Математические структуры и моделирование
2011, вып. 23, с. 86–98

УДК 004.4

ПРОЕКТИРОВАНИЕ ПОШАГОВЫХ ОНЛАЙН-ИГР

Е.В. Непомнящих

Проводится анализ требований, предъявляемых к современным пошаговым онлайн-играм, и наиболее разумных путей их разработки. Приводится универсальный шаблон базы данных, протокола клиент-серверного взаимодействия и иерархии классов на стороне сервера.

Введение

Современная индустрия информационных технологий (ИТ) основана на поиске и использовании универсальных шаблонов проектирования и разработки программных приложений. Данная статья является кратким обобщением наилучших практик, используемых при разработке пошаговых онлайн-игр, с кратким анализом и личным опытом автора.

Пошаговая игра — это жанр игр, основной особенностью которого является то, что игроки совершают ходы по очереди. К пошаговым играм относятся:

– пошаговые стратегии;

– карточные игры;

– настольные игры (шахматы, го, монополия и др.).

Замечание 1. Пошаговые игры накладывают меньше ограничений на сложность протокола взаимодействия по сравнению с играми в реальном времени. А именно, время реакции на то или иное событие, а значит, и качество сети ключевой роли не играют. Игроку обычно отводится от 10 секунд времени на принятие решения.

В большинстве пошаговых игр в любой момент времени решение принимает ровно один игрок. Следовательно, сужается множество запросов, на которые сервер должен адекватно реагировать.

Поэтому при создании протокола следует ориентироваться прежде всего на простоту его реализации и поддержки. Это позволит разработчику извлечь большую прибыль за меньшее время.

Омский государственный университет им. Ф.М. Достоевского
E-mail: enepomnyaschih@gmail.com

1. Предварительный анализ требований

Ниже перечислены требования, предъявляемые к большинству современных браузерных онлайн-игр:

1. Пользователь заходит на сайт и видит список доступных столов.
2. Пользователь может «сесть» за любой стол, если там есть свободные места.
3. При желании, пользователь может играть за несколькими столами одновременно.
4. Игра начинается, как только за столом не остаётся свободных мест.
5. Если пользователь встаёт из-за стола во время игры, его место занимает искусственный интеллект (не самый умный), пока пользователь не вернётся.
6. Время на раздумье ограничено (если долго думает, значит покинул игру).
7. Игра за любым столом ограничена по количеству ходов.

Такие требования наиболее хорошо подходят для игры на очки или деньги. Требования №1 – №4 представляют собой лучшие традиции пошаговых онлайн-игр. Требование №5 мотивирует игрока закончить игру, даже если он проигрывает. Вместе с требованиями №6 и №7 оно гарантирует, что всякая игра ограничена по времени. Пример игры, удовлетворяющей поставленным требованиям — KDice (`http://kdice.com`).

При проектировании базы данных и протокола клиент-серверного взаимодействия будем опираться на перечисленные требования.

1.1. Умный или глупый клиент?

Безусловно, сервер должен полностью владеть логикой приложения (правилами игры), чтобы предупредить его потенциальный взлом. Но стоит ли обучать бизнес-логике клиента?

Это напрямую зависит от того, какой размер имеет полный объем данных о состоянии игры. Если объем данных велик, долго собирается на сервере и передаётся клиенту, то имеет смысл часть логики реализовать на клиенте, чтобы разгрузить сервер. Например, в популярной игре Civilization IV датчик используемой памяти всегда зашкаливает. Возможно ли создать нечто подобное, оставив на клиенте исключительно пользовательский интерфейс?

С другой стороны, чем умнее клиент, тем дороже обойдётся разработка игры.

Замечание 2. От «эрудиции» клиента время разработки сервера никак не зависит. Если пользователь захочет перезагрузить окно браузера, то серверу придётся собирать и компоновать все данные об игре для передачи их клиенту.

Умный клиент может ускорить работу приложения, но он всегда потребует дополнительных ресурсов для своей разработки.

Следующий тест поможет сделать выбор:

Позволяет ли объем канала? Оценивается средний вес полного объема данных о состоянии игры. Далее, умножается на среднее количество запросов к серверу в секунду. Если полученное число превысит объем исходящего канала передачи данных, то глупый клиент недопустим. Если это число превысит 20% исходящего канала, то стоит призадуматься, потянет ли?

Велика ли трудоёмкость? Оценивается трудоёмкость алгоритма сбора данных об игре (в долях секунды). Учитываются все запросы к базе данных. Далее, умножается на среднее количество запросов серверу в секунду. Если время превысит одну секунду, то глупый клиент недопустим. Если это число превысит 200 мс, то стоит призадуматься, потянет ли?

Если объем канала позволяет и трудоёмкость сбора данных невелика, то рекомендуется использовать глупый клиент ввиду его невысокой стоимости.

1.2. Однонаправленный или двунаправленный протокол?

Самый обычный протокол HTTP-взаимодействия между клиентом и сервером (**первый вариант**) предполагает однонаправленные запросы от клиента к серверу. Клиент посылает запрос — сервер на него отвечает. Поскольку состояние игры периодически меняется по причине хода одного из игроков, запросы приходится посылать довольно часто с заданной периодичностью.

Тем не менее возможен и **второй вариант**: клиент и сервер посылают друг другу запросы обоюдно. Клиент посылает запрос, если пользователь кликнул мышкой. Сервер оповещает об этом остальных клиентов. Обычно клиент спит и ждёт сообщений от сервера.

Замечание 3. Запрос полного состояния игры все равно необходимо реализовать в том случае, если требуется позволять участнику продолжать игру даже после разрыва соединения.

Есть ещё и **третий вариант**. Все клиенты и сервер посылают запросы во всех направлениях. Этот вариант хорошо подошёл бы для игры в реальном времени, поскольку он очень быстрый. Но для реализации пошаговой игры он подходит не очень хорошо, потому что нас скорость работы протокола не сильно интересует, зато он самый сложный.

Следующий тест поможет выбрать путь реализации. Если все ответы «да», то следует использовать обоюдное общение клиента и сервера:

1. Ваша технология позволяет использовать двунаправленные запросы (например, в JavaScript на момент выпуска данной статьи это невозможно).

2. Вы владеете технологией достаточно хорошо (иначе затраты на изучение могут оказаться неоправданно велики).

3. У Вас есть запас времени для того, чтобы реализовать рассылку сообщений клиентам.

1.3. Структура базы данных

1.3.1. Игроки (Players)

1. Идентификатор игрока (playerId) — первичный ключ
2. Имя игрока (playerName)
3. Информация «о себе»

1.3.2. Сессии (Sessions)

1. Идентификатор сессии (ticket) — первичный ключ
2. Идентификатор игрока (playerId) — внешний ключ

1.3.3. Столы (Tables)

1. Идентификатор стола (tableId) — первичный ключ
2. Данные о текущем состоянии игры

1.3.4. Места за столами (Slots)

1. Идентификатор места (slotId) — первичный ключ
2. Идентификатор стола (tableId) — внешний ключ, альтернативный ключ
3. Идентификатор игрока (playerId) — внешний ключ, альтернативный ключ
4. Номер места (slotIndex)

1.3.5. Игровые события (Events)

1. Идентификатор события (eventId) — первичный ключ
2. Идентификатор места (slotId) — внешний ключ
3. Порядковый номер события за столом (eventIndex)
4. Тип события (eventType)
5. Описание события (eventData) — произвольные сериализованные данные

2. Серверный интерфейс приложения (API)

2.1. Общие положения

В данной главе приводятся лишь те методы API, которые необходимы для реализации любой пошаговой игры, удовлетворяющей поставленным ранее требованиям.

Список методов:

1. **GetTables** — получение списка игровых столов.
2. **JoinTable** — подключение игрока к столу.
3. **GetTable** — получение полной информации о столе, включая текущее состояние игры.
4. **PlayerTurn** — набор запросов о действиях игрока.
5. **WakeUp** — вернуть управление игроку, если его место занял искусственный интеллект.

Кроме того, понадобится реализовать один из наборов:

	Умный клиент	Глупый клиент
Односторонние запросы	GetEvents	GetUpdate
Двусторонние запросы	NotifyEvents	NotifyUpdate

,

где:

1. **GetEvents** — получение списка произошедших событий
2. **GetUpdate** — получение обновленного состояния игры
3. **NotifyEvents** — оповещение о произошедших событиях
4. **NotifyUpdate** — оповещение об обновлении состояния игры

Большинство запросов требует авторизации (в заголовках `Cookie` должен быть прописан действующий идентификатор сессии). Если требуется дать возможность участникам играть «на нескольких досках одновременно», то почти все запросы должны принимать обязательный параметр `tableId`. Иначе игра должна определяться по идентификатору сессии игрока, что не позволит участникам подключаться к нескольким играм одновременно.

Если клиент умный, т.е. он владеет игровой бизнес-логикой, то списка событий ему достаточно, чтобы самостоятельно построить новое состояние игры. Если клиент глупый, то ему необходимо передавать полную информацию о текущем состоянии игры, а также список возможных действий, которые игрок может выполнить на данном этапе игры. Сервер без труда может передавать полную информацию, но объем передаваемых данных из-за этого несколько увеличивается. Даже если клиент глупый, ему все равно нужен список событий, чтобы показать их на экране со всей соответствующей анимацией.

2.2. Общий формат всех запросов

Некоторые веб-технологии (например, Adobe Flash на момент выпуска данной статьи) не дают возможность читать ответы с HTTP-кодом, отличным от `200 OK`. Поэтому все запросы должны возвращать ответы именно с этим HTTP-кодом.

В связи с этим роль HTTP-кода должно выполнять содержимое тела ответа. Например, можно использовать следующий универсальный формат ответа (JSON-версия):

```
{
    success : true или false,
    error   : null или { // пример ошибки
        code    : "ERROR_AUTHORIZATION",
        message : "You must login to call this action"
    },
    result  : произвольные данные
}
```

2.3. GetTables — получение списка столов

`GET /application/tables`, публичный доступ.
Возвращает список доступных столов для подключения.
По желанию, можно передавать параметры фильтрации, сортировки и пр.
Ответ:

```
[
    {
        tableId   : 10,
        tableName : "My favorite table",
        ... // другие краткие данные
    },
    ...
]
```

Возможные ошибки: нет

2.4. JoinTable — подключение игрока к столу

`GET /table/join`, авторизованный доступ.
Подключает игрока к столу.
Параметры:

1. `tableId` — обязательный целочисленный параметр, идентификатор стола

Ответ: `null`
Возможные ошибки:

1. `ERROR_AUTHORIZATION` — требуется авторизация
2. `ERROR_PARAM_INVALID` — обязательный параметр не указан или параметр принимает неправильное значение
3. `ERROR_NO_SLOT` — все места заняты
4. `ERROR_JOINED_ALREADY` — игрок уже сидит за этим столом

2.5. GetTable — получение полной информации о столе

`GET /table/info`, авторизованный доступ.
Возвращает полную информацию о столе, включая текущее состояние игры.
Параметры:

1. `tableId` — обязательный целочисленный параметр, идентификатор стола

Ответ:

```
{
    gameInfo: {
        players: [
            {
                playerId   : Int,
                playerName : String,
                ...
            },
            ...
        ],
        ...
    },
    ... // см. GetUpdate
}
```

Возможные ошибки:

1. `ERROR_AUTHORIZATION` — требуется авторизация
2. `ERROR_NOT_JOINED` — игрок не сидит за столом
3. `ERROR_PARAM_INVALID` — обязательный параметр не указан или параметр принимает неправильное значение

2.6. PlayerTurn — набор запросов о действиях игрока

`GET /table/turn, /table/pass, /table/double` и др., авторизованный доступ.
Выполняет ход или другое действие игрока.
Параметры:

1. `tableId` — обязательный целочисленный параметр, идентификатор стола
2. Другие параметры, зависящие от типа действия. Например, идентификатор карты в карточной игре

Ответ: `null` или данные о произошедшем событии (запись из таблицы Events).
Возможные ошибки:

1. `ERROR_AUTHORIZATION` — требуется авторизация
2. `ERROR_NOT_JOINED` — игрок не сидит за столом
3. `ERROR_PARAM_INVALID` — обязательный параметр не указан или параметр принимает неправильное значение
4. `ERROR_GAME_STATE` — данное действие запрещено на данном этапе игры. Возможно, игра уже закончилась

5. ERROR_CURRENT_PLAYER — сейчас ходит другой игрок
6. Другие ошибки, зависящие от действия. Например, нельзя дважды удваивать ставку

2.7. WakeUp — вернуть управление от искусственного интеллекта

`GET /table/wakeup`, авторизованный доступ.
Возвращает управление игроку, если его место занимает искусственный интеллект.
Параметры:
1. `tableId` — обязательный целочисленный параметр, идентификатор стола

Ответ: `null`
Возможные ошибки:
1. ERROR_AUTHORIZATION — требуется авторизация
2. ERROR_NOT_JOINED — игрок не сидит за столом
3. ERROR_PARAM_INVALID — обязательный параметр не указан или параметр принимает неправильное значение
4. ERROR_NOT_SLEEPING — искусственный интеллект не включён

2.8. GetEvents — получение списка произошедших событий

`GET /table/events`, авторизованный доступ.
Возвращает список произошедших событий, начиная с указанного порядкового номера.
Параметры:
1. `tableId` — обязательный целочисленный параметр, идентификатор стола
2. `fromIndex` — необязательный целочисленный параметр, номер события, начиная с которого вернуть события

Ответ:

```
[
    {
        eventIndex  : Int,
        eventType   : String,
        tableSlotId : Int,
        ... // другие данные
    },
    ...
]
```

Возможные ошибки:
1. ERROR_AUTHORIZATION — требуется авторизация
2. ERROR_NOT_JOINED — игрок не сидит за столом
3. ERROR_PARAM_INVALID — обязательный параметр не указан или параметр принимает неправильное значение

2.9. GetUpdate — получение обновлённого состояния игры

`GET /table/update`, авторизованный доступ.
Возвращает текущее состояние игры, список доступных действий игрока и список произошедших событий, начиная с указанного порядкового номера.
Параметры:

1. `tableId` — обязательный целочисленный параметр, идентификатор стола
2. `fromIndex` — необязательный целочисленный параметр, номер события, начиная с которого вернуть события

Ответ:

```
{
    gameState: {
        stateId: String, // текущий этап игры
        currentTableSlotId: Int, // текущий игрок
        ... // другие данные
    },

    gameDecision: {
        sleep: true/false, // включен ли иск. интеллект
        ... // данные о допустимых действиях игрока
    },

    gameEvents: ... // см. GetEvents
}
```

Возможные ошибки:

1. `ERROR_AUTHORIZATION` — требуется авторизация
2. `ERROR_NOT_JOINED` — игрок не сидит за столом
3. `ERROR_PARAM_INVALID` — обязательный параметр не указан или параметр принимает неправильное значение

2.10. NotifyEvents — оповещение о произошедших событиях

POST-запрос осуществляется от сервера к клиенту.
Оповещает о произошедшем событии.
Параметры равны свойствам произошедшего события (см. GetEvents).
Ответ: `null`

2.11. NotifyUpdate — оповещение об обновлении состояния игры

POST-запрос осуществляется от сервера к клиенту.
Передает новое состояние игры.
Параметры равны данным о новом состоянии игры (см. GetUpdate).
Ответ: `null`

3. Методика реализации серверной части

В данной главе пойдёт речь о том, как лучше построить процесс разработки для того, чтобы малыми усилиями, но достаточно качественно реализовать приложение пошаговой онлайн-игры. В параграфах описываются последовательные шаги, с которых следует начинать разработку. Следуя этим шагам, разработчику удастся в кратчайшие сроки подготовить предварительную версию сервера приложения для интеграции с клиентской частью.

3.1. Выделить этапы игры

Прежде всего следует определить, на какие этапы можно разбить игровой процесс. Этап игры — это некоторое состояние приложения, определяющее алгоритмы обработки запросов для конкретного игрового стола (`tableId`). Выделение этапов игры позволит разработчику без труда построить корректную и удобную иерархию классов на стороне сервера. Пример для популярной игры «Червы»:

1. Ожидание игроков
2. Сброс трех карт
3. Розыгрыш всех карт
4. Подведение итогов партии и возможный переход к следующей партии
5. Подведение итогов игры, распределение выигрыша

В результате должен получиться детерминированный автомат переходов между этапами игры. Переходы между ячейками автомата осуществляются при действиях игроков и имеют четкие условия. При проектировании сервера следует в первую очередь нарисовать этот автомат. Чем больше ячеек, тем прозрачнее становится принцип работы автомата, и тем проще становятся условия перехода между ячейками.

В приведённом примере имеются следующие переходы: 1 — 2, 2 — 3, 3 — 4, 4 — 2, 4 — 5. Начальное состояние — 1, конечное состояние — 5.

3.2. Создать классы этапов игры

Далее, необходимо создать иерархию классов, однозначно соответствующих этапам игры. Следует повсеместно опираться на паттерн «Шаблонный метод (Template method)» [5] для придания гибкости. Интерфейс базового класса получается примерно такой (используется псевдоязык):

```
class StageBase
{
    // Виртуальные абстрактные public-методы
    function getStageId();          // Идентификатор фазы игры

    // Фиксированные шаблонные public-методы.
    // Несут общий функционал
    function start();               // Когда автомат перешел в это состояние
    function daemon();              // "Демон", вызывается в начале запроса
```

```
    function joinTableAction();     // Обработчик /table/jointable
    function getTableAction();      // Обработчик /table/gettable
    function turnAction(cardId);    // Обработчик /table/turn
    function passAction();          // Обработчик /table/pass
    function doubleAction();        // Обработчик /table/double
    function wakeUpAction();        // Обработчик /table/wakeup
    function getEventsAction();     // Обработчик /table/getevents
    function getUpdateAction();     // Обработчик /table/getupdate
    ...

    // Виртуальные абстрактные protected-методы.
    // Эти методы вызываются из соответствующих public-методов
    function onStart();             // Перегружаемый вход в состояние
    function onDaemon();            // Перегружаемый "демон"

    // замечание: следующие методы по умолчанию выбрасывают
    // ошибку ERROR_GAME_STATE (запрещено на данном этапе игры)
    // Это ядро системы: то, что заставляет программу работать
    function onJoinTableAction();   // Перегружаемый /table/jointable
    function onGetTableAction();    // Перегружаемый /table/gettable
    function onTurnAction(cardId);  // Перегружаемый /table/turn
    function onPassAction();        // Перегружаемый /table/pass
    function onDoubleAction();      // Перегружаемый /table/double
    function onWakeUpAction();      // Перегружаемый /table/wakeup
    function onGetEventsAction();   // Перегружаемый /table/getevents
    function onGetUpdateAction();   // Перегружаемый /table/getupdate

    // Вспомогательные фиксированные шаблонные protected-методы
    function nextPlayer();          // К следующему игроку
    function afterRound();          // Закончился круг
    function runAI();               // Запуск ИИ
    function changeStage(next);     // Сменить этап игры

    // Вспомогательные виртуальные абстрактные protected-методы
    function onNextPlayer();        // Перегружаемая часть nextPlayer
    function onAfterRound();        // Перегружаемая часть afterRound
    function onRunAI();             // Перегружаемая часть runAI
    function onChangeState(next);   // Перегружаемая часть chartStage

    // Виртуальные protected-методы для сериализации состояния игры
    function getGameInfo();
    function getGameState();
    function getGameEvents(lastEvent);
    function getCurrentPlayer();
    function getPlayerCards();
    ...

    // Фиксированные protected-методы валидации запроса.
    // Их удобно вызывать в начале методов onXxxxxAction
    function requireJoin();             // ERROR_NOT_JOINED?
    function requireCurrentPlayer();    // ERROR_CURRENT_PLAYER?
    ...
}
```

Тем самым вносится максимально возможная полиморфность между этапами игры и, с другой стороны, уже реализуется общая часть функционала игры. Это лишь пример того, как это может выглядеть. Конечно, разработчик имеет полную свободу по внесению изменений в данный базовый класс.

Все остальные конкретные классы этапов игры пока остаются пустыми.

Также разработчику понадобится класс `GameModel`, являющийся адаптером над базой данных. Он предоставляет методы доступа к данным, различные игровые алгоритмы и все, что связано с правилами игры. Его тоже следует пока оставить пустым.

Шаблон обработчиков методов API приложения получается примерно такой:

1. Проверятся авторизация, параметры и другие условия
2. Блокируется стол по идентификатору `tableId` (произвольными объектами синхронизации), чтобы другие запросы не могли испортить состояния базы данных
3. Конструируется экземпляр `GameModel`
4. Конструируется экземпляр одного из потомков `StageBase`, в зависимости от текущего этапа игры
5. Вызывается метод `stage.daemon`
6. Вызывается метод `stage.xxxxxAction`, в зависимости от вызванного метода API
7. Стол разблокируется (PHP, например, это делает автоматически)

Вся дальнейшая разработка сводится к наполнению классов этапов игры и игровой модели. Об этом в следующем параграфе.

3.3. Полезные практики при программировании функционала игры

Некоторые практики программирования оказываются крайне полезными при разработке веб-приложений. Они проверены временем, и о них много пишут. Вот краткий список основных практик:

1. Функционально-ориентированная разработка (Feature Driven Development) [8]
2. Разработка через тестирование (Test Driven Development) [9]
3. Рефакторинг (Refactoring) [6]

Применяя методику последовательной (функционально-ориентированной) разработки к пошаговой игре, следует действовать в следующем порядке:

1. Создать интерфейс базового класса `StageBase` и унаследовать от него пустые классы всех этапов игры
2. Добавить проваливающиеся тесты:
    1. Что в игру может зайти 1, 2, 3, 4 игрока
    2. Что в игру не могут зайти неавторизованные пользователи
    3. Что в игру не может зайти больше 4-х игроков
    4. Что один и тот же игрок не может войти в одну игру дважды
5. Внести изменения, необходимые для того, чтобы данные тесты прошли

6. Добавить проваливающиеся тесты:
    1. Что при входе 4-го игрока этап игры меняется
    2. Что в новом состоянии игроки входить в игру не могут
3. Внести изменения, необходимые для того, чтобы данные тесты прошли
4. ... так далее

Процесс естественным образом укладывается в методику разработки через тестирование. Как только на очередном этапе разработки начнут проваливаться старые тесты, то стоит задуматься, какие действия в базовом классе следует разместить в других местах, что нужно добавить/удалить и пр. То есть преобразовать базовый класс путём рефакторинга с целью поиска ошибок.

Заключение

Описанный в статье шаблон проектирования успешно применялся на предприятии в разработке ряда проектов.

Литература

1. Soren Johnson. Game Developer Column 8: Turn-Based vs. Real-Time. URL: `http://www.designer-notes.com/?p=151` (дата обращения: 10.04.2011).
2. Amit's Game Programming Information. URL: `http://www-cs-students.stanford.edu/~amitp/gameprog.html` (дата обращения: 10.04.2011).
3. Материал из свободной энциклопедии. Пошаговая стратегия. URL: `http://ru.wikipedia.org/wiki/Пошаговая_стратегия` (дата обращения: 10.04.2011).
4. Зеленков Ю.А. Введение в базы данных. Ярославль : ЯрГУ, 1997. URL: `http://www.mstu.edu.ru/study/materials/zelenkov/ch_7_1.html` (дата обращения: 10.04.2011).
5. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. СПб. : Питер, 2008. 366 с.
6. Фаулер, М. Рефакторинг // СПб. : Символ-Плюс, 2003. 432 с.
7. Hall J.R. Programming Linux Games // San Francisco : Loki Software, Inc, 2001. 415 с.
8. Палмер С.Р., Фелсинг Д.М. Практическое руководство по функционально-ориентированной разработке ПО // М. : Вильямс. 304 с.
9. TDD - Разработка через тестирование (Test Driven Development). URL: `http://wiki.agiledev.ru/doku.php?id=tdd`