Tutorial 6

Question 1

Pattern	Description
Strategy	a. I allow an object to change its behaviour when it's state changes
State	b. I implement encapsulation by inheritance and allow subclasses decide how to implement steps in an algorithm
Template Method	c. I enable interchangeable behaviours to be encapsulation and request the client (context class) to use delegation to decide which behaviour to use

Question 3

Design by Contract

- Motivation: to make code and systems safer, more maintainable, and reduce the necessity of error checking (especially in large systems), using smart and robust design
- Different style of programming to first year courses (defensive programming)

Defensive Programming/Design

- Every (applicable) component of a system checks for errors, invalid input / parameters, or inconsistent state
- Ensures system does not behave unexpectedly under unforeseen circumstances

- 1511 / 2521 style - place **assert** everywhere
- Fine for small programs

Disadvantages of Defensive Programming

- Doesn’t work well for large, complex systems
- You usually don’t want the entire system to go down in the case of one component receiving invalid input
- Large codebase makes it more likely someone will forget to include error tests
- Large amount of error checking can have a significant performance impact
- **Doesn’t actually prevent errors - just reacts to them**

Design by Contract

- Central Idea: formal **contract** is designed and enforced between **clients** (callers) and **providers** (callees)
- Essentially - methods have a **precondition** which must be met before they can be called.
- Responsibility of caller (client) to meet this precondition
- Provider provides a guaranteed **postcondition** which they fulfill by the end of the method call, assuming the precondition was met
- If precondition not met - no guarantees

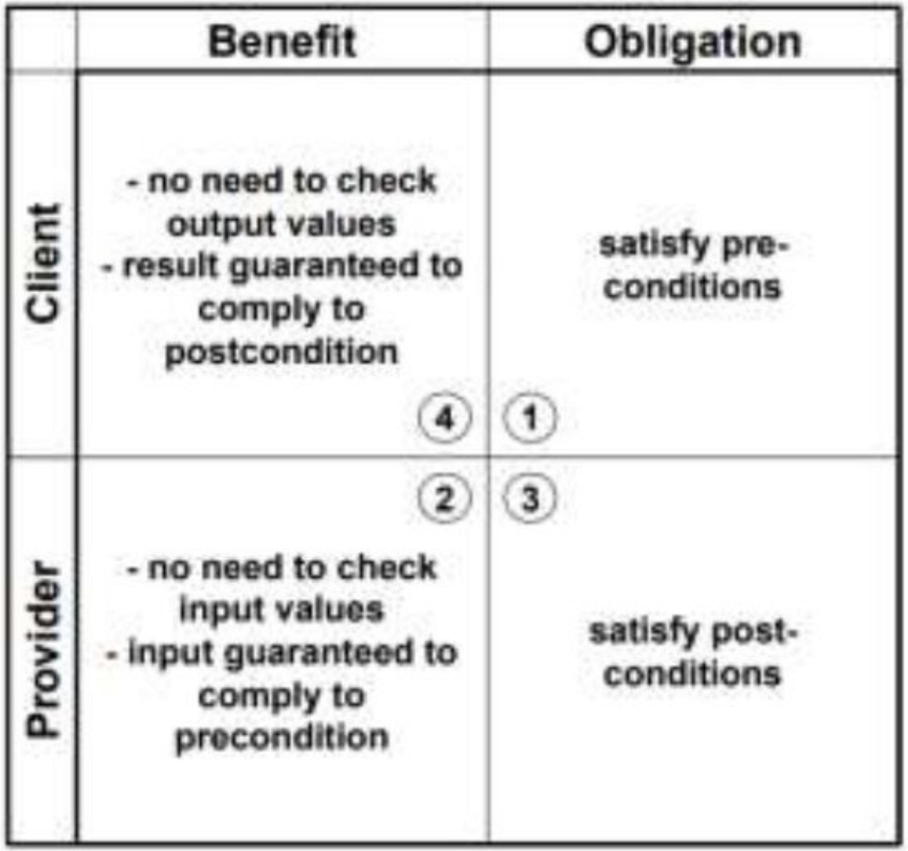
Contract Example

Why DbC?

- Specifying a formal interface between different components makes complex systems easier to understand and more maintainable
- Less likely for programmers to make a mistake when using other components
- Removes / Reduces redundant error checking

- *Critical Components can still be defensively programmed* (places in which someone breaking the contract would cause catastrophe)

Benefits - Obligations



	Benefit	Obligation
Client	- no need to check output values - result guaranteed to comply to postcondition ④	satisfy pre-conditions ①
Provider	② - no need to check input values - input guaranteed to comply to precondition	③ satisfy post-conditions

Class Invariants

- Conditions on class state (fields, etc) which are preserved between different method calls
- Example: **Square** class invariant: **width = height >(=) 0**
- Make it easier to reason about classes and their fields

- Class invariants do not need to be preserved during a methods execution, but must hold before and after (assuming the methods preconditions were met)

LSP & DbC

- In order to satisfy the Liskov Substitution Principle:
- Subclasses cannot make a precondition for a method stronger

- Parent: *@precondition* **amount >= 0**
- Child: *@precondition* **amount >= 10**
- Cannot substitute parent with child when **0 <= amount < 10**

LSP & DbC

- Similarly, post conditions cannot be weaker

- Parent: *@postcondition* return value is an even integer
- Child: *@postcondition* return value is an integer
- Code that calls method assumes return value is even will not work if parent replaced by child

Question 4 - Bank Account

Requirements - Bank Account

- Each bank account should have a current balance and methods implementing deposits and withdrawals
- Money can only be withdrawn from an account if there are sufficient funds
- Each account has a withdrawal limit of $800 per day

Requirements - Internet Account

- In addition to the constraints on BankAccount, there is a limit of 10 Internet payments per month
- Note that Internet payments count as withdrawals, so are subject to the daily limit on withdrawals