Platformy Programistyczne Java

Laboratorium 5 - Wielkie porządki

Plan laboratorium

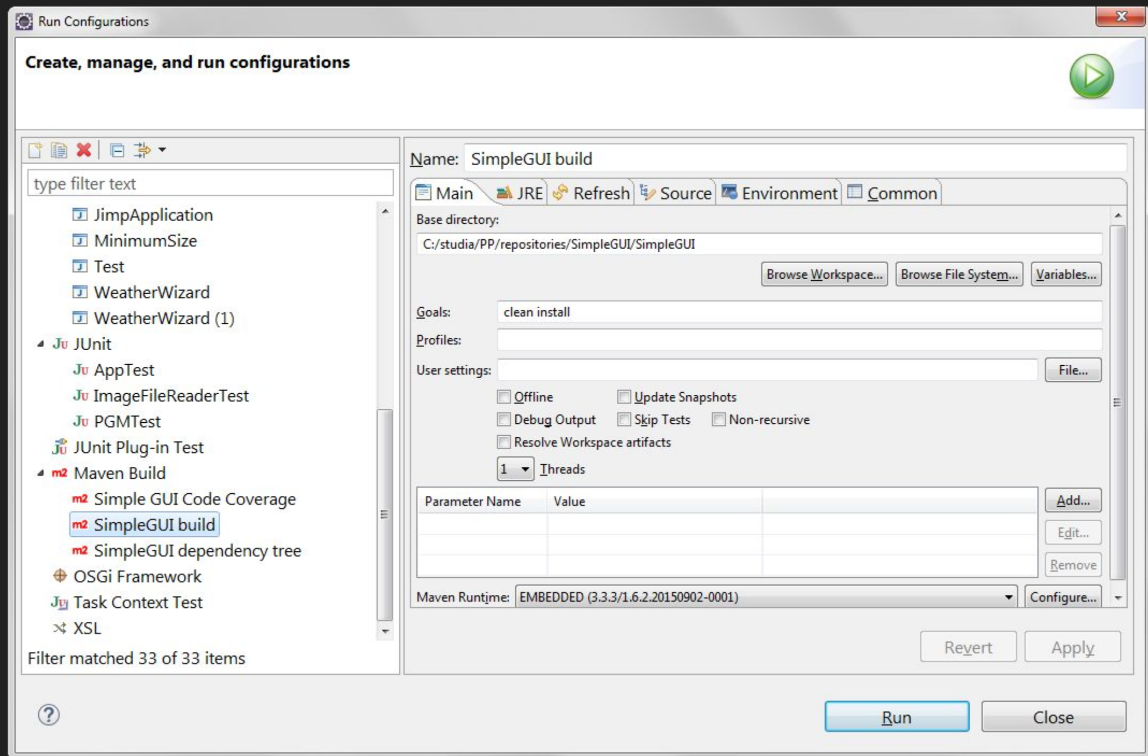
1. Iteracja 4 - Wielkie porządki
    - tag Iteracja4
    - email podsumowujący (raport)
    - Sugerowany jest podział zadań w grupie - jedna osoba pracuje z kodem, druga pisze raport
2. Iteracja 4b - Zadanie dodatkowe - Zapis ASCII Art do pliku graficznego
    - Tag Iteracja4b

Po co piszemy testy?

Raporty pokrycia kodu testami

- Cobertura (patrz kolejne 4 slajdy) - plugin Maven'a
- Emma - plugin instalowany przez Eclipse Marketplace
- IntelliJ Code Coverage runner

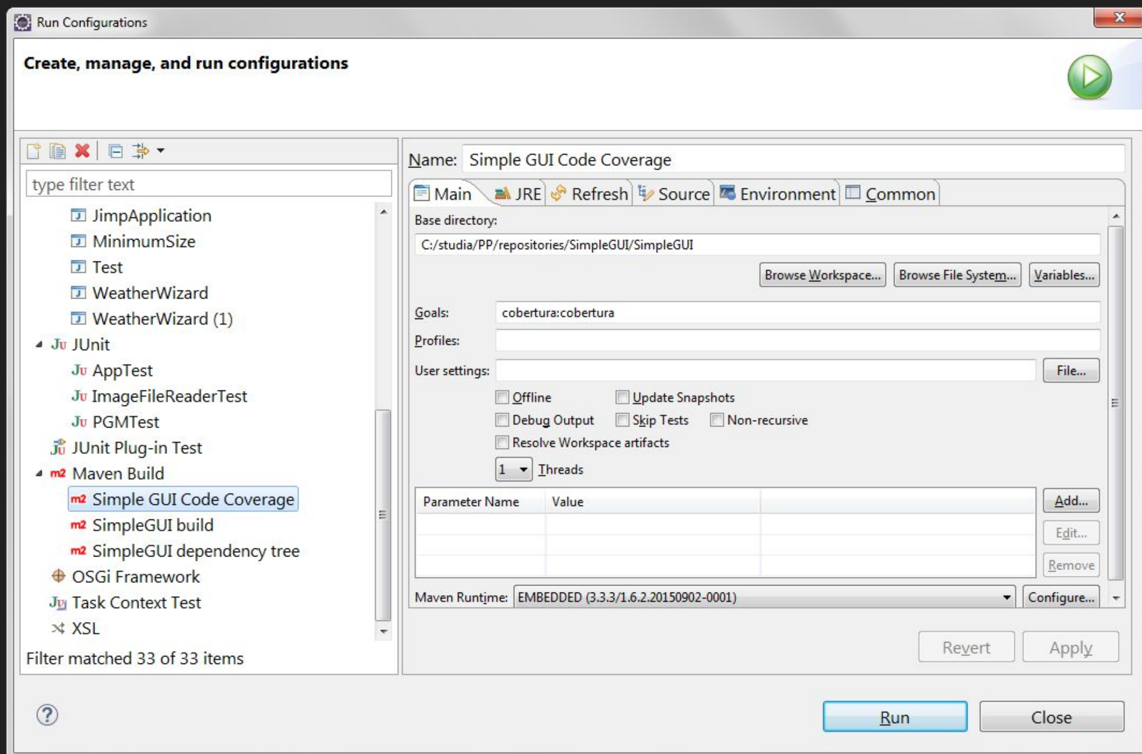
Maven clean install



Goals: clean install

Krok 1: Poprawić testy żeby przechodziły (i żeby aplikacja pomyślnie się zbudowała)

Cobertura - pokrycie kodu testami



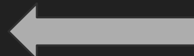
Goals: cobertura:cobertura

Krok 2: Sprawdzić pokrycie aplikacji testami

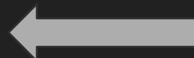
… ale nie uda nam się to

Rozwiązanie - exclusions w maven-resources-plugin

```
<dependency>
	<groupId>org.apache.maven.plugins</groupId>
	<artifactId>maven-resources-plugin</artifactId>
	<version>2.7</version>
	<exclusions>
		<exclusion>
			<groupId>org.slf4j</groupId>
			<artifactId>slf4j-nop</artifactId>
		</exclusion>
		<exclusion>
			<groupId>org.slf4j</groupId>
			<artifactId>slf4j-jdk14</artifactId>
		</exclusion>
	</exclusions>
</dependency>
```

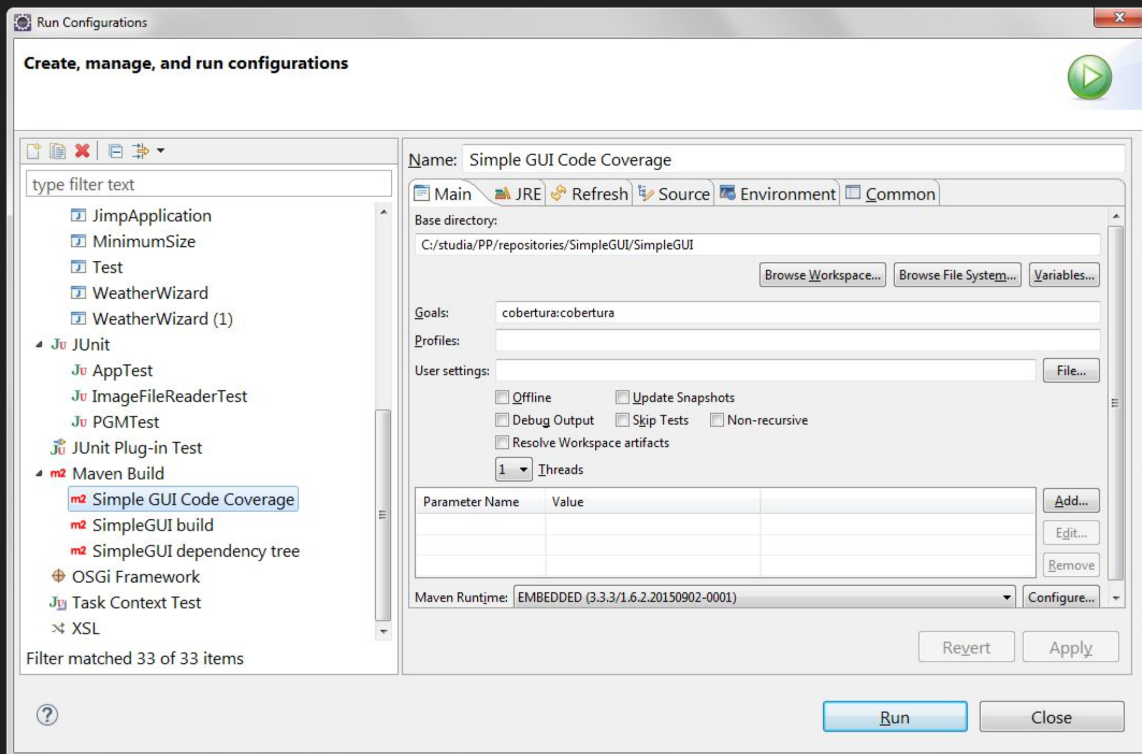


Tę zależność już mamy w pom'ie



Pomarańczowy fragment musimy dodać

Cobertura - pokrycie kodu testami



Teraz już powinno działać.

Raport z wykonania:
index.html
w katalogu
target/site/cobertura

Screenshot(y) z Cobertury proszę umieścić w raporcie.

Co nam mówi raport pokrycia?

- Line coverage - ile linii kodu jest pokrytych testami
- Branch coverage - ile gałęzi kodu (if-else, try-catch) jest pokrytych testami

Od czego zacząć jeżeli mamy niskie pokrycie?

Od tzw. logiki biznesowej

  - Kluczowe funkcjonalności
  - Instrukcje warunkowe
  - Zachowanie aplikacji w sytuacjach wyjątkowych

Po co nam Mocki?

- Test jednostkowy powinien testować pojedynczą ścieżkę kodu w jednej metodzie. Jeżeli wykonanie tej metody oddelegowane jest do innego obiektu i z powrotem, wówczas mamy zależność.
- Żeby pozbyć się zależności używamy Mocków.

1. Mockito
```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>1.10.19</version>
</dependency>
```
2. EasyMock
```
<dependency>
    <groupId>org.easymock</groupId>
    <artifactId>easymock</artifactId>
    <version>3.4</version>
</dependency>
```

Albo Mockito albo EasyMock - - nie potrzebujemy obu!

Zależności należy umieścić POD zależnościami do junit / hamcrest

Przykładowy test - Klasa testowana

```
public class ImageConverter {
    public int getHeightPreservingRatio(BufferedImage image, int
requestedWidth) {
        int imageWidth = image.getWidth();
        int imageHeight = image.getHeight();
        double ratio = ((double) imageHeight / (double) imageWidth);
        return (int) Math.round(requestedWidth * ratio);
    }

}
```

Przykładowy test - Klasa testująca

```
public class ImageConverterTest {

	@Mock private BufferedImage image;
	private ImageConverter converter;

	@Before
	public void setUp() {
		MockitoAnnotations.initMocks(this);
		converter = new ImageConverter();
	}
	@Test
	public void shouldRescaleImageWithPreservedRatio() {
		Mockito.when(image.getWidth()).thenReturn(300);
		Mockito.when(image.getHeight()).thenReturn(150);

		int result = converter.getHeightPreservingRatio(image, 100);

		Assert.assertThat(result, Matchers.is(Matchers.equalTo(50)));
		Mockito.verify(image).getWidth();
		Mockito.verify(image).getHeight();
	}
}
```

Pamiętajcie, że można używać statycznych importów, żeby nie pisać Mockito za każdym razem
(ale Eclipse poradzi sobie lepiej z importami, jeżeli napiszecie mu nazwę klasy przy pierwszym użyciu)

Zadanie 1 - 30 minut

Zastanowić się, dla których fragmentów kodu testy są najbardziej istotne.
Zwiększyć ogólne pokrycie kodu o co najmniej 10%.

W raporcie umieścić screenshoty z raportu Cobertury / Emmy / IntelliJ Code Coverage z przed dopisania testów oraz po dopisaniu testów.

Uzasadnienie wyboru funkcjonalności do pokrycia testami również proszę napisać w raporcie.

Opiszcie też krótko w jakim celu wykorzystaliście Mocki.

Podstawowe zasady OOP

1. Enkapsulacja (hermetyzacja)
   a. Ukrywanie nieistotnych szczegółów
   b. Wyodrębnienie interfejsu
   c. Udopornienie na błędy
2. Dziedziczenie
   a. Hierarchia klas zwiększa czytelność i pozwala na re-używanie kodu
3. Abstrakcja
   a. Pozwala wydzielić ważne elementy obiektu od nieistotnych (pomocniczych)
4. Polimorfizm
   a. Umożliwia pracę na różnych implementacjach tego samego abstrakcyjnego zachowania

Modyfikatory dostępu

Dostęp	Modyfikator	Kto widzi?	Gdzie stosować
Prywatny	`private`	Klasa definiująca	Zmienne, metody do użytku wewnętrznego
Pakietowy (domyślny)	`(brak)`	+ Klasy w danym pakiecie	Metody prywatne (ew. zmienne) jeżeli potrzebne w testach; Inne klasy w pakiecie ale nikt inny
Chroniony	`protected`	+ Klasy dziedziczące	Jeżeli potrzebne w klasach dziedziczących
Publiczny	`public`	+ Wszyscy inni	Główne funkcjonalności

Zadanie 2 - 15 minut

Zanalizować i w miarę potrzeby zmienić modyfikatory dostępu zmiennych i metod w klasach Waszego projektu.

Wybory krótko uzasadnić w raporcie.

Elementy Java 8 - wyrażenia lambda

Przed Java 8 - anonimowe obiekty:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) { [kod] }
});
```

Java 8 - wyrażenia lambda:

```
button.addActionListener( e -> { [kod] });
```

Elementy Java 8 - strumienie

Przed Java 8 - pętle

```
for (int i=0; i<w*h; i++) {
    luminosity[i] = getLuminosity(new Color(rgb[i]));
}
```

Java 8 - strumienie, np strumień intów (teoretycznie bardziej wydajne niż pętle)

```
IntStream.range(0, w*h).forEach(
    i -> luminosity[i] = getLuminosity(new Color(rgb[i])));
```

Zadanie 3 - 10 minut

Zastosujcie w projekcie:

- Przynajmniej jedno wyrażenie lambda
- Przynajmniej jeden strumień

Co, gdzie i jak opiszcie w raporcie.

Executable JAR (wykonywalny plik JAR)

```
<build>
  <plugins>
    ... <!-- inne pluginy -->
    <plugin> <!-- tę sekcję trzeba dodać do pom'a -->
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <!-- w mainClass trzeba wpisać pełną ścieżkę do głównej klasy -->
            <mainClass>pl.edu.pwr.gui.JimpApplication</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Zadanie 4 - 5 minut

Stwórzcie wykonywalny plik JAR swojej aplikacji.

Po zakończeniu tego zadania proszę stworzyć taga Iteracja4.

Iteracja 4 - Raport

W raporcie powinny się znaleźć:

- Screenshoty raportów Cobertury / Emmy / IntelliJ Code Coverage runner: przed iteracją (ale już z poprawionymi starymi testami) oraz po iteracji
- Jakie funkcjonalności uznano za krytyczne i jakie testy w związku z tym dopisano
- W jakim celu wykorzystano mocki
- Jakie modyfikatory dostępu zmieniono i dlaczego
- Jakie fragmenty kodu zmieniono na Java 8

Raport proszę wysłać mailem oraz załączyć zbudowanego i działającego jar'a (uruchamiającego okno główne aplikacji).

Iteracja 4b - Czyli zadanie dodatkowe

Przycisk “Funkcja 1” powinien zapisywać wynikowe ASCII Art do pliku graficznego (a nie tekstowego).

Nazwa przycisku powinna odzwierciedlać funkcjonalność.

Po zakończeniu proszę stworzyć taga Iteracja4b.

Wczytaj obraz

[Opcja 1]

[Opcja 2]

Zapisz do pliku

[Funkcja 1]

[Funkcja 2]

[Miejsce na obrazek]