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?

DRY

Separate things that change from things that stay the same.

DIP: output_employee() depends on something concrete; the Employee type field.

Note that if you ever have a "type()" method in a hierarchy, that is bad practice and should usually be removed.

What are the bad consequences?

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

Not a strong case for a pattern, though a strategy might work. But easiest is just use inheritance, and pass an outstream.

Just pass in the appropriate ostream, or just set the current ostream once at the start of the function.

If this were a method of Employee, it would be something akin to a Template Method

```
void output_employee(Employee* e, bool to_screen) {
      ofstream& of = to_screen ? cout : outfile;
      of << e.name() << e.pay() << endl;
      }
```

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?

RAII, Separate that which changes

What are the bad consequences?

Wow! What happens when you add a new type of Employee? And so on…

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

Create a factory. Maybe you could use a Factory Method, with one for interactive and one for file. And then use a Template Method within it.

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.

LoD (Don't talk to strangers) Two places: The iteration and the calculate shipping.

SRP is also violated.

What are the bad consequences in this code?

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

Could use a façade, or just simply redo the interface of Cart to return the total. Use Internal Iteration.

Hide the Account type and shipping rule inside of account.

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?

The items have to know about promotions. That seems like some nasty coupling. Does each type of item have to handle each type of promotion? On the other hand, maybe you have the flexibility to discount certain items. But overall, I think it's way too coupled for me.

Does it still violate principles? Which ones?

LoD – interesting: it moves it around. The requirements have introduced another dependency.

Loosly coupled designs, Separate things that change

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

No particular patterns. I would just apply a promotion discount to the entire price, once it is calculated. But that presupposes that discounts apply to a whole order, not piecemeal.

Another idea is to put Promotion entirely inside the Cart, and let it worry about it. That might be the best option.

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.)

Separate things that change from things that stay the same. The Open-Closed Principle also kind of applies. And keep coupling low. Of course, you will also want to watch out for DRY, SRP, and as you design, you will want to consider DIP (see below.)

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

I would use inheritance to handle different types of checkout. I might have a checkout hierarchy with normal and express, and use a Template Method in it. There may also be a subtype for international.

Another possibility is to create hierarchies for address, shipping rules, credit cards, etc., and using them as a bunch of Strategies.

I considered using Decorators instead, but I don't think it will work well.

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

Checkout subtypes and Template Method: clean, but the checkout class could get kind of big, and I would begin to worry about the SRP.

The forest of Strategies gives a lot of flexibility, but it may go too far, and fragment the design concepts – taking the SRP principle too far.

My first choice is the Template Method approach.

4. What might be important design considerations?

Some are: types of changes expected, reuse in other parts of the system, ease of understanding the design.