Curso Java Básico

Uma introdução prática usando BlueJ

Objetos bem comportados

Principais conceitos a serem abordados

- Testes e depuração
- Automação de testes

Erros lógicos

- Erros de sintaxe são facilmente identificados pelo compilador.
- Erros de lógica não impedem que o programa compile e execute, mas acarretam resultados errados que podem passar desapercebidos.
  - ***Nunca é cedo demais para iniciar os testes !***

Testando e depurando

- Testar e depurar são habilidades cruciais para o desenvolvedor.
- Teste pesquisa a existência de erros
  - “Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence” (Dijkstra, 1972).
- Depuração pesquisa a causa de erros
  - a manifestação de um erro pode ocorrer bem longe de sua causa

Teste de unidade

- Teste de sistema
  - testa a aplicação completa
- Teste de unidade
  - testa partes individuais de uma aplicação (uma classe, um método, etc...)

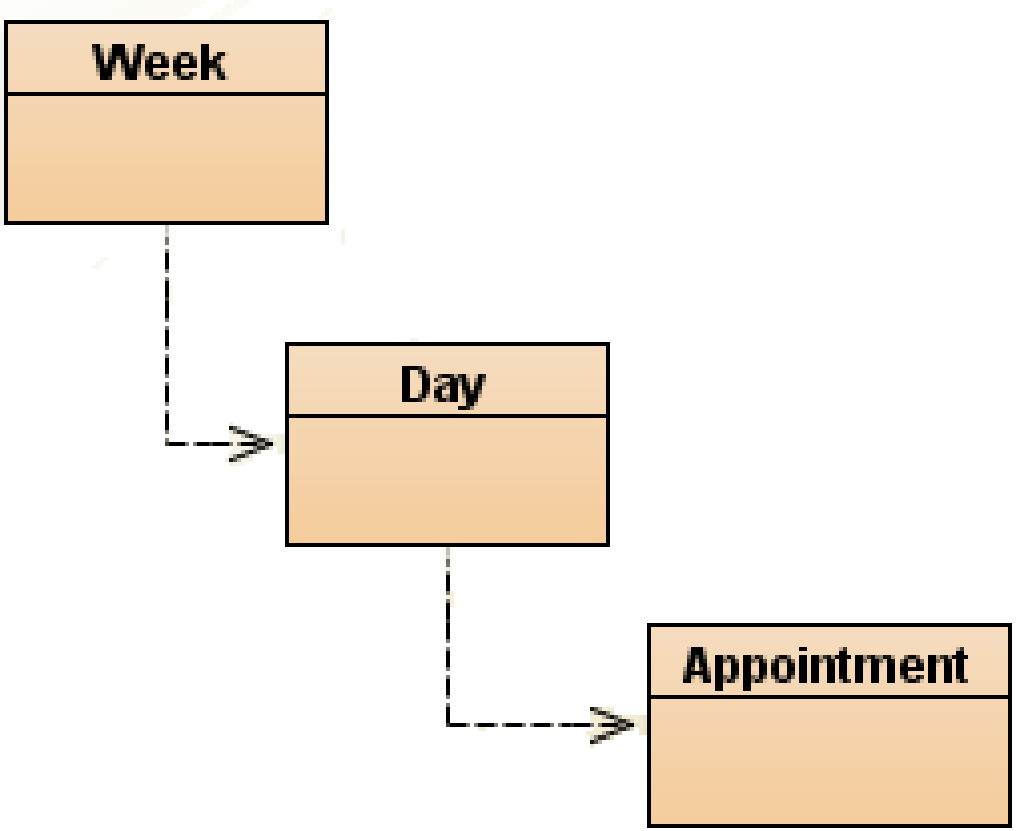
Teste de unidade

- Cada unidade de uma aplicação pode ser testada.
  - Método, classe, módulo (pacote no Java).
- Pode e deve ser feito durante o desenvolvimento.
  - Encontrar e corrigir no início reduz os custos do desenvolvimento

Uma agenda eletrônica

- Usaremos uma aplicação que simula o estágio inicial do desenvolvimento de uma agenda eletrônica onde:
  - Compromissos duram horas inteiras
  - Agendamentos podem ser feitos em horas cheias entre 09:00 e 17:00

Diagrama de classe



Exercício

- **Agenda eletrônica**
  - Inicie o **BlueJ**, abra o projeto *diary-prototype* e crie uma instância de *Day* (informe dia número 1).
  - Crie duas instâncias de *Appointment* (uma reunião de 1 hora e uma palestra de 2). Use o método *makeAppointment* de *Day* para agendar os compromissos.
  - Inspecione a instância de *Day* para verificar os agendamentos e use o método *showAppointments* para listá-los.

Código-fonte: Day

```
public class Day
{
    public static final int START_OF_DAY = 9;
    public static final int FINAL_APPOINTMENT_TIME = 17;
    public static final int MAX_APPOINTMENTS_PER_DAY =
                                    FINAL_APPOINTMENT_TIME -
                                    START_OF_DAY + 1;
    private int dayNumber;
    private Appointment[] appointments;

    public Day(int dayNumber)
    {
        this.dayNumber = dayNumber;
        appointments = new
              Appointment[MAX_APPOINTMENTS_PER_DAY];
    }
    ...
}
```

Fundamentos de testes

- Entender o que a unidade deve fazer, isto é, seu ‘contrato’ e ...
- ... procurar violações.
- Testar os limites:
  - zero, um, cheio.
- Usar:
  - testes positivos (esperamos que funcionem)
  - testes negativos (esperamos que falhem)

Exercício

- **Agenda eletrônica**
  - Crie mais um *Appointment*.
  - Verifique se é possível agendar este compromisso para o mesmo horário de outro compromisso.
  - Verifique se é possível agendar este compromisso para 08:00 ou para 18:00.

Testes de regressão

- Tanto no desenvolvimento quanto na manutenção de software, alterações podem introduzir erros em unidades que estavam funcionando.
- Teste de regressão:
  - consiste em aplicar à versão mais recente do software um conjunto de testes que estavam passando antes, para garantir que não surgiram defeitos em componentes já testados.

Automação de testes

- Uma razão para o teste completo ser frequentemente negligenciado é que, se for manual, ele é demorado e enfadonho.
- Isto torna-se um problema ainda maior se tiver que ser executado diversas vezes.
- Felizmente há técnicas que nos permitem automatizar os testes.

Automação de testes

- Uma das maneiras mais fáceis de automatizar testes de regressão é escrever um programa que funcione como uma amarração de testes (test ring).
- Dentro da classe de amarração, cada método é escrito para representar um único teste.

Código-fonte: OneHourTests

```
public class OneHourTests
{
    private Day day;

    public OneHourTests()
    {
        day = new Day(1);
    }
    ...
}
```

Código-fonte: OneHourTests

```
public void makeThreeAppointments()
{
    day = new Day(1);
    Appointment first =
          new Appointment("Java lecture", 1);
    Appointment second =
          new Appointment("Java class", 1);
    Appointment third =
          new Appointment("Meet John", 1);
    day.makeAppointment(9, first);
    day.makeAppointment(13, second);
    day.makeAppointment(17, third);
    day.showAppointments();
}
```

Código-fonte: OneHourTests

```
public void testDoubleBooking()
{
    makeThreeAppointments();
    Appointment badAppointment = new
        Appointment("Error", 1);
    day.makeAppointment(9, badAppointment);
    day.showAppointments();
}
```

Exercício

- **Agenda eletrônica**
  - Feche o projeto anterior, abra o projeto *diary-testing* e crie uma instância de *OneHourTests*.
  - Chame o método *testDoubleBooking* e verifique se algum compromisso foi substituído.
  - Adicione outros métodos para testar compromissos à classe *OneHourTests*.
  - Crie uma classe *TwoHourTests* para conter um conjunto de testes de compromissos de duas horas.

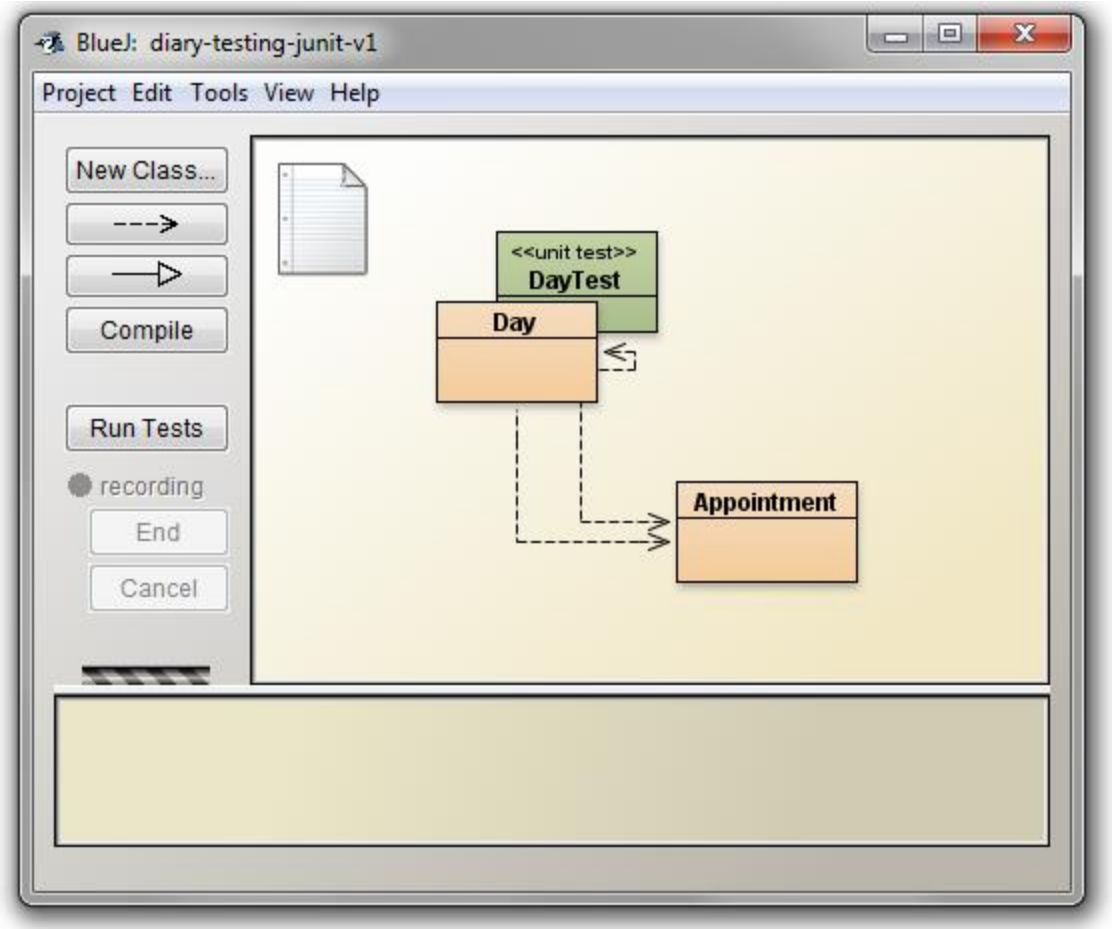
Verificação automática de resultados de teste

- O teste de regressão automatizado seria ainda mais eficaz se também automatizasse a verificação dos resultados.

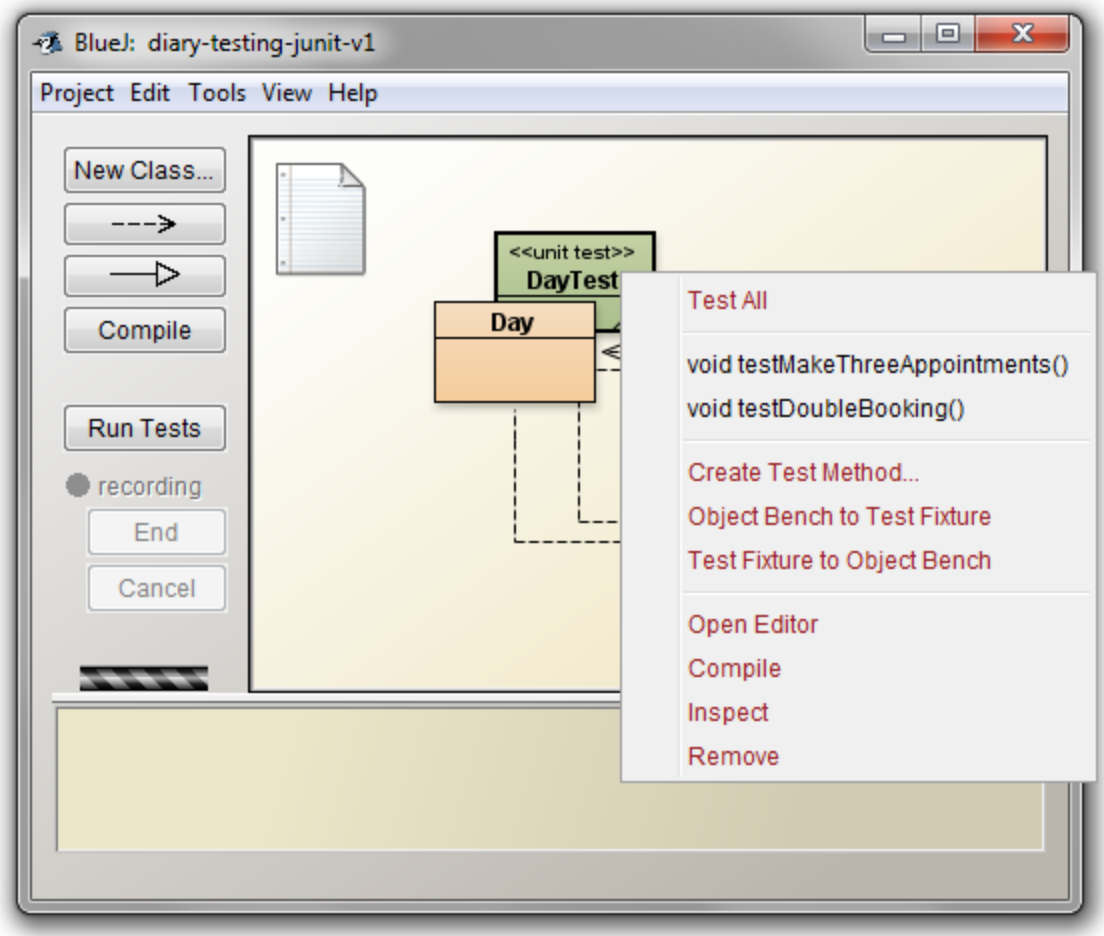
Facilidades de testes

- O **BlueJ** pode criar classes de teste, com estereótipo `<<unit test>>`, baseadas na estrutura de teste do framework JUnit.
- Para usar as facilidades de teste no BlueJ, selecione `show unit testing tools` no menu Tools, opção Preferences, aba Miscellaneous.

Facilidades de testes



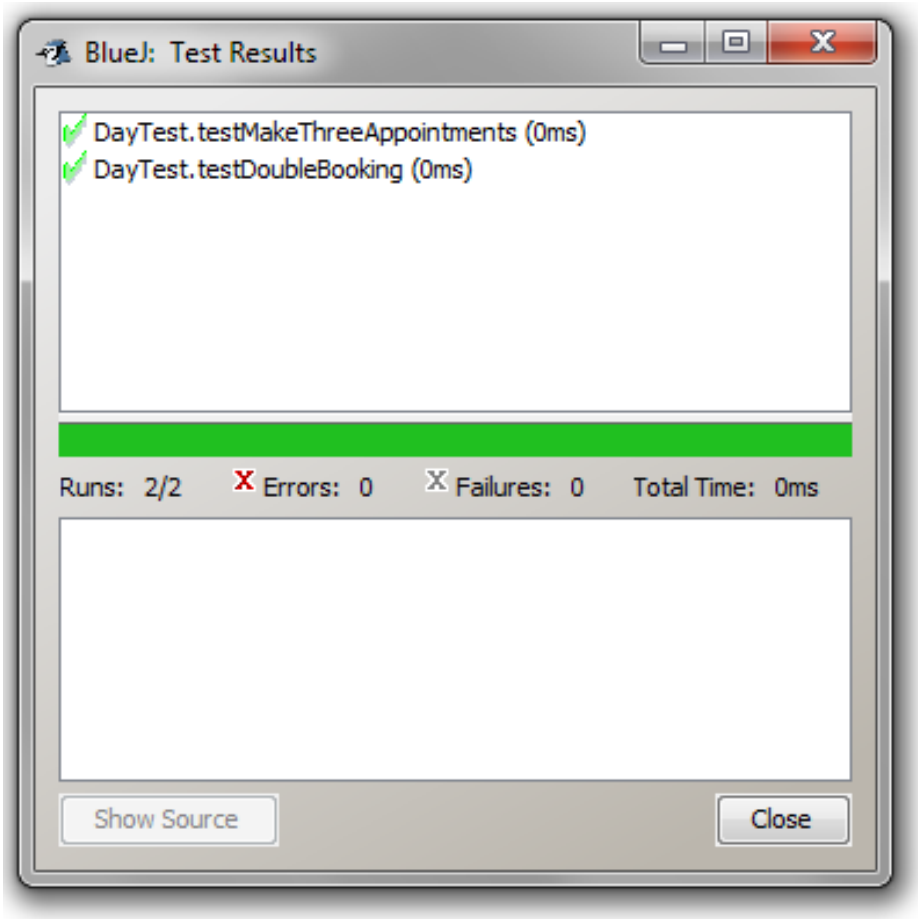
Facilidades de testes



Exercício

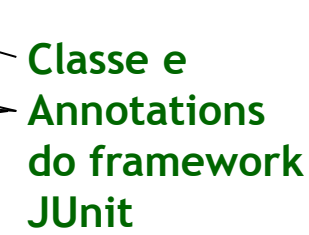
- **Agenda eletrônica**
  - Feche o projeto anterior, abra o projeto *diary-testing-junit-v1* e salve-o como *diary-testing-junit*.
  - Observe a classe de teste DayTest.
  - Selecione *show unit testing tools* no menu Tools, opção Preferences, aba Miscellaneous.
  - Execute os testes automatizados com o botão Run Tests e veja os resultados.
  - Crie uma classe de teste para *Appointment*. Que métodos existem nela?

Resultados dos testes



Código-fonte: DayTest

```
import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class DayTest
{
    public DayTest()
    {
    }

    ...
```



Código-fonte: DayTest

```
    @Before
    protected void setUp()
    {
    }

    @After
    protected void tearDown()
    {
    }
    ...
}
```

Indica método a ser executado antes de cada teste anotado com @Test

Indica método a ser executado após cada teste anotado com @Test

Código-fonte: DayTest

```
@Test
public void testMakeThreeAppointments()
{
  Day day1 = new Day(1);
  Appointment appointm1 =
       new Appointment("Java lecture", 1);
  Appointment appointm2 =
       new Appointment("Java class", 1);
  Appointment appointm3 =
       new Appointment("Meet John", 1);
  assertEquals(true, day1.makeAppointment(9, appointm1));
  assertEquals(true, day1.makeAppointment(13, appointm2));
  assertEquals(true, day1.makeAppointment(17, appointm3));
}
```

Assertiva

Retorno esperado

Chamada de método

Código-fonte: DayTest

```
@Test
public void testDoubleBooking()
{
  Day day1 = new Day(1);
  Appointment appointm1 =
       new Appointment("Java lecture", 1);
  Appointment appointm2 =
       new Appointment("Error", 1);
  assertEquals(true, day1.makeAppointment(9, appointm1));
  assertEquals(false, day1.makeAppointment(9, appointm2));
}
```

Assertiva

Retorno esperado

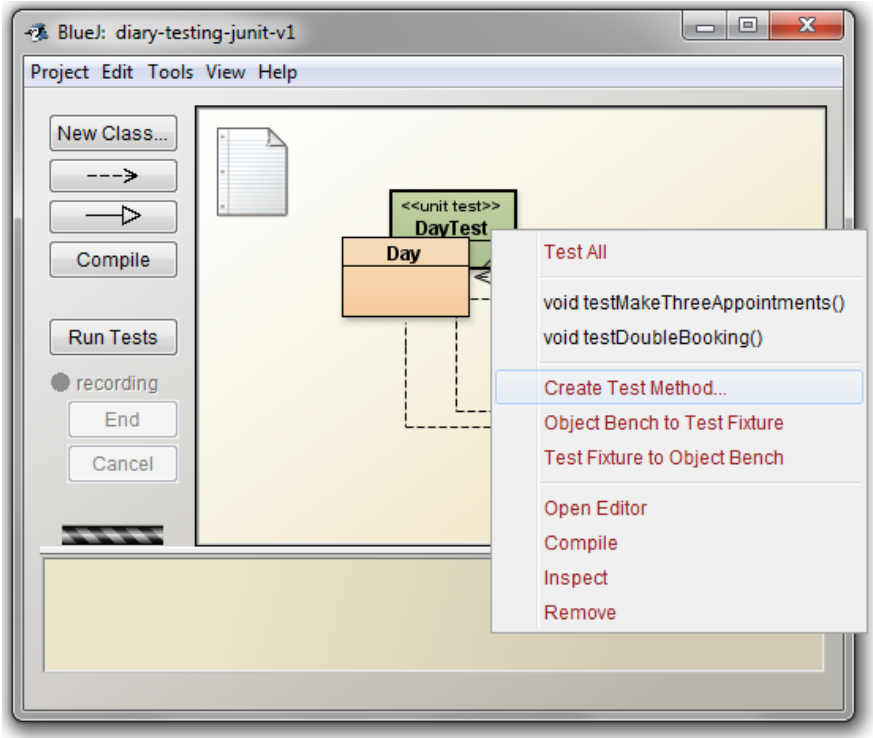
Chamada de método

Anotações

- Anotações são dados sobre o programa que não fazem parte do programa em si e não tem efeito direto sobre sua operação.
- Estes metadados são usados pelo compilador e diversos frameworks:
  - JUnit 4:
    - @Before, @After, @Test, ...
  - JPA e suas implementações:
    - @Entity, @Table, @Column, @Id, ...

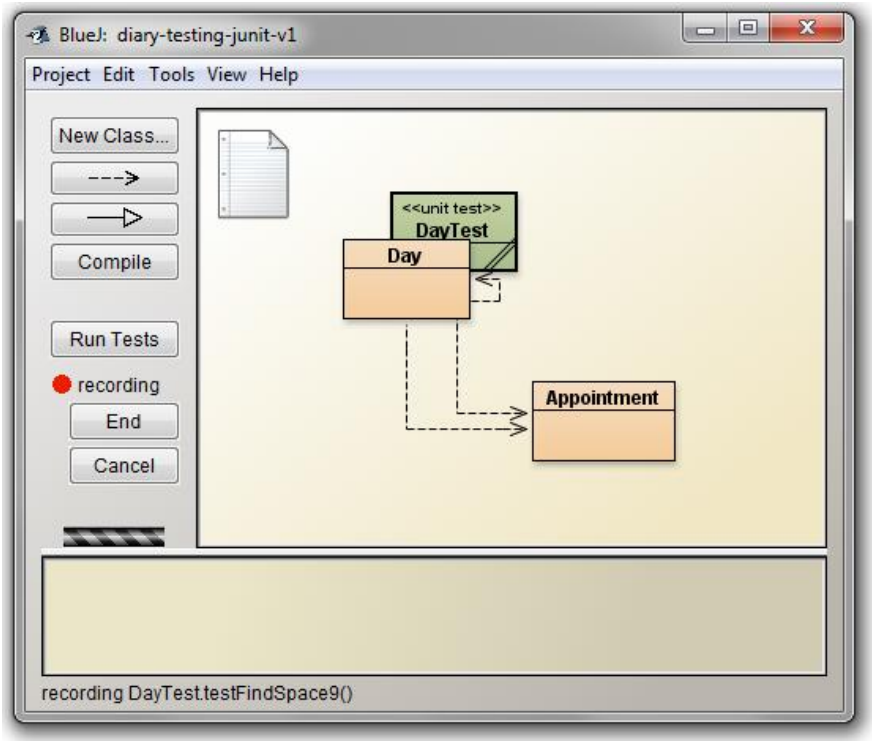
Registrando um teste

- No BlueJ, criamos um método de teste com a opção *Create Test Method do menu pop-up de uma classe de teste*.



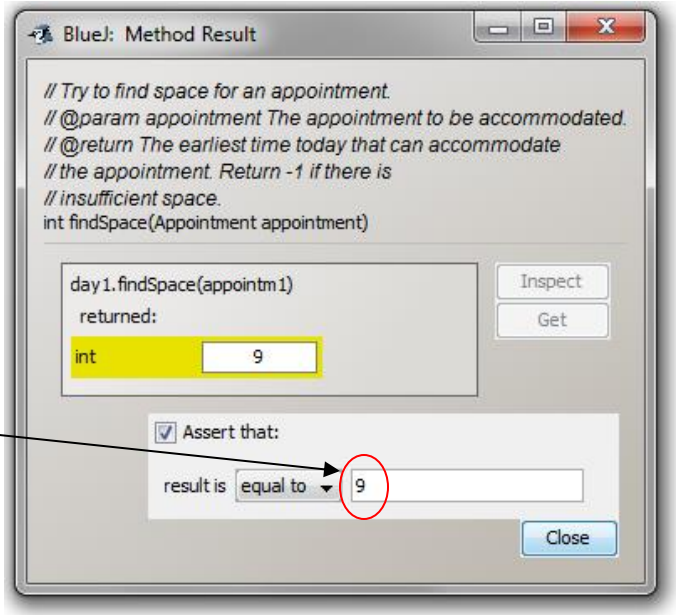
Registrando um teste

- Após informar um nome para o método de teste, o BlueJ gravará as ações feitas até que o botão *End* seja pressionado.



Registrando um teste

- Durante a gravação das ações, para cada chamada de método deve ser informado o retorno esperado, de modo a montar uma assertiva no método de teste.



Retorno esperado

Exercício

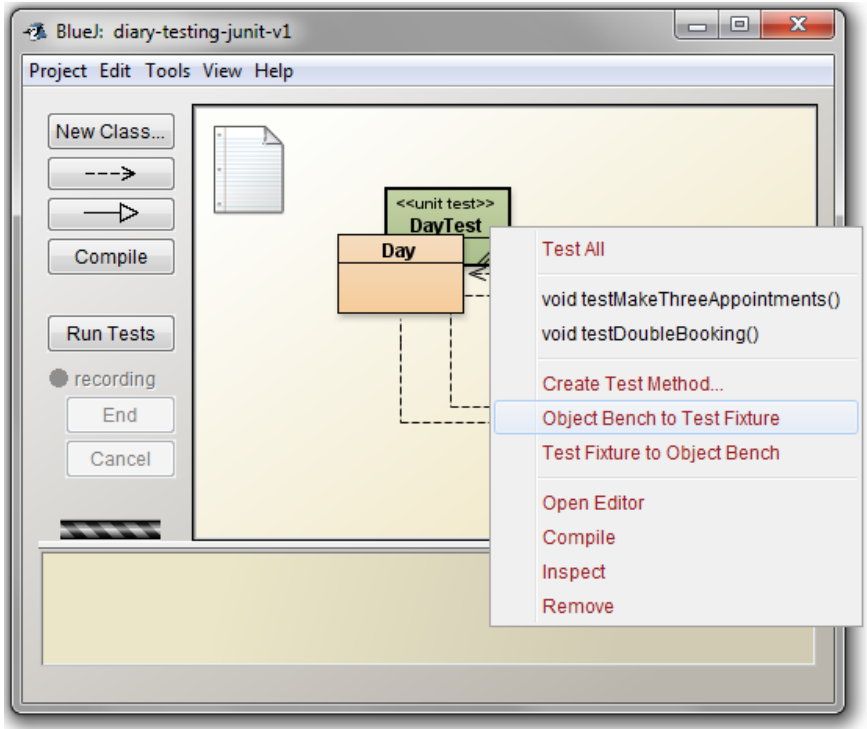
- **Agenda eletrônica**
  - Crie em *DayTest* um método de teste *findSpace9* e grave as seguintes ações:
    - crie um objeto *Day* para o dia 1
    - crie um objeto *Appointment* com duração de uma hora
    - chame o método *findSpace* no objeto *Day*, informe como parâmetro o objeto *Appointment* e informe que o retorno deve ser igual a 9
  - Edite *DayTest* e veja o teste criado.

Usando *fixtures*

- À medida que um conjunto de métodos de testes é construído, é comum criar objetos semelhantes para cada um desses métodos.
  - Cada teste da classe `Day` envolverá pelo menos um objeto `Day` e um ou mais objetos `Appointment`.

Usando *fixtures*

- No BlueJ, podemos criar um conjunto de objetos reusáveis com a opção *Object Bench to Test Fixture*.

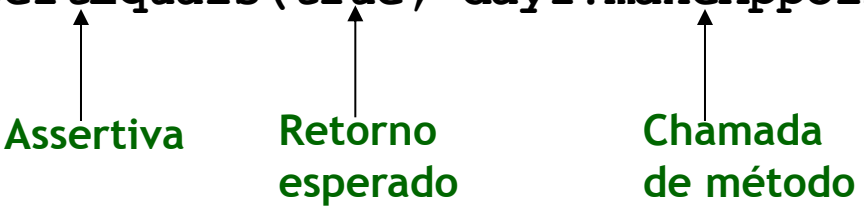


Código-fonte: DayTest

```
public class DayTest
{
    private Day day1;
    private Appointment appointm1;
    private Appointment appointm2;
    private Appointment appointm3;
    ...
    @After
    protected void setUp()
    {
        day1 = new Day(1);
        appointm1 = new Appointment("Java lecture", 1);
        appointm2 = new Appointment("Java class", 1);
        appointm3 = new Appointment("Meet John", 1);
    }
    ...
}
```

Código-fonte: DayTest

```
@Test
public void makeThreeAppointments()
{
  assertEquals(true, day1.makeAppointment(9, appointm1));
  assertEquals(true, day1.makeAppointment(13, appointm2));
  assertEquals(true, day1.makeAppointment(17, appointm3));
}
```



Código-fonte: DayTest

```
@Test
public void doubleBooking()
{
  assertEquals(true, day1.makeAppointment(9, appointm1));
  assertEquals(false, day1.makeAppointment(9, appointm2));
}
```

Exercício

- **Agenda eletrônica**
  - Crie uma instância de *Day* (informe dia número 1).
  - Crie três instâncias de *Appointment* com 1 hora (informe "Java lecture", "Java class" e "Meet John").
  - Mova os objetos da bancada para a fixture de teste.
  - Observe a classe de teste DayTest.

Exercício

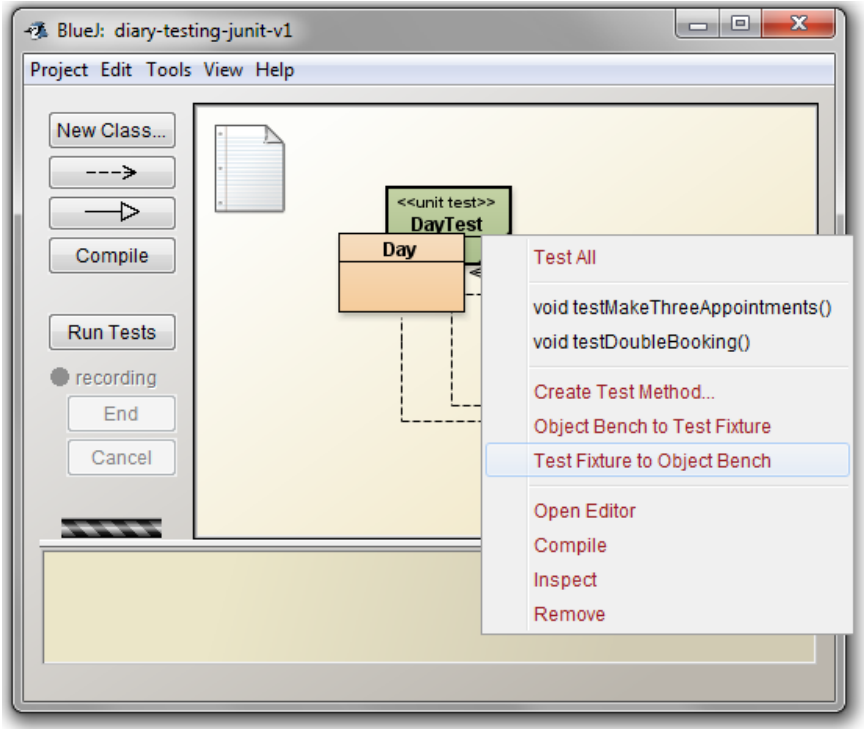
- **Agenda eletrônica**
  - Retire dos métodos de teste a criação e inicialização de objetos que estão na fixture de teste, deixando apenas as instruções **assert** que os usam.
  - Compile o projeto *diary-testing-junit*.
  - Execute todos os testes.
  - Compare o estágio atual do projeto com *diary-testing-junit-v2*.

Exercício

- **Agenda eletrônica**
  - Crie em *DayTest* um método de teste *findSpace10* e grave as seguintes ações:
    - chame o método *makeAppointment* no objeto *Day*, informe como parâmetro um objeto *Appointment* de 1 hora de duração com agendamento para 9 horas e informe que o retorno deve ser igual a *true*
    - chame o método *findSpace* no objeto *Day*, informe como parâmetro outro objeto *Appointment* e informe que o retorno deve ser igual a 10
  - Edite *DayTest* e veja o teste criado.
  - Execute todos os testes.

Usando *fixtures*

- A opção *Test Fixture to Object Bench* permite trazer os objetos de volta para a bancada (para excluir ou incluir mais objetos).



Exercício

- **Agenda eletrônica**
  - Mova os objetos da fixture de teste para a bancada.
  - Crie uma instância de *Appointment* com 2 horas (informe "Business Lunch").
  - Crie uma instância de *Appointment* com 5 horas (informe "Java workshop").
  - Mova os objetos da bancada para a fixture de teste.

Exercício

- **Agenda eletrônica**
  - Crie em *DayTest* um método de teste *busyDay* com as ações:
    - chame 4 vezes o método *makeAppointment*, informando um *Appointment* de 1 hora para 9h, outro de 1 hora para 10h, um de 2 horas para 12h e um de 5 horas para 13h, todos com retorno igual a *true*
    - chame o método *findSpace* no objeto *Day*, informe como parâmetro um objeto *Appointment* de 1 hora e informe que o retorno deve ser igual a -1
  - Edite *DayTest* e veja o teste criado.
  - Execute todos os testes.

Exercício

- **Agenda eletrônica**
  - Execute todos os testes.
  - Compare o estágio atual do projeto com *diary-testing-junit-v3*.

Exercício

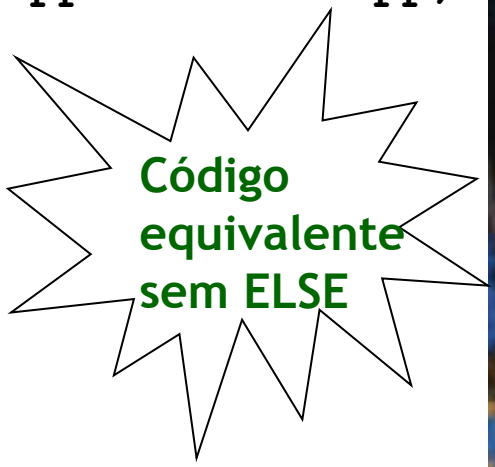
- **Agenda eletrônica**
  - Crie em *DayTest* um método de teste *notOverlapEndOfDay* com as ações:
    - chame o método *makeAppointment*, informe como parâmetro um *Appointment* de 5 hora agendado para 16 horas e informe que o retorno deve ser igual a *false*
  - Inspecione o método *makeAppointment*, faça as alterações necessárias para evitar a exceção e crie o método de teste *notOverlapEndOfDay*.
  - Execute todos os testes.
  - Compare o estágio atual do projeto com *diary-testing-junit-v4*.

Código-fonte: Day

```
public boolean makeAppointment(int time, Appointment app)
{
    if(validTime(time)) {
        int startTime = time - START_OF_DAY;
        if(appointments[startTime] == null) {
            int duration = appointment.getDuration();
            for(int i = 0; i < duration; i++) {
                appointments[startTime + i] = appointment;
            }
            return true;
        }
        else {
            return false;
        }
    }
    else {
        return false;
    }
}
```

Código-fonte: Day

```
public boolean makeAppointment(int time, Appointment app)
{
    if(!validTime(time)) {
        return false;
    }
    int startTime = time - START_OF_DAY;
    if(appointments[startTime] != null) {
        return false;
    }
    int duration = appointment.getDuration();
    for(int i = 0; i < duration; i++) {
        appointments[startTime + i] = appointment;
    }
    return true;
}
```



Código-fonte: Day

```
public boolean makeAppointment(int time, Appointment app)
{
    if(!validTime(time)) {
        return false;
    }
    int duration = appointment.getDuration();
    int endTime = time + duration -1;
    if(endTime > FINAL_APPOINTMENT_TIME) {
        return false;
    }
    int startTime = time - START_OF_DAY;
    if(appointments[startTime] != null) {
        return false;
    }
    for(int i = 0; i < duration; i++) {
        appointments[startTime + i] = appointment;
    }
    return true;
}
```

Verifica se o fim do compromisso está dentro do horário permitido

Exercício

- **Agenda eletrônica**
  - Crie em *DayTest* um método de teste *notOverlapBooking* com as ações:
    - chame o método *makeAppointment*, informe como parâmetro um *Appointment* de 1 hora agendado para 10 horas e informe que o retorno deve ser igual a *true*
    - chame o método *makeAppointment*, informe como parâmetro o *Appointment* de 2 hora agendado para 9 horas e informe que o retorno deve ser igual a *false*
  - Edite *DayTest* e veja o teste criado.
  - Execute todos os testes.

Exercício

- **Agenda eletrônica**
  - Inspecione o método *makeAppointment* e faça as alterações necessárias para corrigir o erro.
  - Execute todos os testes.
  - Compare o estágio atual do projeto com *diary-testing-junit-v5*.

Código-fonte: Day

```
public boolean makeAppointment(int time, Appointment app)
{
    if(!validTime(time)) { return false; }
    int duration = appointment.getDuration();
    int endTime = time + duration -1;
    if(endTime > FINAL_APPOINTMENT_TIME) {
        return false;
    }
    int startTime = time - START_OF_DAY;
    for(int i = 0; i < duration; i++) {
        if(appointments[startTime + i] != null) {
            return false;
        }
    }
    for(int i = 0; i < duration; i++) {
        appointments[startTime + i] = appointment;
    }
    return true;
}
```

Verifica se toda a duração do compromisso está disponível

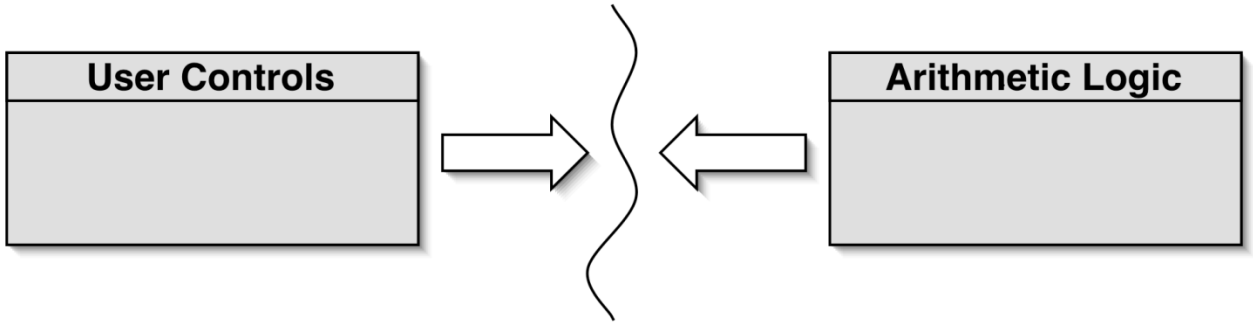
Test First Programming

- Quando o teste é criado antes do código, este é mais conciso e focado.
- Criar o teste antes do código ajuda o desenvolvedor a considerar o que ele realmente precisa codificar.
- Criar primeiro o código pode produzir um software não testável.
- Criar o teste antes conduz a um design de software testável.

Modularização e interfaces

- Aplicações frequentemente consistem em diferentes módulos.
  - por exemplo, para que diferentes equipes possam trabalhar nelas.
- A *interface* entre os módulos precisa ser especificada de maneira clara.
  - suporta desenvolvimento concorrente e independente.
  - aumenta a probabilidade de uma integração bem-sucedida.

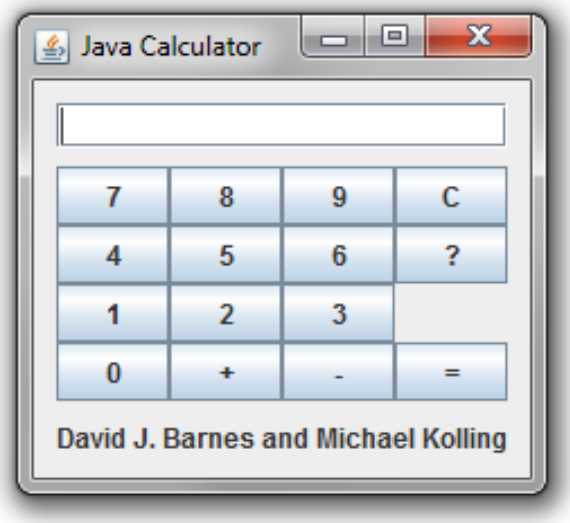
Modularização em uma calculadora



- Cada módulo não precisa conhecer os detalhes de implementação do outro.

Assinaturas de métodos como uma interface

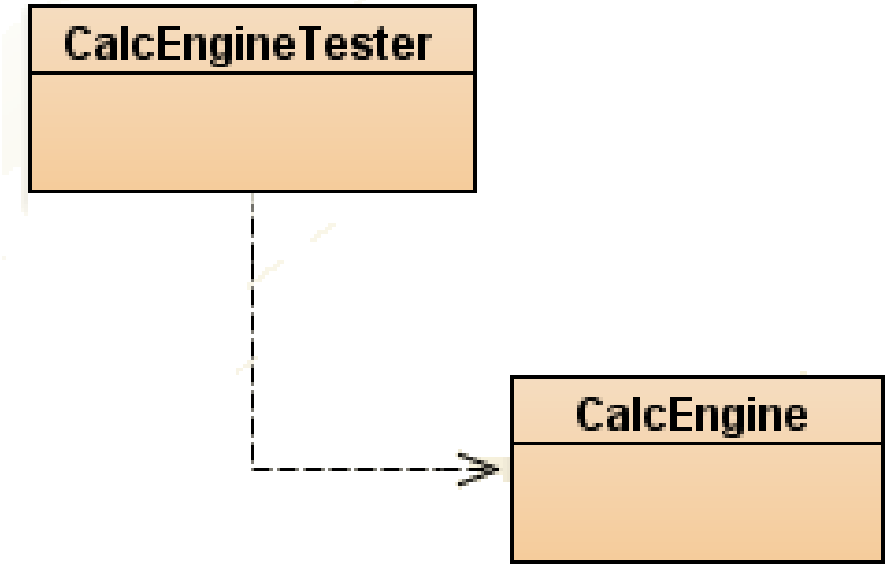
```
// Retorna o valor a ser exibido.
public int getDisplayValue();

// Chamada para quando um botão de
// dígito é pressionado.
public void numberPressed(int number);

// Chamada para quando um operador de
// adição é pressionado.
public void plus();

// Chamada para quando um operador de
// subtração é pressionado.
public void minus();

// Chamada para completar um cálculo.
public void equals();

// Chamada para redefinir a calculadora.
public void clear();
```



Uma calculadora eletrônica

- Usaremos uma aplicação que simula uma calculadora eletrônica com apenas adição e subtração.
- Você foi recrutado para substituir o analista Hacker T.Largebrain, que declarou já ter implementado e testado o módulo de lógica aritmética.
- Antes de documentar a classe, você decidiu explorar seu comportamento.

Diagrama de classe



Exercício

- **Calculadora**
  - Feche o projeto anterior, abra o projeto *calculator-engine-testing* e salve-o como *calculator-engine*.
  - Crie uma instância de *CalcEngineTester*.
  - Chame o método *testAll*. O que você acha da última linha impressa ?
  - Chame o método *testPlus*. O resultado é o mesma impresso no exercício acima ?
  - Chame novamente o método *testPlus*. O resultado se repete ?
  - Faça o mesmo com o método *testMinus*.

Código-fonte: CalcEngineTester

```
public class CalcEngineTester
{
    private CalcEngine engine;

    public CalcEngineTester()
    {
        engine = new CalcEngine();
    }
    ...
}
```

Código-fonte: CalcEngineTester

```
public int testPlus()
{
    // Make sure the engine is in a valid starting state.
    engine.clear();
    // Simulate the presses: 3 + 4 =
    engine.numberPressed(3);
    engine.plus();
    engine.numberPressed(4);
    engine.equals();
    // Return the result, which should be 7.
    return engine.getDisplayValue();
}
```

Código-fonte: CalcEngineTester

```
public int testMinus()
{
    // Make sure the engine is in a valid starting state.
    engine.clear();
    // Simulate the presses: 9 - 4 =
    engine.numberPressed(9);
    engine.minus();
    engine.numberPressed(4);
    engine.equals();
    // Return the result, which should be 5.
    return engine.getDisplayValue();
}
```

Código-fonte: CalcEngineTester

```
public void testAll()
{
 System.out.println("Testing the addition operation.");
 System.out.println("The result is: " + testPlus());
 System.out.println("Testing the subtraction operation.");
 System.out.println("The result is: " + testMinus());
 System.out.println("All tests passed.");
}
```

Exercício

- **Calculadora**
  - Crie uma classe de teste para a classe *CalcEngine*.
  - Crie uma instância de *CalcEngine* e a coloque na fixture.
  - Crie um método de teste com as ações de *testPlus* duplicadas e outro com as ações de *testMinus* duplicadas.
  - Execute todos os testes.

Exercício

- **Calculadora**
  - Feche o projeto anterior, abra o projeto *calculator-full-solution*.
  - Crie uma classe de teste para a classe *CalcEngine* com os mesmos métodos de teste usados em *calculator-engine*.
  - Execute todos os testes.

Revisão

- Erros são fatos reais nos programas.
- Boas técnicas de engenharia de software podem reduzir a ocorrência de erros.
- Habilidade de teste e depuração são essenciais.
- Torne o teste um hábito.
- Automatize o teste quando possível.

Contatos

Câmara dos Deputados

CENIN - Centro de Informática

Carlos Renato S. Ramos

carlosrenato.ramos@camara.gov.br