Instituto Politécnico do Cávado e do Ave
Escola Superior de Tecnologia

Licenciatura
em
Engenharia Informática Médica

Sistema de gestão de urgências de um hospital

Docente: Luís Ferreira

Autor: Ana Margarida Vasco, nº26055

Daniel Macedo, nº26057

Pedro Pinto, nº26049

Novembro e dezembro de 2023

Resumo

O propósito do nosso projeto prático na disciplina de Programação Orientada a Objetos (POO) consiste no desenvolvimento de um sistema de gestão de urgências hospitalares.

Para a implementação deste sistema, foram utilizadas as seguintes classes: Doente, Médico, Pessoa, SalaaEspera, Local, Consultório e Sistema. Cada uma dessas classes desempenha um papel específico no funcionamento global do projeto, contribuindo para uma gestão eficiente e organizada das situações de urgência.

Ao utilizar estas classes de forma integrada, o sistema visa otimizar a gestão de urgências hospitalares, proporcionando uma abordagem estruturada e eficaz para o tratamento de doentes, de acordo com as suas necessidades e prioridades.

Índice

Resumo 1

Índice de Figuras 3

Introdução 5

Revisão da Literatura Relacionada com o Trabalho 6

Classes Utilizadas no Projeto 7

Hospital 7

Consultório 7

SalaaEspera 11

Sistema 14

Local 18

Intervenientes 19

Doente 19

Medico 23

Pessoa 26

Resultados 27

Conclusão 28

Índice de Figuras

Figura 1: Atributos de Consultório 7
Figura 2: Métodos de Consultório 7
Figura 3: Construtor por Omissão 8
Figura 4: Construtor Personalizado 8
Figura 5: Construtor Estático 8
Figura 7: Propriedade do Número 9
Figura 6: Propriedade de Especialidade 9
Figura 8: Propriedade de Estado 9
Figura 9: Propriedade Total de Consultórios 9
Figura 10: Função que Insere Medico no Consultório 10
Figura 11: Função que Verifica se Existe Medico no Consultório 10
Figura 12: Função que Verifica se Existe Doente no Consultório 10
Figura 13: Função que Adiciona Medico no Consultório 10
Figura 14: Função que Adiciona Doente no Consultório 11
Figura 15: Destrutor de classe do Consultório 11
Figura 16: Atributos de SalaaEspera 11
Figura 17: Métodos de SalaaEspera 12
Figura 18: Construtor Instância da SalaaEspera 12
Figura 19: Função que Adiciona Doente na Sala de Espera 12
Figura 20: Função que Conta os Doentes 12
Figura 21: Função que Mostra Lugares Ocupados 13
Figura 22: Função que Verifica se Existe Doentes na Sala de Espera 13
Figura 23: Função que Conta Quantos Doentes Existem na Sala de Espera 13
Figura 24: Função que Retorna uma lista de Doentes ordenados por Prioridade 14
Figura 25: Destrutor de Classe da Sala de Espera 14
Figura 26: Atributos de Sistema 14
Figura 27: Listas de Doentes e Médicos no Sistema 14
Figura 28: Função que Regista um Doente 15
Figura 29: Função que Regista um Médico 15
Figura 30: Função que Encaminha para a Triagem 15
Figura 31: Função que Atualiza o Historico 16
Figura 32: Função que Mostra Informação de um Doente 16
Figura 33: Função que Atruibui um Medico a um Doente 16
Figura 34: Função que Realiza uma Consulta 17
Figura 35: Função que Faz a Triagem 17
Figura 36: Função que Encaminha para a Sala de Espera 18
Figura 37: Enum Estado 18
Figura 38: Class Local (Herança) 18
Figura 39: Atributos de Doente 19
Figura 40: Métodos de Doente 19
Figura 41: Construtor por omissão do Doente 20
Figura 42: Construtor que permite personalizar os atributos do doente 20

Figura 43: Construtor estático do atributo Total de Doentes 20
Figura 45: Propriedade do id do Doente 21
Figura 44: Propriedade do Nome 21
Figura 47: Propriedade do NIF 21
Figura 46: Propriedade do Número de Utente 21
Figura 49: Propriedade de Pulseira 21
Figura 48: Propriedade de Prioridade 21
Figura 50: Propriedade de Sintomas 22
Figura 51: Propriedade de Histórico 22
Figura 52: Propriedade estática do atributo Total de Doentes 22
Figura 53: Método ToString() da classe Doente 22
Figura 54: Função que Atualiza Histórico 22
Figura 55: Função que Mostra Informação do Doente 23
Figura 56: Destrutor da Class Doente 23
Figura 57: Atributos de Médico 23
Figura 58: Métodos de Médico 23
Figura 59: Construtor por omissão de Médico 24
Figura 60: Construtor que permite a personalização dos atributos do Médico 24
Figura 61: Construtor estático do atributo Total de Médicos 24
Figura 63: Propriedade do Nome 25
Figura 62: Propriedade de Especialidade 25
Figura 64: Propriedade do Número de Identificação 25
Figura 65: Propriedade estática do atributo Total de Médicos 25
Figura 66: Método ToString() na classe do Médico 26
Figura 67: Destrutor de class de Médico 26
Figura 68: Atributo de Pessoa (Herança) 26

Introdução

A gestão de um sistema de urgências num hospital tem um papel fundamental na prestação de cuidados de saúde de qualidade. Atualmente, os hospitais enfrentam diversos desafios na organização e priorização de urgências, resultando em tempos de espera prolongados e possível subutilização de recursos médicos e na disponibilidade de recursos eficientes.

Assim, com o aumento constante dos doentes, que procuram as urgências, tende a gerar pressão nos recursos hospitalares o que origina desafios na triagem, no atendimento prioritário e na gestão eficaz das salas de espera.

A eficiência da gestão de urgências é fundamental para salvar vidas e garantir que os doentes recebam tratamento adequado quando necessário. Uma abordagem sistemática pode reduzir os tempos de espera, melhorar a disponibilidade de recursos e aumentar a satisfação do doente.

Este sistema visa preencher falhas na gestão de emergências hospitalares, fornecendo uma solução abrangente e eficiente. De forma a atender às necessidades específicas do ambiente de urgências hospitalares, pretendemos criar uma ferramenta que integre as classes consultório, sala de espera, local, sistema, médico, doente, pessoa. Deste modo, permite uma triagem mais rápida e precisa, disponibilizando doentes com base nas suas condições, prioridades e histórico, criando listas de espera dinâmicas, garantindo que doentes mais urgentes recebam atendimento imediato.

Revisão da Literatura Relacionada com o Trabalho

A gestão eficaz de um sistema de urgências é fundamental num hospital para melhorar a eficiência e a qualidade do atendimento. Nesta gestão é crucial a integração de classes específicas para consultório, médico, doente e salas de espera e permite simplificar a gestão global do sistema.

Num atendimento competente o maior desafio encontra-se no Sistema, na classificação das prioridades e na gestão da lista de espera. Ao longo do tempo, pesquisas têm demonstrado que uma boa gestão das listas de espera é crucial para facilitar o fluxo de doentes. A disponibilidade dinâmica de recursos pode ser facilitada por sistemas informáticos que melhoram a distribuição da carga de trabalho e reduzem os tempos de espera. Através da linguagem C# consegue-se criar comunicação com o utilizador e conter diferentes classes de um sistema hospitalar.

Classes Utilizadas no Projeto

Neste projeto utilizamos as seguintes classes: SalaaEspera, Local, Consultório, Sistema na biblioteca Hospital e Doente, Médico, Pessoa na biblioteca Intervenientes. Utilizamos também uma class Program para o main.

Hospital

Consultório

A classe consultório contém diversos atributos:

- Uma constante para defenir o tamanho máximo do consultório;
- Número do consultório, do tipo inteiro;
- Especialidade do médico, do tipo especialidade;
- Uma variável do tipo Medico chamada medicoResponsavel;
- Uma variável do tipo Doente chamada doenteConsultorio;
- O número total de médicos, do tipo inteiro;
- O número total de consultórios, do tipo inteiro.

```
#region ESTADO

#region ESTADO_CONSULTORIO
const int MAX = 200; //define o tamanho maximo do array

public int numero; //identificador  do consultorio
private Especialidade especialidade; //tipo de especialidade
public Medico medicoResponsavel; //a
public Doente doenteConsultorio;
private int TotMedicos; //total de medicos existentes
#endregion

#region ESTADO_CLASSE
private static int totCon = 0; //Variável de Classe: total de consultótios
#endregion

#endregion
```

Figura 1: Atributos de Consultório

A seguir aos atributos temos os Métodos:

```
#region METODOS

CONST

PROPRIEDADES

OUTROS

OUTROS

DEST

#endregion
```

Figura 2: Métodos de Consultório

No partido CONST temos 3 funções:

- **Construtor por omissão:** Inicializa propriedades padrão, como número do consultório, especialidade (padrão para pediatria), estado (padrão para ocupado) e incrementa uma variável estática totCon.
- **Construtor personalizado:** Permite a personalização de propriedades como número, especialidade, estado, e associação a um médico e doente específicos. Também incrementa totCon.
- **Construtor estático:** Inicializa uma variável estática totCon. Além disso, há comentários que explicam a função de cada construtor. A região #region CONST é usada para agrupar esses construtores visualmente em ambientes de desenvolvimento integrado (IDE).

```
public Consultorio() //metodo particular
{
    numero = 1;
    especialidade = Especialidade.PEDIATRIA;
    estado = Estado.OCUPADO;
    //medicoResponsavel = ;
    //doenteConsultorio = ;
    totCon++;
}
```

Figura 3: Construtor por Omissão

```
public Consultorio(int num, Especialidade esp, Estado est, Medico medico, Doente doente)
{
    numero = num;
    especialidade = esp;
    estado = est;
    medicoResponsavel = medico;
    doenteConsultorio = doente;
    TotMedicos = 0;
    totCon++;
}
```

Figura 4: Construtor Personalizado

```
/// <summary>
/// construtor estatico que inicializa a variavel totCon
/// </summary>
0 references
static Consultorio()
{
    totCon = 0;
}
```

Figura 5: Construtor Estático

Seguidamente, as propriedades, que estão divididas em propriedades de instância e propriedades de classe, as primeiras são utilizadas para obter ou definir os diversos atributos do consultório, já as últimas permitem obter o número total de consultórios, e é um valor controlado apenas internamente.

Propriedades Instância:

```
public int Numero
{
    //inserir valores
    set
    {
        numero = value;
    }
    //manda para fora valores
    get
    {
        return numero;
    }
}
```

Figura 7: Propriedade do Número

```
public Especialidade Especialidade
{
    set
    {
        especialidade = value;
    }
    get
    {
        return especialidade;
    }
}
```

Figura 6: Propriedade de Especialidade

```
public Estado Estado
{
    set
    {
        estado = value;
    }
    get
    {
        return estado;
    }
}
```

Figura 8: Propriedade de Estado

Propriedades classe:

```
public static int TotCon
{
    get
    {
        return totCon;
    }
    //set{} //valor apenas controlado internamente
}
```

Figura 9: Propriedade Total de Consultórios

No parâmetro OUTROS temos algumas funções:

```
public bool InsereMedicoConsultorio(Medico m1)
{
    //Validações
    if (ExisteMedicoConsultorio(m1)) return false;
    medicoResponsavel = m1;
    TotMedicos++;
    return true;
}
```

Figura 10: Função que Insere Medico no Consultório

A função InsereMedicoConsultorio faz o seguinte:

- Adiciona um médico ao consultório, verificando se o médico já existe no consultório.
- Retorna true se a operação for bem-sucedida, false caso contrário.

```
public bool ExisteMedicoConsultorio(Medico m1)
{
    return medicoResponsavel != null;
}
```

Figura 11: Função que Verifica se Existe Medico no Consultório

A função ExisteMedicoConsultorio faz o seguinte:

- Verifica se um médico já está associado ao consultório.
- Retorna true se o médico existir, false caso contrário.

```
public bool ExisteDoenteConsultorio()
{
    return doenteConsultorio != null;
}
```

Figura 12: Função que Verifica se Existe Doente no Consultório

A função ExisteDoenteConsultorio faz o seguinte:

- Verifica se um doente está associado ao consultório.
- Retorna true se houver um doente, false caso contrário.

```
public void AdicionarMedicoConsultorio(Medico medico)
{
    if (medicoResponsavel == null)
    {
        medicoResponsavel = medico;
        Console.WriteLine("Médico {medico.nome} atribuído ao consultório {numero}.");
    }
    else
    {
        Console.WriteLine("O consultório {numero} já possui um médico atribuído.");
    }
}
```

Figura 13: Função que Adiciona Medico no Consultório

A função AdicionarMedicoConsultorio faz o seguinte:

- Adiciona um médico ao consultório, exibindo mensagens adequadas.
- Atribui o médico ao consultório se não houver um médico associado.

```
public void AdicionarDoenteConsultorio(Doente doente, Medico medico)
{
    if (doenteConsultorio == null)
    {
        if (medicoResponsavel == medico)
        {
            doenteConsultorio = doente;
            Console.WriteLine("Doente {doente.nome} adicionado ao consultório {numero} com o médico {medico.Nome} " +
                "como responsável.");
        }
        else
        {
            Console.WriteLine("O médico {medico.nome} não está atribuído ao consultório {Numero}.");
        }
    }
    else
    {
        Console.WriteLine("O consultório {numero} já possui um doente.");
    }
}
```

Figura 14: Função que Adiciona Doente no Consultório

A função AdicionarDoenteConsultorio faz o seguinte:

- Adiciona um doente ao consultório, exibindo mensagens adequadas.
- Verifica se o médico associado ao doente é o mesmo responsável pelo consultório.
- Atribui o doente ao consultório se as condições forem atendidas.

Por último, o destrutor da classe, utilizado para eliminar a classe construída, para limpar a memória.

```
~Consultorio()
{

}
```

Figura 15: Destrutor de classe do Consultório

SalaaEspera

A classe SalaaEspera contém diversos atributos:

- Uma lista de doentes;
- Capacidade máxima da sala de espera, do tipo inteiro;
- Uma lista de recursos disponíveis.

```
#region Atributos
8 references
public List<Doente> doente { get; set; }
1 reference
public int capacidadeMaxima { get; set; }
1 reference
public List<string> recursosDisponiveis { get; set; }

#endregion
```

Figura 16: Atributos de SalaaEspera

A seguir aos atributos temos os Métodos:

```
#region Metodos

Construtores


Outros

DEST

#endregion
```

Figura 17: Métodos de SalaaEspera

Nos construtores temos este construtor instância:

```
public SalaaEspera(int cpMaxima)
{
    doente = new List<Doente>();
    capacidadeMaxima = cpMaxima;
    estado = Estado.VAZIO;
    recursosDisponiveis = new List<string>();
}
```

Figura 18: Construtor Instância da SalaaEspera

Este construtor é utilizado para criar uma instância da classe SalaEspera com capacidade máxima definida (capacidadeMaxima), estado inicial vazio (Estado.VAZIO), e listas vazias de doentes e recursos disponíveis.

No parâmetro outros temos várias funções:

```
public void AdicionarDoenteSalaEspera(Doente d1)
{
    doente.Add(d1);
}
```

Figura 19: Função que Adiciona Doente na Sala de Espera

A função AdicionarDoenteSalaEspera faz o seguinte:

- Adiciona um objeto Doente à lista de doentes na sala de espera.

```
public int ContarDoentes()
{
    return doente.Count;
}
```

Figura 20: Função que Conta os Doentes

A função ContarDoentes faz o seguinte:

- Retorna o número de doentes presentes na sala de espera.

```
public void MostrarLugaresOcupados()
{
    if (doente.Count > 0)
    {
        Console.WriteLine("Lugares ocupados na sala de espera:");
        foreach (var doente in doente)
        {
            Console.WriteLine($"- {doente.nome}");
        }
    }
    else
    {
        Console.WriteLine("Não há doentes na sala de espera no momento.");
    }
}
```

Figura 21: Função que Mostra Lugares Ocupados

A função MostrarLugaresOcupados faz o seguinte:

- Exibe na console os nomes dos doentes presentes na sala de espera, ou informa que não há doentes.

```
public bool ExisteDoenteSalaEspera()
{
    return doente.Count > 0;
}
```

Figura 22: Função que Verifica se Existe Doentes na Sala de Espera

A função ExisteDoenteSalaEspera faz o seguinte:

- Verifica se há pelo menos um doente na sala de espera.
- Retorna true se há doentes, false se não há.

```
public int QuantosDoentesExistemSalaEspera(Pulseira pulseira)
{
    return doente.Count(p => p.Pulseira == pulseira);
}
```

Figura 23: Função que Conta Quantos Doentes Existem na Sala de Espera

A função QuantosDoentesExistemSalaEspera faz o seguinte:

- Conta quantos doentes na sala de espera possuem uma pulseira específica.
- Retorna o número de doentes com a pulseira especificada.

```
public List<Doente> ObterDoentesPorPrioridade()
{
    return doente.OrderByDescending(p => p.Prioridade).ToList();
}
```

Figura 24: Função que Retorna uma lista de Doentes ordenados por Prioridade

A função ObterDoentesPorPrioridade faz o seguinte:

- Retorna uma lista de doentes na sala de espera ordenados por prioridade, do maior para o menor.

Por último, o destrutor da classe, utilizado para eliminar a classe construída, para limpar a memória.

```
~SalaaEspera()
{ }
```

Figura 25: Destrutor de Classe da Sala de Espera

Sistema

A classe Sistema contém diversos atributos:

- Uma lista de lista de Espera;
- Uma lista de lista de médicos;
- Uma lista de consultórios.

```
private List<Doente> listaDeEspera { get; }
3 references
private List<Medico> listaDeMedicos { get; }
1 reference
private List<Consultorio> consultorios { get; }
```

Figura 26: Atributos de Sistema

A seguir temos várias funções:

```
public Sistema()
{
    listaDeEspera = new List<Doente>();
    listaDeMedicos = new List<Medico>();
}
```

Figura 27: Listas de Doentes e Médicos no Sistema

public Sistema faz o seguinte:

- Inicializa duas listas, uma para doentes (listaDeEspera) e outra para médicos (listaDeMedicos), quando um novo sistema é criado.

```
public void RegistarDoente(int id, string nome, int nif, int nutente, Pulseira pulseira, Prioridade prioridade,
    string historico, string sintomas)
{
    Doente novoDoente = new Doente(id, nome, nif, nutente, pulseira, prioridade, historico, sintomas);
    listaDeEspera.Add(novoDoente);
    Console.WriteLine("Doente {id} registado com pulseira {pulseira} na lista de espera.");

    EncaminharParaTriagem(novoDoente); // Chamando função para encaminhar o paciente para triagem
}
```

Figura 28: Função que Regista um Doente

A função RegistarDoente faz o seguinte:

- Regista um novo doente na lista de espera, criando uma instância da classe Doente com os parâmetros fornecidos.
- Adiciona o doente à lista de espera.
- Exibe uma mensagem informando o registo e encaminha o doente para triagem.

```
public void RegistarMedico(string nomeMedico, Especialidade especialidade, int nidentificacao)
{
    Medico novoMedico = new Medico(nomeMedico, especialidade, nidentificacao);
    listaDeMedicos.Add(novoMedico);
    Console.WriteLine($"Médico {nomeMedico} registado com especialidade {especialidade}.");
}
```

Figura 29: Função que Regista um Médico

A função RegistarMedico faz o seguinte:

- Regista um novo médico na lista de médicos, criando uma instância da classe Medico com os parâmetros fornecidos.
- Adiciona o médico à lista de médicos.
- Exibe uma mensagem informando o registo.

```
private void EncaminharParaTriagem(Doente doente)
{
    switch (doente.Pulseira)
    {
        case Pulseira.VERMELHA:
            Console.WriteLine($"O doente {doente.Nome} (urgência muito alta) foi encaminhado para a triagem prioritária.");
            break;
        case Pulseira.AMARELA:
            Console.WriteLine($"O doente {doente.Nome} (urgência alta) foi encaminhado para a triagem.");
            // Lógica específica para doentes com pulseira laranja (urgente)
            break;
        case Pulseira.VERDE:
            Console.WriteLine($"O doente {doente.Nome} (não urgente) foi encaminhado para a triagem normal.");
            // Lógica específica para doentes com pulseira verde (não urgente)
            break;
        default:
            Console.WriteLine("Pulseira inválida para o doente {doente.Nome}.");
            break;
    }
}
```

Figura 30: Função que Encaminha para a Triagem

A função EncaminharParaTriagem faz o seguinte:

- Função privada que encaminha um doente para triagem com base na sua pulseira.
- Exibe mensagens específicas dependendo da pulseira do doente.

```
public void AtualizarHistorico(int id, string historico)
{
    Doente doente = listaDeEspera.Find(d => d.idDoente == id);

    if (doente != null)
    {
        doente.AtualizarHistorico(historico);
        Console.WriteLine($"Histórico atualizado para o doente {doente.Nome} (ID: {doente.idDoente}).");
    }
    else
    {
        Console.WriteLine($"Doente com ID {id} não encontrado na lista de espera.");
    }
}
```

Figura 31: Função que Atualiza o Historico

A função AtualizarHistorico faz o seguinte:

- Atualiza o histórico médico de um doente na lista de espera com base no ID fornecido.
- Exibe mensagens informativas.

```
public string ObterInformacao(int id)
{

    Doente doente = listaDeEspera.Find(d => d.idDoente == id);

    if (doente != null)
    {
        return doente.ObterInformacao();
    }
    else
    {
        return $"Doente com ID {id} não encontrado na lista de espera.";
    }
}
```

Figura 32: Função que Mostra Informação de um Doente

A função ObterInformação faz o seguinte:

- Retorna informações de um doente na lista de espera com base no ID fornecido.

```
public void AtribuirMedico(int idDoente, int nid)
{
    Doente doente = listaDeEspera.Find(d => d.idDoente == idDoente);
    Medico medico = listaDeMedicos.Find(m => m.nidentificacao == nid);

    if (doente != null && medico != null)
    {
        // Aqui você pode adicionar lógica para atribuir o médico ao paciente, se necessário
        Console.WriteLine($"Médico {medico.Nome} atribuído ao doente {doente.Nome}.");
    }
    else
    {
        Console.WriteLine("Doente ou médico não encontrado.");
    }
}
```

Figura 33: Função que Atruibui um Medico a um Doente

A função AtribuirMedico faz o seguinte:

- Atribui um médico a um doente com base nos IDs fornecidos.
- Exibe mensagens informativas.

```
public void RealizarConsulta(int id, Medico medico)
{
    Doente doente = listaDeEspera.Find(d => d.idDoente == id);

    if (doente != null)
    {
        Consultorio consultorioDisponivel = consultorios.FirstOrDefault(c => c.ExisteMedicoConsultorio(medico) &&
        c.ExisteDoenteConsultorio() == null);

        if (consultorioDisponivel != null)
        {
            Medico medicoResponsavel = consultorioDisponivel.medicoResponsavel;

            // Simulação de realização da consulta
            Console.WriteLine($"Consulta realizada para o doente {doente.nome} no consultório " +
                $"{consultorioDisponivel.numero} com o médico {medicoResponsavel.nome}.");

            // Associar o doente ao consultório após a consulta
            consultorioDisponivel.doenteConsultorio = doente;
        }
        else
        {
            Console.WriteLine("Não há consultórios disponíveis para realizar a consulta.");
        }
    }
    else
    {
        Console.WriteLine($"Doente com ID {id} não encontrado na lista de espera.");
    }
}
```

Figura 34: Função que Realiza uma Consulta

A função RealizarConsulta faz o seguinte:

- Realiza uma consulta para um doente com base no ID fornecido e em um médico.
- Verifica consultórios disponíveis e simula a realização da consulta.
- Exibe mensagens informativas.

```
public void RealizarTriagem(Doente doente)
{
    SalaaEspera salaEspera = new SalaaEspera(200);
    switch (doente.Pulseira)
    {
        case Pulseira.VERMELHA:
            doente.Prioridade = Prioridade.MUITO_URGENTE;
            break;
        case Pulseira.AMARELA:
            doente.Prioridade = Prioridade.URGENTE;
            break;
        case Pulseira.VERDE:
            doente.Prioridade = Prioridade.POUCO_URGENTE;
            break;
        default:
            doente.Prioridade = Prioridade.NAO_ESPECIFICADO;
            break;
    }
    EncaminharParaSalaEspera(doente, salaEspera);

    // Mensagem simulada de triagem
    Console.WriteLine($"Triagem realizada para o paciente {doente.nome}. Prioridade: {doente.prioridade}");

    // Adicionar o doente à lista de espera após a triagem
    listaDeEspera.Add(doente);
}
```

Figura 35: Função que Faz a Triagem

A função RealizarTriagem faz o seguinte:

- Realiza a triagem de um doente, atribuindo prioridade com base na pulseira.
- Encaminha o doente para a sala de espera após a triagem.
- Exibe mensagens simuladas de triagem.

```
public void EncaminharParaSalaEspera(Doente doente, SalaaEspera salaEspera)
{
    // Realizar procedimentos de triagem
    RealizarTriagem(doente);

    // Após a triagem, encaminhar o doente para a sala de espera
    salaEspera.AdicionarDoenteSalaEspera(doente);
    Console.WriteLine($"Doente {doente.nome} encaminhado para a sala de espera.");
}
```

Figura 36: Função que Encaminha para a Sala de Espera

A função EncaminharParaSalaEspera faz o seguinte:

- Encaminha um doente para a sala de espera após realizar a triagem.
- Adiciona o doente à lista de espera e à sala de espera.
- Exibe mensagens informativas.

Local

Esta classe é usada como herança das classes acima referidas (Consultório, SalaaEspera e Sistema). Nela inserimos algumas linhas de código que são as seguintes:

```
public enum Estado
{
    VAZIO,
    OCUPADO,
}
```

Figura 37: Enum Estado

Este excerto de código significa que:

- Define um conjunto de valores possíveis para o estado, como VAZIO e OCUPADO.

```
public class Local
{
    5 references
    public Estado estado { get; set; }
}
```

Figura 38: Class Local (Herança)

Esta classe mostra que:

- Possui uma propriedade chamada estado que pode assumir os valores definidos no enum Estado.
- A relação é de composição, indicando que a classe Local tem um membro do tipo Estado.

Intervenientes

Doente

A classe Doente contém diversos atributos:

- Tem o id do doente, do tipo inteiro;
- NIF do doente, do tipo inteiro;
- Número do utente, do tipo inteiro;
- Pulseira do doente, do tipo pulseira;
- Prioridade, do tipo prioridade;
- Historico medico do doente, do tipo string;
- Sintomas do doente, do tipo string;
- O número total de doentes, do tipo inteiro.

```
#region ESTADO

#region ESTADO_DOENTE
public int idDoente;
public int nif; //numero de identificacao fiscal do doente
public int nutente; //numero de utente do doente
public Pulseira pulseira; //cor da pulseira do doente
public Prioridade prioridade;
public string historico; //historico medico do doente
public string sintomas; //sintomas do doente
#endregion

#region ESTADO_CLASSE
private static int totDoe = 0; //Variável de Classe: total de doente
#endregion

#endregion
```

Figura 39: Atributos de Doente

A seguir dos atributos temos os METODOS:

```
#region METODOS

CONST

PROPRIEDADES

Overrides

OUTROS

DEST
#endregion
```

Figura 40: Métodos de Doente

No parâmetro CONST temos:

```
public Doente() //metodo particular
{
    idDoente = 01;
    nome = "Manel";
    nif = 258369147;
    nutente = 147258369;
    pulseira = Pulseira.VERDE;
    prioridade = Prioridade.POUCO_URGENTE;
    historico = "internado, diabetico";
    sintomas = "tosse";
    totDoe++;
}
```

Figura 41: Construtor por omissão do Doente

Este excerto de código significa que:

- Cria um objeto Doente com valores padrão.
- Inicializa os atributos com valores específicos, como idDoente, nome, nif, etc.
- Incrementa a variável estática totDoe, que pode estar a contar o número total de instâncias da classe.

```
public Doente(int Id, string nome, int nif, int nutente, Pulseira pul, Prioridade pri, string hist, string sint)
{
    idDoente = Id;
    this.nome = nome;
    this.nif = nif;
    this.nutente = nutente;
    pulseira = pul;
    prioridade = pri;
    historico = hist;
    sintomas = sint;
    totDoe++;
}
```

Figura 42: Construtor que permite personalizar os atributos do doente

Este excerto de código significa que:

- Cria um objeto Doente com valores personalizados, permitindo a inicialização específica dos atributos.
- Aceita parâmetros para Id, nome, nif, nutente, pul (pulseira), pri (prioridade), hist (histórico) e sint (sintomas).
- Incrementa a variável estática totDoe.

```
static Doente()
{
    totDoe = 0;
}
```

Figura 43: Construtor estático do atributo Total de Doentes

Seguidamente, as propriedades, que estão divididas em propriedades de instância e propriedades de classe, as primeiras são utilizadas para obter ou definir os diversos atributos do consultório, já as últimas permitem obter o número total de consultórios, e é um valor controlado apenas internamente.

Propriedades Instância:

```
public int IdDoente
{ //inserir valores
    set
    {
        idDoente = value;
    }
    //manda para fora valores
    get
    {
        return idDoente;
    }
}
```

Figura 45: Propriedade do id do Doente

```
public string Nome
{
    //inserir valores
    set
    {
        nome = value;
    }
    //manda para fora valores
    get
    {
        return nome;
    }
}
```

Figura 44: Propriedade do Nome

```
public int Nif
{
    set
    {
        nif = value;
    }
    get
    {
        return nif;
    }
}
```

Figura 47: Propriedade do NIF

```
public int Nutente
{
    set
    {
        nutente = value;
    }
    get
    {
        return nutente;
    }
}
```

Figura 46: Propriedade do Número de Utente

```
public Pulseira Pulseira
{
    set
    {
        pulseira = value;
    }
    get
    {
        return pulseira;
    }
}
```

Figura 49: Propriedade de Pulseira

```
public Prioridade Prioridade
{
    set
    {
        prioridade = value;
    }
    get
    {
        return prioridade;
    }
}
```

Figura 48: Propriedade de Prioridade

```
public string Historico
{
    set
    {
        historico = value;
    }
    get
    {
        return historico;
    }
}
```

Figura 51: Propriedade de Histórico

```
public string Sintomas
{
    set
    {
        sintomas = value;
    }
    get
    {
        return sintomas;
    }
}
```

Figura 50: Propriedade de Sintomas

Propriedades classe:

```
public static int TotDoe
{
    get
    {
        return totDoe;
    }
    //set{} //valor apenas controlado internamente
}
```

Figura 52: Propriedade estática do atributo Total de Doentes

Nos overrides temos uma função tostring:

```
public override string ToString()
{
    return String.Format("Ficha de Doente=> Nome: {0} -  Nif: {1} - N.utente: {2} - Pulseira: {3}- Prioridade: " +
        "{4} - Historico: {5} - Sintomas: {6}\n", nome, nif, nutente, pulseira, prioridade, historico, sintomas);
    //return base.ToString();
}
```

Figura 53: Método ToString() da classe Doente

Este excerto de código significa que:

- Sobrescreve o método ToString() da classe base Object.
- Retorna uma string formatada contendo informações relevantes sobre o objeto Doente, como nome, NIF, número de utente, pulseira, prioridade, histórico e sintomas.
- A formatação é realizada usando a função String.Format().

No parâmetro Outros, temos varias funções:

```
public void AtualizarHistorico(string historico)
{
    this.historico += "\n" + historico;
}
```

Figura 54: Função que Atualiza Histórico

A função AtualizarHistorico faz o seguinte:

- Atualiza o histórico médico do doente, acrescentando informações ao histórico existente.

```
public string ObterInformacao()
{
    return $"ID: {idDoente}, Nome: {nome}, Pulseira: {pulseira}, Prioridade:{prioridade}" +
        $" Histórico Médico: {historico}";
}
```

Figura 55: Função que Mostra Informação do Doente

A função ObterInformacao faz o seguinte:

- Retorna uma string formatada contendo informações essenciais sobre o doente, incluindo ID, nome, pulseira, prioridade e histórico médico.

Por último, o destrutor da classe, utilizado para eliminar a classe construída, para limpar a memória.

```
~Doente()
{
}
```

Figura 56: Destrutor da Class Doente

Medico

A classe Medico contém diversos atributos:

- Especialidade do medico, do tipo especialidade;
- Tem o número de identificação do medico, do tipo inteiro;
- Total de médicos, do tipo inteiro.

```
#region ESTADO

#region ESTADO_MEDICO
public Especialidade especialidade; //especialide do medico
public int nidentificacao; //numero de identificacao do medico dentro do hospital
#endregion

#region ESTADO_CLASSE
private static int totMed = 0; //Variável de Classe: total de médicos
#endregion

#endregion
```

Figura 57: Atributos de Médico

A seguir dos atributos temos os METODOS:

```
#region METODOS
CONST

PROPRIEDADES


Overrides

DEST

#endregion
```

Figura 58: Métodos de Médico

No parâmetro CONT temos:

```
public Medico() //metodo particular
{
    nome = "Joaquim";
    especialidade = Especialidade.PEDIATRIA;
    nidentificacao = 25;
    totMed++;
}
```

Figura 59: Construtor por omissão de Médico

Este excerto de linhas de código significa que:

- Inicializa os atributos de um médico com valores específicos, como nome, especialidade e número de identificação.
- Incrementa a variável estática totMed, que pode estar a contar o número total de instâncias da classe.

```
public Medico(string nome, Especialidade esp, int nidentificacao)
{
    this.nome = nome;
    especialidade = esp;
    this.nidentificacao = nidentificacao;
    totMed++;
}
```

Figura 60: Construtor que permite a personalização dos atributos do Médico

Este excerto de linhas de código significa que:

- Aceita parâmetros para nome, esp (especialidade) e nidentificacao (número de identificação).
- Incrementa a variável estática totMed.

```
static Medico()
{
    totMed = 0;
}
```

Figura 61: Construtor estático do atributo Total de Médicos

Este excerto de linhas de código significa que:

- Um construtor estático é chamado automaticamente antes de qualquer instância da classe ser criada ou qualquer membro estático ser acessado.

Seguidamente, as propriedades, que estão divididas em propriedades de instância e propriedades de classe, as primeiras são utilizadas para obter ou definir os diversos atributos do consultório, já as últimas permitem obter o número total de consultórios, e é um valor controlado apenas internamente.

Propriedades instância:

```
public string Nome
{
    //inserir valores
    set
    {
        nome = value;
    }
    //manda para fora valores
    get
    {
        return nome;
    }
}
```

Figura 63: Propriedade do Nome

```
public Especialidade Especialidade
{
    set
    {
        especialidade = value;
    }
    get
    {
        return especialidade;
    }
}
```

Figura 62: Propriedade de Especialidade

```
public int NIfentificacao
{
    set
    {
        nidentificacao = value;
    }
    get
    {
        return nidentificacao;
    }
}
```

Figura 64: Propriedade do Número de Identificação

Propriedades classe:

```
public static int TotMed
{
    get
    {
        return totMed;
    }
    //set{} //valor apenas controlado internamente
}
```

Figura 65: Propriedade estática do atributo Total de Médicos

Nos overrides temos:

```
public override string ToString()
{
    return String.Format("Ficha de Medico=> Nome: {0} - Especialidade: {1} - Número de identificacao:" +
        " {2}\n", nome, especialidade, nidentificacao);
    //return base.ToString();
}
```

Figura 66: Método ToString() na classe do Médico

Este excerto de código ToString significa que:

- Sobrescreve o método ToString() da classe base Object.
- Retorna uma string formatada contendo informações relevantes sobre o objeto Medico, como nome, especialidade e número de identificação.
- A formatação é realizada usando a função String.Format().

Por último, o destrutor da classe, utilizado para eliminar a classe construída, para limpar a memória.

```
~Medico()
{

}
```

Figura 67: Destrutor de class de Médico

Pessoa

Esta classe é usada como herança das classes acima referidas (Doente e Medico). Nela inserimos algumas linhas de código que são as seguintes:

```
public string nome { get; set; }
```

Figura 68: Atributo de Pessoa (Herança)

Atributo de Doente e Medico.

Resultados

O desenvolvimento do sistema de gestão de urgências hospitalares baseado em Programação Orientada a Objetos (POO) resultou na criação de classes como Doente, Médico, Pessoa, Local, Sala de Espera, Consultório e Sistema. Utilizando as propriedades e métodos destas classes, foi possível modelar eficazmente a interação entre doentes, médicos, salas de espera e outros elementos fundamentais para a gestão de urgências hospitalares. A implementação dos enums Pulseira, Prioridade, Estado e Especialidade ofereceu uma abordagem flexível para classificar e priorizar casos médicos, contribuindo para a eficiência do sistema.

O sistema permitiu a criação, registo e atribuição de médicos e doentes, simulação de consultas, triagem e encaminhamento para salas de espera, demonstrando uma capacidade abrangente de gerir eventos urgentes num ambiente hospitalar. A integração das classes, construtores e métodos refletiu uma estrutura coesa e modular, facilitando a manutenção e expansão do sistema no futuro.

Conclusão

Através da implementação do sistema de gestão de urgências hospitalares em C# utilizando POO, conseguimos desenvolver uma solução robusta e flexível para lidar com emergências médicas. A modelagem orientada a objetos proporcionou uma representação clara e estruturada dos elementos do sistema, criando assim uma base sólida para operações hospitalares em tempo real.

Os construtores e métodos personalizados nas classes Doente e Médico possibilitaram a criação de instâncias com valores padrão ou personalizados, proporcionando adaptabilidade ao contexto específico do ambiente hospitalar. A utilização de enums como Pulseira e Prioridade trouxe uma semântica clara e consistente na categorização e priorização de casos médicos.

Os resultados evidenciam uma implementação bem-sucedida do sistema, destacando a eficiência na gestão de doentes, a atribuição de médicos e a realização de consultas. A estrutura modular do código promove a extensibilidade e manutenção, assegurando a capacidade de adaptação às necessidades futuras do hospital. Este projeto representa não apenas uma solução funcional, mas também um exemplo de boas práticas de programação orientada a objetos na construção de sistemas complexos.