CHAMELEON Project

User Guide (Italian version)

Integrated Development Environment

http://www.di.univaq.it/chameleon/

Version	Changes	Author(s)
V2.0	This user guide describes the main functionalities offered by the CHAMELEON IDE to developers of adaptable Java applications. This is a short version that summarizes the long version, which is available only in Italian. We are working for producing the long version also in English. This version of the Integrated Development Environment supports the new version of the Hybrid (static and dynamic) Analysis approach. The preprocessor derives the Application Descriptor that, among other information, also lists the calls to external library methods for each alternative. Specifically, for each external method call in the list, the programmer is now supported by the CHAMELEON IDE to specify the worst-case call-context on the base of the application logic that has been coded. When a device requests an application, during the dynamic phase of the analysis, each library method is then executed by the CHAMELEON Client in the target execution environment, and its actual resource consumption is computed. This consumption is used to specify into the profile the resources consumed by each library method. The advantage of combining the static and the dynamic phase is to avoid to statically perform the expensive inter-procedural call-closure for library methods. In fact, the static inter-procedural call-closure in Java is extremely expensive for even a simple “Hello World” program due to the use of standard libraries (see also the developer guide of the CHAMELEON Mobile Client, Server, and UDDIe Registry, and the guide of the Analyzer).	- Marco Autili, - Paolo Di Benedetto - Paola Inverardi

Table of Contents

1. Installazione

I seguenti paragrafi descrivono i passi necessari alla predisposizione di un ambiente di sviluppo per progetti “Adaptable Java”.

1.1. Installazione del Plug-In ChameleonIDE

L’installazione del plugin ChameleonIDE si rivela piuttosto semplice dal momento che si limita alla copia del file it.univaq.di.ChameleonIDE.jar direttamente nella directory /eclipse/plugins dell’applicazione Eclipse utilizzata.

Ad ogni ripartenza Eclipse si accorge automaticamente della presenza dei nuovi plugins rendendoli così immediatamente disponibili.

Qualora invece non si disponesse direttamente del Plug-In in formato .jar ma comunque del source Project (it.univaq.di.ChameleonIDE), la generazione di un Plug-In distribuibile in formato .jar potrà avvenire in qualunque momento con i passi operativi descritti in Fig.1.

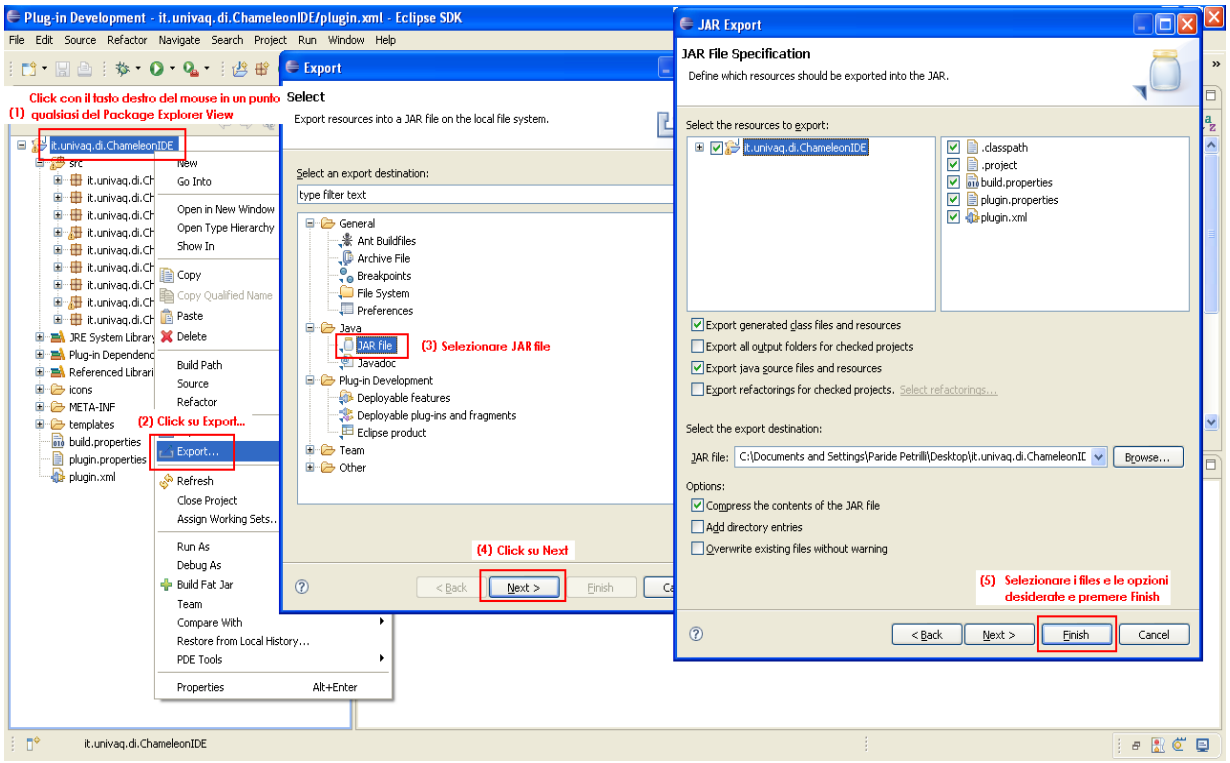

Fig.1: Generazione del plugin it.univaq.di.ChameleonIDE.

La procedura descritta genera il file it.univaq.di.ChameleonIDE.jar nella directory di destinazione prescelta.

1.2. Installazione del Chameleon Programming Model

Allo scopo di rendere più agevole la successiva fase di configurazione ed utilizzo dell’ambiente “Adaptable Java”, per quanto riguarda il Chameleon Programming Model si rendono necessari i seguenti passi volti alla generazione di un unico file .jar “integrato” che risulterà così completo sia delle librerie necessarie allo stesso

Programming Model (jdom.jar e antl.jar), sia delle classi java (Launch.java e Annotation.java) indispensabili invece per il suo utilizzo da parte di applicazioni “esterne” quali il ChameleonIDE.

La generazione di un file .jar integrato presuppone l’installazione del plugin “Fat Jar Eclipse Plug-In” facilmente reperabile dal sito http://fjep.sourceforge.net, che anche in questo caso va semplicemente copiato nella libreria /eclipse/plugins dell’applicazione Eclipse SDK utilizzata (in questo caso è indispensabile una versione di Eclipse completa del PDE - Plug-in Development Environment).

Il Programming Model andrà quindi importato rispettando la struttura di progetto rappresentata in Fig.2 in cui risultano evidenziati sia il nuovo package chameleon.programmingmodel.Launch riservato alle classi precedentemente menzionate che consentono l'utilizzo del prodotto da parte del ChameleonIDE (e di eventuali altre applicazioni "esterne") sia le librerie necessarie alla stessa applicazione (messe a disposizione nel folder jar).

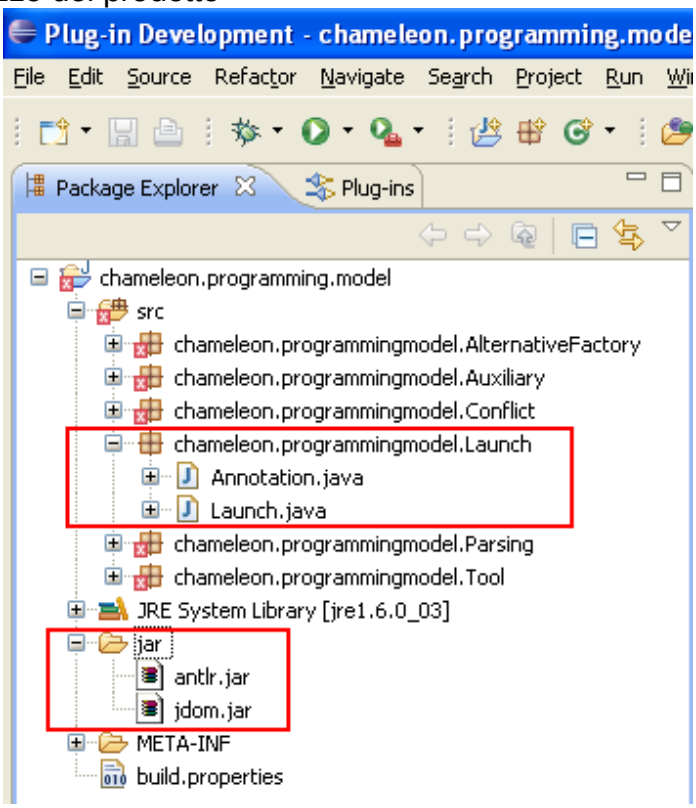

Fig.2: Struttura di progetto

Gli errori rilevati dal Workbench Eclipse in questa fase (le "x" quadrate in rosso di fianco ai packages e al Project Folder) sono temporanei dal momento che non sono ancora stati definiti i path per l'individuazione delle librerie .jar.

Tornando invece brevemente alle classi java, la classe Launch.java si occupa della gestione delle richieste di attivazione del Programming Model da parte di applicazioni "esterne", mentre la classe Annotation.java riveste particolare utilità nell'ambito dello sviluppo dei progetti "Adaptable Java".
Il codice di entrambe le classi è consultabile in appendice.

Si procede quindi con l'indicazione delle Referenced Libraries di progetto ossia delle librerie antl.jar e jdom.jar precedentemente copiate nella cartella jar (con operazione di import oppure con semplice copia/incolla dal file system) seguendo la numerazione dei passi di Fig.3 e Fig.4.

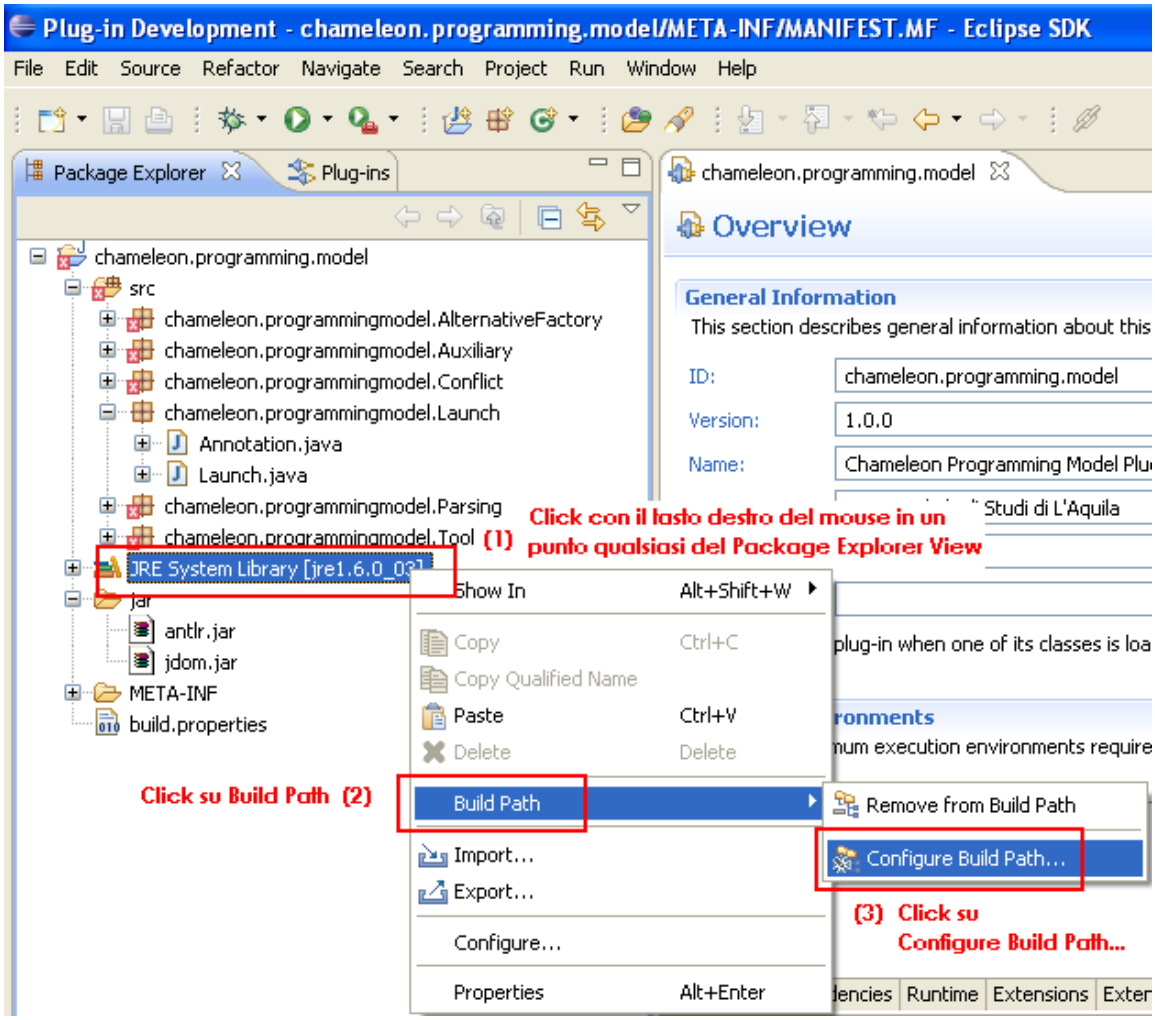

Fig.3: Passi di configurazione del Build Path

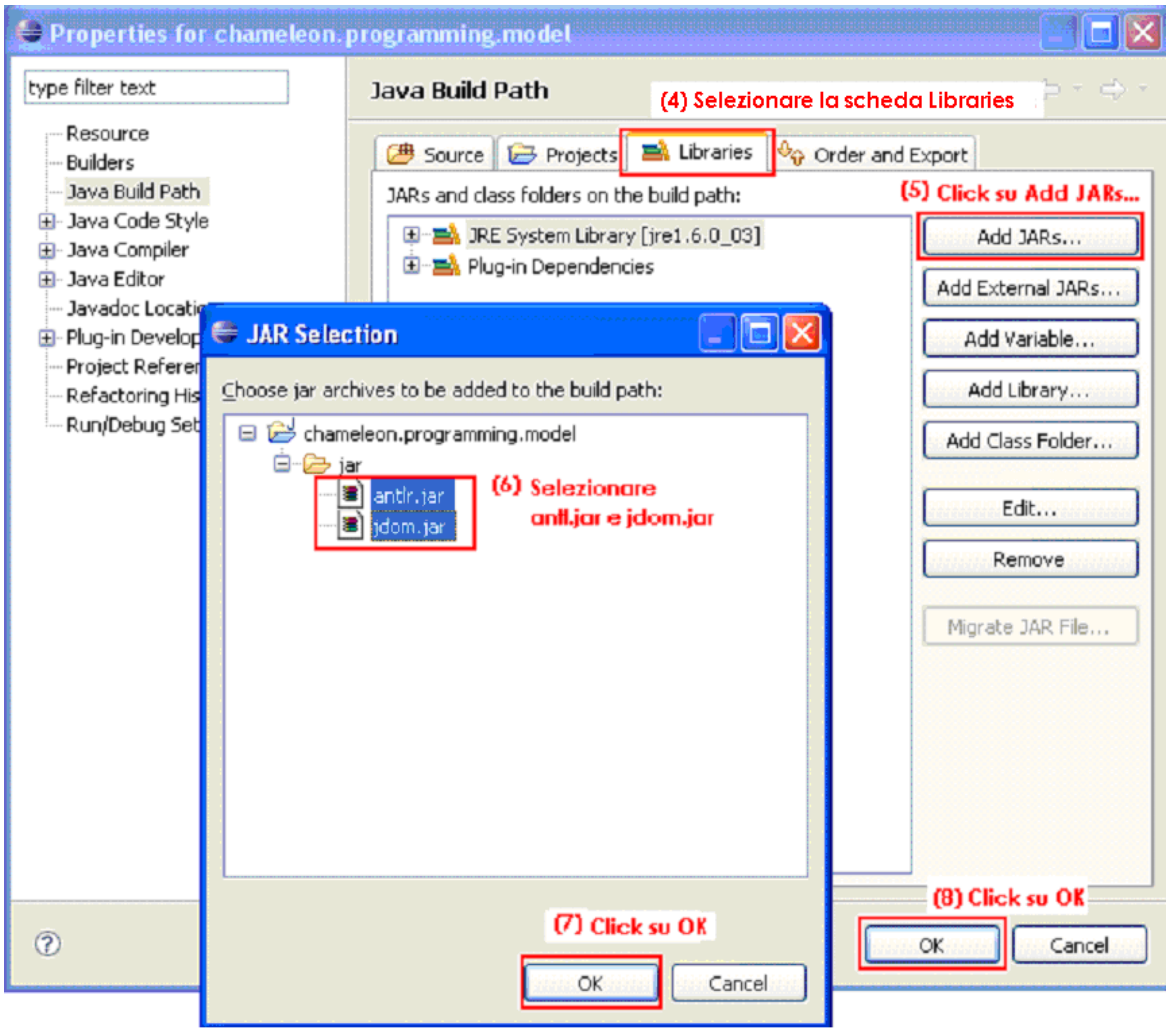

Fig.4: Selezione delle librerie .jar esterne.

Si noterà a questo punto in Fig.5 che le librerie antlr.jar e jdom.jar appaiono automaticamente "trasferite" dal folder jar al folder Referenced Libraries.

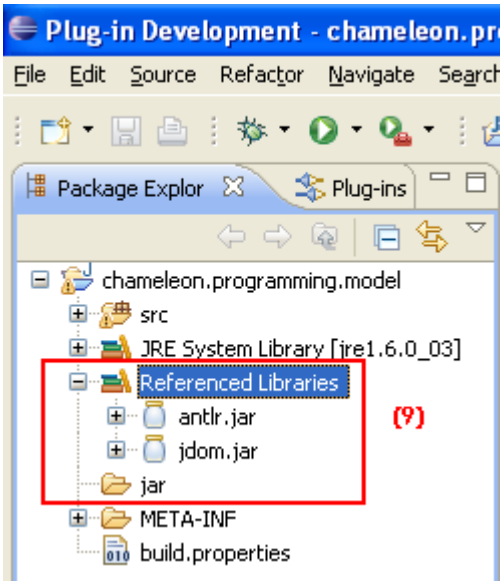

Fig.5: Il progetto dopo la configurazione del Build Path.

A questo punto il processo si conclude con la generazione del file .jar “integrato” mediante l’utilizzo del Fat Jar come descritto in Fig.6 e Fig.7.
Il file risultante (chameleon.programming.model_fat.jar) che compare sotto la directory principale del progetto (Fig.8) andrà quindi copiato nella stessa directory /eclipse/plugins del Plug-In it.univaq.di.ChameleonIDE.

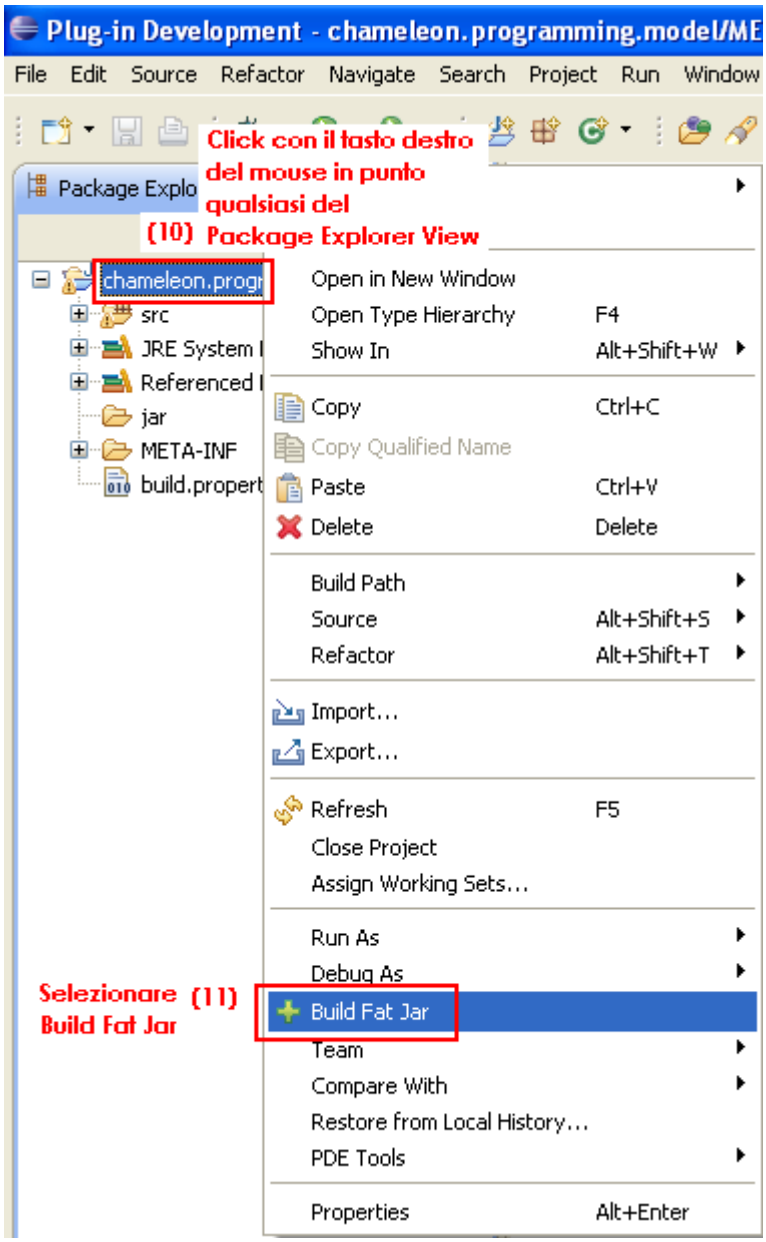

Fig.6: Generazione del _fat.jar (avvio del processo).

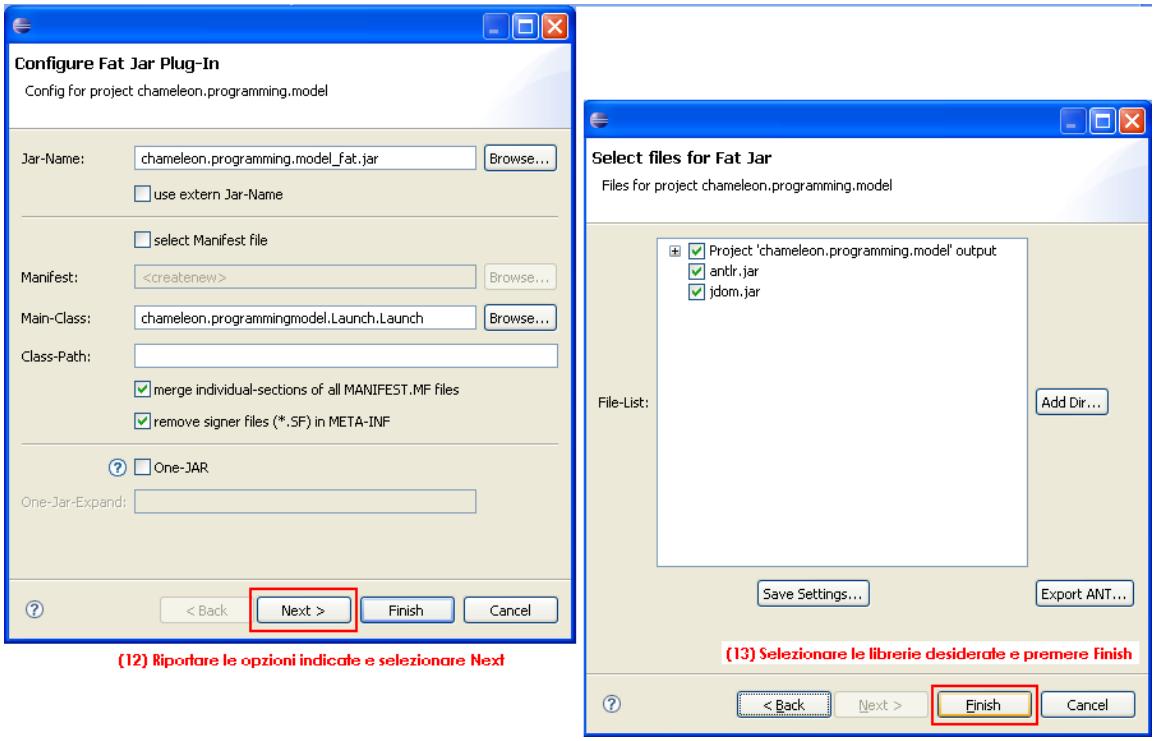

Fig.7: Generazione del _fat.jar (selezione delle librerie).

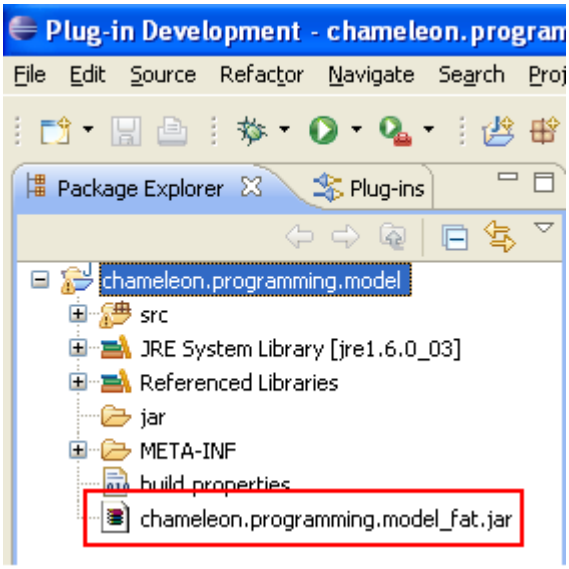

Fig.8: Il file _fat.jar generato.

2. Configurazione del Plug-In ChameleonIDE

Una volta avviato Eclipse la corretta esecuzione dei precedenti passi di installazione ha immediatamente riscontro dal momento che si nota subito la nuova icona nella toolbar, il cui significato, come quello di  tutte le “Adaptable Java Icons”, è riportato in appendice al documento.
Tuttavia si rende necessaria una semplicissima fase di configurazione iniziale affinchè possa successivamente avvenire senza problemi l’interazione tra il ChameleonIDE ed il Chameleon Programming Model.

Una volta generato il nuovo progetto Java su cui si andrà a lavorare, occorrerà indicare nelle Referenced Libraries il file chameleon.programming.model_fat.jar contenente tutte le librerie e le classi necessarie alla fase di “Building”.
Naturalmente, proprio perché tale operazione investe la sola fase di Building, essa potrà avvenire anche in una fase successiva senza precludere le altre funzionalità e gli strumenti messi a disposizione dal Plug-In ChameleonIDE.
Prima di procedere con il Building sarà pertanto sufficiente seguire i passi di Fig.9 e Fig.10.

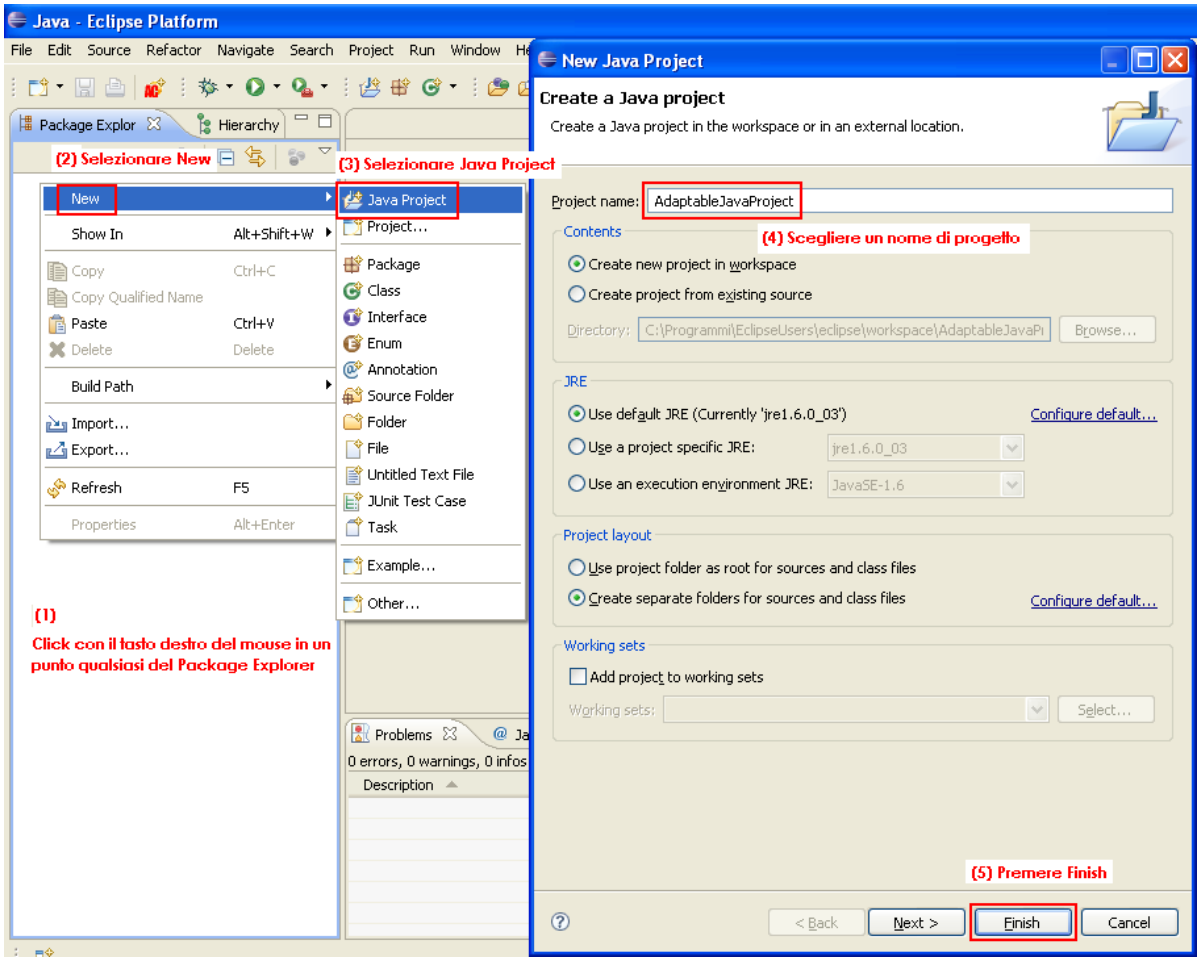

Fig.9: Creazione di un nuovo Java Project.

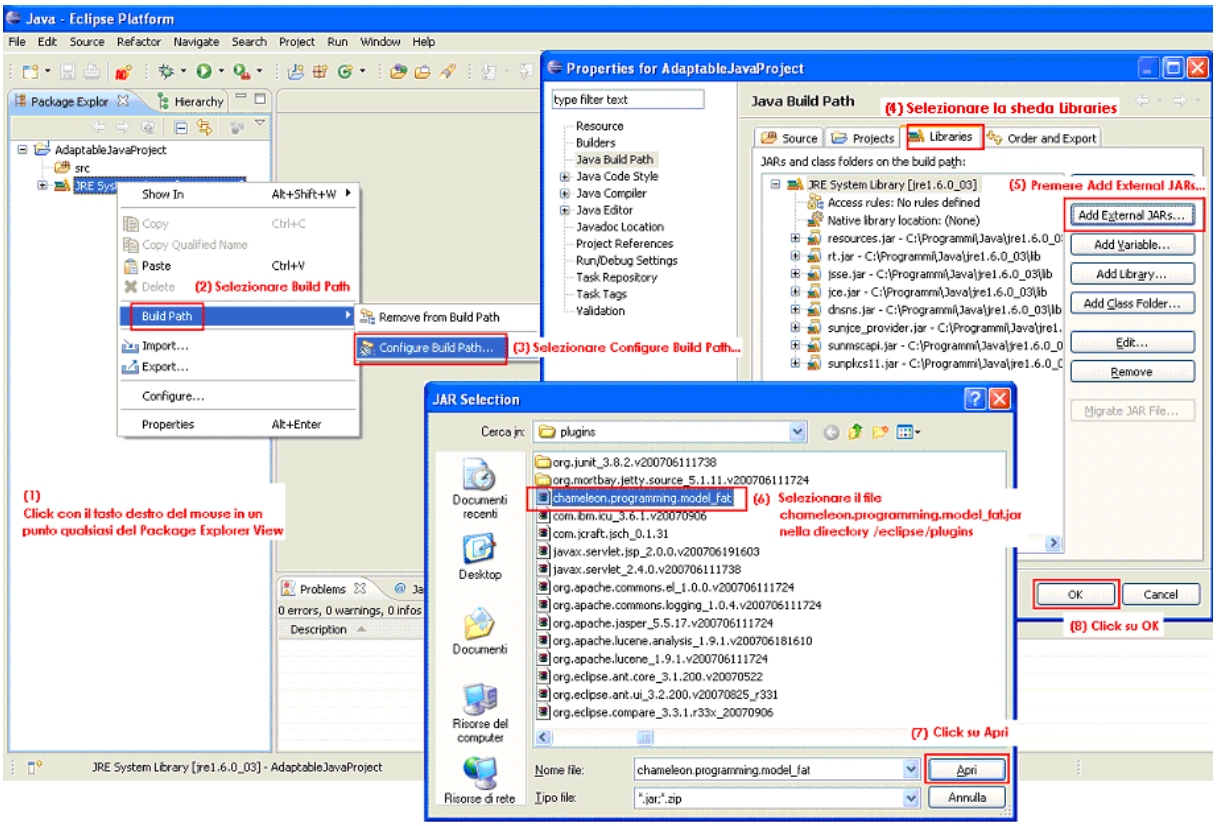

Fig.10: Configurazione del chameleon.programming.model_fat.jar.

3. Adaptable Java Project

A questo punto siamo in grado di realizzare un nuovo progetto "Adaptable Java". Ricordando che di fatto l'ambiente "Adaptable Java" ChameleonIDE non è altro che una "estensione" del Java Development Environment già fornito dalla piattaforma Eclipse, generalmente si presuppone che il primo passo sia volto alla definizione di una nuova classe Java o, meglio, di una classe Java "Adaptable".
La generazione di una nuova classe "Adaptable" può avvenire in diverse modalità:

1. Direttamente dalla toolbar con l'azione associata all'icona vista in precedenza;
2. Dalla barra di menù selezionando in sequenza File, New, Other…, quindi "Adaptable Class" sotto la nuova categoria "Adaptable Java Category" (vedi Fig.11);
3. Eseguendo un click con il tasto destro del mouse in qualunque punto della Package Explorer View in modo da richiamare il pop-up menù di Fig.12 e procedere con gli stessi passi descritti al punto precedente;

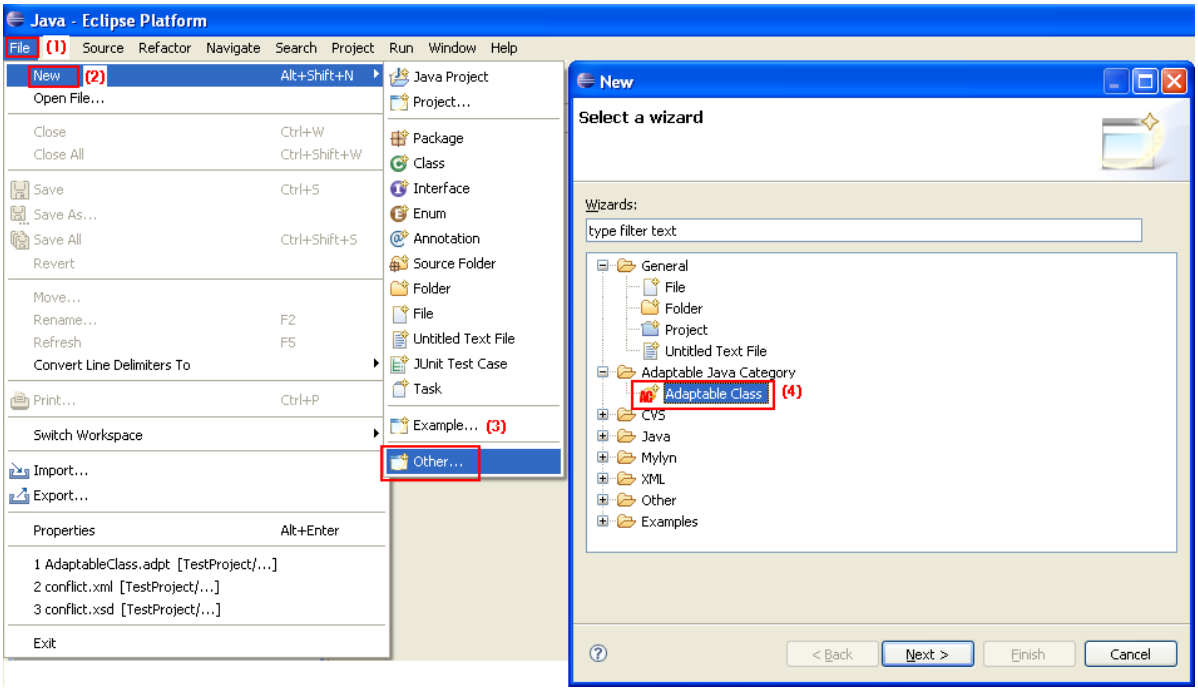

Fig.11: Generazione di una classe Adaptable dal menù generale.

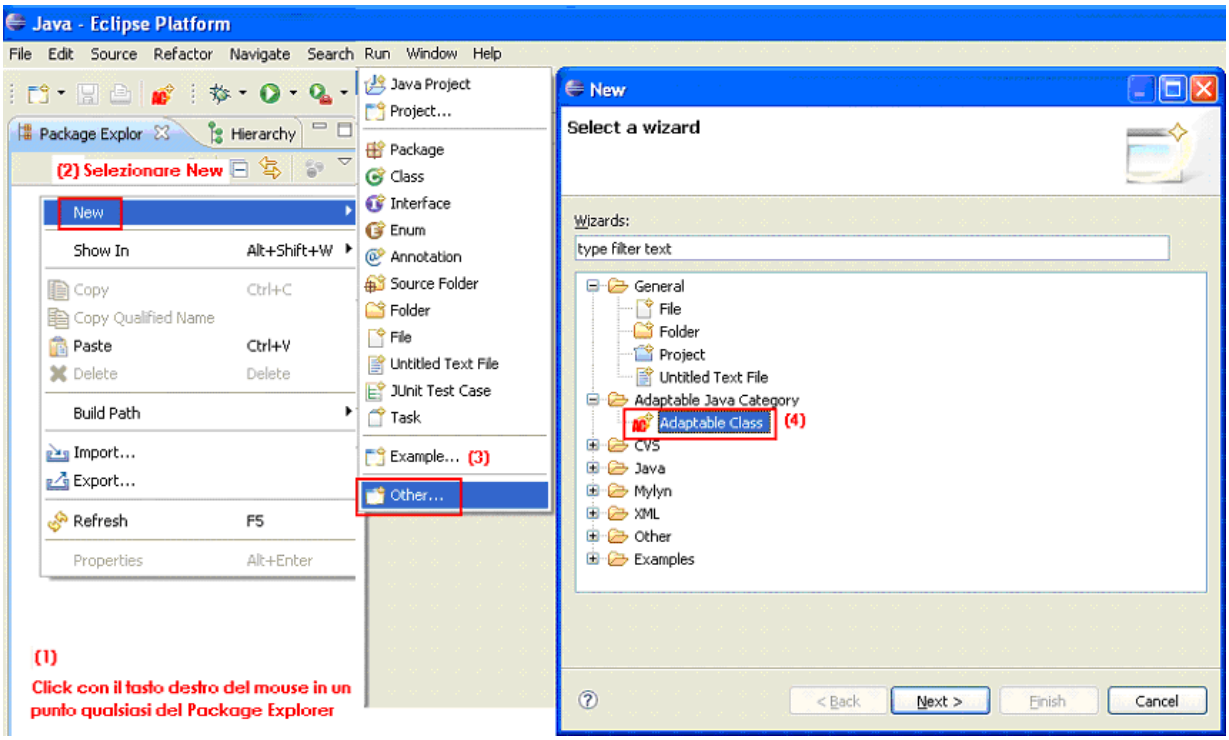

Fig.12: Generazione di una classe Adaptable da pop-up menù.

In tutti i casi suddetti, successivamente alla selezione dell'opzione "Adaptable Class", il wizard prosegue con la presentazione di una serie di dialog il cui compito è quello di consentire la scelta del progetto di appartenenza della nuova classe e di assegnarle un nome (il wizard ne propone uno di default) come illustrato in Fig.13.

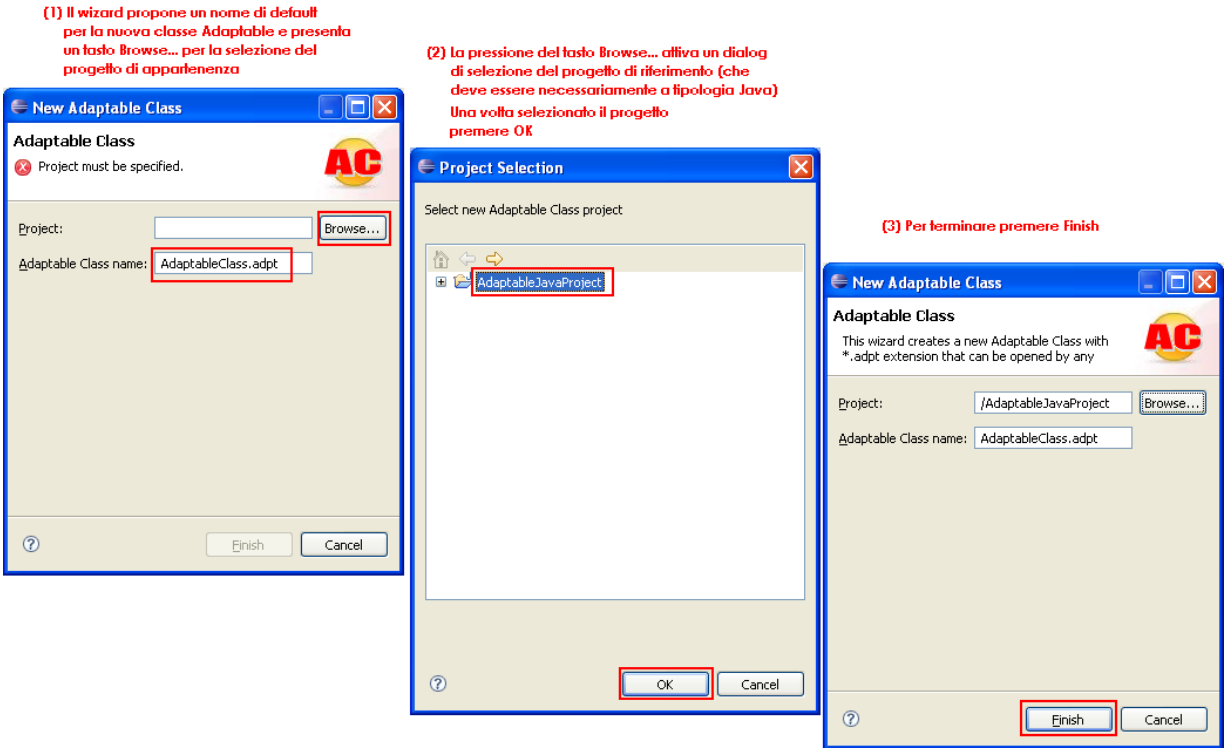

Fig.13: Indicazione dei riferimenti per una nuova classe Adaptable.

L'attivazione del "New Adaptable Class" wizard dalla menù bar e dal pop-up menù può essere resa ancora più immediata inserendo l'azione associata al "new Adaptable Class" direttamente tra le opzioni principali come in Fig.14.
La richiesta avviene con la procedura riportata in Fig.15.

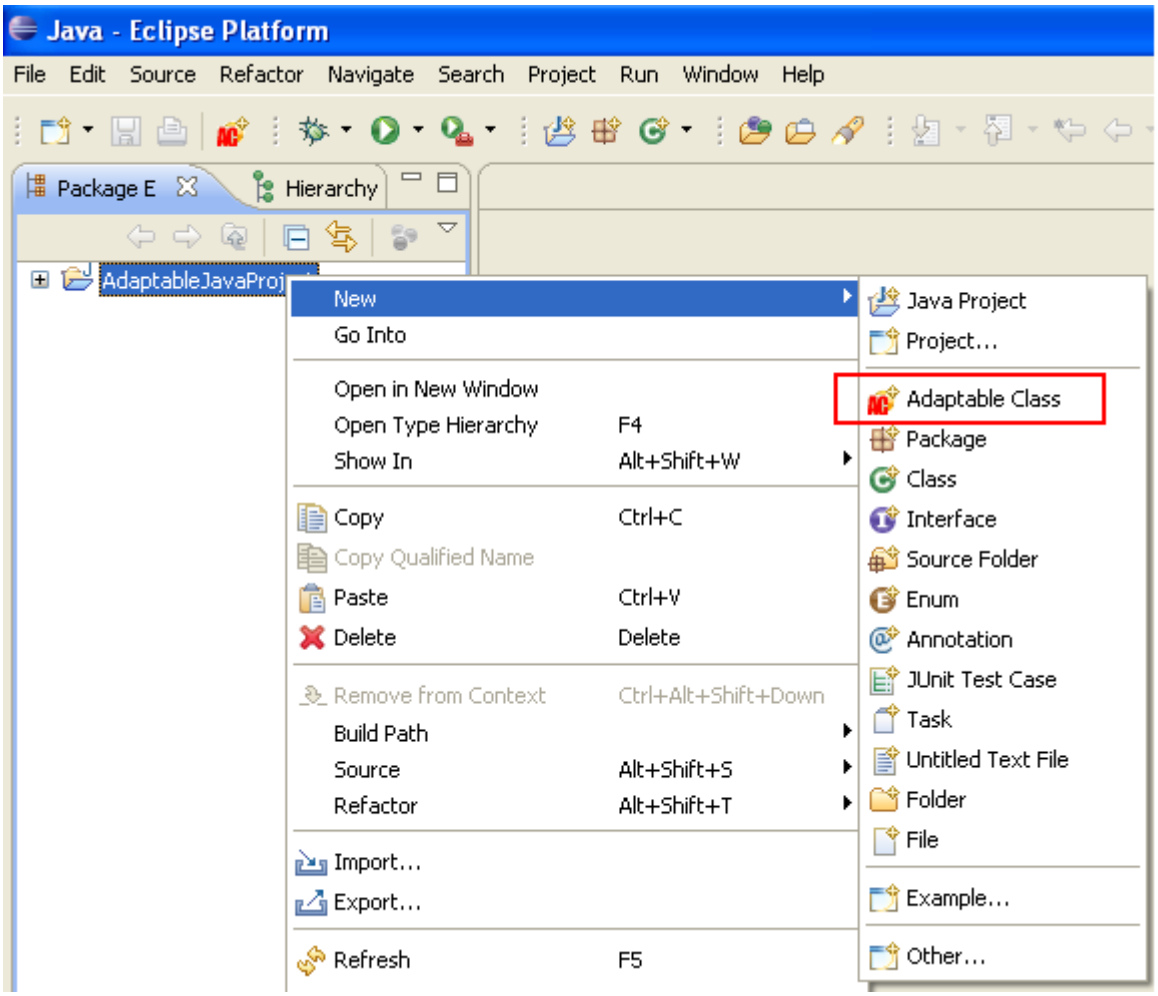

Fig.14: L'azione "New Adaptable Class" tra le opzioni principali del pop-up menù.

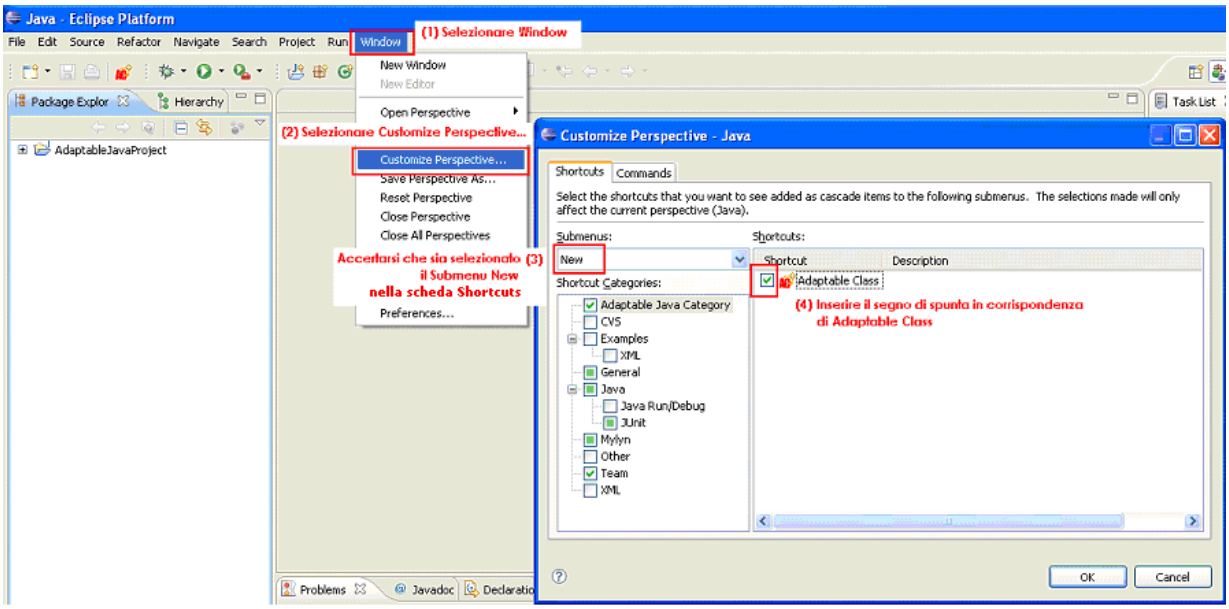

Fig.15: Procedura di richiesta "New Adaptable Class" tra le opzioni principali del pop-up menù.

La pressione del tasto Finish a conferma della creazione di una nuova classe Adaptable provoca le seguenti azioni:

1. Generazione della nuova classe Adaptable con il nome e nel progetto di riferimento indicati nei vari dialog del wizard; La classe viene inizializzata sulla base di un template prestabilito e collocata in un source folder ("/adpt-src"), appositamente creato qualora non risulti già esistente;
2. Generazione, se non esistente, di un file di default dei conflitti (conflict.xml) in una cartella prestabilita ("/adpt-conflict") anch'essa creata se non esistente . Il file viene inizializzato con un template predefinito;
3. Generazione, se non esistente, di un file di default (conflict.xsd) contenente lo schema di definizione del file dei conflitti precedentemente menzionato nella stessa cartella del file dei conflitti ("/adpt-conflict"). Il file viene inizializzato con un template predefinito;
4. Generazione, se non esistente e soltanto con lo scopo di organizzare i vari oggetti del progetto, di un folder destinato a contenere le Resources Definitions ("/adpt-resource").

3.1. Adaptable Java Model decorators

La visualizzazione di tutti gli oggetti generati

in questa fase, che costituiscono il modello "Adaptable Java", è sempre associata a specifici "decorators" che ne rendono più immediata l'individuazione visiva in tutte le fasi di sviluppo e gestione del progetto.
La Fig.16 mostra i vari decorators visualizzati nella Package Explorer View che vengono riproposti in tutti i contesti operativi.

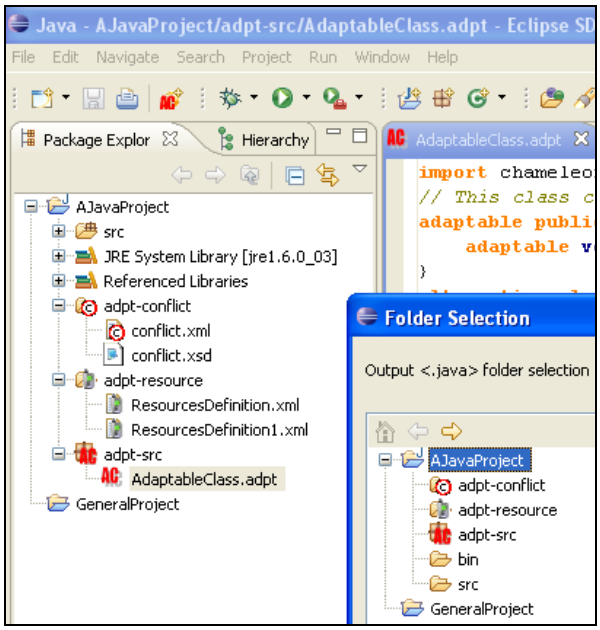

Fig.16: I decorators del modello Adaptable Java.

4. L'Adaptable Java Editor del ChameleonIDE

L'Editor per la gestione del linguaggio Adaptable Java, appositamente realizzato e fornito nel Plug-In ChameleonIDE, nasce con l'ambizione di "emulare" al meglio il JDT Editor di Eclipse per due motivi principali:

- L'indiscussa e universalmente riconosciuta completezza delle funzioni e degli strumenti offerti dal JDT Eclipse che lo hanno reso il principale ambiente IDE al mondo utilizzato nello sviluppo di progetti Java;
- Il tentativo di vincere la diffidenza comunemente riservata a tutti i nuovi tool di sviluppo, proponendo invece ai milioni di sviluppatori Java Eclipse un ambiente che si presenta immediatamente familiare e di semplice apprendimento.

Premesso che l'Editor viene associato automaticamente a tutte le classi Adaptable (estensione .adpt) ed è costruito sulla base dei costrutti e delle keywords del modello Adaptable Java, ne riassumiamo di seguito le funzionalità principali riportando nelle successive pagine alcune schermate esemplificative:

- **Syntax Highlighting**: la visualizzazione delle varie parti di codice secondo una colorazione completamente personalizzabile (commenti singoli, commenti multilinea, parole chiave, costanti, tipi, costrutti "speciali");
- **Font Customization**: delle varie parti di codice secondo font completamente personalizzabili nel tipo, nello stile e nella dimensione del carattere di scrittura;
- **Auto-Indent Strategy**: per l'applicazione automatica dei rientri di linea durante la scrittura del codice;
- **Double-Click Strategy**: per l'individuazione automatica di blocchi logici di codice quando si esegue un doppio click, ad esempio tra due parentesi graffe, tonde o quadre (aperte e chiuse) oppure tra una coppia di doppi apici;
- **Text-Hovering**: per la visualizzazione in modalità "sovrapposta ed evidenziata" della porzione di codice o dei token selezionati per un focus immediato degli stessi;
- **Word Completion**: attivabile come nel JDT con la sequenza di tasti Alt+"/" che scorre la lista delle keywords disponibili in modo da semplificare e velocizzare la scrittura dei programmi;
- **Define Folding Region**: che consente di nascondere momentaneamente blocchi di codice (visualizzando un segno "+" sulla Marker bar alla sinistra) liberando così il campo visivo al codice di interesse al momento. Il blocco nascosto può essere ripresentato in qualsiasi momento cliccando sul simbolo "+", che diverrà a quel punto un simbolo "-"; Il Folding Region si attiva

selezionando il blocco di codice da "nascondere", quindi premendo il tasto destro del mouse in un punto qualsiasi della Editor View, infine selezionando l'azione "Define Folding Region" dal pop-up menù attivato.

- **Content Assist**: per la presentazione di templates completamente personalizzabili volti alla semplificazione del processo di scrittura del codice. I templates disponibili sono attivabili, come nel JDT, con la sequenza di tasti Ctrl+Space e si propongono automaticamente nel rispetto della porzione di codice eventualmente già digitata al momento della loro attivazione. Qualora il template selezionato presentasse delle variabili, esse potranno essere valorizzate una ad una mediante l'utilizzo del tasto di tabulazione, completandosi automaticamente negli altri punti in cui risultassero ripetute, così come avviene nella JDT. La gestione dei templates, particolarmente ricca di contenuti, viene descritta in un apposito paragrafo.

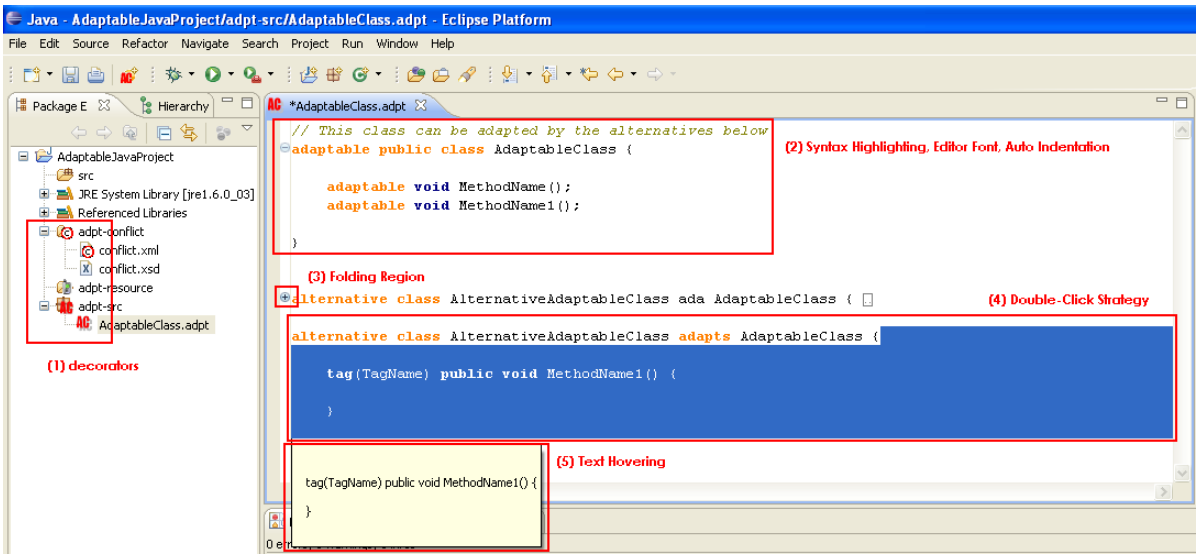

Fig.17: Syntax Hihglighting, Editor Font, Auto Indentation, Double-Click, Text Hovering e Folding.

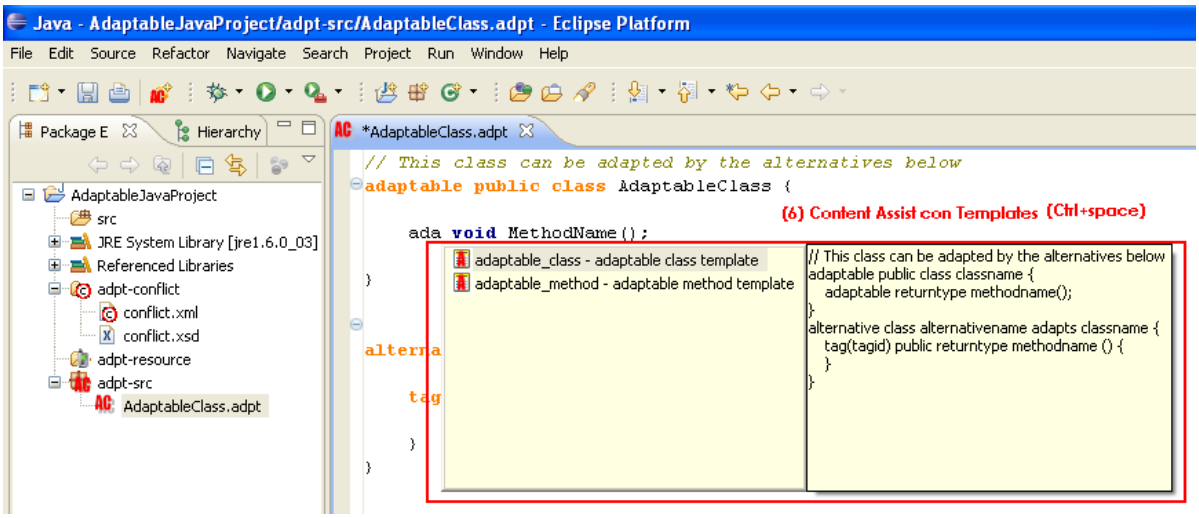

Fig.18: Content Assist con Templates.

4.1. Annotation e Resources Annotation wizard

Particolare interesse riveste la gestione delle Resources Annotation che offre la possibilità di elencare in maniera completamente dinamica le caratteristiche dei dispositivi da indicare nella costruzione di una applicazione adattabile.
Il prodotto consente in fase di editing di attivare un vero e proprio wizard dedicato alla costruzione di una istruzione resourceAnnotation, parametrizzata sulla base delle dichiarazioni presenti nel file ResourceDefinition.xml desiderato, che può essere selezionato direttamente a runtime. La costruzione dinamica dei dialogs, di cui il ChameleonIDE è provvisto, è resa possibile da uno specifico processo di traduzione istantanea in tecnica SAX/DOM che parte da un Resource Definition file di tipo .xml.
Il wizard viene attivato mediante un click con il tasto destro del mouse nel punto in cui si desidera inserire l'istruzione resourceAnnotation, e attraverso opportuni passi guidati si completa con l'inserimento dell'istruzione parametrizzata oltre che l'inserimento dell'istruzione di import del package di riferimento della classe Annotation.java. Anche in questo caso è possibile indicare un nome package a proprio piacimento modificando quello preimpostato di default.
Una rappresentazione dell'intero processo è raffigurata nelle Fig.19, Fig.20 (esempi di dialog dinamici) e Fig.21 mentre due esempi di Resource Definition sono rappresentati in Fig.22.

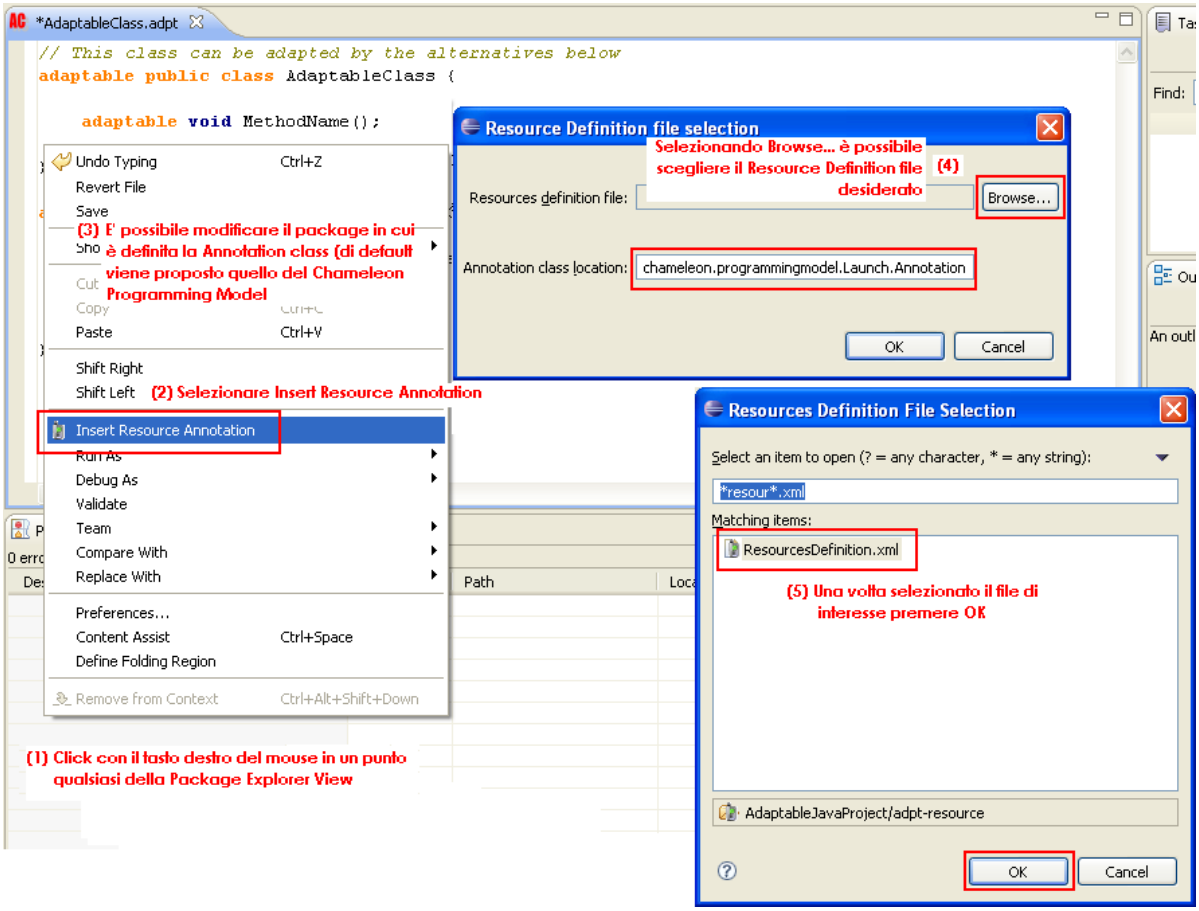

Fig.19: Insert Resource Annotation dialog.

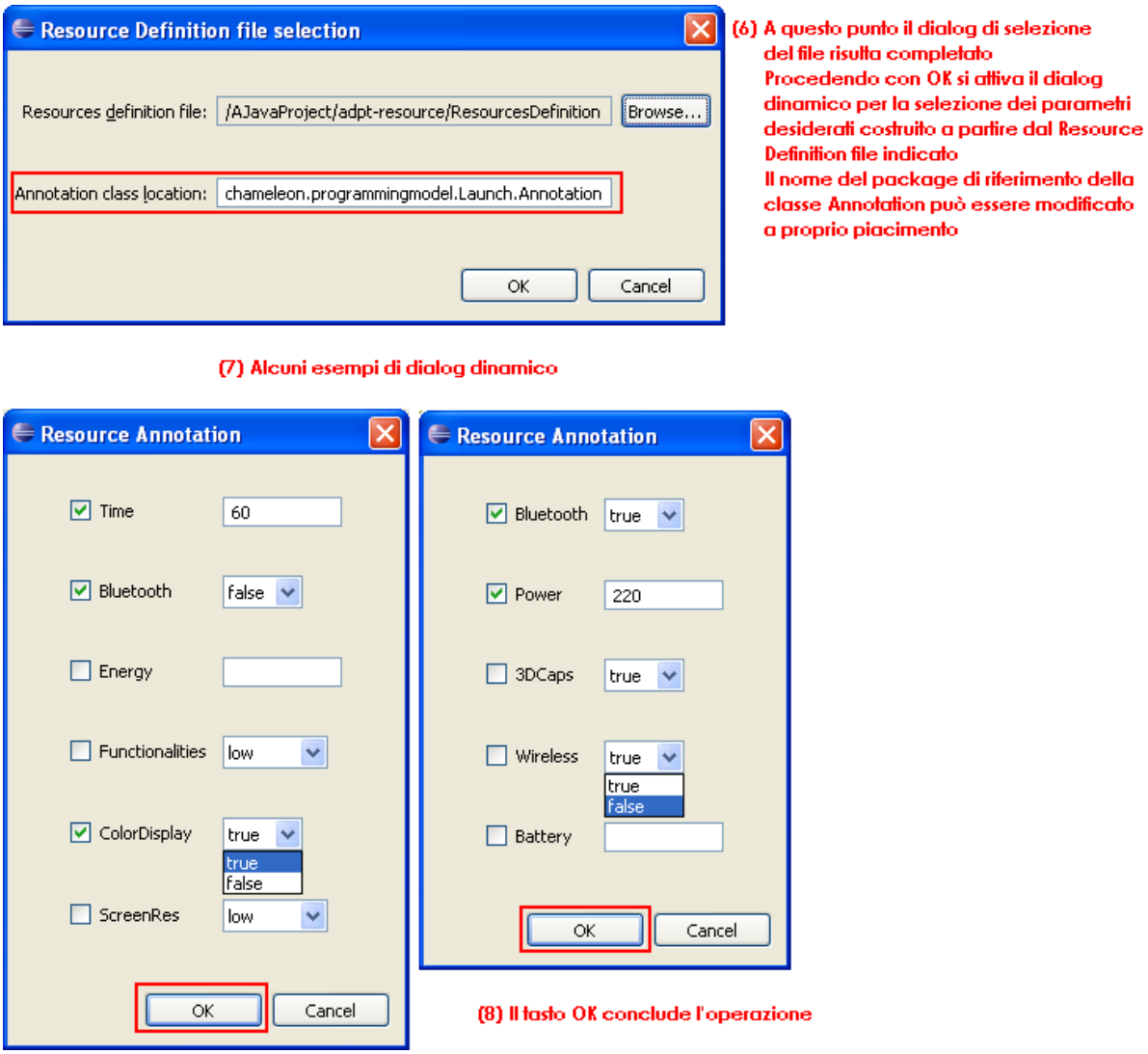

Fig.20: Alcuni dialog "dinamici" di Insert Resource Annotation.

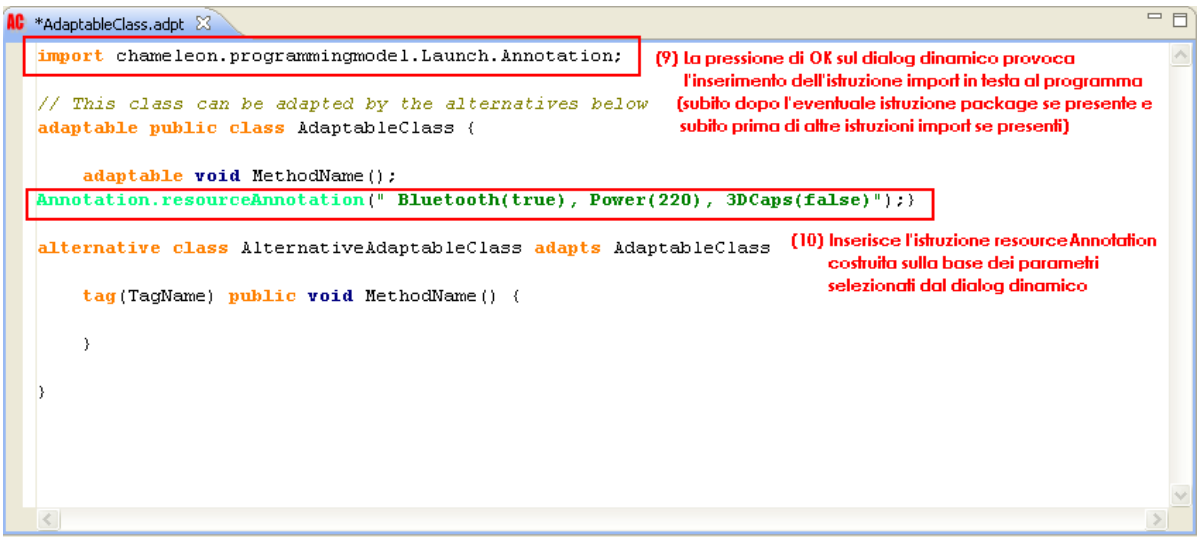

Fig.21: Risultato della Insert Resource Annotation.

```
<?xml version="1.0" encoding="UTF-8"?>

<declaration>
  <enumeration id="res">
     <value id="low"/>
     <value id="medium" />
     <value id="high" />
  </enumeration>

     <resource id="Time" type="natural" summable="true" />
     <resource id="Bluetooth" type="boolean" summable="true" />
     <resource id="Energy" type="natural" summable="true" />
     <resource id="ScreenRes" type="res" summable="true" />
     <resource id="ColorDisplay" type="boolean" summable="true" />
     <resource id="Functionalities" type="res" summable="true" />
</declaration>

=========================================================================

<?xml version="1.0" encoding="UTF-8"?>

<declaration>
  <enumeration id="res">
     <value id="low" />
     <value id="medium" />
     <value id="high" />
  </enumeration>

  <resource id="Power" type="integer" summable="true" />
  <resource id="Bluetooth" type="boolean" summable="true" />
  <resource id="3DCaps" type="boolean" summable="false" />
  <resource id="Battery" type="integer" summable="true" />
  <resource id="Wireless" type="boolean" summable="false" />
</declaration>
```

Fig.22: Due esempi di Resource Definition.

E' inoltre possibile inserire velocemente anche altre istruzioni riferite alle Annotations (callAnnotation e loopAnnotation) avvalendosi dei templates opportunamente predisposti e richiamabili con la sequenza di tasti Ctrl+Space (Content Assist) come in Fig.23.

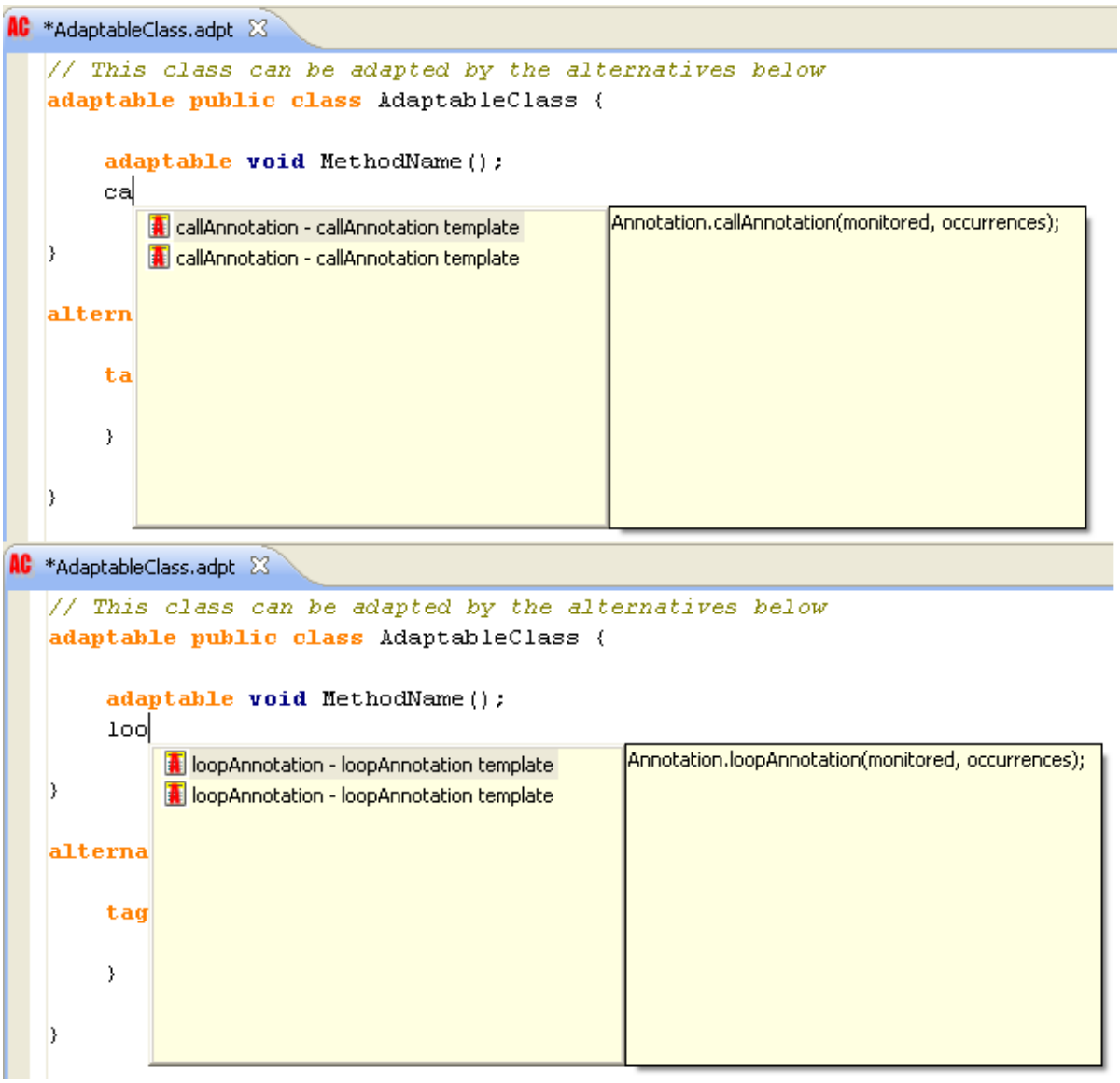

Fig.23: Content Assist con Templates per loopAnnotation e callAnnotation.

5. L'Adaptable Java Build

Una volta editato il nostro programma si passa al processo di Building.
Se non si è provveduto in precedenza è necessario a questo punto operare la procedura descritta nel precedente capitolo "Configurazione del Plug-In ChameleonIDE".
L'attivazione del Building avviene dal menù bar ed è collocata insieme alle altre funzioni di Build di Eclipse (Build All, Build Working Set, Build Automatically, …) presenti sotto la voce Project.
In questo nuovo contesto compare la nuova opzione "Build Adaptable Class" che se richiesta attiva uno specifico dialog mirato alla composizione dei parametri di Building.
I parametri selezionabili riguardano:

- La classe Adaptable da compilare;

- L'indicazione di compilazione di un'intero source folder (quindi di tutte le classi Adaptable in esso presenti). Se richiesta il source folder coinciderà con quello in cui è presente la classe Adaptable selezionata;
- Il Conflict Declaration file da utilizzare (se richiesto)
- Il ResourceDefinition file da utilizzare (se richiesto)
- L'indicazione di compilazione delle classi Java: Se non richiesto esplicitamente con questo parametro il processo di Building termina con la traduzione delle classi Adaptable in classi Java standard (.java), senza proseguire con la generazione dei file .class. Si è preferito infatti lasciare completa libertà di scelta all'utente che potrebbe avere l'esigenza di ultimare il processo in un secondo momento (ad esempio a fronte di qualche particolare caso in cui sia necessario modificare direttamente le classi Java standard) avvalendosi direttamente del Java Incremental Builder di Eclipse (JDT) oppure riattivando nuovamente il Chameleon Builder, questa volta richiedendo il processo completo;
- L'indicazione della directory in cui verranno generati i file .class (richiesto soltanto se in precedenza era stato indicato di eseguire completamente il processo di Building);
- Varie opzioni di compilazione (se richieste) la cui composizione ed il cui significato sono consultabili direttamente sulla pagina http://help.eclipse.org/ attivabile dal tasto alla destra del campo contraddistinto dall'icona .

Al termine del processo di Building compare un dialog di conferma del refresh automatico del workspace e le classi generate appariranno nel folder di output indicato.
Una importante caratteristica del processo di Building è la sua attivazione mediante la tecnica delle Launch Configuration che consente di riproporre sempre gli ultimi processi eseguiti direttamente dalla menù bar, senza ripetere nuovamente l'intero processo di composizione dei parametri di Building.
Premendo infatti il tasto dalla toolbar si attiva automaticamente l'ultima configurazione "Java Application" eseguita, quindi la nostra ultima compilazione effettuata.
E' inoltre possibile mantenere una specifica Launch configuration di interesse richiamandola dal menù "Run As" (procedendo poi con "Open Run Dialog…") e rinominandola. In questo modo non verrà ricoperta da successive richieste di Building (che per default assumono sempre lo stesso nome "BuildAdptClass") e potrà quindi essere riutilizzata all'occorrenza (Fig.24).

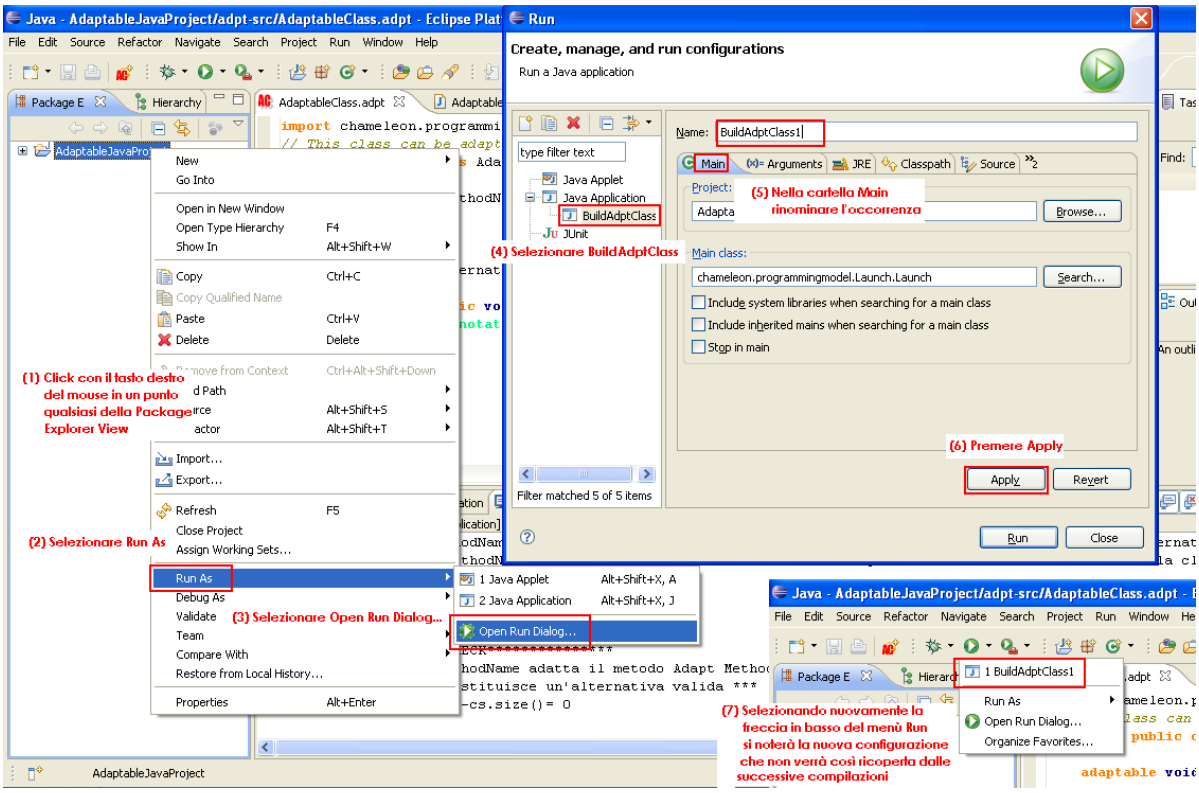

Fig.24: Richiamo e ridenominazione di una Launch Configuration.

Le figure successive (Fig.25, Fig.26 e Fig.27) illustrano il processo di selezione dei parametri di Building, l'attivazione del processo ed il risultato ottenuto.

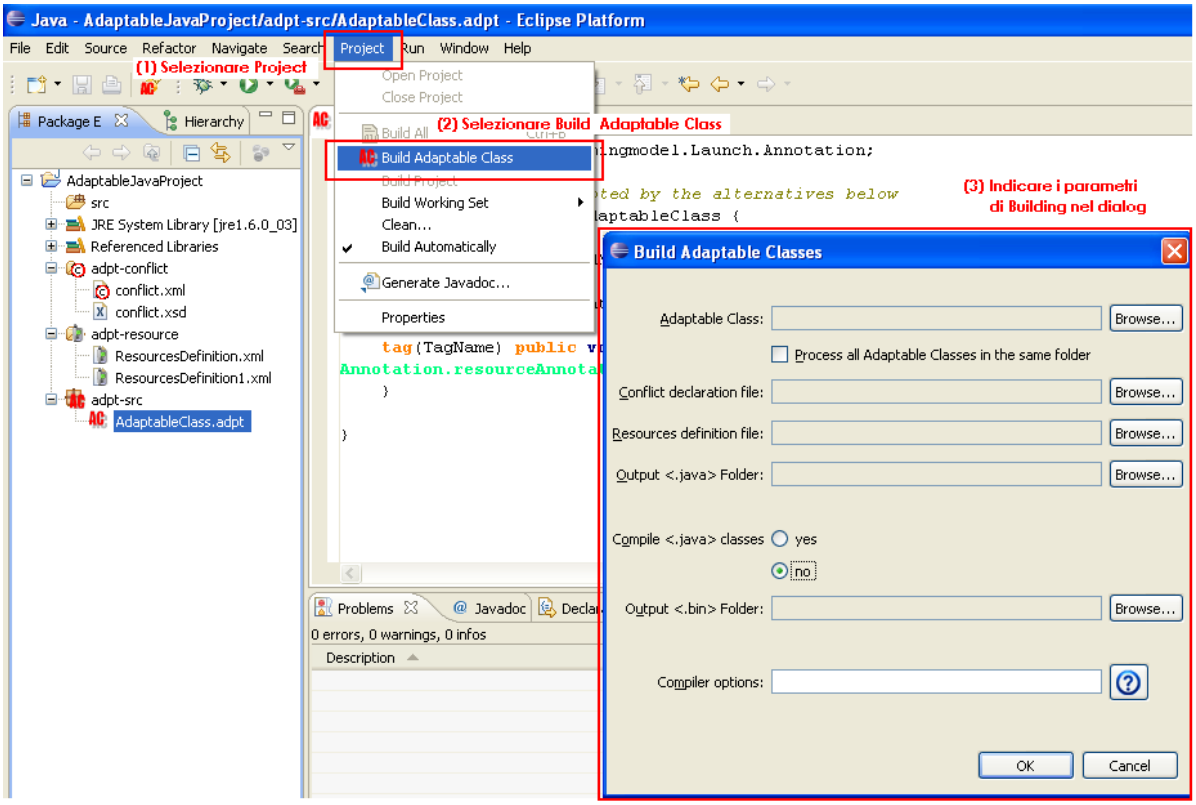

Fig.25: Attivazione del processo di Building.

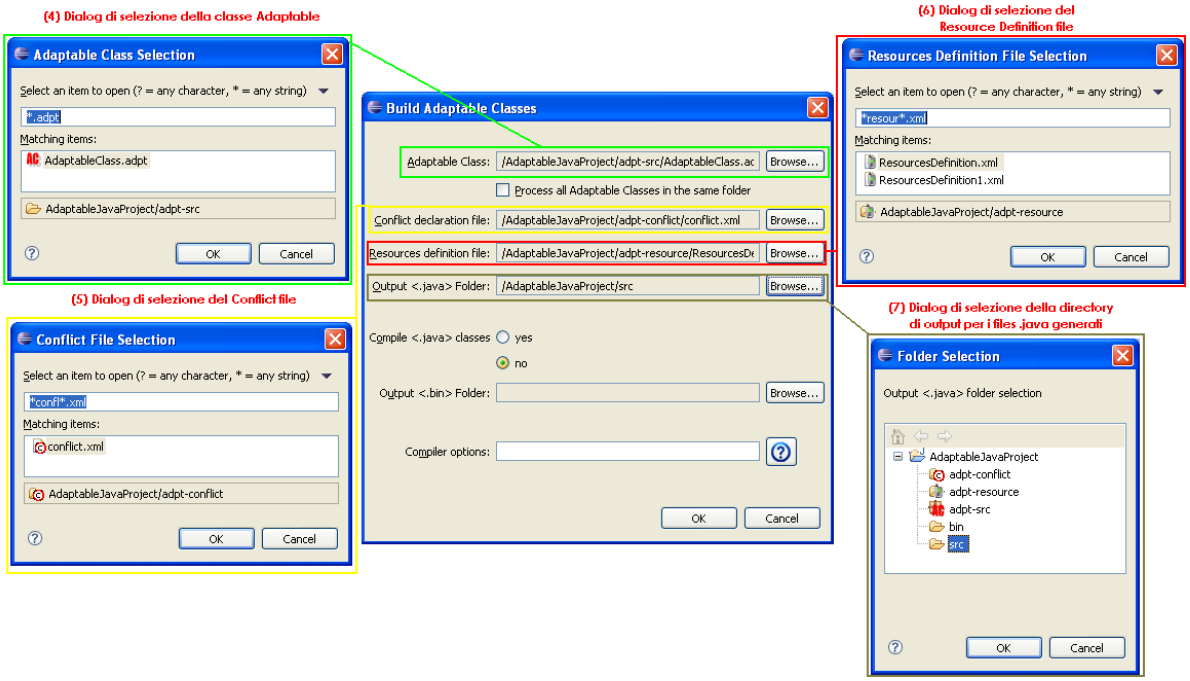

Fig.26: Selezione dei parametri di Building.

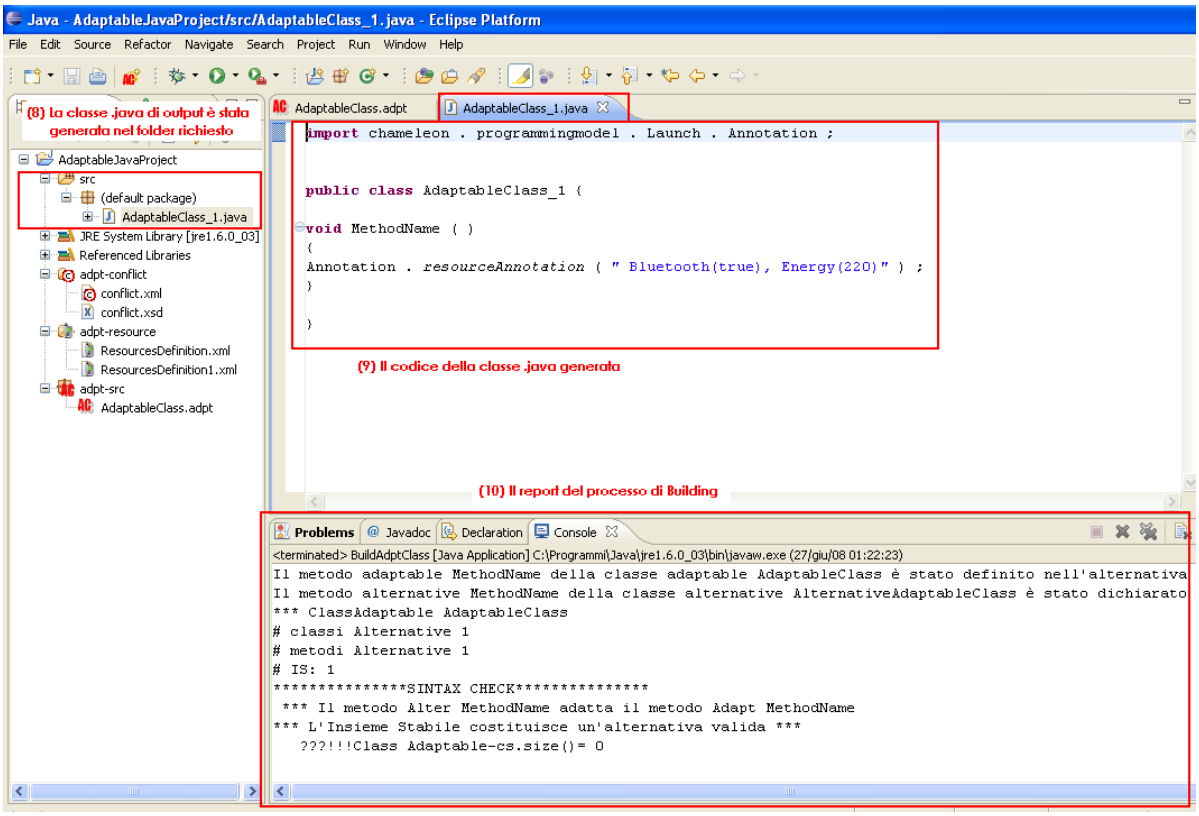

Fig.27: Risultato del processo di Building.

6. Adaptable Java Preferences Pages

Il Plug-In ChameleonIDE offre un'ampio margine di personalizzazione dell'ambiente di sviluppo attraverso le Preferences Pages, già familiari perché di comune utilizzo sia per le configurazioni di Workbench, sia per quelle del Java Development Environment:

- Editor Font: consente la scelta del tipo di carattere da utilizzare nell'Adaptable Java Editor, nonché delle sue dimensioni e dello stile (normale, grassetto, corsivo);
- Syntax Coloring: offre la possibilità di personalizzare i colori di varie porzioni di codice quali
  - Single-line comments;
  - Multi-line comments;
  - Keywords: sia quelle Java standard (p.es. class, public, private, …) , sia le Adaptable Java (p.es. adaptable, adapts, alternative, …) ;
  - Types: ossia i tipi Java (p.es. boolean, void, char, …);
  - Costants: le costanti true, false e null;
  - Specials: ossia le istruzioni Annotation.resourceAnnotation, Annotation.callAnnotation e Annotation.loopAnnotation;
- Templates: Per la definizione dei templates utilizzati dal Content Assist durante la fase di editing. Il Plug-In ChameleonIDE nasce già corredato da una serie di templates di base che può essere ulteriormente arricchita a proprio piacimento e senza limiti da templates personalizzati oppure da tutti o parte dei templates già presenti in contesti compatibili con il contesto Adaptable Java (context=java e context=javadoc) mediante funzioni automatiche di import.

Le Adaptable Java Preferences Pages vengono attivate con la procedura descritta nella figura seguente (Fig.28):

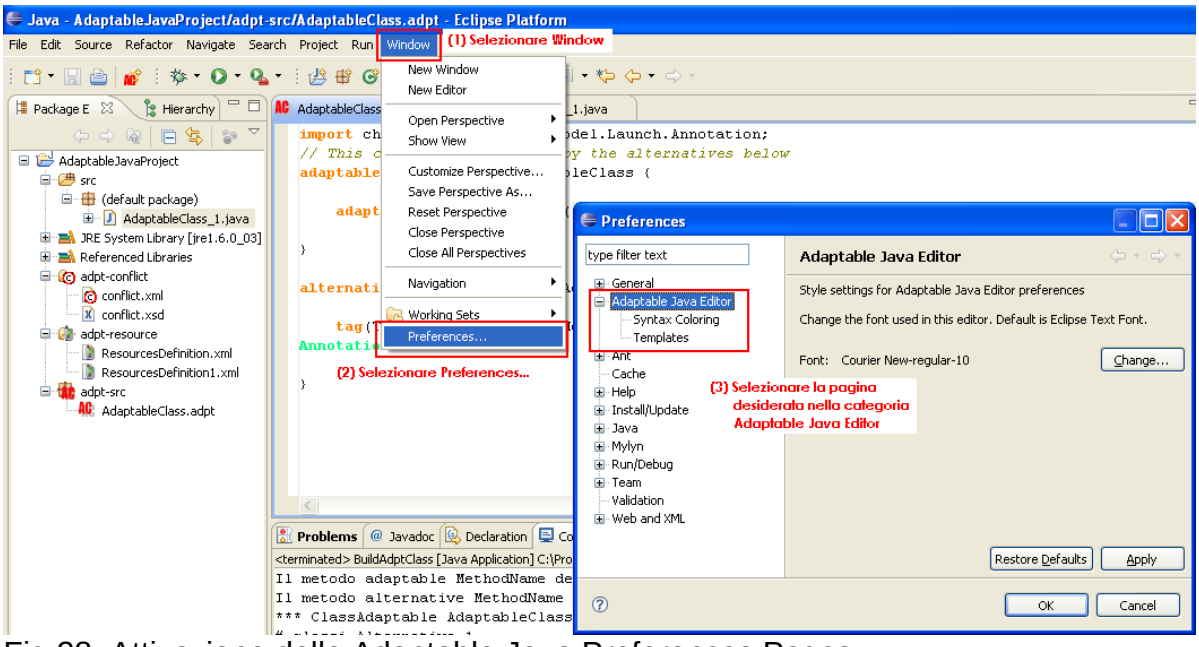

Fig.28: Attivazione delle Adaptable Java Preferences Pages.

Ciascuna delle nuove Preferences Pages consente sempre di ripristinare i valori di default attraverso un apposito comando a bottone (Restore Defaults).
Le figure successive rappresentano il contesto operativo delle Preferences Pages appena descritto.

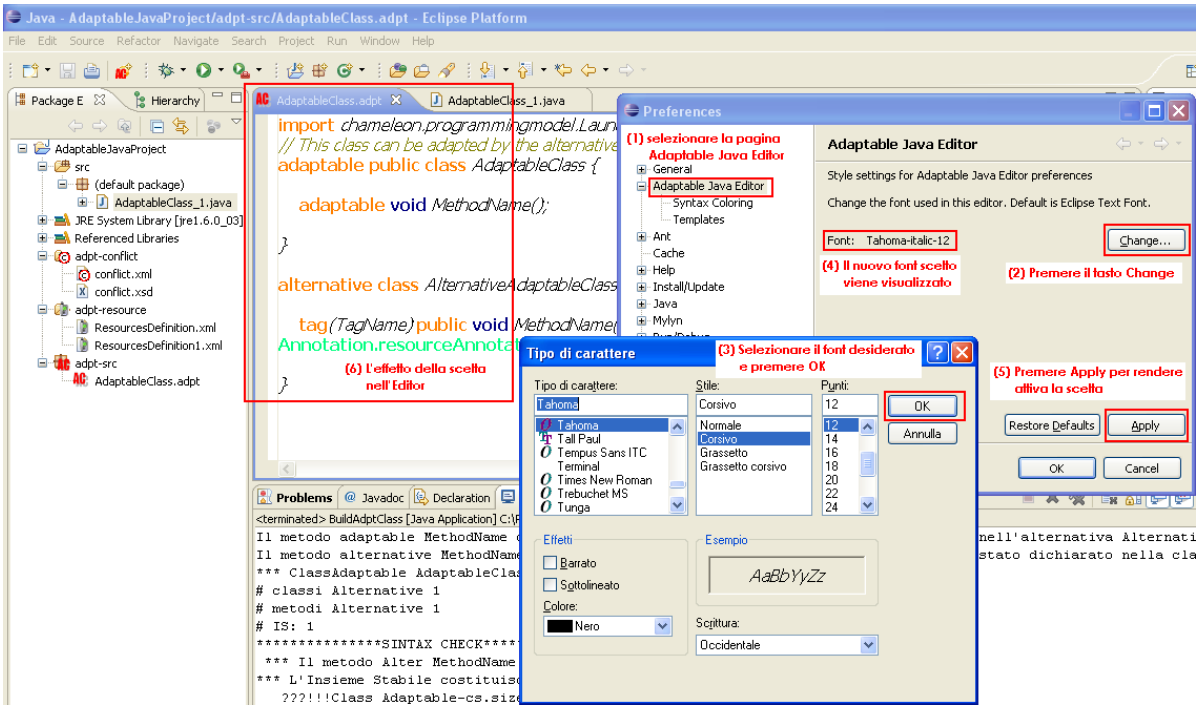

Fig.29: Preferences Page dell'Editor Font ed effetto delle variazioni operate.

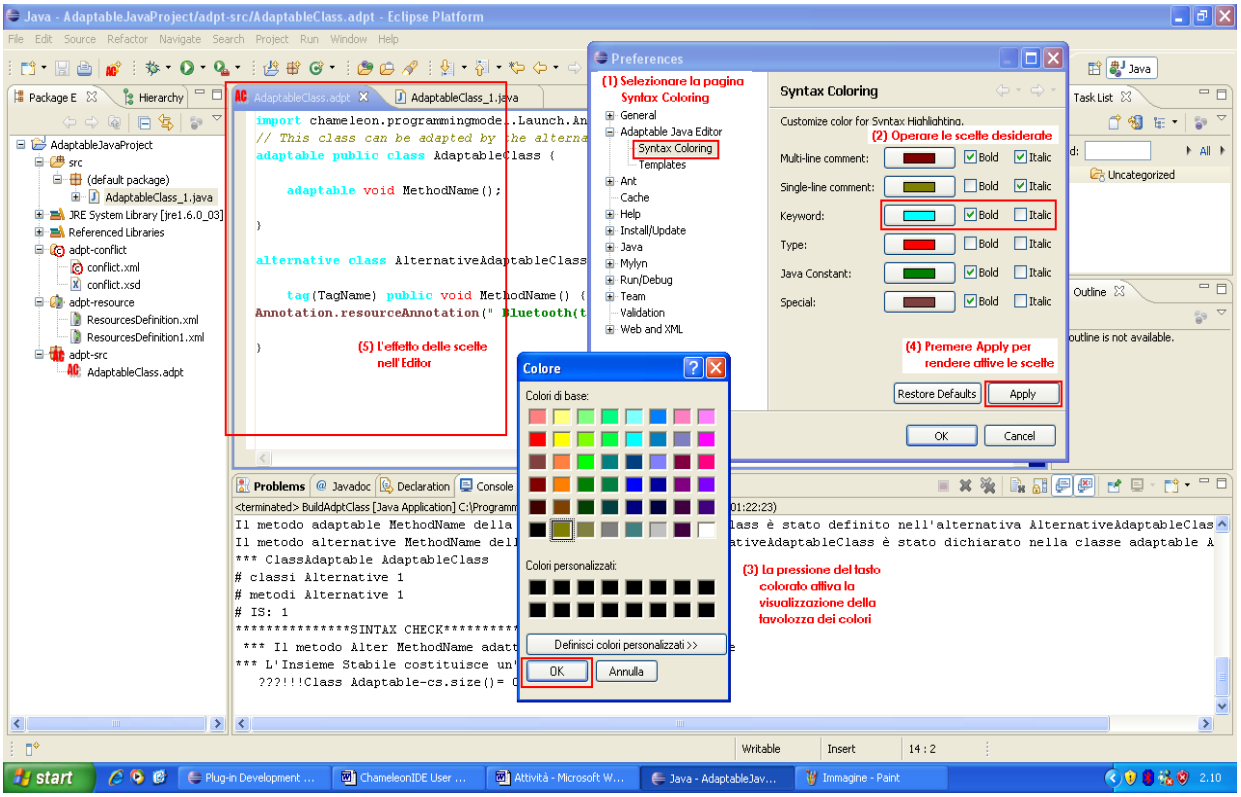

Fig.30: Preferences Page del Syntax Coloring ed effetto delle variazioni operate.

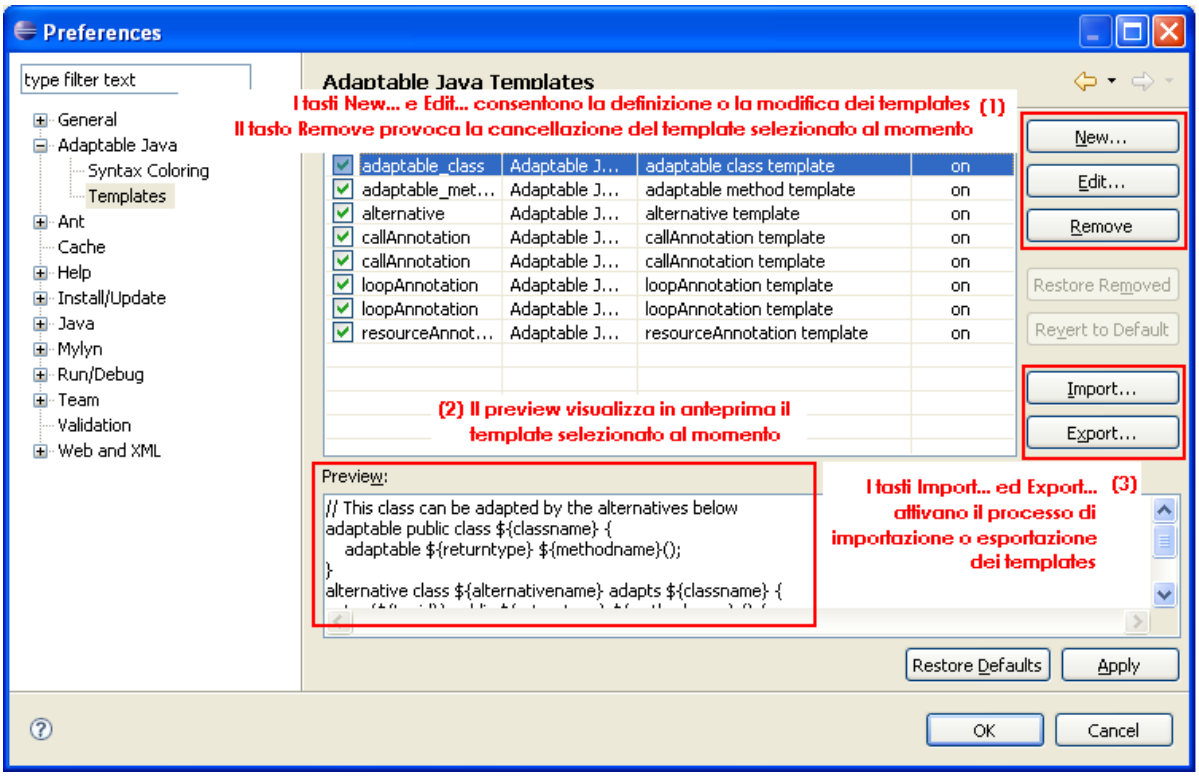

Fig.31: Preferences Page dei Templates (funzioni principale).

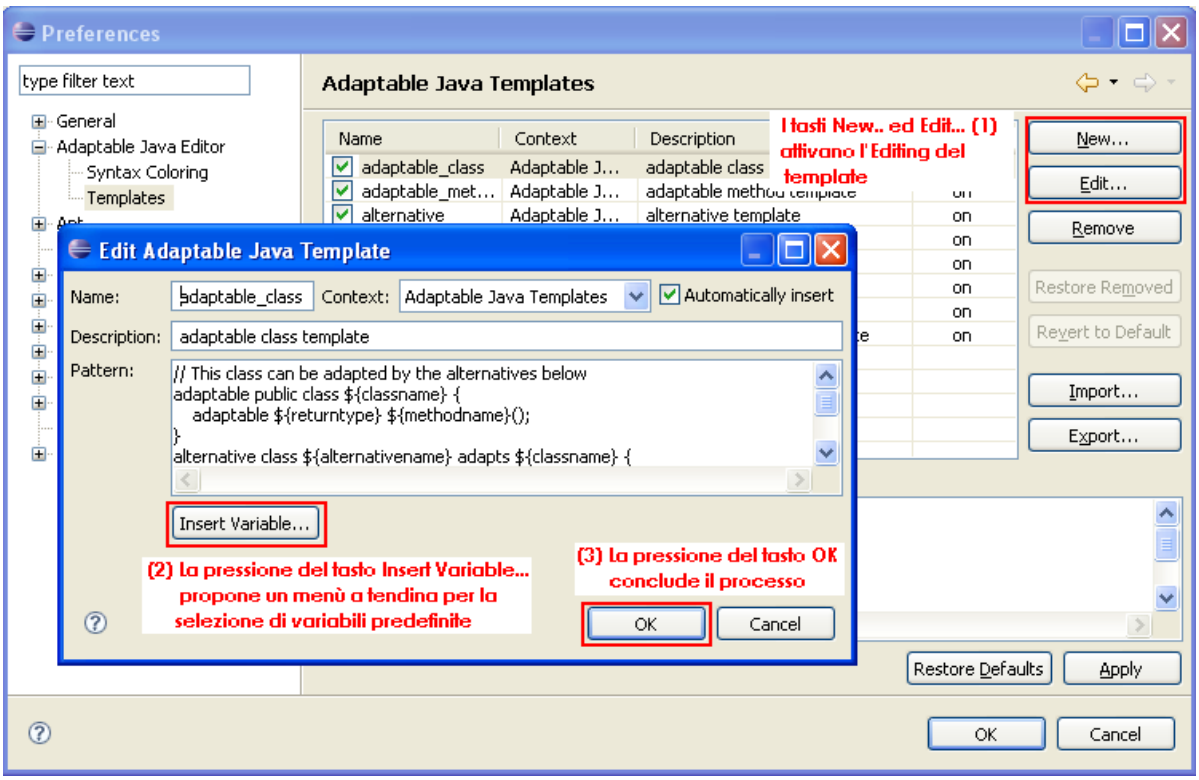

Fig.32: Preferences Page dei Templates (funzioni di editing).

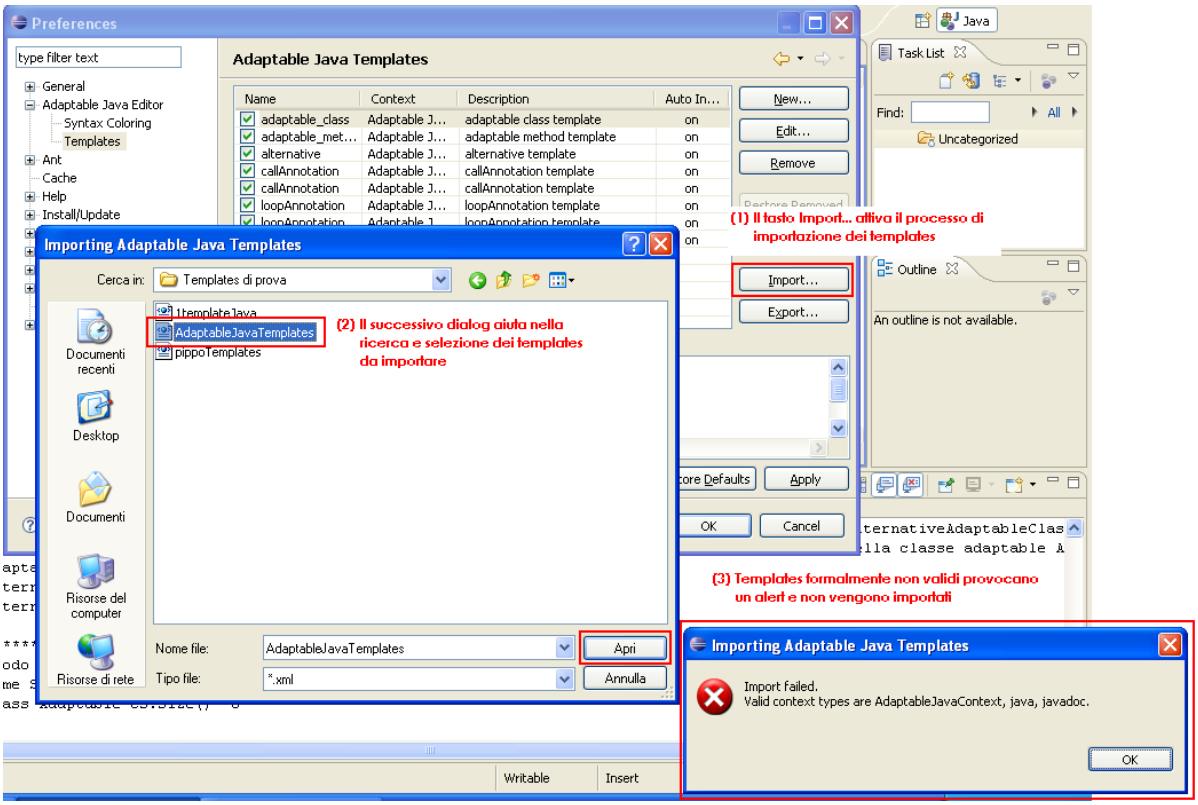

Fig.33: Preferences Page dei Templates (funzione di import).

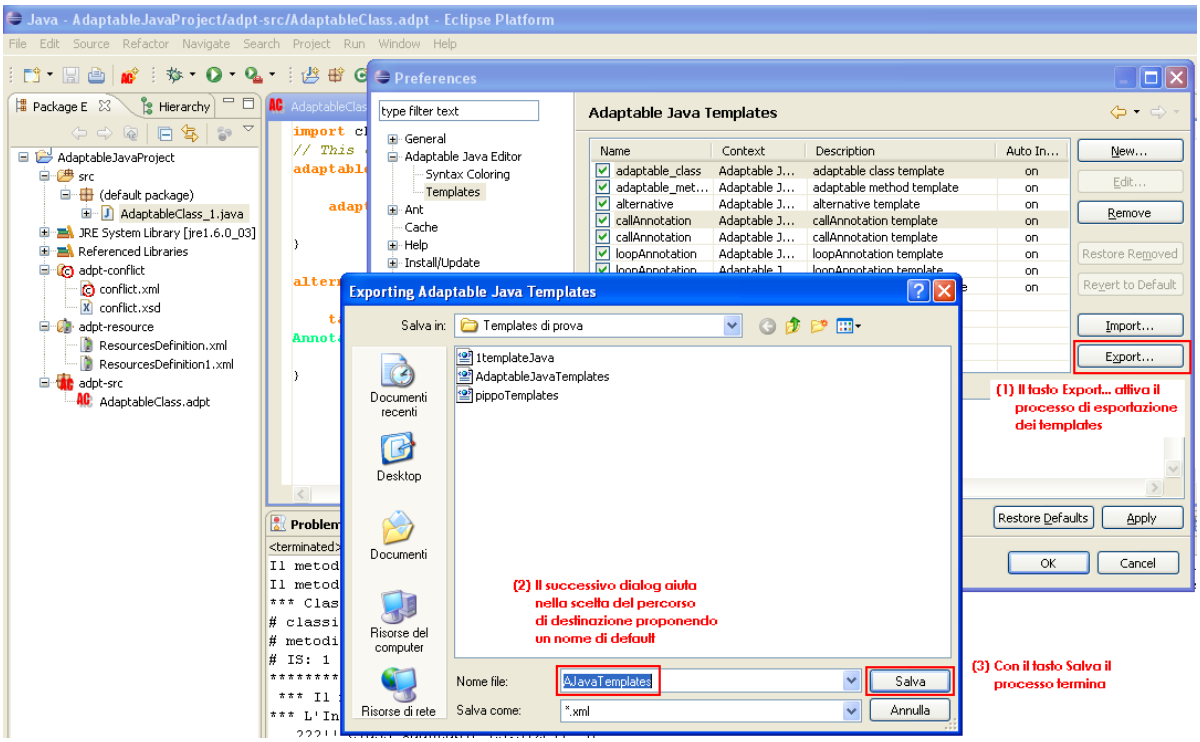

Fig.34: Preferences Page dei Templates (funzione di export).

7. Appendice A - Adaptable Java Icons

	Classe Adaptable
	Adaptable Java class source folder
	Resource Definition file o folder
	Conflict declaration file o folder
	Toolbar Action per New Adaptable Class
	Adaptable Java Template (Content Assist)
	New Adaptable Class Wizard
	Link alla pagina web dei parametri di compilazione

8. Appendice B - Le classi `Launch.java` e `Annotation.java`

La classe `Launch.java`

```
package chameleon.programmingmodel.Launch;

import chameleon.programmingmodel.Tool.*;

/**
 * <p>Title: Launch </p>
 *
 * <p>Description: Questa classe viene usata come esecuzione del
 *                 Programming Model per la fase di Build.
 *                 Parametri ricevuti:
 *                 - Path completo della Adaptable Class da compilare
 *                   (se il Path termina con "/" si intende una directory);
 *                 - Path completo del Conflict file
 *                   (se non previsto viene indicato con "none");
 *                 - Path completo del Resources Definition file
 *                   (se non previsto viene indicato con "none");
 *                 - Path completo indicante la directory di output
 *                   dei file .java prodotti terminante per "/";
 *                 - Path completo indicante la directory di output
 *                   dei file .bin prodotti terminante per "/"
 *                   (se non previsto va indicato con "none");
 *                 - opzioni di compilazione
 *                   (se non previsto va indicato con "none").
 * </p>
 *
 * <p>Copyright: Copyright (c) 2008</p>
 *
 * @author Dipartimento di Informatica UNIVAQ
 * @version 1.0
 */

public class Launch {
   /**
    *
    * Metodo usato per lanciare il tool
    * viene istanziato un oggetto di tipo
    * MainProgModel e quest'ultimo richiama
    * il metodo MainUnit
    *
    * @param args[0]    [String] : file o directory delle adaptable classes
    *                              (se directory termina con "/")
    * @param args[1]    [String] : file dei conflitti | none
    *
    * @param args[2]    [String] : file delle resources definition | none
    *
    * @param args[3]    [String] : directory di output (.java)
    *                              (termina con "/")
    * @param args[4]    [String] : directory di output (.bin) | none
    *                              (se indicata termina con "/")
```

```
    * @param args[5..n] [String] : riga delle opzioni di compilazione | none
    */

    public static void main(String[] args) throws Exception {

        System.out.println("Chameleon Programming Model Build Start ...");

         int numargs=args.length;
         System.out.println("Number of arguments provided ..............: " + numargs);

         if (numargs==0){
                System.out.println("Chameleon Programming Model Build End .....: No argument found");
                return;
         }

        int i = 0;
        while (i < args.length) {
                System.out.println("args["+i+"]: "+args[i]);
                i++;
        }

         MainProgModel prebuild = new MainProgModel();
         prebuild.MainUnit(args);

         }
    }
```

La classe `Annotation.java`

```
package chameleon.programmingmodel.Launch;


public class Annotation {
        public static void resourceAnnotation(String annotation) {
                // annotation deve essere una lista separata da virgole di ResID(Valore)
        }

        public static void loopAnnotation(boolean monitored, int occurrences) {
                // se monitored è vero vuol dire che il programmatore non è certo che il
                // ciclo verrà ripetuto al max occurrences volte, quindi il numero di volte
                // viene gestito come una variabile.
                // Le variabili appartenenti a cicli monitored vengono stampate in un file
                // LoopAnnotation.xml
                // ognuna associata con il rispettivo valore di occurrences
                // se monitored=false si assume che il ciclo viene ripetuto
                // occurrences volte.
        }

        public static void loopAnnotation(int occurrences) {
                // equivale a chiamare loopAnnotation(false, occurrences);
        }

        public static void callAnnotation(boolean monitored, int occurrences) {
                // messa prima di una chiamata a funzione (mutualmente) ricorsiva,
                // dice quante volte la chiamata deve essere ripetuta
                // monitored: come per le loopAnnotation
        }

        public static void callAnnotation(int occurrences) {
                // equivale a chiamare callAnnotation(false, occurrences);
        }

}
```