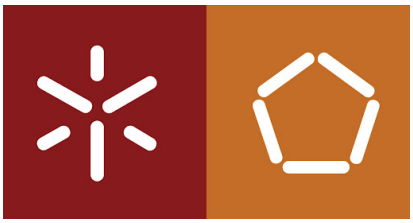

Universidade do Minho

Mestrado em Engenharia Informática

Projeto de Informática

Estudo do tema: CPI *Governance Automation*

Cláudia Ribeiro (PG49998) Daniel Azevedo (PG50311)
Joaquim Roque (PG50502) Nuno Mata (PG44420)
Pedro Araújo (PG50684) Rodrigo Pires Rodrigues (PG50726)
Rui Guilherme Monteiro (PG50739)

Ano Letivo 2023/2024

Conteúdo

1 Introdução

A integração eficiente de sistemas desempenha um papel crucial nas operações empresariais modernas. Este guia concentra-se em fornecer diretrizes e melhores práticas para otimizar o uso do SAP CPI (SAP Cloud Platform Integration) e da linguagem Groovy. Essas recomendações visam melhorar a qualidade, a segurança e o desempenho das integrações, garantindo que sua organização aproveite ao máximo a plataforma. Abordaremos tópicos como o uso do adaptador ProcessDirect, *scripts* Groovy para mapeamentos, estrutura de pacotes de importação, regras do CPILint e mais.

Além disso, o guia também discutirá as "Regras de Verificação do CPILint", destacando tanto as regras existentes quanto as potenciais a serem adicionadas. Implementar essas melhores práticas e regras de verificação do CPILint pode contribuir para uma integração mais eficaz e confiável, beneficiando a operação e os resultados em contexto empresarial.

2 Melhores Práticas

2.1 SAP CPI

O SAP CPI (SAP Cloud Platform Integration) é uma plataforma de integração em nuvem desenvolvida pela SAP. O objetivo principal do SAP CPI é facilitar a integração de sistemas e aplicações de negócios, permitindo que as organizações conectem e automatizem processos em toda a empresa.

Na presente subsecção, iremos discutir as melhores praticas na plataforma SAP CPI.

2.1.1 Usar o Adaptador ProcessDirect para criar IFlows

O adaptador ProcessDirect é uma ferramenta que permite reutilizar configurações comuns de adaptadores em várias interfaces. Isso simplifica o desenvolvimento e a manutenção de integrações, onde se pode economizar tempo e esforço ao usar configurações predefinidas.

2.1.2 Groovy *Script* para Mapeamentos

O uso de *scripts* Groovy é recomendado para mapear dados nas integrações. Groovy é uma linguagem de programação que oferece flexibilidade e facilidade de manutenção ao criar transformações e mapeamentos de dados dentro do SAP CPI.

2.1.3 Usar *Reader* ao acessar o corpo da mensagem

Ao acessar o corpo das mensagens em *scripts* Groovy, é mais eficiente usar um leitor (Reader) em vez de converter toda a mensagem em uma string, especialmente quando se lidar com cargas de mensagem maiores. Isso ajuda a economizar recursos e aprimorar o desempenho.

2.1.4 IFlows Configuráveis

Nesta prática, deve-se manter os parâmetros externos ao IFlow. Isso permite fazer alterações nos aspetos do IFlow sem precisar editar o IFlow em si, tornando-o mais flexível e fácil de adaptar às necessidades em evolução.

2.1.5 ID de Mensagem de Aplicação

Usar o ID de Mensagem de Aplicação (SAP_ApplicationID) no cabeçalho das mensagens ajuda a identificar mensagens de forma exclusiva. Isso pode ser útil para fins de monitoramento e análise, tornando mais fácil rastrear e distinguir diferentes mensagens.

2.1.6 Tratamento de erros nos IFlows

O desenvolvedor deve estar preparado para lidar com falhas nas mensagens, pois o SAP CPI coloca a responsabilidade pelo tratamento de erros nas mãos dos desenvolvedores. Isso inclui estratégias para detetar, registar e lidar com erros de integração.

2.1.7 Projetar artefactos de integração para suportar uma paisagem de 2 níveis

Esta prática envolve projetar seus artefactos de integração de forma que possam ser facilmente duplicados e configurados para diferentes ambientes, como ambientes de não produção e produção. Isso é importante para facilitar a gestão e a escalabilidade das integrações em diferentes cenários.

2.1.8 Gestão de Versões

Recomenda-se o uso do WebUI do SAP CPI para gerir versões de artefactos de integração. Isso ajuda a controlar e rastrear alterações nos artefactos, garantindo que é possível voltar a versões anteriores, se necessário, e também permite o controle externo de versões usando o Git, proporcionando uma maneira estruturada de gerir e documentar mudanças nas integrações.

2.2 Groovy

Groovy é uma linguagem de programação orientada a objetos, usada sobretudo no desenvolvimento de *scripts*. Destaca-se pela grande compatibilidade e similaridade com Java, quer ao nível da interoperabilidade do código, quer ao nível da sintaxe. Isto deve-se sobretudo ao facto de que ambas são linguagens cujo código é compilado para *bytecode*, interpretado pela JVM em *run-time*.

Nesta subsecção, discutem-se algumas das melhores práticas a ter em conta ao escrever código Groovy.[1]

2.2.1 Não usar ponto e vírgula

Uma característica semelhante a outras linguagens de *scripting* (e.g. Python ou POSIX Shell), em que, apesar de permitidos, os pontos e vírgulas não são necessários, sendo inclusive desaconselhados.

2.2.2 *Keyword return* opcional

Por forma a tornar o código de uma determinada função/rotina mais concisa, a *keyword return* é opcional. O valor de retorno será a última expressão, à semelhança de Rust.

2.2.3 Evitar usar *def* e tipo em simultâneo

Em Groovy, uma variável pode ser declarada usando a sintaxe *à la* Java, ou então com a *keyword def*, evitando assim especificar o tipo. É também possível usar a *keyword* **e** o tipo, o que não é aconselhado, uma vez que aumenta a verbosidade do código sem benefício algum.

```
String name     = "example"  // Bom
def name        = "example2" // Bom
def String name = "example3" // Mau
```

2.2.4 *public* por omissão

À semelhança de algumas das outras regras até aqui discutidas, Groovy faz algumas assunções em relação ao código. Concretamente, o modificador de visibilidade *public* é o valor por omissão para as classes e para os métodos, pelo que não é necessário usá-lo explicitamente.

2.2.5 Omitir parênteses

Em algumas circunstâncias, é possível omitir os parênteses tipicamente usados ao invocar um método, um pouco à semelhança do que acontece em Perl.

2.2.6 *Getters* e *setters*

Não é necessário implementá-los explicitamente, uma vez que o compilador os gera automaticamente para cada campo declarado numa classe, desde que esta siga a estrutura de um *bean*[2].

[1] https://groovy-lang.org/style-guide.html
[2] https://en.wikipedia.org/wiki/JavaBeans

2.2.7 Construtor por omissão para *beans*

No caso dos *beans*, é possível inicializar os campos de cada objeto sem que a classe defina um construtor explícito, comparável ao que acontece com as *structs* em C:

```
class Server {
    String addr
    int port
}

def server = Server(addr: "localhost", port: 8888)
```

2.2.8 Métodos *with()* e *tap()*

Os métodos *with()* e *tap()*, implicitamente presentes em todas as classes, permitem evitar o uso repetido do nome da variável ao efetuar várias operações com a mesma:

```
// mau
def socket = new Socket(...)
socket.bind(...)
socket.listen(...)
socket.accept
socket.close

// bom
def socket = new Socket(...).with {
    bind(...)
    listen(...)
    accept
    close
    it // referência à própria variável num closure
}

// melhor
def socket = new Socket(...).tap {
    bind(...)
    listen(...)
    accept
    close
}
```

2.2.9 Interpolação de *strings*

Groovy permite interpolação de *strings* com um sintaxe semelhante a Javascript, o que evita usar o operador + como em Java.

```
// mau
String lang = "Java"
String s = "This is an interpolated string in " + lang + "..."

// bom
String lang = "Groovy"
String s = "This is an interpolated string in ${lang}..."
```

2.2.10 *Null checking*

O operador ?. verifica se o operando do lado esquerdo é *null* antes de tentar chamar um método ou acessar um campo do mesmo, evitando assim *ifs* aninhados:

```
// mau
if (order != null)
    if (order.customer != null){
        def address = order.customer.address
        if (address != null)
            println address
    }
// bom
println order?.customer?.address
```

A adicionar a isto, existe também o operador ?: para facilitar este tipo padrões. Este operador não é mais que o operador ternário, mas sem a segunda expressão:

```
// mau
def result = name != null ? name : "Unknown"
// bom
def result = name ?: "Unknown"
```

3 Estrutura dos pacotes de importação SAP CPI

Os pacotes de importação SAP CPI são recursos pré-configurados e pré-construídos que podem ser importados para o ambiente de integração do SAP CPI, e o seu objetivo é acelerar e facilitar o processo de criação e implementação de integrações entre diferentes sistemas e projetos. Eles geralmente contém modelos, configurações, fluxos de integração, mapeamentos de dados e outros artefactos pré-desenvolvidos que são projetados para serem reutilizáveis e por isso podem ser adaptados e personalizados para as necessidades específicas de integração de um sistema.

Uma das vantagens de usar estes pacotes é o acesso às práticas recomendadas de sistemas anteriores. Assim, evita-se a necessidade de começar do zero no desenvolvimento de um novo projeto de integração, permitindo uma implementação mais rápida e eficiente de soluções de integração. Uma outra vantagem é a promoção da consistência e a padronização nas integrações, pois muitos dos desafios e cenários comuns já são abordados nos modelos pré-configurados, reduzindo assim a complexidade e o risco associado ao desenvolvimento de integrações personalizadas a partir do zero.

Os pacotes de importação no SAP CPI têm uma estrutura semelhante à de outros sistemas de gestão de integração. Os pacotes são usados para agrupar objetos relacionados e facilitar a gestão de integrações. Os seguintes são os elementos básicos da estrutura de pacotes de importação no SAP CPI:

Pacote: O pacote é a unidade fundamental de organização no SAP CPI. Ele é usado para agrupar objetos de integração relacionados, como *integration flows*, artefactos de integração, mapeamentos, entre outros. Cada pacote tem um nome exclusivo que o identifica.

Artefactos: Os artefactos são os objetos reais que são importados e geridos dentro de um pacote. Isso pode incluir fluxos de integração, mapeamentos, adaptadores, configurações e outros elementos de integração.

Namespaces: Os *namespaces* são usados para criar uma hierarquia lógica nos pacotes, tornando mais fácil organizar e categorizar os artefactos. Por exemplo, você pode ter um *namespace* para "Integrações SAP"e dentro dele, criar pacotes para diferentes tipos de integrações.

A estrutura de pacotes e *namespaces* é flexível e pode ser projetada para atender às necessidades específicas de organização da empresa. Há possibilidade de criar pacotes aninhados para criar uma estrutura hierárquica, o que pode ser útil para manter a organização e a clareza em ambientes com muitas integrações.

Esta estrutura ajuda a gerir eficientemente as suas integrações, permite uma colaboração mais eficaz entre os membros da equipa e ajuda a manter a rastreabilidade de objetos relacionados. É importante planear e projetar a estrutura de pacotes com cuidado para garantir que atenda às necessidades específicas do projeto de integração que seja desenvolvido.

4 Regras de verificação do CPILint

Para fazer um estudo detalhado das regras que já existem no CPILint e identificar regras que podem ser adicionadas, foi necessário examinar as Melhores Práticas recomendadas pela SAP para o desenvolvimento em SAP CPI, bem como considerar os requisitos específicos do projeto.

4.1 Regras Atuais do CPILint (Existentes)

Para organizar e agrupar as regras, podemos considerar várias categorias ou grupos lógicos com base nas suas funcionalidades ou áreas de aplicação.

- **Regras de Segurança**:
  - **CleartextBasicAuthNotAllowed**: Proíbe a autenticação básica não criptografada.
  - **UnencryptedDataStoreWriteNotAllowed**: Impede a escrita de dados não criptografados num repositório de dados.
  - **UnencryptedEndpointsNotAllowed**: Não permite o uso de pontos de *endpoints* não criptografados.
  - **ClientCertSenderChannelAuthNotAllowed**: Exige autenticação do cliente por certificado nos canais de envio.
  - **CsrfProtectionRequired**: Requer a proteção contra Cross-Site Request Forgery (CSRF).
- **Regras de Adapter**:
  - **ReceiverAdapters**: Regras relacionadas aos adaptadores de receção de dados. Pretende garantir que apenas adaptadores recetores específicos sejam usados nos nossos fluxos de integração.
  - **SenderAdapters**: Regras relacionadas aos adaptadores de envio de dados. Pretendem garantir que apenas adaptadores emissores específicos sejam usados nos nossos fluxos de integração.
- **Regras de Tipos de *Routers***:
  - **MultiConditionTypeRoutersNotAllowed**: o fluxo de integração não pode conter passos/etapas do *router* configuradas com ambas condições XML e não XML (não é possível ter os dois tipos ao mesmo tempo).
- **Regras de *Scripting* e Linguagens de Script**:
  - **ScriptingLanguages**: Permite especificar quais linguagens de *script* são permitidas nos nossos fluxos de integração e quais não são.
- **Regras de Mapeamento**:
  - **MappingTypes**: Permite garantir que apenas certos tipos de mapeamento são usados nos nossos fluxos de integração.
- **Regras de Gestão de Recursos**:
  - **DuplicateResourcesNotAllowed**: Permite indicar que os fluxos de integração não devem conter recursos duplicados, ou seja, recursos idênticos que aparecem em vários fluxos de integração.
  - **IflowDescriptionRequired**: Permite garantir que todos os nossos fluxos de integração tenham uma descrição.

  - **NamingConventions**: Permite verificar se as convenções de nomenclatura para canais, etapas de fluxo etc. estão a ser seguidas.

- **Regras de Versões e Bibliotecas**:
  - **XsltVersions**: Permite garantir que apenas versões XSLT específicas sejam usadas nos nossos fluxos de integração.
  - **JavaArchives**: Permite especificar quais arquivos Java são permitidos nos nossos fluxos de integração e quais não são.

- **Regras de Processamento de Canal**:
  - **MatchingProcessDirectChannelsRequired**: Esta regra garante que para cada canal recetor do *ProcessDirect* haja um canal emissor com o mesmo endereço.

Questões

- **Verificação de Adaptadores**: Existem regras que proíbem o uso de adaptadores não autorizados ou não recomendados? As regras atuais verificam se apenas adaptadores específicos são usados em cenários apropriados?
- **Verificação de Mapeamento**: As regras atuais garantem que apenas ferramentas de mapeamento aprovadas sejam usadas? As regras atuais verificam a conformidade com as melhores práticas de mapeamento de dados?
- **Segurança**: As regras atuais verificam se as melhores práticas de segurança estão a ser seguidas? Há regras para garantir o tratamento adequado de dados sensíveis?
- **Documentação**: As regras atuais abordam a qualidade da documentação das interfaces de integração e do código Groovy? Elas verificam se há descrições suficientes e informações relevantes?

Da resposta a esta questões surgem ideias de novas regras possíveis a implementar.

4.2 Potenciais Regras a Serem Adicionadas

- **Convenções de Nomenclatura**:
  - Adicionar regras que verifiquem se as convenções de nomenclatura estão a ser seguidas para garantir a consistência e a clareza no código.
- **Otimização de desempenho**:
  - Verificar se o código atende às melhores práticas de otimização de desempenho.
- **Manipulação de Erros**:
  - Incluir regras para verificar se o código trata de maneira adequada e lida com erros e exceções.
  - Garantir que os fluxos de integração tenham um tratamento adequado de erros.
- **Conformidade com Padrões de Codificação Groovy**:
  - Implementar regras que verifiquem se o código Groovy segue as melhores práticas da linguagem.
  - Garantir a legibilidade e a manutenção do código Groovy.
- **Validação de Dados**:
  - Implementar regras para verificar a validação adequada de dados de entrada e saída.
- **Documentação**:
  - Criar regras que exijam a documentação adequada para cada elemento do projeto, como descrições detalhadas para canais etc.
- **Detetar dependências entre pacotes**
  - Fazer a deteção automática de dependências entre pacotes utilizados no Groovy.
- **Utilizar API existente para verificar as vulnerabilidades dos ficheiros JAR**
  - Por exemplo, utilizar Maven para fazer a verificação da existência de vulnerabilidades dos ficheiros JAR.
- **Detetar se há a inclusão de informação sensível**
  - Por exemplo, a inclusão de ficheiros que incluam credenciais deve ser evitada.

5 Conclusão

Este estudo inicial representa um passo importante na busca pela melhoria na revisão e governança das interfaces de integração desenvolvidas na plataforma SAP CPI. A criação do CPILint como uma aplicação de revisão automatizada desempenhará um papel fundamental na melhoria do desempenho, segurança e conformidade dessas integrações.

Atualmente, a revisão manual de artefatos criados no SAP CPI é um processo que consome tempo e recursos. A introdução do CPILint como ferramenta de automação simplificará esse processo, permitindo uma revisão mais rápida e eficiente. Isso resultará em economia de tempo e recursos valiosos para a equipe de desenvolvimento e garantirá que as integrações estejam em conformidade com as melhores práticas da SAP.

Além disso, a capacidade do CPILint de aplicar um conjunto de regras predefinidas, com foco em segurança, conformidade e desempenho, é um avanço significativo. No entanto, para atingir plenamente os objetivos deste projeto, é fundamental expandir o conjunto de regras do CPILint, especialmente em relação ao código Groovy. Isso garantirá que o CPILint seja uma ferramenta abrangente que pode cobrir uma ampla gama de melhores práticas recomendadas pela SAP.

Essa expansão contribuirá para a criação de uma plataforma de gestão de processos de integração completa e eficaz, que atenderá às necessidades de nossos clientes. Ao concluir este projeto, estaremos mais perto de fornecer uma solução que otimizará os fluxos de trabalho e a qualidade das integrações em contexto empresarial, ao mesmo tempo em que economiza tempo e recursos preciosos.