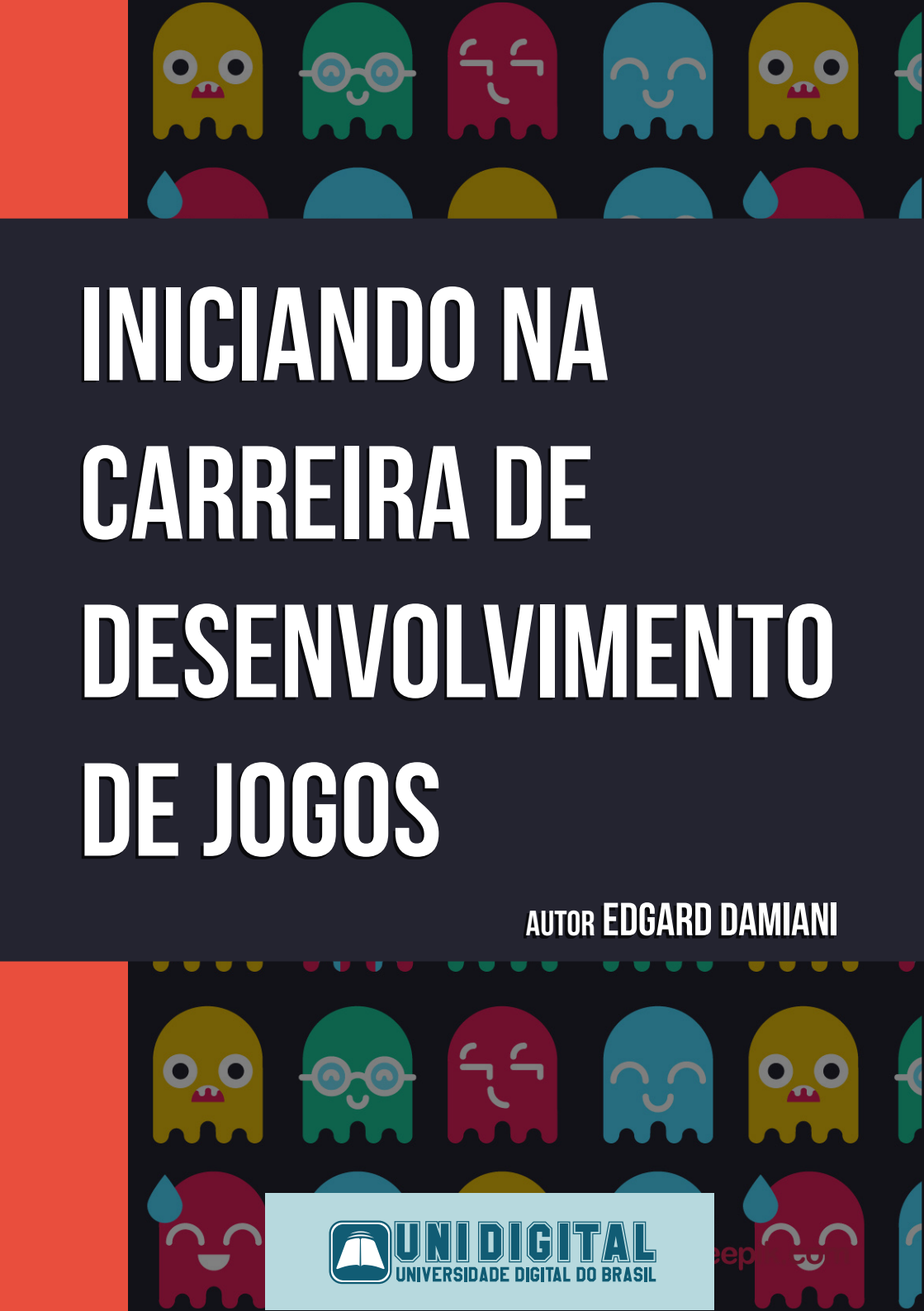
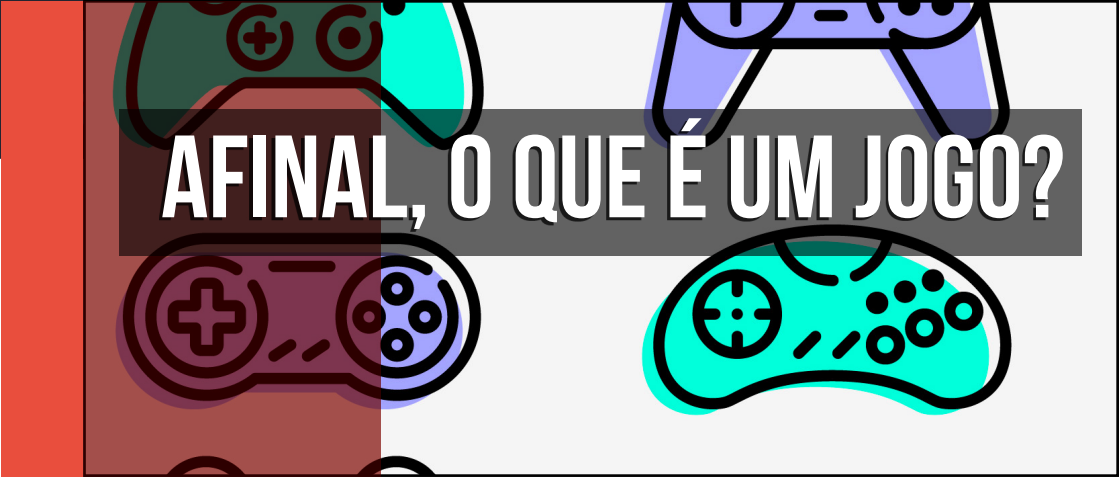


INICIANDO NA CARREIRA DE DESENVOLVIMENTO DE JOGOS

AUTOR EDGARD DAMIANI

UNIDIGITAL
UNIVERSIDADE DIGITAL DO BRASIL



AFINAL, O QUE É UM JOGO?

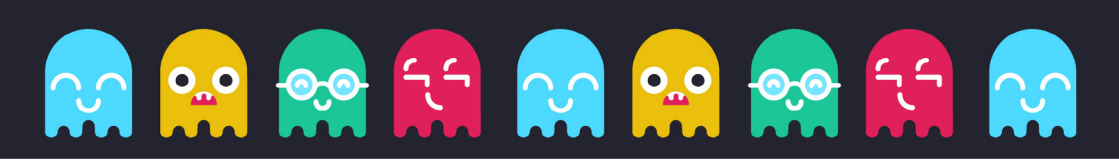
> MUITO DO QUE FALAMOS PARA EXPRESSAR NOSSOS PENSAMENTOS PODE ADQUIRIR UMA ROUPAGEM ABSOLUTA COM UMA FACILIDADE ENORME, MAS O BOM E VELHO DEUS DO TEMPO SEMPRE SE ENCARREGA DE ACABAR COM ESSE TIPO DE POSICIONAMENTO. O QUE ESCREVO AQUI NÃO DEVE SER VISTO COMO A VERDADE, E SIM COMO ALGO QUE DEVE SER LIDO, REFLETIDO E ABSORVIDO À MEDIDA QUE RESSOAR DENTRO DE VOCÊ, LEITOR; PORTANTO, MANTENHA PARA SI AQUILO QUE CONCORDAR, E NÃO PENSE DUAS VEZES EM DESCARTAR O RESTO.

Introdução

Quando falamos das grandes áreas de desenvolvimento de jogos, costumamos englobar todos os aspectos técnicos sob a alcunha de "programação", mas será que isso é uma boa definição do ponto de vista prático?

Tecnicamente falando, a programação é uma fase da engenharia de software, a qual abarca não só a programação em si, mas também todas as fases que vêm antes e depois da mesma. Falar apenas de programação acaba estreitando o foco do desenvolvedor, correndo o risco de fazê-lo esquecer que existem fases como planejamento, teste, implantação, manutenção etc. – dar a devida importância a todas as fases de desenvolvimento pode ser um fator decisivo para a longevidade de um projeto de software.

Este artigo tem como objetivo dar uma breve introdução sobre o desenvolvimento de jogos do ponto de vista da engenharia de software, concentrando na fase de planejamento e mostrando como os aspectos técnicos e não-técnicos do desenvolvimento de um jogo se mesclam durante todo o processo, levantando questões importantes sobre como



manter um certo grau de independência entre os departamentos de game design, multimídia e engenharia de software.

Ao longo do artigo você verá vários links da Wikipedia para verbetes importantes – caso não conheça algum dos termos destacados, sugiro que interrompa a leitura do artigo e leia o conteúdo do link antes de seguir adiante (sempre que possível, os links estarão em português).

O QUE É UM JOGO?

Quando falamos de jogos, é muito comum usar a palavra sem defini-la, o que pode causar inúmeros erros de interpretação. Sendo assim, antes de começar a discussão, vamos tentar definir o termo de um ponto de vista técnico:

"Jogo é uma atividade interativa em que os elementos participantes realizam ações baseadas em regras, dentro de um campo bem-definido e separado do mundo real, a fim de superar desafios propostos interna ou externamente para se alcançar um determinado objetivo."

Um jogo, portanto, tem três aspectos centrais:

- Interatividade.
- Entidades e regras.
- Imersão.

A interatividade é o aspecto mais óbvio de um jogo – afinal, não há jogo sem interatividade! As entidades e regras, por outro lado, definem os elementos constituintes de um jogo ou o chamado domínio – por exemplo, jogadores, obstáculos e itens são elementos constituintes de um jogo, enquanto as regras definem como tais entidades interagem entre si, bem como os objetivos a serem alcançados. Por fim, a imersão determina o aspecto de separação do mundo lúdico em relação ao mundo real, a fim de tal mundo lúdico poder ser experimentado como uma realidade em si.

Curiosamente, esses três aspectos – interatividade, entidades/regras e imersão – podem ser dispostos linearmente no famoso esquema

Entrada-Processamento-Saída de processamento de dados:

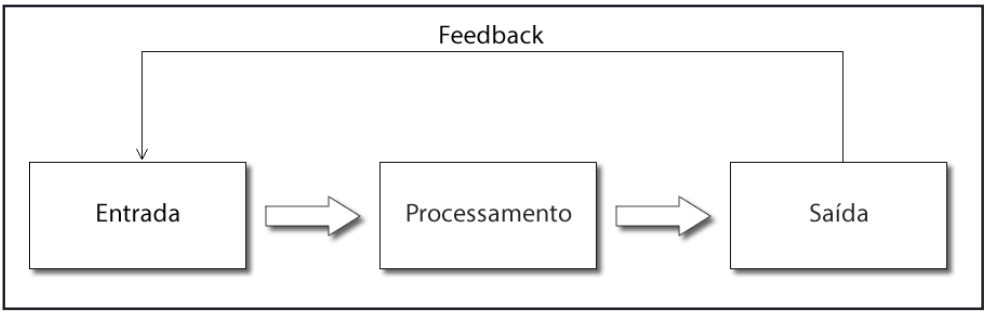


Figura 1. Esquema Entrada-Processamento-Saída.

Ou seja, o jogador interage com o jogo (entrada), o qual irá processar as entradas do jogador e redefinir os estados de todos os elementos de jogo (processamento) e apresentar os resultados ao jogador (saída) em um processo contínuo de retroalimentação ou feedback.

Loop de jogo

Sendo assim, do ponto de vista técnico, qual é a grande diferença entre um jogo e um aplicativo comum? Na maioria dos casos (e existem algumas exceções aqui), a grande diferença fica por conta do tempo levado desde a entrada até a saída e, consequentemente, até o processo de retroalimentação, ou seja, o tempo de resposta do ciclo de processamento de dados tende a ser muito curto, a fim de criar o que se chama de *experiência em tempo real.*

Quanto mais curto o tempo de resposta, melhor o jogo será capaz de responder às ações do usuário, facilitando o processo de imersão do jogador dentro do jogo. Esse ponto é tão fundamental que poderíamos dizer que ele é, literal e metaforicamente, o coração de um jogo – metafórico porque ele determina um ritmo, por assim dizer, de processamento do jogo, e literal porque ele cria o conceito central de *loop de jogo.*

O loop de jogo, portanto, é a estrutura central de um jogo que cuida para que a entrada, o processamento e a saída ocorram de maneira

ordenada e orquestrada, sempre respeitando o tempo de cada batida do coração metafórico.

Sempre que ocorre uma batida do coração do loop de jogo, o jogo avança um pouquinho, e esse avanço quase imperceptível de cada batida do coração gera um ritmo constante que cria a ilusão de que o universo daquele jogo está vivo. Podemos, então, modificar ligeiramente a figura anterior para esquematizar o panorama técnico geral de um jogo:

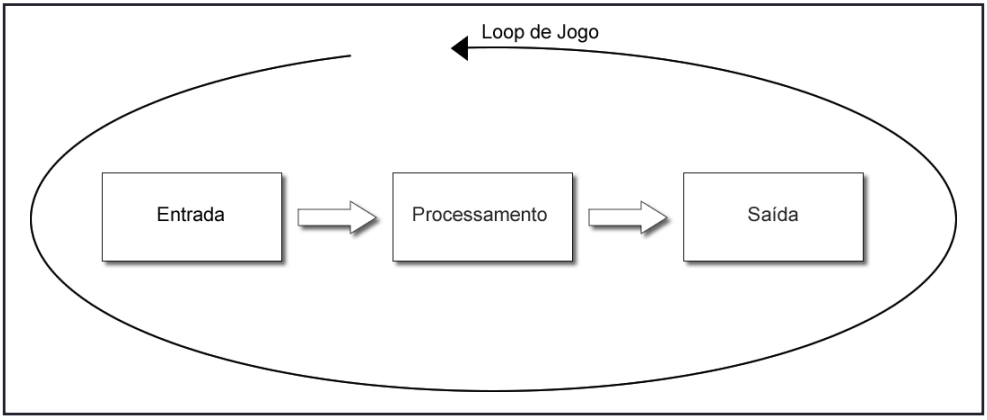


Figura 2. O esquema Entrada-Processamento-Saída como loop de jogo.

Usando uma pseudolinguagem de programação, poderíamos dizer de maneira simplificada que o loop de jogo é algo assim:

```
Enquanto jogo estiver sendo executado
      Capture entradas de usuário
      Atualize o jogo de acordo com as regras
      Atualize a inteligência artificial
      Atualize a física
      Atualize os gráficos
      Atualize o áudio
      ...
      Renderize a cena

Fim Enquanto
```

Em termos técnicos, quem provê funcionalidades como captura de entrada de usuário, física, gráficos, inteligência artificial etc. é uma biblioteca chamada game engine, ou mecanismo de jogo. O objetivo de um game engine (ou de qualquer engine, na verdade) é prover funcionalidades de alto nível que possam ser reutilizadas em vários jogos – por exemplo, captura de joystick, carregamento de imagens e modelos 3D, cálculos de colisão e aplicação de forças, cálculo do caminho percorrido por um personagem, reprodução de arquivos MP3 etc. Sem um game engine, desenvolver um novo jogo sempre significaria recomeçar da estaca zero, gerando um ciclo de desenvolvimento mais longo (e mais caro) do que o necessário.

Do ponto de vista da arquitetura de software, poderíamos dizer que o game engine cria uma camada de abstração entre o jogo em si e as APIs de um sistema operacional, que são bibliotecas de baixo nível com acesso (quase) direto ao hardware:

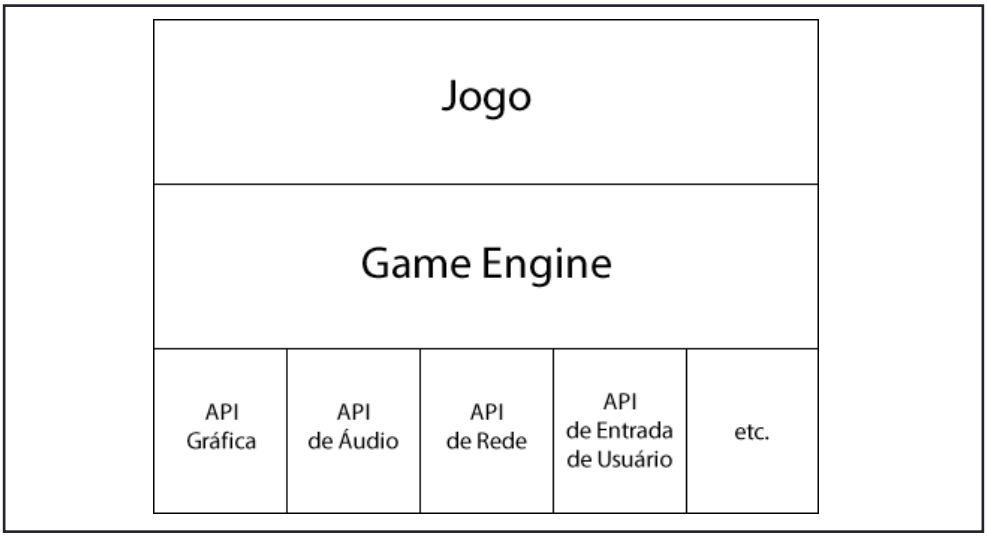
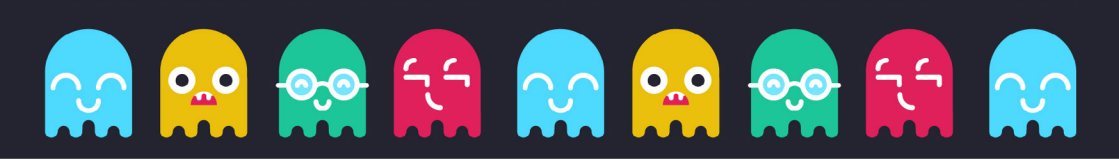


Figura 3. Especificação de um jogo dentro de uma arquitetura de 3 camadas.

Agora que sabemos como o game engine e o jogo conversam entre si, podemos nos concentrar na camada superior e fazer a grande



pergunta: como estruturar um jogo?

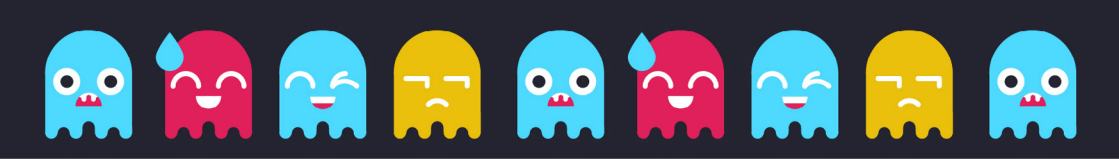
O processo de transformação de uma ideia em software é um processo delicado, exigindo uma atenção especial por parte das pessoas envolvidas na sua criação. Simplesmente sair programando sem um projeto prévio pode até funcionar em situações simples como a criação de um Pong, mas no caso de jogos mais complexos faz-se necessário estabelecer metodologias de desenvolvimento e uma arquitetura geral que permita dividir a complexidade do processo em partes menores e mais gerenciáveis.

Tenha em mente, portanto, que quando falamos de arquitetura ainda não estamos falando de código – o objetivo neste nível é determinar um plano de ataque inicial que nos permita criar uma base de código mais flexível e que siga certos princípios e boas práticas de desenvolvimento, mas antes de prosseguirmos no assunto, vamos subir um pouco mais o nosso ponto de vista e entender o panorama geral de desenvolvimento de um jogo.

A tríade Game Design/ Multimídia/Engenharia de Software

A criação de um jogo eletrônico pode envolver literalmente centenas de pessoas com dezenas de papéis e cargos distintos, mas podemos resumir o processo de desenvolvimento central em três grandes áreas: game design, multimídia e engenharia de software.

A área de game design, de um modo muito geral, é responsável por criar e cuidar daquilo que se chama *jogabilidade*, ou seja, a habilidade ou capacidade de jogar. Ao contrário do que se pode pensar, a jogabilidade em si não está inicialmente ligada ao ambiente técnico – muitos game designers criam seus primeiros protótipos utilizando materiais nada tecnológicos como tabuleiros, miniaturas ou peões, dados, fitas métricas etc. Estou enfatizando este ponto aqui para deixar claro



que a essência do trabalho de um game designer são as entidades e as regras de um jogo, que podem ser expressas futuramente de mil maneiras distintas no ambiente tecnológico. (Obviamente, existem jogos cujas regras só podem ser expressas adequadamente em um meio digital – nesses casos, as áreas de game design e programação acabam se mesclando, mas sem mudar o fato de que a essência do game design é o trabalho de criar e equilibrar a jogabilidade.)

Uma vez que a área de game design tenha considerado as entidades e regras como suficientemente desenvolvidas em nível de protótipo, duas frentes de trabalho surgirão: a primeira deverá considerar os desafios técnicos de construção do jogo, e a segunda deverá considerar os aspectos estéticos ou multimídia do mesmo. Engenharia de software e desenvolvimento multimídia, portanto, dependem de um bom game design para que suas tarefas possam correr normalmente, lembrando que design significa projeto, e qualquer implementação depende de uma fase prévia de projeto para que possa ter um rumo adequado, garantindo um mínimo de foco a fim de os desenvolvedores não ficarem pisando uns nos pés dos outros. (O que não significa, é claro, que isso não vá acontecer; hoje sabe-se que é praticamente impossível tentar definir perfeitamente toda a fase de análise e projeto antes de se iniciar a fase de desenvolvimento, e é por isso que as metodologias ágeis promovem a ideia de desenvolvimento iterativo.)

Para que cada elemento dessa tríade possa fazer sua parte, é importante que suas áreas ou interesses possam ocorrer o máximo possível em paralelo, para que cada um possa continuar realizando seu trabalho com um mínimo de independência em relação às outras duas áreas. O grande desafio, então, é organizar o desenvolvimento de forma que a área técnica não se torne um gargalo para as demais áreas e vice-versa. Nesse sentido, a responsabilidade acaba ficando nas costas da engenharia de software, já que, em última instância, o objetivo é produzir um software.

Como, então, garantir que cada área consiga fazer a sua parte com um mínimo de independência? A solução é dividir o processo de desenvolvimento do software considerando a existência dessas três áreas, e por coincidência essa arquitetura específica existe desde a primeira versão do primeiro sistema operacional gráfico.

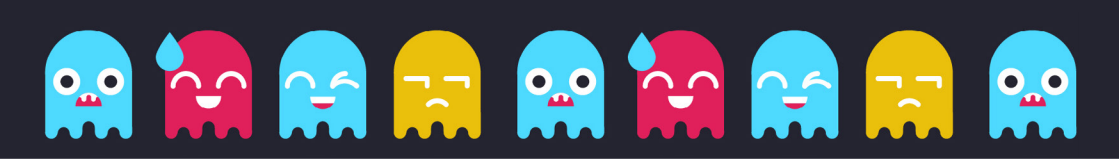
MVC e amigos

Existem inúmeros problemas que são recorrentes no mundo do desenvolvimento de software, e vários desses problemas recorrentes tiveram suas soluções devidamente catalogadas tanto no nível de arquitetura quanto no nível de projeto. Cada problema com sua respectiva solução e aplicação é conhecido como padrão de arquitetura

Figura 4. Ilustrando Arquitetura de Software

ou padrão de projeto (dependendo do nível em que estivermos conversando), com os padrões de arquitetura representando soluções mais amplas do que os padrões de projeto. O objetivo dos padrões de arquitetura e/ou dos padrões de projeto é criar um vocabulário conceitual de desenvolvimento que possa ser usado para melhor estruturar o projeto como um todo, facilitando a visão global do projeto e, em última instância, organizando o processo de programação em si.

Um padrão de arquitetura amplamente conhecido é o Modelo-Visão-



Controlador ou MVC. Nesse padrão, um sistema é dividido em três grandes áreas:

- Modelo: Define o conjunto de entidades e as regras de interação entre elas.
- Visão: Define a apresentação do modelo ao usuário.
- Controlador: Captura a entrada de dados do usuário.

Se reordenarmos os elementos para Controlador-Modelo-Visão, veremos que essa ordem guarda uma coincidência muito grande com a sequência Entrada-Processamento-Saída.

Mas o que exatamente o padrão MVC soluciona? O padrão MVC foi criado para resolver problemas relacionados a interfaces de usuário, e a sua principal ideia é a separação de interesses, ou seja, cada elemento do padrão MVC agrupa funcionalidades semelhantes, permitindo que os demais elementos concentrem-se nas suas áreas de atuação.

Tomemos como exemplo o modelo: seu objetivo é definir o núcleo de processamento de um programa, independentemente de como o usuário estiver interagindo com ele ou de como o programa será apresentado ao usuário.

Se imaginarmos um jogo como o Space Invaders, podemos entender o modelo como sendo a cena de jogo (ou melhor, os elementos de cena que interajam com as entidades de jogo), a nave do jogador, as naves inimigas, os tiros disparados pelas naves e as regras de interação entre todos eles – no entanto, no modelo não estamos interessados em como as entidades serão representadas, e sim no que as entidades fazem.

Observe a figura a seguir:

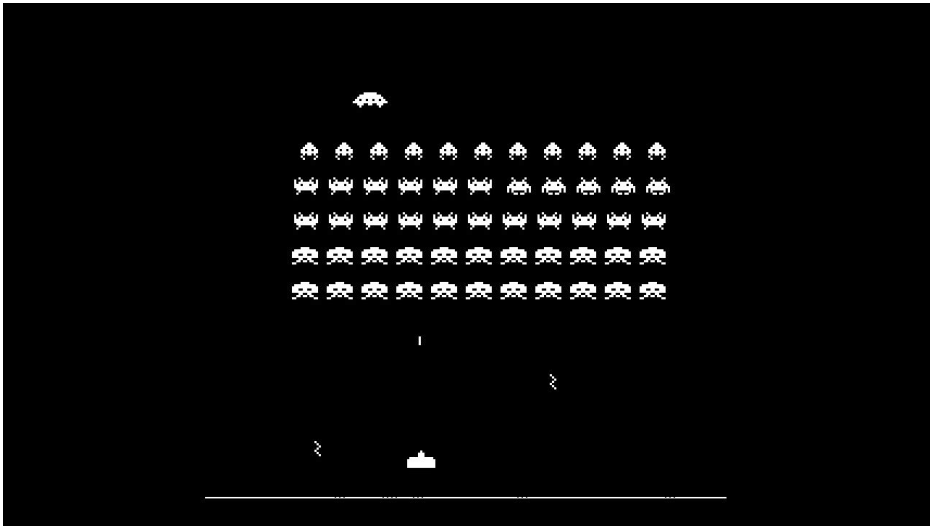


Figura 5. O jogo Space Invaders como visto pelo jogador.

Nela, podemos ver o jogo tal como será apresentado ao jogador. No modelo, no entanto, podemos imaginar o jogo assim:

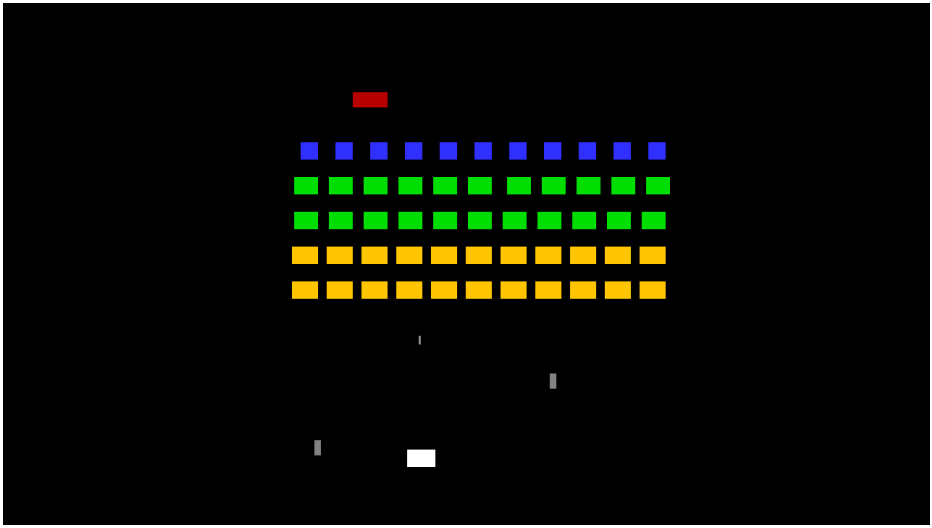


Figura 6. O mesmo Space Invaders visto do ponto de vista do programador do modelo.

Aqui, estamos preocupados apenas com os *aspectos abstratos* de cada entidade como posição, tamanho, área de colisão, velocidade etc., sem nos preocuparmos com o aspecto estético de tais entidades (os retângulos foram pintados de cores diferentes apenas para deixar claro que cada cor representa um tipo de entidade). Dessa forma, o programador do modelo fica livre para se concentrar nos algoritmos do jogo em si, sem se distrair com detalhes como se as naves serão 2D ou 3D, se um efeito sonoro será reproduzido ao disparar um tiro etc.

A visão, por outro lado, cuida exatamente dos detalhes negligenciados pelo modelo – para isso, é importante que a visão tenha acesso aos dados do modelo, para que as entidades possam ser representadas da maneira correta em termos de posição, tamanho, estado atual etc. Se imaginarmos um jogo como o Prince of Persia, veremos que alguns elementos fazem parte do modelo e outros da visão. Um exemplo claro disso é o movimento do personagem principal: ao movimentá-lo para o lado, o personagem irá se deslocar na tela, ao mesmo tempo em que apresenta uma sequência de imagens que dão a sensação de que ele está realmente andando até chegar ao ponto de destino (o famoso ciclo de caminhada ou walking cycle). Na figura a seguir, estamos representando o personagem como um retângulo que se movimenta da esquerda para a direita:

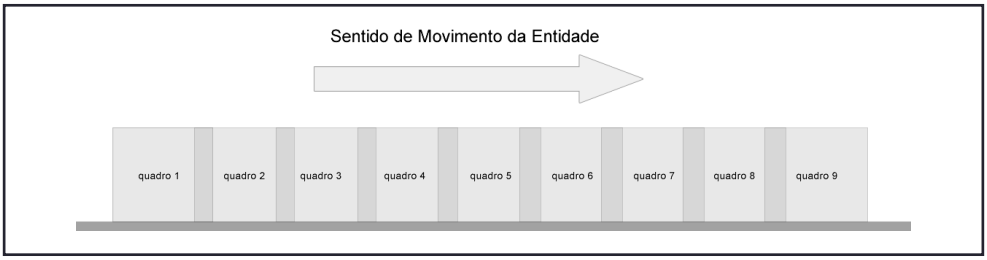


Figura 7. Sequência de nove quadros sobrepostos representando o movimento da entidade.

Este seria o nível de implementação do modelo, no qual ele se encarregaria do deslocamento da entidade pela cena e de estabelecer qual é o estado atual do mesmo (parado, movendo-se para a esquerda, movendo-se para a direita, pulando etc.).

A visão, por sua vez, pegaria o estado atual da entidade que representa

o príncipe dentro do modelo e faria isso:

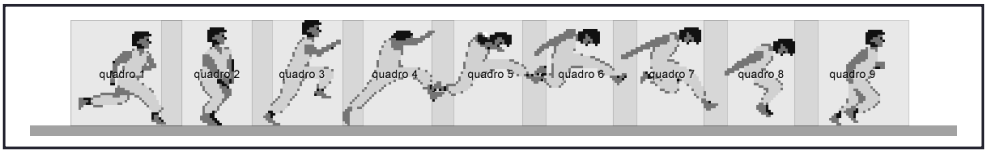


Figura 8. Os mesmos nove quadros com os respectivos sprites sobrepostos.

A visão, portanto, ficaria responsável por capturar o estado do personagem definido no modelo e apresentar a sequência de imagens equivalente a tal estado.

Até aqui tratamos o personagem como se ele estivesse sendo movido por uma força oculta, mas na verdade precisamos de um elemento que sirva de ponte entre o jogador e o jogo em si (ou seja, o modelo), e esse componente é o controlador. A grande vantagem de usar um controlador está representada na figura a seguir:

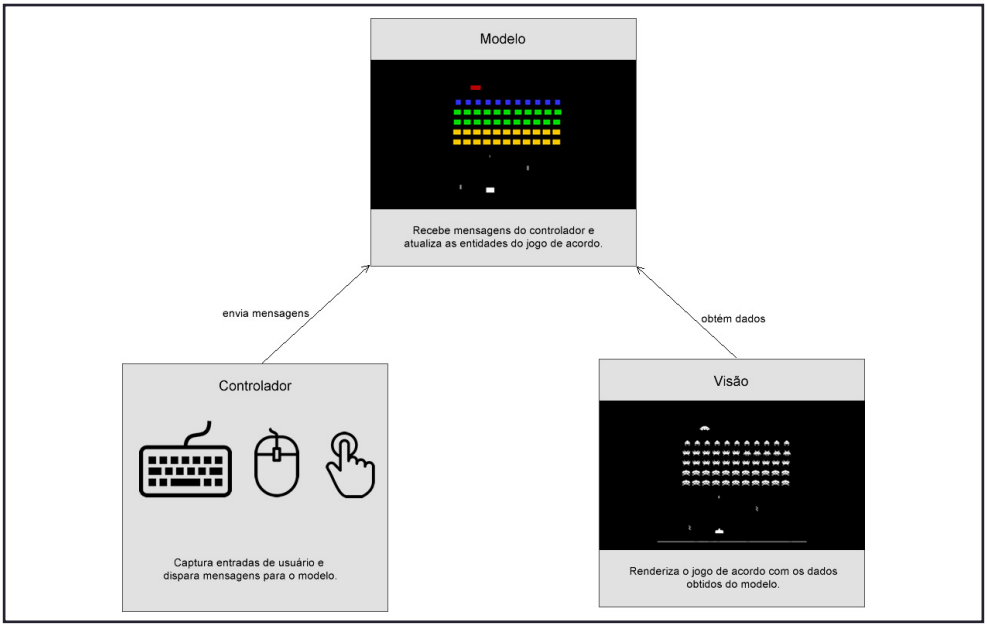


Figura 9. O controlador dentro do esquema geral do MVC.

Perceba que o controlador recebe qualquer tipo de entrada de usuário, mas envia mensagens padronizadas para o modelo – por exemplo, independentemente de o jogador pressionar a seta para a direita no teclado ou o direcional direito no joystick, o modelo receberá uma mensagem "mova o personagem para a direita". Com isso, conseguimos mais uma vez isolar o modelo de certos detalhes de implementação, permitindo que o programador do modelo concentre-se em trabalhar as entidades e as regras entre elas.

Exemplo prático: Pong

Para tentar explicar certos conceitos, vamos usar o bom e velho Pong como exemplo. Inicialmente, surge a ideia de fazer um jogo no qual dois jogadores devem rebater uma bola a fim de impedir que seu oponente a devolva, marcando um ponto. A criação dessa ideia, mais os detalhes de como os jogadores vão interagir com a bola, como cada entidade vai se comportar dentro do jogo, como será o sistema de pontuação etc. – tudo isso entra no campo de atuação do game designer.

Em seguida, a ideia deverá ser transformada em software, e para isso precisamos determinar em nível de projeto quais são os elementos principais do jogo, como mostra a figura a seguir:

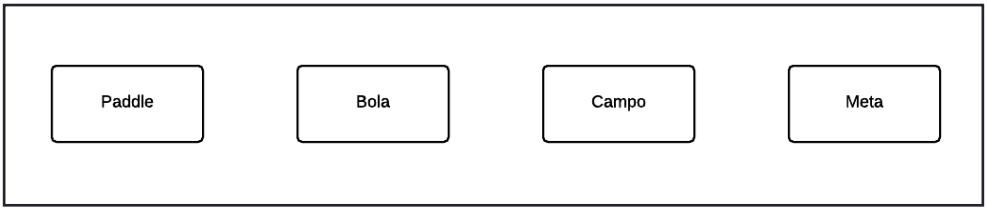


Figura 10. Entidades de jogo.

Agora que sabemos quais são as entidades, podemos determinar as interações básicas entre elas:

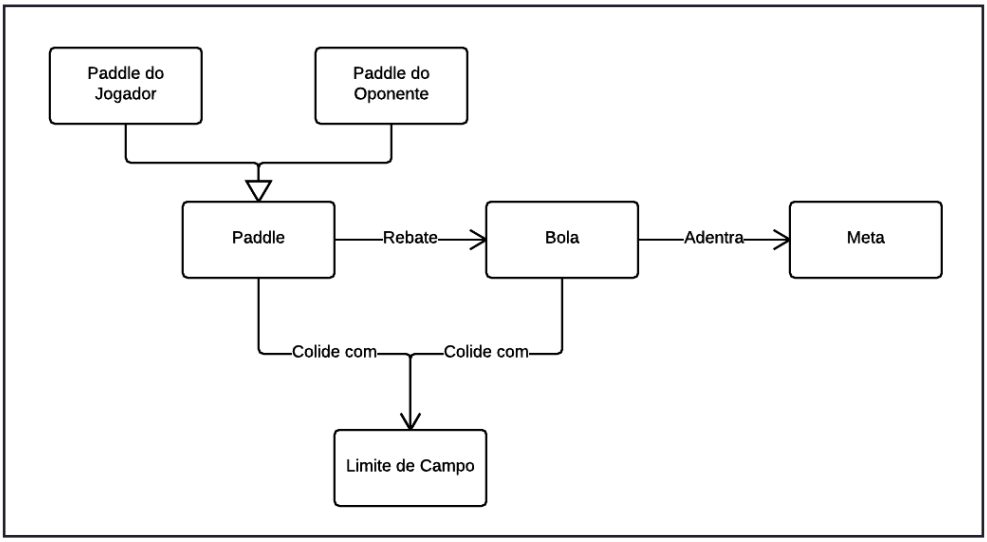


Figura 11. Relações entre as entidades de jogo.

O que estamos fazendo é determinar o modelo de domínio do jogo, sem nos preocuparmos com detalhes de implementação relacionados à entrada de usuário ou à apresentação do jogo ao jogador.

Perceba, no entanto, que houve uma alteração entre as figuras 9 e 10: o que previamente chamamos de "Campo" virou "Limite de Campo" na figura 10 – isso é um acontecimento perfeitamente normal quando estamos projetando um software, pois a compreensão do mesmo vai aumentando conforme avançamos em seu desenvolvimento.

Vamos, agora, adicionar o controlador ao nosso projeto, supondo que o paddle do jogador possa ser controlado por teclado, mouse ou toques em uma touch screen:

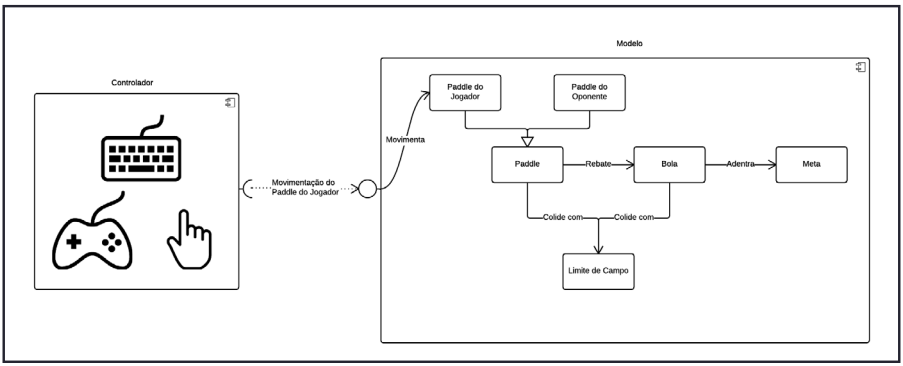


Figura 12. Relação entre controlador e modelo no Pong.

Perceba que o controlador gera mensagens ao modelo, mas o modelo não faz a menor ideia do tipo de hardware que as está gerando, já que a mensagem apenas indica que o paddle deve ser movido. (Poderíamos deixar a mensagem mais genérica ainda, indicando apenas que houve um "movimento para a esquerda", por exemplo, sem indicar qual entidade deve ser movimentada.)

Por fim, precisamos adicionar a visão ao projeto:

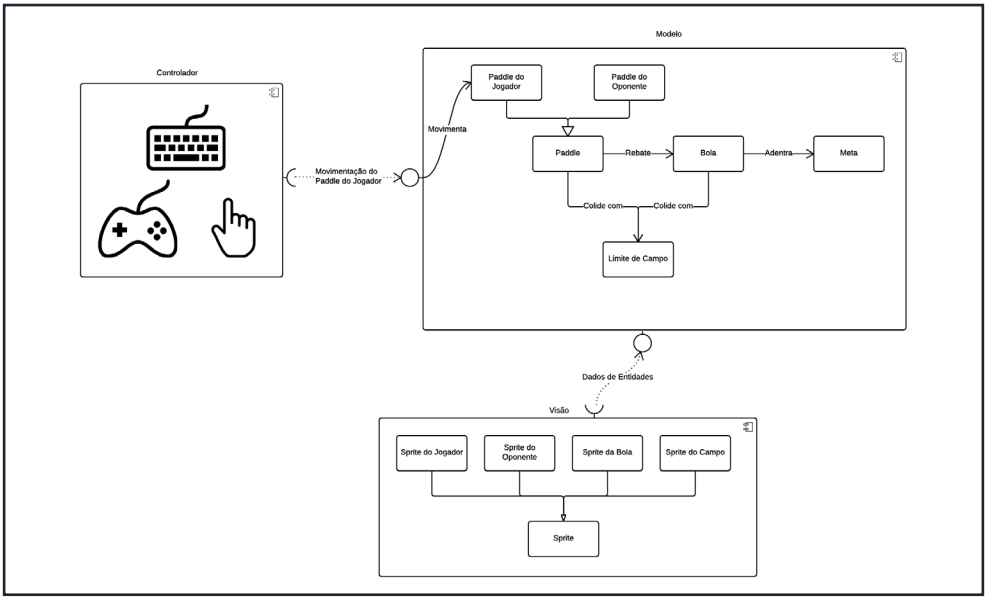


Figura 13. Inclusão da visão no esquema MVC do Pong.

Perceba que a visão possui elementos semelhantes ao modelo, com cada um desses elementos obtendo informações sobre a sua contraparte abstrata para que o jogo possa ser corretamente apresentado ao jogador.

Conforme vamos avançando o projeto, pouco a pouco os elementos conceituais vão se transformando em objetos propriamente ditos, com uma representação cada vez mais próxima do mundo da programação. Vejamos, então, como dar mais um passo nesse sentido:

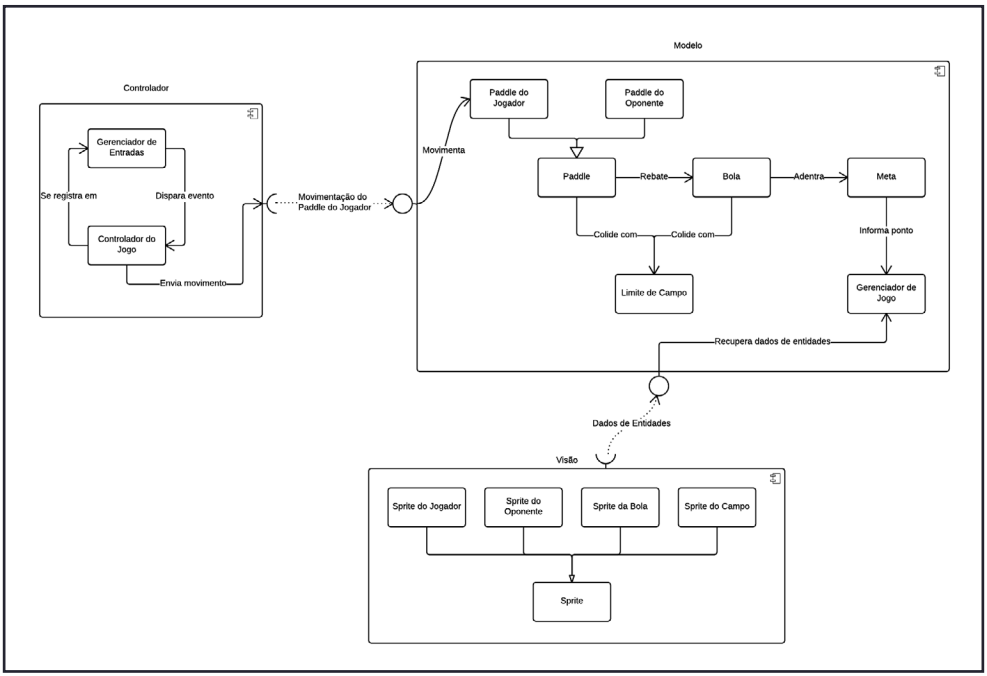
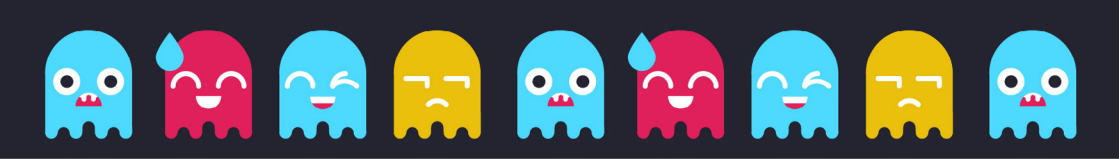


Figura 14. Versão mais detalhada do esquema MVC do Pong.

Desta vez, criamos dois objetos dentro do controlador: um Gerenciador de Eventos, responsável por receber os dados brutos vindos do usuário, e o Controlador do Jogo propriamente dito, responsável por receber os eventos do gerenciador, interpretá-los e enviar os comandos corretos ao modelo – perceba que a forma como o Gerenciador de Entradas e



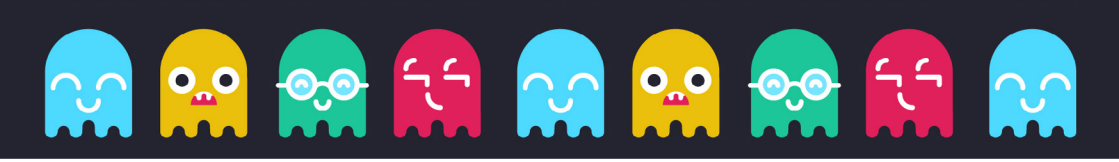
o Controlador do Jogo se relacionam formam um padrão de projeto chamado Publicador/Assinante (Publisher/Subscriber) ou Observador (Observer).

Outro elemento que foi adicionado ao diagrama foi o Gerenciador de Jogo dentro do modelo – o primeiro ponto motivador de criação do mesmo foi o reconhecimento de que alguém deveria armazenar o placar do jogo, entre outras informações globais (por exemplo, o jogo está em execução ou em um estado de reinício de partida?).

Desse ponto em diante, o diagrama pode começar a ser transformado em classes de programação, mostrando seus elementos internos como variáveis e métodos; após isso, inicia-se o processo de programação em si, o qual irá fornecer feedback sobre a validade e veracidade dos diagramas criados.

Agora, o que fazer caso fique constatado que um diagrama de projeto não pode ser traduzido fielmente em código? Lembrando do espírito do processo de desenvolvimento iterativo, devemos retornar ao diagrama errôneo e corrigi-lo, para que a representação abstrata do código e o código em si continuem em sincronia. Apesar de isso parecer trabalho em dobro, ao fazer isso você estará mantendo a documentação do software atualizada, auxiliando enormemente nos processos de teste e manutenção (lembre-se, desenvolver um software não significa apenas programar!).

Por fim, é importante entender que os diagrams em nível de projeto e em nível de implementação nunca terão uma representação um-para-um entre seus elementos constituintes, já que a implementação sempre terá mais detalhes do que o projeto – por exemplo, os diagramas de projeto não estão indicando como as entidades e os sprites serão armazenados, mas os diagramas de implementação deverão de alguma forma deixar claro como isso vai acontecer (por exemplo, usando repositórios de objetos ou object pools). Nesse nível, os padrões de projeto começam a ganhar proeminência, já que eles possuem uma granularidade tal que permite uma tradução quase direta em código.



O QUE VEM DEPOIS DISSO?

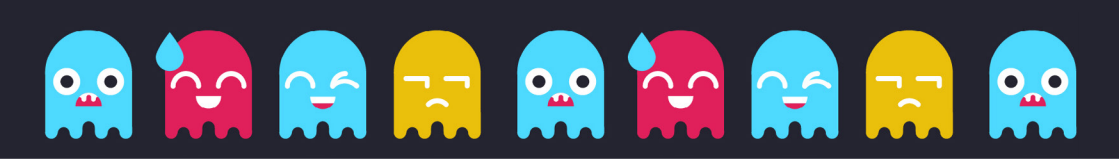
Os próximos passos envolveriam preencher as lacunas entre a representação em nível de projeto e a criação de código, mas infelizmente preciso encerrar este artigo antes que ele vire um livro! De qualquer forma, gostaria de comentar rapidamente um ponto que gera muitas críticas ao padrão MVC: a criação da interface gráfica.

Um dos grandes problemas do padrão MVC é que ele foi concebido em um contexto no qual todas as interações do usuário do software ocorreriam em um mesmo plano, por assim dizer, mas um jogo possui pelo menos dois planos de interação: o jogo em si, que recebe determinadas entradas de usuário – por exemplo, setas direita e esquerda do teclado –, e a interface gráfica de usuário ou GUI (Graphical User Interface), que recebe um conjunto diferente de entradas – por exemplo, um clique de mouse sobre um botão de menu.

Se você tentar gerenciar tudo isso de maneira planificada, certamente terá problemas de confusão entre as camadas da GUI e do jogo em si, principalmente naqueles casos em que as entradas de usuário forem muito similares, como ocorre, por exemplo, ao tratar eventos de toque em um dispositivo móvel.

Apesar de estarmos cutucando os limites do padrão MVC clássico, é importante entender que o padrão não é estanque. Aliás, muito pelo contrário: do padrão clássico surgiram variantes como MVP, MVVM, HMVC (MVC hierárquico) etc., mostrando que dificilmente uma solução será tão abrangente a ponto de conseguir abraçar todo o universo de desenvolvimento de software. Aliás, o padrão HMVC (ou alguma variante do mesmo) pode ser visto como um bom candidato para resolver o dilema jogo/GUI, permitindo que um controlador global informe sub-controladores sobre o disparo de um evento, com o controlador global definindo certos critérios para que um sub-controlador receba ou não uma determinada mensagem.

Outro ponto que poderia ser discutido mais amplamente seria a redistribuição do padrão MVC em quatro partes, e não três, mas isso já é uma história para outro e-book.



Conclusão

A engenharia de software enquanto disciplina e metodologia é um assunto vastíssimo que deveria ser parada obrigatória para qualquer programador, mesmo que ele não tenha interesse em seguir tal carreira. Cada vez mais, grandes empresas de jogos estão usando o título "Programador Sênior" para indicar programadores que deverão não só gerenciar equipes de programadores, mas também gerenciar o processo de desenvolvimento em si, fatalmente caindo no universo da engenharia de software. Sendo assim, cada vez mais as empresas estão demonstrando que a sequência evolutiva Desenvolvedor Júnior > Desenvolvedor Pleno > Desenvolvedor Sênior caracteriza uma ampliação contínua da visão do desenvolvedor, até o ponto em que não ser capaz de enxergar o todo de um projeto – e não ser capaz de gerenciá-lo – pode indicar estagnação profissional.

Eu já ouvi muitos programadores dizerem que não gostam de diagramar ou trabalhar em nível de projeto porque isso acaba gerando burocracia e papelada sem valor, e é verdade que excesso de análise e projeto pode levar à chamada "paralisia da análise", na qual os analistas e projetistas ficam tão obcecados com os detalhes que nunca chegam à fase de implementação. De qualquer forma, é importante entender que isso é um caso extremo, e que o conceito de desenvolvimento iterativo surgiu justamente para evitar que tais extremos ocorressem.

Simplesmente sentar e começar a programar pode parecer algo lógico, mas em projetos de grande porte isso pode significar meses, ou até mesmo anos de trabalho jogados no lixo caso os programadores tenham decidido seguir um rumo que se mostrará um beco sem saída lá na frente. Projetar um software ajuda a entender certos becos com antecedência, gerando um certo gasto de tempo inicial que pode significar uma economia enorme de tempo e dinheiro no futuro.

Portanto, não subestime o poder da engenharia de software, mas também não a abrace como se fosse uma religião – mantenha cada coisa em seu lugar e dando a elas suas devidas proporções, e certamente você terá uma experiência de desenvolvimento de jogos muito mais satisfatória no futuro.


UNI DIGITAL
UNIVERSIDADE DIGITAL DO BRASIL