Code Samples Showing Principles

(Violation of the principles…)

not a QUIZ!

```
ofstream outfile ("data.txt");
void output_employee(Employee* e, bool to_screen) {
      if (e->type() == SALARIED){
            if (to_screen)
                  cout << e.name() << ", salary:" << e.salary() << endl;
            else
                  outfile << e.name() << ", salary:" << e.salary() << endl;
      } else if (e->type() == SALES){
            if (to_screen)
                  cout << e.name() << ", commission: " << e.comm() << endl;
            else
                  outfile << e.name() << ", commission: " << e.comm() <<
endl;
      }
}
```

What (at least) 2 principles are violated?

What are the bad consequences?

How would you fix it? Explain, and show some code

You have the Employee hierarchy we have looked at: Four subtypes of Employees: Hourly, Sales, Salaried, Part Time.

```
// Read employee records from a stream (file)
void import_records(EmployeeList& employees, ifstream& ifs)
{
       int e_type;
       Employee* e_temp;
       while (ifs >> etype) {
              // Read common data such as name, employee number…
              switch (etype){
                     case HOURLY: // read data, create an Hourly Employee in e_temp
                            break;
                     case SALES: // read data, create Sales Employee in e_temp
                            break;
                     case SALARIED: // read data, create Salaried Employee in e_temp
                            break;
                     case PARTTIME: // read data, create Part Time Employee in e_temp
                            break;
                     default:
                     // Assume proper error handling code.
              }
              employees.add(e_temp);
       }
}
…
void add_new_hourly(EmployeeList& employees)
{
       Employee* e_temp;
       cout << "Enter name: ";
       string n; cin >> n;        // Assume you get other employee data
       // Ask for information to decide what kind of employee it is, then
       // create it and get specialized information.
       // And so on…
       // Eventually:
       employees.add(e_temp);
}
void add_new_salaried (EmployeeList& employees) { // similar to above }
```

What principles are violated, besides DRY?

What are the bad consequences?

How would you fix it? Explain, and show some code

This is part of an e-commerce program. This is part of the checkout process.

```
// Checkout: total up the sale
void checkout_total(Account& a, Cart& cart)
{
	float total = 0;
	CartItemIterator ci;		// Cart Iterator follows C++ conventions
	for (ci = cart.begin(); ci != cart.end(); ci++) {
		CartItem item = *ci;	// (gets what the iterator points to)
		total += item.price();
	}

	// Calculate Shipping
	// Account type influences shipping cost
	AccountType at = a.atype();
	ShippingRule srule = at.shiprule();
	total += srule.calc_shipping(total);	// calculate shipping, add to total

	// etc…
}
```

What principle is violated? Explain the principle.

What are the bad consequences in this code?

How would you fix it? Explain, and show some code

The same program, with one small change. Now, the price of an item may be discounted for certain promotions, which are associated with the account.

```
// Checkout: total up the sale
void checkout_total(Account& a, Cart& cart)
{
	float total = 0;
	Promotion promo = a.getPromo();
	CartItemIterator ci;		// Cart Iterator follows C++ conventions
	for (ci = cart.begin(); ci != cart.end(); ci++) {
		CartItem item = *ci;	// (gets what the iterator points to
		total += item.price(promo);	// Get price, which might be discounted
	}

	// Calculate Shipping
	// Account type influences shipping cost
	AccountType at = a.atype();
	ShippingRule srule = at.shiprule();
	total += srule.calc_shipping(total);	// calculate shipping, add to total

	// etc…
}
```

You see that the promotion object is passed to each item. Do you like this? Why or why not?

Does it still violate principles? Which ones?

How would you fix it? Explain, and show some code

Suppose you have the checkout function (or method) as described above. (Yes, it has been cleaned up.) Now you need to implement an express checkout. It does the same thing, but the user doesn't have to input certain information because we have it stored for them. You also have to make some changes for international checkout – different countries have different address formats, and may have different laws for credit cards, etc.

1. As you design the new functionality, what principles should you follow? (All of them, of course! But one principle stands out as particularly relevant.)

2. What different designs possibilities can you think of? Consider which patterns might apply.

3. What are the tradeoffs (relative advantages and disadvantages) among the designs you come up with?

4. What might be important design considerations?