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 inventory staff to manage available components and materials, and its production staff to fill the orders. Bicycle frames can be made in any size and style from a variety of materials. 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
Class Identification

<table>
<thead>
<tr>
<th>“Simple” Objects</th>
<th>Container Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order</td>
<td>Order queue</td>
</tr>
<tr>
<td>Customer</td>
<td>Customer db</td>
</tr>
<tr>
<td>Bicycle</td>
<td>Inventory</td>
</tr>
<tr>
<td>Frame</td>
<td>Replacement map</td>
</tr>
<tr>
<td>Component</td>
<td>Billing history</td>
</tr>
<tr>
<td>Order status</td>
<td></td>
</tr>
</tbody>
</table>

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

Order Class Form

- Fields: Order number, Bicycle, Customer, Status
- Operations
  - Accessors: getOrderNum, getBicycle, getCustomer, getStatus
  - Mutators setBicycle, setCustomer, setStatus

Order Class Declaration

```java
class Order {
    // methods will go here
    private:
        int ord_num ordeon;
        Bicycle bike_spec;
        Customer cust;
        status_type curstatus;
        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 Customer&);
        const ord_num getOrderNum() const;
        const Bicycle getBicycle() const;
        const Customer getCustomer() const;
        status_type getStatus() const;
        void setBicycle(const Bicycle&);
        void setCustomer(const Customer&);
        void setStatus(const status_type&);
    private:
        // fields go 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: use feature from Topic 12
- What about persistence? (keeping values between executions)

User Interface for Orders

<table>
<thead>
<tr>
<th>Items</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Design Bike</td>
</tr>
<tr>
<td></td>
<td>Submit Order</td>
</tr>
<tr>
<td>Total:</td>
<td>X</td>
</tr>
<tr>
<td>Clear Items</td>
<td></td>
</tr>
</tbody>
</table>

User Interface for Bike Design

Design A Bike

<table>
<thead>
<tr>
<th>Geometry &amp; Style</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Material:</td>
<td></td>
</tr>
<tr>
<td>Color:</td>
<td></td>
</tr>
<tr>
<td>Components:</td>
<td></td>
</tr>
<tr>
<td>Total Cost</td>
<td>Price</td>
</tr>
</tbody>
</table>

| Clear | Add to Order |

User Interface Classes

Order UI

Bike Design UI

Component Selector UI

Geometry Selector UI

Style Selector UI

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

```
Inventory
  * Component
```

- **Part No**
- **Description**
- **Cost**

### 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

```cpp
class Inventory {
public:
  Inventory();
  void add(const Component&);
  void remove(const part_number&);
  bool reserve(const part_number&);
  list<CompByType(const Component)> const;
  bool search(const part_number&);
  int count(const part_number&);
private:
  // data structure for Inventory
};
```

### Substitution List

```
Substitution List
  * Substitution
```

- **Part No**

### 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.

```cpp
class SubstitutionList {
public:
  SubstitutionList(Inventory&); // create empty list
  int part_number findSubstitute(const part_number&); const;
  list<Substitution> removeInvalid(); // need better return type
  Substitution add(const Substitution&); // remove
  delete(const Substitution&);
private:
  // 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

class OrderTable {
  public:
    OrderTable(Inventory& i, ProductionQueue& p, CustomerDB& c); // empty constructor
    void useOrderTable(OrderTable& o); // association
    OrderList listOrders(const OrderNum&); const;
    Order& nextOrder(); const;
    remove(const OrderNum&);
  private:
    Inventory* inv; ProductionQueue* pi;
    // data structure for collection of order objects
    OrderList* offset; // two way association
    // code to initialize data structure goes here
};

Order DB Constructor

OrderTable(Inventory& i, ProductionQueue& p, CustomerDB& c); // empty constructor

  { // code to initialize data structure goes here
  }

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)

Production Queue Impl.

class ProductionQueue {
  public:
    ProductionQueue(); // empty queue
    void useOrderTable(OrderTable& o); // association
    OrderList listOrders(const OrderNum&); const;
    void remove(const Order Num&);
    const Order& nextOrder();
    void completeOrder(const OrderNum&);
  private:
    Queue q;
    OrderTable* orderDB;
};
Scenarios

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

Scenario: Adding Order

Before adding check that order can be filled and that customer is in database:
1. Check components available
   1a. If not, find substitute
2. Check customer in database

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

Next Order Problem

What to do if there are no new orders in order db?
- Can’t keep Order & 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