КОНЦЕПЦИЯТА ЗА МНОГОКРАТНА УПОТРЕБА

СУ „Св. Климент Охридски“
Факултет по Математика и Информатика
Увод в Софтуерното Инженерство

Съдържание

- Въведение
  - Понятие за многократна употреба
  - Видове
- COTS продукти
- Компонентно-базирано софтуерно инженерство
- Услуги и архитектура, ориентирана към услуги
- Шаблони на проектирането
- Програмна генерация
- Моделно-управлявана разработка

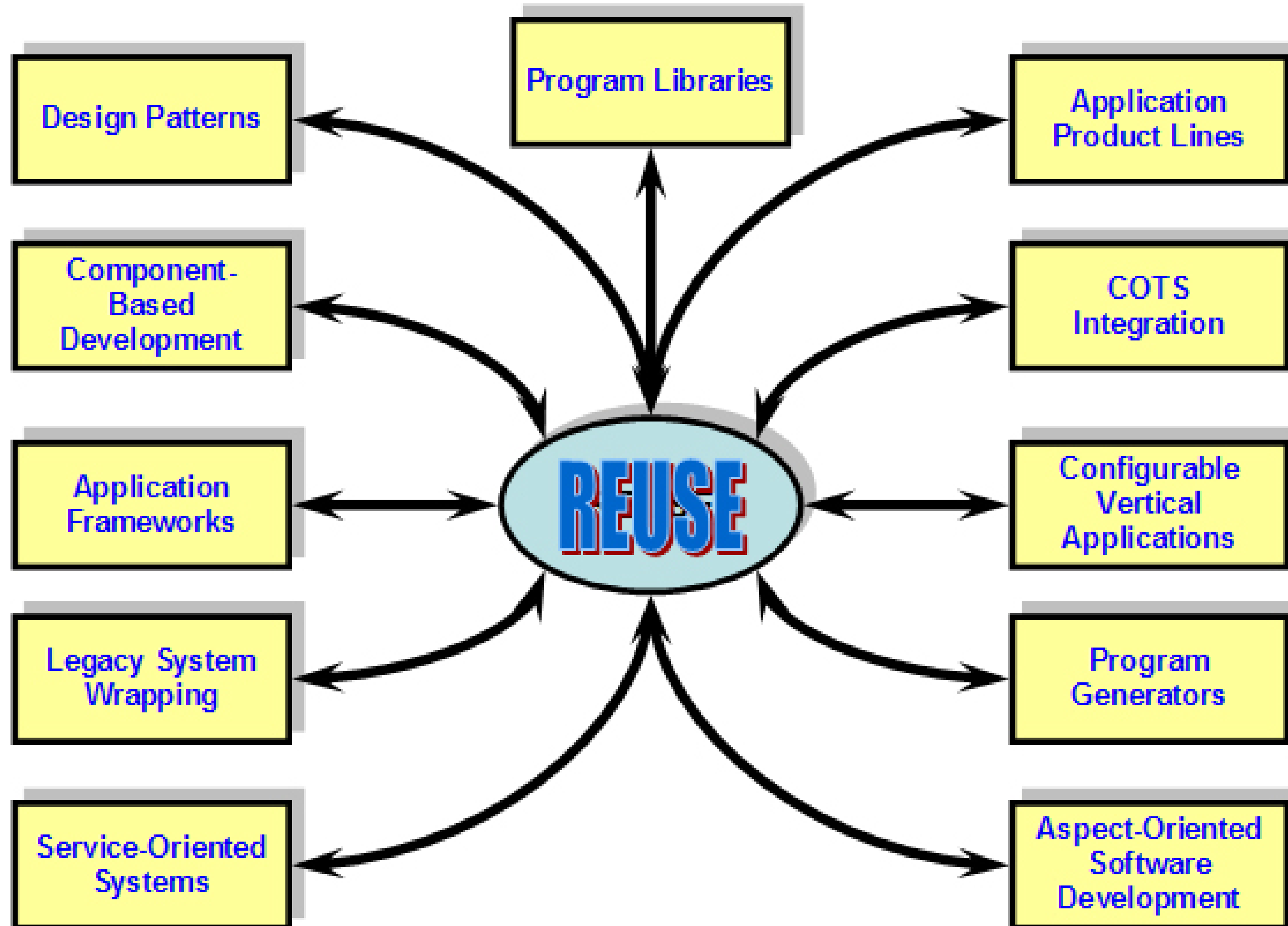
Понятие за многократна употреба

- В повечето инженерни дисциплини (в хардуерния свят) системите се проектират чрез композиране на вече съществуващи компоненти
- В софтуерното инженерство първоначално акцентът е бил поставен върху писането на код
- В последно време се цели производството на по-добри продукти, по-бързо и на по-ниска цена.
- Необходими са систематични правила за многократна употреба

Възможности за многократна употреба на софтуер

- Компоненти
- Възможен е подход на различни нива
  - Методи/функции – **Обектен подход**
  - Отделни компоненти
  - Цели приложения и системи

Многократна употреба

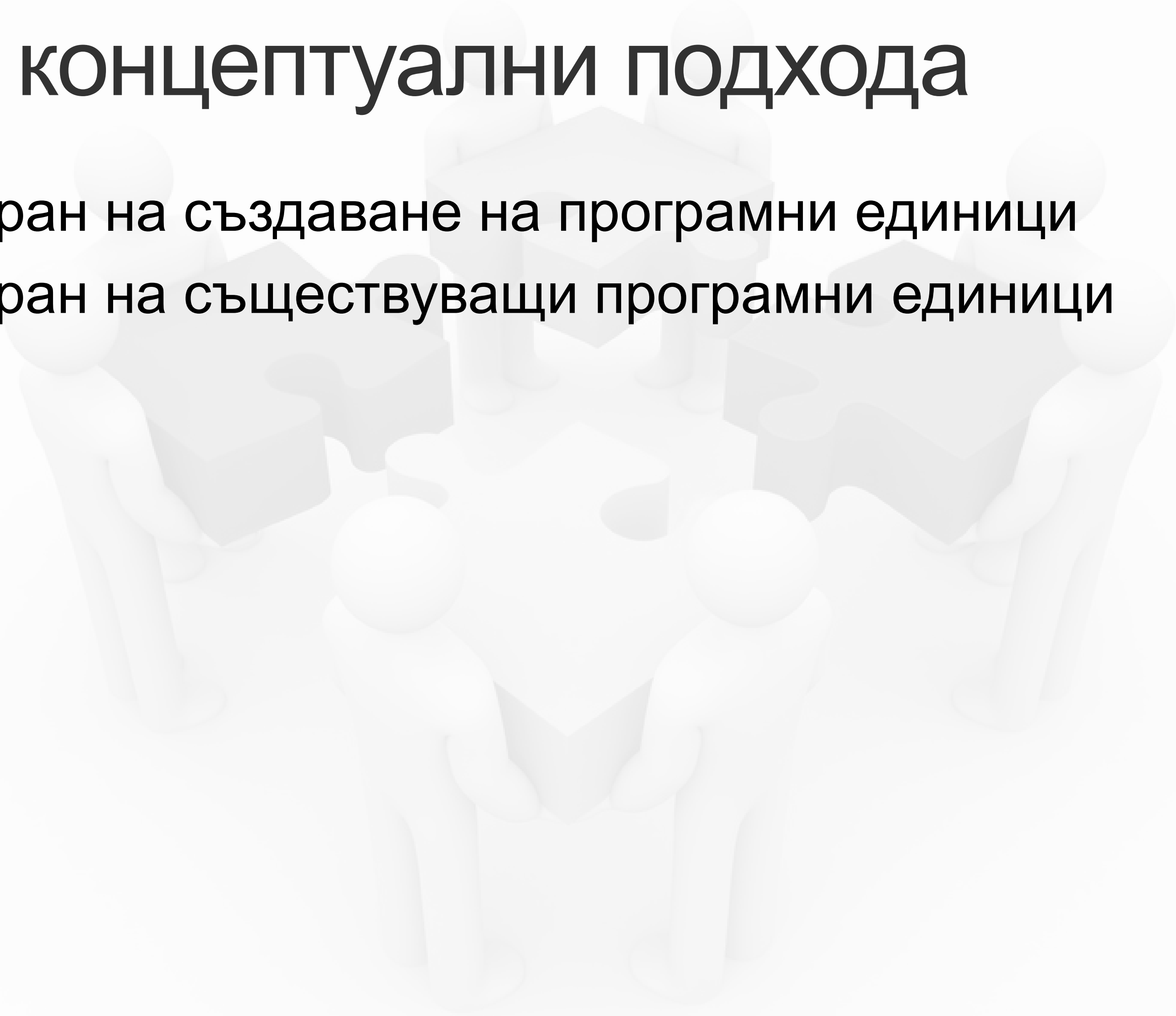


Ползи от многократната употреба

- Увеличава се предвидимостта по отношение на
  - Качеството на софтуера
  - Процеса за разработка
- Увеличаване на скоростта на разработка
- По-ефективно използване на ресурсите (спестява се повторението на рутинни дейности)
- Придържане към стандартите

- Около 15% от кода, който се пише, е за нови цели
  - Теоретично, може да се използват отново до 85% от него
- Няма друг начин да се създават големи приложения
- За съжаление, не всичко и не винаги е толкова идеално - няма универсален подход за многократна употреба

Два концептуални подхода



- Базиран на създаване на програмни единици
- Базиран на съществуващи програмни единици

Многократна употреба, базирана на съществуващи програмни единици

- COTS (Component-Off-The-Shelf) продукти и компонентно-базирано разработване
- Аспектно ориентирано разработване
- Архитектура, ориентирана към услуги

Многократна употреба, базирана на COTS

- COTS системите обикновено са цели приложения, които предлагат API (Application Programming Interface)
- Практически приложимо в широк кръг области
- Основно предимство – по-бърза разработка и евентуално на по-ниска цена

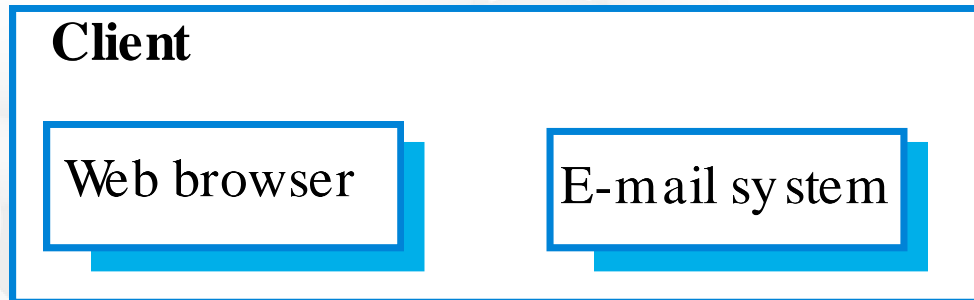
Основни въпроси

- Кои COTS продукти предлагат най-подходяща функционалност?
  - Възможно е да съществуват повече от един потенциални кандидати за интегриране в системата
- Как ще се обменят данни между модулите?
  - Всеки модул има собствен формат и структура на данните
- Кои части от продукта ще се използват
  - Най-често се предоставя повече функционалност, от това което е нужно. Понякога е нужно да се предотврати достъпа до ненужната функционалност.
  - Обвивен модул (wrapper)

Система за електронни поръчки

- На страната на клиента се използват стандартни програми за електронна поща и навигация в web
- На страната на сървъра се използват
  - съществуваща вече във организацията система за поръчки
  - COTS система за електронна търговия
  - COTS система за електронна поща
  - Всички те изискват реализацията на адаптер, за да може да си обменят данни

Система за електронни поръчки



Server

E-commerce system

Adaptor

Ordering and invoicing system

E-mail system

Adaptor

Наследени (Legacy) компоненти и системи

- Съществуващите системи, които доказано работят и са полезни, може да бъдат използвани като компоненти в по-големи системи
- За целта е необходимо да се напише т.нар. *обвивка* (*wrapper*), през интерфейсите на който да се осъществява цялата комуникация с наследения компонент
- По принцип това е скъпо, но в повечето случаи е много по-изгодно, отколкото да се пренапише наследения компонент (система)

Проблеми при използването на COTS продукти

- Липса на контрол върху функционалността и производителността
  - Възможно е понижение на реалната ефективност на COTS продуктите
- Понякога интеграцията на COTS продукти е доста труден процес
- Липса на контрол върху развитието и появата на нови версии на продуктите
- Недостатъчна поддръжка от производителите на COTS продуктите

Компонентно-базирана разработка

- Основни понятия
  - Компонент
  - Компонентно-Базирано Софтуерно Инженерство (КБСИ)
- Компонентните са по-общо понятие от обектите
  - Един компонент може да е изграден от един или няколко обекта
- Може да се изпълняват самостоятелно
- Чрез компонентите се постига по-високо ниво на абстракция

Софтуерен компонент - дефиниция

Градивна единица на софтуерните системи, вътрешната структура и процесите, на която са видими само за интегратора на системата, но не и за останалите компоненти. Компонентите могат да осъществяват връзка помежду си единствено чрез входно-изходни точки, наречени интерфейси (interfaces).

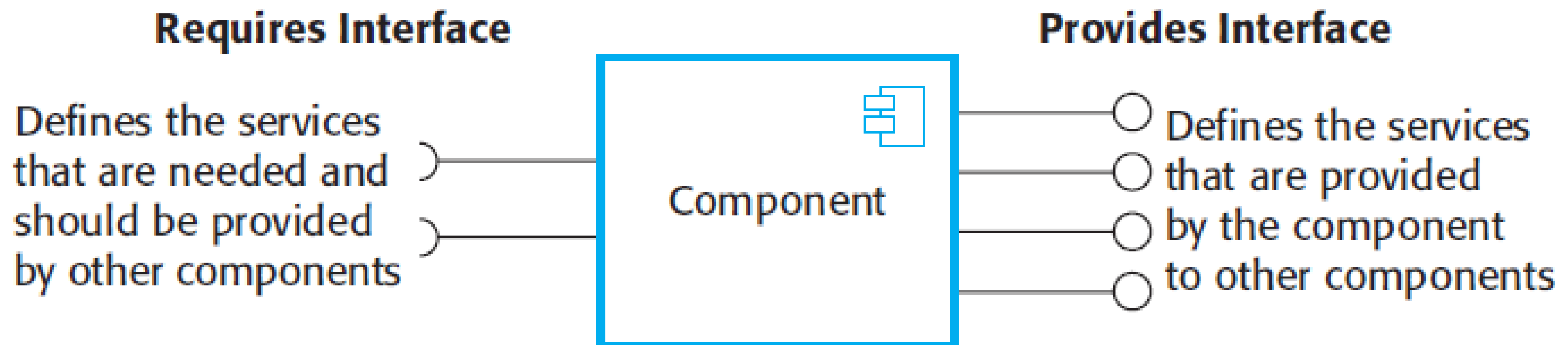
Компоненти

- Предоставят една и съща услуга, без значение от платформата на която се изпълняват или програмния език, на който са реализирани
- Независими самостоятелно изпълними единици.
  - Целта е компонентите да не се прекомпилират, преди интеграцията им с други компоненти

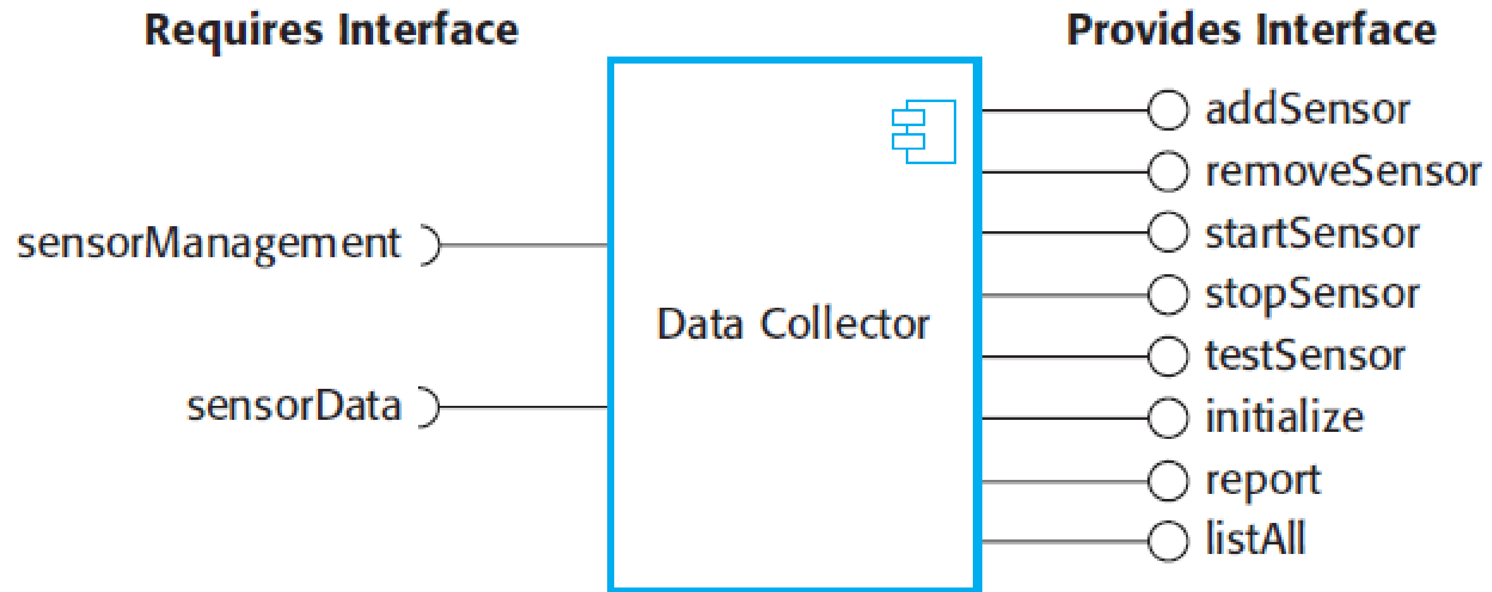
Интерфейси на компонентите

- Входен (requires) интерфейс
  - Дефинира какво трябва да се предостави на компонента, така че той да се изпълни според спецификацията
- Изходен (provides) интерфейс
  - Дефинира услугата (или резултата от нея), която компонентът предоставя, в следствие на изпълнението му и при условие, че са удовлетворени изискванията на входния интерфейс

UML – компонентни диаграми



UML – компонентни диаграми



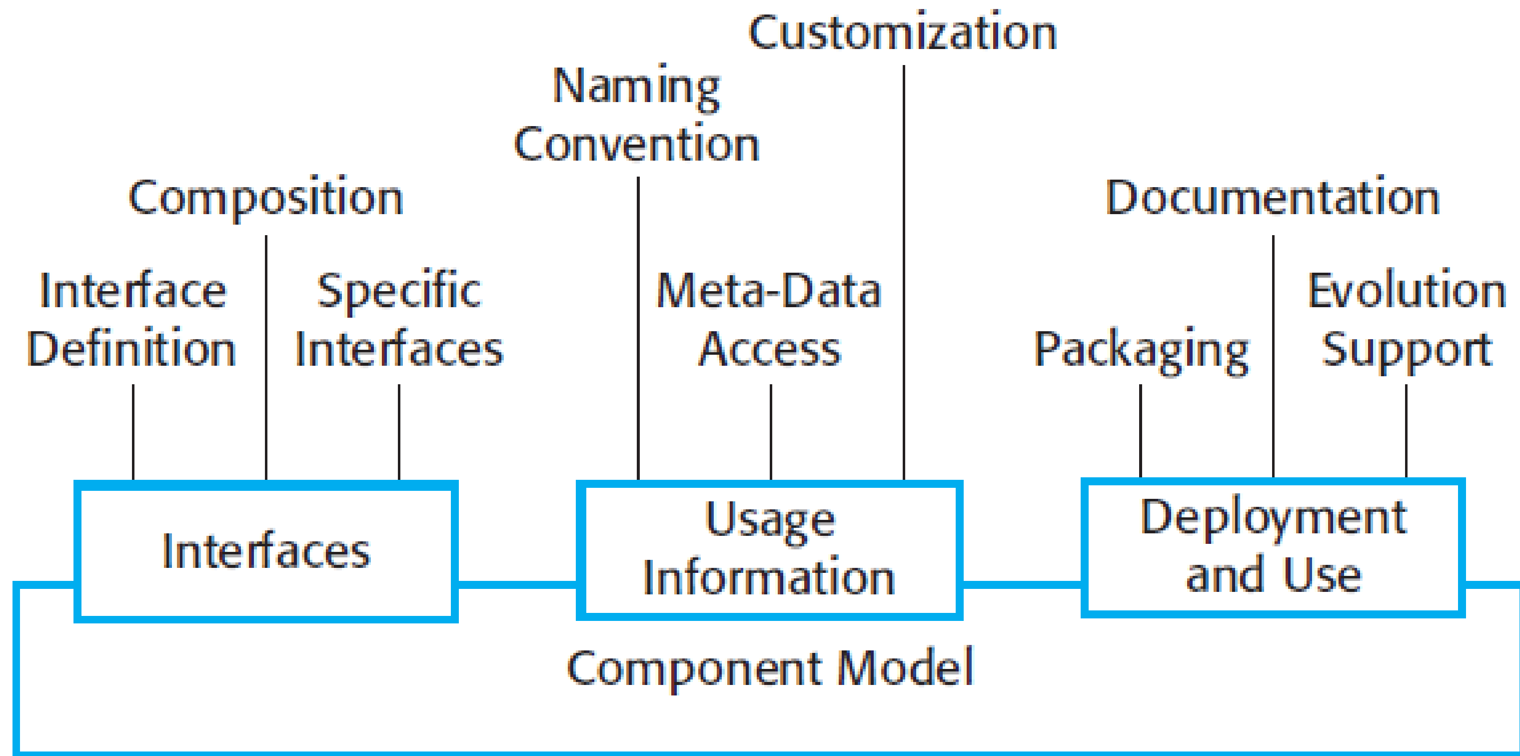
Компоненти и COTS

- COTS продуктите и компонентите са две много близки понятия
- Може да се каже че COTS е частен случай на Компонентно-Базираното Софтуерно Инженерство (КБСИ)

Компонентни модели

- Дефиниция на стандарт за реализация, употреба, документация и метод за разпространение на компоненти
- Определя правилата за дефиниция на интерфейсите
- Примери за компонентни модели
  - EJB (Enterprise Java Beans)
  - COM+, .NET
  - CCM (Corba Component Model)

Basic elements of a component model



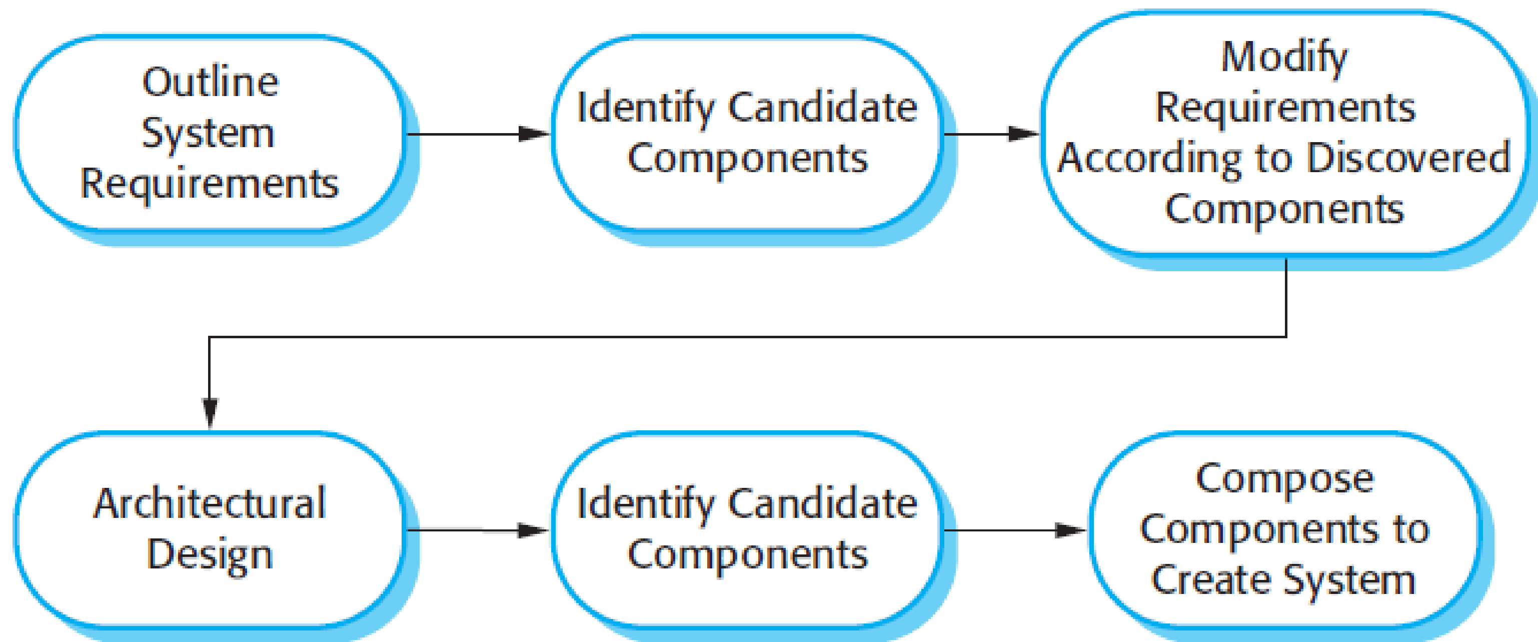
Процес за разработка в КБСИ

- За разлика от основните процеси, тук има някои разлики, които се дължат на необходимостта от търсене и оценка на компоненти за интегриране в системата – т.нар. *Evaluation phase*
- Най-общо съществуват 3 вида компоненти
  - Комерсиални компоненти
  - Компоненти с отворен код
  - Компоненти налични от предишни проекти

COTS

- Препоръките са да не се променя кода на компонента, дори да има достъп до него

Процес в КБСИ



Разработка на компоненти в КБСИ

- Какво означава един компонент да може да се използва многократно?
  - Да е разработен с тази идея
- Дали разходите за разработка на такъв компонент ще бъдат оправдани?

Компонент за многократна употреба

- Компонентът да е възможно най-обобщен
- Данните да са свързани със точно определени и често използвани понятия от проблемната област
- Да се прави валидация на входните данни
- Отстраняване на специфичните за системата/приложението публични методи.
- Добавяне на обработка на изключения
- Добавяне на интерфейс за конфигурация, чрез който компонентта да се адаптира
- Интеграция на необходимите външни компоненти, за да се намалят зависимостите.

- И още - http://hoskinator.blogspot.com/2006/06/10-tips-on-writing-reusable-code.html

- Съществува баланс между многократната употреба (reusability) и приложимостта (usability)
  - Колкото по-обобщен е даден интерфейс, толкова е по-вероятно да се използва многократно, но тогава е по-сложен и труден за употреба

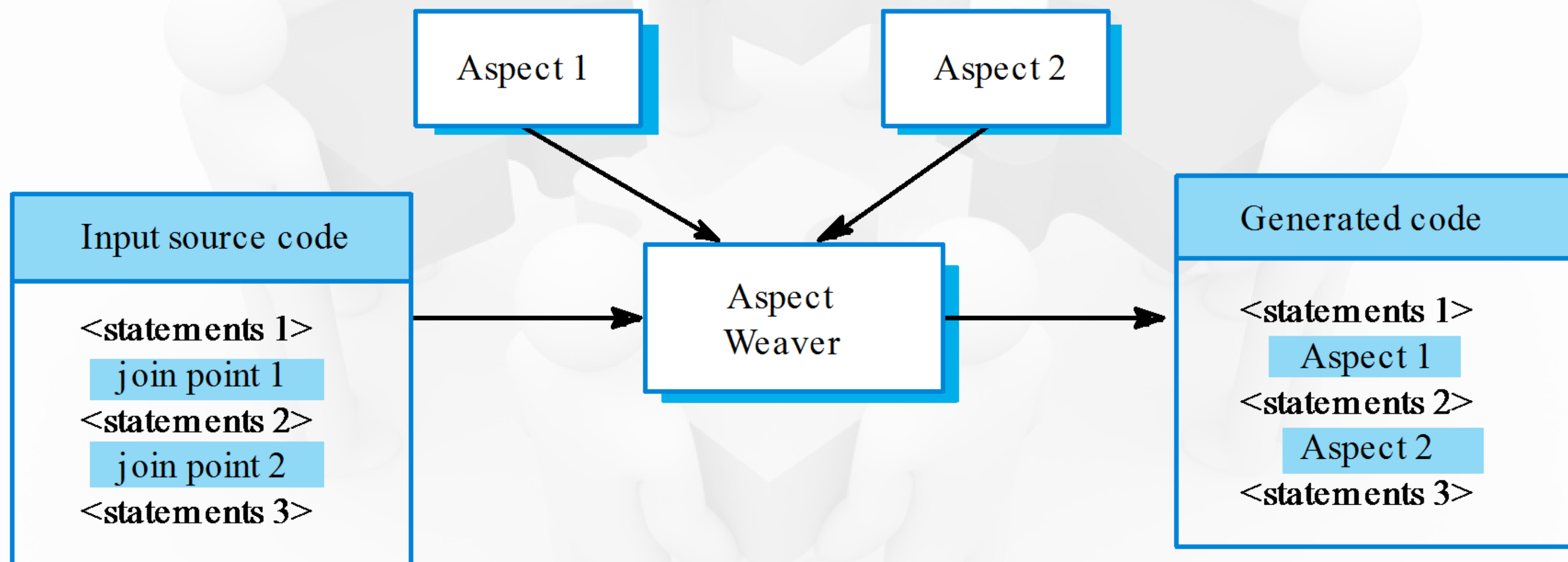
Поточна/продуктова линия (product line)

- Приложение с обобщена функционалност, което може да се адаптира и използва в специфична област
- Описва се като архитектура
- Адаптацията предполага някои от следните подходи
  - Конфигурация на компонент или система
  - Добавяне на нови компоненти в системата
  - Избор на компоненти от предварително подготвени библиотеки
  - Промяна на подсистеми
- Позната е още под понятието фамилия от приложения (application family)

Аспектно-ориентирано разработване

- Аспектно-ориентираното разработване засяга т.нар. разделяне на функциите (separation of concerns).
- Често е много трудно те да се отделят чрез стандартните методи на проектиране. Например:
  - Всички компоненти трябва да наблюдават състоянието си
  - Всички компоненти трябва да изпълняват политики по сигурността
- Припокриващата се функционалност се разработва като аспекти, които динамично се вмъкват в програмата

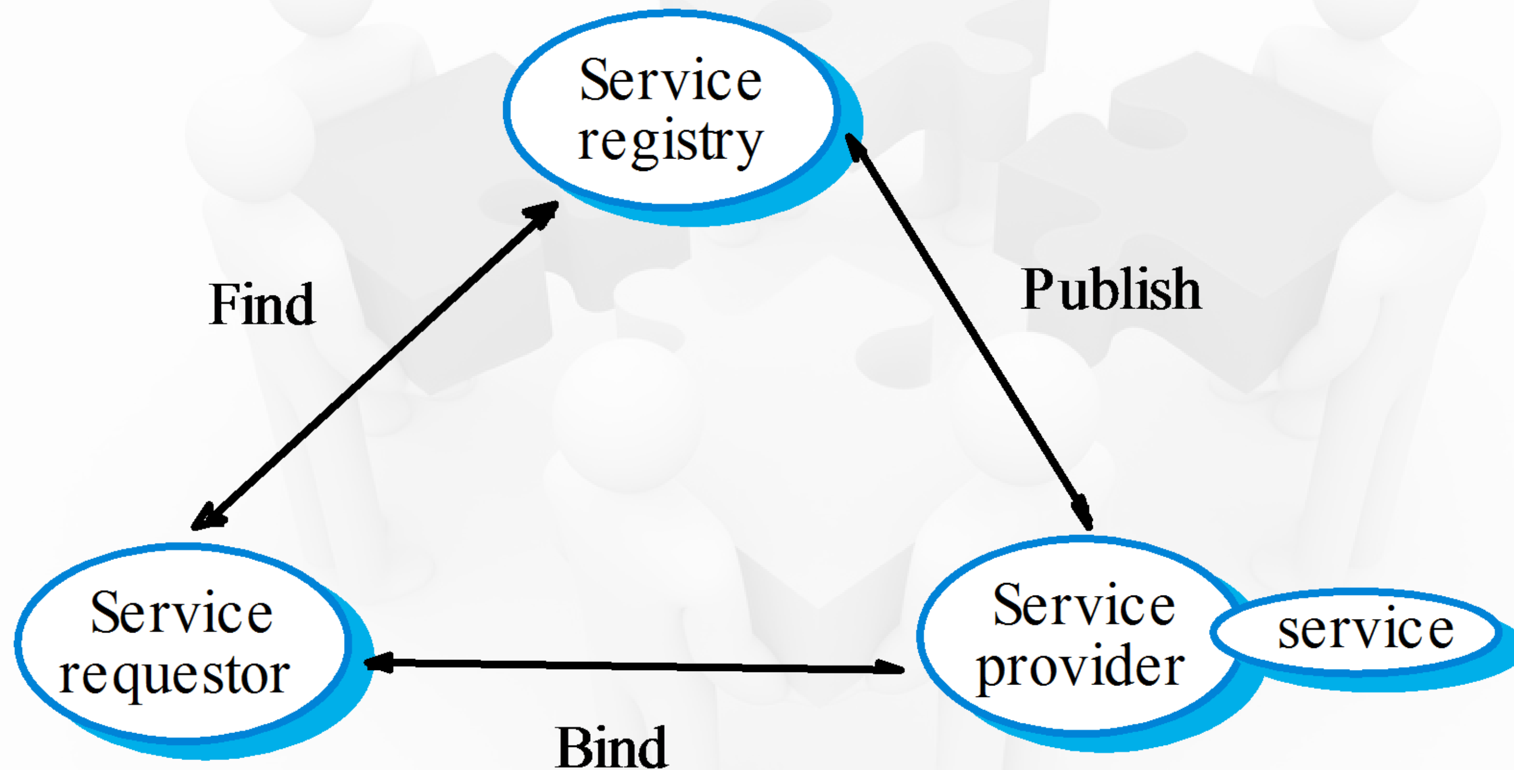
Схема за аспектно-ориентирано разработване



Архитектура ориентирана към услуги

- Основава се на идеята за услуги, достъпни “отвън”
- Уеб-услугата е стандартизиран подход за осигуряване на многократна употреба на някаква функционалност, чрез достъп през интернет

Уеб-услуги



Стандарти за уеб-услуги

- SOAP - Simple Object Access Protocol
- WSDL - Web Services Description Language
- UDDI - Universal Description, Discovery and Integration

- Базирани са на XML
  - Услугите може да се изпълняват на различни платформи и да са написани на различни програмни езици

XML (eXtensible Markup Language)

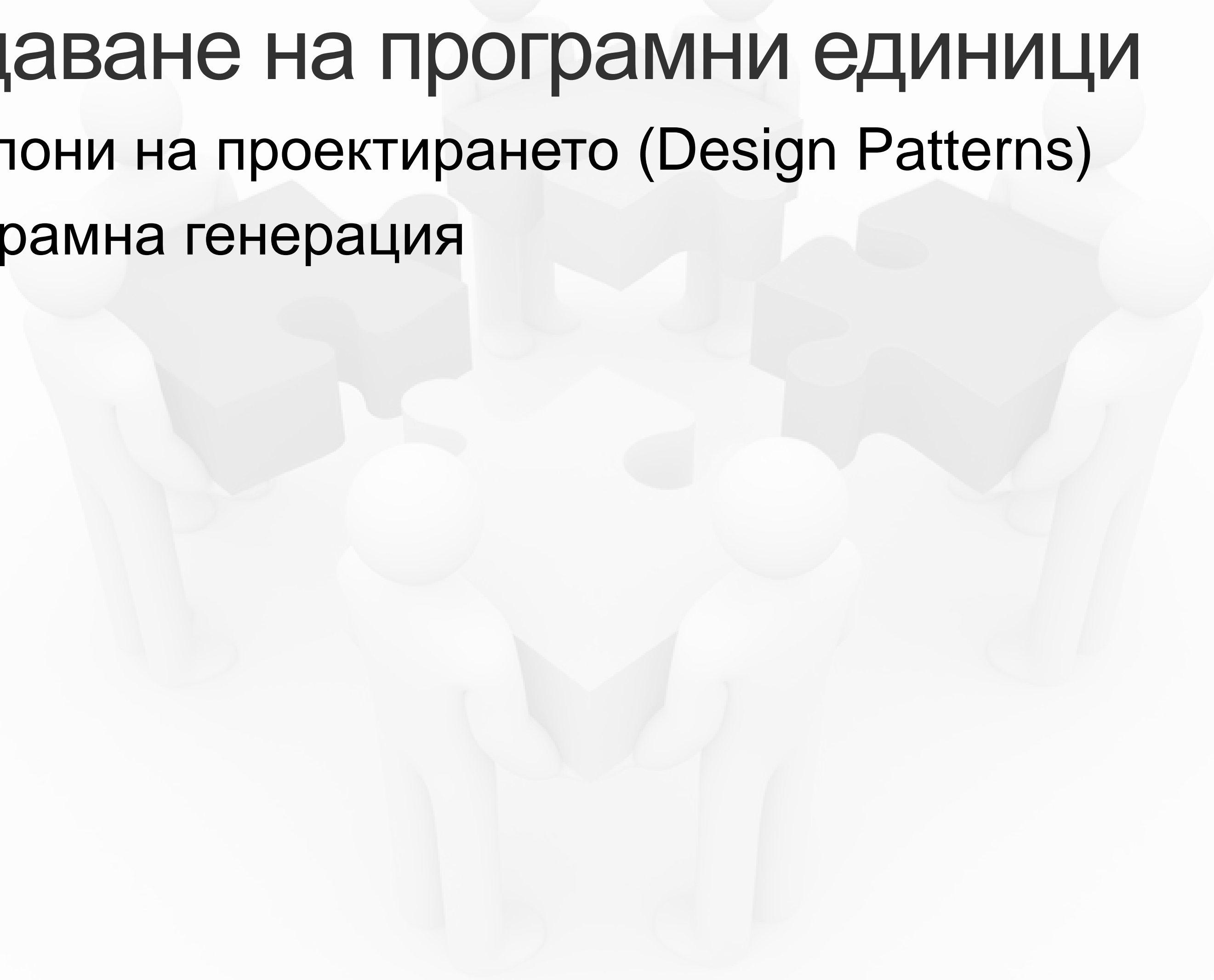
```
<note>
   <to>Toni</to>
   <from>Jani</from>
   <heading>Reminder</heading>
   <body>Don't forget me this
   weekend!</body>
</note>
```

XML (eXtensible Markup Language)

```
<breakfast_menu>
    <food>
        <name>Belgian Waffles</name>
        <price>$5.95</price>
        <description>
                two of our famous Belgian Waffles with
    plenty of real maple syrup
        </description>
        <calories>650</calories>
    </food>
    <food>
    …..
    </food>
</breakfast_menu>
```

Многократна употреба, базирана на създаване на програмни единици

- Шаблони на проектирането (Design Patterns)
- Програмна генерация



Шаблони на проектирането

- Шаблонът на проектирането (design pattern) е техника за многократна употреба на някакво абстрактно знание за дадена задача и решението й
- Описание на задачата и най-съществените моменти от решението
- Абстрактност – предполага възможност за прилагане в различни ситуации
- Най-често използваните свойства на класовете са наследяване и полиморфизъм

История на шаблоните за проектиране

- Обектно-ориентирани езици за програмиране
- Обектно-ориентиран процес за разработване
- Шаблони на проектирането
- Защо се използват шаблони на проектирането
  - Дефинирани са на общодостъпен език
  - Независими са от езика за програмиране

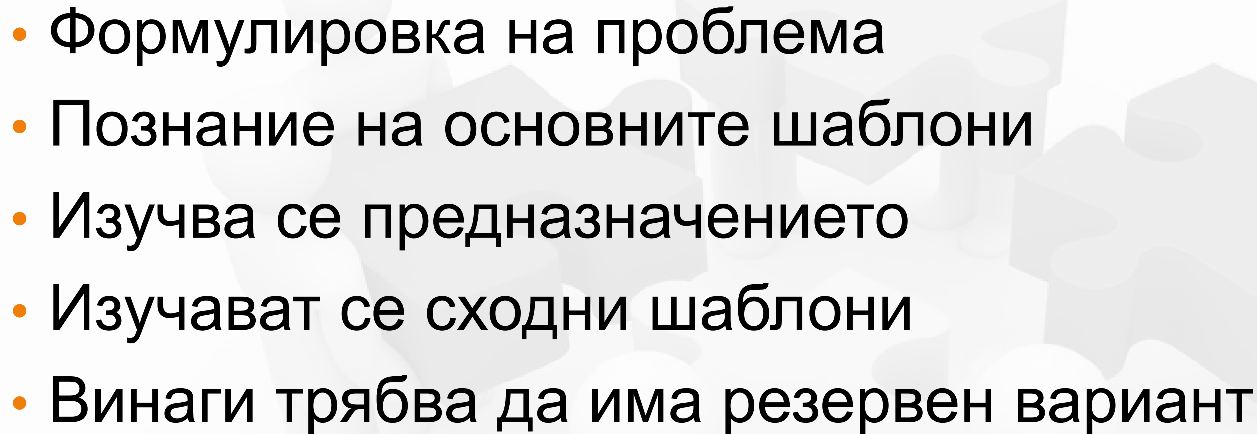
Класификация според предназначението

- Създателни (creational)
  - Използват се за създаване на обектите
- Структурни (structural)
  - Използват се за улесняване на интеграцията на класове или обекти
- Поведенчески (behavioural)
  - Представят методи за комуникация между обектите

Класификация според обхвата

- Шаблони на класовете
  - Засягат отношения между класовете, които не се променят след компилацията
- Шаблони на обектите
  - Засягат отношения между обекти. Тези отношения може да се променят по време на изпълнение

Как се избира шаблон



- Формулировка на проблема
- Познание на основните шаблони
- Изучва се предназначението
- Изучават се сходни шаблони
- Винаги трябва да има резервен вариант

Примери

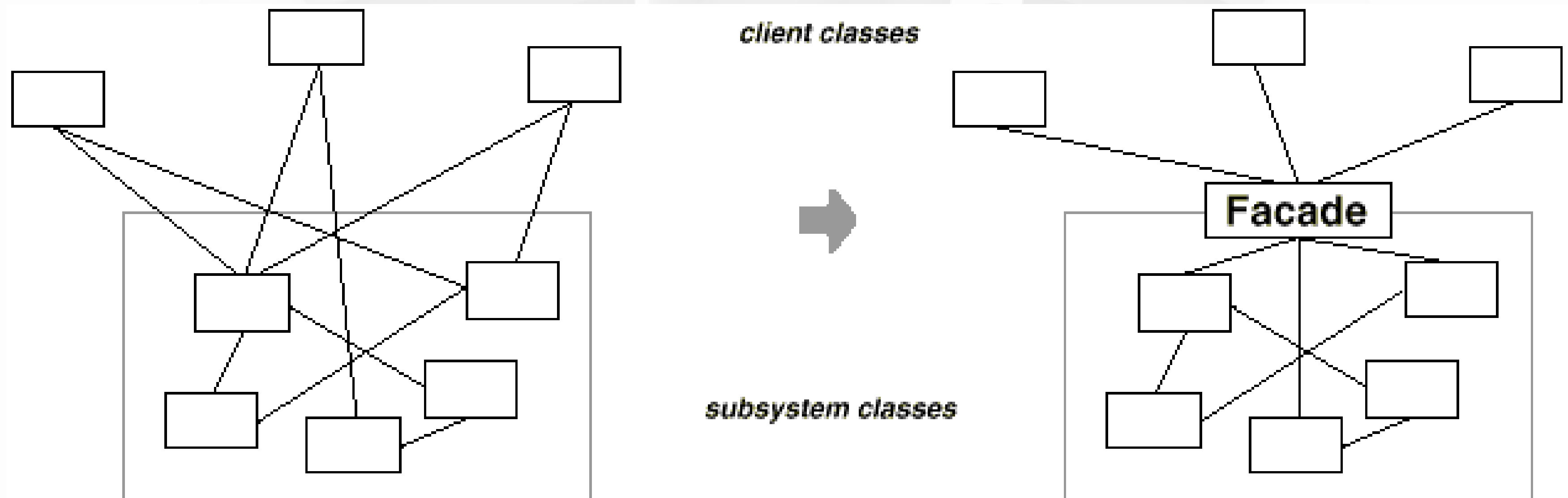
- Фасада (Facade)
- Единичен екземпляр (Singleton)
- И много други – съществуват над 30 шаблона

- Описани са от т.нар. “Gang of the four”
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994.

Шаблон “Фасада” (Façade)

- Осигурява единен достъп към набор интерфейси на дадена система
- Дефинира интерфейс от по-високо ниво, който прави системата по-лесна за употреба

Мотивация



Роли

- Фасада
  - Указано й е кои класове на подсистемата са отговорни за дадена заявка
  - Предава клиентските заявки към съответните обекти
- Класове на подсистемата
  - Реализират функционалността
  - Нямат никаква представа за фасадата – нямат указатели към нея

Реализация (1)

```
public interface Store {
  public Goods getGoods();
}

public class FinishedStore implements Store {

  public Goods getGoods() {
      FinishedGoods finishedGoods = new
      FinishedGoods();
      return finishedGoods;
  }
}
```

Реализация (2)

```
public class StoreKeeper {
        public RawGoods getRawGoods() {
            RawStore store = new RawStore();
            RawGoods rawGoods = (RawGoods)store.getGoods();
    return rawGoods;
}
public PackingMaterialGoods getPackingMaterialGoods() {
    PackingStore store = new PackingStore();
    PackingGoods packingGoods = (PackingGoods)store.getGoods();
    return packingGoods;
}
public FinishedGoods getFinishedGoods() {
    FinishedStore store = new FinishedStore();
    FinishedGoods finishedGoods = (FinishedGoods)store.getGoods();
    return finishedGoods;
}
}
```

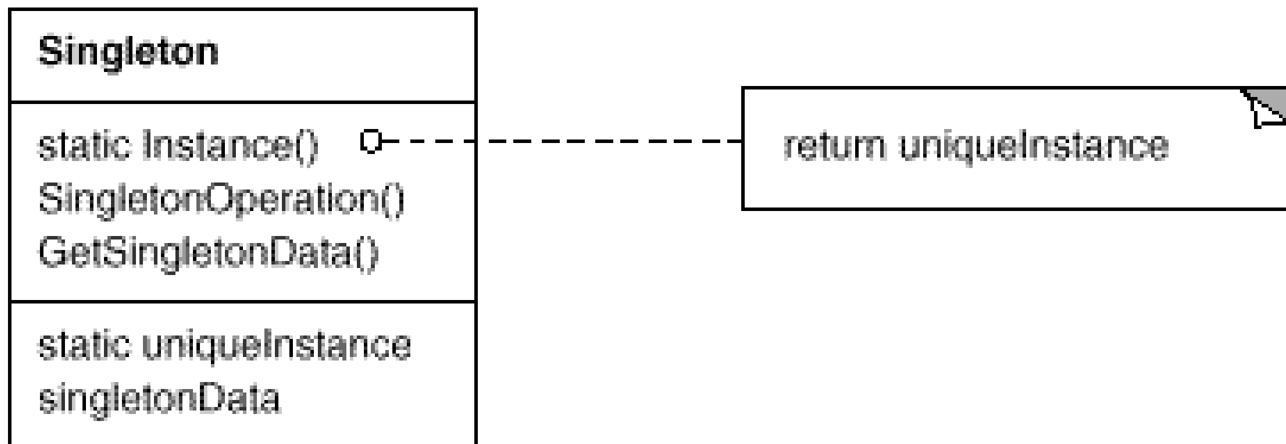
Реализация - клиент

```
public class Client {
	public static void  main(String[] args)
{
		StoreKeeper keeper = new
	StoreKeeper();
		RawGoods rawGoods =
			keeper.getRawGoods();
	}
}
```

Шаблон Singleton

- Понякога се налага даден клас да има точно един екземпляр
- Примери
  - Комуникационен обект
  - Обект за управление на графичните прозорци
  - Файлова система
- Самият клас трябва да следи единствения си екземпляр

Шаблон Singleton



Реализация - Singleton

- “Забрана” на създаването на обекти
- Механизъм за връщане на единствения екземпляр

```
public class SingletonObject {
 private static SingletonObject ref;

    private SingletonObject() {

    }

    public static SingletonObject
getSingletonObject() {
      if (ref == null)
                ref = new SingletonObject();
      return ref;
    }
}
```

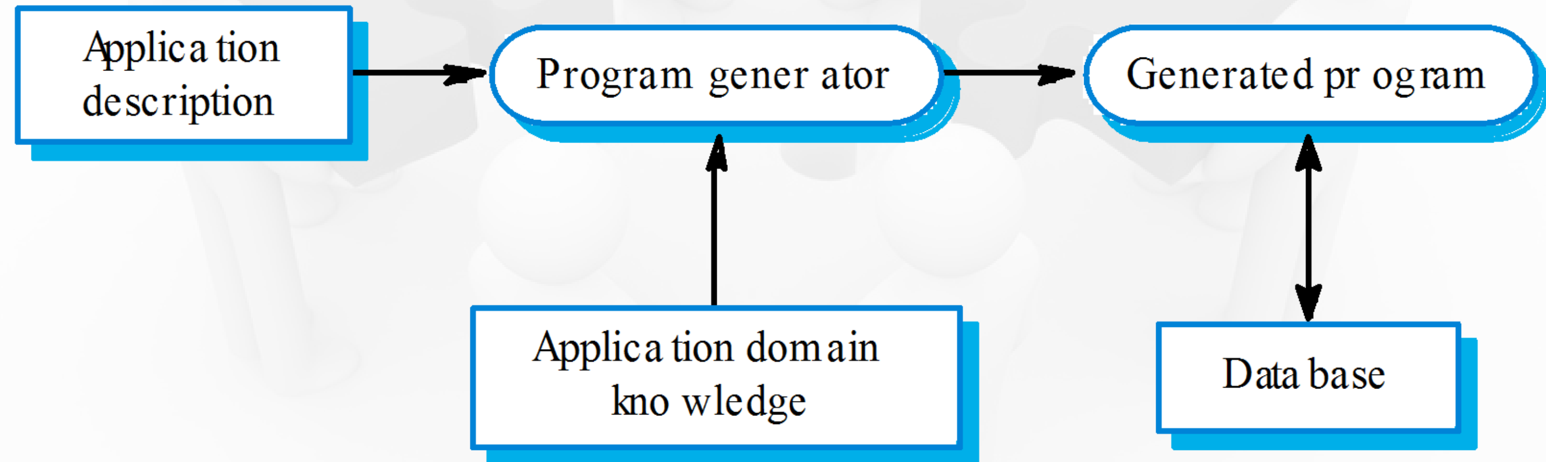
Програмна генерация

- Програмните генератори използват вече дефинирани шаблони и алгоритми
- Те са вградени в генератора и може да се параметризират.
- Програмата се генерира автоматично
- Програмната генерация е възможна, когато може да се дефинира съответствие между понятията от приложната област и програмния код
- Използва се специфичен (за проблемната област) език

Програмна генерация

- Ефективен метод за разработка
- Приложимостта му е в ограничен набор от приложни области
- От гледна точка на крайния потребител е по-лесно в сравнение със стандартните компонентни подходи

Схема на използване на програмния генератор

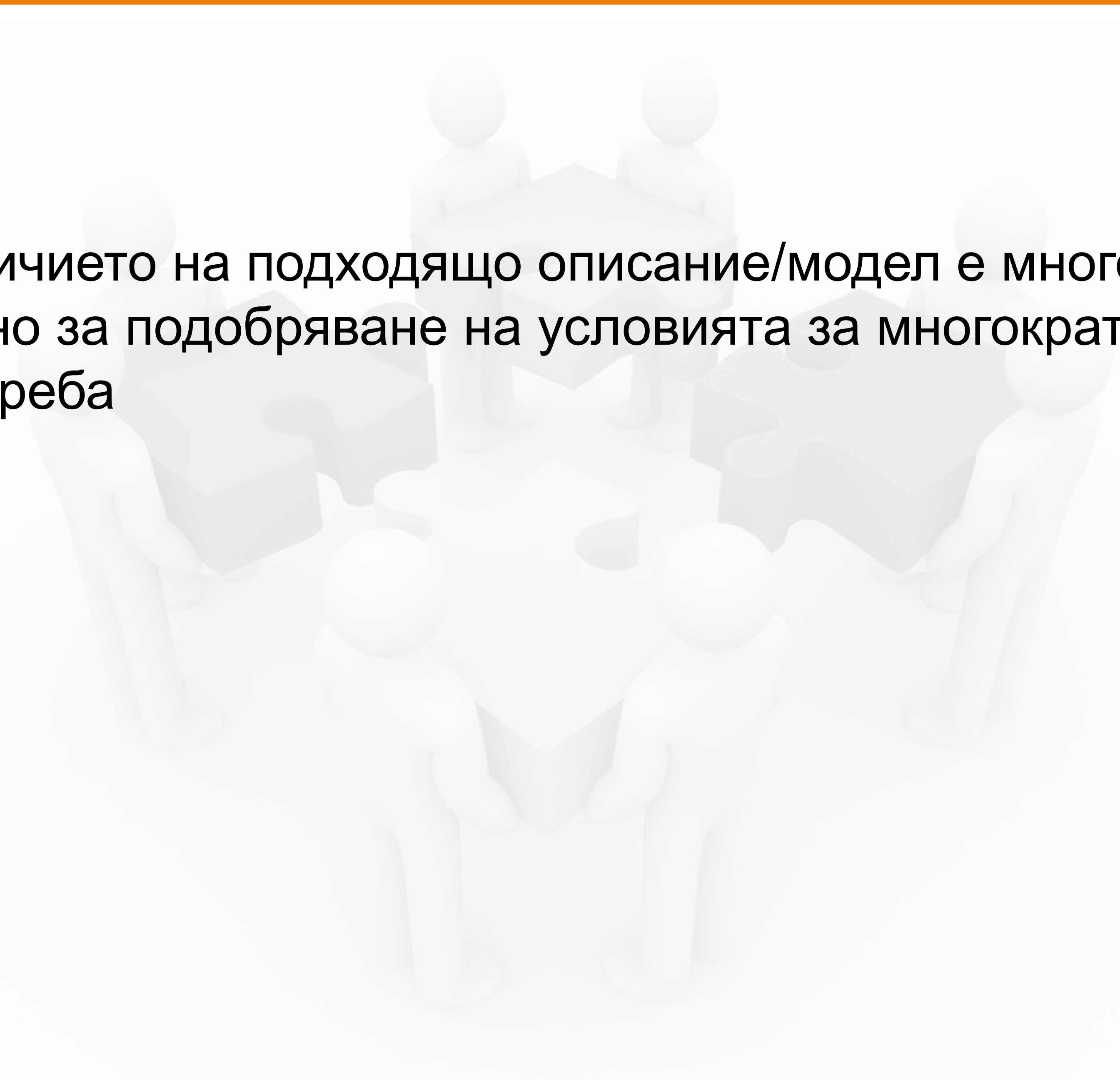


Обобщение

- Ползи от многократната употреба
  - Увеличена надеждност
  - Намален риск
  - По-ефективно използване на човешките ресурси
  - Съобразяване със стандартите
  - Ускорена разработка
  - Намалена цена

Обобщение

- Недостатъци на многократната употреба
  - Увеличена цена на поддръжката
  - Липса на достатъчно инструменти
  - Споделена отговорност
  - Трудност при намирането на подходящи компоненти



- Наличието на подходящо описание/модел е много важно за подобряване на условията за многократна употреба

Моделно ориентирана разработка

- Model-driven engineering (MDE)
- Подход, при който основния резултат от разработката са модели, а не програми.
- Кодът на програмите, които се изпълняват на някаква платформа (хардуерна/софтуерна) се генерира автоматично от моделите.
- Чрез MDE се увеличава нивото на абстракция в софтуерното инженерство

Приложимост на MDE

- Поредният *silver bullet*?
- Плюсове
  - Намалява се необходмостта от познания по специфични детайли за код, платформи и т.н.
  - Генерацията на код, може да намали разходите за мигриране на системите към нови платформи
- Минуси
  - Моделите не винаги са точни по отношение на реализацията
  - Разходите заразработка на съответните генератори на код за нови платформи може да са значителни

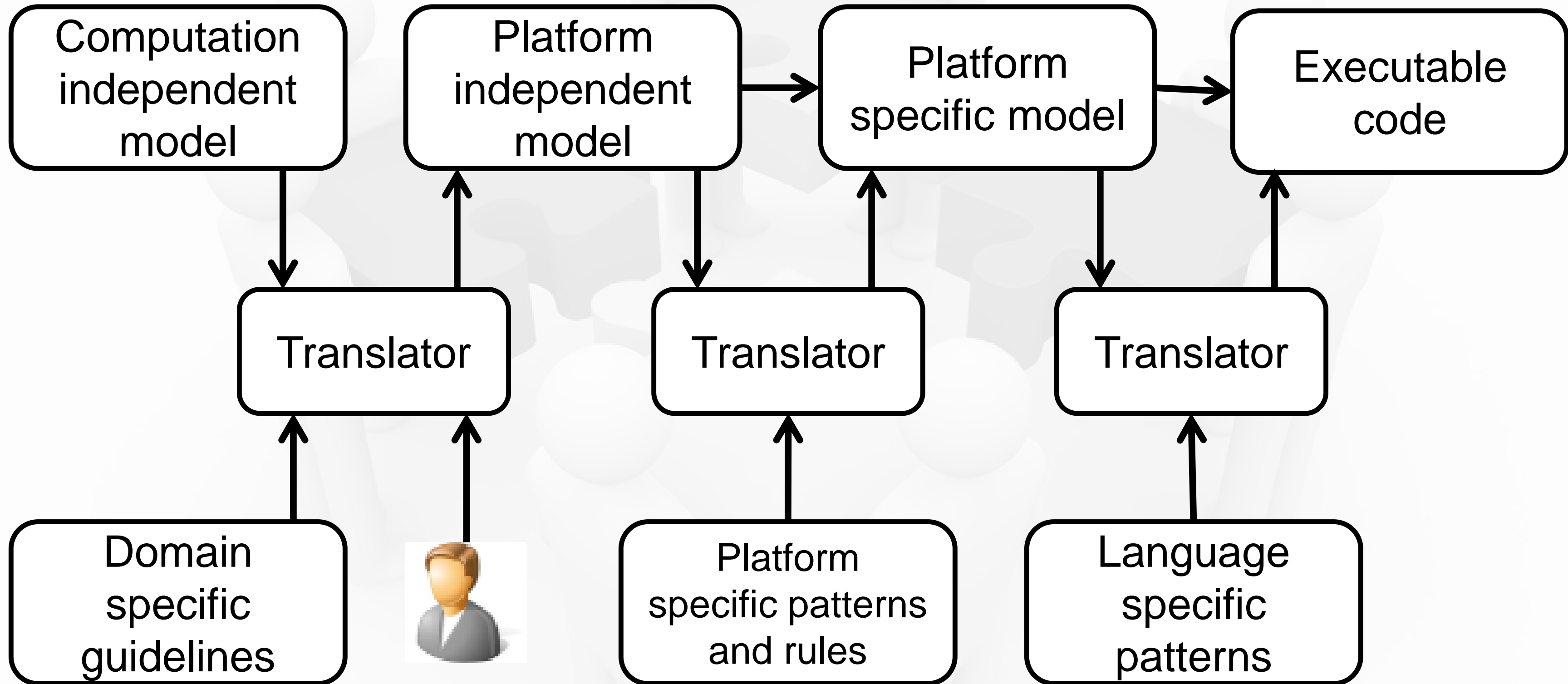
Архитектура ориентирана към модели

- Използва някои диаграми на UML, за да се опише и моделира софтуерната система
- Създават се модели с различно ниво на абстракция
- Целта е да се генерира работеща програма автоматично, без ръчна намеса.

Видове модели в MDE

- A computation independent model (CIM)
  - Моделират се основните абстракции в приложната област - domain models.
- A platform independent model (PIM)
  - Моделира се изпълнението на програмата, без да се прави връзка с нейната реализация. Най-често такъв модел се прави със статични UML диаграми, които показват структурата на системата и как тя реагира на външни и вътрешни събития.
- Platform specific models (PSM)
  - Трансформации от PIM към отделен модел за всяко ниво. Всеки модел добавя някакви специфични за платформата детайли.

Трансформации в MDE



Трансформации в MDE