Ant Tasks for SmartFrog

Transforming deployment and testing of software

http://smartfrog.org/

Ant: the build framework

- Language: DTD-free XML
- Syntax: declarative listing of steps to build *targets in* a software *project*, dependencies between targets.
- Extensibility: tasks and types, listeners, selectors, conditions, more
- Embeddable
- Primary Focus: building software
- Others: deployment, installation, cross-platform scripts.

Ant tasks

- Java classes, usually extend org.apache.tools.ant.Task
- IntrospectionHelper automatically maps from XML attributes and elements to setter methods

```
public void setClasspathRef(Reference classpathRef) {
    this.classpathRef = classpathRef;
}
public void addClasspath(Path classpath) {
    this.classpath = classpath;
}
```

- Two lifecycle callbacks: init() and execute()
- Tasks are assumed to finish their work after execute()
- Composition by Project.createTask()

SmartFrog: the deployment framework

- Language: SmartFrog 1.0
- Syntax: declaration and customisation of components of a deployed system.
- Inheritance, aggregation and early/late resolution
- Extensibility: components
- Embeddable
- Focus: *deployment of software*

SmartFrog Components

- Java classes, usually extend org.smartfrog.sfcore.Prim
- Manually extract settings from the resource tree

```
wsddResource=sfResolve(Axis.WSDD_RESOURCE,"",true);
```

- More complex lifecycle

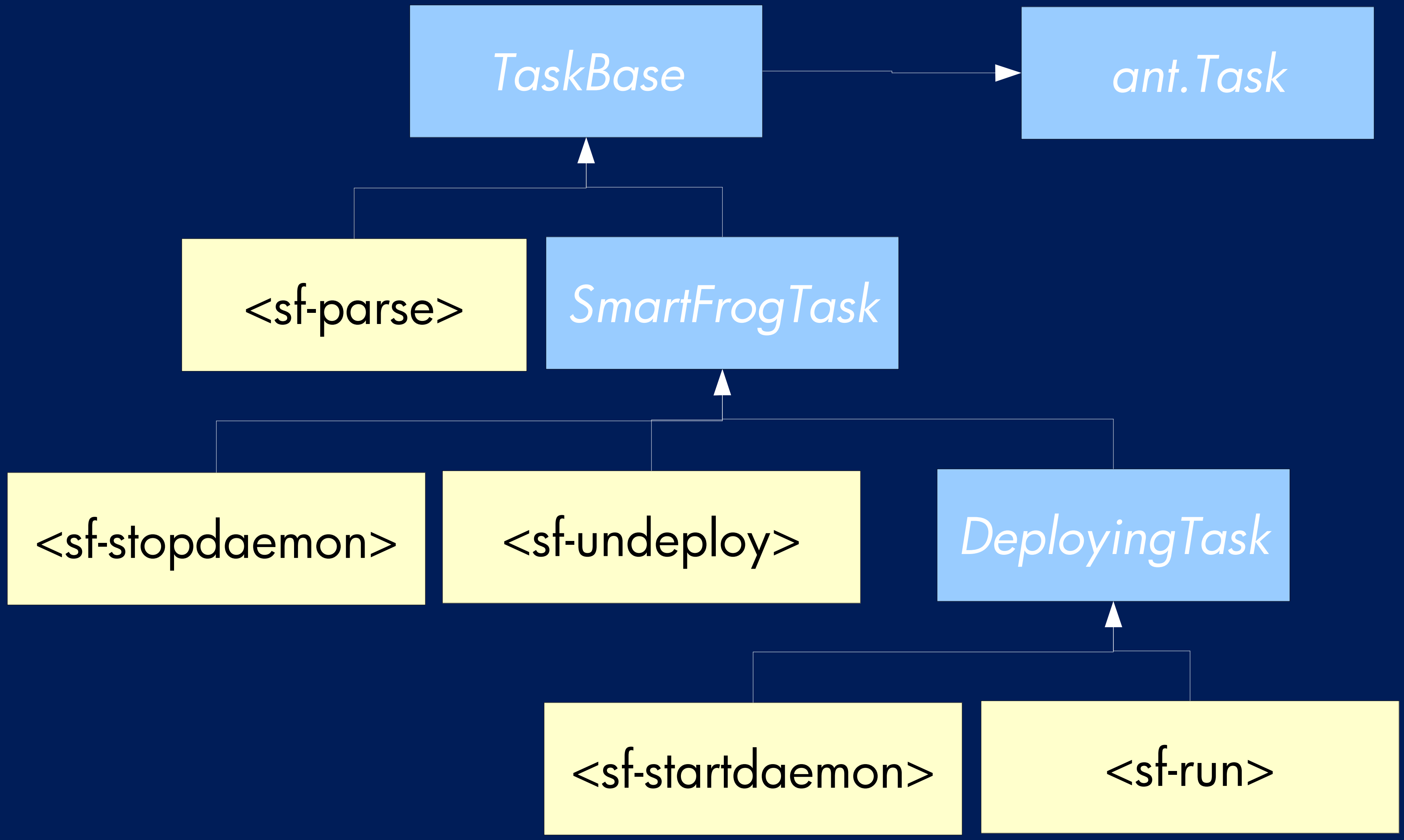
```
sfStart()
sfPing()
sfTerminateWith()
```

- Components run from started till terminated: in threads or external programs.
- Composition as per the deployment declaration: use RMI to talk to (potentially remote) components.

SmartFrog Ant Tasks

Run SmartFrog,
deploy and undeploy applications,
from Ant.

Five tasks, three new base classes



<sf-parse>

Pre-deployment validation of .sf file

```
<sf-parse file="valid.sf" verbose="true"/>
```

Parses one file, displays resolved description

```
<sf-parse>
  <source dir="." includes="**/*.sf"/>
</sf-parse>
```

Parses many files

<sf-deploy>

Deploy application(s)

```
<sf-deploy host="server">
  <application name="app"
     descriptor="org/example/deploy.sf"/>
</sf-deploy>

<sf-deploy failonerror="false">
  <application name="app1" file="files/deploy.sf"/>
  <application name="app2" file="files/deploy2.sf"/>
</sf-deploy>
```

Best practice: one application per task.

<sf-run>

Run app, terminate on exit

```
<sf-run timeout="5000">
  <application name="app"
     descriptor="org/example/deploy.sf"/>
</sf-run>

<sf-run failonerror="false">
  <application name="app1" file="files/deploy.sf"/>
  <application name="app2" file="files/deploy2.sf"/>
</sf-run>
```

Can validate deployment; blocks build till finished.

Inline Applications

- The <application> element supports inline deployment descriptors with Ant property expansion

```
<sf-deploy >
  <application name="app">
    #include "org/smartfrog/components.sf"
    Server extends Prim {
        port ${port};
        sfClass "org.example.appserverImpl";
    }
    sfConfig extends Server{
        sfProcessHost "${deployment.host}";
    }
  </application>
</sf-deploy>
```

<sf-undeploy>

Undeploy an application, including the daemon itself

```
<sf-undeploy host="server" application="test"/>

<sf-undeploy application="test" failonerror="false"/>

<sf-undeploy application="rootProcess" />
```

Only one app per undeploy; could be extended to take a list instead.

<sf-startdaemon>

Start daemon, setting up classpath and initial apps

```
<sf-startdaemon iniFile="default.ini"
  initialSmartfrogFile="default.sf" spawn="true" />

<sf-startdaemon logStackTraces="true" >
  <application name="app"
    descriptor="${resource.sf}"/>
  <assertions enableSystemAssertions="true">
    <enable/>
  </assertions>
</sf-startdaemon>
```

Issues with starting the daemon from Ant

- Thread blocks until the daemon exits.
- Daemon exits when Ant is stopped.
- Unless spawn=true, when all output gets lost (set smartfrog properties to redirect stdout and stderr)
- OK for simple testing; run in <parallel> with tests.

```
<parallel>
 <sf-startdaemon  timeout="${long.timeout}" />
 <sequential>
   <sf-block/>
   <sf-undeploy application="rootProcess" />
 </sequential>
</parallel>
```

<sf-block>

- Declare this with the Ant1.6 task extension model, <presetdef> : -

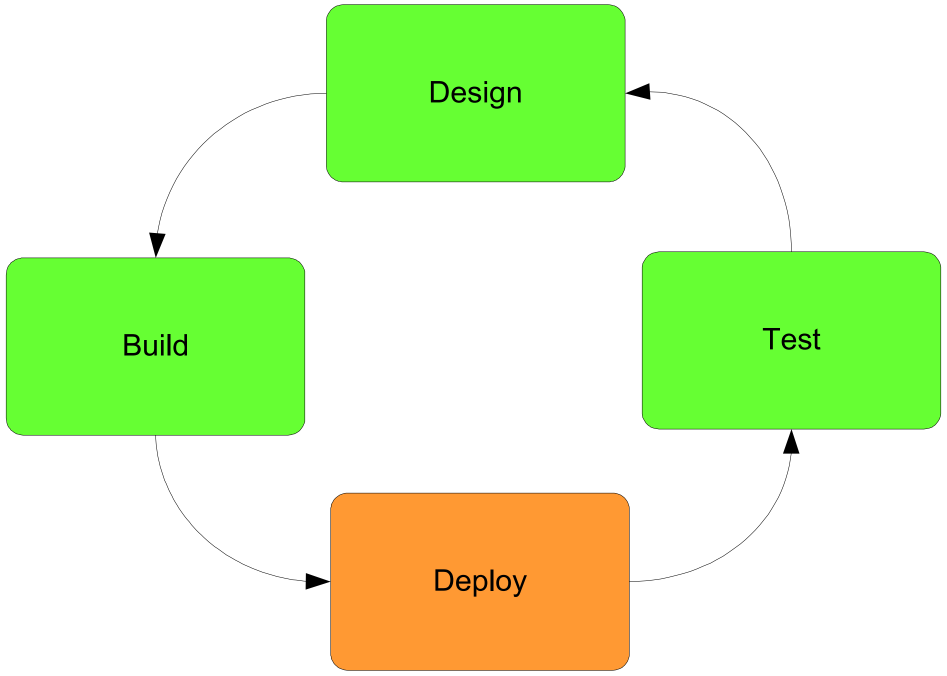
```
  <presetdef name="sf-block">
      <waitfor maxwait="10" maxwaitunit="second">
       <socket server="localhost" port="3800"/>
     </waitfor>
  </presetdef>
```

Use:

```
<sf-block timeoutproperty="smartfrog.missing"/>
<sf-block maxwait="15" />
```

Use with <sf-startdaemon/> to delay other threads

Deployment-centric development



No more waterfall between develop and deploy

Integrating deployment into Ant

- Validate .sf file before deploying <sf-parse>
- Deploy to a running daemon (local or remote) <sf-deploy>
- Run tests: <junit>
- Undeploy: <sf-undeploy>
- Generate test reports <junitreport>
- Fail on test failure. <fail>

- Also: start/stop a daemon if none was running

Classpaths

- SmartFrog tasks use the classpath the tasks were declared with.
- Or any nested <classpath>, classpathref attribute
- <sf-run> and <sf-startdaemon> can be given a classpath that includes all the JAR files of the app to deploy
- To deploy to a running daemon, you need to get the latest JAR files to the destination

Codebase

All deploying tasks take a <codebase> element that lists the codebases for deployment

<codebase url="http://server/data/project.jar"/>

<codebase file="dist/project.jar"/>

File attributes are turned into absolute URLs (e.g. file://c|/project/dist/project.jar)

URLs must be visible from all deploying nodes.

Copy to a shared filestore or web site for deploying to anything other than localhost

Security Issues

- When running security off, you don't need to do anything
- But anyone with port 3800 access can deploy anything they like.
- Create a CA, then sign the SmartFrog jars and your own redistributable JARs with issued certificates
- <jar> all redistributable components (native binaries &c) into resource files.
- You cannot (yet) use inline deployment descriptors to deploy to secured systems.

Futures

- JUnit component to run JUnit tests on remote nodes
- Output to be <junitreport> compatible
- Logging components for distributed logging
- Jetty component to host Jetty
- Axis component

Watch the CVS repository for these

"We could test everything, from everywhere."

Patrick Goldsack
hp

i n v e n t