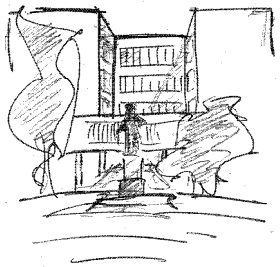
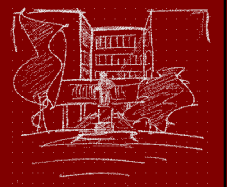
Развој софтвера 5



Саша Малков
Универзитет у Београду
Математички факултет
2023/2024

[P290]
Развој софтвера
Саша Малков



Тема 8
Принципи пројектовања софтвера

Основни принципи дизајна софтвера

- При пројектовању софтвера се руководимо неким начелима
  - ради лакшег разрешавања дилеме
  - ради лакшег доношња одлука
- Називају се ***принципи пројектовања***
  - зато што се односе на структурно пројектовање називају се и ***принципи дизајнирања*** или ***принципи дизајна***

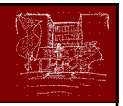
Основни принципи дизајна софтвера (2)

- Представљање принципа пројектовања је тешко одвојити од образаца за пројектовање
  - то су две веома испреплетане теме
  - сваки образац за пројектовање представља практичну примену једног или више принципа пројектовања
  - да бисмо разумели неки принцип, често ћемо као примере наводити неке обрасце или пример кода који подсећају на обрасце

Основни принципи дизајна софтвера (3)

- Принципи обично имају облик правила или препорука
- Циљ им је
  - да нам усмере пажњу на најважније аспекте проблема
  - да нам помогну да препознамо важније критеријуме квалитета у конкретним случајевима

Основни принципи дизајна софтвера (4)



- Различити аутори у својим књигама представљају различите скупове принципа
  - али су циљеви и примена тих принципа углавном међусобно сагласни
  - често су ти скупови *скоро* еквивалентни
  - али се ипак заједно надопуњују и лакше разумеју

Основни принципи дизајна софтвера (5)

- Обично се деле:
  - према пореклу:
    - ОО методологије
    - агилни развој
    - и друго
  - према домену примене:
    - распоређивање одговорности
    - груписање
    - раздвајање
    - и сл.
  - према значају
  - или комбиновано

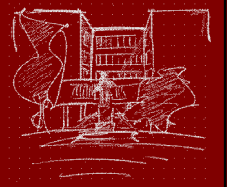
Основни принципи дизајна софтвера (6)

- Представићемо три скупа принципа:
  - **кључни принципи ОО дизајна**
  - **принципи додељивања одговорности**
  - **принципи обликовања целина**
- Бавићемо се главним аспектима представљених принципа
  - релативно укратко
  - детаљно разматрање би захтевало много више простора

[P290]

Развој софтвера

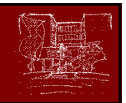
Саша Малков



Тема 8.1

Принципи пројектовања софтвера – Кључни принципи ОО дизајна

Кључни принципи ОО дизајна



- Обично се истичу као 5 кључних принципа ОО дизајна
  - у основним облицима потичу из ОО методологија
  - често се приписују Роберту Мартину
    - није их осмислио али их је утицао на њихову форму и груписање
  - обично се примењују на класе
    - али подједнако важе и за друге врсте целина (модули, компоненте, пакети, функције, методи,...)
  - означавају се скраћеницом ***SOLID***

Кључни принципи ОО дизајна (2)

- **Принцип јединствене одговорности**
  - *SRP – Single Responsibility Principle*
- **Принцип отворености и затворености**
  - *OCP – Open-Closed Principle*
- **Принцип заменљивости**
  - *LSP – Liskov Substitution Principle*
- **Принцип раздвајања интерфејса**
  - *ISP – Interface Segregation Principle*
- **Принцип инверзне зависности**
  - *DIP – Dependency Inversion Principle*

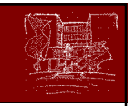
Принцип јединствене одговорности

- *SRP – Single Responsibility Principle*
- **Класа би требало да има само један разлог да се мења**
- Алтернативни облици:
  - један модул би требало да има једну намену
  - један модул би требало да има само један разлог за мењање
- Указује на значај јасног разлога за декомпозицију
  - једна компонента – једна функција
  - једна компонента – једна тачка променљивости
  - ...

Принцип јединствене одговорности (2)

- Ако се наруши
  - повећава се нестабилност дела зато што има више разлога да се мења
- Пример примене
  - образац Посетилац
    - хијерархија класа је одговорна само за структурирање целине
    - одговорности за различите додатне послове се распоређују на различите посетиоце

Принцип отворености и затворености



- *OCP – Open-Closed Principle*
- **Елементи софтвера би требало да буду отворени за проширивање и затворени за мењање**
- Структурни елементи софтвера би требало:
  - да могу да се проширују, ако се за тим укаже потреба
    - тако да се при томе углавном не мења њихов интерфејс
  - да се не мењају ако се мењају околности
    - и никако не би требало да се мења интерфејс

Принцип отворености и затворености (2)

- Уобичајена примена је употреба хијерархија класа
  - базна класа одређује апстрактан интерфејс
  - отвореност за проширивање се остварује простором за додавање нових класа у хијерархију
  - затвореност за промене се остварује стабилним интерфејсом базне класе
- Ако се наруши
  - повећава се нестабилност како конкретног елемента, тако и повезаних структурних елемената
  - обично за резултат има тзв. лавину промена

Принцип отворености и затворености (3)

- Пример примене
  - образац Посетилац
    - допуштено је проширивање хијерархије, додавањем нових класа, а без мењања базне класе
    - допуштено је додавање новог понашања, додавањем нових посетилаца
    - није дозвољено мењање интерфејса хијерархије – базне класе

Принцип заменљивости

- *LSP – Liskov Substitution Principle*
- **Подтипови морају да могу да замене базне типове**
- У суштини представља дефиницију појма подтипа, као типа који испуњава услов заменљивости:
  - ако за сваки објекат ***a*** типа ***A*** постоји неки објекат ***t*** типа ***T*** такав да за све програме ***P***, у којима се појављује тип ***T***, важи да се понашање програма ***P*** не мења када се објекат ***a*** замени објектом ***t***,
  - онда *C* представља подтип типа *T*
- Овај принцип је у основи хијерархијског полиморфизма

Принцип заменљивости (2)

- Иако изгледа једноставно, понекад је све осим тога
- Пример проблема је однос *Правоугаоник – Квадрат*
  - у питању је наизглед једноставан однос наслеђивања
  - примарни критеријум наслеђивања је понашање
    - квадрат је специјалан случај правоугаоника
    - значи, класа *Квадрат* би требало да наследи класу *Правоугаоник*
- Сваки аспект понашања *Квадрата* мора да се уклапа у општи шаблон понашања *Правоугаоника*
  - у супротном је нарушен принцип заменљивости
- Изгледа једноставно?

Принцип заменљивости (3)

- Нека у класи *Правоугаоник* постоји метод *постaвиШирину*
  - мења ширину
  - не мења висину
  - линеарно пропорционално мења површину
  - и даље све изгледа једноставно?
- Да ли би метод *постaвиШирину* у класи *Квадрат* требало да мења само ширину или и ширину и висину?
  - ако мења само ширину, онда то више *није Квадрат*
  - ако мења и ширину и висину, онда то више *није Правоугаоник*
    - мења висину, што се не дешава у случају *Правоугаоника*
    - мење површину квадратно пропорционално, а не линеарно

Принцип заменљивости (4)

- Решење претходног проблема није сасвим једноставно
  - Мора да се направи другачији интерфејс
    - који може да одржи стабилност у описаном случају
  - Уместо два одвојена метода *постaвиШирину* и *постaвиВисину* уводи се само један метод *постaвиШиринуИВисину*
    - ако аргументи метода нису сагласни, мора да се избацује изузетак
    - није идеално, јер представља сложенији интерфејс, али пружа сагласност са принципом заменљивости

Принцип раздвајања интерфејса

- *ISP – Interface Segregation Principle*
- **Клијенти (корисници) не би требало да буду приморани да зависе од метода (интерфејса) које не користе.**
- Основна идеја је да би интерфејси требало да буду заокружени и потпуни
  - ако неко користи интерфејс, онда би требало да најчешће користи практично цео интерфејс, а не само његов део

Принцип раздвајања интерфејса (2)

- Односи се првенствено на видове непожељних зависности
  - зависност корисника интерфејса од прешироког интерфејса
    - ако се често користи само део интерфејса, онда то сугерише да компонента има више одговорности
  - додавање елемената интерфејса читавој хијерархији иако је потребан само неким класама
  - додавање класе хијерархији зато што јој је важан део интерфејса

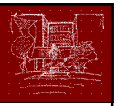
Принцип раздвајања интерфејса (3)

- Кључно питање је шта је одговорност које класе?
- Додатне одговорности
  - не би требало додавати као проширење одговорности
  - него као посебне класе
- Раздвајање интерфејса може да доводи до вишеструког наслеђивања
  - на пример, машина за прање и машина за сушење...

Принцип раздвајања интерфејса (4)

- Пример нарушавања овог принципа је писање класа са веома широким интерфејсом
  - На пример, класа ***std::string***
- Алтернатива широком интерфејсу класе је да имамо мноштво функција које раде са објектима класе?
- Нарушавање овог принципа се често толерише, када су у питању библиотеке
  - зато што су релативно стабилне
  - зато што имају специфичну улогу у софверу који их користи
  - али ако ипак дође до промене, онда је она веома скупа...

Принцип инверзне зависности



- *DIP – Dependency Inversion Principle*
- **Модули високог нивоа не смеју да зависе од модула ниског нивоа. И једни и други би требало да зависе од апстракција**
- Алтернативни облици:
  - Апстракције не смеју да зависе од појединости.
    Појединости би требало да зависе од апстракције
  - Програмирај према интерфејсу а не према имплементацији

Принцип инверзне зависности (2)

- При планирању и пројектовању апстрактног интерфејса, наравно, имамо у виду конкретне случајеве које он мора да подржи
- Али када једном направимо општи интерфејс, он у програмском коду не зависи од конкретних случајева
- Напротив, они зависе од њега
- Зато се принцип и назива принципом *инверзне* зависности
  - стварна имплементациона зависност иде у супротном смеру од одговарајуће концептуалне зависности

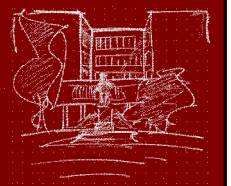
Принцип инверзне зависности (3)

- Ово је врло уопштен принцип са бројним облицима примене
  - Сама идеја полиморфизма почива на њему
  - Велики број образаца почива на примени овог принципа
    - Састав
    - Посматрач
    - Посетилац
    - ...

[P290]

Развој софтвера

Саша Малков



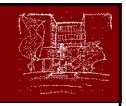
Тема 8.2

Принципи пројектовања софтвера
-
Принципи додељивања одговорности

Принципи додељивања одговорности

- Односе се првенствено на распоређивање одговорности
  - обично се приписују Крегу Ларману
  - пре свега се односе на ОО методологије
  - под одговорношћу се мисли на оно што класа *зна* и *уме*
  - означавају се скраћеницом *GRASP* (*General Responsibility Assignment Software Patterns*)
  - Наравно, и већ описани принципи се тичу одговорности, као што се и ови односе на зависности

Принципи додељивања одговорности (2)



- **Информациони експерт** – (енгл. *Information Expert*)
- **Стваралац** – (енгл. *Creator*)
- **Висока кохезија** – (енгл. *High Cohesion*)
- **Ниска спрегнутост** – (енгл. *Low Coupling*)
- **Контролер** – (енгл. *Controller*)
- **Полиморфизам** – (енгл. *Polymorphism*)
- **Измишљотина** – (енгл. *Fabrication*)
- **Индирекција** – (енгл. *Indirection*)
- **Изоловане променљивости** – (енгл. *Protected Variations*)

Информациони експерт

- **Одговорност за обављање посла додељивати класи која има информације неопходне за обављање тог посла**
- Основна идеја
  - лоцирање знања и одговарајућег понашања на истом месту
  - енкапсулирање знања
  - смањивање количине размене информација
  - задржавање одговарајуће осе променљивости унутар класе

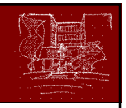
Информациони експерт (2)

- Последице одступања
  - Добијају се бар два елемента, један који садржи информације и други који има понашање, који морају да буду чврсто повезани
- Примери одступања
  - Неке архитектуре почивају на управо супротном концепту, када се исте информације користе у различитим пословима
    - архитектура Модел-поглед-контролер раздваја модел од контролера и погледа
    - зато што модел морају да користе и контролер и поглед

Стваралац

- **Одговорност за прављење објекта класе А доделити класи Б ако важи бар један од услова:**
  - **инстанца класе Б представља композицију инстанци класе А;**
  - **инстанца класе Б реферише инстанце класе А;**
  - **инстанца класе Б блиско користи инстанце класе А;**
  - **инстанца класа Б има информације за иницијализацију инстанци класе А и преноси их при прављењу**
- Основна идеја је смањивање броја зависности
  - неко мора да прави објекте класе А и то ће бити оса променљивости
  - ако је класа Б већ повезана са класом А, тако да већ постоји једна оса променљивости од А према Б...
  - ...нову зависност је најбоље увести тако да стоји уз већ постојећу

Стваралац (2)

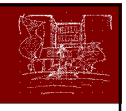


- Пример је образац Апстрактна фабрика
  - ако се праве и користе објекти различитих фамилија класа...
  - одговорност за прављење објеката се препушта апстрактној фабрици...
  - она бира конкретну класу коју прави, на основу контекста, аргумената,...
- ...конкретно...
  - ако програм подржава више врста корисничких интерфејса
  - онда би класа *ФабрикаИнтерфејса* могла да има методе *направиПрозор*, *направиДугме* и друге
    - који према контексту праве објекте који одговарају потребној врсти интерфејса
  - у програму се ел.интерфејса не праве непосредно већ помоћу фабрике
- Ово је пример четвртог случаја превиђеног принципом Стваралац

Висока кохезија

- **Додељивати одговорности тако да кохезија остане висока**
- Ово је један од најопштијих принципа пројектовања
- О кохезији је већ било довољно речи

Ниска спрегнутост

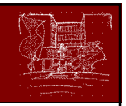


- **Додељивати одговорности тако да спрегнутост остане ниска**
- Ово је један од најопштијих принципа пројектовања
- О спрегнутости је већ било довољно речи

Контролер

- **Одговорности за примање или обрађивање порука о системским догађајима додељивати класи која:**
  - **представља цео систем, уређај или подсистем**
    - **тзв. фасадни контролер**
  - **или представља сценарио случаја употребе у коме се појављују системски догађаји**
    - **и при томе обично носи назив попут *<НазивСлучаја>Руковалац*, *<НазивСлучаја>Координатор*, *<НазивСлучаја>Сесија* и слично**
    - **тј. представља контролер сесије или случаја употребе**

Контролер (2)



- Основна идеја:
  - груписати реаговање на догађаје на једном месту
  - са тог места управљати активностима на основу догађаја
- Примери
  - Појам контролера је присутан у различитим моделима решавања проблема корисничког интерфејса
    - архитектура „Модел-поглед-контролер“
    - архитектура „Презентација-апстракција-контролер“
    - и већина других
  - На пример, сигнали и слотови у окружењу *Qt*

Полиморфизам

- **Када се нека врста понашања мења према типовима, онда је потребно одговорности за одговарајуће понашање распоредити по тим типовима, применом полиморфизма**
- Основна идеја
  - елиминисање експлицитних провера типова из програмског кода
  - уместо тога се користе полиморфни методи
- Мотивација
  - експлицитне провере типова имају тенденцију да се понављају…
  - …а свако понављање је лоше јер отежава одржавање

Полиморфизам (2)

- Хијерархијски полиморфизам
  - интерфејсом базне класе се предвиди метод који ради за сваку класу другачије, уместо да експлицитно проверава тип
- Параметарски полиморфизам
  - параметарским типом се очекује тип који има неки интерфејс (тј. уме да нешто уради)
  - у конкретној примени се то разрешава према конкретним типовима, уместо да се експлицитно проверава
- Обрасци и рефакторисања
  - већи број образаца и рефакторисања се ослања на овај принцип

Измишљотина

- **Тесно повезан и заокружен скуп одговорности доделити вештачки уведеној класи, која не представља концепт из домена проблема – која је измишљена да би омогућила високу кохезију, ниску спрегнутост и вишеструку употребу**
- Основна идеја
  - доследно моделирање структуре домена врло често носи са собом велики број сложених зависности...
  - ...тако да не постоји природна декомпозиција која их разрешава...
  - ...тада морамо да измишљамо *потпуно нове функционално еквивалентне концепте*

Измишљотина (2)

- Примери
  - убацивање једноставних концепата:
    - образац Фасада
    - образац Адаптер
  - увођење сложенијих концепата:
    - образац Посетилац
    - образац Декоратер
  - и многи други

Индирекција

- **Доделити одговорности објекту који је посредник између других компоненти или сервиса, тако да они не морају да буду непосредно спрегнути**
- Основна идеја
  - замењивање непосредне зависности посредном
  - ...уз додавање стабилног концепта као посредника
  - ...или уз промену смера зависноти
  - ...или уз груписање оса променљивости
  - ...и друго

Индирекција (2)

- Обично се тежи да се зависности посредника уреде тако да
  - друге класе зависе од интерфејса посредника
  - да имплементација посредника зависи од интерфејса других компоненти
- Примери
  - Апстрактна фабрика сакрива начин прављења објеката
  - Адаптер, Мост, Фасада сакривају од корисника специфичности имплементације па чак и интерфејса

Изоловане променљивости

- **Доделити одговорности тако да се обликује стабилан интерфејс око препознатих тачака предвидиве променљивости или нестабилности**
- Основна идеја
  - односи се на декомпоновање према променљивости
  - енкапсулирање тачака променљивости
  - постављање стабилног интерфејса између тачака променљивости и елемената који зависе од њих

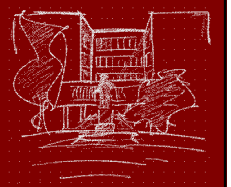
Изоловане променљивости

- Спада у основне и најопштије принципе пројектовања
  - као и принципи Висока кохезија и Ниска спрегнутост
- Повезан је са многим другим принципима
  - Индирекција, Инверзна зависност, Заменљивост,...
- Може се повезати са практично сваким примером доброг дизајна
  - Типичан пример је образац Адаптер

[P290]

Развој софтвера

Саша Малков



Тема 8.3

Принципи пројектовања софтвера – Принципи обликовања целина

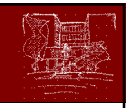
Принципи обликовања целина

- Односе се првенствено на груписање елемената у целине
- Тесно су повезани са општим принципима пројектовања
  - Висока кохезија, Ниска спрегнутост, Изоловане променљивости
  - може се рећи да представљају њихово даље прецизирање
- Делимо их у две групе:
  - Принципи груписања (или принципи кохезије)
  - Принципи раздвајања (или принципи спрегнутости)

Принципи обликовања целина (2)

- Принципи груписања
  - **Принцип еквивалентности издања и поновљиве употребе**
  - **Принцип заједничке затворености**
  - **Принцип заједничке употребе**
- Принципи раздвајања
  - **Принцип стабилне зависности**
  - **Принцип стабилне апстракције**
  - **Принцип ацикличних зависности**

П.еквивалентности издања и поновљиве употребе

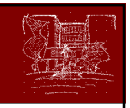


- (енгл. *REP – The Reuse-Release Equivalence Principle*)
- **Гранула поновљиве употребе је гранула објављивања**
- Основна идеја
  - Тесна веза између поновљиве употребе и објављивања, у оба смера:
    - (1) ако би једна целина требало да се користи на више места, онда би њена спецификација требало да се објави као једна целина
    - (2) ако нешто објављујемо, онда морамо да рачунамо да ће то неко и да користи
- Објављивање = јавно (шта год то значило) дефинисање интерфејса и обећавање (шта год то значило) да ће бити имплементиран и одржаван

П.еквивалентности издања и поновљиве употребе (2)

- Овај принцип представља основ за функциналну декомпозицију
- Свака објављена целина мора да има
  - стабилан интерфејс
    - тј. јасну и добро дефинисану функцију
    - имплицира функционалну повезаност компоненти
  - добру енкапсулацију
    - тј. да сакрије од корисника сву интерну сложеност
    - имплицира чврсту повезаност саставних делова једне компоненте

Принцип заједничке употребе



- (енгл. *CRP – The Common-Reuse Principle*)
- **Класе у пакету се користе заједно. Ако се користи једна од класа у пакету, онда се користе све.**
- Основна идеја
  - Односи се првенствено на логичку и структурну декомпозицију (пакети)
  - У исти пакет је потребно ставити класе које се заједно користе
    - Ако већ немамо функционалну кохезију, остваријемо секвенцијалну, комуникациону или процедуралну

Принцип заједничке употребе (2)

- Пример
  - Класе апстрактног дрвета синтаксе
    - имају различите функције – свака моделира другу врсту чвора
    - али се обично користе заједно
    - ставићемо их у исти пакет
    - и вероватно придружити и одговарајућу фабрику
- Овај принцип нам помаже и да одлучимо да неке класе *не* стављамо у исти пакет
  - ако нису функционално зависне
  - па се још и не користе заједно

Принцип заједничке затворености

- (енгл. *CCP – The Common-Closure Principle*)
- **Делови целине би требало да буду заједно и подједнако затворени у односу на исте врсте промена.**
  Ако целина мора да се мења, онда се претпоставља да морају да се мењају и (сви) делови те целине али не и друге целине.
- Основна идеја
  - Ако међу структурним елементима постоје јаке међузависности, онда би требало да припадају истој целини
  - Тако се више оса зависности групише на једном месту
  - Односи се на различите врсте кохезије: функционалну, секвенцијалну, комуникациону, процедуралну

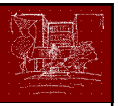
Принцип заједничке затворености (2)

- Пример
  - Ако имамо скуп функција за форматирање делова извештаја...
    - оне могу да буду релативно међусобно независне
    - али ако се промени формат извештаја, мењаће се и већина тих функција
  - Онда је добро да их групишемо у једну класу...
    - иако то није функционална кохезија, биће комуникациона или процедурална
    - таква класа обично има улогу у индирекцији или измишљеном концепту

Принцип стабилне зависности

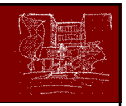
- (енгл. *SDP – The Stable-Dependencies Principle*)
- **Смер зависности би требало да се поклапа са смером пораста стабилности**
- Нестабилност = значајна вероватноћа да ће се елемент мењати
- Основна идеја
  - Непоклапање ова два смера прети да доведе до осе променљивости која иде у смеру пораста стабилности, што онда прети да произведе *лавину промена*
  - Ако је елемент стабилан, то значи да многи зависе од њега, па онда не би смело да се деси да и он има осетљиве зависности

Принцип стабилне зависности (2)



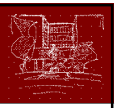
- Потенцијално нарушавање настаје када год имамо елемент који на неки начин повезује “два света”
  - што није реткост и некада не може да се избегне
  - али може да се води рачуна о врстама и смеровима зависности
- Овај принцип не указује на то како проблем може да се реши, већ пре представља неку врсту детектора проблема
- Уочен проблем се решава инверзијом зависности, измишљањем и сл.

Принцип стабилне зависности (3)



- На пример:
  - Образац Фасада повезује спољашњост и унутрашњост компоненте
    - Ако је интерфејс апстрактан, онда су све зависности *према* фасади, што је у реду
    - Ако није, онда постоје зависности од фасаде према унутра, што не ваља
  - Базна класа хијерархије класа
    - Ако има апстрактан интерфејс, опет су све зависности према њој - у реду је
    - Ако нема довољно апстрактан интерфејс, онда постоје зависности од ње према класама хијерархије, што не ваља
  - и слично

Принцип стабилне апстракције



- (енгл. *SAP – The Stable-Abstractions Principle*)
- **Целина би требало да буде апстрактна онолико колико је стабилна**
- Основна идеја
  - Повезивање апстрактности и стабилности
  - Сугерише (са принципом стабилне зависности) да би смер зависности требало да се поклапа са смером пораста апстрактности
  - Апстрактни елементи софтвера би требало да зависе од што мање других елемената
    - а ако већ морају, онда би требало да то буду *још апстрактнији* елементи
- Повезан је са принципима Стабилне зависности, Изоловане променљивости, Инверзне зависности и другима

Принцип стабилне апстракције (2)

- Примери
  - Има их много
  - Најочигледнији су исти као за случај стабилних зависности
    - Фасада
    - Апстрактна базна класа хијерархије

Принцип ацикличних зависности

- (енгл. *ADP – The Acyclic-Dependencies Principle*)
- **Не допуштати цикличне зависности пакета.**
- Основна идеја
  - Цикличне зависности отежавају локализовање промена
    - ако су А, Б и Ц циклично зависни елементи, па морамо да променимо А, онда ћемо можда морати да променимо Б, па ћемо можда морати да променимо и Ц, па ћемо можда поново морати да променимо А...
  - Разбијају се применом инверзне зависности, измишљањем и друго.

Принцип ацикличних зависности (2)

- Пример
  - Архитектура Модел-поглед-контролер почива на цикличним зависностима
    - контролер јавља моделу да се ажурира, модел јавља погледу да се ажурира, а поглед јавља контролеру да су потребне нове промене
  - Цикличност у МПК се превазилази:
    - увођењем обрасца Посматрач у однос погледа и модела
    - преусмеравањем управљања догађајима са погледа на контролер
      - што је вид уопштења обрасца Посматрач

Други принципи

- У литератури може да се нађе још много принципа за пројектовање
- Ако довољно повећамо скуп принципа, биће доста преклапања
  - и међу већ представљеним принципима има преклапања
- Постоје и мање општи принципи, који се односе на конкретне програмске језике
  - прилагођавање принципа конкретним језицима и алатима
  - додатни принципи који се односе на стил програмирања
  - на пример, постоје тзв. *идиоми C++-а*

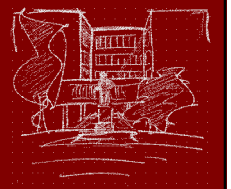
Примери принципа

- Одлични примери примене принципа пројектовања су
  - Обрасци за пројектовање
  - Рефакторисања
- При изучавању образаца и рефакторисања добро је да се размишља о примењним принципима дизајна

Литература за тему

- *Robert C. Martin*, ***Agile Software Development – Principles, Patterns and Practices***, *Prentice Hall, 2003.*
- *Craig Larman*, ***Applying UML and Patterns***, *3.ed, Pearson, 2004.*
- *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides*, ***Design Patterns: Elements of Reusable Object-Oriented Software***, *Addison-Wesley, 1995. Izdanje na srpskom jeziku:* ***Gotova rešenja: Elementi objektno orijentisanog softvera***, *CET, 2002.*
- *Martin Fowler*, ***Refactoring: Improving the Design of Existing Code***, *Addison-Wesley, 1999. Izdanje na srpskom jeziku:* ***Refaktorisanje: Poboljšanje dizajna postojećeg koda***, *CET.*

Хвала на пажњи!



МАТФ
Универзитет у Београду
Математички факултет