Platformă de e-learning și curriculă e-content pentru învățământul superior tehnic

Instrumente pentru Dezvoltarea Programelor

14. Instrumente UI. MVC. Swing. Tehnici de dezvoltarea programelor folosind Swing.

Obiective

Scopul acestui laborator este familiarizarea cu Swing, infrastructura Java pentru interfete grafice. Intrebuintarea acesteia urmareste evidentierea unor principii de proiectare.

Subiecte atinse:

- Paradigma *Model-View-Controller* (MVC)
- Swing si legatura sa cu MVC
- Design patterns: *Observer*, *Command*, *Decorator*

Model-View-Controller (MVC)

Paradigma MVC se refera la **separarea** logicii interne a unei aplicatii de partea sa de prezentare, oferind posibilitatea modificarii **independente** a acestor componente si inlesnind **reutilizarea**.

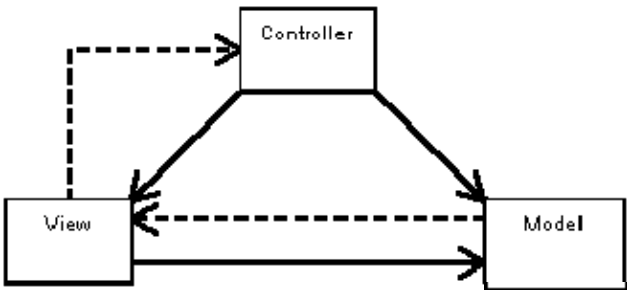
Cele 3 **componente** sunt:

- ***model***: datele cu care lucreaza aplicatia
- ***view***: prezentarea datelor catre utilizator, de obicei in forma unei interfete grafice. Pot exista mai **multe** view-uri asociate cu un acelasi model.
- ***controller***: capteaza si proceseaza actiunile utilizatorilor.

Conceptul poate fi perceput la **nivel**:

- **arhitectural**: se refera la structura intregii aplicatii, in care diferitele componente isi asuma rolurile de mai sus (exemplu: Grails framework)
- **de design**: utilizat in contextul unor scenarii mai restranse (exemplu: componentele vizuale Swing).

Figura de mai jos ilustreaza fluxul de control:



Interactiunile pot fi:

- **directe** (linii continue): apeluri directe de metode ale obiectelor. Exemplu: *controller*-ul apeleaza metode ale modelului pentru a-l actualiza in urma actiunilor utilizatorului
- **indirecte** (linii intrerupte): generare si captare evenimente. Exemplu: modelul genereaza un eveniment in momentul unei modificari, ce poate fi captat de *view*, fara ca modelul sa fie constient de implementarea particulara a acestuia.

Java Foundation Classes (JFC)

JFC reprezinta API-ul pus la dispozitie de Java pentru construirea interfetelor grafice. Componentele sale sunt:

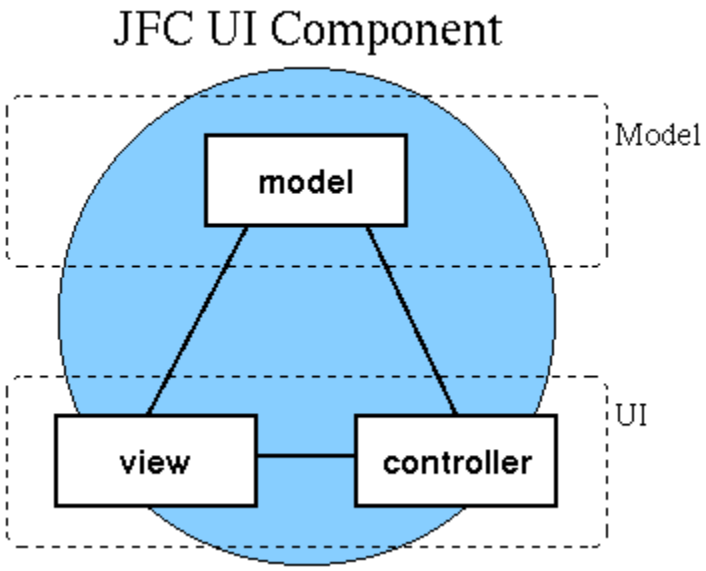
- AWT
- Swing
- Java2D

AWT reprezinta tehnologia initiala. Principalul dezavantaj al acesteia este **dependenta** de platforma, expunand particularitatile native (specifice platformei) ale componentelor grafice (*widgets* - window gadgets). In plus, aspectul vizual al componentelor (*look-and-feel*) este dictatat de platforma gazda. Din aceste motive, componentele AWT sunt considerate *heavyweight*.

Swing reprezinta tehnologia actuala. Arhitectural, se situeaza **deasupra** AWT, adaugand drept caracteristica principala **independenta** de platforma. In acelasi timp, aspectul vizual al componentelor este **decuplat** de implementare, existand posibilitatea inlocuirii acestuia, chiar la runtime (*pluggable look-and-feel*). Redarea grafica se face programatic (folosind Java2D), spre deosebire de AWT, care deleaga aceasta sarcina sistemului de operare. Swing este construit pe principiul **MVC**, prezentat mai sus.

Swing

Swing are la baza o varianta modificata a arhitecturii **MVC**, in care *view*-ul si *controller*-ul se imbina:



Toate componentele vizuale Swing sunt construite pe baza acestei arhitecturi. **Modelele** se impart in 2 categorii:

- ***GUI state***: retin starea componentei. Exemplu: modelul unui buton retine informatii precum apasat/neapasat, activat/dezactivat etc.;

- ***application-specific data***: se refera la datele cu care opereaza o aplicatie. Exepmplu: inregistrarile proprii unui tabel.

Printre **avantajele** izolarii modelului, enumeram:

- **reutilizabilitate**: posibilitatea de a oferi mai multe vederi asupra acelorasi date. Exemplu: bar chart si pie chart pe baza acelorasi informatii statistice
- ***wrapping***: se pot defini modalitati diferite de obtinere a datelor, in spatele unei interfete uniforme. Exemplu: un model poate tine datele in memorie sub forma unui vector, in timp ce altul poate realiza *query*-uri catre o baza de date, in ambele cazuri in mod transparent.
- posibilitatea **inlantuirii** modelelor: un model poate ***decora*** un alt model, adaugand un plus de functionalitate (sortare, filtrare), fara a modifica modelul (datele) originale.

Swing imbina flexibilitatea, prin realizarea acestei separatii, cu viteza de dezvoltare, oferind implementari ***default*** pentru toate modelele. De exemplu, pentru o lista (JList), ordinea crescatoare a functionalitatii modelului specific ar fi:

1. ListModel (interfata): varful ierarhiei tuturor modelelor de lista
2. AbstractListModel (clasa abstracta): defineste modalitatea de gestiune a ascultatorilor de evenimente, dar nu ofera nicio implementare a accesului la date
3. DefaultListModel: implementarea default, retine datele sub forma unui Vector

Un program Swing este orientat pe evenimente (***event-driven***), asteptandu-se, intr-o bucla, aparitia unor evenimente pe care utilizatorul le doreste tratate. Secventa de mai jos defineste un *handler* de tratare evenimentului de apasare a unui buton (poate fi asimilat unui controller):

```
JButton b = new JButton("Click me!");
b.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      Object source = e.getSource(); // obiectul care a generat evenimentul
      ...
   }
}
```

Se constata ca logica de mai sus se inscrie in design pattern-ul ***Observer***, in care un **subiect** este urmarit de unul sau mai multi **observatori** care asteapta notificari de la acesta.

Atentie! Programul este ***single-threaded***, in sensul ca *handler*-ele de eveniment ruleaza intotdeauna **succesiv**. De aceea este imperativ ca prelucrarile de lunga durata sa ruleze pe fire de executie **separate**, altfel riscandu-se **blocari** temporare ale interfetei (care nu mai poate raspunde actiunilor utilizatorului inainte ca *handler*-ul curent sa-si incheie executia). Pentru astfel de task-uri, ce se doresc executate in fundal, urmand ca la final sa aibe loc unificarea cu interfata, Java pune la dispozitie clasa SwingWorker, discutata in laboratorul viitor.

In Swing, thread-ul pe care ruleaza *handler*-ele de evenimente se numeste *Event Dispatch Thread* (EDT). Metoda SwingUtilities.invokeLater(Runnable) forteaza executia unei secvente pe acest fir (vezi scheletul de laborator).

Design patterns specifice

Swing utilizeaza o serie de design patterns foarte raspandite:

- *Observer*
- *Command*
- *Decorator*

Observer

La baza, aceste design pattern defineste o relatie unu-la-mai-multi intre obiecte, astfel incat, atunci cand un obiect (***subject***) isi schimba starea, toate celelalte (***observers***) sunt instiintate de acest lucru. Este folosit cand se doreste pastrarea **consistentei** intre obiectele din sistem, urmarindu-se, in acelasi timp, **evitarea** unui cuplaj prea strans, care ar impiedica **varierea comportamentului** si **reutilizarea**.

Iata un exemplu, in care o serie de date statistice capata 3 reprezentari vizuale diferite. In momentul in care datele **se modifica**, interfetele trebuie **notificate**. Este interesat de mentionat ca **schimbarile** pot fi declansate chiar de unul din **observatori** (de exemplu, la apasarea unei taste).

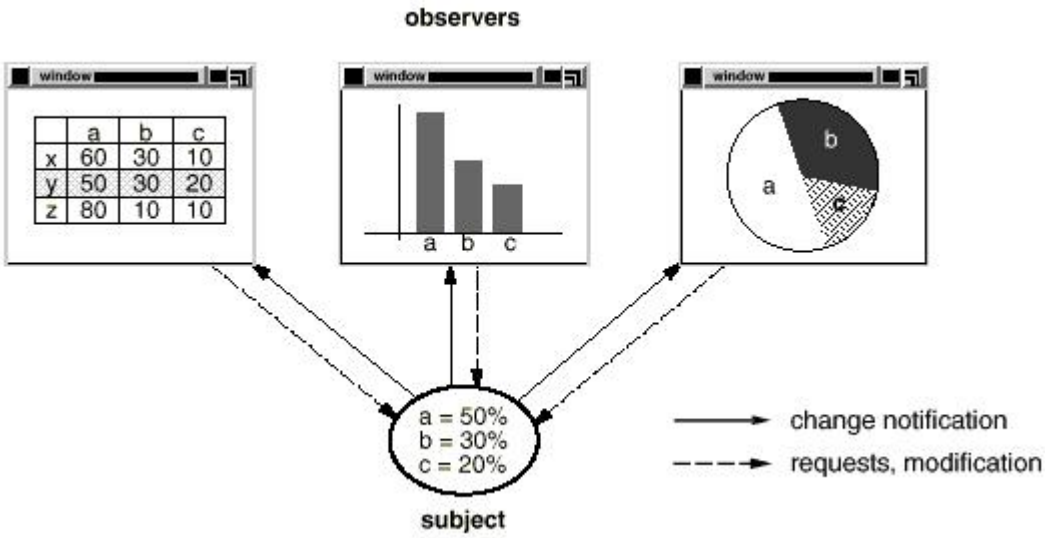
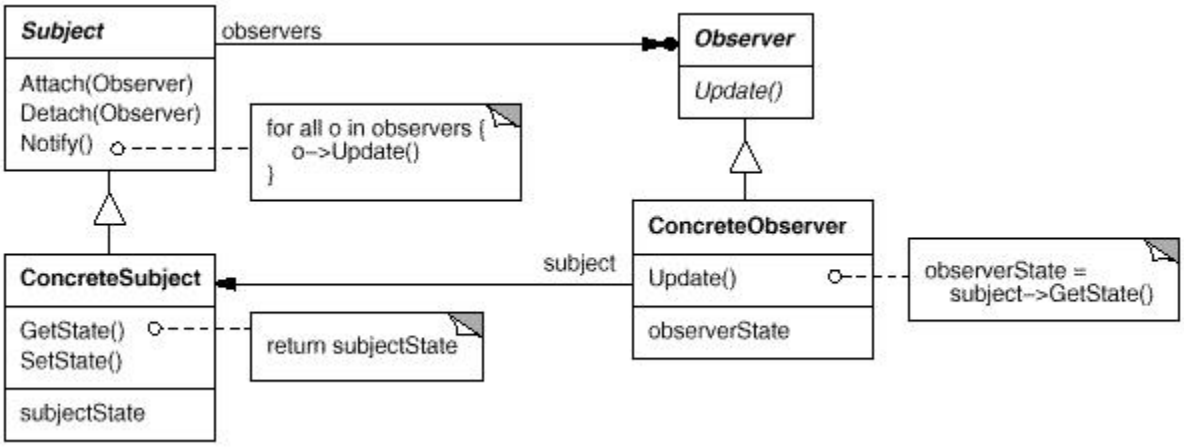


Diagrama de clase:



Entitati:

- Subject
  - reprezinta obiectul observat
  - tine lista tuturor observatorilor
  - ofera posibilitatea ca acestia sa se inregistreze, apeland attach
- Observer
  - reprezinta un obiect observator
  - ofera posibilitatea notificarii din exterior, expunand metoda update
- ConcreteSubject
  - implementarea propriu-zisa a obiectului observat
  - la schimbarea starii, notifica toti observatorii inregistrati
- ConcreteObserver
  - implementarea propriu-zisa a obiectului observator

In secventa de cod, de mai sus, subiectul este butonul, iar observatorul este instanta clasei care implementeaza ActionListener.

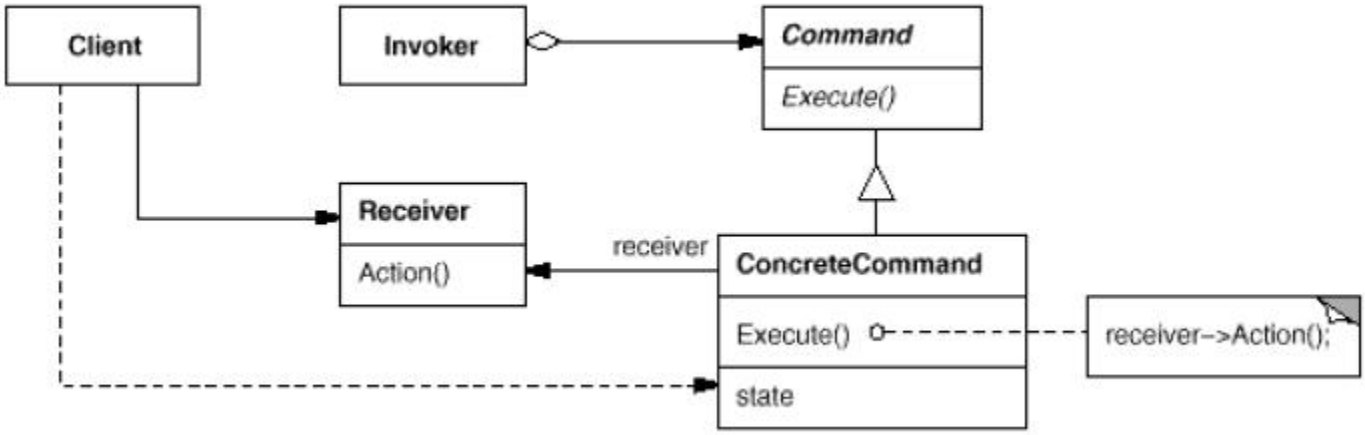
Command

Ideea principala este de a **incapsula** o comanda, intr-un obiect. Printre **beneficii**, enumeram:

- **decuplarea** intre entitatea care dispune executarea comenzii si entitatea care o executa. Efectul unei comenzi poate fi schimbat dinamic. Decuplarea este necesara pentru a permite, spre exemplu, implementarea clasei buton, in **absenta** informatiilor despre efectul apasarii lui. De asmenea, se permite asocierea aceluiasi efect cu 2 surse diferite (element de meniu si buton de pe toolbar, cu efect de *paste*).
- posibilitatea de a **intarzia** executarea unei comenzi, pana la momentul potrivit. Exemplu: caching, pentru evitarea generarii prea multor comenzi intermediare, si rularea mai multor comenzi deodata.

- **memorarea** efectului unei comenzi, pentru a permite **revenirea** la o stare anterioara (undo/redo)
- posibilitatea definirii de comenzi **compuse** (macro-uri), care grupeaza alte comenzi

Diagrama de clase:



Entitati:

- Command
  - obiectul comanda
- ConcreteCommand
  - implementarea particulara a comenzii
  - apeleaza metode ale obiectului receptor
- Invoker
  - declanseaza comanda
- Receiver
  - realizeaza, efectiv, operatiile aferente comenzii generate
- Client
  - defineste obiectul comanda si efectul ei

Observati ca **nu** exista o delimitare clara intre *Observer* si *Command*. Un observator poate fi privit ca un obiect comanda.

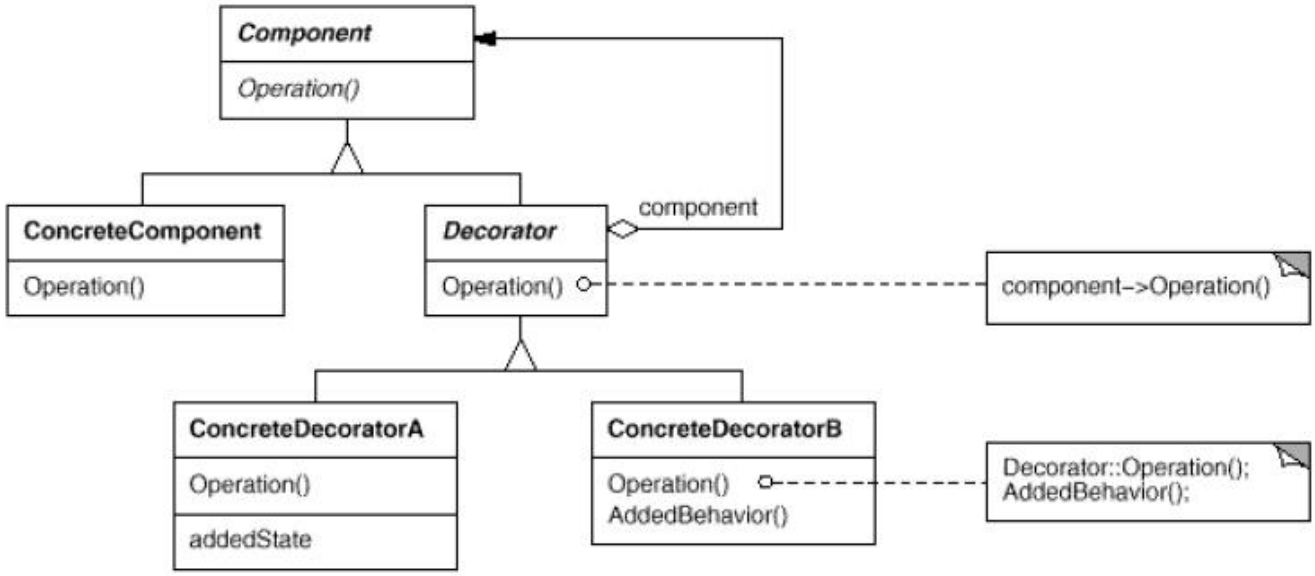
Decorator

Acest design pattern propune o modalitate de a **adauga** functionalitate obiectelor, **dinamic** (la cerere), **evitand** subclasarea. De asemenea, obiectul rezultat prin decorare poate fi utilizat in locul obiectului decorat. Acest lucru permite **inlantuirea**, pe oricate nivele, a decoratorilor, in mod **transparent** pentru entitatile care utilizeaza obiectul respectiv. De asemenea, **previne** explozia de clase, rezultata din definirea explicita, prin subclasare, a tuturor **combinatiilor** posibile de functionalitati.

Exemple de utilizare:

- decorarea **componentelor grafice**, prin adaugarea de elemente vizuale. Exemplu: adaugarea unui contur si a unei bare de derulare pentru o lista. Lista rezultata prin decorare va putea fi folosita oriunde s-ar fi folosit lista initiala.
- decorarea **modelelor** (despre care am vorbit in sectiunea ***Swing***).

Diagrama de clase:



Entitati:

- Component
  - reprezinta obiecte a caror functionalitate poate fi adaugata dinamic
- Decorator
  - reprezinta un decorator generic, ce ofera **aceeasi** interfata cu cea a obiectului decorat
- ConcreteComponent
  - implementare particulara a unui obiect decorabil
- ConcreteDecorator
  - implementare particulara a unui decorator

Observati cum decoratorul **imbina** mostenirea (*inheritance*) cu compunerea (*composition*). Obiectul decorator tine o **referinta** la obiectul decorat, caruia ii inainteaza cererile, eventual modificate, si, simultan, expune aceeasi **interfata**. Acest lucru este necesar pentru **transparenta** obiectului decorat in raport cu clientul sau.

Exercitii

Observatii:

- Utilizati scheletul de laborator (http://elf.cs.pub.ro/idp/_media/laboratoare/l4/idp_lab_4_skel.zip)
- Punctele marcate cu (T) au asociat un comentariu TODO in cod. Exemplu: pentru exercitiul 1 gasiti comentariul // TODO 1
- Cititi enuntul fiecarui exercitiu complet si cu atentie!

1. (**1p**) Studiati fisierul Main.java. Observati functia main. Rulati aplicatia si observati dispunerea componentelor. Repozitionati fereastra Swing in centrul ecranului.
2. (**1p**) (T) Populati **modelul** de lista cu 3 prescurtari de cursuri pe care le-ati facut. Ce observati cand rulati?
3. (**1p**) (T) Modificati ***handler***-ul butonului Add, astfel incat variabila text sa contina textul introdus in campul de editare. Adaugati functionalitatea de Add, **fara duplicate**, in lista.
4. (**1.5p**) (T) Prin analogie, definiti ***handler***-ul ce trateaza apasarea butonului Remove. Efectul va fi inlaturarea elementului selectat din lista stanga (list). Verificati ca, intr-adevar, exista un element selectat in lista respectiva si afisati un mesaj corespunzator in acest caz.
5. (**1.75p**) Definiti un model custom de lista, numit ReverseListModel, care sa **decoreze** modelul din program:
    - Rolul clasei este de oferi, in ordine inversa, elementele modelului decorat. Pentru aceasta, va retine o **referinta** la modelul pe care-l decoreaza.
    - Care este alegerea fireasca pentru mostenire: ListModel, AbstractListModel, DefaultListModel?
    - Clasa va fi definita intr-un fisier separat. Puteti apela, din meniul Eclipse, New → Class, dupa care alegeti clasa parinte. Avantajul este ca metodele ce trebuie implementate (in cazul mostenirii interfetelor/claselor abstracte) vor fi deja definite cu corp vid.
6. (**1.75p**) (T) Reinitializati lista mirror astfel incat sa plaseze un obiect ReverseListModel **deasupra** modelului existent: mirror → ReverseListModel → model initial. Rulati aplicatia. De ce lista din dreapta nu se mai actualizeaza la adaugarea sau stergerea de elemente? **Hints**:
    - Implicit, la initializarea unei liste cu un model, lista **se inregistreaza** sa asculte evenimentele generate de modelul respectiv. La interpunerea obiectului ReverseListModel, lista se inregistreaza la acest model nou, fara a mai avea o legatura directa cu vechiul model (care contine efectiv elementele).
    - Este necesar ca modelul custom, definit de voi, sa **intercepteze** evenimentele generate de modelul decorat si sa le inainteze, prin apeluri fire*, incat sa ajunga, in final, la lista. In acest fel, un eveniment va urma calea: model initial → ReverseListModel → mirror.

7. (**2p**) Adaugati functionalitatea de ***undo/redo***, pentru operatiile Add si Remove, utilizand pattern-ul *Command*.
    - Definiti o **interfata**, Undoable, ce constituie punctul de plecare pentru actiunile de mai sus. Interfata va contine **metodele** undo si redo.
    - Definiti cate o **clasa** ce implementeaza Undoable, pentru fiecare din cele 2 operatii pe lista (add/remove). In clase veti tine informatiile pe care le considerati necesare.
    - Definiti o clasa, UndoManager, care va gestiona **istoricul** comenzilor executate, respectiv anulate (*undone*). **Hints**:
        - Puteti folosi o **lista** (colectie) de obiecte Undoable
        - Pentru **deplasarea** eleganta, de-a lungul listei, puteti folosi un ListIterator, cu metodele add, next si previous
        - Aveti in vedere faptul ca executarea unei actiuni, dupa executarea a cel putin unui undo, conduce la **eliminarea** comenzilor pana la sfarsitul listei
    - Modificati **handler**-ele butoanelor Add si Remove, astfel incat sa memoreze comenzile executate, folosind clasele definite mai sus
    - Adaugati 2 **butoane**, Undo si Redo, cu comportamentul intuitiv.

Resurse utile

- MVC (http://en.wikipedia.org/wiki/Model-view-controller)
- Tutorial Swing (http://java.sun.com/docs/books/tutorial/uiswing/)
- Observer (http://en.wikipedia.org/wiki/Observer_pattern)
- Command (http://en.wikipedia.org/wiki/Command_pattern)
- Decorator (http://en.wikipedia.org/wiki/Decorator_pattern)