manage available components, and its production staff to fill the orders. Bicycle frames are available in many sizes and styles. The customer can also choose the components for the bicycle, but since components may not be available when the order is finally filled the system needs to keep track of suitable replacements.

System Features
• Customer can place order
• Customer can track orders
• Sales can manage orders
  – Billing
  – Update status information
  – Shipping

System Features (2)
• Management of inventory of components and frame materials
• Identification of replacement components
• Production staff to fill orders

User Views of System
• Customer view: place & track orders
• Sales office: satisfying order, collecting $
• Stock person: managing inventory
• Production staff: building bicycle
Aside: Use-Case Diagram

Class Identification

“Simple” Objects
- Order
- Customer
- Bicycle
- Frame
- Component
- Order status

Container Objects
- Order queue
- Customer db
- Inventory
- Replacement map

Class Fields

- Order: customer, bicycle, status
- Customer: name, address, billing Info
- Bicycle: frame, components
- Frame: style, size, material, color
- Frame size: dimensions
- Component: part name, manufacturer, etc.

Class Relationships

Rethinking Relationships

- Does an Order really have a Customer?
- More like Customer has Order(s)
- Don’t want to have to maintain
  - multiple copies of order objects
  - pointers to order objects
- Use order numbers and customer ids to cross reference

Using Cross-Referencing

- Alternatives:
  - Customer stores list of order numbers
  - Order database is indexed by customer id and order number, Customer has no information about order
- Order stores customer id in both alternatives
- Note: Similar issue in Bicycle class
Rethinking Conclusion

- Lose direct relationship in diagram

<table>
<thead>
<tr>
<th>Order</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>order num</td>
<td>cust id</td>
</tr>
</tbody>
</table>

- Must have 2-way association between Customer and Order DBs

Order Class Form

- Fields: Order number, Bicycle, Customer, Status

  - Operations
    - Accessors: getOrderIdNum, getBicycle, getCurrent, getStatus
    - Mutators: setBicycle, setCurrent, setStatus

Order Class Declaration

```java
class Order {
    // methods will go here
    private:
        ord_num orderNo; // unique
        Bicycle bikeSpec; // only one, could be many
        cust_id cust; // one customer per order
        status_type currStatus;
        Order(); // no orders without order num bens
};
```

Order Class Declaration (2)

```java
class Order {
    public:
        Order(const ord_num&);
        Order(const ord_num&, const Bicycle&, const cust_id&);
        ord_num getOrderIdNum() const;
        Bicycle& getBicycle() const;
        cust_id getCurrent() const;
        status_type getStatus() const;
        void setBicycle(const Bicycle&);
        void setCurrent(const cust_id&);
        void setStatus(const status_type&);
    private:
        // fields here
};
```

Order Numbers

- Order class on previous slide shows how to set order numbers from outside.
- How could we set order numbers from “inside” class? (hint: static variables)
- What about persistence? (keeping values between executions)

User Interface for Orders

<table>
<thead>
<tr>
<th>Items</th>
<th>Design Bike</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total:</th>
<th>Clear Items</th>
</tr>
</thead>
</table>
User Interface for Bike Design

<table>
<thead>
<tr>
<th>Geometry &amp; Style:</th>
<th>Material:</th>
<th>Color:</th>
<th>Components:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choose</td>
<td>Choose</td>
<td>Choose</td>
<td>Standard group</td>
</tr>
</tbody>
</table>

Total Cost: $0.00

User Interface Classes

- Order UI
- Bike Design UI
- Component Selector UI
- Frame Selector UI

Class Relationships (2)

- Inventory
- Order DB
- Production Queue
- Substitution List
- Customer DB
- Order

What Happened to the Names?

- Where did the old names go?
- As designer we can change our mind about how things are structured or named before final document
- Must be careful to document changes, and/or update documents.

Inventory Class Form

- Fields: collection of components
- Methods
  - Accessors
    - listCompByType(type) – return list
    - search(part_number), count(part_number)
  - Mutators
    - add(component), remove(part_number)
    - reserve(part_number) – return false if cannot
Inventory Class Implementation

class Inventory{
public:
    Inventory();
    void addItem(const Component&);
    void removeItem(const part_numbers&);
    bool reserve(part_numbers);
    Component list(const Component&);
    bool search(const part_numbers);
    int count(const part_numbers);
private:
    // data structure for Inventory
};

Substitution List

Substitution List Class Form

• Fields: collection of substitutions, Inventory association
• Methods
  – Accessors
    • findSubstitute(part_number)
  – Mutators
    • Add(substitution)
    • Delete(substitution)
    • removeInvalid() – remove and return list of invalid substitutions

Substitution List Class Impl.

class SubstitutionList {
public:
   SubstitutionList(Inventory&); // create empty list
   part_numbers findSubstitute(const part_numbers&);
   List<Substitution> removeInvalid(); // need better return type!
   add(const Substitution&);
   delete(const Substitution&); // data structure for collection of substitutions
   Inventory* inv;
};

Constructor for Subst List

SubstitutionList(Inventory& i) : inv(i) {
    // initializations for data structure
}

Order DB Class Form

• Field: Collection of Orders
• Methods:
  – Accessors
    • find(OrderNum), find(Customer)
  – Mutators
    • remove(OrderNum), add(Order)
    • getProductionOrders()
  – Static associations
Order DB Implementation

```cpp
class OrderTable {
public:
    OrderTable(Inventory*, ProductionQueue*, CustomerDbl); // order db
    Order& find(const OrderNum& const); // find an order in db
    Order* getFirstOrder(); // return first order
    void add(const Order&); // add an order to db
    void remove(const OrderNum&); // remove an order from db
private:
    // data structure for collection of order objects:
    Inventory* Inv; ProductionQueue* pqueue; CustomerDbl* custs;
};
```

Order DB Constructor

```cpp
OrderTable(Inventory& Inv, ProductionQueue& pqueue, CustomerDbl& cust)
```

Production Queue Class Form

- Field: queue of orders
- Mutator
  - Accessor:
    - listOrders(order_status)
    - getStatus(OrderNum)
  - Mutator:
    - Remove(OrderNum)
    - nextOrder() – get next available order
    - completeOrder(OrderNum)

Scenario: Adding Order

- Will consider:
  - Adding order to order db
  - Getting next order from production queue
  - Completing order in production queue

Scenarios

- Will consider:
  - Adding order to order db
  - Getting next order from production queue
  - Completing order in production queue

Production Queue Impl.

```cpp
class ProductionQueue {
public:
    ProductionQueue(); // empty queue
    void useOrderTable(OrderTable&); // association
    void listOrders(order_status); // order db
    void remove(const OrderNum&); // remove an order from db
    void completeOrder(const OrderNum&); // complete an order
private:
    Queue q;
    OrderTable* orderedb;
};
```
Adding an Order

Scenario: Get Next Order

- Logic:
  1. If there is an order not in production, return that order
  2. If there is not, get new orders from Order db, return first
  3. If there are none, return ?
- Found a problem: need to rethink return type, or methods

Get Next Order (nonempty)

Get Next Order (Empty Queue)

Next Order Problem
- What to do if there are no new orders in order db?
- Can’t keep Orders getNext() as is

Scenario: Completing Order
- Logic:
  1. Remove from queue
  2. Send back to Order DB as “completed”
- Issues:
  - How do we send back?
Completing Order

Other Implementation Issues

- What kind of data structures?
  - Inventory – find by part number
  - Substitution list – find by part number
  - Order db – find by order number and cust name
    (or maybe an id?)
  - Production queue – find first order not being worked on, find specific orders being worked on

Data Structures

- “Find” operations: use of index structures
  - Binary search trees, hashing, b-trees
  - Standard Template Library has indexed containers `map<>` and `multimap<>`
- “Find first” implies queue structure
- Can put off data structure details until later, but can recognize needs as go along
  Ex. Production queue really needs two data structures.

Notes

- Design is not a linear thought process
- Problems found during design are easier to fix than problems found during coding.

Moral: more time should be spent on design than coding