**1. Поведенческие шаблоны проектирования**

Поведенческие шаблоны описывают способы взаимодействия объектов и распределения между ними обязанностей. Рассмотрим каждый шаблон как «проблема → решение» через призму инкапсуляции, принципа «разделяй и властвуй» и ортогональных стратегий, а также кратко оценим влияние многопоточности.

**1.1 Шаблон «Состояние» (State)**

**Определение.**  
Шаблон позволяет объекту менять своё поведение при изменении внутреннего состояния, словно класс объекта меняется во время выполнения.

* **Проблема:** в клиентском коде накапливаются условные операторы if/else или switch, зависящие от поля «текущее состояние».
* **Решение:** инкапсулировать каждое состояние в отдельный класс-реализатор интерфейса State и делегировать ему обработку.

**Примеры использования:**

1. **Игровой персонаж:** разные режимы (бег, прыжок, приседание). Каждый режим реализован как отдельное состояние с методами move(), attack().
2. **Сетевое соединение:** состояния Connecting, Connected, Disconnected, каждое обрабатывает запросы по-своему.
3. **Реактивная форма ввода:** фокус/нефокус, enabled/disabled/readonly состояния поля.

**Принципы:**

* **Инкапсуляция:** логика каждого состояния скрыта в отдельном классе.
* **Divide & Conquer:** разделили сложную машину состояний на независимые модули.
* **Ортогональность:** новые состояния добавляются без правки существующих.

**Многопоточность:**  
При разделении состояний на объекты нужно обеспечить атомарность переключения (volatile-поле или synchronized). В противном случае два потока могут одновременно изменить состояние, и клиент получит неконсистентное поведение.

**1.2 Шаблон «Посредник» (Mediator)**

**Определение.**  
Посредник централизует взаимодействие множества объектов, убирая прямые связи между ними.

* **Проблема:** сеть взаимосвязанных объектов с кросс-ссылками становится сложной и труднораспространяемой.
* **Решение:** ввести объект «посредник», через который все объекты обмениваются сообщениями.

**Примеры использования:**

1. **Чат-сервер:** каждый клиент шлёт сообщение на сервер-посредник, который рассылает его всем остальным.
2. **GUI-фреймворки (Qt):** класс QDialog как посредник между виджетами (кнопки, поля ввода) управляет их взаимодействием, реагируя на сигналы/слоты.
3. **Управление логикой торговой площадки:** брокер-посредник обрабатывает заявки продавцов и покупателей, сопоставляет ордера.

**Принципы:**

* **Инкапсуляция:** взаимодействие объектов скрыто в классе посредника.
* **Divide & Conquer:** сложные связи превращаются в тривиальные «клиент–посредник–другой клиент».
* **Ортогональность:** клиенты не знают о существовании друг друга.

**Многопоточность:**  
Посредник становится точкой синхронизации: нужно защищать его методы, иначе сообщения могут «пересечься» или быть потеряны. Часто используют очередь сообщений и отдельный рабочий поток для обработки.

**1.3 Шаблон «Наблюдатель» (Observer)**

**Определение.**  
Позволяет одним объектам (наблюдателям) подписываться на события другого объекта (субъекта) и автоматически получать уведомления об изменениях.

* **Проблема:** жесткая связь «издатель → подписчик», когда изменение одного объекта должно приводить к реакциям многих других.
* **Решение:** субъект хранит список наблюдателей и рассылает им уведомления при изменениях.

**Примеры использования:**

1. **Qt-слоты и сигналы:** любой QObject может эмитить сигнал (например, valueChanged(int)), а другие объекты подключаться к этому сигналу через connect().
2. **UI-фреймворк в MVC:** модель уведомляет представление об изменении данных.
3. **Система логирования:** разные листенеры (файл, консоль, сеть) подписываются на генерацию логов.

**Принципы:**

* **Инкапсуляция:** субъект не знает, что делают наблюдатели, он лишь рассылает уведомления.
* **Divide & Conquer:** единичная точка публикации событий вместо множества вызовов.
* **Ортогональность:** легко добавлять/удалять подписчиков без изменения субъекта.

**Многопоточность:**  
Если событие генерится в одном потоке, а обработчики работают в других, нужна безопасная очередь уведомлений или механизмы пост-dispatch (например, QMetaObject::invokeMethod(..., Qt::QueuedConnection)), чтобы избежать гонок и deadlock’ов.

**2. Что такое дизайн ПО и влияние многопоточности на дизайн**

**Определение “Дизайн ПО”.**  
Дизайн программного обеспечения (software design) — это процесс преобразования функциональных и нефункциональных требований в спецификации архитектурных и детальных решений, описывающих структуры данных, алгоритмы, модули и взаимодействие компонентов на уровне кода. Дизайн является связующим звеном между требованиями (что система должна делать) и реализацией (как она это делает).

**Ключевые аспекты дизайна:**

1. **Модулярность:** разбивка системы на независимые модули с чётко определёнными интерфейсами.
2. **Абстракция и инкапсуляция:** сокрытие деталей реализации за понятными API.
3. **Повторное использование:** использование шаблонов проектирования, библиотек, фреймворков.
4. **Соответствие принципам SOLID:** обеспечение расширяемости, устойчивости к изменениям.

**Влияние многопоточности на дизайн:**

* **Уровень API и модулей:** нужно проектировать thread-safe интерфейсы или явно документировать, что класс не является потокобезопасным.
* **Разделение ответственности:** вводить отдельный слой работы с потоками (task-менеджер, пул потоков), а не разбрасывать синхронизацию по всему коду.
* **Синхронизация и дедлоки:** дизайн должен минимизировать количество локов и их пересечение, по возможности использовать неблокирующие структуры (lock-free), каналы или события.
* **Иммутабельность:** по возможности делать объекты неизменяемыми, чтобы упростить работу с ними из разных потоков.
* **Производительность vs Простота:** в дизайне многопоточности всегда компромисс между скоростью и читаемостью/отлаживаемостью кода. Часто вводят уровни (многопоточность внутри модуля, а между модулями — только асинхронные сообщения).

**Итого**, поведенческие паттерны помогают организовать взаимодействие объектов, применяя инкапсуляцию, принцип «разделяй и властвуй» и ортогональные стратегии. Многопоточность требует в дизайне чёткого разделения ответственности за синхронизацию, использования thread-safe механизмов и акцентирования на иммутабельности, чтобы сохранить простоту и надёжность системы.