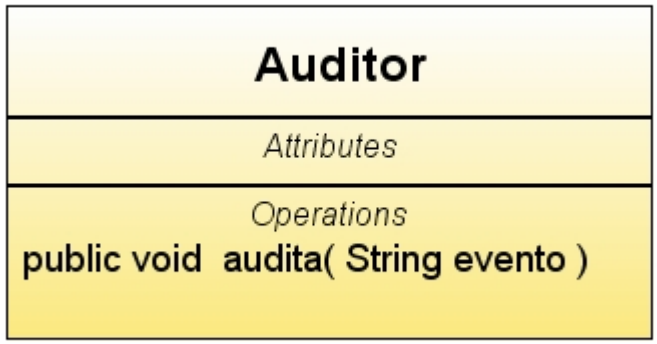
MAC-0462

Padrões arquiteturais para sistemas baseados em componentes reconfiguráveis

- Microkernel
- Service Locator
- Dependency Injection

Exemplo

- Classe Auditor
  - Salva registros de eventos no banco de dados
  - Deve registrar a hora do evento



Não reconfigurável

```
package estatico;
import java.util.Date;

public class Auditor {
  public void audita(String evento) {
    Date horaAtual = new Date(System.currentTimeMillis());
    salvaNoBanco(evento, horaAtual);
  }

  private void salvaNoBanco(Object... valores) {}
}
```

Por que reconfigurável?

- Flexibilidade no ambiente de implantação
  - Poder adicionar componentes in-loco
  - Poder trocar componentes
    - No exemplo, queremos poder usar diversos relógios
    - RelogioDoSistema – Igual à versão mostrada
    - RelogioDistribuido – Tempo de um servidor de rede
- Facilitar testes unitários
  - Trocar componentes reais por mocks ou stubs
    - No exemplo queremos um RelogioStub, que sempre retorne um valor fixo
    - É necessário para tornar os testes reproduzíveis

Microkernel (1)

- Nasceu como arquitetura de Sistemas Operacionais
- Descrito como pattern no *Pattern Oriented Software Architecture* 1
- Trataremos da variante *Microkernel System with indirect Client-Server connections*
  - É a variante aplicada no Jboss
  - O Microkernel intermedeia interações entre componentes (*servers* na terminologia do POSA)

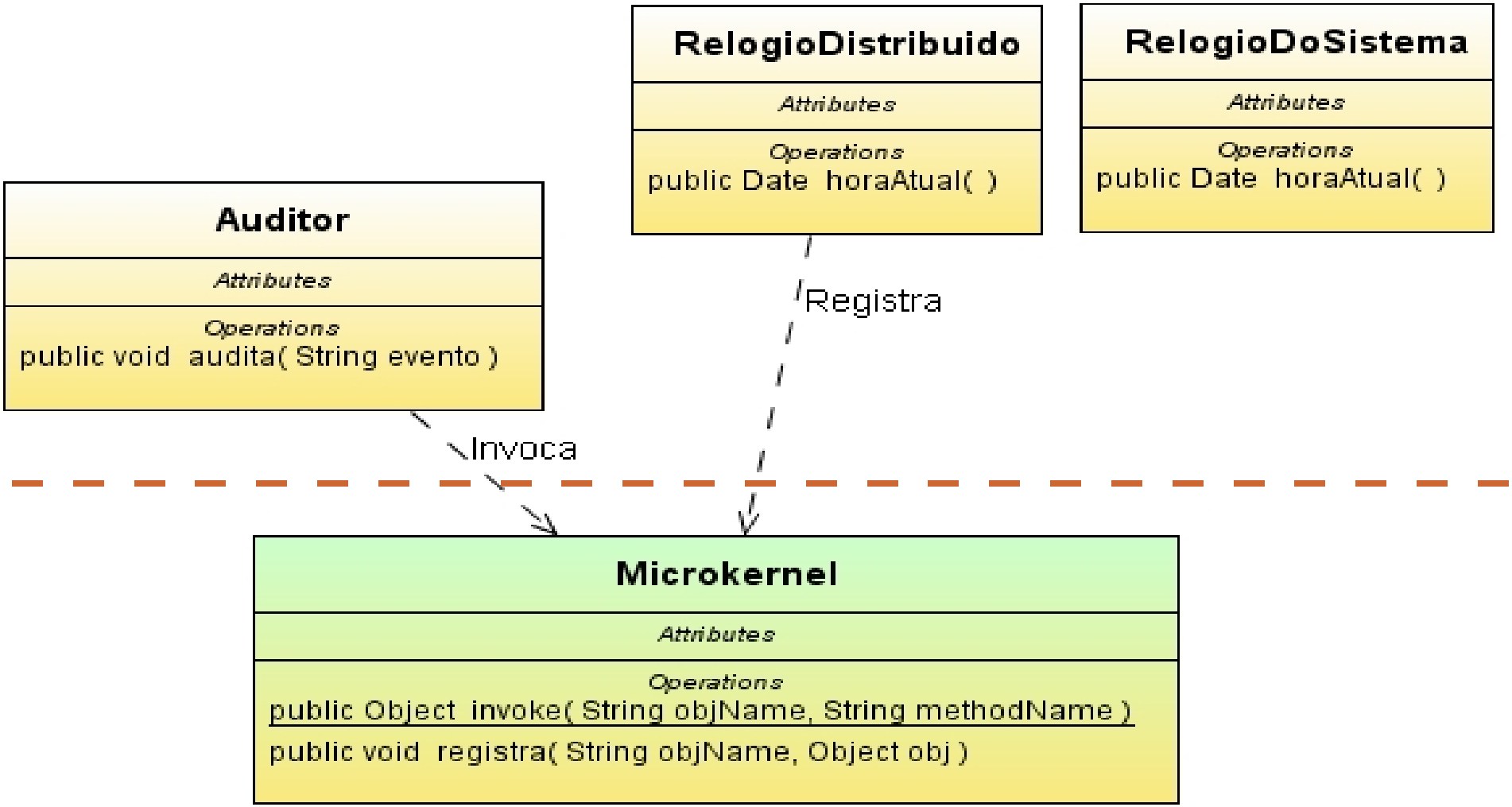
Microkernel (2)

```
package microkernel;
import java.util.Date;

public class Auditor {
  public void audita(String evento) {
    Date horaAtual =
      (Date)Microkernel.invoke("relogio", "horaAtual");
    salvaNoBanco(evento, horaAtual);
  }

  private void salvaNoBanco(Object... valores) {}
}
```

Microkernel (3)



Microkernel (4)

- Os componentes são bastante desacoplados
- O microkernel pode implementar reconfiguração dinâmica
  - Até as interfaces dos componentes podem mudar
- O microkernel tem controle sobre todas as invocações
- Modelo de programação desajeitado
- Burla verificação estática pelo compilador
  - Ferramentas baseadas em tipagem estática também não funcionam

Service Locator (1)

- Descrito no *Core J2EE Patterns*
  - Ajuda a lidar com a complexidade das APIs
  - Encapsula lookup JNDI
- Não é restrito ao ambiente J2EE
- O ServiceLocator localiza e retorna referências para os componentes

Service Locator (2)

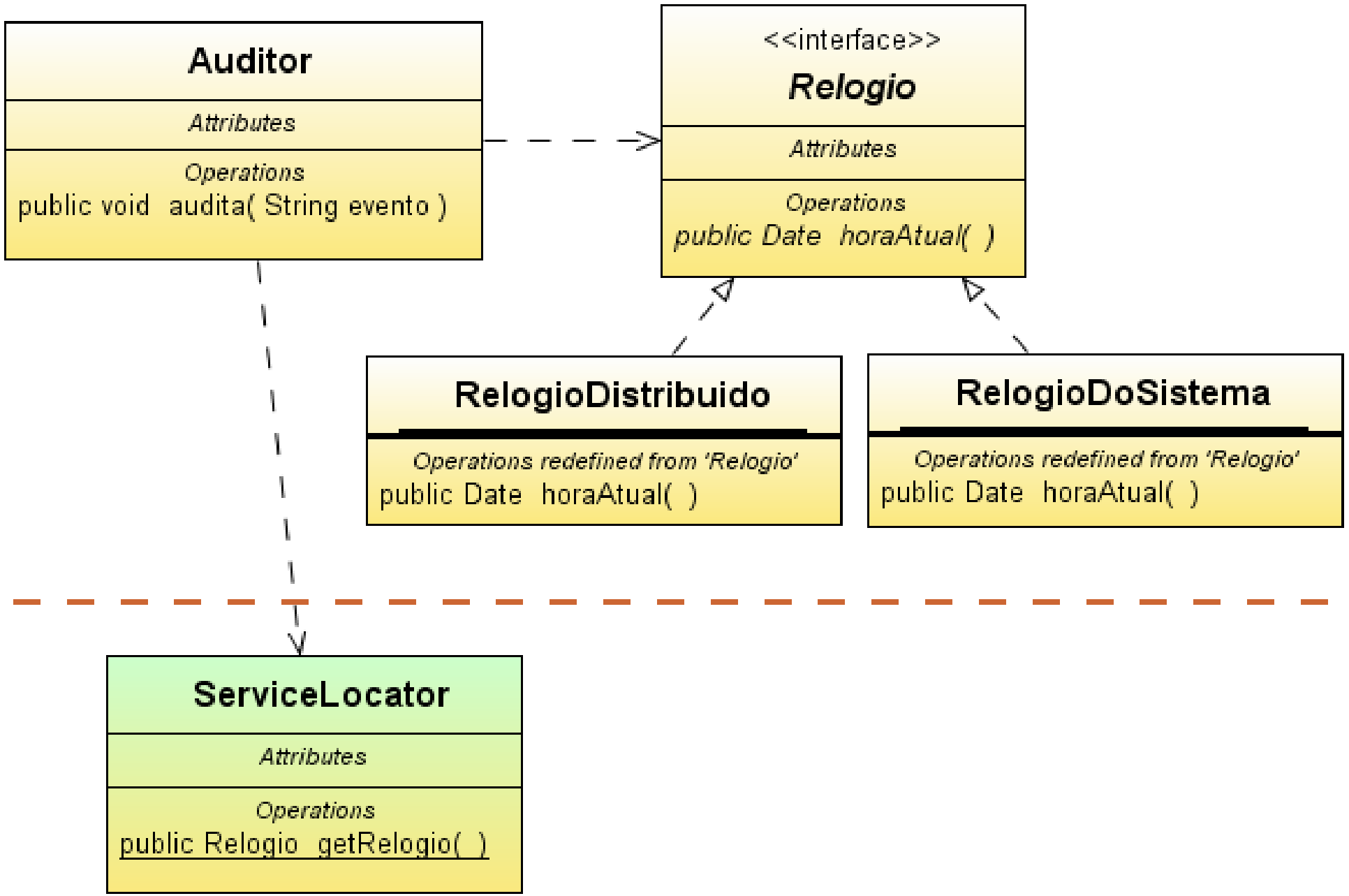
```
package service_locator;

import java.util.Date;

public class Auditor {
  public void audita(String evento) {
    Relogio r = ServiceLocator.getRelogio();
    Date horaAtual = r.horaAtual();
    salvaNoBanco(evento, horaAtual);
  }

  private void salvaNoBanco(Object... valores) {}
}
```

Service Locator (3)



Service Locator (4)

- Cliente depende de Interface específica
  - Compilador pode verificar as chamadas
  - Ferramentas baseadas em verificação de tipos funcionam
- Busca de referência para componente é explícita
- Cada novo componente implica em alterar o ServiceLocator
- Cliente depende do Service Locator
  - Ruim para integração de componentes com origens diversas
- Interface de um componente não pode mudar

Variação Service Locator Dinâmico (1)

```
package dyn_service_locator;

import java.util.Date;

public class Auditor {
  public void audita(String evento) {
    Relogio r = (Relogio)ServiceLocator.locate("relogio");
    Date horaAtual = r.horaAtual();
    salvaNoBanco(evento, horaAtual);
  }

  private void salvaNoBanco(Object... valores) {}
}
```

Variação Service Locator Dinâmico (2)

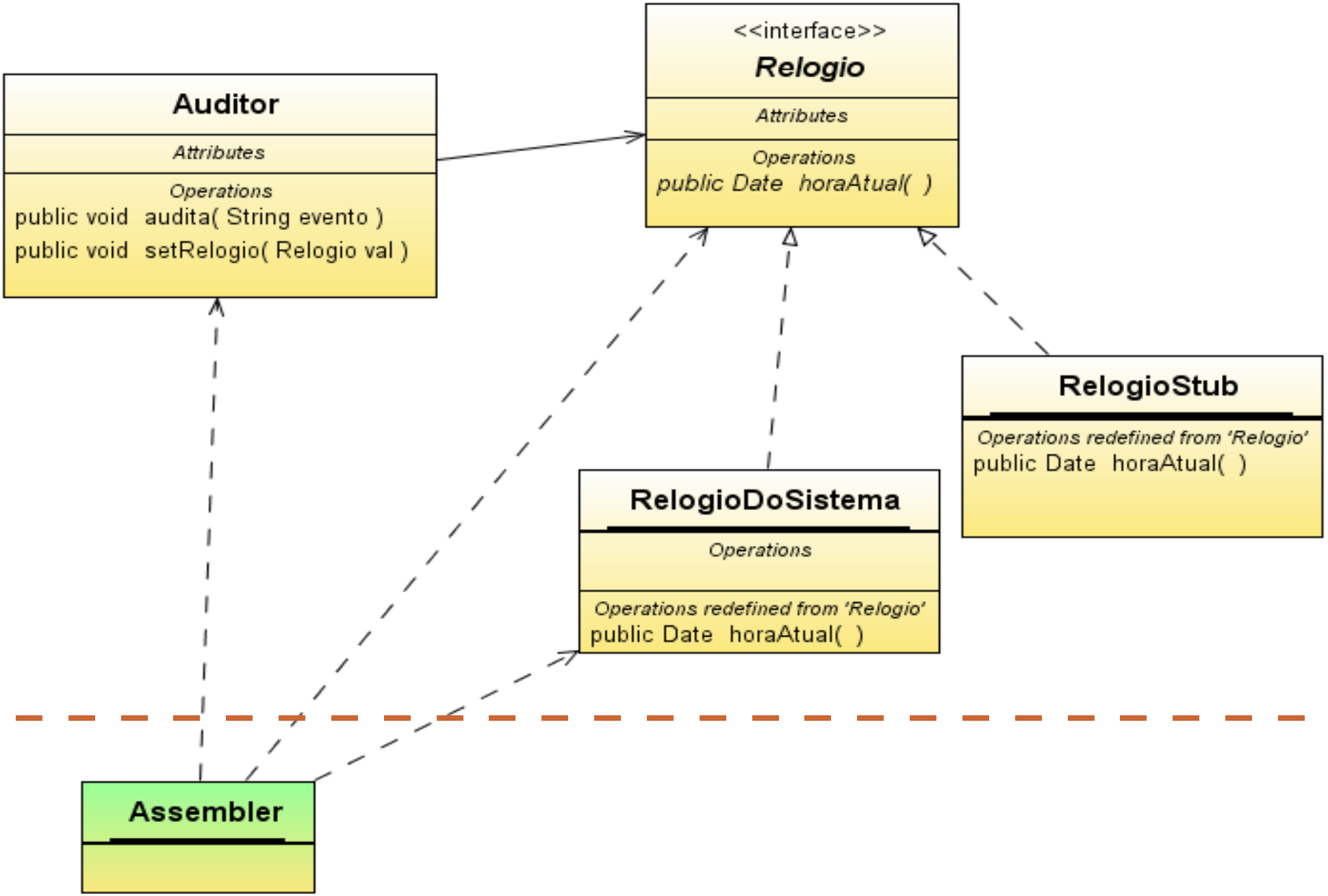
- O ServiceLocator dinâmico
  - Não precisa ser alterado para lidar com novos componentes
  - É implementado de maneira mais simples

- O código é menos explícito
- Certos bugs não são detectados pelo compilador
- Ferramentas baseadas em análise estática não funcionam

Dependency Injection

- Também chamado de Inversion of Control
  - Nomenclatura ambígua (IoC é um conceito amplo e antigo)
  - A “inversão” se dá com um framework injetando as dependências nos componentes
- O framework é chamado de DI Container (ou IoC container)
- Exemplos
  - Spring
  - Picocontainer/Nanocontainer
  - EJB3 (de forma limitada)

DI – Setter Injection (1)



DI – Setter Injection (2)

```
package setterdi;

import java.util.Date;

public class Auditor {
  private Relogio relogio;

  public void setRelogio(Relogio r) {this.relogio = r;}

  public void audita(String evento) {
    Date horaAtual = relogio.horaAtual();
    salvaNoBanco(evento, horaAtual);
  }

  private void salvaNoBanco(Object... valores) {}
}
```

DI – Setter Injection (3)

- *Arquivo de configuração do Spring (spring.xml):*

```
<beans>
   <bean id="Auditor" class="setterdi.Auditor">
      <property name="relogio">
         <ref local="Relogio"/>
      </property>
   </bean>

   <bean id="Relogio" class="setterdi.RelogioDoSistema"/>
</beans>
```

DI – Setter Injection (4)

- Código de inicialização usando o Spring:

```
public class Assembler {
  public static void main(String[] args) {
    ApplicationContext container =
      new FileSystemXmlApplicationContext("spring.xml");

    Auditor auditor = (Auditor)container.getBean(“Auditor”);
    auditor.audita(“Retirada de R$1,99”);
  }
}
```

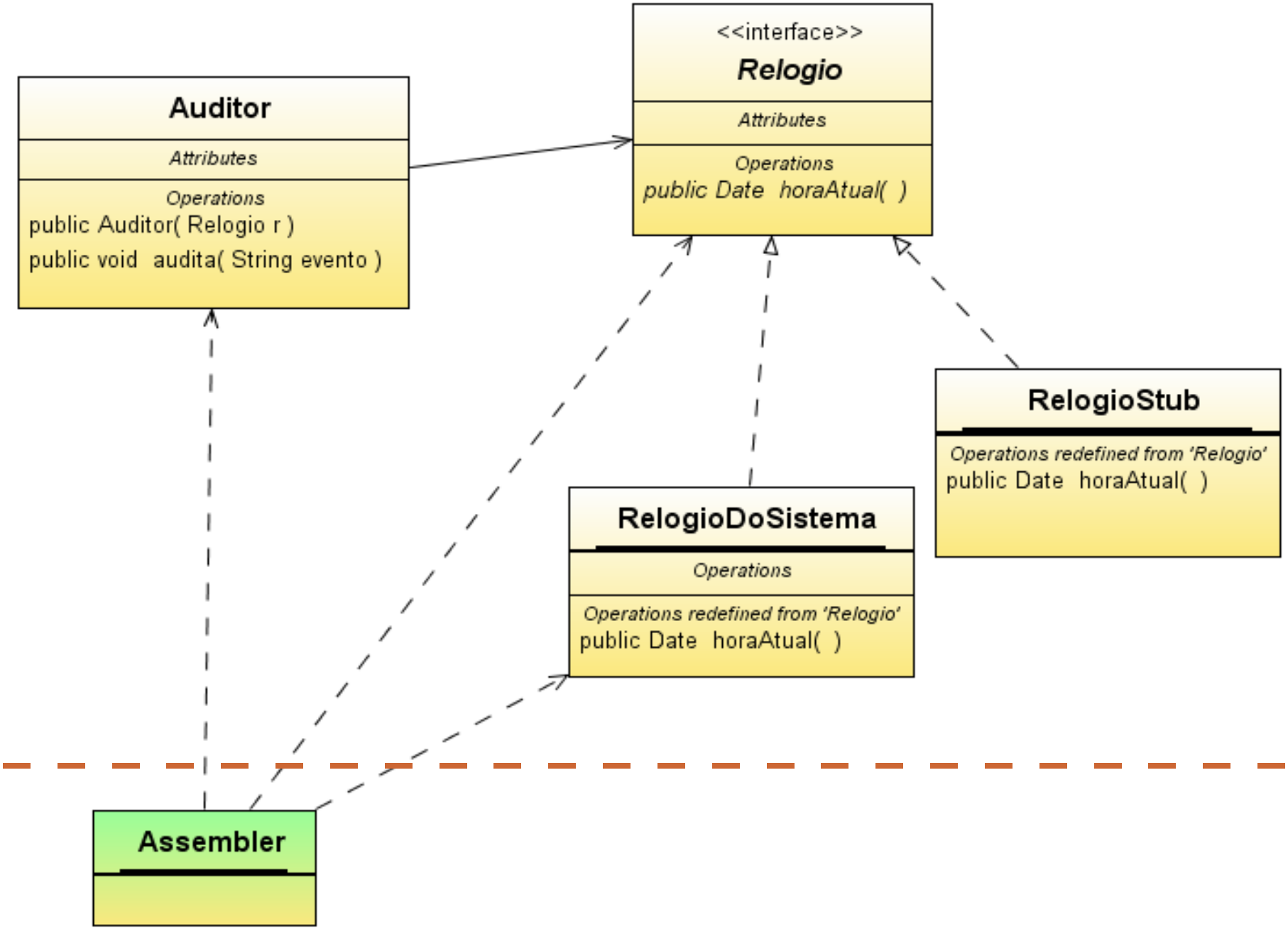
DI – Setter Injection (5)

- *Arquivo de configuração com autowiring:*

```
<beans>
 <bean id="Auditor" class="setterdi.Auditor"
  autowire=”byType” />

 <bean id="Relogio" class="setterdi.RelogioDoSistema"/>
</beans>
```

DI – Constructor Injection (1)



DI – Constructor Injection (2)

```
package constrdi;

import java.util.Date;

public class Auditor {
  private final Relogio relogio;

  public Auditor(Relogio r) { this.relogio = r; }

  public void audita(String evento) {
    Date horaAtual = relogio.horaAtual();
    salvaNoBanco(evento, horaAtual);
  }

  private void salvaNoBanco(Object... valores) {}
}
```

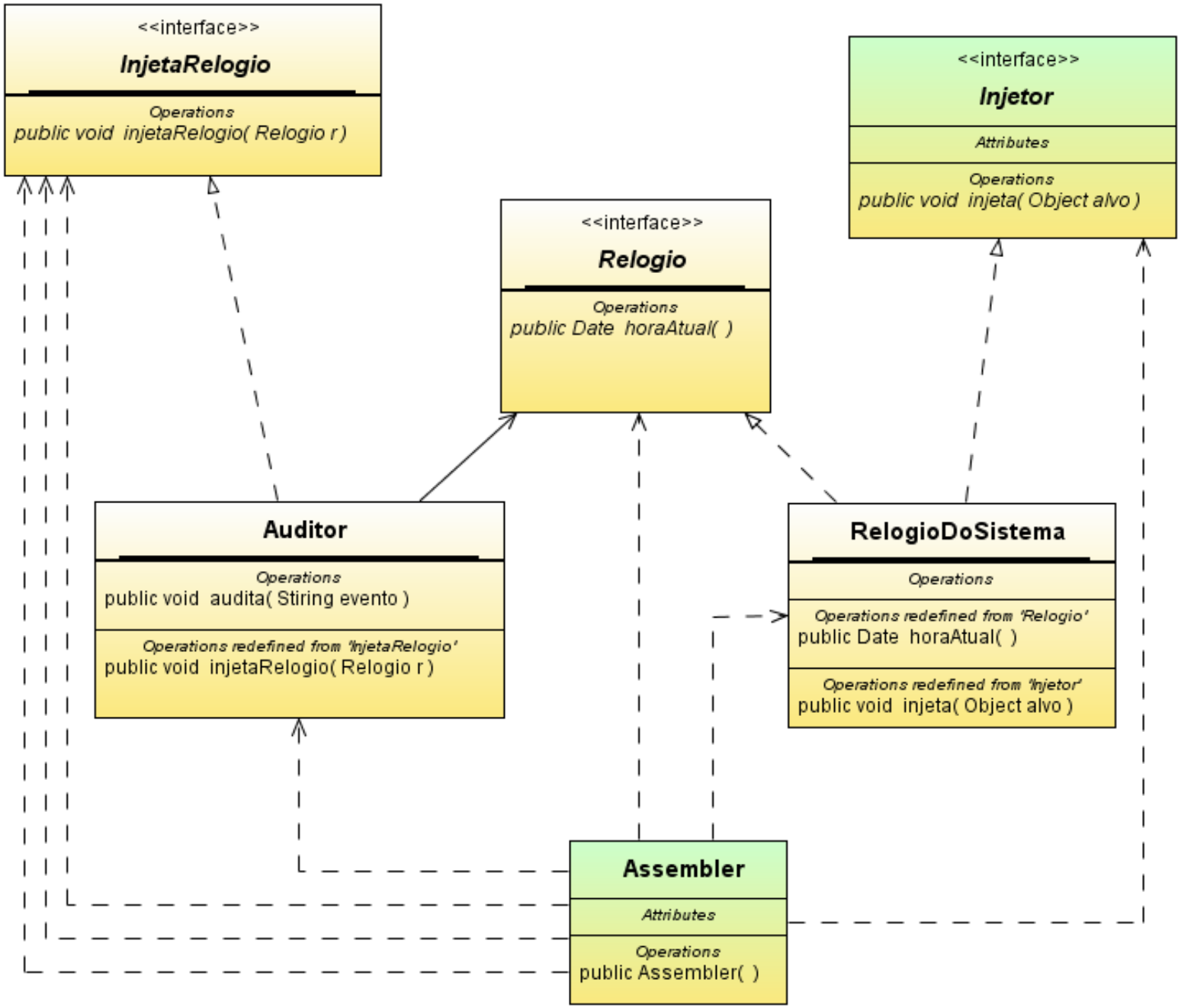
DI – Constructor Injection (3)

- *Arquivo de configuração do Spring (spring.xml):*

```
<beans>
   <bean id="Auditor" class="constrdi.Auditor">
      <constructor-arg>
         <ref local="Relogio"/>
      </constructor-arg>
   </bean>

  <bean id="Relogio" class="constrdi.RelogioDoSistema"/>
</beans>
```

DI – Interface Injection (1)



DI – Interface Injection (2)

```
public interface InjetaRelogio {
  public injetaRelogio(Relogio r);
}
public class Auditor implements InjetaRelogio {
  private Relogio relogio;
  public void injetaRelogio(Relogio r) {this.relogio = r;}
  ...
}

public interface Injetor {
  public void injeta(Object alvo);
}
public class RelogioDoSistema implements Relogio, Injetor {
  public void injeta(Object alvo) {
    ((InjetaRelogio)alvo).injetaRelogio(this);
  }
}
```

Interface Injection (3)

- Código de inicialização com container fictício

```
public class Assembler {
  public static void main(String[] args) {
    Auditor auditor = new Auditor();
    Relogio relogio = new Relogio();

    Container container = new Container();
    container.registerComponent("Auditor", auditor);
    container.registerComponent("Relogio", relogio);

    container.registerInjetor(InjetaRelogio.class, relogio);
    container.setup();
  }
}
```

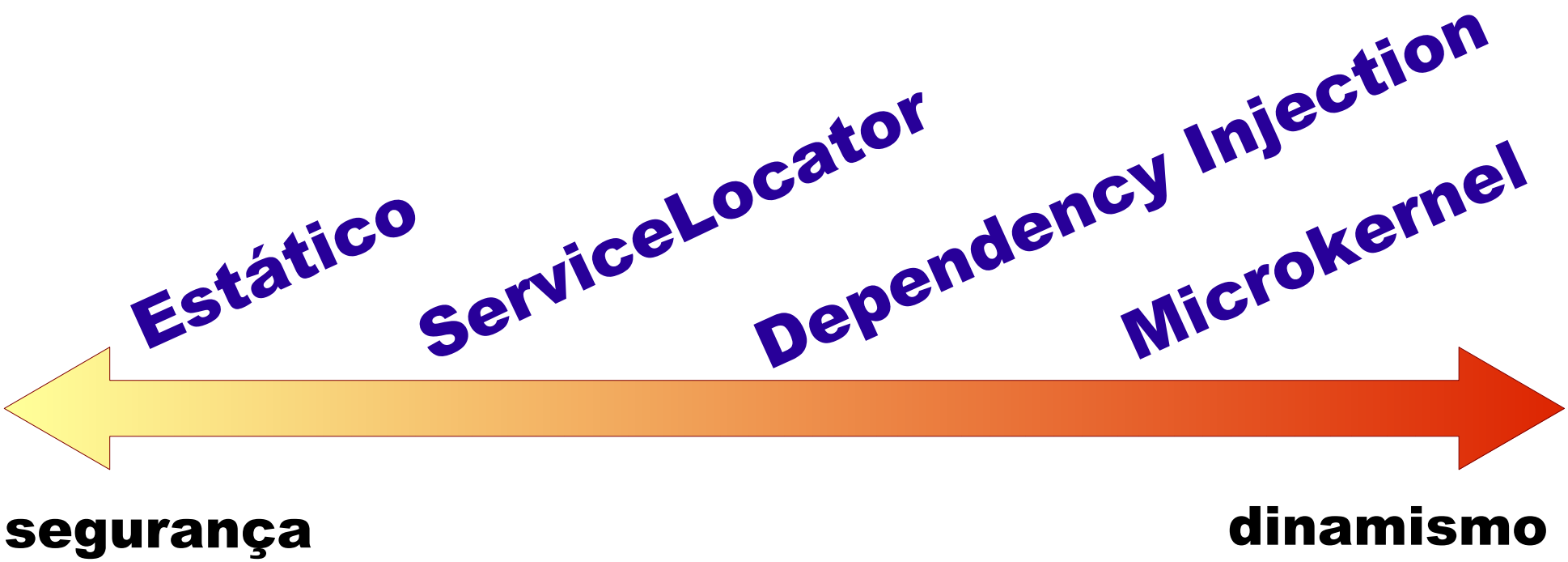
Dependency Injection - Variedades

- Setter
  - Configuração mais explícita (não depende de ordem de parâmetros)
  - Facilita grande variedade de configurações de um componente
  - Não garante invariantes na construção
    - Não permite campos **final** denotando imutabilidade
- Constructor
  - Objeto pode garantir invariantes na construção
  - Configuração depende de ordem dos parâmetros do construtor
- Interface: - Pouco usada atualmente

Dependency Injection – Análise

- Componentes não dependem do framework.
- Fácil integração de um componente em sistema de terceiros
- Testes unitários são fáceis
- Objetos não específicos de um framework são fácilmente integráveis
- Configuração dinâmica dá margem a erros
- Ferramentas baseadas em análise estática não funcionam
- Interface de um componente não pode mudar

Comparando os patterns (1)



Comparando os patterns (2)

	Facilidade de uso	Integração	Testabilidade
Microkernel	X	X	-
Dependency Injection	-	√	√
Service Locator	-	X	-